S.Y.B.Sc.

(Computer Science)

SEMESTER - III (CBCS)

# DATABASE MANAGEMENT SYSTEMS

SUBJECT CODE : USCS304

**Programme Co-ordinator :** **Shri Mandar Bhanushe**
Head, Faculty of Science and Technology,
IDOL, University of Mumbai, Mumbai

**Course Co-ordinator :** **Mr. Sumedh Shejole**
Asst. Professor,
IDOL, University of Mumbai, Mumbai

**Editor :** **Mr. Kashafurrehman Ansari**
Assistant professor,
Karmaveer Bhaurao Patil College,
Vashi, Navi-Mumbai

**Writers** **Dr. Radha R**
Shrimathi Devkunvar Nanalal Bhatt
Vaishnav College for Women, Chennai

: **Dr. T. Parimalam**
Nandha Arts and Science College, Kerala

: **Dr. Mahesh Kishanrao Kulkarni**
Loknete Gopinathji Munde Arts, Science &
Commerce College, Ratnagiri

**May 2022, Print - I**

**Published by** : Director
Institute of Distance and Open Learning ,
University of Mumbai,
Vidyanagari, Mumbai - 400 098.

**DTP Composed and Printed by** : Mumbai University Press
Vidyanagari, Santacruz (E), Mumbai - 400098

# CONTENTS

❖❖❖❖

| **Course:** **USCS304** | **TOPICS (Credits : 02 Lectures/Week:03)** **Database Management Systems** | |
|---|---|---|
| **Objectives:** To develop understanding of concepts and techniques for data management and learn about widely used systems for implementation and usage. **Expected Learning Outcomes:** 1. Master concepts of stored procedure and triggers and its use. 2. Learn about using PL/SQL for data management 3. Understand concepts and implementations of transaction management and crash recovery | | |
| **Unit I** | **Stored Procedures:** Types and benefits of stored procedures, creating stored procedures, executing stored procedures, altering stored procedures, viewing stored procedures. **Triggers:** Concept of triggers, Implementing triggers – creating triggers, Insert, delete, and update triggers, nested triggers, viewing, deleting and modifying triggers, and enforcing data integrity through triggers. **Sequences**: creating sequences, referencing, altering and dropping a sequence. **File Organization and Indexing:** Cluster, Primary and secondary indexing, Index data structure: hash and Tree based indexing, Comparison of file organization: cost model, Heap files, sorted files, clustered files. Creating, dropping and maintaining indexes. | **15L** |
| | **Fundamentals of PL/SQL:** Defining variables and constants, PL/SQL expressions and comparisons: Logical Operators, Boolean Expressions, CASE Expressions Handling, Null Values in Comparisons and Conditional Statements, PL/SQL Datatypes: Number Types, Character Types, Boolean Type, Datetime and Interval Types. | |

| | | |
|---|---|---|
| **Unit II** | **Overview of PL/SQL Control Structures:** Conditional Control: IF and CASE Statements, IF-THEN Statement, IF-THEN-ELSE Statement, IFTHEN-ELSIF Statement, CASE Statement, Iterative Control: LOOP and EXIT Statements, WHILE-LOOP, FOR-LOOP, Sequential Control: GOTO and NULL Statements | **15L** |
| **Unit III** | **Transaction Management:** ACID Properties, Serializability, Two-phase Commit Protocol, Concurrency Control, Lock Management, Lost Update Problem, Inconsistent Read Problem , Read-Write Locks, Deadlocks Handling, Two Phase Locking protocol.<br>**DCL Statements:** Defining a transaction, Making Changes Permanent with COMMIT, Undoing Changes with ROLLBACK, Undoing Partial Changes with SAVEPOINT and ROLLBACK<br>**Crash Recovery:** ARIES algorithm. The log based recovery, recovery related structures like transaction and dirty page table, Write-ahead log protocol, check points, recovery from a system crash, Redo and Undo phases. | **15L** |

**Textbook(s):**

1) Ramakrishnam, Gehrke, Database Management Systems, Bayross, McGraw-Hill,3rd Edition
2) Abraham Silberschatz, Henry F. Korth,S. Sudarshan , Database System Concepts, 6th Edition
3) Ivan Bayross, "SQL,PL/SQL -The Programming language of Oracle", B.P.B. Publications

**Additional Reference(s):**

1) Ramez Elmasri & Shamkant B.Navathe, Fundamentals of Database Systems, Pearson Education
2) Robert Sheldon, Geoff Moes, Begning MySQL, Wrox Press.
3) Joel Murach, Murach's MySQL, Murach

# Unit I

# 1

# STORED PROCEDURE

**Unit Structure**

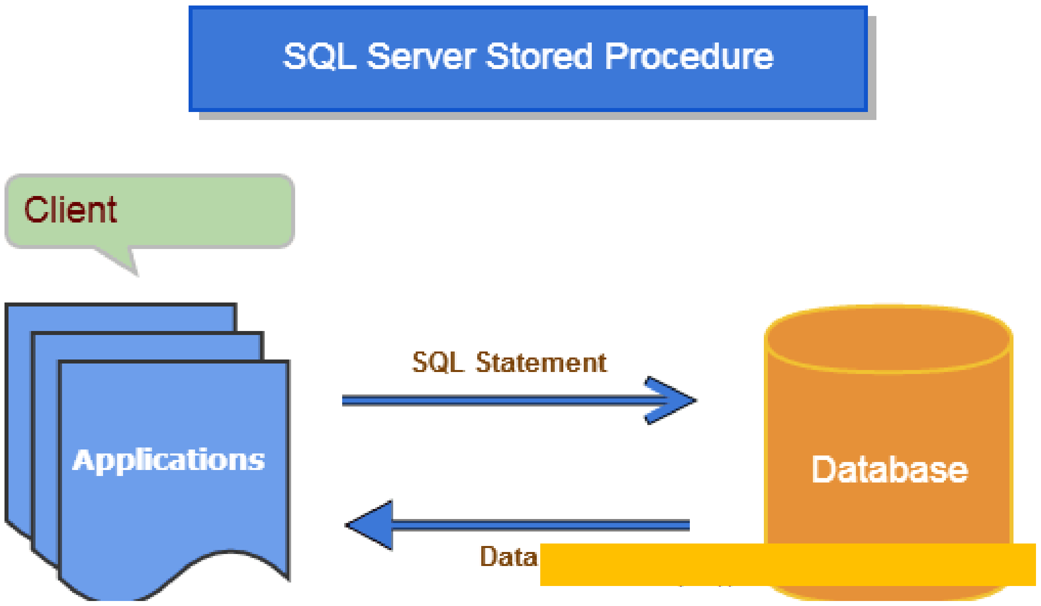
## 1.0 OBJECTIVE

- After going through this chapter the students will be able to:
- Know the different type of procedure declaration in PL/SQL
- Declare a procedure with input and output parameters
- Modify the procedure and delete a procedure
- Know how to view the content in a stored procedure

## 1.1 INTRODUCTION

**Stored Procedure :**

A procedure is a subprogram, subroutine in any language which is used to do some well defined function. It has a name, list of parameters and statements of the particular language within that. In database terminology, when a procedure is built to do some task on a database and stored in the database , it is called a stored procedure. It is a pre-compiled collection of SQL statements stored in a database server. In MySQL procedures are stored in the MySQL database server.



In terms of database a stored procedure consists of a set of Structured Query Language (SQL) statements which can be reused and shared by many programs. They can access or modify data in a database.It can accept input and output parameters. We can do the database operations like Select, Insert, Update, Delete etc in a database.

## 1.2 TYPES OF STORED PROCEDURES

1. User Defined Procedure
2. System Stored Procedure
3. Temporary Stored Procedure : The temporary procedures are also user-defined procedures and are permanent procedures. They are stored in tempdb databases. They are of two types, local and global. Local temporary procedures starts with (#)
4. Remote Stored Procedure:
5. Extended Stored Procedure:

### 1.2.1 User Defined Procedure

The procedure created by the user and stored in the database is called User Defined Procedures. It can be created in a user-defined database or System database except the Resource database. The procedure can be developed in either Transact-SQL or as a reference to a Microsoft.NET framework common runtime language (CLR) method.

Transact-SQL stored procedures handle SQL statements INSERT, UPDATE, and DELETE statements with or without parameters. The output is the row data, as a result of a SELECT statement.

CLR stored procedures are .NET objects which run in the memory of the database. Complex logic can be implemented using them as they use the .NET framework and using its classes. They include Functions, Triggers etc. It allows the coding to be done in one of .NET languages like C#, Visual Basic and F#.

### 1.2.2 System Stored procedure

These types of procedure are used to do administrative activities of a SQL Server and are prefixed with sp_. Because of this it is better not to use this prefix when naming user-defined procedures. These procedures are physically stored in the internal Resource database. They logically appear in the sys schema of user defined and system defined databases.

### 1.2.3 Temporary Stored Procedure

The temporary procedures are also user-defined procedures and are permanent procedures. They are stored in tempdb databases. They are of two types, local and global. Local temporary procedures starts with (#)

### 1.2.4 Remote Stored Procedures

These procedures are created and stored on remote servers. With the proper permission the users can access them from various servers. The criteria is that the remote server has to be configured and proper login mapping must be done.

### 1.2.5 Extended Stored Procedure

The extended procedures help in creating external routines in other programming languages and can be loaded and run dynamically in an SQL Server. The extended procedure starts with xp_ prefix in the Master database. They are useful in building an interface to external programs.

## 1.3 BENEFITS OF STORED PROCEDURE

- Modular Programming: The main purpose of procedures, subroutines, functions is to create modules and make use of them again and again whenever needed. The aim is to reuse the code wherever needed instead of writing them again.

- Network traffic reduced : Many individual SQL statements meant for a specific task can be put together as a Stored procedure and can be executed with a single statement, i.e by calling the name of the procedure along with parameters. They are executed on the server-side and perform a set of actions and return the results to the client-side. If this encapsulation of procedure is not given, then every individual line of code has to travel the network between client and server, which greatly slows down the traffic.

- Faster execution: As the stored procedures, query plans are kept in memory after the initial execution, when it is to be executed again, no need for reparsing or re- optimizing on subsequent executions. This increases the performance of the application.

- Enforced consistency: Since the users modify the data only through stored procedures, problems occurring due to modifications are eliminated.

- Reduced operator and programmer errors: Since on calling the procedures, limited information is passed like name of procedures, input parameters the likelihood of errors in SQL is greatly eliminated.

- Automated complex or sensitive transactions: The integrity on tables can be assured if all the modifications on them are done through these stored procedures.

- Stronger security: Multiple users and client programs can perform operations on underlying database objects through a procedure, even if the users and programs do not have direct permissions on those underlying objects. The procedure controls what processes and activities are performed and protects the underlying database objects. This eliminates the requirement to grant permissions at the individual object level and simplifies the security layers.

- When a procedure is accessed over the network, only the call to execute the procedure is visible. This prevents malicious users from accessing databases, tables etc as nothing is visible .
- Using procedure parameters helps guard against SQL injection attacks. Since parameter input is treated as a literal value and not as executable code, it is more difficult for an attacker to insert a command into the Transact-SQL statement(s) inside the procedure and compromise security.

## 1.4 CREATING A STORED PROCEDURE

The procedure contains a header and a body.

- **Header:** The header contains the name of the procedure and the parameters or variables passed to the procedure.
- **Body:** The body contains a declaration section, execution section and exception section

### 1.4.1 Passing parameters in procedure

When you want to create a procedure or function, you have to define parameters .There are three ways to pass parameters in procedure:

1. **IN parameters only: Using** IN parameters the inputs are passed to the procedure. By default the parameters are of IN type. Variables, expressions can be passed as IN parameters. The value of the parameter cannot be overwritten by the procedure or the function.
2. **OUT parameters:** The OUT parameter used for getting results from a procedure. From the calling statement, these must be always a variable to hold the value returned by the procedure. The value of the parameter can be overwritten by the procedure or function.
3. **INOUT parameters:** The INOUT parameter can be used for giving both input and getting output from procedure. The value of the parameter can be overwritten by the procedure or function. From the calling statement, these must be always a variable to hold the value returned by the procedure

Note: A procedure may or may not return any value. Once created they will be stored as database objects.

**Syntax:**

```
CREATE OR REPLACE PROCEDURE <procedure_name>
(<parameter1 IN/OUT <datatype>
<parameter2 IN/OUT <datatype>
…
)
[IS | AS]
        <declaration_part>
```

```
BEGIN
        <execution part>
EXCEPTION
        <exception handling part>
END;
```

**Syntax explanation:**

- CREATE PROCEDURE instructs the compiler to create a new procedure. Keyword 'OR REPLACE' instructs the compiler 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.

**Example :**

before executing any program in PL/SQL type the following in the SQL> prompt to see the output.

*SQL>set serveroutput on;*

The above command will enable the dbms_output.put_line().

**1.4.1.1 Procedure without parameter**

**Example :** In the below example a procedure with name welcome is created. There is no parameter passed. First the created procedure has to be created. so open a notepad or any editor and type the following code with extension as .sql

the name of the below program is ***greetings_proc.sql***

```
create or replace procedure welcome
as
begin
dbms_output.put_line('welcome to Mumbai University');
dbms_output.put_line('This is Database Management System course');
end;
/
```

**Code Explanation**

**Line 1-3**: Creating Procedure '*welcome*'

**Line 4-6** : Printing the information on the screen

First the above procedure has to be created. in order to do that run the following command:

```
SQL> @ c:/sql_prgs/greetings_proc.sql;
```

*output:*

*Procedure created.*

All the sql programs are stored in the **c:/sql_prgs** folder. so when the above program is executed and when there is no error the sql command line will return as Procedure created.

Now the created procedure has to be called with EXEC command as below:

output:

```
SQL> execute welcome();

welcome to Mumbai University

This is Database Management System course

PL/SQL procedure successfully completed.
```

**Example**

In this example, we are going to use a select statement to list a record in a table called 'employee_csc'. So the first step is to create a table.

**Table creation:**

```
1.    Create table employee_csc(ename varchar2(30),street varchar2(40),
city varchar2(30), eid number(3), primary key(eid));
```

The next step is to insert data into the above table :

To do that the following command can be used and run 'n' number of times to add data dynamically.

```
SQL> insert into employee_csc values('&ename','&street','&city',&eid);
```

After few insertion the table looks like this:

```
SQL> select * from employee_csc;

ENAME      STREET          CITY          EID
---------- --------------- --------------- ----------
```

```
anitha     1st street      chennai             100
aiswarya   2nd street      chennai             101
chandra    2nd street      chennai             102
hema       3rd street      chennai             103
lalitha    metha street    mumbai              104
raman      krishnan street bangalore           105
harini     kalam street    andhra              106
danush     ragav street    bangalore           107
david      kamaraj street  calcutta            108
ananthi    rajaji          chennai             109
sundar     2nd cross st    hydreabad           110
raveena    3rd cross st    erode               111
12 rows selected.
```

The following procedure will display the employee name and employee id of a particular employee.

```
1   create or replace procedure SelectEmp
2   as
3   o_ename varchar2(30);
4   o_eid number(10);
5   begin
6   select  ename,eid into o_ename,o_eid from employee_csc where
    eid=100;
7
8   dbms_output.put_line('employee name = ' ||o_ename);
9   dbms_output.put_line('employee id = ' ||o_eid);
    end;
    /
```

**Code Explanation**

**Line 1-4**: Creating Procedure '*SelectEmp*' , with the local variables.

**Line 5-9** : A particular employee is queried from database and printed the information on the screen now creating the procedures and then executing will produce the result as follows:

*SQL> @c:/sql_prgs/SelectEmp.sql;*

Procedure created.

*SQL> exec SelectEmp;*

*employee name = anitha*

*employee id = 100*

PL/SQL procedure successfully completed.

## 1.5 EXECUTING A STORED PROCEDURE

The stored procedure can be executed by using EXECUTE or EXEC statement followed by the name of the stored procedure along with a parameter list if any. This has been already done in the previous example

The above procedure can be executed as

SQL>*execute welcome();*

*SQL> exec SelectEmp;*

*Stored Procedure with one parameter*

```
create or replace procedure SelectUser
(id in number)
is
o_ename varchar2(30);
o_eid number(10);
begin
select ename,eid into o_ename,o_eid from employee_csc where eid=id;
dbms_output.put_line('employee name = ' ||o_ename);
dbms_output.put_line('employee id = ' ||o_eid);
end;
/
```

**Code Explanation**

**Line 1-5**: Creating Procedure '*Select User*' , with one input parameter and two local variables.

**Line 6-10** : A particular employee whose value is passed as input parameter is queried from database and printed the information on the screen

### *1.5.1 Executing the stored procedure with one parameter*

The stored procedure can be executed by using EXECUTE or EXEC statement followed by the name of the stored procedure along with a parameter list if any.

The above procedure must be complied and then executed as

```
SQL> @c:/sql_prgs/selectuser.sql;

Procedure created.

SQL> @c:/sql_prgs/selectuser.sql;

Procedure created.

SQL>exec SelectUser(110);

employee name = sundar

employee id = 110

PL/SQL procedure successfully completed.
```

### 1.5.2 Creating and Executing Procedure with multiple input parameters

Example : The following procedure is used to insert a record in the table *employee_csc*.

```
create or replace procedure insertemployee
(iname in varchar2,istreet in varchar2 ,icity in varchar2,ieid in number)
is
begin
insert into employee_csc values (iname,istreet,icity,ieid);
end;
/
```

**Code Explanation**

**Line 1-3**: Creating Procedure '*insertemployee*' , with four input parameters .

**Line 4-6** : The input parameters are inserted into the table using insert command.

compiling and executing the above procedure as follows:

```
SQL> @c:/sql_prgs/insertemployee.sql;

Procedure created.

SQL> exec insertemployee('radha','3rd street','erode',112);

PL/SQL procedure successfully completed.
```

Now whether the data has been inserted or not can be checked with select statement as follows:

```
SQL> select * from employee_csc;

ENAME      STREETT         CITY              EID
---------- -------------- -------------- ----------
anitha     1st street      chennai           100
aiswarya   2nd street      chennai           101
chandra    2nd street      chennai           102
hema       3rd street      chennai           103
lalitha    metha street    mumbai            104
raman      krishnan street bangalore         105
harini     kalam street    andhra            106
danush     ragav street    bangalore         107
david      kamaraj street  calcutta          108
ananthi    rajaji          chennai           109
sundar     2nd cross st    hydreabad         110
raveena    3rd cross st    erode             111
radha      3rd street      erode             112
```

***13 rows selected.***

*Another way to execute is to call it within the PL/SQL block like below.*

*PL/SQL program to call procedure*

Let's see the code to call the above created procedure.

```
1   begin
2   insertemployee('ramani','1st street','bangalore',113);
3   dbms_output.put_line('record inserted successfully');
4   end;
    /
```

**Code Explanation**

**Line 1-4**: PL/SQL block is created.

**Line 2** : The procedure 'insertemployee' is called here with input parameters for the table record

*After executing the above as follows:*

*SQL> @c:/sql_prgs/insertemployee_call.sql;*

*record inserted successfully*

*PL/SQL procedure successfully completed.*

**1.5.3 Creating a stored procedure with default parameters values**

A stored procedure can be created with a default parameter. when the procedure is called without parameters it will take the default value declared in the procedure else it will take the value passed by the user at the time of execution.

```
create or replace procedure SelectUser
(id in number :=105)
is
o_ename varchar2(30);
o_eid number(10);
begin
select ename,eid into o_ename,o_eid from employee_csc where
eid=id;
dbms_output.put_line('employee name = ' ||o_ename);
dbms_output.put_line('employee id = ' ||o_eid);
end;
/
```

**Code Explanation**

**Line 1**: PL/SQL block is created.

**Line 2** : The procedure 'SelectUser' is created with an input parameter

**Line 4-5**: local parameters are declared

**Line 6-10** : with the input parameter as the criteria the row is extracted from table *employee_csc* and placed in local parameter. The data in the local parameter is displayed.

In the above procedure the default value can be given in the passing parameter as above by the assignment statement :=

So when executing this if the parameter is not given, the procedure take the value 105, that is the default value which we assign. If the value is passed as parameter than it will ignore the default value and take the passed value for further processing. Both the outputs are given as follows:

```
SQL> exec SelectUser();

employee name = raman

employee id = 105

PL/SQL procedure successfully completed.

SQL> exec SelectUser(107);

employee name = danush

employee id = 107

PL/SQL procedure successfully completed.
```

### 1.5.4 Creating a stored procedure with an output parameter

**Example 3** Create a procedure to calculate simple interest. Principal, rate of interest and no. of years are given as input.

```
1    --this program calculates simple interest
2    declare
3    n_principle number(6);
4    n_years number(4);
5    n_interest number(6,2);
6    n_ans number(8,2);
7    --procedure starts
8     procedure simpleinterest(p in number,n in number, r in
     number, si out number)
9    is
10   begin
11   si:=(p*n*r)/100;
12   end;
13   --main starts
14   begin
15   n_principle:=&p;
16   n_years:=&n;
17   n_interest:=&r;
18   simpleinterest(n_principle,n_years,n_interest,n_ans);
19   dbms_output.put_line('simple interest is ' || n_ans);
20   end;
     /
```

**Code Explanation**

Line 2 : anonymous PL/SQL is declared.

Line 3-6 : local variables are declared

Line 8 : a procedure to calculate simple interest is created with 3 input parameter and one output parameter

Line 11 : the calculation of simple interest is done and is stored in output parameter.

Line 18 : the procedure is called with parameters.

Line 19 : the value returned from the procedure is printed.

**Output**

```
SQL> @d:/plsql/proc_ex1.sql
Enter value for p: 4000
old  15: n_principle:=&p;
new  15: n_principle:=4000;
Enter value for n: 4
old  16: n_years:=&n;
new  16: n_years:=4;
Enter value for r: 5.0
old  17: n_interest:=&r;
new  17: n_interest:=5.0;
simple interest is 800
PL/SQL procedure successfully completed.
```

Example

consider the following table

```
SQL> desc employee_csc;
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 ENAME                                              VARCHAR2(30)
 STREET                                             VARCHAR2(40)
 CITY                                               VARCHAR2(30)
 EID                                       NOT NULL NUMBER(3)
 EMAIL                                              VARCHAR2(100)
```

The following procedure will return the employee name of a particular id using OUT parameter.

```
1   create or replace procedure employee_name_detail(id in number, e_name out varchar2)
2   is
3   begin
4   select ename into e_name from employee_csc where eid=id;
5   end;
    /
```

**Code Explanation**

**Line 1** : Creation of procedure “employee_name_detail” with one input and one output parameters

**Line 3-5** : select statement is used to extract a record with a particular employee id.

The procedure can be called from a PL/SQL block as follows:

```
1   declare
2   e_name varchar2(30);
3   begin
4   employee_name_detail(100,e_name);
5   dbms_output.put_line(e_name);
6   end;
    /
```

**Code Explanation**

**Line 1** : Creation of PL/SQL block to call the procedure “employee_name_detail” with one input and one output parameters

**Line 2** : declaration of local parameter

Line 3-6 : call the procedure “employee_name_detail” and the value returned is printed.

*executing the above procedure and its call from PL/SQL can be done as follows*

```
SQL> @ c:\sql_prgs\employee_name_detail.sql;

Procedure created.

SQL> set serveroutput on;

SQL> @ c:\sql_prgs\employee_name_detail_call.sql;

anitha

PL/SQL procedure successfully completed.
```

*The following procedure will return the number of records in a table*

```
1 create or replace procedure find_rows(cnt out number)
2 as
3 begin
4 select count(*) into cnt from employee_csc;
5 end;
  /
```

**Code Explanation**

**Line 1** : Creation of procedure "find_rows" with one output parameter.

**Line 3-5** : select statement is used to extract the number of records in the *employee_csc* table.

The above procedure will be called from a PL/SQL block as follows

```
set serveroutput on
declare
r_count number;
begin
find_rows(r_count);
dbms_output.put_line('number of records in table ='||r_count);
end;
/
```

**Code Explanation**

**Line 2** : declaration of PL/SQL block.

**Line 3 : declaration of local parameter**

**Line 4-7** : calling the procedure "find_rows" and the value returned is printed.

on executing both the PL/SQL blocks

```
SQL> @ c:\sql_prgs\find_rows.sql;

Procedure created.

SQL> @ c:\sql_prgs\find_rows_call.sql;

number of records in table =14

PL/SQL procedure successfully completed.
```

***once the procedure is compiled with the following command***

```
SQL> @ c:\sql_prgs\find_rows.sql;

Procedure created.
```

the procedure can be called from command line as follows:

```
SQL> var lcnt number;

SQL> exec find_rows(:lcnt);

PL/SQL procedure successfully completed.

SQL> print lcnt;

      LCNT
----------
        14
```

## 1.6 ALTERING OR MODIFYING A STORED PROCEDURE

A stored procedure can be recompiled explicitly using ALTER PROCEDURE statement. Due to this implicit run-time recompilation is eliminated which in turn prevents run-time compilation errors and performance overhead.

Syntax

```
ALTER PROCEDURE <procedure_name> COMPILE;
```

Note: This does not change the declaration of an existing procedure. so if anything has to be modified inside procedure use CREATE OR REPLACE PROCEDURE command instead of ALTER to achieve the same function

```
Alter  procedure SelectUser

(id in number :=105)

is

o_ename varchar2(30);

o_eid number(10);

o_city varchar2(30);

begin

select ename,city,eid into o_ename,o_city,o_eid from employee_csc
where eid=id;

dbms_output.put_line('employee name = ' ||o_ename);

dbms_output.put_line('city name = ' ||o_city);

dbms_output.put_line('employee id = ' ||o_eid);

end;

/
```

## 1.7 DELETING A STORED PROCEDURE

A created procedure can be deleted by using DROP PROCEDURE or DROP PROC statement.

**Syntax for drop procedure**

1. DROP PROCEDURE procedure_name;

*Example of drop procedure*

1. DROP PROCEDURE INSERTUSER;

## 1.8 VIEWING STORED PROCEDURES

The source code in a stored procedures can be viewed by using the following command:

syntax:

```
select text
from user_source
where name='STORED-PROC-NAME'
and type='PROCEDURE'
order by line;
```

note : the procedure name must be given in CAPITAL letters, because SQL stores the procedure name in ALL capital letters.

Example

```
SQL> select text from user_source where name='SELECTEMP' and
type='PROCEDURE' order by line;
```

output

```
TEXT
----------------------------------------------------------------------------
procedure SelectEmp
as
o_ename varchar2(30);
o_eid number(10);
begin
select  ename,eid into o_ename,o_eid from employee_csc where eid=100;
dbms_output.put_line('employee name = ' ||o_ename);
dbms_output.put_line('employee id = ' ||o_eid);
end;

9 rows selected.
```

## 1.9 SUMMARIZATION

- A stored procedure is a PL/SQL program stored inside a database schema.
- A procedure consists of declarative, executive and exception sections.
- A procedure can be called with its name along with a list of parameters enclosed within parentheses.
- There are three types of parameters: IN , OUT, IN OUT where IN type is used to pass input to the procedure, OUT type is used to return a value from a procedure and IN OUT type is used to pass input and return a value from a procedure.
- The formal parameters of a procedure must match the actual parameters of a calling procedure.

## 1.10 REFERENCES

https://www.sqlshack.com/sql-server-stored-procedures-for-beginners/

https://www.javatpoint.com/stored-procedure-in-sql-server

http://www.mathcs.emory.edu/~cheung/Courses/377/Syllabus/6-PLSQL/storedproc.html

https://plsql-tutorial.com/plsql-passing-parameters-procedure-function.htm

Questions:

1. Write a short note on Stored Procedures.
2. Write a short note on the procedure to create a stored procedure.
3. State and explain various types of stored procedures.
4. Write a short note on passing parameters in stored procedures.

❖❖❖❖

2

# TRIGGERS AND SEQUENCES

**Unit Structure**

## 2.0 OBJECTIVE

This chapter would make the students to understand the following concepts:

- Get to know the necessity of Triggers and Sequences
- Know to create Triggers and sequences
- Know to delete a Trigger

- Understand and to use nested Triggers
- Know to refer a sequence
- Know to alter a created sequence
- Know to delete a sequence

## 2.1 INTRODUCTION

**Triggers**

Triggers are stored programs, which are automatically executed or fired when some event occurs. Triggers could be defined on the table, view, schema, or database with which the event is associated.

Before we deep dive into the understanding of what triggers are and how they play a very important role in any well-designed database application. Let's try to start with an example that we can relate to.

**Scenario 1:**

One of the core features of a social media network today is the notification system. They help the user stay connected and get to know about any updates that have happened in their social circle. Now, let's take two users Ram and Raj. Ram creates a post about his vacation on the social media network. His friend Raj is commenting on the same post. How will we intimate Ram about this change?

We can handle it either in the application layer, which would be our server application written in java / node.js / or any backend technology. But handling in the application layer would add up more logic that might take more time to execute as there would be multiple round trips between database and server application. Or lets get into a simpler way where we add a trigger procedure that would update Ram's notification count which would reference this new comment insert operation.

We will try to design this system by the end of this chapter.

## 2.2 OVERVIEW

Database triggers are stored procedures / programs in a RDBM system which gets automatically executed when an event occurs. Triggers play an integral part in a well-designed database application. Triggers can be used to

1) Validate data changes made to a table
2) Maintain integrity by automating maintenance tasks
3) Create rules that govern administration of the database.

There are five broad types of events to which trigger procedures can be attached to:

1) Data manipulation language (DML) statements (DELETE, INSERT, or UPDATE)
2) Data Definition Language (DDL) statements (CREATE, ALTER, DROP)
3) Database events (SERVERERROR, LOGON, LOGOFF, STARTUP, SHUTDOWN)
4) INSTEAD OF
5) Suspended Statements.

## 2.3 TRIGGER CLASSIFICATION

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 in 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.*

## 2.4 IMPLEMENTING TRIGGERS

### *2.4.1 Creating a Trigger*

***Syntax of a trigger statement / procedure***

1 **CREATE [OR REPLACE] TRIGGER***TRIGGER_NAME*

2 **{BEFORE | AFTER}***TRIGGERING_EVENT***ON***TABLE_NAME*

3 [**FOR EACH ROW]**

4 **[FOLLOW | PRECEDES***ANOTHER_TRIGGER***]**

5 **[ENABLE / DISABLE]**

6 **[WHEN***CONDITION***]**

7 **DECLARE**

*DECLARATION STATEMENTS*

**BEGIN**

*EXECUTABLE STATEMENTS*

**EXCEPTION**

*EXCEPTION_HANDLING STATEMENTS*

**END;**

Trigger statements can be broken down into two parts, header and body. Header part is all about telling the RDBMS on how and when to run a trigger. Consider this as a metadata that helps the database to execute the procedure defined in the body when necessary conditions are met.

Lets deep dive into Header statements and try to understand what it means.

Line 1: **CREATE** keyword instructs DBMS to create a trigger with the specified trigger_name. **The TRIGGER** keyword always follows the **CREATE** keyword. Sometimes we would like to update a trigger which already exists or change its properties, then we can use the optional keyword **OR REPLACE** next to **CREATE**.

Line 2: A trigger needs to run only on occurrence of an event, we specify this event as triggering_event. We would also need to specify on which table this trigger has to be attached. A trigger can execute either before or after an event. It's important to understand the business use case to specify the timing. For example if we want a trigger to execute before an insert event. Then mostly we would try to do a sanity or validation of the data. Where as notifications for example we would like to do it after an insert operation

Line 3: FOR EACH ROW, specifies if a trigger is going to be a row level trigger or statement level trigger. Let's assume we have 10 statements, but it just affects one row of data, then this trigger executes only once based on if a row is inserted, updated or deleted. If this statement is omitted, the database defaults to for each statement and it will execute on a number of statements.

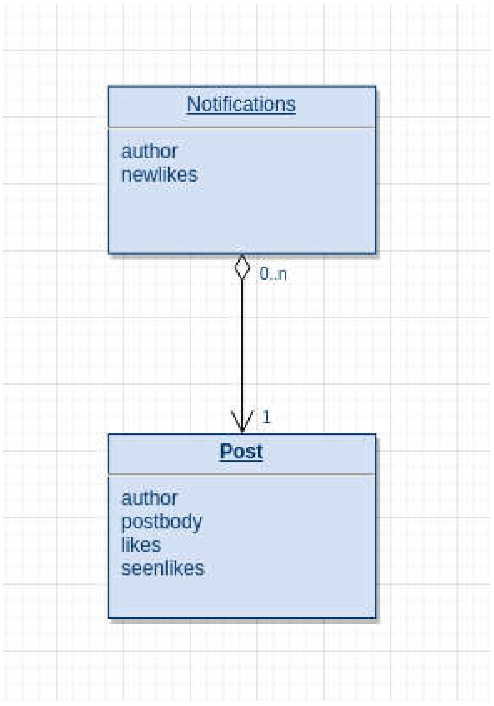
Line 4: FOLLOWS / PRECEDES.

For each trigger event Insert, update or delete we can specify multiple trigger procedures. There could be instances where we would want to specify the order of execution. Follows / Precedes helps to specify the order of execution of a trigger.

Line 5: ENABLE/DISABLE, this statement specifies if the created trigger is set to enabled status on creation. If a trigger is enabled, it would start executing from the time of creation. On disabled state an explicit enabling is required before execution

**Creating a trigger:**

Before we create our trigger we might need to create two tables for this specific example like in the diagram below.



There could be multiple notifications to a single post. So let's get started with creating the Post table.

```
CREATE TABLE POST (
   author varchar2(255),
   postbody varchar2(255),
   likes number,

   seenlikes number);
```

Now let’s create a Notifications table. This would be the table that would be modified with DML operation to insert a notification.

```
CREATE TABLE NOTIFICATIONS (

   author varchar2(255),

   newlikes number

);
```

Now, we have our two tables. There is nothing stopping us to create a trigger, for this scope of the book we will only concentrate on DML instructions.

Let's create our trigger in three steps.

Step 1:  Define the header.

We would want to create a trigger of name notification_new_likes. There could be scenario on a shared database where the same trigger name might exists. So, we will make sure to add OR REPLACE keywords.

```
CREATE OR REPLACE TRIGGER notifications_new_likes
```

Step 2: Decide on event and when to run

Since this is a notification system it would be apt only when we execute this after the operation to a table / post has occurred. If we operate it earlier and then the insert operation or update operation fails, we would have to add the extra overload of doing a rollback of this trigger. Which could add more complications to the system. We would like to create a notification to each row changed and not the number statements that runs.

```
before insert or update on POST

for each row
```

Step 3: Code the logic

Since this is our initial trigger, let’s try to make it really simple by printing hello world. As we proceed, we can iterate over this.

```
begin

   dbms_output.put_line('Hello World trigger');

end;
```

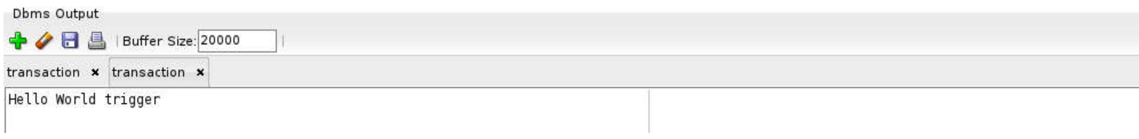
### *2.4.2 Insert and Update using a Trigger*

***Example 1:***

Full source code.

```
CREATE OR REPLACE TRIGGER notifications_new_likes

before insert or update on POST

for each row

begin

   dbms_output.put_line('Hello World trigger');

end;
```

Try inserting or updating any data in the POST table. We should see output something like this



```
Hello World trigger
```

Congratulations, you have created your very first trigger.

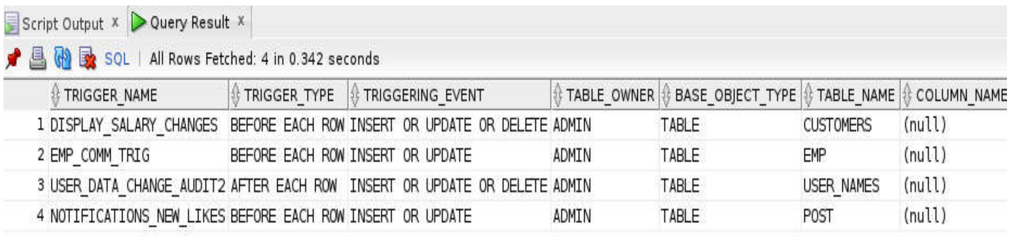
Obviously this trigger is not going to do anything amazing. So, let's try to create a new trigger which will do some DML operations on another table.

Before that let's try to view our trigger, there are two ways to see a trigger.

On command prompt we can give

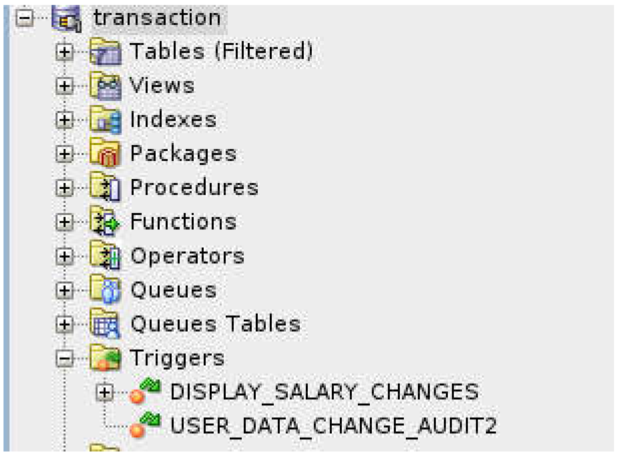
```
SELECT * FROM USER_TRIGGERS;
```

This should list all our triggers, something like this



| | TRIGGER_NAME | TRIGGER_TYPE | TRIGGERING_EVENT | TABLE_OWNER | BASE_OBJECT_TYPE | TABLE_NAME | COLUMN_NAME |
|---|---|---|---|---|---|---|---|
| 1 | DISPLAY_SALARY_CHANGES | BEFORE EACH ROW | INSERT OR UPDATE OR DELETE | ADMIN | TABLE | CUSTOMERS | (null) |
| 2 | EMP_COMM_TRIG | BEFORE EACH ROW | INSERT OR UPDATE | ADMIN | TABLE | EMP | (null) |
| 3 | USER_DATA_CHANGE_AUDIT2 | AFTER EACH ROW | INSERT OR UPDATE OR DELETE | ADMIN | TABLE | USER_NAMES | (null) |
| 4 | NOTIFICATIONS_NEW_LIKES | BEFORE EACH ROW | INSERT OR UPDATE | ADMIN | TABLE | POST | (null) |

Another way would be to use a nice GUI like SQLDeveloper from Oracle to list all the triggers by clicking on Triggers from the left pane.



Let's get our notifications populated when an update operation occurs. Following trigger will run when there is an update operation on Post rows.

```
CREATE TRIGGER notifications_new_likes

after update on POST

for each row

begin

   if (UPDATING and (:NEW.likes - :NEW.seenlikes > 0)) then

      INSERT INTO NOTIFICATIONS values (:NEW.author, :NEW.likes
- :NEW.seenlikes);

   end if;

end;
```

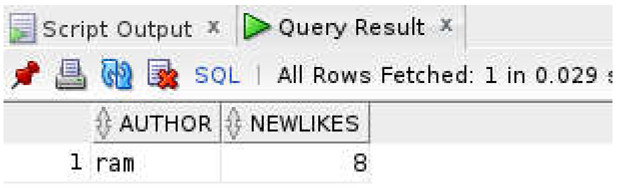
:NEW variable will have the new modified data of the table row. When we get a new update to a row in Post. Then this trigger gets executed and only when the row is in UPDATING state we do a logic to create a new data in the notifications table.

Now, we will try to update a value in post.

```
insert into post (author, postbody, likes, seenlikes) values ('ram', 'sample
blog', 0, 2);
```

```
update POST set likes=10 where author = 'ram';
```

Let's try to go to the notifications table and check what would be the new likes count for author ram.



|   | AUTHOR | NEWLIKES |
|---|---|---|
| 1 | ram | 8 |

That’s it our trigger has modified the Notifications table to get the un-read likes count. Of course the notification system isn’t this simple but has more complicated use cases. What we have tried to achieve here is an example pathway to kickstart your imagination on possible use cases of triggers.

Let's try to update our trigger with an advanced use case of deleting the notification record if the user has caught up with all the notifications.

```
CREATE OR REPLACE TRIGGER notifications_new_likes

after insert or update on POST

for each row

begin

   if (UPDATING and (:NEW.likes - :NEW.seenlikes > 0)) then

 INSERT INTO NOTIFICATIONS values (:NEW.author, :NEW.likes -
:NEW.seenlikes);

   end if;

   if (UPDATING and (:NEW.likes - :NEW.seenlikes = 0)) then

  DELETE FROM NOTIFICATIONS WHERE AUTHOR =
:NEW.author;

   end if;

end;
```

***Example 2:***

Let us consider a trigger that checks the value of salary before inserting or updating the works_csc table and ensures that salary below 20,000 is not inserted. It acts before insertion or update. Let us consider the table *works_csc* which has the following table structure and data

```
SQL> desc works_csc;
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 EID                                       NOT NULL NUMBER(5)
 CID                                                VARCHAR2(3)
 SALARY                                             NUMBER(9)
```

```
SQL> select * from works_csc;
       EID CID       SALARY
---------- ---- ----------
       101 c2        35000
       102 c3        35000
       103 c4        50000
       104 c2        30000
       105 c3        30000
       106 c1        40000
       108 c3        30000
       109 c3        28000
8 rows selected.
```

```
1    create or replace trigger min_sal_chk
2    before  insert or update on works_csc
3    for each row
4    when (new.salary<20000)
5    begin
6    raise_application_error(-20000, 'sal must be more than 20000');
7    end;
     /
```

**Code Explanation:**

**Line 1-3 :** Creation of trigger '*min_sal_chk*' which will be triggered before insertion or updation of each row in *works_csc* table

**Line 4:** the condition of when the trigger is triggered is given.

**Line 5-7:** This will raise error , if the salary on insertion or updation is below 20000

Execution of triggers during insertion

```
SQL> insert into works_csc values(112,'c1',15000);
insert into works_csc values(112,'c1',15000)
        *
ERROR at line 1:
ORA-20000: sal must be more than 20000
ORA-06512: at "SYSTEM.MIN_SAL_CHK", line 2
ORA-04088: error during execution of trigger 'SYSTEM.MIN_SAL_CHK'
```

Execution of triggers during updation

```
SQL> update works_csc set salary=15000 where eid=103;
update works_csc set salary=15000 where eid=103
        *
ERROR at line 1:
ORA-20000: sal must be more than 20000
ORA-06512: at "SYSTEM.MIN_SAL_CHK", line 2
ORA-04088: error during execution of trigger 'SYSTEM.MIN_SAL_CHK'
```

***Example 4:*** The following trigger executes BEFORE to convert the empname field from lowercase to uppercase.

```
create or replace trigger emp_trig
before
insert on employee
for each row
begin
:new.empname:=upper(:new.empname);
end;
/
```

**Code Explanation:**

**Line 1-4 :** Creation of trigger '*emp_trig*' which will be triggered when an insertion into table *employee* takes place at row level

**Line 5-7:** This will convert the existing employee name into uppercase letters.

Execution of insert command :

```
SQL> insert into employee values(113,'rajan','eldams road','chennai');

1 row created.
```

The record with eid has empname entered as uppercase.

```
SQL> select * from employee;


     EID EMPNAME                      STREET          CITY
---------- ---------------------------- -------------- ---------------
     100 anitha                      1st street           calcutta
     101 aiswarya                     2nd street   chennai
     102 chandra                      2nd street  chennai
     103 hema                         3rd street   chennai
     104 lalitha                     metha street mumbai
     105 raman                       krishnan street      bangalore
     106 harini                      kalam street andhra
     107 danush                      ragav street  bangalore
     108 david                       kamaraj street     calcutta
     109 ananthi                     rajaji street        chennai
     113 RAJAN                      eldams road  chennai
      112 krish                       3rd street   bangalore
12 rows selected.
```

***Example 5:*** We write a trigger to fire before the insert takes place.

```
SQL> create table person(id int,name varchar2(30),dob date,primary
key(id));

Table created.
```

On execution of an insert command the trigger will be triggered:

```
SQL> insert into person values(10,'anitha','28-sep-1996');

before insert of anitha

1 row created.
```

***Example 6***: In the following example, a database should not allow one to modify one’s dob. In this case the following trigger helps to achieve this:

```
1   create or replace trigger person_update_trig
2   before
3   update of dob on person
4   for each row
5   begin
6   raise_application_error(-20000,'cannot change date of birth ');
7   end;
    /
```

**Code Explanation**

**Line 1-4** : Creation of trigger ‘*person_update_trig*’ which will be triggered before the update of date of birth field (*dob*) of a person in the table ‘*person*’ at row level.

**Line 5-7 :** Error will be raised when the user tries to change dob of a *person* in the table

When the update of the dob field takes place the above trigger is triggered.

```
SQL> update person set dob='3-aug-1996';

update person set dob='3-aug-1996'

       *

ERROR at line 1:

ORA-20000: cannot change date of birth

ORA-06512: at "SYSTEM.PERSON_UPDATE_TRIG", line 2

ORA-04088: error during execution of trigger
'SYSTEM.PERSON_UPDATE_TRIG'
```

**: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 new value of 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

These clauses 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. |
| :OLD | INVALID. | VALID | VALID |

**Example 8**

The price of a product changes constantly. It is important to maintain the history of the prices of the products. Create a trigger to update the "Product_price_history" table when the price of the product is updated in the "Product" table. Create the "Product" table and "Product_ price_ history" table with the following fields respectively a.Product_price_history (product_id number(5), product_namevarchar2(32), supplier_name varchar2(32), unit_price number(7,2) )

b. Product (product_id number(5), product_name varchar2(32), supplier_name varchar2(32), unit_price number(7,2) )

1. Create the Price_history_trigger and execute it.

2. Update the price of a product. Once the update query is executed, the trigger fires and should update the 'Product_price_history' table.

*SQL> create table product(product_id number(5),product_name varchar2(32), supplier_name varchar2(32),unit_price number(7,2));*

*Table created.*

*SQL> create table product_price_history(product_id number(5), product_name varchar2(32),supplier_name varchar2(32),unit_price number(7,2));*

*Table created.*

Trigger :price_history_trig.sql

```
1   create or replace
2   trigger price_history_trig
3   before update of unit_price on product
4   for each row
5   begin
6      insert into product_price_history
7      values
8
9   (:old.product_id,:old.product_name,:old.supplier_name,:old.unit_price
    );
    end;
    /
```

**Code Explanation:**

**Line 1-4 :** Creation of trigger ‘price_history_trig’ which will trigger before updation on field *unit_price* of *product* table takes place at **ROW-**level.

**Line 5-9 :** Whenever there is change in the *unit_price* of the *product* table those values will be backed up in the *product_price_history* table.

```
SQL> @e:/plsql/price_history_trig.sql

 Trigger created.
```

The product table consists of the following values

```
SQL> select * from product;

 PRODUCT_ID PRODUCT_NAME       SUPPLIER_NAME
        UNIT_PRICE

---------- -------------- ------------------- ----------

        100   files          bismi                       10

        101    pen           karthik printers          15

        102    pencil        nataraj                   20
```

now when we try to update the trigger will be executed:

```
SQL> update product set unit_price=12 where product_id=100;

1 row updated.

SQL> select * from product;

PRODUCT_ID PRODUCT_NAME        SUPPLIER_NAME
        UNIT_PRICE

---------- -------------- ------------------- ----------

        100   files          bismi                     12

        101   pen            karthik printers          15

        102   pencil         nataraj                   20
```

in the product_price_history table the old value is saved by the trigger automatically;

```
SQL> select * from product_price_history;

PRODUCT_ID PRODUCT_NAME        SUPPLIER_NAME
        UNIT_PRICE

---------- -------------- ------------------- ----------

        100   files          bismi                     10
```

### 2.4.3 Deleting through a Trigger

Example 3: This example demonstrates use of triggers to keep information on deleted records.

First create a table to hold deleted records as backup by the following command.

```
SQL> create table works_bkup (eid number(3),cid varchar2(4),salary number(7,2),deldate date);

Table created.
```

Now the trigger is created. Whenever a deletion takes place the deleted record is entered into this back up table along with the time of deletion.

```
1  create or replace trigger bkup_rec
2  after delete on works_csc
3  for each row
4  begin
5  insert into works_bkup values(:old.eid,:old.cid,:old.salary,sysdate);
6  end;
   /
```

**Code Explanation:**

**Line 1-3 :** Creation of trigger '*bkup_rec*' will be triggered whenever a deletion in *works_csc* table takes place at **ROW** level

**Line 4- 6:** insertion into backup table '*works_bkup*' is done here.

Execution of trigger.

```
SQL> @e:/books/sql_prgs/works_trig2.sql;

Trigger created.

SQL> delete from works_csc where eid=100;

1 row deleted.

SQL> select * from works_csc;

       EID CID       SALARY
---------- ---- ----------
```

```
       101 c2          35000
       102 c3          35000
       103 c4          50000
       104 c2          30000
       105 c3          30000
       106 c1          40000
       108 c3          30000
       109 c3          28000
8 rows selected.
SQL> select * from works_bkup;


       EID CID       SALARY DELDATE
---------- ---- ---------- ---------
       100 c1          45000 16-SEP-17
```

***Example** 7*: Let's take a simple example to demonstrate the trigger which will enforce conditions while doing insertion, updation and deletion. In this example, we are using the *employee_table* table which has the following data.

| **ID** | **NAME** | **AGE** | **ADDRESS** | **SALARY** |
|---|---|---|---|---|
| 1 | Ramesh | 23 | Allahabad | 20000 |
| 2 | Suresh | 22 | Kanpur | 22000 |
| 3 | Mahesh | 24 | Ghaziabad | 24000 |
| 4 | Chandan | 25 | Noida | 26000 |
| 5 | Alex | 21 | Paris | 28000 |
| 6 | Sunita | 20 | Delhi | 30000 |

Let's write a program to create a row level trigger for the *employee_table* table that would fire for INSERT or UPDATE or DELETE operations performed on the *employee_table* table. This trigger will display the salary difference between the old values and new values:

```
1.    CREATE OR REPLACE TRIGGER salary_changes
2.    BEFORE DELETE OR INSERT OR UPDATE ON employee_table
3.    FOR EACH ROW
4.    WHEN (NEW.ID > 0)
5.    DECLARE
6.    s_diff number;
7.    BEGIN
8.      s_diff := :NEW.salary  - :OLD.salary;
9.      dbms_output.put_line('Old salary: ' || :OLD.salary);
10.     dbms_output.put_line('New salary: ' || :NEW.salary);
11.     dbms_output.put_line('Salary difference: ' || s_diff);
12.   END;
13.   /
```

**Code Explanation:**

**Line 1-4 :** Creation of trigger '*salary_changes*' whenever there is a change in the employee_table at **ROW** level also ensures at line 4, that the id must be present.

**Line 6:** Declaring variable.

**Line 8:** salary difference is calculated.

**Line 9-11**: Displays the old , new salary and the difference among them.

After the execution of the above code at SQL Prompt, it produces the following result.

Trigger created.

**Check the salary difference by procedure:**

Use the following code to get the old salary, new salary and salary difference after the trigger is created.

```
1.  DECLARE
2.   total_rows number(2);
3.  BEGIN
```

```
UPDATE  employee_table
SET salary = salary + 5000;
IF sql%notfound THEN
 dbms_output.put_line('no employee record updated');
 ELSIF sql%found THEN
 total_rows := sql%rowcount;
 dbms_output.put_line( total_rows || ' employee updated ');
   END IF;
END;
 /
```

Output:

```
Old salary: 20000
New salary: 25000
Salary difference: 5000
Old salary: 22000
New salary: 27000
Salary difference: 5000
Old salary: 24000
New salary: 29000
Salary difference: 5000
Old salary: 26000
New salary: 31000
Salary difference: 5000
Old salary: 28000
New salary: 33000
Salary difference: 5000
Old salary: 30000
New salary: 35000
Salary difference: 5000
6 customers updated
```

**Note:** As many times you executed this code, the old and new both salary is incremented by 5000 and hence the salary difference is always 5000.

After the execution of the above code again, you will get the following result.

```
Old salary: 25000
New salary: 30000
Salary difference: 5000
Old salary: 27000
New salary: 32000
Salary difference: 5000
Old salary: 29000
New salary: 34000
Salary difference: 5000
Old salary: 31000
New salary: 36000
Salary difference: 5000
Old salary: 33000
New salary: 38000
Salary difference: 5000
Old salary: 35000
New salary: 40000
Salary difference: 5000
6 customers updated
```

***Important Points***

Following are two very important points that should be noted carefully.

- OLD and NEW references are used for record level triggers; these are not available for table level triggers.
- If you want to query the table in the same trigger, then you should use the AFTER keyword, because triggers can query the table or change it again only after the initial changes are applied and the table is back in a consistent state.

## 2.5 VIEWING, DELETING AND MODIFYING TRIGGERS

**Viewing Triggers**

To know the information about triggers the following data dictionaries can be used.

- USER_TRIGGERS
- ALL_TRIGGERS
- DBA_TRIGGERS

**SYNTAX**

**SELECT TRIGGER_TYPE, TRIGGERING_EVENT,TABLE_NAME**

**FROM USER_TRIGGERS WHERE TRIGGER_NAME='TRIGGER NAME';**

In the above syntax the name of the trigger to the right hand side of variable TRIGGER_NAME must be given in all capitals.

Example : the following command will be used to view the information about trigger '*min_sal_chk*'. Note in the command the trigger name is given in all capitals.

*SQL>select trigger_type, triggering_event,table_name from user_triggers where trigger_name='MIN_SAL_CHK';*

Output

```
TRIGGER_TYPE    TRIGGERING_EVENT     TABLE_NAME
--------------            ------------------- -----------------------------
BEFORE EACH ROW INSERT OR UPDATE     WORKS_CSC
```

To view the content of trigger use the variable trigger_body as follows:

*SQL> select trigger_body from user_triggers where trigger_name='MIN_SAL_CHK';*

output

```
TRIGGER_BODY
---------------------------------------------------------------------------
begin
raise_application_error(15000,'salary must be more than 20000');
end;
```

**Modifying Triggers**

A trigger can not be altered by using the ALTER TRIGGER option. It is used only to recompile, enable or disable a trigger. If it is required to modify a trigger, use CREATE OR REPLACE TRIGGER statement. The OR REPLACE option allows you to overwrite the existing trigger with a new version of it.

There are two ways to prevent a trigger from running. One way is disabling a trigger, this would not remove the trigger from the RDBMS system but would not execute on events.

ALTER TRIGGER NOTIFICATIONS_NEW_LIKES DISABLE;

**Deleting a Trigger**

Now, if we really don't want a trigger even for reference in the future. We can delete a trigger by,

DROP TRIGGER NOTIFICATIONS_NEW_LIKES;

## 2.6 ENFORCING DATA INTEGRITY THROUGH TRIGGERS

**Update and delete trigger for parent table**

Example: The following trigger ensures that when a foreign key is deleted or updated from a child table, then its value in the parent table is made null (referential integrity)

Before doing this let us consider the two tables *employee_csc* and *dept*. The tables structure are given below

```
SQL> desc employee_csc;

 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 ENAME                                              VARCHAR2(30)
 STREET                                             VARCHAR2(40)
 CITY                                               VARCHAR2(30)
 EID                                       NOT NULL NUMBER(3)
 EMAIL                                              VARCHAR2(100)
 DEPTNO                                             NUMBER(2)
```

```
SQL> desc dept;

 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 DEPTNO                                    NOT NULL NUMBER(3)
 DNAME                                              VARCHAR2(10)
```

In the *dept* table the column deptno is the primary key which is the foreign key in *employee_csc* table. This is done as follows:

```
SQL> alter table employee_csc add foreign key(deptno) references dept(deptno);

Table altered.
```

So when an insertion into employee_csc takes place where the deptno is not in the *dept* table then it raises an error. This is because only those in the *dept* table alone can be inserted in the *employee_csc* table. In the below example a deptno '5' is inserted into *employee_csc* where it is the foreign key. But an error occurs that deptno is not in *dept* table.

```
SQL>insert into employee_csc values('karthik','rajaji
street','bangalore',114,'karthik@gmail.com',5);

insert into employee_csc values('karthik','rajaji
street','bangalore',114,'karthik@gmail.com',5)

*

ERROR at line 1:

ORA-02291: integrity constraint (SYSTEM.SYS_C007000) violated - parent
key not

found
```

The above is said to be database integrity. Now we can try to force integrity through triggers.

```
create or replace trigger dept_check
after delete or update of deptno on dept
for each row
begin
if updating and :old.deptno != :new.deptno or deleting then
update employee_csc set employee_csc.deptno=null
where employee_csc.deptno=:old.deptno;
end if;
end;
/
```

**Code Explanation:**

**Line 1-3 :** Creation of trigger '*dept_check*' that will be triggered after deletion or updation on field *deptno* in the *dept* table at **ROW** level takes place .

**Line 5-9 :** ensures before deletion or updation of field *deptno* in the *dept* table the corresponding foreign key values in *employee_csc* table is made as NULL

```
SQL> select ename,eid,deptno from employee_csc;
ENAME              EID    DEPTNO
--------------- ---------- ----------
anitha           100        1
aiswarya          101        2
chandra           102        2
hema             103        2
lalitha          104        1
raman             105        1
harini           106        3
danush            107        3
david            108        3
ananthi           109        4
sundar            110        4
raveena           111        4
radha            112        1
ramani            113        1
14 rows selected.
```

Now the trigger is executed as follows:

*SQL> @ c:\sql_prgs\dept_check_trigger.sql;*

*Trigger created.*

Next to test the above trigger, we will delete a deptno from dept table as follows:

```
SQL> delete dept where deptno=1;
1 row deleted.
SQL> select * from dept;
   DEPTNO DNAME
---------- ----------
        2 English
        3 CSC
        4 Physics
```

Now to know whether the corresponding dependent foreign key (deptno=1) value is replaced with NULL in *employee_csc*, we will use the select statement as follows.

```
SQL> select ename,eid,deptno  from employee_csc;

ENAME              EID    DEPTNO
-------------- ---------- ----------
anitha            100
aiswarya            101        2
chandra             102        2
hema               103        2
lalitha            104
raman               105
harini             106        3
danush              107        3
david              108        3
ananthi             109        4
sundar              110        4
raveena             111        4
radha              112
ramani              113
14 rows selected.
```

We can see that when the delete command is issued in the dept table (primary key value) the trigger is triggered and the foreign key value in employee_csc table is replaced with NULL.

**Delete cascade trigger for parent table**

```
create or replace trigger dept_cascade_delete
after delete on dept
for each row
-- Before row is deleted from dept
-- delete all rows from employee_csc table whose deptno value is same as
-- dept table
begin
delete from employee_csc where
employee_csc.deptno=:old.deptno;
end;
/
```

**Code Explanation:**

**Line 1-3 :** Creation of trigger ‘ *dept_cascade_delete*’ which will be triggered when a *deptno* is deleted in the *dept* table.

**Line 4-6 :** Whenever a *deptno* is deleted in the *dept* table the corresponding rows having values in *employee_csc* table will be deleted.

Now execute the trigger as follows:

*SQL> @ c:\sql_prgs\dept_check_trigger1.sql;*

*Trigger created.*

Now execute the following command where a primary key value is deleted from dept.

```
SQL> delete from dept where deptno=3;

1 row deleted.

SQL> select * from dept;

   DEPTNO DNAME

---------- ----------

       2 English

       4 Physics
```

Now the trigger will be triggered and now check the employee_csc table to check whether corresponding data is deleted in it (3 rows)

```
SQL> select ename,eid,deptno from employee_csc;


ENAME              EID     DEPTNO

--------------- ---------- ----------

anitha            100

aiswarya           101        2

chandra            102        2

hema              103        2

lalitha           104

raman             105

ananthi           109        4
```

```
sundar          110      4
raveena         111      4
radha           112
ramani          113

11 rows selected.
```

## 2.7 NESTED TRIGGERS

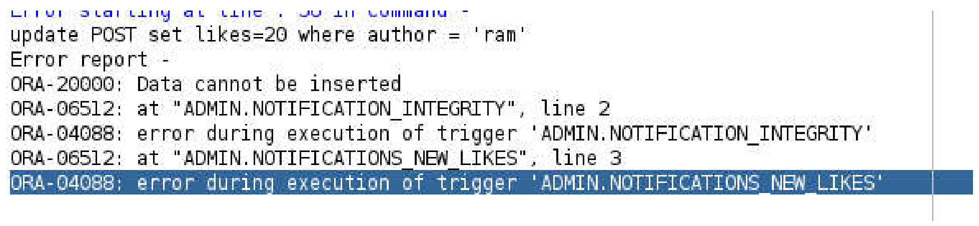
A nested trigger or recursive trigger is a trigger that gets executed because of another trigger. For example, let's create a trigger on the notifications table to prevent someone from directly inserting into the table. This ensures integrity of the data as we control the insertion only via an update trigger.

```
CREATE OR REPLACE TRIGGER NOTIFICATION_INTEGRITY

BEFORE  INSERT OR UPDATE ON NOTIFICATIONS

FOR EACH ROW

BEGIN

   raise_application_error(-20000

            , 'Data cannot be inserted');

END;
```

Now, no one can insert any data into Notifications table. Let's try to insert a new data in POST,

```
update POST set likes=20 where author = 'ram';
```

Output:



```
[illegible]
update POST set likes=20 where author = 'ram'
Error report -
ORA-20000: Data cannot be inserted
ORA-06512: at "ADMIN.NOTIFICATION_INTEGRITY", line 2
ORA-04088: error during execution of trigger 'ADMIN.NOTIFICATION_INTEGRITY'
ORA-06512: at "ADMIN.NOTIFICATIONS_NEW_LIKES", line 3
ORA-04088: error during execution of trigger 'ADMIN.NOTIFICATIONS_NEW_LIKES'
```

So, our integrity trigger prevents any data insertion and this also shows how triggers can be nested.

## 2.8 ADVANTAGES OF TRIGGERS

These are the following advantages of Triggers:

- Trigger generates some derived column values automatically
- Enforces referential integrity
- Event logging and storing information on table access
- Auditing
- Synchronous replication of tables
- Impose security authorizations
- Prevents invalid transactions

## 2.9 SEQUENCES

A sequence is an object in PL\SQL to generate unique sequences that can be assigned to auto numbering field or primary key where a unique ID is required.

For example, the banking sector might use this feature extensively where they would be required to generate unique numbers based on certain constraints for credit card or one time password (OTP).

## 2.10 CREATING SEQUENCE

Syntax of creating a sequence

```
CREATE SEQUENCE sequence_name

MINVALUE value

MAXVALUE value

START WITH value

INCREMENT BY value

CACHE vale;
```

Let's try to do a breakdown of above lines and try to understand what each line is expected to do.

Line 1: Create sequence keyword defines the sequence object with given sequence_name

Line 2: MINVALUE of the sequence to generate

Line 3: MAXVALUE of the sequence. Largest maximum value would be 9999999999999999999999999999

Line 4, 5: We specify a specific start value for our sequence to start. If this is omitted, minvalue becomes the start value. We also specify how this sequence has to be incremented.

Line 6: CACHE, is nothing but how many sequences have to be computed and kept in cache for performance optimization. For example, OLA might use 20000 for their OTP generator as it would improve their performance greatly.

Let's create an invoice sequence for a company, the specifications would be it should be in increments of 1 and should start from 0 and can go up-to 1,00,000 invoices.

CREATE SEQUENCE invoice_number

MINVALUE 1

START WITH 1

INCREMENT BY 1

CACHE 10;

## 2.11 REFERENCING A SEQUENCE

Now, we have created a sequence. This has to be referenced somewhere so we can see how this works. Let's create an invoice table for this company.

CREATE TABLE INVOICE (

INVOICENO Number,

INVOICEITEM VARCHAR2(255)

);

Our table is ready, let's try to insert some values into it by referencing the Sequence object we just created.

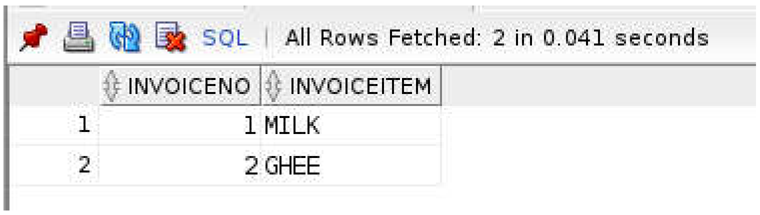
*INSERT INTO INVOICE VALUES (invoice_number.NEXTVAL, 'MILK');*

*INSERT INTO INVOICE VALUES (invoice_number.NEXTVAL, 'GHEE');*

Give,

*SELECT * FROM INVOICE;*

Output



All Rows Fetched: 2 in 0.041 seconds

| | INVOICENO | INVOICEITEM |
|---|---|---|
| 1 | 1 | MILK |
| 2 | 2 | GHEE |

## 2.12 ALTERING A SEQUENCE

Now we don't like to have our sequence in increments of 1, rather we would love to have increments of 10.

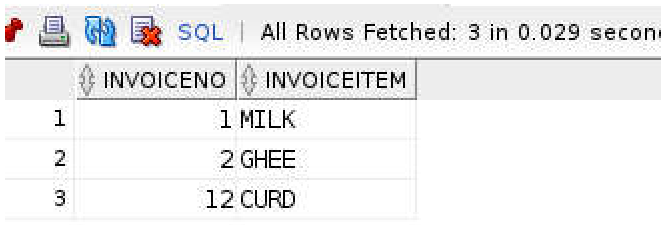
*ALTER SEQUENCE invoice_number*

*INCREMENT BY 10;*

Let's test our recent change by inserting another milk product to our invoice item.

*INSERT INTO INVOICE VALUES (invoice_number.NEXTVAL, 'CURD');*

Output



SQL | All Rows Fetched: 3 in 0.029 secon

|   | INVOICENO | INVOICEITEM |
|---|---|---|
| 1 | 1 | MILK |
| 2 | 2 | GHEE |
| 3 | 12 | CURD |

## 2.13 DELETING A SEQUENCE

To delete a sequence, we use the following syntax

DROP SEQUENCE sequence_name;

Example, if we wish to delete our invoice sequence we can give following SQL query

DROP SEQUENCE invoice_number;

## 2.14 SUMMARIZE

In this chapter an introduction to Triggers and how to create, delete them are discussed.

- A trigger is a stored procedure which gets fired by default when an incident occurs on a database.
- There are two types of triggers namely Row-level and statement-level triggers
- Row level trigger gets triggered for a row only.
- The old and new qualifiers are used in row-level triggers which are not compatible with statement-level triggers.
- After triggers are fired post the execution of a DML statement prior to the commit statement.
- The concept of nested Triggers is explained.
- An introduction to Sequence and its creation, reference and altering the created sequence and deleting a sequence are discussed with examples.

## 2.15 LIST OF REFERENCES

1. Nilesh Shah," Database systems using ORACLE- A simplified guide to SQL and PL/SQL.
2. https://www.tutorialspoint.com/plsql/plsql_triggers.htm
3. https://www.studytonight.com/plsql/plsql-triggers
4. https://www.softwaretestinghelp.com/triggers-in-pl-sql/
5. https://www.geeksforgeeks.org/sql-sequences/

## 2.16 QUESTIONS

1. Write a short note on triggers.
2. Write a short note on Trigger Classification
3. Write a short note on Implementing Triggers.
4. State and explain various aadvantages of Triggers

❖❖❖

3

# FILE ORGANIZATION AND INDEXING

**Unit Structure**

## 3.0 OBJECTIVE

At the end of this chapter the students will be able to:

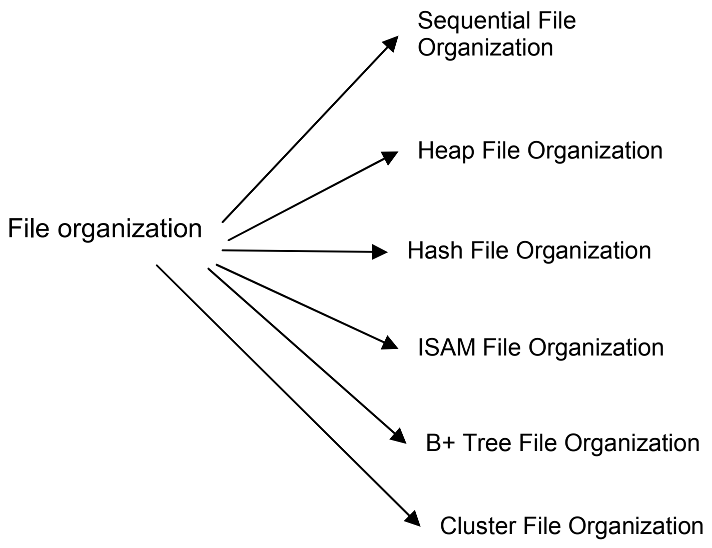
- Describe how files and records can be placed on disks and the effective way in which the records are organized in files
- Know the various commonly used file organizations used in database.
- Describe various indexes commonly used in database environments
- Understand the data structures which can support the various indexes
- To do manipulation with indexes on database

## 3.1 INTRODUCTION -FILE ORGANIZATION

A file is a collection of related information that can be stored in secondary storages.

File organization is a logical relationship among the records in a particular file. This defines how the files are mapped onto secondary storage in terms of disk blocks.

## 3.2 TYPES OF FILE ORGANIZATION



### 3.2.1 Sequential File Organization

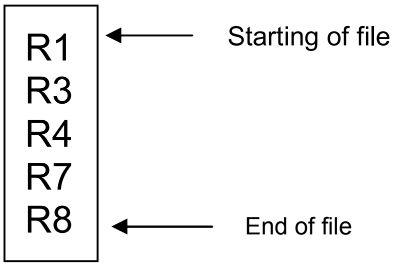
This is the easiest method of file organization. There are two ways to implement sequential file organization.

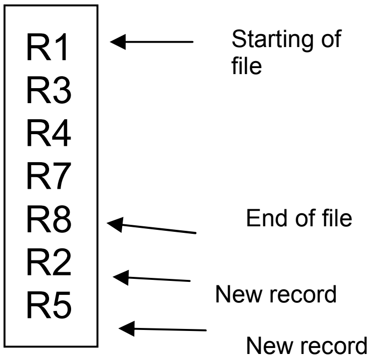
- Pile File Method
- Sorted File Method

**Pile File Method**

In this method the records are stored in a sequence. The records are inserted in the order of their arrival. In order to do any updation or deletion of a record, the whole file which is stored as a memory block has to be searched. When it is found , it will be updated or deleted accordingly.

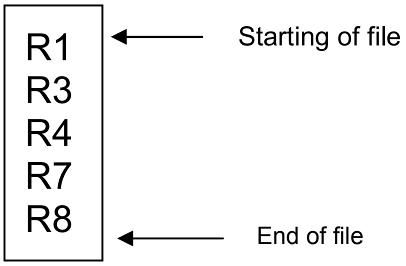
Let us consider the following records in a file (R1,R3,R4,R7,R8) , where R1 is the first record and R8 is the last record. Now when a new record says R2 is to be added it will be added at the end, i.e after R8 and so on.

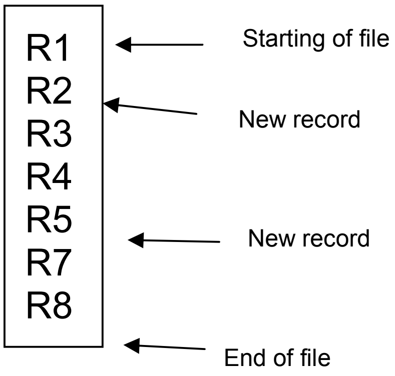




**Sorted File method**

In this the new record is inserted at the end, and then it will be sorted either in ascending or descending order. In case of updating a key on which the data is sorted, first it will update the record and then again sorted and the updated record will find its place.





Advantages of Sequential file organization

- File can be stored in the order it comes and then can be sorted.
- When all the records are to be processed like employee pay slip generation, student grade printing etc, this method is apt.

Disadvantage of sequential file organization

- As the sorting takes place each time a record is inserted or updated, most of the time is spent on this sorting operation and it needs space for the movement of data.

- In order to search for a particular record the file pointer has to go through all the records before it reaches the particular record, which is very time consuming.

### 3.2.2 Heap file organization

In this organization, the file is stored in a data block. So when a new record comes it will be stored in any of the data blocks which has space. If a data block is full, the new record is stored in some other block, which need not be the next data block. It is the responsibility of software to manage the records. Heap file organization does not support sorting, indexing etc.

Advantages of Heap file organization

- Fetching and retrieving records is faster for small databases
- When there is a large amount of data to be stored, this method is best as it finds wherever the memory block is free, it will occupy.

Disadvantages of Heap file organization

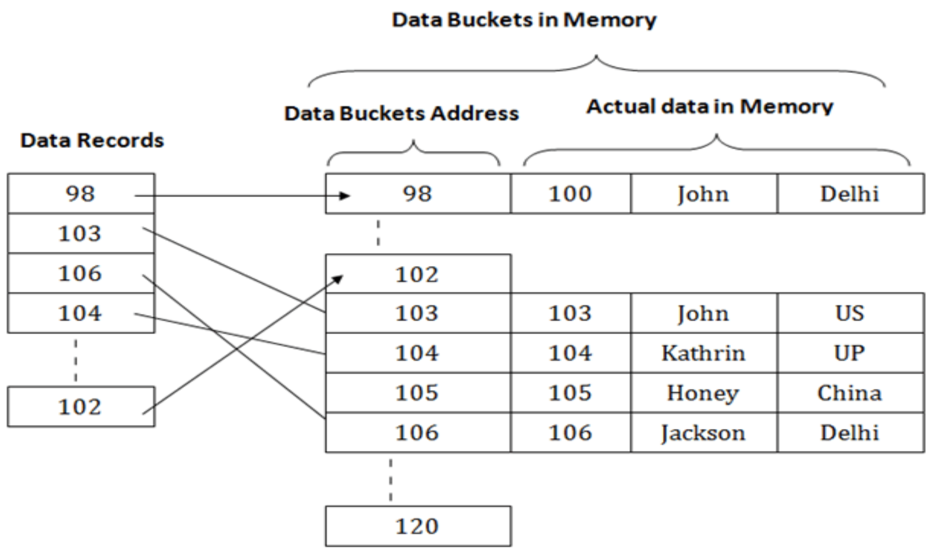
- Problem of unused memory block.
- This method will be a big problem when a large database is stored in this organization. As any search starts from the beginning of the file, it takes a lot of time for any updation, deletion etc,.

### 3.2.3 Hash file organization:

In a huge database structure, it is very inefficient to search all the index values and reach the desired data. Hashing technique is used to calculate the direct location of a data record on the disk without using index structure.

In this technique, data is stored at the data blocks whose address is generated by using the hashing function. The memory location where these records are stored is known as data bucket or data blocks.

In this, a hash function can choose any of the column value to generate the address. Most of the time, the hash function uses the primary key to generate the address of the data block. A hash function is a simple mathematical function to any complex mathematical function. We can even consider the primary key itself as the address of the data block. That means each row whose address will be the same as a primary key stored in the data block.



The above diagram shows data block addresses same as primary key value. This hash function can also be a simple mathematical function like exponential, mod, cos, sin, etc. Suppose we have a mod (5) hash function to determine the address of the data block. In this case, it applies mod (5) hash function on the primary keys and generates 3, 3, 1, 4 and 2 respectively, and records are stored in those data block addresses

Types of hashing

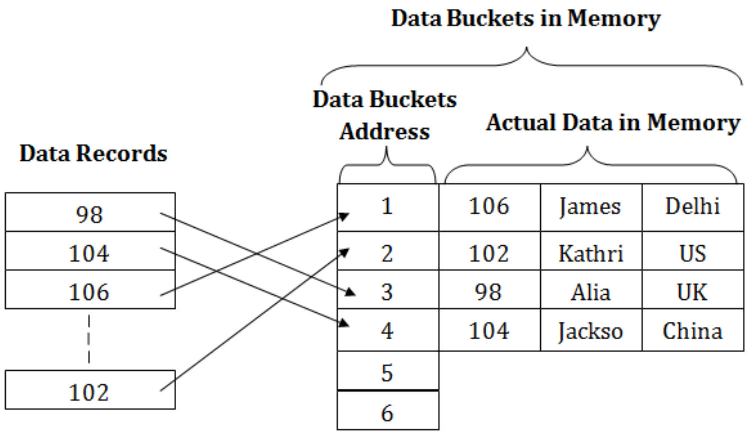
1) Static Hashing

2) Dynamic Hashing Technique

**Static Hashing**:

Static hashing uses a static hash function, so the resultant bucket address will always be the same.

Example, if we generate a hash for EMP_ID = 103 using a static has function mod (5) will always result in 3.



**Operations of Static Hashing,**

- Searching a record

When we need to find a record already stored in a bucket, static hashing really fast to retrieve it.

- Insert a record

When we want to insert a record into RDBMS, static hashing is really fast to insert the data.

**Dynamic Hashing:**

- The dynamic hashing method is used to overcome the problems of static hashing like bucket overflow.
- In this method, data buckets grow or shrink as the records increases or decreases. This method is also known as Extendable hashing method.
- This method makes hashing dynamic, i.e., it allows insertion or deletion without resulting in poor performance.

**Search a key:**

- First, calculate the hash address of the key.
- Check how many bits are used in the directory, and these bits are called as i.
- Take the least significant i bits of the hash address. This gives an index of the directory.
- Now using the index, go to the directory and find bucket address where the record might be.

**Insert a key:**

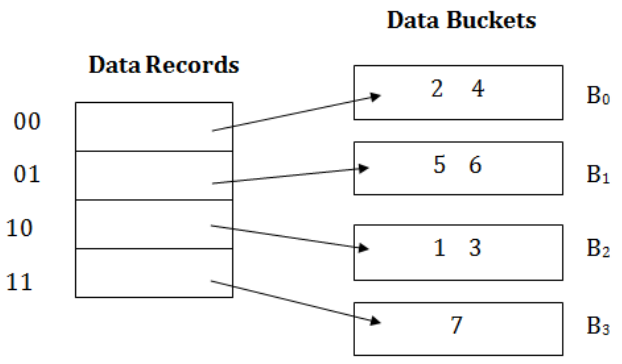
- Firstly, you have to follow the same procedure for retrieval, ending up in some bucket.
- If there is still space in that bucket, then place the record in it.
- If the bucket is full, then we will split the bucket and redistribute the records.

**Example:**

Consider the following grouping of keys into buckets, depending on the prefix of their hash address:

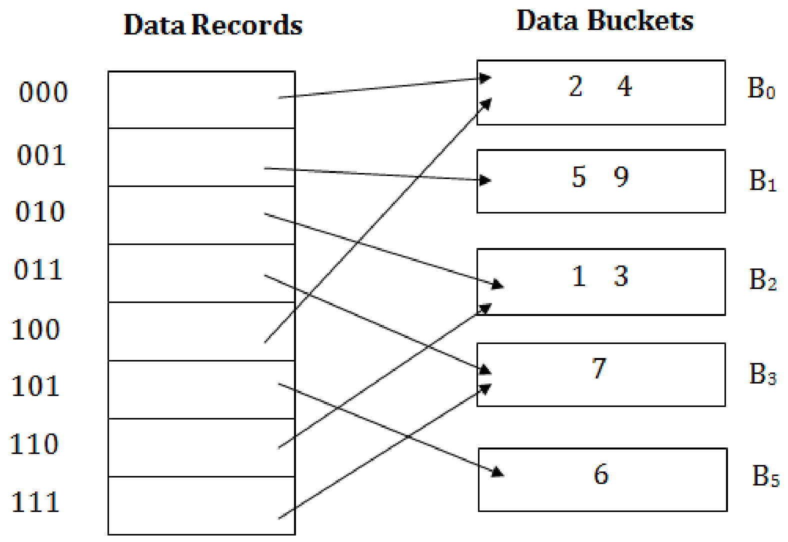
| Key | Hash address |
|---|---|
| 1 | 11010 |
| 2 | 00000 |
| 3 | 11110 |
| 4 | 00000 |
| 5 | 01001 |
| 6 | 10101 |
| 7 | 10111 |

The last two bits of 2 and 4 are 00. So it will go into bucket B0. The last two bits of 5 and 6 are 01, so it will go into bucket B1. The last two bits of 1 and 3 are 10, so it will go into bucket B2. The last two bits of 7 are 11, so it will go into B3.



***Insert key 9 with hash address 10001 into the above structure:***

- Since key 9 has hash address 10001, it must go into the first bucket. But bucket B1 is full, so it will get split.
- The splitting will separate 5, 9 from 6 since last three bits of 5, 9 are 001, so it will go into bucket B1, and the last three bits of 6 are 101, so it will go into bucket B5.
- Keys 2 and 4 are still in B0. The record in B0 pointed by the 000 and 100 entry because last two bits of both the entry are 00.
- Keys 1 and 3 are still in B2. The record in B2 pointed by the 010 and 110 entry because last two bits of both the entry are 10.
- Key 7 are still in B3. The record in B3 pointed by the 111 and 011 entry because last two bits of both the entry are 11.



**Advantages**

- In this method, the performance does not decrease as the data grows in the system. It simply increases the size of memory to accommodate the data.

- In this method, memory is well utilized as it grows and shrinks with the data. There will not be any unused memory lying.

- This method is good for the dynamic database where data grows and shrinks frequently.

**Disadvantages:**

- In this method, if the data size increases then the bucket size is also increased. These addresses of data will be maintained in the bucket address table. This is because the data address will keep changing as buckets grow and shrink. If there is a huge increase in data, maintaining the bucket address table becomes tedious.

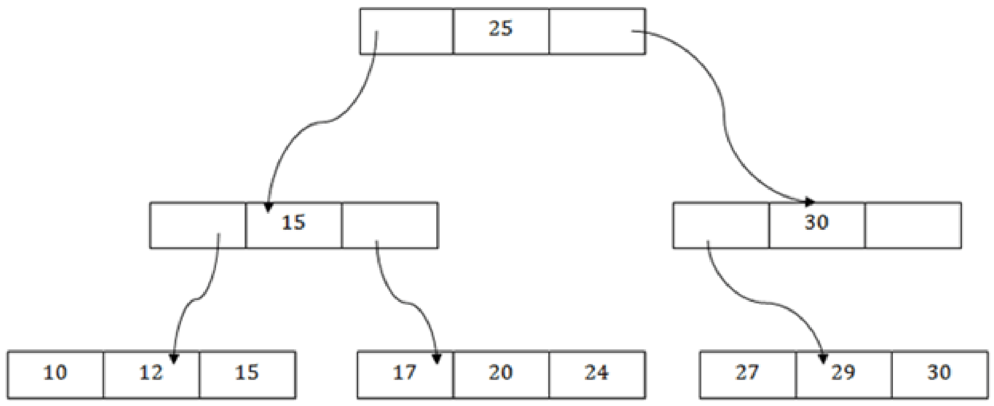
- In this case, the bucket overflow situation will also occur. But it might take little time to reach this situation than static hashing.

### 3.2.4 B+ File Organization

- B+ tree file organization is the advanced method of an indexed sequential access method. It uses a tree-like structure to store records in File.

- It uses the same concept of key-index where the primary key is used to sort the records. For each primary key, the value of the index is generated and mapped with the record.

- The B+ tree is similar to a binary search tree (BST), but it can have more than two children. In this method, all the records are stored only at the leaf node. Intermediate nodes act as a pointer to the leaf nodes. They do not contain any records.



***The above B+ tree shows that:***

- There is one root node of the tree, i.e., 25.
- There is an intermediary layer with nodes. They do not store the actual record. They have only pointers to the leaf node.
- The nodes to the left of the root node contain the prior value of the root and nodes to the right contain the next value of the root, i.e., 15 and 30 respectively.
- There is only one leaf node which has only values, i.e., 10, 12, 17, 20, 24, 27 and 29.
- Searching for any record is easier as all the leaf nodes are balanced.
- In this method, searching any record can be traversed through the single path and accessed easily.

***Advantages of B+ tree file organization***

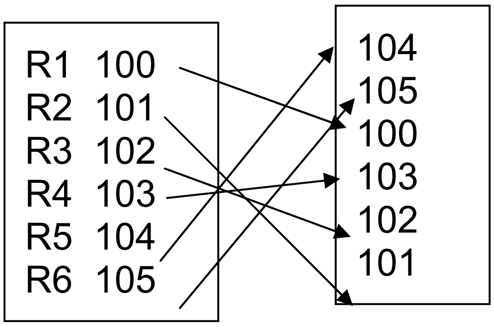
- In this method, searching becomes very easy as all the records are stored only in the leaf nodes and sorted in the sequential linked list.
- Traversing through the tree structure is easier and faster.
- The size of the B+ tree has no restrictions, so the number of records can increase or decrease and the B+ tree structure can also grow or shrink.
- It is a balanced tree structure, and any insert/update/delete does not affect the performance of the tree.

***Disadvantage of B+ tree file organization***

- This method is inefficient for the static method.

### 3.2.5 Indexed Sequential Access Method (ISAM)

ISAM is an advanced sequential file organization method where the records are stored using the primary key concept. An index is generated for each primary key and mapped with the record. This index contains the address of the record in the file.



Next to the primary key R1,R2 etc is the index which is nothing but the address where the record is stored in memory. So if a record is to be retrieved, it will be done through its index.

Here the storage area is divided into three parts namely prime area, overflow area and indexed area.

**Prime area** : In prime area the records are placed in sequential order.

**Overflow area** : When the prime area is full, the records will be stored here.

**Indexed area** : The index of the file is stored here. Index contains track number and highest key field value on that track.

**Advantages of ISAM**

- Since the search with indexing is very fast, searching a record in a huge database is quick and easy.
- This method supports range retrieval and partial retrieval like students with rollno starting from 45 to 60 and to fetch the students whose name starts with 'AN'.

**Disadvantage of ISAM**

- In order to store index value extra space is needed.
- New record insertion leads to reconstruction to maintain sequence.
- When a record is deleted the space used by it must be released. Otherwise the performance of the database will be slowed down.

### 3.2.6 Cluster File Organization

In cluster file organization, two or more related tables are stored within a file and so known as cluster. Using the primary key and foreign key attributes these two or more than them are mapped together and stored only once in the same data block. The key columns (primary and foreign key) are stored in this joined table only once. This reduces the cost of searching and retrieving the records from various tables as they are linked in one cluster.

Consider the two tables *employee_csc* and *works_csc* where EID is the primary key in *Employee_csc* and it is the foreign key in works_csc. Let CID is the primary key in *works_csc*

Employee_csc ← Primary key

| ENAME | STREET | CITY | EID |
|---|---|---|---|
| anitha | 1st street | chennai | 100 |
| aiswarya | 2nd street | chennai | 101 |
| chandra | 2nd street | chennai | 102 |
| hema | 3rd street | chennai | 103 |
| lalitha | metha street | mumbai | 104 |
| raman | krishnan street | bangalore | 105 |
| harini | kalam street | andhra | 106 |
| danush | ragav street | bangalore | 107 |
| david | kamaraj street | calcutta | 108 |
| ananthi | rajaji street | chennai | 109 |

Foreign

works_csc Primary

| SALARY | EID | CID | |
|---|---|---|---|
| 45000 | 100 | c1 | |
| 35000 | | 101 | c2 |
| 35000 | | 102 | c3 |
| 50000 | | 103 | c4 |
| 30000 | | 104 | c2 |
| 30000 | | 105 | c3 |
| 40000 | | 106 | c1 |
| 30000 | | 108 | c3 |
| 28000 | | 109 | c3 |

After a full outer join the the two tables are joined and can been seen in a cluster file like

Cluster key

```
EID EMPNAME        STREET          CITY     MID   MNAME   EID CID
----- ----------   ---------------  ----------  ---  ----------        ----- ----
  100 anitha     1st street        calcutta    m1  ajith    100 c1
  101 aiswarya   2nd street        chennai     m4  janani   101 c2
  102 chandra    2nd street        chennai     m6  jothi    102 c3
  103 hema       3rd street        chennai     m5  krishnan 103 c4
  104 lalitha    metha street      mumbai      m3  karthik  104 c2
  105 raman      krishnan street   bangalore   m2  hari     105 c3
  106 harini     kalam street      andhra
  107 danush     ragav street      bangalore   m7  dhanush  107 c4
  108 david      kamaraj street    calcutta
  109 ananthi    rajaji street     chennai
  112 krish      3rd street        bangalore
```

Using cluster key EID the two tables are stored as once and any insertion, deletion or updation can be done directly on these which will carry the operation in the individual tables also.

There are two types of cluster file organization

- **Indexed clusters** : In this the records are grouped based on cluster key and stored as one. In the above example employee_csc and works_csc are grouped based on cluster key EID and all related records are stored together
- **Hash clusters**: In this instead of cluster key, a hash key value is generated and stored in the joined table in the memory data block together.

**Advantages of Cluster file organization**

- When information from the related tables are to be extracted frequently, this method is the best.
- When there is a 1:M mapping between the tables, this organization works efficiently

**Disadvantages of Cluster file organization**

- This method is not suitable for very large database as the performance is low
- If the joining condition on which the tables are joined changes then complete rework or traversing back will take place. So when there is a change in joining condition another cluster only has to be formed.
- This method is not suitable for tables with 1:1 conditions.

## 3.3 INDEXING

### 3.3.1 Introduction

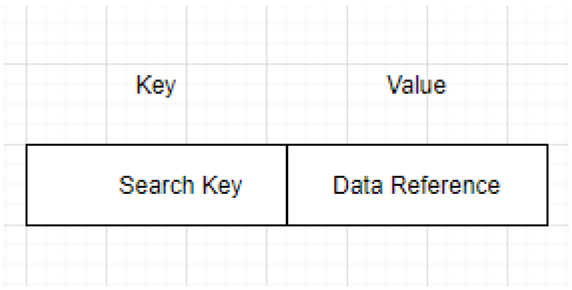
Imagine a database comprising millions of records of data, when we query this database what do you expect to happen ? Do we think this request will be optimal ? Will the users be happy about the response time ?

The very plain answer to this question is no. The database will start to slow up and become more clogged due to the huge volume of data it has to parse through to find what we need. We can solve this sluggishness problem in multiple ways, but we will concentrate only on indexing in this chapter.

To put it in plain words. Indexing is a technique to optimize the performance of a database by reducing the number of disk operations required on a given query. Indexing can be achieved by using specialized data structures in an RDBMS system.

Indexes are in general created using one or more database columns. As an example, we will look at the primary index which is typically a key-value pair. Whenever we create a primary index for a table, RDBMS creates a separate table consisting of a key which is the database column we specify and value for this key will be the reference to the data in the table.

Does this data structure ring a bell?. Yep, this is our good old hashmap. Any guess on what would be the read time for a hashmap (O(1)). Figure 1 shows how index exists in a database system.



The first column is the search key, which is nothing but a candidate key or primary key we set in a table. Usually these values could be sorted for a faster discovery.

The second column is nothing but a data reference or reference memory location where it points to a specific memory location in the database table. Imagine this as a linked list node, which we can reference. Practically we will have several complex data structures to handle row data. Which we will try to cover later in this chapter.

### *3.3.2 Database Indexing attributes*

Choosing an index is a careful process, there are a lot of inputs required in selecting an optimal index column. Few of the attributes are,

**1) Access Type:**

Access type refers to how we are going to access our data in the table. For example, most common ways of accessing data would be value based or range based.

Value based data examples, would be student details, ticket information. In all these access we would probably be looking for a cluster of relation data relating to an individual or real world modelled entity.

Range base data examples would be Stock exchange and other financial data.

**2) Access Time:**

Refers to the time required to find a particular data element or set of elements

**3) Insertion time:**

Refers to time taken to the time taken to find the appropriate space to insert the record

**4) Deletion time:**

Time taken to find an item and delete it. This would also include time taken to update the index data structure.

**5) Space Overhead**

It refers to additional space required to maintain an index.

### 3.3.3 Types of index files:

There are mainly three types of indexing

- Primary Index
- Secondary Index
- Clustering Index

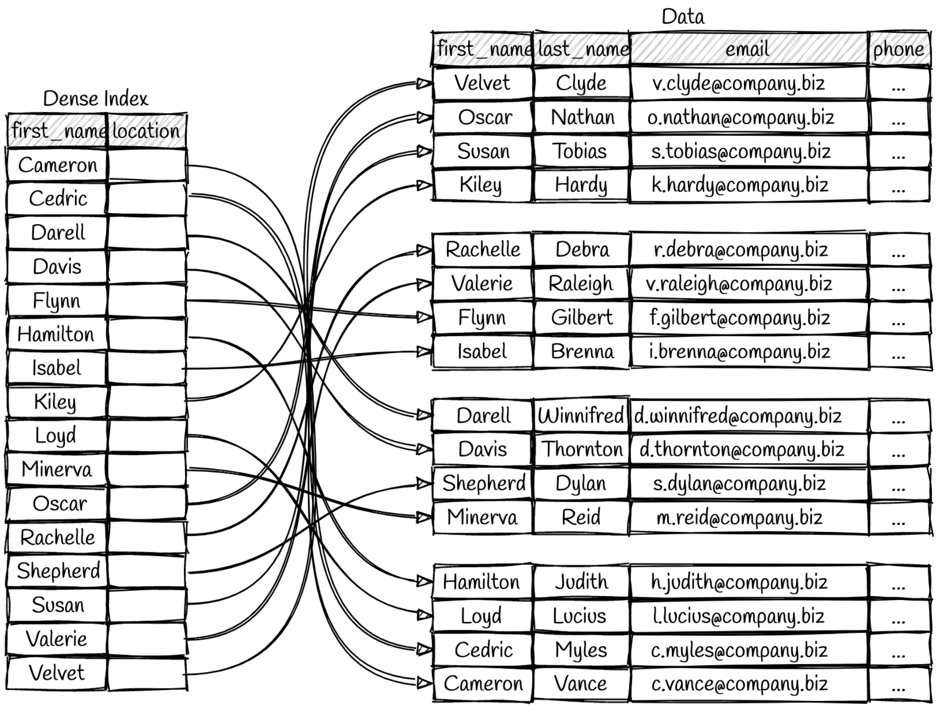
#### 3.3.3.1 Primary Index

If the index is created with the Primary key of a table, then it is called primary indexing. These keys are unique to each record. The records are stored in sorted order of primary key and so the searching operation is very efficient. The primary index can be classified as:

1) Dense Index
2) Sparse Index

## Dense Index:

In a dense index, a record is created for every search key valued in the database. This helps you to search faster but needs more space to store index records. In this Indexing, method records contain search key value and points to the actual record on the disk.

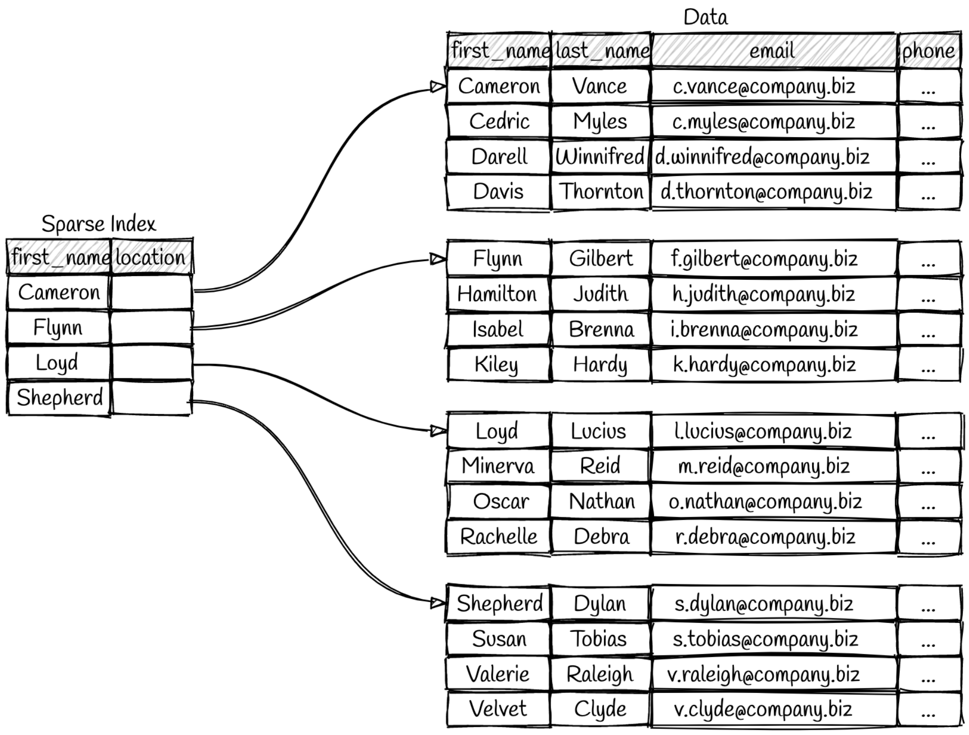

Dense Index
first_name | location
Cameron
Cedric
Darell
Davis
Flynn
Hamilton
Isabel
Kiley
Loyd
Minerva
Oscar
Rachelle
Shepherd
Susan
Valerie
Velvet
Data
first_name | last_name | email | phone
Velvet | Clyde | v.clyde@company.biz | ...
Oscar | Nathan | o.nathan@company.biz | ...
Susan | Tobias | s.tobias@company.biz | ...
Kiley | Hardy | k.hardy@company.biz | ...
Rachelle | Debra | r.debra@company.biz | ...
Valerie | Raleigh | v.raleigh@company.biz | ...
Flynn | Gilbert | f.gilbert@company.biz | ...
Isabel | Brenna | i.brenna@company.biz | ...
Darell | Winnifred | d.winnifred@company.biz | ...
Davis | Thornton | d.thornton@company.biz | ...
Shepherd | Dylan | s.dylan@company.biz | ...
Minerva | Reid | m.reid@company.biz | ...
Hamilton | Judith | h.judith@company.biz | ...
Loyd | Lucius | l.lucius@company.biz | ...
Cedric | Myles | c.myles@company.biz | ...
Cameron | Vance | c.vance@company.biz | ...

As we can see from the above image, a dense index is a strongly mapped index. Where all records are referenced to an index or key.

### *Sparse Index:*

Sparse index record that appears for only some values in the file. Sparse Index helps you to resolve the issues of dense Indexing in DBMS. In this method of indexing technique, a range of index columns stores the same data block address, and when data needs to be retrieved, the block address will be fetched.

However, since sparse Index stores index records for only some search-key values. It needs less space, less maintenance overhead for insertion, and deletions but It is slower compared to the dense Index for locating records.



Sparse Index

| first_name | location |
|---|---|
| Cameron | |
| Flynn | |
| Loyd | |
| Shepherd | |

Data

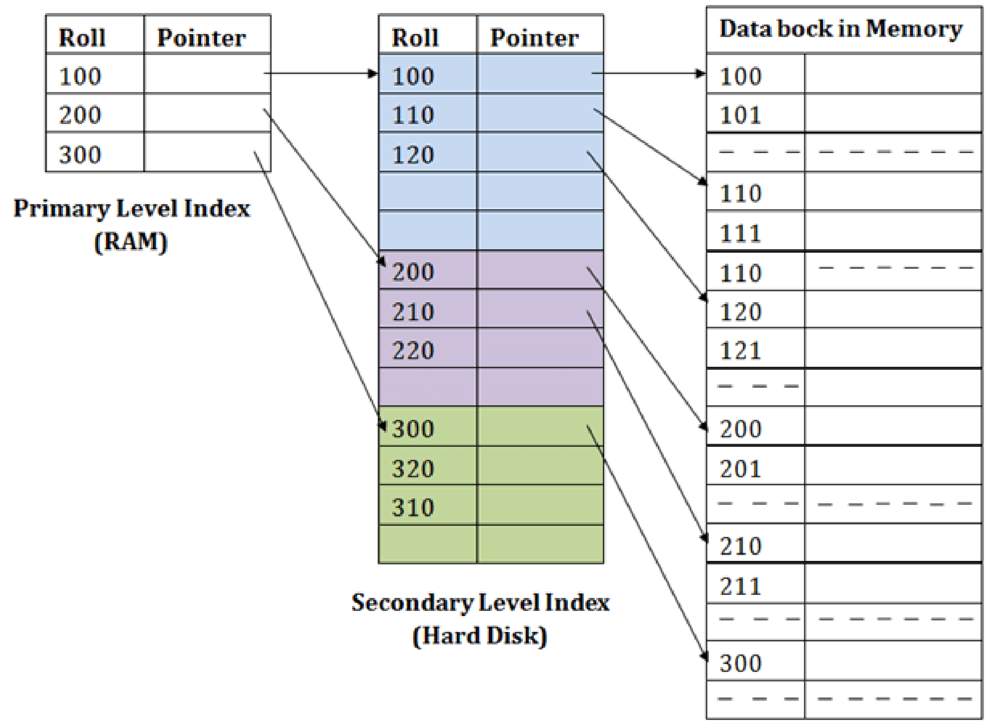
| first_name | last_name | email | phone |
|---|---|---|---|
| Cameron | Vance | c.vance@company.biz | ... |
| Cedric | Myles | c.myles@company.biz | ... |
| Darell | Winnifred | d.winnifred@company.biz | ... |
| Davis | Thornton | d.thornton@company.biz | ... |
| Flynn | Gilbert | f.gilbert@company.biz | ... |
| Hamilton | Judith | h.judith@company.biz | ... |
| Isabel | Brenna | i.brenna@company.biz | ... |
| Kiley | Hardy | k.hardy@company.biz | ... |
| Loyd | Lucius | l.lucius@company.biz | ... |
| Minerva | Reid | m.reid@company.biz | ... |
| Oscar | Nathan | o.nathan@company.biz | ... |
| Rachelle | Debra | r.debra@company.biz | ... |
| Shepherd | Dylan | s.dylan@company.biz | ... |
| Susan | Tobias | s.tobias@company.biz | ... |
| Valerie | Raleigh | v.raleigh@company.biz | ... |
| Velvet | Clyde | v.clyde@company.biz | ... |

In real-world applications, we may encounter a lot of utility to Sparse index. If data size is too huge to process, we can use sparse indexing.

### 3.3.3.2 Secondary Indexing

A field which has a unique value for each record can generate a secondary Index in DBMS, and it should be a candidate key. We also know it as a non-clustering index.

This two-level database indexing technique is used to reduce the mapping size of the first level. For the first level, a large range of numbers is selected because of this; the mapping size always remains small.



**Example**

- If you want to find the record of roll 111 in the diagram, then it will search the highest entry which is smaller than or equal to 111 in the first level index. It will get 100 at this level.

- Then in the second index level, again it does max (111) <= 111 and gets 110. Now using the address 110, it goes to the data block and starts searching each record till it gets 111.

- This is how a search is performed in this method. Inserting, updating or deleting is also done in the same manner.

### 3.3.3.3 Cluster Indexing

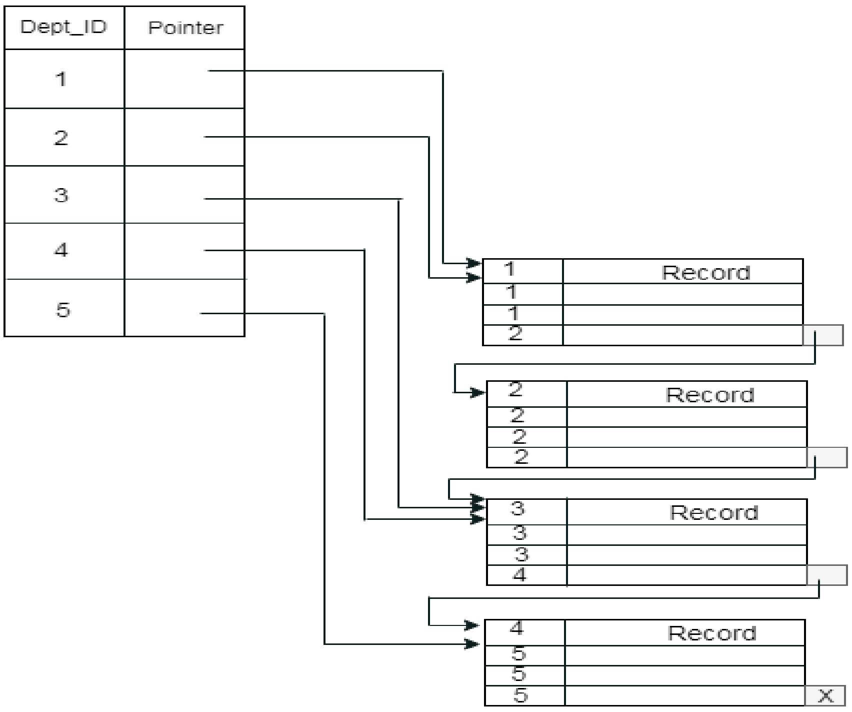
- A clustered index can be defined as an ordered data file. Sometimes the index is created on non-primary key columns which may not be unique for each record.

- In this case, to identify the record faster, we will group two or more columns to get the unique value and create an index out of them. This method is called a clustering index.

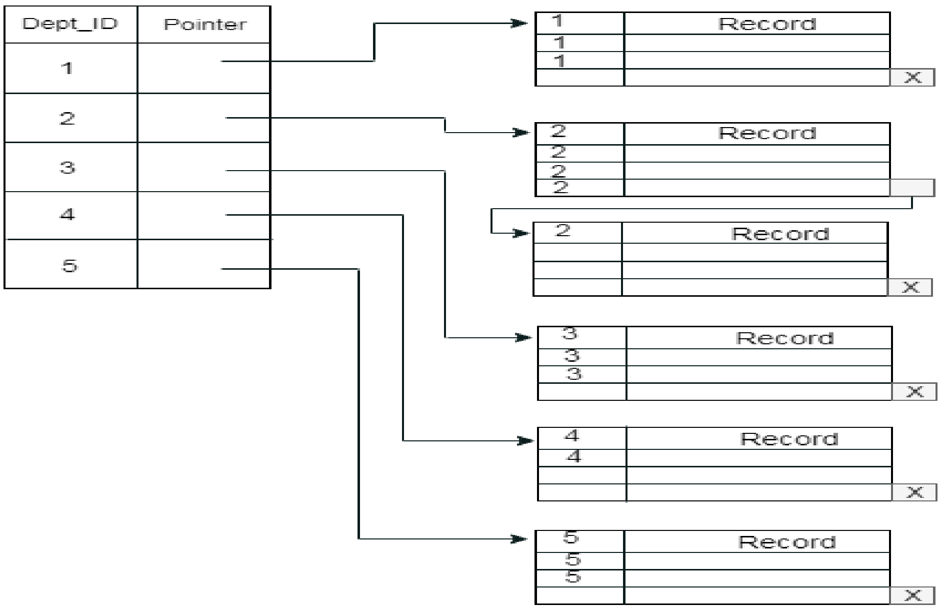
- The records which have similar characteristics are grouped, and indexes are created for these groups.

**Example**

suppose a company contains several employees in each department. Suppose we use a clustering index, where all employees which belong to the same Dept_ID are considered within a single cluster, and index pointers point to the cluster as a whole. Here Dept_Id is a non-unique key.



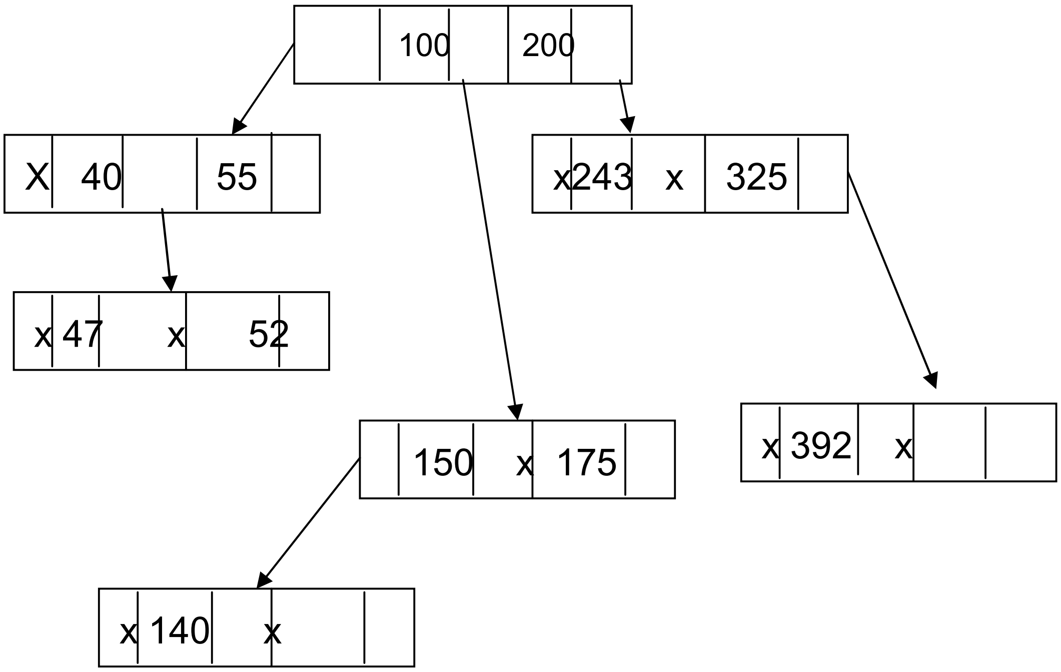
The previous schema is a little confusing because one disk block is shared by records which belong to the different cluster. If we use separate disk blocks for separate clusters, then it is called a better technique.



### 3.3.3.4 Tree based indexing - B-Tree Indexing

B-Tree index is a multilevel index format technique which is a balanced binary search tree.

In B-Tree indexing, all leaf nodes are interlinked with a link list, which leads to both random and sequential access. In this added advantage is it follows binary search which makes the searching faster. Since it has two pointers in each of its nodes, two-way search is possible. The below picture is an example of a m-way search tree where m represents the number of pointers in a particular node. If m=3, then each node contains 3 pointers, and each node would then contain 2 values.



## 3.4 COMPARISON OF FILE ORGANIZATIONS

### 3.4.1 Cost of various operation of DBMS on different types of files

| File Type | Scan | Equality Search | Range Search | Insert | Delete |
|---|---|---|---|---|---|
| Heap | PD | 0.5PD | PD | 2D | Search + D |
| Sorted | PD | $D\log_2(P)$ | $D\log_2(P)$ + matching pages | Search + PD | Search + PD |
| Clustered Tree Index | 1.5PD | $D\log_F(1.5P)$ | $D\log_F(1.5P)$ + matching pages | Search + D | Search + D |
| Unclustered Tree Index | PDR + Read index | D + $D\log_F(0.15P)$ | $D\log_F$(index size) + D*matching records | 3D + $D\log_F$(index size) | Search + 2D |
| Unclustered Hash Index | PDR + Read index | 2D | PD | 4D | Search + 2D |

|  |  |  |
|---|---|---|
| Where | P | no. of pages in the file. |
|  | D | amount of time required to read or write on a page. |
|  | R | no. of records in a particular page. |

***Heap Files***

Scan: Cost is PD since we have to retrieve each of P pages with each page taking D time.

Equality Search: If exactly one record matches the desired equality search then on average we must scan half of the file, assuming the record exists in only that part of the file. Hence cost is 0.5PD.

Range Search: This entire file must be scanned for matching records. So cost is PD.

Insert: If records are inserted at the end of page the time taken is fetching the page and writing back the page. So the cost is 2D.

Delete: Here time taken is searching for relevant records and writing back the page after deleting records from it. So the cost is Search + D.

**Sorted Files**

Scan: Cost is PD since we have to retrieve each of P pages with each page taking D time.

Equality Search: If we assume that the equality search is specified on the field by which the file is sorted, then we can search for the record by the help of binary search. Hence cost is $D\log_2(P)$.

Range Search: It is an equality search for all matching records. So the cost is $D\log_2(P)$ + matching pages. Insert: To insert the record while preserving the sorted order, first we have to search for the correct position in the file, add record and then fetch and rewrite all subsequent pages. So the cost is Search + PD. Delete: Here we search for a record, remove the record from the page, and rewrite the subsequent pages to fill the space created by the record which is deleted. Hence cost is Search + PD.

**Clustered Tree Index**

Scan: Here the effective number of pages is 1.5 times more than pages in heap files since page occupancy is 67%. So, Cost is 1.5PD since we have to retrieve all the pages with each page taking D time.

Equality Search: If data records are ordered as data entries in some index, then we do F-ary search. So cost in $D\log_F(1.5P)$.

Range Search: It is an equality search for all matching records. So the cost is $D\log_F(1.5P)$ + matching pages. Insert: Here time required is for searching the correct position for record in the page and writing back the page. So the cost is Search + D.

Delete: Similar to insert, first search for page, delete a record from it and write back the page. Cost is Search + D.

### *Unclustered Tree Index*

Scan: Here each record takes D time to read from a single page. So reading an R record from a page takes DR time. Hence total cost for P pages is PDR + Read index.

Equality Search: If we assume that data index size is one-tenth of data record, then no. leaf pages are

0.15P. So cost incurred is D + $D\log_F(0.15P)$.

Range Search: It includes equality search and matching pages. So cost is $D\log_F$ (index size) + D*matching records.

Insert: Time required is for searching the page, fetching it, adding records and writing back the page. So the cost is 3D + $D\log_F$ (index size).

Delete: First we search for the page where the record to be deleted is located, then fetch the page, remove record and write back the page. So the cost is Search + 2D.

### *Unclustered Hash Index*

Scan: Here each record takes D time to read from a single page. So reading an R record from a page takes DR time. Hence total cost for P pages is PDR + Read index.

Equality Search: If search is on the search key of hashed file, then total cost is of only getting the relevant page of data entry and record, so cost is 2D.

Range Search: This search can be as bad as scanning the whole file. Hence cost incurred in this is of retrieving all the pages. So cost is PD.

Insert: Here by using the search key, we can read the relevant pages, add a record to it and then write back the page. So the cost involved with it is 4D.

Delete: Cost involved with it is searching for the record, reading the page, deleting the record and writing back the page. So the cost is Search + 2D.

### *3.4.2 Comparison of I/O Costs*

- A heap file has good storage efficiency and supports fast scanning and insertion of records. However, it is slow for searches and deletions.
- A sorted file also offers good storage efficiency, but insertion and deletion of records is slow. Searches are faster than in heap files.
- A clustered file offers all the advantages of a sorted file and supports inserts and deletes efficiently. Searches are even faster than in sorted

files, although a sorted file can be faster when a large number of records are retrieved sequentially, because of blocked I/O efficiencies.

- Unclustered tree and hash indexes offer fast searches, insertion, and deletion, but scans and range searches with many matches are slow. Hash indexes are a little faster on equality searches, but they do not support range searches.

## 3.5 CREATING, DROPPING AND MAINTAINING INDEXES

### 3.5.1 Creating the index

When a new table is created with a primary key, Oracle automatically creates a new index for the primary columns.

Other than the primary key one can create indexes based on other columns using CREATE INDEX command.

**Syntax**

```
CREATE INDEX index_name

ON table_name(column1 [, column2,..])
```

1. The name of the index has to be specified for creation of index. The index name must be a meaningful one. For easy identification and remembrance it can consists of table name and column name along with suffix _I as follows:

*<table_name>_<column_name>_I*

2. The name of the table_name must be followed by one or more column on which the index is to be build

Consider the table *employee_csc*

```
SQL> desc employee_csc;
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 ENAME                                              VARCHAR2(30)
 STREET                                             VARCHAR2(40)
 CITY                                               VARCHAR2(30)
 EID                                       NOT NULL NUMBER(3)
 EMAIL                                              VARCHAR2(100)
 DEPTNO                                             NUMBER(2)
```

To view all indexes of a table, the following query can be used:

```
SELECT
    index_name,
    index_type,
    visibility,
    status
FROM
    all_indexes
WHERE
    table_name='TABLE NAME';
```

Now applying the syntax for our *employee_csc* table

```
SQL> select index_name,index_type,visibility,status from all_indexes where table_name='EMPLOYEE_CSC';

output

INDEX_NAME                  INDEX_TYPE              VISIBILIT STATUS
---------------------------- -------------------------  ---------  --------

SYS_C006987                  NORMAL                  VISIBLE   VALID
```

**Creating an index on one column**

Suppose, to look into table for the employees having same name,

```
SQL> create index emp_ename_i on employee_csc(ename);

Index created.
```

Now, showing the indexes will show the newly created index

```
SQL> select index_name,index_type,tablespace_name from
user_indexes where table_name='EMPLOYEE_CSC';

EMP_CITY_I              NORMAL          SYSTEM

EMP_ENAME_I             NORMAL          SYSTEM

SYS_C006987             NORMAL          SYSTEM
```

### 3.5.2 Altering the index

After creation of indexes, the attribute of that index can be changed using the ALTER INDEX command.

Syntax

```
ALTER [UNIQUE] INDEX <index name> ON <table name>
(<column(s)>);
```

Where UNIQUE - defines the index as a unique constraint for the table.

<index name> - name of the index table

<table name> - name of the base table on which index is created

<column(s)> - name of the columns in the table

```
SQL> alter index emp_ename_i rename to emp_ename_idx2;

Index altered.

Now we can list the index files again to see the change in name

SQL> select index_name,index_type,tablespace_name from
user_indexes where table_name='EMPLOYEE_CSC';

EMP_CITY_I              NORMAL          SYSTEM

EMP_ENAME_IDX2          NORMAL          SYSTEM

SYS_C006987             NORMAL          SYSTEM
```

We can disable the index using the alter index as follows

```
SQL> alter index emp_ename_idx2 unusable;
Index altered.
We can see the status by using the following command
SQL> select index_name,index_type,status from user_indexes where
table_name='EMPLOYEE_CSC';
EMP_CITY_I              NORMAL           VALID
EMP_ENAME_IDX2          NORMAL           UNUSABLE
SYS_C006987             NORMAL           VALID
```

We can enable the index using the alter index as follows

```
SQL> alter index emp_ename_idx2 rebuild;
Index altered.
We can see the status change as follows:
SQL> select index_name,index_type,status from user_indexes where
table_name='EMPLOYEE_CSC';
EMP_CITY_I              NORMAL           VALID
EMP_ENAME_IDX2          NORMAL           VALID
SYS_C006987             NORMAL           VALID
```

### 3.5.3 Removing the index

The created index can be dropped using the command

Syntax

```
DROP INDEX <index_name>
```

Now applying this to the created index:

```
SQL> DROP INDEX EMPLOYEE_CSC_ENAME_I;
Index dropped.
```

## 3.6 SUMMARIZATION

- No file organization is superior to the other one. Depending upon the situations one has to use accordingly.
- If selection queries are frequent, sorting and building indexes of the file is important.
- Hash based indexes are good for equality search.
- Sorted files and tree-based indexes are best for range search and equality search.
- One file can have several indexes based on different search key
- Indexes can be classified as clustered vs unclustered , primary vs secondary and dense vs sparse.
- Careful selection of index keys is important to speed up queries.
- Heap file is efficient in terms of space occupancy and insertion, but inefficient for search and deletion.
- Sorted files are efficient in terms of space occupancy but inefficient for insertion and deletion.
- Clustered tree index has the overhead in space occupancy. But perform well in insertion, deletion and searching.
- We can use CREATE INDEX to create indexes for columns other than primary key.
- Using ALTER INDEX modifications can be done on created index
- Using DROP INDEX the index can be deleted.

## 3.7 REFERENCES

1. https://www.geeksforgeeks.org/file-organization-in-dbms-set-1/
2. https://www.javatpoint.com/dbms-file-organization
3. https://www.tutorialspoint.com/dbms/dbms_file_structure.htm
4. https://www.javatpoint.com/indexing-in-dbms
5. https://www.geeksforgeeks.org/indexing-in-databases-set-1/
6. https://www.guru99.com/indexing-in-database.html
7. https://www.tutorialspoint.com/dbms/dbms_indexing.htm

❖❖❖❖

4

# FUNDAMENTALS OF PL/SQL

**Unit Structure**

## 4.0 OBJECTIVES

This chapter makes you to understand the basic concepts in PL/SQL.

It guides you to write PL/SQL block on yourself for a given problem.

## 4.1 OVERVIEW OF PL/SQL

The PL/SQL programming language was developed by Oracle Corporation in the late 1980s and early 90's to enhance the capabilities of SQL as a procedural extension language for SQL and the Oracle relational database. This is a combination of SQL embedded with the procedural features of programming languages.

- PL/SQL is a high-performance transaction-processing language and completely portable.
- PL/SQL provides a integrated, interpreted and OS independent programming environment.
- This can also be called directly from the command-line SQL*Plus interface.
- It has an option from external programming language calls to the database.
- PL/SQL is also available in Times Ten in-memory database, IBM DB2 and so on.

**Features of PL/SQL**

PL/SQL is built-in with SQL.

- It provides error checking facility.
- It offers numerous data types.
- It offers different programming structures.
- It has structured programming through functions and procedures.
- It has object-oriented programming.
- It has the development of web applications and server pages.

**Advantages of PL/SQL**

- SQL is the standard database language and PL/SQL is strongly integrated with SQL and supports static SQL and dynamic SQL.
- Static SQL supports DML operations and transaction control from PL/SQL block. In Dynamic SQL, easy to embed DDL statements on PL/SQL program blocks.
- PL/SQL allows sending an entire block of statements to the database at one time. Reduces network traffic, provides high performance on applications.
- PL/SQL gives high productivity to programmers as it can query, transform, and update data in a database.
- PL/SQL saves time on design and debugging by strong features, such as exception handling, encapsulation, data hiding, and object-oriented data types.
- Portable Applications can be written in PL/SQL.
- High security level.

- Access to predefined SQL packages.
- Support for Object-Oriented Programming.
- Support for developing Web Applications and Server Pages.

**PL/SQL Block Structure**

PL/SQL programs are divided and written in logical blocks of code. The blocks ar also have two different types.

1. Anonymous Block : A Block of code without name
2. Named Block : A Block of code has a specific name such as function name, subprogram name like, any valid name for the block.

Each block has three sections.

| **S.No** | **Sections & Description** |
|---|---|
| 1 | **Declarations** This section starts with the keyword **DECLARE**. It is an optional section and defines all variables, cursors, subprograms, and other elements to be used in the program. |
| 2 | **Executable Commands** This section is enclosed between the keywords **BEGIN** and **END.** It consists of the executable statements of the program. It should have at least one executable statement, NULL command is used to indicate that nothing should be executed. |
| 3 | **Exception Handling** This section starts with the keyword **EXCEPTION.** This optional section contains exception(s) that handle errors in the program. |

Every PL/SQL statement ends with a semicolon (;). PL/SQL blocks can be nested within other PL/SQL blocks using BEGIN and END.

The syntax of PL/SQL block structure **DECLARE**

**<declarations section>**

**BEGIN**

**<executable command(s)>**

**EXCEPTION**

**<exception handling>**

**END;**

**Example :**

```
DECLARE

msg  varchar2(20):= 'Hello World';

BEGIN

dbms_output.put_line (msg);

END;
```

The end; line signals the end of the PL/SQL block. To run the code from the SQL command line, use / at the beginning of the first blank line after the last line of the code.This produces the following result

```
Hello World
```

**The PL/SQL Comments**

Program comments are explanatory statements that can be included in the PL/SQL code that you write and helps anyone reading its source code. All programming languages allow some form of comments.

The PL/SQL supports single-line and multi-line comments. All characters available inside any comment are ignored by the PL/SQL compiler.

- Single-line comments ,the line start by  -- (double hyphen)
- Multi-line comments must enclosed by /* and */.

```
DECLARE

  -- variable declaration

  var1 varchar2(20):= 'Hello World';

BEGIN

    dbms_output.put_line(var1);

END;

/
```

When the above code is executed at the SQL prompt, it produces the following result

```
Hello World
```

## 4.2 THE PL/SQL IDENTIFIERS

PL/SQL identifiers are constants, variables, exceptions, procedures, cursors, and reserved words. This consists of a letter optionally followed by more letters, numerals, dollar signs, underscores, and number signs. This should not exceed 30 characters. By default, identifiers are not case-sensitive. The identifier can be named integer or INTEGER to represent a numeric value. You cannot use a reserved keyword as an identifier.

**Variable Declaration in PL/SQL**

PL/SQL variables must be declared in the declaration section or in a package as a global variable. When you declare a variable, PL/SQL allocates memory for the variable's value and the storage location is identified by the variable name.

The syntax for declaring a variable is

variable_name [CONSTANT] datatype [NOT NULL] [:= | DEFAULT initial_value]

Where, *variable_name* is a valid identifier in PL/SQL

sales number(10, 2);

pi CONSTANT double precision := 3.1415;

name varchar2(25);

address varchar2(100);

When you provide a size, scale or precision limit with the data type, it is called a **constrained declaration**. Constrained declarations require less memory than unconstrained declarations.

**Example:**

sales number(10, 2);

name varchar2(25);

address varchar2(100);

Initializing Variables in PL/SQL

**DEFAULT**

PL/SQL assigns it a default value of NULL. If you want to initialize a variable with a value other than the NULL value, you can do so during the declaration, using either of the following −

The **DEFAULT** keyword

The **assignment**(:=)operator

**Example:**

counter binary_integer := 0;

greetings varchar2(20) DEFAULT 'Have a Good Day';

You can also specify that a variable should not have a **NULL** value using the **NOT NULL** constraint. If you use the NOT NULL constraint, you must explicitly assign an initial value for that variable.

It is a good programming practice to initialize variables properly otherwise, sometimes programs would produce unexpected results

**Constants**

A constant holds a value that once declared, does not change in the program. A constant declaration specifies its name, data type, and value, and allocates storage for it. The declaration can also impose the NOT NULL constraint.

**Declaring a Constant**

A constant is declared using the CONSTANT keyword. It requires an initial value and does not allow that value to be changed.

**Example:**

**PI CONSTANT NUMBER := 3.141592654;**

**PL/SQL Block:**

```
DECLARE
  -- constant declaration
  pi constant number := 3.141592654;
  -- other declarations
  radius number(5,2);
  dia number(5,2);
  circumference number(7, 2);
  area number (10, 2);
BEGIN
  -- processing
  radius := 9.5;
  dia := radius * 2;
  circumference := 2.0 * pi * radius;
  area := pi * radius * radius;
  -- output
  dbms_output.put_line('Radius: ' || radius);
```

```
   dbms_output.put_line('Diameter: ' || dia);
   dbms_output.put_line('Circumference: ' || circumference);
   dbms_output.put_line('Area: ' || area);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
Radius: 9.5
Diameter: 19
Circumference: 59.69
Area: 283.53
```

**Literals**

A literal is an explicit numeric, character, string, or Boolean value not represented by an identifier. For example, TRUE, 786, NULL, 'tutorials point' are all literals of type Boolean, number, or string. PL/SQL, literals are case-sensitive. PL/SQL supports the following kinds of literals

| | |
|---|---|
| Numeric Literals | 050 78 -14 0 +32767<br>6.6667 0.0 -12.0 3.14159 +7800.00<br>6E5 1.0E-8 3.14159e0 -1E38 -9.5e-3 |
| Character Literals | 'A' '%' '9' ' ' 'z' '(' |
| String Literals | 'Hello, world!'<br>'Tutorials Point'<br>'19-NOV-12' |
| BOOLEAN Literals | TRUE, FALSE, and NULL. |
| Date and Time Literals | DATE '1978-12-25';<br>TIMESTAMP '2012-10-29 12:01:01'; |

## 4.3 PL/SQL EXPRESSIONS AND COMPARISONS

Expressions are constructed using operands and operators. An operand may be a variable or constant that contributes value to an expression.

Simple arithmetic expression is:

-X / 2 + 3

Unary operators like negation operator (-) operate on one operand; binary operators like the division operator (/) operate on two operands.

**PL/SQL OPERATORS**

An operator is a symbol that tells the compiler to perform specific mathematical or logical operation.

Types of operators

**Arithmetic Operators**

**Relational Operators**

**Comparison Operators**

**Logical Operators**

**Arithmetic Operators**

Following table shows all the arithmetic operators supported by PL/SQL. Let us assume variable A holds 20 and variable B holds 15, then

| **Operator** | **Description** | **Example** |
|---|---|---|
| + | Adds two operands | A + B = 35 |
| - | Subtracts second one from the first one | A - B = 5 |
| * | Multiplies both operands | A * B = 300 |
| / | Divides numerator by de-numerator | A / B = 1 |
| ** | Exponentiation operator, raises one operand to the power of other | A ** 2 = 400 |

**Relational Operators**

Relational operators compare two expressions or values and return a Boolean result. Following table shows all the relational operators supported by PL/SQL. Consider the variable A has 10 and variable B has 20, then –

| Operator | Description | Example |
|---|---|---|
| = | Checks if the values of two operands are equal or not, if yes then the condition is true. | (A = B) is not true. |
| != <br> <> <br> ~= | Checks if the values of two operands are equal or not, if values are not equal then the condition is true. | (A != B) is true. |
| > | Checks if the value of the left operand is greater than the value of right operand, if yes then the condition is true. | (A > B) is not true. |
| < | Checks if the value of the left operand is less than the value of the right operand, if yes then the condition is true. | (A < B) is true. |
| >= | Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition is true. | (A >= B) is not true. |
| <= | Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition is true. | (A <= B) is true. |

**Comparison Operators**

Comparison operators are used for comparing one expression to another. The result is from TRUE, FALSE or NULL.

| Operator | Description | Example |
|---|---|---|
| LIKE | The LIKE operator compares a character, string, or CLOB value to a pattern and returns TRUE if the value matches the pattern else FALSE. | If 'Zara Ali' like 'Z%_i' returns true, whereas, 'Nuha Ali' like 'Z% A_i' returns false. |
| BETWEEN | The BETWEEN operator tests whether a value lies in a specified range. x BETWEEN a AND b means that x >= a & x <= b. | If x is 10 then, x between 15 and 20 returns true, x between 5 and 10 returns true, but between 11 and 20 returns false. |

| | | |
|---|---|---|
| IN | The IN operator tests set membership. x IN (set) means that x is equal to any member of set. | If x = 'm' then, x in ('a', 'b', 'c') is false but x in ('m', 'n', 'o') is true. |
| IS NULL | The IS NULL operator returns the BOOLEAN value TRUE if its operand is NULL or FALSE if it is not NULL. Comparisons involving NULL values is always NULL. | If x = 'm', then 'x is null' is false. |

**Logical Operators**

Following table shows the Logical operators supported by PL/SQL. All these operators work on Boolean operands and produce Boolean results. Let us consider variable A has true and variable B has false.

| **Operator** | **Description** | **Example** |
|---|---|---|
| and | Called the logical AND operator. If both the operands are true then condition is true. | A and B is false. |
| or | Called the logical OR Operator. If any of the two operands is true then condition becomes true. | A or B is true. |
| not | Called the logical NOT Operator. Used to reverse the logical state of its operand. If a condition is true then Logical NOT operator will make this false. | not (A and B) is true. |

**PL/SQL Operator Precedence**

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Some operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator.

For example, x = 7 + 5 * 2; here, x is assigned 17, not 24 because operator * has higher precedence than +, so it first gets multiplied with 5*2 and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear in bottom. Higher precedence operators will be evaluated first.

The precedence of operators : =, <, >, <=, >=, <>, !=, ~=, ^=, IS NULL, LIKE, BETWEEN, IN.

| **Operator** | **Operation** |
|---|---|
| ** | exponentiation |
| +, - | identity, negation |
| +, - | multiplication, division |
| +, -, \|\| | addition, subtraction, concatenation |
| comparison | |
| NOT | logical negation |
| AND | conjunction |
| OR | inclusion |

**CASE Expressions**

There are two types of expressions used in CASE statements: simple and searched. These expressions correspond to the type of CASE statement in which they are used.

**Simple CASE expression**

A simple CASE expression selects a result from one or more alternatives and returns the result. It contains a block that stretch over several lines, it really is an expression that forms part of a larger statement, like assignment or a procedure call. The CASE expression uses a selector, an expression whose value determines which alternative to return.

**Searched CASE Expression**

A searched CASE expression lets you test different conditions instead of comparing a single expression with different values. This expression has no selector. Each WHEN clause contains a search condition that yields a BOOLEAN value, so you can test different variables or multiple conditions in a single WHEN clause.

**Null Values in Comparisons, Conditional Statements**

When working with nulls, you can avoid some common mistakes by keeping in mind the following rules:

- Comparisons in null values provide always NULL only.
- Using the logical operator NOT with a null provides NULL
- In conditional statements, if condition is NULL, its associated sequence of statements is not executed.
- If the expression is a simple CASE statement or CASE expression is NULL, it cannot be matched by WHEN NULL in condition. Here, need to use the searched case syntax and test WHEN expression IS NULL.

## 4.4. PL/SQL DATA TYPES

The PL/SQL variables, constants and parameters must have a valid data type, which specifies a storage format, conditions with valid range of values.

| S. No | Category & Description |
|---|---|
| 1 | **Scalar**<br><br>Single values which has no internal components, like **NUMBER, DATE** or **BOOLEAN**. |
| 2 | **Large Object (LOB)**<br><br>Pointers to large objects which are stored separately from other data items, such as text, graphic images, video clips, and sound waveforms. |
| 3 | **Composite**<br><br>Data items that have internal components that can be accessed individually. For example, collections and records. |
| 4 | **Reference**<br><br>Pointers to other data items. |

**Numeric Types**

Following table shows the numeric data types and their sub-types –

| S. No | Data Type & Description |
|---|---|
| 1 | **PLS_INTEGER**<br>Signed integer in range -2,147,483,648 through 2,147,483,647, represented in 32 bits |
| 2 | **BINARY_INTEGER**<br>Signed integer in range -2,147,483,648 through 2,147,483,647, represented in 32 bits |
| 3 | **BINARY_FLOAT**<br>Single-precision floating-point number |
| 4 | **BINARY_DOUBLE**<br>Double-precision floating-point number |
| 5 | **NUMBER(prec, scale)**<br>Numeric values with fixed or floating-point number with absolute value in range 1E-130 to (but not including) 1.0E126. A NUMBER variable can also represent 0 |
| 6 | **DEC(prec, scale)**<br>Fixed-point type specified in ANSI with maximum precision of 38 decimal digits |
| 7 | **DECIMAL(prec, scale)**<br>Fixed-point type specified in IBM with maximum precision of 38 decimal digits |
| 8 | **NUMERIC(pre, secale)**<br>Floating type with 38 decimal digits |
| 9 | **DOUBLE PRECISION**<br>Floating-point type specified in ANSI with maximum precision of 126 binary digits (approximately 38 decimal digits) |
| 10 | **FLOAT**<br>Floating-point type specified in ANSI and IBM with maximum precision of 126 binary digits (approximately 38 decimal digits) |

| | |
|---|---|
| 11 | **INT**<br><br>Integer type specified in ANSI with maximum precision of 38 decimal digits |
| 12 | **INTEGER**<br><br>Integer type specified in ANSI and IBM with maximum precision of 38 decimal digits |
| 13 | **SMALLINT**<br><br>Integer type specified in ANSI and IBM with maximum precision of 38 decimal digits |
| 14 | **REAL**<br><br>Floating-point type with 63 binary digits as maximum precision (approximately 18 decimal digits) |

Following is a valid declaration –

```
DECLARE
  num1 INTEGER;
  num2 REAL;
  num3 DOUBLE PRECISION;
BEGIN
  null;
END;
/
```

**Character Types**

Following Table shows the character data types and their sub-types

| **S. No** | **Data Type & Description** |
|---|---|
| 1 | **CHAR**<br><br>Fixed-length character string with maximum size of 32,767 bytes |
| 2 | **VARCHAR2**<br><br>Variable-length character string with maximum size of 32,767 bytes |

| | |
|---|---|
| 3 | **RAW**<br><br>Variable-length binary or byte string with maximum size of 32,767 bytes, not interpreted by PL/SQL |
| 4 | **NCHAR**<br><br>Fixed-length national character string with maximum size of 32,767 bytes |
| 5 | **NVARCHAR2**<br><br>Variable-length national character string with maximum size of 32,767 bytes |
| 6 | **LONG**<br><br>Variable-length character string with maximum size of 32,760 bytes |
| 7 | **LONG RAW**<br><br>Variable-length binary or byte string with maximum size of 32,760 bytes, not interpreted by PL/SQL |
| 8 | **ROWID**<br><br>Physical row identifier, the address of a row in an ordinary table |
| 9 | **UROWID**<br><br>Universal row identifier (physical, logical, or foreign row identifier) |

**Boolean Types**

The **BOOLEAN** data type stores logical values that are used in logical operations. The logical values are the Boolean values **TRUE** and **FALSE** and the value **NULL**.

However, SQL has no data type equivalent to BOOLEAN. Therefore, Boolean values cannot be used in

- SQL statements
- Built-in SQL functions
- functions invoked SQL statements

**Date Time Types**

The **DATE** data type is used to store fixed-length date-times, which include the time of day in seconds. Valid dates from 1st January , 4712 BC to 31st December , 9999 AD.

The default date format is set by the Oracle initialization parameter NLS_DATE_FORMAT.

For example, 'DD-MON-YY ' is the default one, which includes a two-digit number for the day of the month, first three characters of the month name, and the last two digits of the year.

Eg. 01-SEP-12.

Each DATE includes the century, year, month, day, hour, minute, and second.

The table shows the Valid Date-Time Values and its Interval Types.

| **Name of the Field** | **Date-Time Values** | **Interval Values** |
|---|---|---|
| YEAR | -4712 to 9999 (excluding year 0) | Any nonzero integer |
| MONTH | 01 to 12 | 0 to 11 |
| DAY | 01 to 31 (limited by the values of MONTH and YEAR, according to the rules of the calendar for the locale) | Any nonzero integer |
| HOUR | 00 to 23 | 0 to 23 |
| MINUTE | 00 to 59 | 0 to 59 |
| SECOND | 00 to 59.9(n), where 9(n) is the precision of time fractional Seconds | 0 to 59.9(n), where 9(n) is the precision of interval fractional seconds |
| TIMEZONE_HOUR | -12 to 14 (range accommodates daylight savings time changes) | Not applicable |
| TIMEZONE_MINUTE | 00 to 59 | Not applicable |
| TIMEZONE_REGION | Found in the dynamic performance view V$TIMEZONE_NAMES | Not applicable |
| TIMEZONE_ABBR | Found in the dynamic performance view V$TIMEZONE_NAMES | Not applicable |

## Large Object (LOB) Data Types

Large Object (LOB) data types refer to large data items such as text, graphic images, video clips, and sound waveforms. LOB data types allow efficient, random, piecewise access to this data. Following are the predefined PL/SQL LOB data types –

| **Data Type** | **Description** | **Size** |
|---|---|---|
| BFILE | store large binary objects in operating system files outside the database. | System-dependent. Cannot exceed 4 gigabytes (GB). |
| BLOB | store large binary objects in the database. | 8 to 128 terabytes (TB) |
| CLOB | store large blocks of character data in the database. | 8 to 128 TB |
| NCLOB | store large blocks of NCHAR data in the database. | 8 to 128 TB |

**Summary**

PL/SQL Blocks contain three sections: Declaration, Execution and Exception.

PL/SQL Expressions used to retrieve particular data for the database.

PL/SQL Operators: Arithmetic Operators, Relational Operators, Comparison Operators, Logical Operators.

PL/SQL Data types: Numeric, Character, Boolean and Date Types.

**Review Questions**

1. Explain PL/SQL Block Structure with simple example.
2. Discuss briefly on Fundamentals of PL/SQL.
3. How you declare variables and constants in PL/SQL?
4. List out the PL/SQL Operators and Explain.
5. Give brief note on PL/SQL Datatypes.

❖❖❖❖

# 5

# CONTROL STRUCTURES

**Unit Structure**

## 5.0 OBJECTIVES

This chapter makes you to understand the concepts in PL/SQL control structures

Helps to improve your coding with the help of decision making and iterative statements.

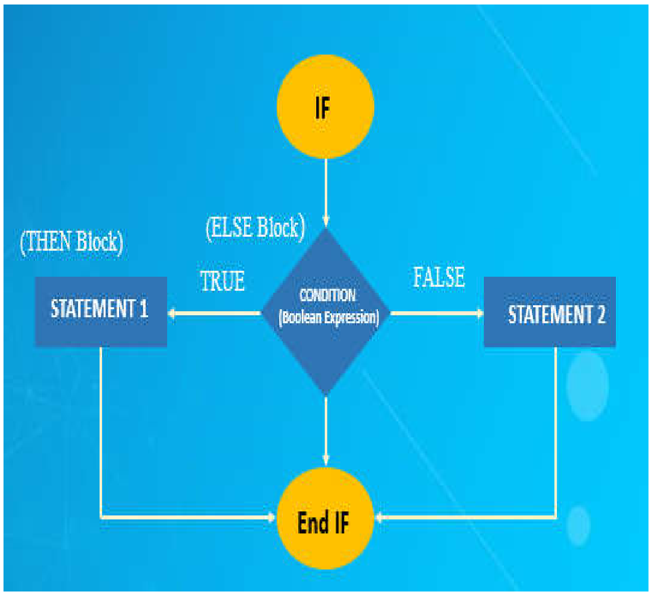
## 5.1 CONDITIONAL CONTROL

PL/SQL has conditional or selection statements for decision-making.

- IF....THEN....END IF.
- IF....THEN....ELSE....END IF.
- IF....THEN....ELSIF....END IF.
- CASE....END CASE.
- Searched CASE.

## IF....THEN....END IF

The IF...THEN....END IF statement is also known as a simple IF statement. A simple IF statement performs action statements if the result of the condition is TRUE. Otherwise the condition is FALSE, action statements not performed and the program continues with the next statement in the block.



**Syntax**

```
IF condition(s) then

Action statements

END IF;
```

Above Statement show a simple IF statement with an output statement which will be performed if the day is ‘SUNDAY’. The statement is skipped, if the day is not ‘SUNDAY’

**Example:**

```
        SQL> Declare
        V_ Day Varchar2(a) := ‘& Day’;
 Begin
        IF(V_ DAY =’SUNDAY’) then
DBMS_ OUTPUT PUT_ LINE( SUNDAY is  A HOLIDAY!);
End if;
End;
/
Enter value for day: SUNDAY
SUNDAY IS A HOLIDAY
```

**IF...THEN...ELSE...END IF**

The IF...THEN...ELSE...END IF statement is an extension of the simple IF statement. It provides action statements for the TRUE outcome as well as for the FALSE outcome.

**Syntax**

```
IF condition(s) then
 Action statements 1;
 Else
 Action statements 2;
End if;
```

If the condition TRUE, action statements 1 are performed. If the condition is FALSE, action statements 2 in is ELSE part are performed. One set of statements is skipped in any case. Figure show if the entered age is 18 or older, age is displayed with string ADULT, otherwise, age is displayed with string MINOR.

**Example:**

```
SQL> Set server output on
SQL> Declare
 2   V_ age number(2) := ‘& Age;
 3 Begin
 4 IF (V_ Age >=18) Then
 5 DBMS_ OUTPUT. PUT_LINE(‘Age:’|| V_ age||-Adult’);
 6 else
7 DBMS_OUTPUT. PUT_LINE( ‘Age: || V_age||. Minor’)
8 End if;
9 End;
10 /
Enter value for age: 21
Age : 21-Adult
 SQL>/
Enter value for age :12
Age :12-Minor
```

**IF...THEN...ELSIF...END IF**

This statement is the form of another if statement where the conditions can continue by multiple conditions in single if statement.

**Syntax**

```
IF condition(s)1 Then

Action statements 1

ElSIF condition(s)2 Then

Action statements

.........................

ELSE IF condition(s) N then

Action statement N

[ELSE

Else action statements]

End if;
```

the word ELSIF, which does not have the last E in ELSE. ELSIF is a single word, but END IF uses the words.

**Example:**

```
SQL> Declare

2 V_ pos  number (1) :=& position;

3 Begin

4 IF V_ Pos=1 then

5 DBMS_OUTPUT.PUT_LINE (‘20% increase’);

6 Elsif V_ Pos=2 then

7 DBMS_ OUTPUT.PUT_LINE (‘15% increase’);

8 Elsif V_Pos =3 then

9  DBMS_OUTPUT.PUT_LINE (‘10% increase’);

10 Elsif V_Pos=4 then

11 DBMS_OUTPUT.PUT_LINE (‘%% increase’);

12 Else

13 DBMS_OUTPUT.PUT_LINE (‘NO increase’);

14 End if;
```

```
15 End;

16 /

Enter value for position :2

15% increase
```

## CASE

In Previous chapter, the features case and searched case comes under expressions topic. Now we see the syntax and how to use in PL/SQL Block.

- The CASE Statement is an alternative to the IF...THEN...ELSIF...END IF statement.
- This statement starts with keyword CASE and ends with the keywords END CASE.
- The body of the statement contains WHEN clauses with values or conditions and action statements.
- When a WHEN condition evaluates to TRUE its actions statements are executed.

**Syntax**

```
CASE[ Variable- name]

WHEN condition1/value  THEN action- statement1;

WHEN condition1/value  2 THEN action- statement 2;

.......................................................

WHEN condition1/value  N THEN action- statement N;

ELSE action- statement;

END CASE
```

**Example:**

```
SQL> Declare /* Example of case */

2 V_ num number := & Any-num;

3 V_Res number;

4 Begin

5 V_Res := Mod ( V_ num ,2);

6 CASE V_Res

7 When 0 then DBMS_OUTPUT.PUT_LINE ( V_num||'is even');

8 ELSE DBMS_OUTPUT.PUT_LINE (V_ num|| 'is odd');
```

```
9 end case;

10 End;

11 /

Enter value for any- num : 5

5 is odd
```

**SEARCHED CASE**

A statement with a value is known as a CASE statement and a statement with condition is known as a searched CASE statement. This statement does not use variable- name as a selector but a CASE uses variable- name as a selector.

**Example:**

```
SQL>

Declare

2 V_ num number :=& Any- num;

3 Begin

4 case

5 When mod (V_ num2) =0 Then

6 DBMS_OUTPUT.PUT_LINE (V_ num|| ‘is odd’);

7 else

8 DBMS_OUTPUT.PUT_LINE (V_ num||’ is odd’);

9 End case;

10 End;

11 /

Enter value for any num :5

5 is odd.
```

**NESTED IF**

The nested IF statement contains an IF statement within another IF statement. If the condition of the outer IF statement is TRUE, then the corresponding IF statement is performed.

Consider the following conditions.

- Male 25 or over
- Male under 25

- Female 25 or over
- Female under 25.

**Example:**

```
SQL> Declare
2 V_ Gender char :='& sex';
3 V_age number (2) :='& Age';
4 V_char number(3,2);
5 Begin
6 IF (V_ Gender ='19') then /* Male*/
7 IF (V_age>=25) then
8 V_charge:=0.05;
9 Else
10 V_charge :=0.01;
11 End if;
12 Else /* Female */
13 IF (V_age>=25) then
14 V_charge :=0.06;
17 End if;
18 End if;
19 DBMS_OUTPUT. PUT_LINE (Gender :|| V_Gender);
20 DBMS_ OUTPUT.PUT_LINE ('Age:='To-char (V_age));
21 DBMS_OUTPUT.PUT_LINE ('SURCHARGE:'|| To-
char(V_charge));
22 End;
23 /
Enter value for sex:F
Enter value for age :18
Gender: F
Age: 18
Surcharge: 06
```

## 5.2 ITERATIVE CONTROL

In general, statements are executed sequentially: The first statement in a function is executed first, then followed by the second, next and so on. Some situation when you need to execute a block of code several times. For this execution programming languages provide control structures that allow for more complicated execution paths.

A loop statement used to execute a statement or group of statements multiple times. A loop repeats a statement or a series of statements a specific number of times as defined by the programmer.

Types of Looping Statements

- Basic loop
- WHILE loop
- FOR loop

Each loop has their own syntax and works differently.

**BASIC LOOP**

A basic loop is a loop that is performed repeatedly. Once a loop is entered all statements in the loop are executed. Once the bottom of the loop is reached control shift back to the top of the loop. The loop will continue infinitely is a logical error in programming. The only way to terminate a basic loop is by adding an EXIT statement inside the loop.

**Syntax**

```
Loop

Looping statement 1;

Looping statement 2;

............................

Looping statement  N;

EXIT [When condition];

End loop;
```

The EXIT statement in a loop could be independent statement. We can also add a condition with the optional WHEN clause that will end the loop when the condition becomes true.

**Example:**

```
EXIT WHEN V_count>10;
```

The condition is not checked at the top of the loop, but it is checked inside the body of loop. The loop is performed at least once, because the condition is tested after entering the body of the loop is known as Post_ test loop.

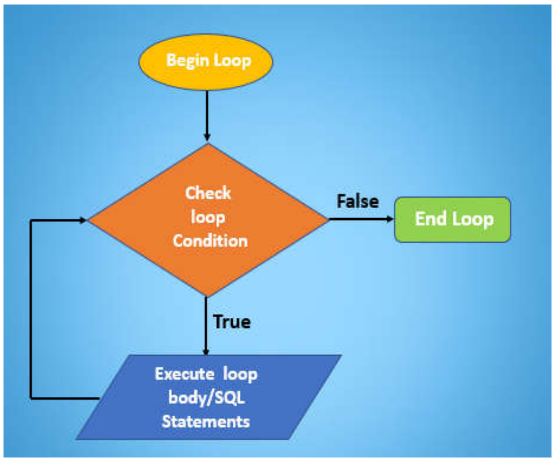
**Example:**

```
 SQL>Set Serveroutput on

SQL> Declare

2  V_ count         number(2);

3  V_ sum                   number(2):=0;

4  V_Avg                    number(3,1);

5  Begin

6  V_ count : =1;

7  Loop

8  V_sum := V_sum + V_count

9  V_count := V_count +1;

10  Exit when V_ count >10;

11 End loop;

12 V_Avg := V_sum 1( V_ count -1);

13  DBMS-OUTPUT.PUT-LINE (Average of 1 to 10 1&|| To- char (V_Avg));

14

15  End;

16  /

Average of 1 to 10 1& 5.5

 SQL>
```

**WHILE LOOP**

The WHILE loop is an alternative to the basic loop. It is performed as long as the condition is true. This terminates when the condition become false. If the condition is false in beginning, then the loop is not performed at all.

The WHILE loop does not need an EXIT statement to terminate.



**Syntax**

```
WHILE condition loop
 Looping statement 1;
 Looping statement 2;
 .................................
Looping statement n;
End loop;
```

**Example:**

```
SQL> Declare
2  V_ count         number(2);
3  V_ sum                    number(2):=0;
4  V_Avg                     number(3,1);
5  Begin
6  V_ count : =1;
7  While V_ count < = 10 Loop
8  V_sum := V_sum + V_count
9  V_count := V_count +1;
10  End loop;
11 V_Avg := V_sum 1( V_ count -1);
12  DBMS-OUTPUT.PUT_LINE (Average of 1 to 10 1&|| To- char
(V_Avg));
13  End;
14  /

Average of 1 to 10 1& 5.5
```

**Basic loop & While Loop**

| **Basic Loop** | **While Loop** |
|---|---|
| It is performed as long as the condition is false. | It is performed as long as the condition is true. |
| It testes the condition inside the loop ( Post-test loop). | It checks condition before entering the loop ( Pre-test Loop). |
| It is Performed at least one time. | It is performed zero or more times. |
| It needs the EXIT statement to terminate | no need for an Exit statement. |

**FOR LOOP**

The For loop is the simplest loop. We do not have to initialize, test and increment/ decrement the loop control variable separately. The counter used here is implicitly declared as an integer and it is destroyed on the loop's termination. It may be used within the loop body in an assignment statement as a target variable.

**Syntax**

FOR Counter W(Reservse) lower...upper loop

Looping statement 1

Looping statement 2

................................

Looping statement N

End loop;

**Counter Increment/ Decrement**

The counter varies from the lower value to the upper value incrementing by one with every loop execution. The counter starts with higher value and decrementing by one with every loop execution. To reverse the order the keyword Reverse is used to make higher to lower value.

SQL> Declare

2 V_ count number(2);

3 V_ sum number(2):=0;

```
4  V_Avg   number(3,1);
5  Begin
6  For V_ count in 1.. Loop
7  V_ sum :=V_ sum +V_ sum;
8  End loop;
9  V_ Avg := V_sum/10;
10  DBMS-OUTPUT.PUT-LINE (Average of 1 to 10 1&|| To- char (V_Avg));
11
12  End;
13  /
Average of 1 to 10 1& 5.5
```

**NESTED LOOP**

We can use a loop within another loop. Loop can be nested to many levels, when the inner loop ends it does it does not automatically end the outer loop enclosing it. We can quit the outer loop by label each loop inside the inner loop and then using the EXIT statement. The loop labels use the same naming rules as those used for identifies.

The label is enclosed using << and >> two pairs of angel brackets.

**Eg**

```
<< outer- loop>>
Loop
EXIT WHEN condition;
<< inner- loop>>
Loop
..........
EXIT outer- loop WHEN condition; /* exit outer-loop*/
EXIT WHEN condition         /* exit inner- loop*/
................................................................
End Loop inner- loop   /* label optional*/
...........................................
End loop outer-loop     /* label optional */
```

**Loop Control Statements**

Loop control statements change execution from its normal sequence.

PL/SQL supports the following control statements.

| S.No | Control Statement & Description |
|---|---|
| 1 | **EXIT** statement<br><br>The Exit statement completes the loop and control passes to the statement immediately after the END LOOP. |
| 2 | **CONTINUE** statement<br><br>Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating. |
| 3 | **GOTO** statement<br><br>Transfers control to the labelled statement. Though it is not advised to use the GOTO statement in your program. |

## 5.3 SEQUENTIAL CONTROL

**GOTO Statement**

The GOTO statement allows you to transfer control to a labeled block or statement.

The syntax of the GOTO statement:

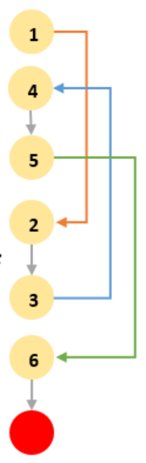
**GOTO label_name;**

The label_name is the name of a label that identifies the target statement. In the program, you surround the label name with double enclosing angle brackets as shown below:

**<<label_name>>;**

When PL/SQL executes a GOTO statement, it passes control to the first executable statement after the label.

```
BEGIN
   GOTO second_message;

  <first_message>
   DBMS_OUTPUT.PUT_LINE('Hello');
   GOTO the_end;

   <second_message>;
   DBMS_OUTPUT.PUT_LINE('PL/SQL GOTO Demo');
   GOTO first_message;

   <the_end>
   DBMS_OUTPUT.PUT_LINE('and good bye...');
END;
```



This code will execute like

- GOTO second_message statement is encountered firsr, therefore, the control is passed to the statement after the second_message label.
- GOTO first_message is encountered in second, so the control is transferred to the statement after the first_message label.
- Next , GOTO the_end is reached, hence the control is passed to the statement after the the_end label.

The output is:

```
PL/SQL GOTO Demo
Hello
and good Bye...
```

**NULL Statement**

The NULL statement is a NULL keyword followed by a semicolon ( ;). The NULL statement does nothing except that it passes control to the next statement.

The NULL statement is useful to:

- **Improve code readability**
- **Give target for a GOTO statement**
- **provide placeholders for subprograms**

**Improving code readability**

This code sends an email to employees whose job titles are SalesRepresentative.

```
IF jobtitle = 'SalesRepresentative' THEN
   send_email;
END IF;
```

If the employees job title are not SalesRepresentative then do nothing, this logic is not explicitly mentioned in the code.

An ELSE clause that consists of a NULL statement to clearly state that no action is needed for other employees.

```
IF jobtitle = 'SalesRepresentative' THEN
      send_email;
ELSE
      NULL;
END IF;
```

**Give target for a GOTO statement**

When using a GOTO statement, you need to specify a label followed by at least one executable statement.

This example has a GOTO statement to quickly move to the end of the program if no further processing is required:

```
DECLARE
  b_status BOOLEAN
BEGIN
  IF b_status THEN
        GOTO end_of_program;
  END IF;
  -- further processing here
  -- ...
  <<end_of_program>>
  NULL;
END;
```

Error will occur, if there is no NULL statement after the end_of_program label.

**Provide placeholders for subprograms**

The following example creates a procedure named apprreq - request_for_approval that doesn't have the code in the body. PL/SQL requires one executable statement in the body of the procedure to compile successfully. we add a NULL statement to the body as a placeholder. Later you can fill the real code.

```
CREATE PROCEDURE apprreq ( cusmer_id NUMBER )
AS
BEGIN
  NULL;
END;
```

Now, you have a good understanding of PL/SQL NULL statement and how to apply it in your daily programming tasks.

## Summary

PL/SQL Control Structures has three type: Condition Control, Iterative Control and Sequence Control.

In Condition Control if and case statements are used to make decisions and perform executions.

In Iterative Control, three types loops used with different syntaxes.

In Sequence Control GOTO and NULL Statements are performed.

## Review Questions

1. Write in detail about Conditional Control Structures.
2. Discuss the different Loop statements available in PL/SQL.
3. Explain the usage of sequence control statements with simple example.
4. How do you write a PL/SQL block for decision making purpose? Give Example.
5. How you come out of infinite Loop?

## Unit 2: Simple PL/SQL Programs

### 1. Add Two Numbers

```
Declare

Var1 integer;

Var2 integer;

Var3 integer;

Begin

Var1:=&var1;

Var2:=&var2;

Var3:=var1+var2;

Dbms_output.put_line(var3);

end;
```

### 2. Prime Number

```
Declare

n number;

i number;

flag number;

 begin
```

```
i:=2;
flag:=1;
n:=&n;
for i in 2..n/2 loop
  if mod(n,i)=0 then
flag:=0;
exit;
end if;
      end loop;
      if flag=1 then
dbms_output.put_line('prime');
else
dbms_output.put_line('not prime');
              end if;
          end;
```

**3. <u>Factorial Number</u>**

```
declare
n number;
fac number:=1;
i number;
         begin
n:=&n;
for i in 1..n
loop
fac:=fac*i;
end loop;
dbms_output.put_line('factorial='||fac);
          end;
```

## 4. <u>Print a Table of Number</u>

```
declare
n number;
i number;
begin
n:=&n;
for i in 1..10
loop
dbms_output.put_line(n||' x '||i||' = '||n*i);
end loop;
end;
```

## 5. <u>Reverse of a number</u>

```
declare
n number;
i number;
rev number:=0;
r number;
begin
n:=&n;
while n>0
loop
r:=mod(n,10);
rev:=(rev*10)+r;
n:=trunc(n/10);
end loop;
dbms_output.put_line('reverse is '||rev);
end;
```

## 6. Fibonacci Series

```
declare
first number:=0;
second number:=1;
third number;
n number:=&n;
i number;
begin
dbms_output.put_line('Fibonacci series is:');
dbms_output.put_line(first);
dbms_output.put_line(second);
for i in 2..n
loop
third:=first+second;
first:=second;
second:=third;
dbms_output.put_line(third);
end loop;
end;
```

## 7. Check number is odd or even

```
declare
n number:=&n;
begin
if mod(n,2)=0
then
dbms_output.put_line('number is even');
else
dbms_output.put_line('number is odd');
end if;
end;
```

**8. Palindrome Number**

```
declare
   n number;
   m number;
   rev number:=0;
   r number;
begin
   n:=12321;
   m:=n;
   while n>0
   loop
      r:=mod(n,10);
      rev:=(rev*10)+r;
      n:=trunc(n/10);
   end loop;
   if m=rev
   then
      dbms_output.put_line('number is palindrome');
   else
      dbms_output.put_line('number is not palindrome');
   end if;
end;
```

**9. Swap Two Numbers**

```
declare
   a number;
   b number;
   temp number;
begin
   a:=5;
   b:=10;
   dbms_output.put_line('before swapping:');
   dbms_output.put_line('a='||a||' b='||b);
   temp:=a;
   a:=b;
   b:=temp;
   dbms_output.put_line('after swapping:');
   dbms_output.put_line('a='||a||' b='||b);
end;
```

**10. <u>Greatest of three numbers</u>**

```
declare
a number:=10;
b number:=12;
c number:=5;
begin
dbms_output.put_line('a='||a||' b='||b||' c='||c);
if a>b AND a>c
then
dbms_output.put_line('a is greatest');
else
if b>a AND b>c
then
dbms_output.put_line('b is greatest');
else
dbms_output.put_line('c is greatest');
end if;
end if;
end;
/
```

**Reference Links**

1. **https://www.thecrazyprogrammer.com/plsql-programs-examples**
2. **https://www.tutorialspoint.com/plsql/plsql_basic_syntax.htm**
3. **http://www.euroinformatica.ro/pl-sql-overview/**

**Reference Books**

1. Database Systems using Oracle, Nilesh Shah, 2nd edition, PHI
2. Database Management Systems, Gerald V. Post, 3rd edition, TMH
3. Database Management Systems, Majumdar & Bhattacharya, 2007, TMH.
4. Fundamentals of RDBMS and Oracle 9i, T. Parimalam, S.N. Sathalakshmi, N. Moorthy,2012.

❖❖❖❖

6

# TRANSACTION MANAGEMENT

**Unit Structure**

**Objectives:**

- Will be able to explain the principle of transaction management design.
- Understand transactions and their properties (ACID) & the anomalies that occur without ACID.
- UNDERSTAND the locking protocols used to ensure Isolation & the logging techniques used to ensure Atomicity and Durability.
- Understand Recovery techniques used to recover from crashes.
- Explains the concurrency control and recovery algorithms.
- Applies transaction processing mechanisms in relational databases.

**Transaction Management**

A transaction is a set of logically related operations. For example, you are transferring money from your bank account to your friend's account, the set of operations would be like this: Simple Transaction Example

1. Read your account balance.
2. Deduct the amount from your balance.
3. Write the remaining balance to your account.
4. Read your friend's account balance.

5. Add the amount to his account balance.

6. Write the new updated balance to his account.

This whole set of operations can be called a transaction. Although I have shown you read, write and update operations in the above example but the transaction can have operations like read, write, insert, update, delete.

## 6.1 ACID PROPERTIES

A transaction is a very small unit of a program and it may contain several lowlevel tasks. A transaction in a database system must maintain Atomicity, Consistency, Isolation, and Durability − commonly known as ACID properties − in order to ensure accuracy, completeness, and data integrity.

**Atomicity** − This property states that a transaction must be treated as an atomic unit, that is, either all of its operations are executed or none. There must be no state in a database where a transaction is left partially completed. States should be defined either before the execution of the transaction or after the execution/abortion/failure of the transaction.

**Consistency** − The database must remain in a consistent state after any transaction. No transaction should have any adverse effect on the data residing in the database. If the database was in a consistent state before the execution of a transaction, it must remain consistent after the execution of the transaction as well.

**Durability** − The database should be durable enough to hold all its latest updates even if the system fails or restarts. If a transaction updates a chunk of data in a database and commits, then the database will hold the modified data. If a transaction commits but the system fails before the data could be written on to the disk, then that data will be updated once the system springs back into action.

**Isolation** − In a database system where more than one transaction are being executed simultaneously and in parallel, the property of isolation states that all the transactions will be carried out and executed as if it is the only transaction in the system. No transaction will affect the existence of any other transaction.

## 6.2 SERIALIZABILITY

When multiple transactions are being executed by the operating system in a multiprogramming environment, there are possibilities that instructions of one transactions are interleaved with some other transaction.

**Schedule** − A chronological execution sequence of a transaction is called a schedule. A schedule can have many transactions in it, each comprising of a number of instructions/tasks.

**Serial Schedule** − It is a schedule in which transactions are aligned in such a way that one transaction is executed first. When the first transaction completes its cycle, then the next transaction is executed. Transactions are ordered one after the other. This type of schedule is called a serial schedule, as transactions are executed in a serial manner.

In a multi-transaction environment, serial schedules are considered as a benchmark. The execution sequence of an instruction in a transaction cannot be changed, but two transactions can have their instructions executed in a random fashion. This execution does no harm if two transactions are mutually independent and working on different segments of data; but in case these two transactions are working on the same data, then the results may vary. This ever-varying result may bring the database to an inconsistent state.

To resolve this problem, we allow parallel execution of a transaction schedule, if its transactions are either serializable or have some equivalence relation among them.

**Distributed Two-phase Commit**

Distributed two-phase commit reduces the vulnerability of one-phase commit protocols. The steps performed in the two phases are as follows −

**Phase 1: Prepare Phase**

After each slave has locally completed its transaction, it sends a “DONE” message to the controlling site. When the controlling site has received “DONE” message from all slaves, it sends a “Prepare” message to the slaves.

The slaves vote on whether they still want to commit or not. If a slave wants to commit, it sends a “Ready” message.

A slave that does not want to commit sends a “Not Ready” message. This may happen when the slave has conflicting concurrent transactions or there is a timeout.

**Phase 2: Commit/Abort Phase**

After the controlling site has received “Ready” message from all the slaves

The controlling site sends a “Global Commit” message to the slaves.

The slaves apply the transaction and send a “Commit ACK” message to the controlling site.

When the controlling site receives "Commit ACK" message from all the slaves, it considers the transaction as committed.

After the controlling site has received the first "Not Ready" message from any slave.

The controlling site sends a "Global Abort" message to the slaves.

The slaves abort the transaction and send a "Abort ACK" message to the controlling site.

When the controlling site receives "Abort ACK" message from all the slaves, it considers the transaction as aborted.

## 6.3 CONCURRENCY CONTROL LOCK MANAGEMENT

Concurrency Control in Database Management System is a procedure of managing simultaneous operations without conflicting with each other. It ensures that Database transactions are performed concurrently and accurately to produce correct results without violating data integrity of the respective Database.

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, it would have a mix of READ and WRITE operations and hence the concurrency is a challenge.

DBMS Concurrency Control is used to address such conflicts, which mostly occur with a multi-user system. Therefore, Concurrency Control is the most important element for proper functioning of a Database Management System where two or more database transactions are executed simultaneously, which require access to the same data.

**Potential problems of Concurrency**

Here, are some issues which you will likely to face while using the DBMS 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.

**Reasons for using Concurrency control method is DBMS:**

To apply Isolation through mutual exclusion between conflicting transactions.

To resolve read-write and write-write conflict issues.

To preserve database consistency through constantly preserving execution obstructions.

The system needs to control the interaction among the concurrent transactions. This control is achieved using concurrent-control schemes.

Concurrency control helps to ensure serializability.

**Example**

Assume that two people who go to electronic kiosks at the same time to buy a movie ticket for the same movie and the same show time.

However, there is only one seat left in for the movie show in that particular theatre. Without concurrency control in DBMS, it is possible that both moviegoers will end up purchasing a ticket. However, concurrency control method does not allow this to happen. Both moviegoers can still access information written in the movie seating database. But concurrency control only provides a ticket to the buyer who has completed the transaction process first.

**Concurrency Control Protocols**

Different concurrency control protocols offer different benefits between the amount of concurrency they allow and the amount of overhead that they impose. Following are the Concurrency Control techniques in DBMS:

1. Lock-Based Protocols
2. Two Phase Locking Protocol
3. Timestamp-Based Protocols
4. Validation-Based Protocols

**Lock-based Protocols**

Lock Based Protocols in DBMS is a mechanism in which a transaction cannot Read or Write the data until it acquires an appropriate lock. Lock based protocols help to eliminate the concurrency problem in DBMS for simultaneous transactions by locking or isolating a particular transaction to a single user.

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 in DBMS 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 in DBMS based on their uses. If a lock is acquired on a data item to perform a write operation, it is called an exclusive lock.

**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.

**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.

**Simplistic Lock Protocol**

This type of lock-based protocols allows transactions to obtain a lock on every object before beginning operation. Transactions may unlock the data item after finishing the 'write' operation.

**Pre-claiming Locking**

Pre-claiming lock protocol helps to evaluate operations and create a list of required data items which are needed to initiate an execution process. In the situation when all locks are granted, the transaction executes. After that, all locks release when all of its operations are over.

### Starvation

Starvation is the situation when a transaction needs to wait for an indefinite period to acquire a lock.

Following are the reasons for Starvation:

1. When waiting scheme for locked items is not properly managed
2. In the case of resource leak
3. The same transaction is selected as a victim repeatedly

## Two Phase Locking Protocol

Two Phase Locking Protocol also known as 2PL protocol is a method of concurrency control in DBMS that ensures serializability by applying a lock to the transaction data which blocks other transactions to access the same data simultaneously. Two Phase Locking protocol helps to eliminate the concurrency problem in DBMS.

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.

In the above-given diagram, you can see that local and global deadlock detectors are searching for deadlocks and solve them with resuming transactions to their initial states.

### Strict Two-Phase Locking Method

Strict-Two phase locking system is almost similar to 2PL. The only difference is that Strict-2PL never releases a lock after using it. It holds all the locks until the commit point and releases all the locks at one go when the process is over.

**Centralized 2PL**

In Centralized 2 PL, a single site is responsible for lock management process. It has only one lock manager for the entire DBMS.

**Primary copy 2PL**

Primary copy 2PL mechanism, many lock managers are distributed to different sites. After that, a particular lock manager is responsible for managing the lock for a set of data items. When the primary copy has been updated, the change is propagated to the slaves.

**Distributed 2PL**

In this kind of two-phase locking mechanism, Lock managers are distributed to all sites. They are responsible for managing locks for data at that site. If no data is replicated, it is equivalent to primary copy 2PL. Communication costs of Distributed 2PL are quite higher than primary copy 2PL

## Timestamp-based Protocols

Timestamp based Protocol in DBMS is an algorithm which uses the System Time or Logical Counter as a timestamp to serialize the execution of concurrent transactions. The Timestamp-based protocol ensures that every conflicting read and write operations are executed in a timestamp order.

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.

**Advantages**:

Schedules are serializable just like 2PL protocols

No waiting for the transaction, which eliminates the possibility of deadlocks!

**Disadvantages:**

Starvation is possible if the same transaction is restarted and continually aborted

## Validation Based Protocol

Validation based Protocol in DBMS also known as Optimistic Concurrency Control Technique is a method to avoid concurrency in transactions. In this protocol, the local copies of the transaction data are updated rather than the data itself, which results in less interference while execution of the transaction.

The Validation based Protocol is performed in the following three phases:

1. Read Phase
2. Validation Phase
3. Write Phase

**Read Phase**

In the Read Phase, the data values from the database can be read by a transaction but the write operation or updates are only applied to the local data copies, not the actual database.

**Validation Phase**

In Validation Phase, the data is checked to ensure that there is no violation of serializability while applying the transaction updates to the database.

**Write Phase**

In the Write Phase, the updates are applied to the database if the validation is successful, else; the updates are not applied, and the transaction is rolled back.

**Characteristics of Good Concurrency Protocol**

An ideal concurrency control DBMS mechanism has the following objectives:

Must be resilient to site and communication failures.

It allows the parallel execution of transactions to achieve maximum concurrency.

Its storage mechanisms and computational methods should be modest to minimize overhead.

It must enforce some constraints on the structure of atomic actions of transactions.

## 6.4 LOST UPDATE PROBLEM

The lost update problem occurs when two concurrent transactions, T1 and T2, are updating the same data element and one of the updates is lost.

The lost update problem occurs when 2 concurrent transactions try to read and update the same data. Let's understand this with the help of an example.

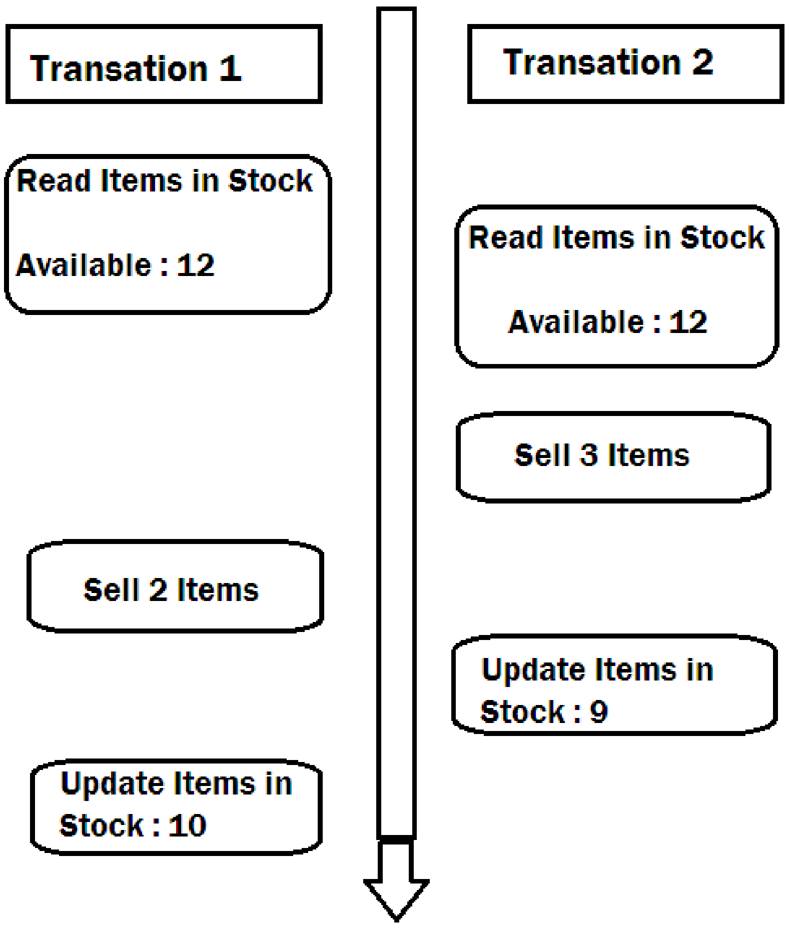
Suppose we have a table named "Product" that stores id, name, and ItemsinStock for a product. It is used as part of an online system that displays the number of items in stock for a particular product and so needs to be updated each time a sale of that product is made.

The table looks like this:

| Id | Name | ItemsinStock |
|---|---|---|
| 1 | Laptops | 12 |

Now consider a scenario where a user arrives and initiates the process of buying a laptop. This will initiate a transaction. Let's call this transaction, transaction 1.

At the same time another user logs into the system and initiates a transaction, let's call this transaction 2. Take a look at the following figure.



Transaction 1 reads the items in stock for laptops which is 12. A little later transaction 2 reads the value for ItemsinStock for laptops which will still be 12 at this point of time. Transaction 2 then sells three laptops, shortly before transaction 1 sells 2 items.

Transaction 2 will then complete its execution first and update ItemsinStock to 9 since it sold three of the 12 laptops. Transaction 1 commits itself. Since transaction 1 sold two items, it updates ItemsinStock to 10.

This is incorrect, the correct figure is 12-3-2 = 7

**Working Example** of Lost Update Problem

Let's us take a look at the lost update problem in action in SQL Server. As always, first, we will create a table and add some dummy data into it.

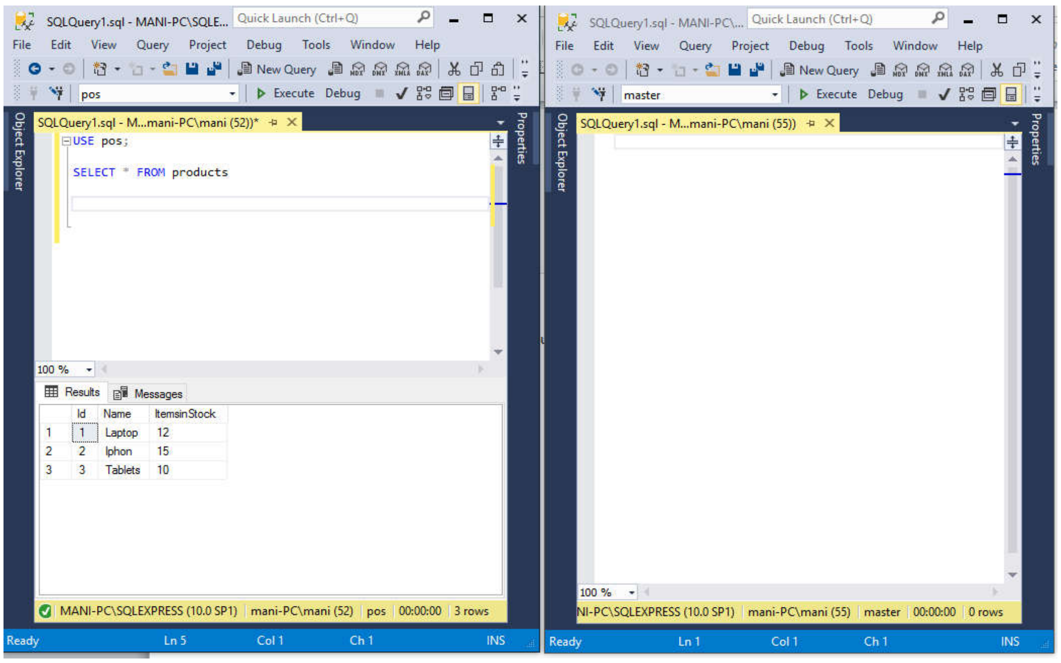
As always, be sure that you are properly backed up before playing with new code. If you're not sure, see this article on SQL Server backup.

Execute the following script on your database server.

<span style="font-size: 14px;">CREATE DATABASE pos;

```
USE pos;

CREATE TABLE products

(Id INT PRIMARY KEY,

        Name VARCHAR(50) NOT NULL,

        ItemsinStock INT NOT NULL )

INSERT into products

VALUES

(1, 'Laptop', 12),

(2, 'Iphon', 15),

(3, 'Tablets', 10)</span>
```

Now, open two SQL server management studio instances side by side. We will run one transaction in each of these instances.


SQLQuery1.sql - MANI-PC\SQLE...
Quick Launch (Ctrl+Q)
File Edit View Query Project Debug Tools Window Help
New Query
pos
Execute Debug
SQLQuery1.sql - M...mani-PC\mani (52))*
Object Explorer
Properties
100 %
Results
Messages
Ready Ln 5 Col 1 Ch 1 INS
SQLQuery1.sql - MANI-PC\...
master
SQLQuery1.sql - M...mani-PC\mani (55))
Ready Ln 1 Col 1 Ch 1 INS

```
USE pos;

SELECT * FROM products
```

| | Id | Name | ItemsinStock |
|---|---|---|---|
| 1 | 1 | Laptop | 12 |
| 2 | 2 | Iphon | 15 |
| 3 | 3 | Tablets | 10 |

MANI-PC\SQLEXPRESS (10.0 SP1) | mani-PC\mani (52) | pos | 00:00:00 | 3 rows

NI-PC\SQLEXPRESS (10.0 SP1) | mani-PC\mani (55) | master | 00:00:00 | 0 rows

Add the following script to the first instance of SSMS.

```
<span style="font-size: 14px;">USE pos;

-- Transaction 1

BEGIN TRAN

DECLARE @ItemsInStock INT

SELECT @ItemsInStock = ItemsInStock

FROM products WHERE Id = 1
```

```
WaitFor Delay '00:00:12'

SET @ItemsInStock = @ItemsInStock - 2

UPDATE products SET ItemsinStock = @ItemsInStock

WHERE Id = 1

Print @ItemsInStock

Commit Transaction</span>
```

This is the script for transaction 1. Here we begin the transaction and declare an integer type variable "@ItemsInStock". The value of this variable is set to the value of the ItemsinStock column for the record with Id 1 from the products table. Then a delay of 12 seconds is added so that transaction 2 can complete its execution before transaction 1. After the delay, the value of @ItemsInStock variable is decremented by 2 signifying the sale of 2 products.

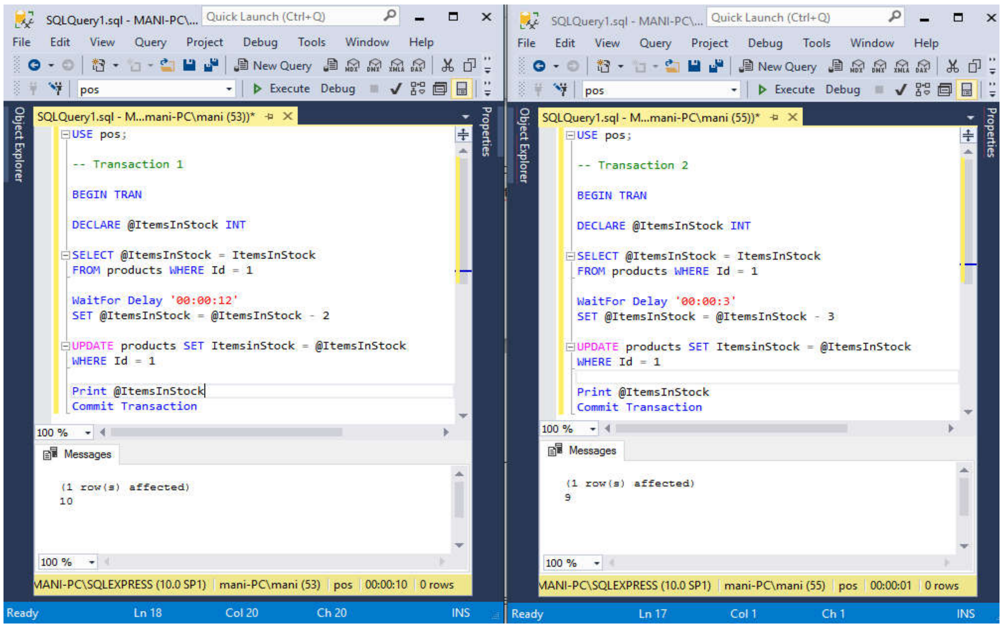
Finally, the value for ItemsinStock column for the record with Id 1 is updated with the value of @ItemsInStock variable. We then print the value of @ItemsInStock variable on the screen and commit the transaction.

In the second instance of SSMS, we add the script for transaction 2 which is as follows:

```
<span style="font-size: 14px;">USE pos;

-- Transaction 2

BEGIN TRAN

DECLARE @ItemsInStock INT

SELECT @ItemsInStock = ItemsInStock

FROM products WHERE Id = 1

WaitFor Delay '00:00:3'

SET @ItemsInStock = @ItemsInStock - 3

UPDATE products SET ItemsinStock = @ItemsInStock

WHERE Id = 1

Print @ItemsInStock

Commit Transaction</span>
```

The script for transaction 2 is similar to transaction 1. However, here in transaction 2, the delay is only for three seconds and the decrement in the value for @ItemsInStock variable is three, as it is a sale of three items.

Now, run transaction 1 and then transaction 2. You will see transaction 2 completing its execution first. And the value printed for @ItemsInStock variable will be 9. After some time transaction 1 will also complete its execution and the value printed for its @ItemsInStock variable will be 10.


SQLQuery1.sql - MANI-PC\...
Quick Launch (Ctrl+Q)
File Edit View Query Project Debug Tools Window Help
New Query
pos
Execute Debug
Object Explorer
Properties
SQLQuery1.sql - M...mani-PC\mani (53))*
SQLQuery1.sql - M...mani-PC\mani (55))*
100 %
Messages
MANI-PC\SQLEXPRESS (10.0 SP1) | mani-PC\mani (53) | pos | 00:00:10 | 0 rows
MANI-PC\SQLEXPRESS (10.0 SP1) | mani-PC\mani (55) | pos | 00:00:01 | 0 rows
Ready Ln 18 Col 20 Ch 20 INS
Ready Ln 17 Col 1 Ch 1 INS

```
USE pos;

-- Transaction 1

BEGIN TRAN

DECLARE @ItemsInStock INT

SELECT @ItemsInStock = ItemsInStock
FROM products WHERE Id = 1

WaitFor Delay '00:00:12'
SET @ItemsInStock = @ItemsInStock - 2

UPDATE products SET ItemsinStock = @ItemsInStock
WHERE Id = 1

Print @ItemsInStock
Commit Transaction
```

```
(1 row(s) affected)
10
```

```
USE pos;

-- Transaction 2

BEGIN TRAN

DECLARE @ItemsInStock INT

SELECT @ItemsInStock = ItemsInStock
FROM products WHERE Id = 1

WaitFor Delay '00:00:3'
SET @ItemsInStock = @ItemsInStock - 3

UPDATE products SET ItemsinStock = @ItemsInStock
WHERE Id = 1

Print @ItemsInStock
Commit Transaction
```

```
(1 row(s) affected)
9
```

Both of these values are wrong, the actual value for ItemsInStock column for the product with Id 1 should be 7.

It is important to note here that the lost update problem only occurs with read committed and read uncommitted transaction isolation levels. With all the other transaction isolation levels, this problem does not occur.

**Read Repeatable Transaction Isolation Level**

Let's update the isolation level for both the transactions to read repeatable and see if the lost update problem occurs. But before that, execute the following statement to update the value for ItemsInStock back to 12.

Update products SET ItemsinStock = 12

**Script For Transaction 1**

<span style="font-size: 14px;">USE pos;

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ

-- Transaction 1

BEGIN TRAN

DECLARE @ItemsInStock INT

SELECT @ItemsInStock = ItemsInStock

```
FROM products WHERE Id = 1

WaitFor Delay '00:00:12'

SET @ItemsInStock = @ItemsInStock - 2

UPDATE products SET ItemsinStock = @ItemsInStock

WHERE Id = 1

Print @ItemsInStock

Commit Transaction</span>
```

**Script For Transaction 2**

```
<span style="font-size: 14px;">USE pos;

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ

-- Transaction 2

BEGIN TRAN

DECLARE @ItemsInStock INT

SELECT @ItemsInStock = ItemsInStock

FROM products WHERE Id = 1

WaitFor Delay '00:00:3'

SET @ItemsInStock = @ItemsInStock - 3

UPDATE products SET ItemsinStock = @ItemsInStock

WHERE Id = 1

Print @ItemsInStock

Commit Transaction</span>
```

Here in both the transactions, we have set the isolation level to repeatable read.

Now run transaction 1 and then immediately run transaction 2. Unlike the previous case, transaction 2 will have to wait for transaction 1 to commit itself. After that the following error occurs for transaction 2:

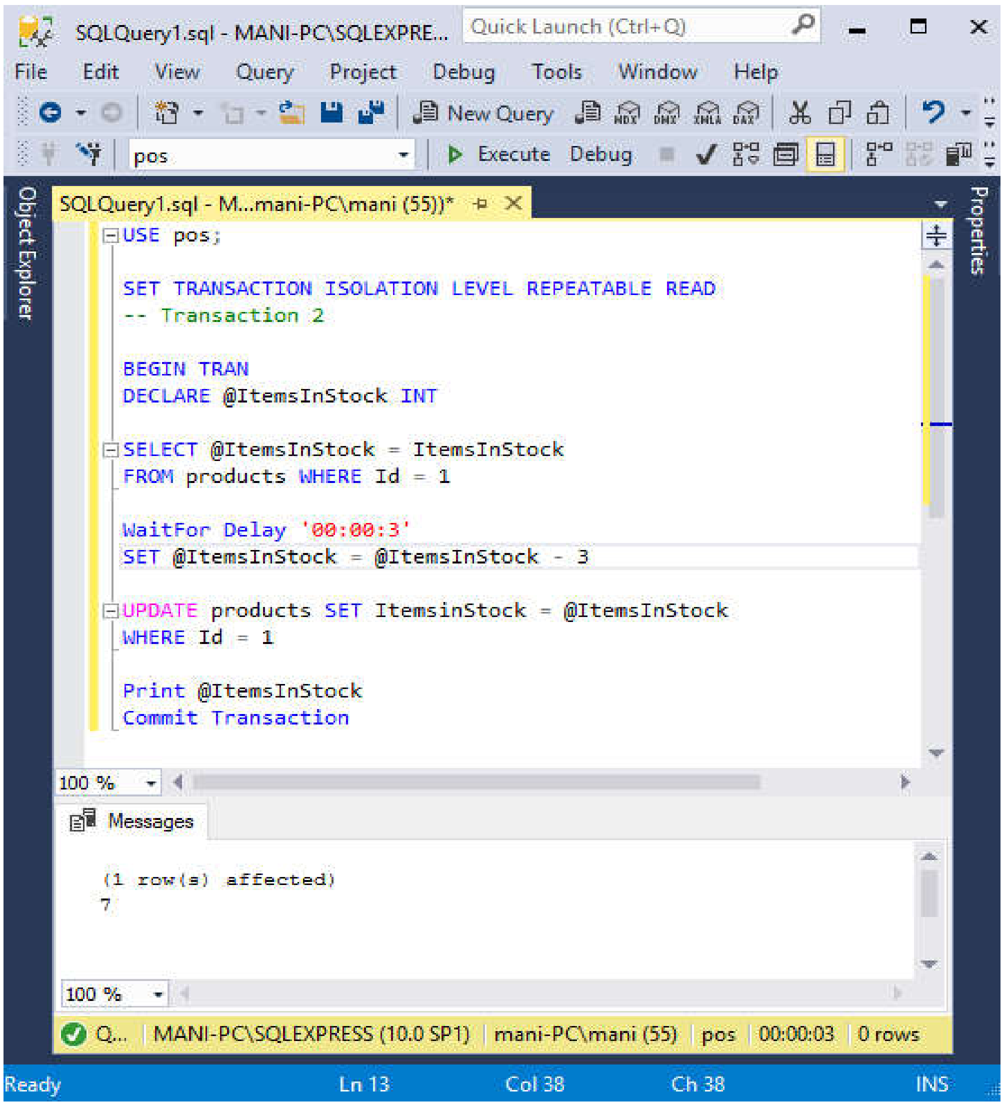
```
Msg 1205, Level 13, State 51, Line 15
```

```
Transaction (Process ID 55) was deadlocked on lock resources with another process and has been chosen as the deadlock victim. Rerun the transaction.
```

This error occurs because repeatable read locks the resource which is being read or updated by transaction 1 and it creates a deadlock on the other transaction that tries to access the same resource.

The error says that transaction 2 has a deadlock on a resource with another process and that this transaction has been blocked by the deadlock. This means that the other transaction was given access to the resource while this transaction was blocked and not given access to the resource.

It also says to rerun the transaction as the resource is free now. Now, if you run transaction 2 again, you will see the correct value of items in stock i.e. 7. This is because transaction 1 had already decremented the IteminStock value by 2, transaction 2 further decrements this by 3, therefore 12 – (2+3) = 7.



```
USE pos;

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
-- Transaction 2

BEGIN TRAN
DECLARE @ItemsInStock INT

SELECT @ItemsInStock = ItemsInStock
FROM products WHERE Id = 1

WaitFor Delay '00:00:3'
SET @ItemsInStock = @ItemsInStock - 3

UPDATE products SET ItemsinStock = @ItemsInStock
WHERE Id = 1

Print @ItemsInStock
Commit Transaction
```

```
(1 row(s) affected)
7
```

SQLQuery1.sql - MANI-PC\SQLEXPRE...
Quick Launch (Ctrl+Q)
File Edit View Query Project Debug Tools Window Help
New Query
pos
Execute Debug
SQLQuery1.sql - M...mani-PC\mani (55))*
Object Explorer
Properties
100 %
Messages
Q... MANI-PC\SQLEXPRESS (10.0 SP1) mani-PC\mani (55) pos 00:00:03 0 rows
Ready Ln 13 Col 38 Ch 38 INS

## 6.5 INCONSISTENT READ PROBLEM

The problem is that the transaction might read some data before they are changed and other data after they are changed, this cause Inconsistent Retrievals

Unrepeatable read (or inconsistent retrievals) occurs when a transaction calculates some summary (aggregate) function over a set of data while other transactions are updating the data.

The problem is that the transaction might read some data before they are changed and other data after they are changed, thereby yielding inconsistent results.

In an unrepeatable read, the transaction T1 reads a record and then does some other processing during which the transaction T2 updates the record. Now, if T1 rereads the record, the new value will be inconsistent with the previous value.

**Example**:

Consider the situation given in figure that shows two transactions operating on three accounts :

| Account-1 | Account-2 | Account-3 |
|---|---|---|
| Balance = 200 | Balance = 250 | Balance = 150 |

| **Transaction- A** | **Time** | **Transaction- B** |
|---|---|---|
| ----- | t0 | ---- |
| Read Balance of Acc-1 sum <-- 200 Read Balance of Acc-2 | t1 | ---- |
| Sum <-- Sum + 250 = 450 | t2 | ---- |
| ---- | t3 | Read Balance of Acc-3 |
| ---- | t4 | Update Balance of Acc-3 150 --> 150 - 50 --> 100 |
| ---- | t5 | Read Balance of Acc-1 |
| ---- | t6 | Update Balance of Acc-1 200 --> 200 + 50 --> 250 |
| ---- Read Balance of Acc-3 | t7 | COMMIT |
| Sum <-- Sum + 250 = 450 | t8 | ---- |

Transaction-A is summing all balances;while, Transaction-B is transferring an amount 50 from Account-3 to Account-1.

Here,the result produced by Transaction-A is 550,which is incorrect. if this result is written in database, database will be in inconsistent state, as actual sum is 600.

Here,Transaction-A has seen an inconsistent state of database, and has performed inconsistent analysis.

## 6.6 READ-WRITE LOCKS

**Read Locks:**

1. Multiple read locks can be acquired by multiple threads at the same time.
2. When a thread has a read lock on a row/table, no thread can update/insert/delete data from that table. (Even if the thread trying to write data doesn't require a write lock.)
3. A row/table cannot have a read and a write lock at the same time.

**Write Locks:**

1. When a row/table has a write lock, it cannot be read by another thread if they have a read lock implemented in them but can be read by other threads if no read lock is implemented (i.e simple Select query).

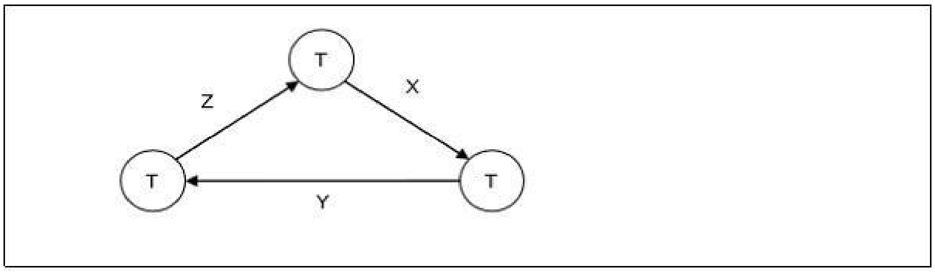
## 6.7 DEADLOCKS HANDLING

Deadlock refers to a specific situation where two or more processes are waiting for each other to release a resource or more than two processes are waiting for the resource in a circular chain.

Deadlock is a state of a database system having two or more transactions, when each transaction is waiting for a data item that is being locked by some other transaction. A deadlock can be indicated by a cycle in the wait-for-graph. This is a directed graph in which the vertices denote transactions and the edges denote waits for data items.

**For example**, in the following wait-for-graph, transaction T1 is waiting for data item X which is locked by T3. T3 is waiting for Y which is locked by T2 and T2 is waiting for Z which is locked by T1. Hence, a waiting cycle is formed, and none of the transactions can proceed executing.

.



**Deadlock Handling in Centralized Systems**

There are three classical approaches for deadlock handling, namely −

- Deadlock prevention.
- Deadlock avoidance.
- Deadlock detection and removal.

All of the three approaches can be incorporated in both a centralized and a distributed database system.

**Deadlock Prevention**

The deadlock prevention approach does not allow any transaction to acquire locks that will lead to deadlocks. The convention is that when more than one transactions request for locking the same data item, only one of them is granted the lock.

One of the most popular deadlock prevention methods is pre-acquisition of all the locks. In this method, a transaction acquires all the locks before starting to execute and retains the locks for the entire duration of transaction. If another transaction needs any of the already acquired locks, it has to wait until all the locks it needs are available. Using this approach, the system is prevented from being deadlocked since none of the waiting transactions are holding any lock.

**Deadlock Avoidance**

The deadlock avoidance approach handles deadlocks before they occur. It analyzes the transactions and the locks to determine whether or not waiting leads to a deadlock.

The method can be briefly stated as follows. Transactions start executing and request data items that they need to lock. The lock manager checks whether the lock is available. If it is available, the lock manager allocates the data item and the transaction acquires the lock. However, if the item is locked by some other transaction in incompatible mode, the lock manager runs an algorithm to test whether keeping the transaction in waiting state will cause a deadlock or not. Accordingly, the algorithm decides whether the transaction can wait or one of the transactions should be aborted.

There are two algorithms for this purpose, namely wait-die and wound-wait. Let us assume that there are two transactions, T1 and T2, where T1 tries to lock a data item which is already locked by T2. The algorithms are as follows −

**Wait-Die** − If T1 is older than T2, T1 is allowed to wait. Otherwise, if T1 is younger than T2, T1 is aborted and later restarted.

**Wound-Wait** − If T1 is older than T2, T2 is aborted and later restarted. Otherwise, if T1 is younger than T2, T1 is allowed to wait.

**Deadlock Detection and Removal**

The deadlock detection and removal approach runs a deadlock detection algorithm periodically and removes deadlock in case there is one. It does not check for deadlock when a transaction places a request for a lock. When a transaction requests a lock, the lock manager checks whether it is available. If it is available, the transaction is allowed to lock the data item; otherwise the transaction is allowed to wait.

Since there are no precautions while granting lock requests, some of the transactions may be deadlocked. To detect deadlocks, the lock manager periodically checks if the wait-forgraph has cycles. If the system is deadlocked, the lock manager chooses a victim transaction from each cycle. The victim is aborted and rolled back; and then restarted later. Some of the methods used for victim selection are −

Choose the youngest transaction.

Choose the transaction with fewest data items.

Choose the transaction that has performed least number of updates.

Choose the transaction having least restart overhead.

Choose the transaction which is common to two or more cycles.

This approach is primarily suited for systems having transactions low and where fast response to lock requests is needed.

**Deadlock Handling in Distributed Systems**

Transaction processing in a distributed database system is also distributed, i.e. the same transaction may be processing at more than one site. The two main deadlock handling concerns in a distributed database system that are not present in a centralized system are transaction location and transaction control. Once these concerns are addressed, deadlocks are handled through any of deadlock prevention, deadlock avoidance or deadlock detection and removal.

### Transaction Location

Transactions in a distributed database system are processed in multiple sites and use data items in multiple sites. The amount of data processing is not uniformly distributed among these sites. The time period of processing also varies. Thus the same transaction may be active at some sites and inactive at others. When two conflicting transactions are located in a site, it may happen that one of them is in inactive state. This condition does not arise in a centralized system. This concern is called transaction location issue.

This concern may be addressed by Daisy Chain model. In this model, a transaction carries certain details when it moves from one site to another. Some of the details are the list of tables required, the list of sites required, the list of visited tables and sites, the list of tables and sites that are yet to be visited and the list of acquired locks with types. After a transaction terminates by either commit or abort, the information should be sent to all the concerned sites.

### Transaction Control

Transaction control is concerned with designating and controlling the sites required for processing a transaction in a distributed database system. There are many options regarding the choice of where to process the transaction and how to designate the center of control, like −

One server may be selected as the center of control.

The center of control may travel from one server to another.

The responsibility of controlling may be shared by a number of servers.

### Distributed Deadlock Prevention

Just like in centralized deadlock prevention, in distributed deadlock prevention approach, a transaction should acquire all the locks before starting to execute. This prevents deadlocks.

The site where the transaction enters is designated as the controlling site. The controlling site sends messages to the sites where the data items are located to lock the items. Then it waits for confirmation. When all the sites have confirmed that they have locked the data items, transaction starts. If any site or communication link fails, the transaction has to wait until they have been repaired.

Though the implementation is simple, this approach has some drawbacks −

Pre-acquisition of locks requires a long time for communication delays. This increases the time required for transaction.

In case of site or link failure, a transaction has to wait for a long time so that the sites recover. Meanwhile, in the running sites, the items are locked. This may prevent other transactions from executing.

If the controlling site fails, it cannot communicate with the other sites. These sites continue to keep the locked data items in their locked state, thus resulting in blocking.

**Distributed Deadlock Avoidance**

As in centralized system, distributed deadlock avoidance handles deadlock prior to occurrence. Additionally, in distributed systems, transaction location and transaction control issues needs to be addressed. Due to the distributed nature of the transaction, the following conflicts may occur −

Conflict between two transactions in the same site.

Conflict between two transactions in different sites.

In case of conflict, one of the transactions may be aborted or allowed to wait as per distributed wait-die or distributed wound-wait algorithms.

Let us assume that there are two transactions, T1 and T2. T1 arrives at Site P and tries to lock a data item which is already locked by T2 at that site. Hence, there is a conflict at Site P. The algorithms are as follows −

.**Distributed Wound-Die**

➢ If T1 is older than T2, T1 is allowed to wait. T1 can resume execution after Site P receives a message that T2 has either committed or aborted successfully at all sites.

➢ If T1 is younger than T2, T1 is aborted. The concurrency control at Site P sends a message to all sites where T1 has visited to abort T1. The controlling site notifies the user when T1 has been successfully aborted in all the sites.

**Distributed Wait-Wait**

➢ If T1 is older than T2, T2 needs to be aborted. If T2 is active at Site P, Site P aborts and rolls back T2 and then broadcasts this message to other relevant sites. If T2 has left Site P but is active at Site Q, Site P broadcasts that T2 has been aborted; Site L then aborts and rolls back T2 and sends this message to all sites.

➢ If T1 is younger than T1, T1 is allowed to wait. T1 can resume execution after Site P receives a message that T2 has completed processing.

**Distributed Deadlock Detection**

Just like centralized deadlock detection approach, deadlocks are allowed to occur and are removed if detected. The system does not perform any checks when a transaction places a lock request. For implementation, global wait-for-graphs are created. Existence of a cycle in the global wait-for-graph indicates deadlocks. However, it is difficult to spot deadlocks since transaction waits for resources across the network.

Alternatively, deadlock detection algorithms can use timers. Each transaction is associated with a timer which is set to a time period in which a transaction is expected to finish. If a transaction does not finish within this time period, the timer goes off, indicating a possible deadlock.

Another tool used for deadlock handling is a deadlock detector. In a centralized system, there is one deadlock detector. In a distributed system, there can be more than one deadlock detectors. A deadlock detector can find deadlocks for the sites under its control. There are three alternatives for deadlock detection in a distributed system, namely.

**Centralized Deadlock Detector** − One site is designated as the central deadlock detector.

**Hierarchical Deadlock Detector** − A number of deadlock detectors are arranged in hierarchy.

**Distributed Deadlock Detector** − All the sites participate in detecting deadlocks and removing them.

## 6.8 TWO-PHASE LOCKING (2PL)

The two-phase locking protocol divides the execution phase of the transaction into three parts. In the **first part**, when the execution of the transaction starts, it seeks permission for the lock it requires. In the **second part**, the transaction acquires all the locks. The **third phase** is started as soon as the transaction releases its first lock. In the third phase, the transaction cannot demand any new locks. It only releases the acquired locks.

**There are two phases of 2PL:**

**Growing phase**: In the growing phase, a new lock on the data item may be acquired by the transaction, but none can be released.

**Shrinking phase**: In the shrinking phase, existing lock held by the transaction may be released, but no new locks can be acquired.

In the below example, if lock conversion is allowed then the following phase can happen:

Upgrading of lock (from S(a) to X (a)) is allowed in growing phase.

Downgrading of lock (from X(a) to S(a)) must be done in shrinking phase.

**Example:**

| | **T1** | **T2** |
|---|---|---|
| 0 | LOCK – S(A) | |
| 1 | | LOCK – S(A) |
| 2 | LOCK – X(B) | |
| 3 | -- | -- |
| 4 | UNLOCK(A) | |
| 5 | | LOCK –X(C) |
| 6 | UNLOCK(B) | |
| 7 | | UNLOCK(A) |
| 8 | | UNLOCK(C) |
| 9 | -- | -- |

The following way shows how unlocking and locking work with 2-PL.

**Transaction T1:**

Growing phase: from step 1-3

Shrinking phase: from step 5-7

Lock point: at 3

**Transaction T2:**

Growing phase: from step 2-6

Shrinking phase: from step 8-9

Lock point: at 6

Two-Phase Locking (2PL) is a concurrency control method which divides the execution phase of a transaction into three parts. It ensures conflict serializable schedules. If read and write operations introduce the first unlock operation in the transaction, then it is said to be Two-Phase Locking Protocol.

This protocol can be divided into two phases,

1. In Growing Phase, a transaction obtains locks, but may not release any lock.

2. In Shrinking Phase, a transaction may release locks, but may not obtain any lock.

Two-Phase Locking does not ensure freedom from deadlocks.

Types of Two – Phase Locking Protocol

Following are the types of two – phase locking protocol:

1. Strict Two – Phase Locking Protocol
2. Rigorous Two – Phase Locking Protocol
3. Conservative Two – Phase Locking Protocol

**1. Strict Two-Phase Locking Protocol**

- Strict Two-Phase Locking Protocol avoids cascaded rollbacks.
- This protocol not only requires two-phase locking but also all exclusive-locks should be held until the transaction commits or aborts.
- It is not deadlock free.
- It ensures that if data is being modified by one transaction, then other transaction cannot read it until first transaction commits.
- Most of the database systems implement rigorous two – phase locking protocol.

**2. Rigorous Two-Phase Locking**

- Rigorous Two – Phase Locking Protocol avoids cascading rollbacks.
- This protocol requires that all the share and exclusive locks to be held until the transaction commits.

**3. Conservative Two-Phase Locking Protocol**

- Conservative Two – Phase Locking Protocol is also called as Static Two – Phase Locking Protocol.
- This protocol is almost free from deadlocks as all required items are listed in advanced.
- It requires locking of all data items to access before the transaction starts.

❖❖❖❖

7

# DCL STATEMENTS

**Unit Structure**

## 1 INTRODUCTION TO DCL

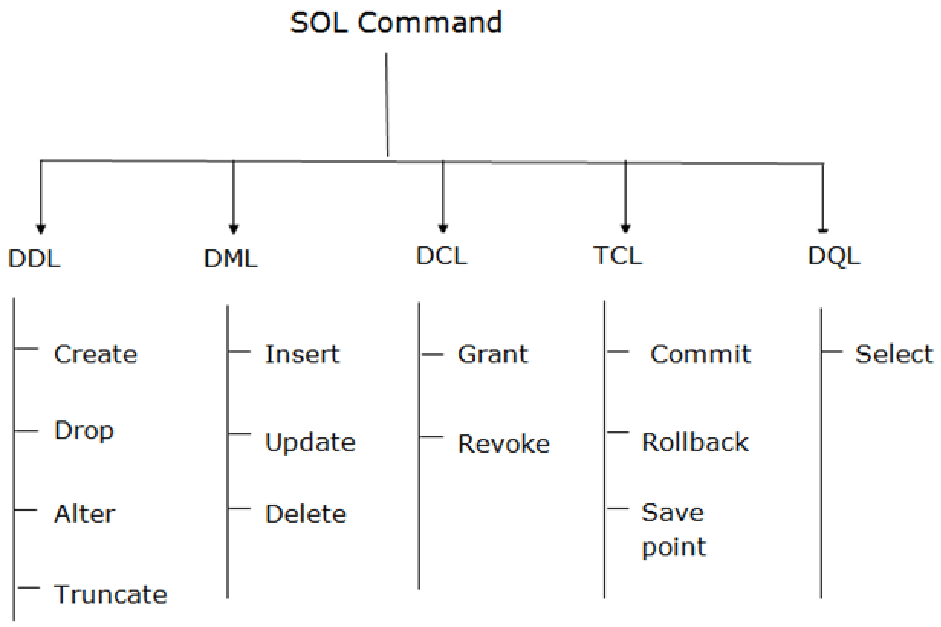
SQL Commands

SQL orders are directions. It is utilized to speak with the data set. It is additionally used to perform explicit assignments, capacities, and questions of information.

SQL can perform different undertakings like make a table, add information to tables, drop the table, change the table, set authorization for clients.

Types of SQL Commands

There are five types of SQL commands: DDL, DML, DCL, TCL, and DQL.



data control language (DCL) is utilized to get to the put away information. It is chiefly utilized for repudiate and to allow the client the necessary admittance to an information base. In the data set, this language doesn't have the element of rollback.

- It is a part of the structured query language (SQL).
- It helps in controlling access to information stored in a database. It complements the data manipulation language (DML) and the data definition language (DDL).
- It is the simplest among three commands.
- It provides the administrators, to remove and set database permissions to desired users as needed.
- These commands are employed to grant, remove and deny permissions to users for retrieving and manipulating a database.
- Grant
- Revoke

A data control language (DCL) is a syntax similar to a computer programming language used to control access to data stored in a database (Authorization). In particular, it is a component of Structured Query Language (SQL). Data Control Language is one of the logical group in SQL Commands. SQL is the standard language for relational database management systems. SQL statements are used to perform tasks such as insert data to a database, delete or update data in a database, or retrieve data from a database.

Though database systems use SQL, they also have their own additional proprietary extensions that are usually only used on their system. For Example Microsoft SQL server uses Transact-SQL (T-SQL) which is an extension of SQL. Similarly Oracle uses PL-SQL which is their proprietary extension for them only. However, the standard SQL commands such as "Select", "Insert", "Update", "Delete", "Create", and "Drop" can be used to accomplish almost everything that one needs to do with a database.

Examples of DCL commands include:

GRANT to allow specified users to perform specified tasks.

REVOKE to remove the user accessibility to database object.

The operations for which privileges may be granted to or revoked from a user or role apply to both the Data definition language (DDL) and the Data manipulation language (DML), and may include CONNECT, SELECT, INSERT, UPDATE, DELETE, EXECUTE, and USAGE.

Data control language (DCL) is used to access the stored data. It is mainly used for revoke and to grant the user the required access to a database. In the database, this language does not have the feature of rollback.

- It is a part of the structured query language (SQL).
- It helps in controlling access to information stored in a database.
- It complements the data manipulation language (DML) and the data definition language (DDL).
- It is the simplest among three commands.
- It provides the administrators, to remove and set database permissions to desired users as needed.

These commands are employed to grant, remove and deny permissions to users for retrieving and manipulating a database.

DCL stands for Data Control Language. DCL is used to control user access in a database. This command is related to the security issues.

Using DCL command, it allows or restricts the user from accessing data in database schema.

DCL commands are as follows,

1. GRANT
2. REVOKE

It is used to grant or revoke access permissions from any database user.

**2. GRANT COMMAND**

SQL Grant command is specifically used to provide privileges to database objects for a user. This command also allows users to grant permissions to other users too.

**Syntax:**

grant privilege_name on object_name

to {user_name | public | role_name}

Here privilege_name is which permission has to be granted, object_name is the name of the database object, user_name is the user to which access should be provided, the public is used to permit access to all the users.

## 3 REVOKE :

Revoke command withdraw user privileges on database objects if any granted. It does operations opposite to the Grant command. When a privilege is revoked from a particular user U, then the privileges granted to all other users by user U will be revoked.

**Syntax:**

revoke privilege_name on object_name

from {user_name | public | role_name}

**Example:**

grant insert,

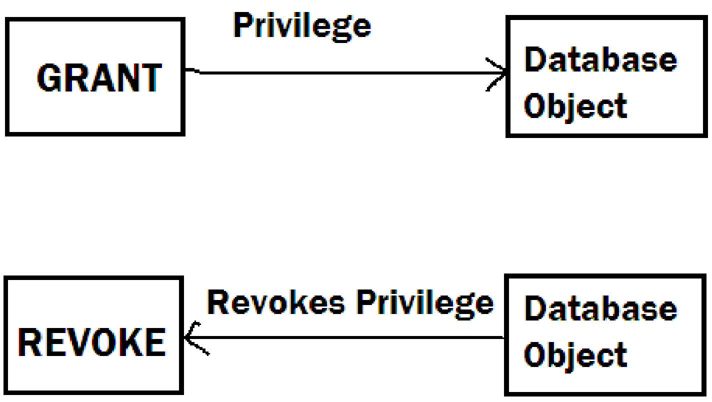
select on accounts to Ram

By the above command user ram has granted permissions on accounts database object like he can query or insert into accounts.

revoke insert,

select on accounts from Ram

By the above command user ram's permissions like query or insert on accounts database object has been removed.



**Grant and Revoke Command**

**Difference between GRANT and REVOKE command.**

| GRANT | REVOKE |
|---|---|
| **GRANT command** allows a user to perform certain activities on the database. | **REVOKE command** disallows a user to perform certain activities. |
| It grants access privileges for database objects to other users. | It revokes access privileges for database objects previously granted to other users. |
| **Example:**<br><br>GRANT privilege_name<br>ON object_name<br>TO<br><br>{<br>user_name\|PUBLIC\|role_name<br>}<br><br>[WITH GRANT OPTION]; | **Example:**<br><br>REVOKE privilege_name<br>ON object_name<br><br>FROM<br>{<br>user_name\|PUBLIC\|role_name<br>} |

## 7.1 DEFINING A TRANSACTION

A transaction, in the context of a database, is a logical unit that is independently executed for data retrieval or updates. Experts talk about a database transaction as a “unit of work” that is achieved within a database design environment.

In relational databases, database transactions must be atomic, consistent, isolated and durable summarized as the ACID acronym. Engineers have to look at the build and use of a database system to figure out whether it supports the ACID model or not. Then, as newer kinds of database systems have emerged, the question of how to handle transactions becomes more complex.

In traditional relational database design, transactions are completed by COMMIT or ROLLBACK SQL statements, which indicate a transaction’s beginning or end. The ACID acronym defines the properties of a database transaction, as follows:

**Atomicity:** A transaction must be fully complete, saved (committed) or completely undone (rolled back). A sale in a retail store database illustrates a scenario which explains atomicity, e.g., the sale consists of an inventory reduction and a record of incoming cash. Both either happen together or do not happen—it's all or nothing.

**Consistency:** The transaction must be fully compliant with the state of the database as it was prior to the transaction. In other words, the transaction cannot break the database's constraints. For example, if a database table's Phone Number column can only contain numerals, then consistency dictates that any transaction attempting to enter an alphabetical letter may not commit.

**Isolation**: Transaction data must not be available to other transactions until the original transaction is committed or rolled back.

**Durability**: Transaction data changes must be available, even in the event of database failure.

For reference, one of the easiest ways to describe a database transaction is that it is any change in a database, any "transaction" between the database components and the data fields that they contain.

However, the terminology becomes confusing, because in enterprise as a whole, people are so used to referring to financial transactions as simply "transactions." That sets up a central conflict in tech-speak versus the terminology of the average person.

A database "transaction" is any change that happens. To talk about handling financial transactions in database environments, the word "financial" should be used explicitly. Otherwise, confusion can easily crop up. Database systems will need specific features, such as PCI compliance features, in order to handle financial transactions specifically.

As databases have evolved, transaction handling systems have also evolved. A new kind of database called NoSQL is one that does not depend on the traditional relational database data relationships to operate.

While many NoSQL systems offer ACID compliance, others utilize processes like snapshot isolation or may sacrifice some consistency for other goals. Experts sometimes talk about a trade-off between consistency and availability, or similar scenarios where consistently may be treated differently by modern database environments. This type of question is changing how stakeholders look at database systems, beyond the traditional relational database paradigms.

Oracle PL/SQL transaction oriented language. Oracle transactions provide a data integrity. PL/SQL transaction is a series of SQL data manipulation statements that are work logical unit. Transaction is an atomic unit all changes either committed or rollback.

At the end of the transaction that makes database changes, Oracle makes all the changes permanent save or may be undone. If your program fails in the middle of a transaction, Oracle detect the error and rollback the transaction and restoring the database.

You can use the COMMIT, ROLLBACK, SAVEPOINT, and SET TRANSACTION command to control the transaction.

**COMMIT:** COMMIT command to make changes permanent save to a database during the current transaction.

**ROLLBACK:** ROLLBACK command execute at the end of current transaction and undo/undone any changes made since the begin transaction.

**SAVEPOINT:** SAVEPOINT command save the current point with the unique name in the processing of a transaction.

**AUTOCOMMIT:** Set AUTOCOMMIT ON to execute COMMIT Statement automatically.

SET TRANSACTION: PL/SQL SET TRANSACTION command set the transaction properties such as read-write/read only access.

## 7.2 MAKING CHANGES PERMANENT WITH COMMIT

Committing a transaction means making permanent the changes performed by the SQL statements within the transaction.

Before a transaction that modifies data is committed, the following has occurred:

Oracle has generated undo information. The undo information contains the old data values changed by the SQL statements of the transaction.

Oracle has generated redo log entries in the redo log buffer of the SGA. The redo log record contains the change to the data block and the change to the rollback block. These changes may go to disk before a transaction is committed.

The changes have been made to the database buffers of the SGA. These changes may go to disk before a transaction is committed.

**When a transaction is committed, the following occurs:**

The internal transaction table for the associated undo tablespace records that the transaction has committed, and the corresponding unique system change number (SCN) of the transaction is assigned and recorded in the table.

The log writer process (LGWR) writes redo log entries in the SGA's redo log buffers to the redo log file. It also writes the transaction's SCN to the redo log file. This atomic event constitutes the commit of the transaction.

Oracle releases locks held on rows and tables.

Oracle marks the transaction complete.

The COMMIT statement to make changes permanent save to a database during the current transaction and visible to other users,

**Commit Syntax**

SQL>COMMIT [COMMENT "comment text"];

Commit comments are only supported for backward compatibility. In a future release commit comment will come to a deprecated.

Commit Example

```
SQL>BEGIN

    UPDATE emp_information SET emp_dept='Web Developer'

        WHERE emp_name='Saulin';

    COMMIT;

END;
```

## 7.3 UNDOING CHANGES WITH ROLLBACK

Rolling back means undoing any changes to data that have been performed by SQL statements within an uncommitted transaction. Oracle uses undo tablespaces (or rollback segments) to store old values. The redo log contains a record of changes.

Oracle lets you roll back an entire uncommitted transaction. Alternatively, you can roll back the trailing portion of an uncommitted transaction to a marker called a savepoint.

All types of rollbacks use the same procedures:

Statement-level rollback (due to statement or deadlock execution error)

Rollback to a savepoint

Rollback of a transaction due to user request

Rollback of a transaction due to abnormal process termination

Rollback of all outstanding transactions when an instance terminates abnormally

Rollback of incomplete transactions during recovery

In rolling back an entire transaction, without referencing any savepoints, the following occurs:

Oracle undoes all changes made by all the SQL statements in the transaction by using the corresponding undo tablespace.

Oracle releases all the transaction's locks of data. The transaction ends.

The ROLLBACK statement ends the current transaction and undoes any changes made during that transaction. If you make a mistake, such as deleting the wrong row from a table, a rollback restores the original data. If you cannot finish a transaction because an exception is raised or a SQL statement fails, a rollback lets you take corrective action and perhaps start over.

**ROLLBACK Syntax**

```
SQL>ROLLBACK [To SAVEPOINT_NAME];
```

ROLLBACK Example

```
SQL>DECLARE
    emp_id   emp.empno%TYPE;
BEGIN
    SAVEPOINT dup_found;
    UPDATE emp SET eno=1
           WHERE empname = 'Forbs ross'
EXCEPTION
    WHEN DUP_VAL_ON_INDEX THEN
        ROLLBACK TO dup_found;
END;
/
```

Above example statement is exception raised because eno = 1 is already so DUP_ON_INDEX exception rise and rollback to the dup_found savepoint named.

## 7.4 UNDOING PARTIAL CHANGESWITH SAVEPOINT

You can declare intermediate markers called savepoints within the context of a transaction. Savepoints divide a long transaction into smaller parts.

Using savepoints, you can arbitrarily mark your work at any point within a long transaction. You then have the option later of rolling back work performed before the current point in the transaction but after a declared savepoint within the transaction. For example, you can use savepoints throughout a long complex series of updates, so if you make an error, you do not need to resubmit every statement.

Savepoints are similarly useful in application programs. If a procedure contains several functions, then you can create a savepoint before each function begins. Then, if a function fails, it is easy to return the data to its state before the function began and re-run the function with revised parameters or perform a recovery action.

After a rollback to a savepoint, Oracle releases the data locks obtained by rolled back statements. Other transactions that were waiting for the previously locked resources can proceed. Other transactions that want to update previously locked rows can do so.

When a transaction is rolled back to a savepoint, the following occurs:

- Oracle rolls back only the statements run after the savepoint.
- Oracle preserves the specified savepoint, but all savepoints that were established after the specified one are lost.
- Oracle releases all table and row locks acquired since that savepoint but retains all data locks acquired previous to the savepoint.
- The transaction remains active and can be continued.

Whenever a session is waiting on a transaction, a rollback to savepoint does not free row locks. To make sure a transaction does not hang if it cannot obtain a lock, use FOR UPDATE ... NOWAIT before issuing UPDATE or DELETE statements. (This refers to locks obtained before the savepoint to which has been rolled back. Row locks obtained after this savepoint are released, as the statements executed after the savepoint have been rolled back completely.)

**Transaction Naming**

You can name a transaction, using a simple and memorable text string. This name is a reminder of what the transaction is about. Transaction names replace commit comments for distributed transactions, with the following advantages:

It is easier to monitor long-running transactions and to resolve in-doubt distributed transactions.

You can view transaction names along with transaction IDs in applications. For example, a database administrator can view transaction names in Enterprise Manager when monitoring system activity.

Transaction names are written to the transaction auditing redo record, if compatibility is set to Oracle9i or higher.

LogMiner can use transaction names to search for a specific transaction from transaction auditing records in the redo log.

You can use transaction names to find a specific transaction in data dictionary views, such as V$TRANSACTION.

**How Transactions Are Named**

Name a transaction using the SET TRANSACTION ... NAME statement before you start the transaction.

When you name a transaction, you associate the transaction's name with its ID. Transaction names do not have to be unique; different transactions can have the same transaction name at the same time by the same owner. You can use any name that enables you to distinguish the transaction.

**Commit Comment**

In previous releases, you could associate a comment with a transaction by using a commit comment. However, a comment can be associated with a transaction only when a transaction is being committed.

Commit comments are still supported for backward compatibility. However, Oracle strongly recommends that you use transaction names. Commit comments are ignored in named transactions.

SAVEPOINT savepoint_names marks the current point in the processing of a transaction. Savepoints let you rollback part of a transaction instead of the whole transaction.

**SAVEPOINT Syntax**

```
SQL>SAVEPOINT SAVEPOINT_NAME;
```

SAVEPOINT Example

```
SQL>DECLARE
    emp_id   emp.empno%TYPE;
BEGIN
    SAVEPOINT dup_found;
    UPDATE emp SET eno=1
          WHERE empname = 'Forbs ross'
EXCEPTION
    WHEN DUP_VAL_ON_INDEX THEN
```

```
        ROLLBACK TO dup_found;
END;
```

/Autocommit

No need to execute COMMIT statement every time. You just set AUTOCOMMIT ON to execute COMMIT Statement automatically. It's automatic execute for each DML statement. set auto commit on using following statement,

**AUTOCOMMIT Example**

```
SQL>SET AUTOCOMMIT ON;
```

You can also set auto commit off,

```
SQL>SET AUTOCOMMIT OFF;
```

**Set Transaction**

SET TRANSACTION statement is use to set transaction are read-only or both read write. you can also assign transaction name.

**SET TRANSACTION Syntax**

```
SQL>SET TRANSACTION [ READ ONLY | READ WRITE ]
          [ NAME 'transaction_name' ];
```

Set transaction name using the SET TRANSACTION [...] NAME statement before you start the transaction.

**SET TRANSACTION Example**

```
SQL>SET TRANSACTION READ WRITE NAME 'tran_exp';
```

## SAVEPOINT For Reverting Partial Changes

SAVEPOINT gives name and identification to the present transaction processing point. It is generally associated with a ROLLBACK statement. It enables us to revert some sections of a transaction by not touching the entire transaction.

As we apply ROLLBACK to a SAVEPOINT, all the SAVEPOINTS included following that particular SAVEPOINT gets removed [that is if we have marked three SAVEPOINTS and applied a ROLLBACK on the second SAVEPOINT, automatically the third SAVEPOINT will be deleted.]

A COMMIT or a ROLLBACK statement deletes all SAVEPOINTS. The names given to SAVEPOINT are undeclared identifiers and can be reapplied several times inside a transaction. There is a movement of SAVEPOINT from the old to the present position inside the transaction.

A ROLLBACK applied to a SAVEPOINT affects only the ongoing part of the transaction. Thus a SAVEPOINT helps to split a lengthy transaction into small sections by positioning validation points.

**Syntax for transaction SAVEPOINT:**

SAVEPOINT < save_n>;

Here, save_n is the name of the SAVEPOINT.

Let us again consider the TEACHERS table we have created earlier.

Code implementation of ROLLBACK WITH SAVEPOINT:

```
INSERT INTO TEACHERS VALUES (4, 'CYPRESS', 'MICHEAL');
SAVEPOINT s;
INSERT INTO TEACHERS VALUES (5, 'PYTHON', 'STEVE');
INSERT INTO TEACHERS VALUES (6, 'PYTEST', 'ARNOLD');
ROLLBACK TO s;
INSERT INTO TEACHERS VALUES (7, 'PROTRACTOR',
'FANNY');
COMMIT;
```

Next, the below query is executed:

```
SELECT * FROM TEACHERS;
```

**Output of the above code should be:**

| CODE | SUBJECT | NAME |
|---|---|---|
| 2 | UFT | SAM |
| 1 | SELENIUM | TOP |
| 3 | JMETERE | TONK |
| 4 | CYPRESS | MICHEAL |
| 7 | PROTRACTOR | FANNY |

In the above code, after ROLLBACK with SAVEPOINT s is applied, only two more rows got inserted, i.e. teachers with CODE 4 and 7, respectively. Please note teachers with code 1, 2, and 3 have been added during the table creation.

## 7.5 ROLLBACK TO UNDO CHANGES

If a present transaction is ended with a ROLLBACK statement, then it will undo all the modifications that are supposed to take place in the transaction.

A ROLLBACK statement has the following features as listed below:

The database is restored with its original state with a ROLLBACK statement in case we have mistakenly deleted an important row from the table.

In the event of an exception which has led to the execution failure of a SQL statement, a ROLLBACK statement enables us to jump to the starting point of the program from where we can take remedial measures.

The updates made to the database without a COMMIT statement can be revoked with a ROLLBACK statement.

**Syntax for transaction ROLLBACK:**

ROLLBACK;

Syntax for transaction ROLLBACK with SAVEPOINT:

ROLLBACK [TO SAVEPOINT < save_n>];

Here, the save_n is the name of the SAVEPOINT.

Let us consider the TEACHERS table we have created earlier.

Code implementation with ROLLBACK:

```
DELETE FROM TEACHERS WHERE CODE= 3;
 ROLLBACK;
```

Next, the below query is executed:

```
SELECT * FROM TEACHERS;
```

**Output of the above code should be:**

| CODE | SUBJECT | NAME |
|---|---|---|
| 2 | UFT | SAM |
| 1 | SELENIUM | TOP |
| 3 | JMETERE | TONK |

In the above code, we have executed a DELETE statement which is supposed to delete the record of the teacher with CODE equal to 3. However, because of the ROLLBACK statement, there is no impact on the database, and deletion is not done.

❖❖❖❖

8

# CRASH RECOVERY

**Unit Structure**

DBMS is a highly complex system with hundreds of transactions being executed every second. The durability and robustness of a DBMS depends on its complex architecture and its underlying hardware and system software. If it fails or crashes amid transactions, it is expected that the system would follow some sort of algorithm or techniques to recover lost data.

**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 −**

1. Logical errors − Where a transaction cannot complete because it has some code error or any internal error condition.
2. 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 −

1. **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.

2. **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 −

1. It should check the states of all the transactions, which were being executed.
2. A transaction may be in the middle of some operation; the DBMS must ensure the atomicity of the transaction in this case.
3. It should check whether the transaction can be completed now or it needs to be rolled back.

4. 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 −

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

## 8.1 ARIES ALGORITHM

**(Algorithm for Recovery and Isolation Exploiting Semantics (ARIES))**

Algorithm for Recovery and Isolation Exploiting Semantics (ARIES) is based on the Write Ahead Log (WAL) protocol. Every update operation writes a log record which is one of the following :

1. Undo-only log record:

Only the before image is logged. Thus, an undo operation can be done to retrieve the old data.

2. Redo-only log record:

Only the after image is logged. Thus, a redo operation can be attempted.

3. Undo-redo log record:

Both before images and after images are logged.

In it, every log record is assigned a unique and monotonically increasing log sequence number (LSN). Every data page has a page LSN field that is set to the LSN of the log record corresponding to the last update on the page. WAL requires that the log record corresponding to an update make it to stable storage before the data page corresponding to that update is written to disk. For performance reasons, each log write is not immediately forced to disk. A log tail is maintained in main memory to buffer log writes. The log tail is flushed to disk when it gets full. A transaction cannot be declared committed until the commit log record makes it to disk.

Once in a while the recovery subsystem writes a checkpoint record to the log. The checkpoint record contains the transaction table and the dirty page table. A master log record is maintained separately, in stable storage, to store the LSN of the latest checkpoint record that made it to disk. On restart, the recovery subsystem reads the master log record to find the checkpoint's LSN, reads the checkpoint record, and starts recovery from there on.

**The recovery process actually consists of 3 phases:**

**Analysis:**

The recovery subsystem determines the earliest log record from which the next pass must start. It also scans the log forward from the checkpoint record to construct a snapshot of what the system looked like at the instant of the crash.

**Redo:**

Starting at the earliest LSN, the log is read forward and each update redone.

**Undo:**

The log is scanned backward and updates corresponding to loser transactions are undone.

Attention reader! Don’t stop learning now. Get hold of all the important CS Theory concepts for SDE interviews with the CS Theory Course at a student-friendly price and become industry ready.

## 8.2 LOG BASED RECOVERY

Log is nothing but a file which contains a sequence of records, each log record refers to a write operation. All the log records are recorded step by step in the log file. We can say, log files store the history of all updates activities.

Log contains start of transaction, transaction number, record number, old value, new value, end of transaction etc. For example, mini statements in bank ATMs.

If within an ongoing transaction, the system crashes, then by using log files, we can return back to the previous state as if nothing has happened to the database.

The log is kept on disk so that it is not affected by failures except disk and failures.

Example :

Different types of log records are as follows −

<Ti, Xi, V1, V2> − update log record, where Ti=transaction, Xi=data, V1=old

data, V2=new value.

<Ti, start> − Transaction Ti starts execution.

<Ti, commit> − Transaction Ti is committed.

<Ti, abort> − Transaction Ti is aborted

The log records can be written as follows −

Create a log for the given transaction T1 and T2.

| **T1** | **T2** | **Log** |
|---|---|---|
| Read A | Read A | <T1, start> |
| A=A-2000 | A=A+5000 | <T1,A,5000, 3000> |
| Write A | Write A | <T1, B, 8000, 10000> |
| Read B | Read B | <T1, commit> |
| B=B+2000 | B= B+7000 | <T2, start> |
| Write B | Write B | <T2, A, 3000, 8000> |
| | | <T2, B, 10000, 17000> |
| | | <T2, commit> |

## Log Based Recovery

Log-based recovery provides the facility to maintain or recover data if any failure may occur in the system. Log means sequence of records or data, each transaction DBMS creates a log in some stable storage device so that we easily recover data if any failure may occur. When we perform any operation on the database at that time it will be recorded into a log file. Processing of the log file should be done before the original transaction is applied to the database. Why we use log-based recovery the main reason is that the Atomicity property of transaction states that we can either execute the whole transaction or nothing else, modification of the aborted transaction is not visible to the database, and modification of the transaction is visible so that reason we use log-based recovery system.

**Syntax:**

<TRX, Start>

Explanation: In the above syntax, we use TRX and start in which TRX means transaction and when a transaction in the initial state that means we write start a log.

<TRX, Name, 'First Name', 'Last Name' >

Explanation:In this syntax where TRX means transaction and name is used to First Name and Last Name. When we modify the name First Name to the Last Name then it writes a separate log file for that.

<TRX, Commits>

Explanation:In the above syntax, we use two-variable transactions as TRX and commits, when transaction execution is finished then it is written into another log file that means the end of the transaction we called commits.

**Log-Based Recovery in DBMS**

Log-based recovery uses the following term for execution as follows.

**Transaction Identifier**: It used to uniquely identify the transaction.

**Data item Identifier**: It is used to uniquely identify the used data in the database.

**Old Value**: It is the value of data before the write operation of a transaction.

**New Value**: It is the value of data after the write operation of a transaction.

Let's see the transaction with various log types.

First, we start the transaction by using <TRX, Start> this syntax after that we perform the write operation of the transaction that means we update the database. After the write operation, we check whether the transaction is committed or aborted.

For recovery purposes, we use the following two operations as follows.

**Undo (TRX):** This command is used to restore all records updated by transactions to the old value.

**Redo (TRX):** This command is used to set the value of all records updated by a transaction to the new value.

**Transaction Modification Techniques**

There are two different types we use in database medication and that are helpful in the recovery system as follows. Transaction Modification Techniques as follows:

**1. Immediate Database Modification**

In this type, we can modify the database while the transaction is an inactive state. Data modification done by an active transaction is called an uncommitted transaction. When a transaction is failed or we can say that a system crash at that time, the transaction uses the old transaction to bring the database into a consistent state. This execution can be completed by using the undo operation.

Example:

<TRX1 start>

<TRX1 X, 2000, 1000>

<TRX1 Y, 3000, 1050>

<TRX1 commit>

<TRX2 start>

<TRX2 Z, 800, 500>

<TRX2 commit>

**Explanation**: In the above example we consider the banking system, the transaction TRX1 is followed by TRX2. If a system crash or a transaction fails in this situation means during recovery we do redo transaction TRX1 and undo the transaction TRX2 because we have both TRX start and commit state in the log records. But we don't have a start and commit state for transaction TRX2 in log records. So undo transaction TRX2 done first the redo transaction TRX1 should be done.

**2. Deferred Modification Technique**

In this technique, it records all database operations of transactions into the log file. In this technique, we can apply all write operations of transactions on the database if the transaction is partially committed. When a transaction is partially committed at that time information in the log file is used to execute deferred writes. If the transaction fails to execute or the system crashes or the transaction ignores information from the log file. In this situation, the database uses log information to execute the transaction. After failure, the recovery system determines which transaction needs to be redone.

Example

(X) (Y)

<TRX1 start> <TRX1 start>

<TRX1 X, 850><TRX1 X , 850>

<TRX1 Y, 105><TRX1 Y, 1050>

<TRX1 commit>

<TRX2 start>

<TRX 2 Z, 500>

**Explanation**: If the system fails after write Y of transaction TRX1 then there is no need to redo operation because we have only <TRX1 start> in log record but don't have <TRX1 commit>. In the second transaction Y, we can do the redo operation because we have <TRX1 start> and <TRX1 commit> in log disk but at the same time, we have <TRX2 start> but don't have <TRX2 commit> as shown in the above transaction.

(Z)

<TRX1 start>

<TRX1 X , 850>

<TRX1 Y, 1050>

<TRX1 commit>

<TRX2 start>

<TRX2 Z, 500>

<TRX2 commit>

**Explanation**: In the above transaction, we have <TRX start> and <TRX commit> in log disk so we can redo operation during the recovery system.

Suppose we need to restore records from binary logs and by default, server creates binary logs. At that time you must know the name and current location of the binary log file, so by using the following statement we can see the file name and location as follows.

show binary logs;

**Explanation**: In the above statement, we use the show command to see binary logs. Illustrate the final result of the above statement by using the following snapshot.

**Example** show master status;

**Explanation**: Suppose we need to determine the current binary log file at that time we can use the above statement.

## 8.3 RECOVERY RELATED STRUCTURES

Structures Used for Database Recovery: Several structures of an Oracle database safeguard data against possible failures. The following sections briefly introduce each of these structures and its role in database recovery.

### Database Backups

A database backup consists of operating system backups of the physical files that constitute an Oracle database. To begin database recovery from a media failure, Oracle uses file backups to restore damaged datafiles or control files.

Oracle offers several options in performing database backups; "Database Backup", for more information.

### The Redo Log

The redo log, present for every Oracle database, records all changes made in an Oracle database. The redo log of a database consists of at least two redo log files that are separate from the datafiles (which actually store a database's data). As part of database recovery from an instance or media failure, Oracle applies the appropriate changes in the database's redo log to the datafiles, which updates database data to the instant that the failure occurred.

A database's redo log can be comprised of two parts: the online redo log and the archived redo log, discussed in the following sections.

### The Online Redo

Log Every Oracle database has an associated online redo log. The online redo log works with the Oracle background process LGWR to immediately record all changes made through the associated instance. The online redo log consists of two or more preallocated files that are reused in a circular fashion to record ongoing database changes;

### The Archived (Offline) Redo Log

Optionally, you can configure an Oracle database to archive files of the online redo log once they fill. The online redo log files that are archived are uniquely identified and make up the archived redo log. By archiving filled online redo log files, older redo log information is preserved for more extensive database recovery operations, while the pre-allocated online redo log files continue to be reused to store the most current database changes;

### Rollback Segments

Rollback segments are used for a number of functions in the operation of an Oracle database. In general, the rollback segments of a database store the old values of data changed by ongoing transactions (that is, uncommitted transactions). Among other things, the information in a rollback segment is used during database recovery to "undo" any "uncommitted" changes applied from the redo log to the datafiles. Therefore, if database recovery is necessary, the data is in a consistent state after the rollback segments are used to remove all uncommitted data from the datafiles; see "Rollback Segments".

**Control Files**

In general, the control file(s) of a database store the status of the physical structure of the database. Certain status information in the control file (for example, the current online redo log file, the names of the datafiles, and so on) guides Oracle during instance or media recovery

## 8.4 DIRTY PAGES TABLE

This table is used to represent information about dirty buffer pages during normal processing. It is also used during restart recovery.It is implemented using hashing or via the deferred- writes queue mechanism. Each entry in the table consists of 2 fields :

1. PageID and
2. RecLSN

During normal processing , when a non-dirty page is being fixed in the buffers with the intention to modify , the buffer manager records in the buffer pool (BP) dirty-pages table , as RecLSN , the current end-of-log LSN , which will be the LSN of the next log record to be written. The value of RecLSN indicates from what point in the log there may be updates. Whenever pages are written back to nonvolatile storage , the corresponding entries in the BP dirty-page table are removed. The contents of this table are included in the checkpoint record that is written during normal processing. The restart dirty-pages table is initialized from the latest checkpoint's record and is modified during the analysis of the other records during the analysis pass. The minimum RecLSN value in the table gives the starting point for the redo pass during restart recovery.

ARIES maintains two data structures and adds one more field to log record:

1. **Transaction table**: It contains all the transactions that are active at any point of time (i.e. are started but not committed/aborted). The table also stores the LSN of last log record written by the transaction in “lastLSN” field.

2. **Dirty page table**: Contains an entry for each page that has been modified but not written to disk. The table also stores the LSN of the first log record that made the associated page dirty in a field called “recoveryLSN” (also called “firstLSN”). This is the log record from which REDO need to restart for this page.

3. In addition log records are also updated to contain a field called “prevLSN” which points to previous log record for the same transaction. This creates a linked list of all log records

for a transaction. When a new log record is created, "lastLSN" from transaction table is filled into its "prevLSN" field. And the LSN of current log record becomes the "lastLSN" in transaction table. Here is updated log record table with prevLSN filled in:

4.

| LSN | Prev LSN | Transaction ID | Type | Page ID |
|---|---|---|---|---|
| 1 | NIL | T1 | UPDATE | P3 |
| 2 | NIL | T2 | UPDATE | P2 |
| 3 | 1 | T1 | COMMIT | |
| 4 | CHECKPOINT | | | |
| 5 | NIL | T3 | UPDATE | P1 |
| 6 | 2 | T2 | UPDATE | P3 |
| 7 | 6 | T2 | COMMIT | |

During checkpointing, a checkpoint log record is created. This log record contains the content of both "Transaction table" and "Dirty page table". "Analysis" phase starts by reading last checkpoint log record to get the information about active transactions and dirty pages. Here is content of "Transaction table" and "Dirty page table" at the checkpoint stage in above table at LSN 4:

**Transaction Table**

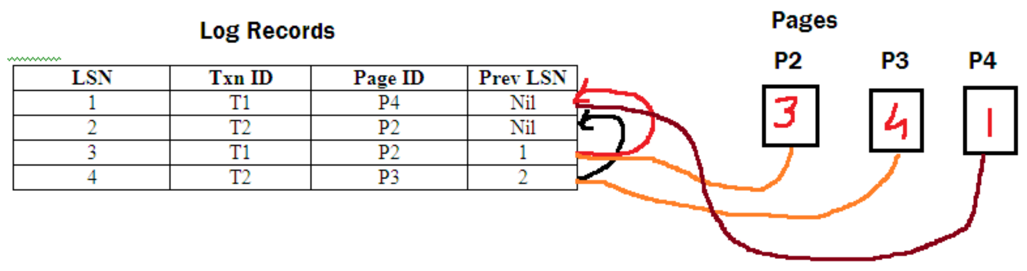
| Transaction ID | Last LSN | Status |
|---|---|---|
| T1 | 3 | Commit |
| T2 | 2 | In Progress |

**Dirty Page Table**

| Page ID | Recovery LSN |
|---|---|
| P3 | 1 |
| P2 | 2 |

This whole setup can be visualized in following picture, pay attention to LSN for P2 in the "pages" list and in dirty table (dirty page table has the first LSN, whereas the P2 page has the last LSN):

**Log Records :**


Log Records

| LSN | Txn ID | Page ID | Prev LSN |
|---|---|---|---|
| 1 | T1 | P4 | Nil |
| 2 | T2 | P2 | Nil |
| 3 | T1 | P2 | 1 |
| 4 | T2 | P3 | 2 |

**Transaction Table**

| Txn ID | Last LSN |
|---|---|
| T1 | 3 |
| T2 | 4 |

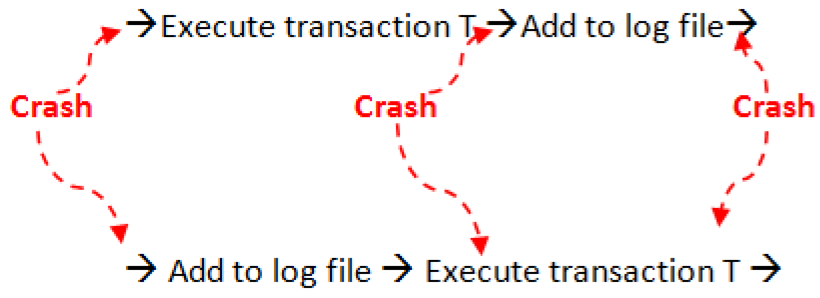
**Dirty Page Table**

| Page ID | LSN |
|---|---|
| P1 | 2 |
| P2 | 4 |
| P3 | 1 |

**ARIES Data Structures**

## 8.5 WRITE-AHEAD LOG IN DBMS

We have learnt that logs have to kept in the memory, so that when there is any failure, DB can be recovered using the log files. Whenever we are executing a transaction, there are two tasks – one to perform the transaction and update DB, another one is to update the log files. But when these log files are created – Before executing the transaction, or during the transaction or after the transaction? Which will be helpful during the crash ?


→Execute transaction T→Add to log file→
Crash
Crash
Crash
→ Add to log file → Execute transaction T →

When a log is created after executing a transaction, there will not be any log information about the data before to the transaction. In addition, if a transaction fails, then there is no question of creating the log itself. Suppose there is a media failure, then how a log file can be created? We will lose all the data if we create a log file after the transaction. Hence it is of no use while recovering the data.

Suppose we created a log file first with before value of the data. Then if the system crashes while executing the transaction, then we know what its previous state / value was and we can easily revert the changes. Hence it is always a better idea to log the details into log file before the transaction is executed. In addition, it should be forced to update the log files first and then have to write the data into DB. i.e.; in ATM withdrawal, each stages of transactions should be logged into log files, and stored somewhere in the memory. Then the actual balance has to be updated in DB. This will guarantee the atomicity of the transaction even if the system fails. This is known as Write-Ahead Logging Protocol.

But in this protocol, we have I/O access twice – one for writing the log and another for writing the actual data. This is reduced by keeping the log buffer in the main memory – log files are kept in the main memory for certain pre-defined time period and then flushed into the disk. The log files are appended with data for certain period, once the buffer is full or it reaches the time limit, then it is written into the disk. This reduces the I/O time for writing the log files into the disk.

Similarly retrieving the data from the disk is also needs I/O. This can also be reduced by maintaining the data in the page cache of the main memory. That is whenever a data has to be retrieved; it will be retrieved from the disk for the first time. Then it will be kept in the page cache for the future reference. If the same data is requested again, then it will be retrieved from this page cache rather than retrieving from the disk. This reduces the time for retrieval of data. When the usage / access to this data reduce to some threshold, then it will be removed from page cache and space is made available for other data.

## 8.6 CHECKPOINT

1. The checkpoint is a type of mechanism where all the previous logs are removed from the system and permanently stored in the storage disk.

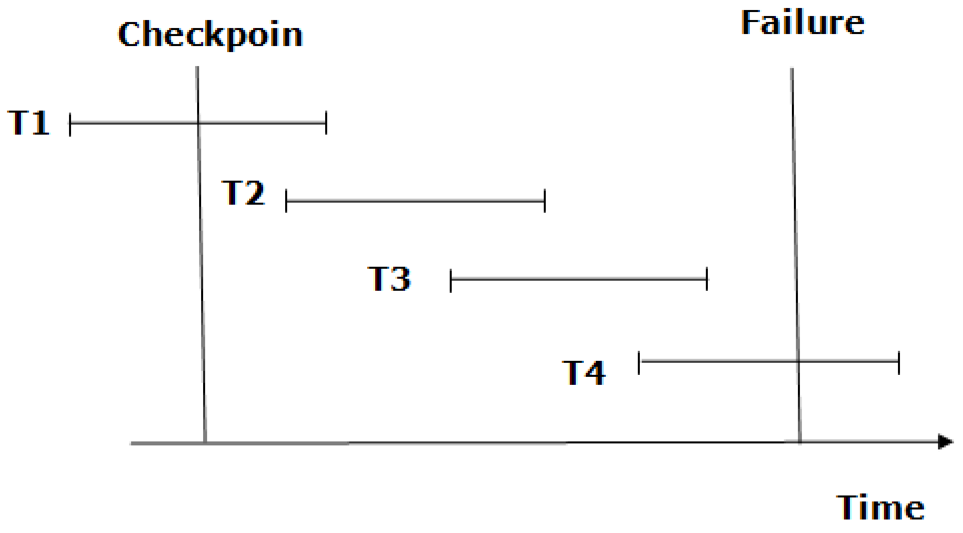
2. The checkpoint is like a bookmark. While the execution of the transaction, such checkpoints are marked, and the transaction is executed then using the steps of the transaction, the log files will be created.

3. When it reaches to the checkpoint, then the transaction will be updated into the database, and till that point, the entire log file will be removed from the file. Then the log file is updated with the new step of transaction till next checkpoint and so on.

4. The checkpoint is used to declare a point before which the DBMS was in the consistent state, and all transactions were committed.

**Recovery using Checkpoint**

In the following manner, a recovery system recovers the database from this failure:



- The recovery system reads log files from the end to start. It reads log files from T4 to T1.
- Recovery system maintains two lists, a redo-list, and an undo-list.
- The transaction is put into redo state if the recovery system sees a log with <Tn, Start> and <Tn, Commit> or just <Tn, Commit>. In the redo-list and their previous list, all the transactions are removed and then redone before saving their logs.
- For example: In the log file, transaction T2 and T3 will have <Tn, Start> and <Tn, Commit>. The T1 transaction will have only <Tn, commit> in the log file. That's why the transaction is committed after the checkpoint is crossed. Hence it puts T1, T2 and T3 transaction into redo list.
- The transaction is put into undo state if the recovery system sees a log with <Tn, Start> but no commit or abort log found. In the undo-list, all the transactions are undone, and their logs are removed.

For example: Transaction T4 will have <Tn, Start>. So T4 will be put into undo list since this transaction is not yet complete and failed amid.

## 8.7 RECOVERY FROM 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.

Computers crash for a variety of reasons. Random computer crashes are both frustrating and difficult for an average user to diagnose, but underneath the surface of a computer crash are five likely culprits examined below.

**1: Corrupted System Registry Files**

Every Windows-based PC has something called a Windows registry. The registry contains several files that are integral to the performance and operation of your computer. Over time, some of those files can become corrupted, be misplaced or get lost altogether. When that happens, the system registry becomes compromised – and frequent crashes are all-too-common symptoms. The best way to rule this possibility in or out is by running a Windows registry cleaning program. Such programs scan your Windows registry for problems then automatically make repairs. If you run a registry cleaner and the crashes persist, they are probably being caused by a different issue.

**2: Disorganized Files**

Windows operating systems handle file organization in a way that isn't very intuitive. Basically, they break files up and fit them into gaps in the computer's memory. As time goes by, these disorganized files can prompt frequent crashes. Luckily, a great optimization solution is built right into Windows-based PCs: the disk defragmentation utility. Although its location on a computer varies, you can generally locate it within the System and Security section inside the Control Panel. By running a defrag once every few months, you may be able to keep those pesky computer crashes at bay.

**3: Malicious Software**

Malicious software can take many different forms. Sometimes, it's a virus that is accidentally unleashed after opening a strange email; other times, its adware that tags along with other information that is automatically downloaded from a website. Whatever type it is, there's no question that malicious software can wreak havoc on a computer's performance. Happily, there are many topnotch programs out there that regularly scan your computer for the presence of such problems – and that help guard against them, too. Buy one, install it and use it regularly; your crash issues may come to an end.

**4: Too Little Available Memory**

When you buy a new computer, it feels like there's no end to the amount of memory that it has. Of course, this isn't true at all. As never-ending as the available memory on your PC may initially seem, the fact is that it can be depleted with incredible speed. You can find out for sure by checking the information within "My Computer." If it appears that your available memory is low, you can use a PC cleanup program to remove unnecessary files; such programs remove things like temporary Internet files and other file debris that can suck away much-needed memory.

**5: Overheating**

If you've run through all of the preceding possibilities and continue experiencing frequent crashes, a hardware issue could be to blame. An easy one to rule out is overheating. A computer's CPU, or central processing unit, includes a fan that is designed to keep it running cool. Sometimes, the fan wears down and doesn't work as efficiently; other times, it's just not able to handle the work that your computer has to do. In either case, buying a bigger, better fan isn't very expensive. If it puts an end to your PC crashing problem, it will have been more than worth it.

## 8.8 REDO PHASE UNDO PHASE

**REDO PHASE:-**

1. Redo phase is the second phase where all the transactions that are needed to be executed again take place.
2. It executes those operations whose results are not reflected in the disk.
3. It can be done by finding the smallest LSN of all the dirty page in dirty page table that defines the log positions, & the Redo operation will start from this position
4. This position indicates that either the changes that are made earlier are I the main memory or they have already been flaunted to the disk.
5. Thus, for each change recorded in the log, the Redo phase determines whether or not the operations have been re-executed.

**UNDO PHASE:-**

1. In the Undo phase, all the transaction, that is listed in the active transaction set here to be undone.
2. Thus the log should be scanned background from the end & the recovery manages should Undo the necessary operations.
3. Each time an operations is undone, a compensation log recorded has been written to the log.
4. This process continues until there is no transaction left in the active transaction set.
5. After the successful competition of this phase, database can resume its normal operations.

❖❖❖❖

.