

Unit I

1

STORED PROCEDURE

Unit Structure

1.0 Objective

1.1. Introduction

1.2 Types of Stored Procedure

1.2.1 User Defined Procedures

1.2.2 System Stored procedure

1.2.3 Temporary Stored Procedure

1.2.4 Remote Stored Procedures

1.2.5 Extended Stored Procedure

1.3 Benefits of Stored Procedure

1.4 Creating Stored Procedures

1.4.1 Passing parameters in procedure

1.4.1.1 Procedure without parameter

1.5 Executing Stored Procedures

1.5.1 Executing the stored procedure with one parameter

1.5.2 Creating and Executing Procedure with multiple input parameters

1.5.3 Creating a stored procedure with default parameters values

1.5.4 Creating a stored procedure with an output parameter

1.6 Altering Stored Procedures

1.7 Deleting a Stored Procedure

1.8 Viewing Stored Procedures

1.9 Summarization

1.10 References

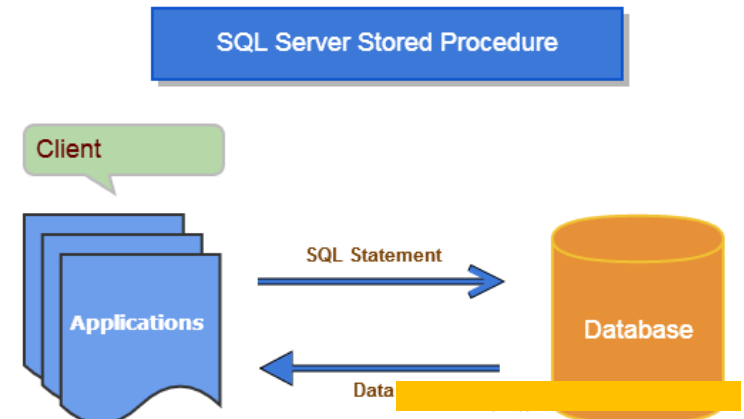
1.0 OBJECTIVE

- 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>
```

Stored Procedure

Database Management Systems 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*

1	create or replace procedure welcome
2	as
3	begin
4	dbms_output.put_line('welcome to Mumbai University');
5	dbms_output.put_line('This is Database Management System
6	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
-------	--------	------	-----

Stored Procedure

Database Management Systems

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	hydreaad	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
7	eid=100;
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

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

Stored Procedure

Database Management Systems

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 compiled 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*.

1	create or replace procedure insertemployee
2	(iname in varchar2,istreet in varchar2 ,icity in varchar2,ieid in
3	number)
4	is
5	begin
6	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	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
radha	3rd street	erode	112

Stored Procedure

Database Management Systems **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.

```
1 create or replace procedure SelectUser
2 (id in number :=105)
3 is
4 o_ename varchar2(30);
5 o_eid number(10);
6 begin
7 select ename,eid into o_ename,o_eid from employee_csc where
8 eid=id;
9 dbms_output.put_line('employee name = ' ||o_ename);
10 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.

Stored Procedure

Database Management Systems

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:/plsqli/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)

Stored Procedure

Database Management Systems

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,
2 e_name out varchar2)
3 is
4 begin
5 select ename into e_name from employee_csc where eid=id;
6 end;
7 /
```

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;
7 /
```

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

1	set serveroutput on
2	declare
3	r_count number;
4	begin
5	find_rows(r_count);
6	dbms_output.put_line('number of records in table =' r_count);
7	end;
	/

Stored Procedure

Database Management Systems

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;
```

Stored Procedure

Database Management Systems

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.



Stored Procedure

TRIGGERS AND SEQUENCES

Unit Structure

2.0 Objective

2.1 Introduction

2.2 Overview

2.3 Trigger Classification

2.4 Implementing Triggers

2.4.1 Creating a Trigger

2.4.2 Insert and update using a Trigger

2.4.3 Deleting through a Trigger

2.5 Viewing, deleting and modifying Triggers

2.6 Enforcing data integrity through Triggers

2.7 Nested triggers

2.8 Advantages of Triggers

2.9 Sequences

2.10 Creating Sequence

2.11 Referencing a Sequence

2.12 Altering a sequence

2.13 Deleting a Sequence:

2.14 Summarize

2.15 List of References

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 let's 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

Triggers and Sequences

Database Management Systems

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.

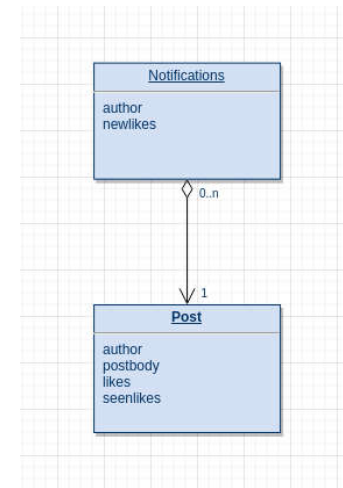
Line 4: **FOLLOWS / PRECEDES**.

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;
```

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



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;
```

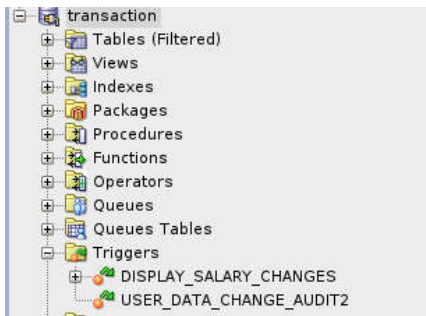
This should list all our triggers, something like this

Script Output x Query Result x

 All Rows Fetched: 4 in 0.342 seconds

	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

Triggers and Sequences

Database Management Systems

```
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.

1	create or replace trigger emp_trig
2	before
3	insert on employee
4	for each row
5	begin
6	:new.empname:=upper(:new.empname);
7	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_name varchar2(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	(:old.product_id,:old.product_name,:old.supplier_name,:old.unit_price
9);
	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

```

```

4. UPDATE employee_table
5. SET salary = salary + 5000;
6. IF sql%notfound THEN
7.   dbms_output.put_line('no employee record updated');
8. ELSIF sql%found THEN
9.   total_rows := sql%rowcount;
10.  dbms_output.put_line( total_rows || ' employee updated ');
11. END IF;
12. END;
13. /

```

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.

```
1 create or replace trigger dept_check
2 after delete or update of deptno on dept
3 for each row
4 begin
5 if updating and :old.deptno != :new.deptno or deleting then
6 update employee_csc set employee_csc.deptno=null
7 where employee_csc.deptno=:old.deptno;
8 end if;
9 end;
10 /
```

Triggers and Sequences

Database Management Systems

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

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

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

```
1 create or replace trigger dept_cascade_delete
2 after delete on dept
3 for each row
4 -- Before row is deleted from dept
5 -- delete all rows from employee_csc table whose deptno value is
6 same as
7 -- dept table
8 begin
9 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_prgrs\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

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.

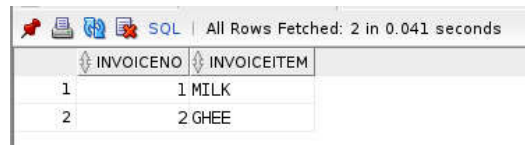
```
INSERT INTO INVOICE VALUES (invoice_number.NEXTVAL, 'MILK');
```

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

Give,

```
SELECT * FROM INVOICE;
```

Output



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



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. Nilesch 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



Triggers and Sequences

FILE ORGANIZATION AND INDEXING

Unit Structure

3.0 Objective

3.1 Introduction -File organization

3.2 Types of File organization

3.2.1 Sequential File Organization

3.2.2 Heap File organization

3.2.3 Hash File Organization

3.2.4 B+ Tree File Organization

3.2.5 ISAM File Organization

3.2.6 Cluster File Organization

3.3. Indexing

3.3.1 Introduction

3.3.2 Database Indexing attributes

3.3.3 Types of index files

3.3.3.1 Primary Indexing

3.3.3.2 Secondary Indexing

3.3.3.3 Cluster Indexing

3.3.3.4 Tree based indexing - B-Tree Indexing

3.4 Comparison of file organization

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

3.4.2 Comparison of I/O Costs

3.5 Creating, dropping and Altering indexes

3.5.1 Creating Indexes

3.5.2 Altering the Indexes

3.5.3 Removing the indexes

3.6 Summarization

3.7 References

3.0 OBJECTIVE

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

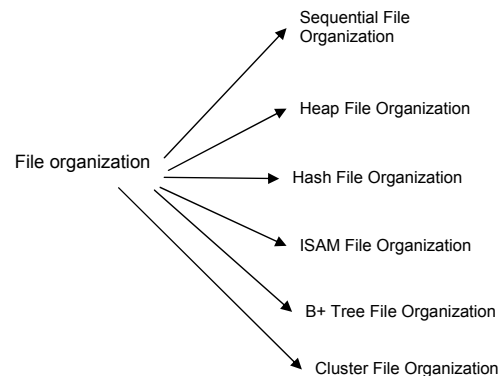
- Describe how files and records can be placed on disks and the effective way in which the records are organized in files
- Know the various commonly used file organizations used in database.
- Describe various indexes commonly used in database environments
- Understand the data structures which can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

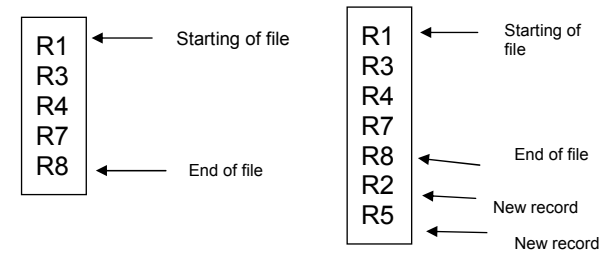
- Pile File Method
- Sorted File Method

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

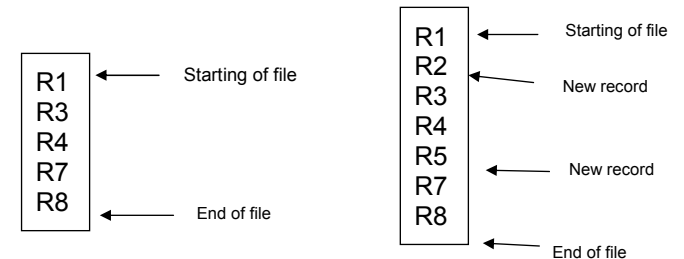
deletion of a record, the whole file which is stored as a memory block has to be searched. When it is found, it will be updated or deleted accordingly.

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

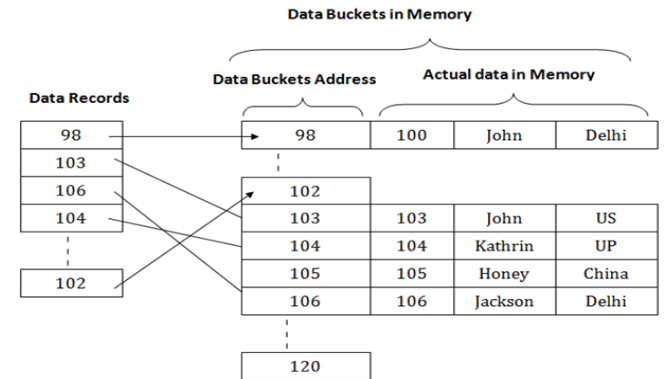
- Problem of unused memory block.
- This method will be a big problem when a large database is stored in this organization. As any search starts from the beginning of the file, it takes a lot of time for any updation, deletion etc.,

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

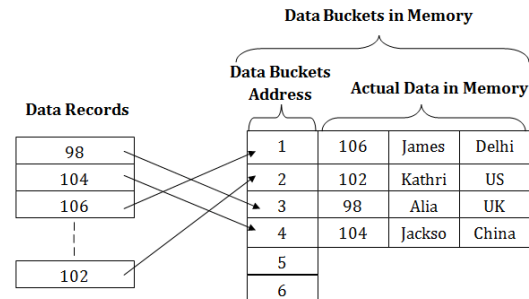
Types of hashing

- 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:

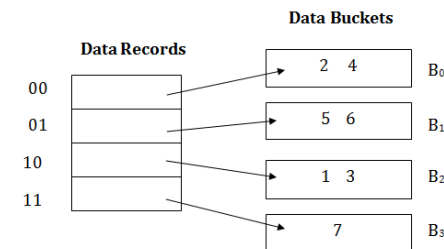
- Firstly, you have to follow the same procedure for retrieval, ending up in some bucket.
- If there is still space in that bucket, then place the record in it.
- If the bucket is full, then we will split the bucket and redistribute the records.

Example:

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

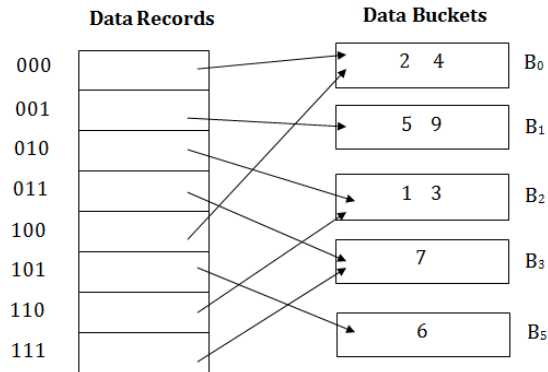
Key	Hash address
1	11010
2	00000
3	11110
4	00000
5	01001
6	10101
7	10111

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.

at the leaf node. Intermediate nodes act as a pointer to the leaf nodes. They do not contain any records.



Advantages

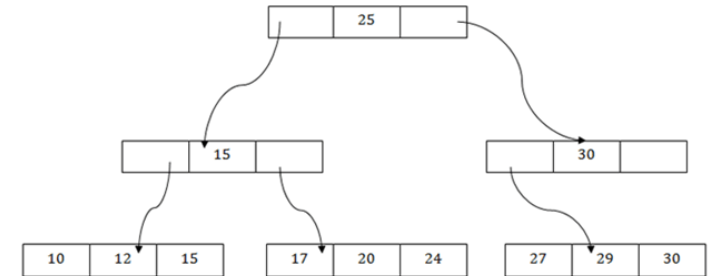
- In this method, the performance does not decrease as the data grows in the system. It simply increases the size of memory to accommodate the data.
- In this method, memory is well utilized as it grows and shrinks with the data. There will not be any unused memory lying.
- This method is good for the 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



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

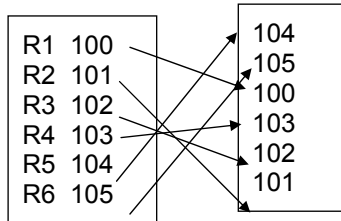
- In this method, searching becomes very easy as all the records are stored only in the leaf nodes and sorted in the sequential linked list.
- Traversing through the tree structure is easier and faster.
- The size of the B+ tree has no restrictions, so the number of records can increase or decrease and 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

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

← Primary key

works_csc

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

Foreign

← Primary

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

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				
110	krish	3rd street	bangalore				



Cluster key

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.

3.3.1 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.

Key	Value
Search Key	Data Reference

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

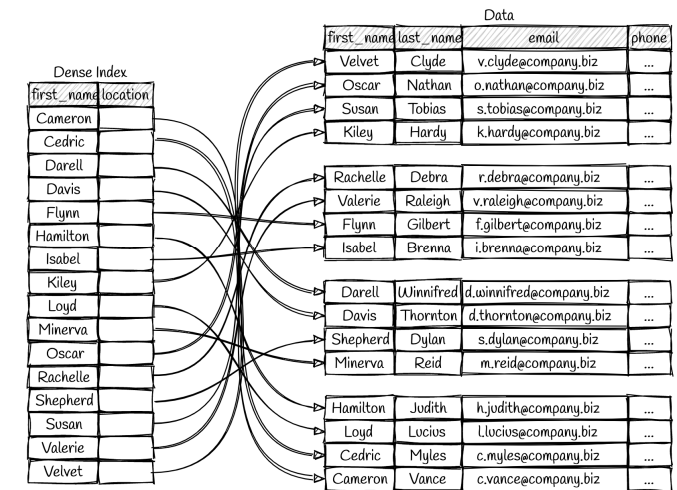
- Primary Index
- Secondary Index
- Clustering Index

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

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.

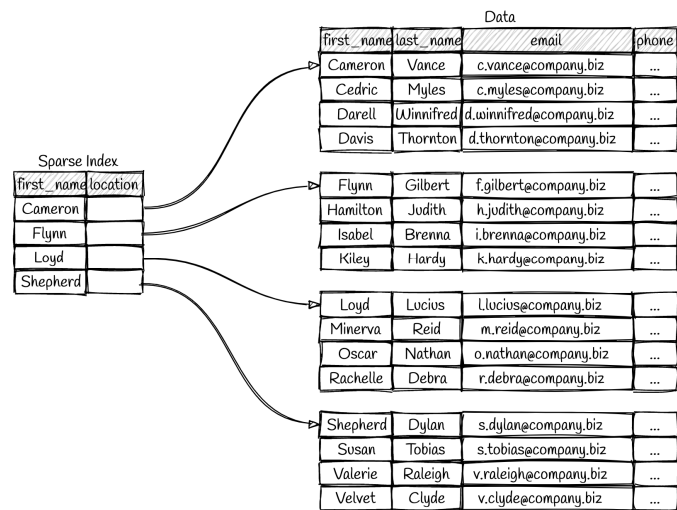


As we can see from the above image, a dense index is a strongly mapped index. Where all records are referenced to an index or key.

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.

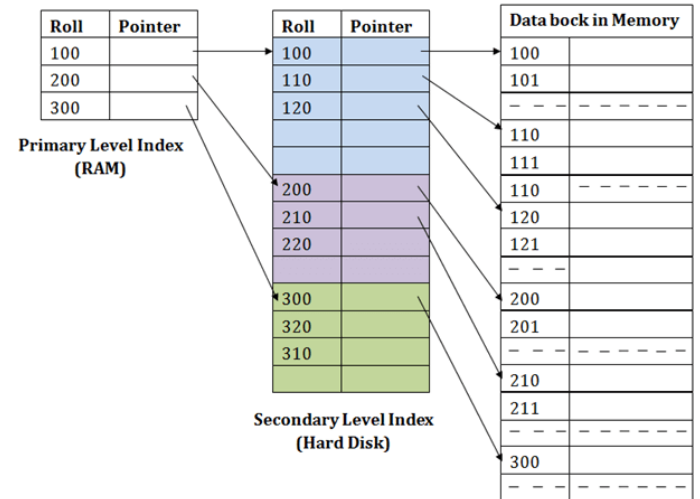


In real-world applications, we may encounter a lot of utility to Sparse index. If data size is too huge to process, we can use sparse indexing.

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 $\max(111) \leq 111$ 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.

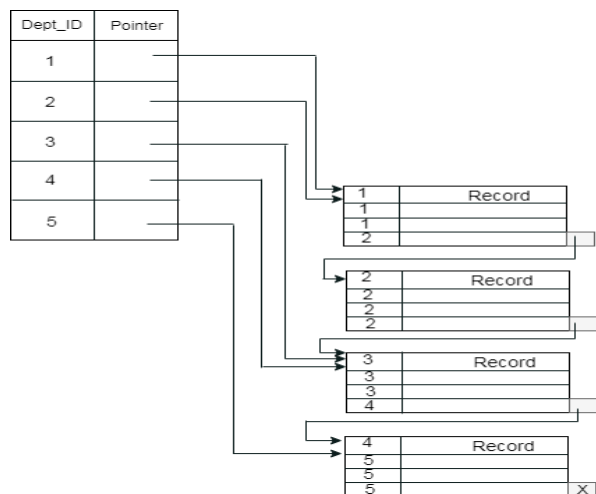
3.3.3.3 Cluster Indexing

- 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.

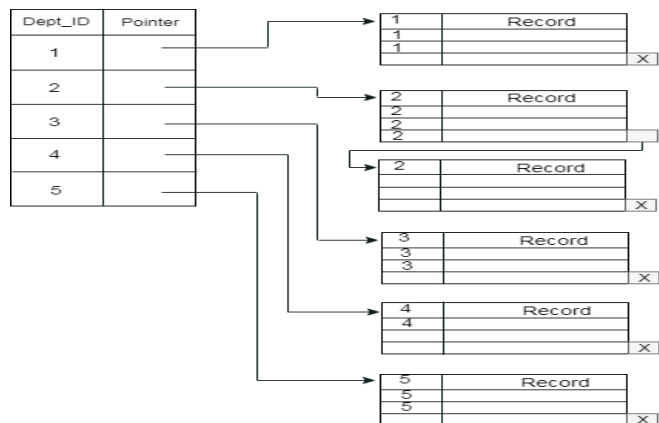
Example

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.



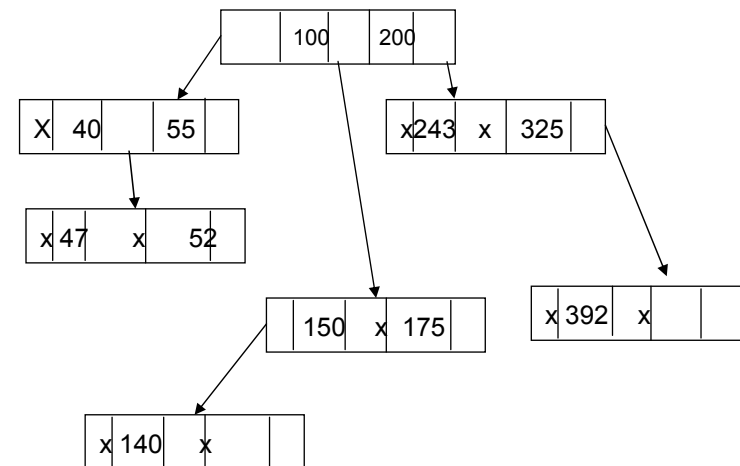
The previous schema is a little confusing because one disk block is shared by records which belong to the different cluster. If we use separate disk blocks for separate clusters, then it is called a better technique.



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 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	$D \log_r(1.5P)$	$D \log_r(1.5P)$ + matching pages	Search + D	Search + D
Unclustered Tree Index	PDR + Read index	D + $D \log_r(0.15P)$	$D \log_r(\text{index size})$ + $D * \text{matching records}$	3D + $D \log_r(\text{index size})$	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 is $D\log_F(1.5P)$.

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 $D\log_F(\text{index size}) + D \times \text{matching records}$.

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

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	VISIBILITY	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

File Organization and Indexing Database Management Systems 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.