Unit- 7: Developing Stored Procedures, DML Triggers and Indexing

What is a procedure in SQL?

- A procedure in SQL (often referred to as **stored procedure**), is a reusable unit that encapsulates the specific business logic of the application. A SQL procedure is a group of SQL statements and logic, compiled and stored together to perform a specific task,
- Named SQL block that optionally accepts some input in the form of parameters, performs some task and may or may not returns a value is called stored procedure.

Advantages of procedures in SQL

- The main purpose of stored procedures in SQL is to hide direct SQL queries from the code and improve the performance of database operations such as select, update, and delete data. Other advantages of procedure in SQL are:
  - Reduces the amount of information sent to the database server. It can become a more important benefit when the bandwidth of the network is less.
  - Enables the reusability of code
  - Enhances the security since you can grant permission to the user for executing the stored procedure instead of giving permission on the tables used in the stored procedure.
  - Support nested procedure calls to other SQL procedures or procedures implemented in other languages.
  - Stored procedure has a unique named block of code which is compiled and stored in the database

PL/SQL

- PL/SQL is a combination of SQL along with the procedural features of programming languages. It was developed by Oracle Corporation in the early 90's to enhance the capabilities of SQL
- A PL/SQL procedure is a reusable unit that encapsulates specific business logic of the application. Technically speaking, a PL/SQL procedure is a named block stored as a schema object in the Oracle Database.

Managing Stored Procedures using PL/SQL

- The PL/SQL stored procedure or simply a procedure is a PL/SQL block which performs one or more specific tasks. It is just like procedures in other programming languages.
- A procedure has a header and a body. The header consists of the name of the procedure and the parameters or variables passed to the procedure. The body consists or declaration section, execution section and exception section similar to a general PL/SQL Block.
- A procedure is similar to an anonymous PL/SQL block but it is named for repeated usage

Parts of a PL/SQL Procedure

Each PL/SQL subprogram has a name, and may also have a parameter list. Like anonymous PL/SQL blocks, the named blocks will also have the following three parts –

S.No	Parts & Description
1	**Declarative Part** It is an optional part. However, the declarative part for a subprogram does not start with the DECLARE keyword. It contains declarations of types, cursors, constants, variables, exceptions, and nested subprograms. These items are local to the subprogram and cease to exist when the subprogram completes execution.
2	**Executable Part** This is a mandatory part and contains statements that perform the designated action.
3	**Exception-handling part** This is again an optional part. It contains the code that handles the exceptions (run time errors).

Creating procedure

The following illustrates the basic syntax of creating a procedure in PL/SQL:

```
CREATE [OR REPLACE ] PROCEDURE procedure_name (parameter_list)
IS
[declaration statements]
BEGIN
[execution statements]
EXCEPTION
[exception handler]
END [procedure_name ];
```

PL/SQL procedure header

- A procedure begins with a header that specifies its name and an optional parameter list.
- Each parameter can be in either IN, OUT, or INOUT mode. The parameter mode specifies whether a parameter can be read from or write to. IN represents the value that will be passed from outside and OUT represents the parameter that will be used to return a value outside of the procedure.

IN	OUT	INOUT
An IN parameter is read-only. You can reference an IN parameter inside a procedure, but you cannot change its value. Oracle uses IN as the default mode. It means that if you don't specify the mode for a parameter explicitly, Oracle will use the IN mode.	An OUT parameter is writable. Typically, you set a returned value for the OUT parameter and return it to the calling program. Note that a procedure ignores the value that you supply for an OUT parameter.	An INOUT parameter is both readable and writable. The procedure can read and modify it.

- Note that OR REPLACE option allows you to overwrite the current procedure with the new code.

PL/SQL procedure body

The procedure body has three parts. The executable part is mandatory whereas the declarative and exception-handling parts are optional. The executable part must contain at least one executable statement.

Example of creating a procedure

In this example, we are going to insert record in **user (id, name)** table.

Now write the procedure code to insert record in user table

```
CREATE OR REPLACE PROCEDURE proc_insert_user
(id IN NUMBER, name IN VARCHAR2)
IS
BEGIN
insert into user values(id,name);
END ;
/
```

Executing a PL/SQL procedure

The following shows the syntax for executing a procedure:

```
EXECUTE procedure_name( arguments);
Or
EXEC procedure_name( arguments);
```

We can also call a PL/SQL stored procedure using an Anonymous PL/SQL block or using Named PL/SQL block.

```
BEGIN
  procedure_name;
END;
/
```

ALTER PROCEDUR

The **ALTER PROCEDURE** statement allows changes to be made to an existing stored procedure. Depending on the vendor, the kind and degree of change varies widely.

In Oracle, this command simply recompiles a PL/SQL stored procedure, but does not allow the code to be changed. Instead, use the Oracle command CREATE OR REPLACE PROCEDURE to achieve the same functionality.

The PL/SQL ALTER PROCEDURE statement is used to explicitly recompile a standalone stored procedure. Explicit recompilation eliminates the need for implicit run-time recompilation and prevents associated run-time compilation errors and performance overhead.

Note: This statement does not change the declaration or definition of an existing procedure.

Syntax:

```
ALTER PROCEDURE procedure_name
COMPILE;
```

Removing a procedure

To delete a procedure, you use the DROP PROCEDURE followed by the procedure's name that you want to drop as shown in the following syntax:

```
DROP PROCEDURE procedure_name;
```

Passing Parameters to Stored Procedure

We can pass parameter to the stored procedure as in following example

```
CREATE OR REPLACE PROCEDURE emp_sal( dep_id NUMBER, sal_raise NUMBER)
IS
BEGIN
 UPDATE emp SET salary = salary * sal_raise WHERE department_id = dep_id;
END;
/
```

This procedure will accept two parameters which is the department id and the numeric value for salary raise. First parameter which is the dep_id, is used to determine the ID of the department. The second parameter which is sal _ raise will become the multiplication factor in the salary raise.

In PL/SQL, we can pass parameters to procedures and functions in three ways.

1. **IN type parameter:** These types of parameters are used to send values to stored procedures.
2. **OUT type parameter:** These types of parameters are used to get values from stored procedures. This is similar to a return type in functions.
3. **IN OUT parameter:** These types of parameters are used to send values and get values from stored procedures.

NOTE: If a parameter is not explicitly defined a parameter type, then by default it is an IN type parameter.

Default Parameters

- We can specify default values for parameters of stored procedure by using DEFAULT keyword.
- When we want to create a PL/SQL procedure that accepts several parameters and some of those parameters should be made optional and contain default values, we should specify default values in stored procedure
- A default value cannot be assigned to the OUT parameter.
- The procedure caller can omit the parameters if default values are declared for the variables within the procedure.

- The ability to provide a default value for a variable declaration is optional. To do so, you must provide the declaration of the variable with the keyword DEFAULT followed by the value
- If a default value is declared, then you needn't specify a value for the parameter when the function or procedure is called. If you do specify a value for a parameter that has a default value, the specified value overrides the default.

Example:

```
CREATE OR REPLACE PROCEDURE update_depart_emp_sal ( dep_id IN  NUMBER, sal_raise NUMBER DEFAULT .05 )
IS
BEGIN
 UPDATE emp SET salary = salary * sal_raise WHERE department_id = dep_id;
END;
/
```

Here Default value for sal_raise is 0.5

Returning data from stored procedure

We can return data from stored procedure using OUT parameter.

Example:

```
CREATE OR REPLACE PROCEDURE P_GET_SAL (P_EMPID NUMBER, P_SAL OUT NUMBER)
IS
BEGIN
SELECT SALARY INTO P_SAL
FROM EMPLOYEE
```

```
WHERE  EMPLOYEE_ID=P_EMPID;
END;
```

Functionally, the code above may be used as in this example:

SQL> VAR G_SAL NUMBER;

SQL> EXEC P_GET_SAL(100, :G_SAL);

PL/SQL procedure successfully completed.

SQL> PRINT G_SAL;

Using the Return Statement

- The RETURN statement is used to unconditionally and immediately terminate an SQL procedure by returning the flow of control to the caller of the stored procedure.
- In PL/SQL code the RETURN keyword can appear in any subprogram as a statement. As soon as it is encountered in the subprogram body, Oracle server immediately sends the execution control back to the immediate calling code or host environment. No further statements are processed in the block following the RETURN statement.
- In the example below, a procedure named "P_RET_VAL" is created which returns the control to the calling environment based on the value of P_EMPNO. It returns if it is NULL.

```
CREATE OR REPLACE PROCEDURE P_RET_VAL(P_EMPNO NUMBER DEFAULT NULL)
IS

BEGIN

IF P_EMPNO IS NULL THEN
 RETURN;
ELSE
 DBMS_OUTPUT.PUT_LINE(P_EMPNO);
END IF;
END;
```

Note:

As a clause, RETURN is a mandatory element in a function definition. It defines the data type of the value returned by the function.

```
CREATE OR REPLACE FUNCTION totalCustomers
RETURN number
 IS
   total NUMBER(2) := 0;
BEGIN
   SELECT count(*) INTO total
   FROM customers;
    RETURN total;
END;
/
```

Calling function

```
DECLARE
   c NUMBER(2);
BEGIN
   c := totalCustomers();
   dbms_output.put_line('Total no. of Customers: ' || c);
END;
/
```

Triggers

- A **trigger** is a statement that is executed automatically by the system as a side effect of a modification to the database.
- To design a trigger mechanism, we must:
    - Specify the conditions under which the trigger is to be executed.
    - Specify the actions to be taken when the trigger executes.
- invokes it—that is, the trigger fires—whenever its triggering event occurs. While a trigger is disabled, it does not fire.
- In Oracle, trigger is a named PL/SQL block stored in the database and executed automatically when a triggering event takes place. The event can be any of the following:

- A data manipulation language (DML) statement executed against a table e.g., INSERT, UPDATE, or DELETE. For example, if you define a trigger that fires before an INSERT statement on the customers table, the trigger will fire once before a new row is inserted into the customers table.
- A data definition language (DDL) statement executes e.g., CREATE or ALTER statement. These triggers are often used for auditing purposes to record changes of the schema.
- A system event such as startup or shutdown of the Oracle Database.
- A user event such as login or logout.

Like a stored procedure, a trigger is a named PL/SQL unit that is stored in the database and can be invoked repeatedly. Unlike a stored procedure, you can enable and disable a trigger, but you cannot explicitly invoke it. While a trigger is enabled, the database automatically

Triggers are useful in many cases such as the following:

- Enforcing complex business rules that cannot be established using integrity constraint such as UNIQUE, NOT NULL, and CHECK.
- Preventing invalid transactions.
- Gathering statistical information on table accesses.
- Generating value automatically for derived columns.
- Auditing sensitive data.

Creating a trigger in Oracle

To create a new trigger in Oracle, you use the following CREATE TRIGGER statement:

```
CREATE [OR REPLACE] TRIGGER trigger_name

{BEFORE | AFTER } triggering_event ON table_name

[FOR EACH ROW]

[FOLLOWS | PRECEDES another_trigger]

[ENABLE / DISABLE ]

[WHEN condition]

DECLARE

    declaration statements

BEGIN

    executable statements

EXCEPTION

    exception_handling statements

END;
```

- The **CREATE** keyword specifies that you are creating a new trigger. The OR REPLACE keywords are optional. They are used to modify an existing trigger.
  - Even though the **OR REPLACE** keywords are optional, they appear with the CREATE keyword in most cases.
  - **CREATE OR REPLACE** keywords will replace an existing trigger if it already exists and create a new trigger if the trigger does not
- **Trigger name** Specify the name of the trigger that you want to create after the CREATE OR REPLACE keywords.
- The **BEFORE** or **AFTER** option specifies when the trigger fires, either before or after a triggering event e.g., INSERT, UPDATE, or DELETE
- The **table_name** is the name of the table associated with the trigger.

- The clause **FOR EACH ROW** specifies that the trigger is a row-level trigger. A row-level trigger fires once for each row inserted, updated, or deleted.
  - Besides the row-level triggers, we have statement-level triggers. A statement-trigger fire once regardless of the number of rows affected by the triggering event. If you omit the FOR EACH ROW clause, the CREATE TRIGGER statement will create a statement-level trigger
- The **ENABLE / DISABLE** option specifies whether the trigger is created in the enabled or disabled state. Note that if a trigger is disabled, it is not fired when the triggering event occurs.
  - By default, if you don't specify the clause ENABLE / DISABLE , the trigger is created with the enabled state.
- For each triggering event e.g., INSERT, UPDATE, or DELETE, you can define multiple triggers to fire. In this case, you need to specify the firing sequence using the **FOLLOWS** or **PRECEDES** option.

Types of Triggers in Oracle

Triggers can be classified based on the following parameters.

Classification based on the timing

- **BEFORE Trigger**: It fires before the specified event has occurred.
- **AFTER Trigger**: It fires after the specified event has occurred.
- **INSTEAD OF Trigger**: An INSTEAD OF trigger is a trigger that allows you to update data in tables via their view which cannot be modified directly through DML statements.

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)

:NEW and :OLD Clause

- In a row level trigger, the trigger fires for each related row. And sometimes it is required to know the value before and after the DML statement.
- Oracle has provided two clauses in the RECORD-level trigger to hold these values. We can use these clauses to refer to the old and new values inside the trigger body.
- **:NEW –** It holds a new value for the columns of the base table/view during the trigger execution
- **:OLD –** It holds old value of the columns of the base table/view during the trigger execution
- This clause should be used based on the DML event. Below table will specify which clause is valid for which DML statement (INSERT/UPDATE/DELETE).

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

Enabling and Disabling Triggers

Triggers can be enabled or disabled. To enable or disable the trigger, an ALTER (DDL) statement needs to be given for the trigger that disable or enable it.

Below are the syntax for enabling/disabling the triggers.

```
ALTER TRIGGER <trigger_name> [ENABLE|DISABLE];
ALTER TABLE <table_name> [ENABLE|DISABLE] ALL TRIGGERS;
```

- The first syntax shows how to enable/disable the single trigger.
- The second statement shows how to enable/disable all the triggers on a particular table.

Limitations of Triggers

- Hard to maintain since this may be a possibility that the new developer doesn't able to know about the trigger defined in the database and wonder how data is inserted, deleted or updated automatically.
- They are hard to debug since they are difficult to view as compared to stored procedures, views, functions, etc.
- Excessive or over use of triggers can slow down the performance of the application since if we defined the triggers in many tables then they kept automatically executing every time data is inserted, deleted or updated in the tables (based on the trigger's definition) and it makes the processing very slow.
- If complex code is written in the triggers, then it will slow down the performance of the applications.
- The cost of creation of triggers can be more on the tables on which frequency of DML (insert, delete and update) operation like bulk insert is high.
- No transaction control statements are allowed in a trigger. ROLLBACK, COMMIT, and SAVEPOINT cannot be used.

Multi Row Enabled Trigger

- Row-level triggers fires once for each row affected by the triggering event such as INSERT, UPDATE, or DELETE.
- To create a new row-level trigger, we use the CREATE TRIGGER statement with the FOR EACH ROW clause.
- Row Level Trigger is fired each time row is affected by Insert, Update or Delete command. If statement doesn't affect any row, no trigger action happens.
- A trigger if specified FOR EACH ROW; it is fired for each of the table being affected by the triggering statement. For example if a trigger needs to be fired when rows of a table are deleted, it will be fired as many times the rows are deleted.

If FOR EACH ROW is not specified, it is application to a statement and the trigger is executed at a statement level.

```
CREATE OR REPLACE TRIGGER trigger_name
    BEFORE | AFTER
    INSERT OR DELETE OR UPDATE OF column1, column2, …
    ON table_name
    FOR EACH ROW
    REFERENCING OLD AS old_name
    NEW AS new_name
    WHEN (condition)
DECLARE
    …
BEGIN
    …
EXCEPTION
    …
END;
```

Statement Level Trigger

- A statement-level trigger is fired whenever a trigger event occurs on a table regardless of how many rows are affected. In other words, a statement-level trigger executes once for each transaction.
- For example, if you update 1000 rows in a table, then a statement-level trigger on that table would only be executed once.
- Due to its features, a statement-level trigger is not often used for data-related activities like auditing the changes of the data in the associated table. It's typically used to enforce extra security measures on the kind of transaction that may be performed on a table.
- By default, the statement CREATE TRIGGER creates a statement-level trigger when you omit the FOR EACH ROW clause.

Row Level Triggers	**Statement Level Triggers**
Row level triggers executes once for each and every row in the transaction.	Statement level triggers executes only once for each single transaction.
Specifically used for data auditing purpose.	Used for enforcing all additional security on the transactions performed on the table.
"FOR EACH ROW" clause is present in CREATE TRIGGER command.	"FOR EACH ROW" clause is omitted in CREATE TRIGGER command.
Example: If 1500 rows are to be inserted into a table, the row level trigger would execute 1500 times.	Example: If 1500 rows are to be inserted into a table, the statement level trigger would execute only once

Basic Concept of Indexing

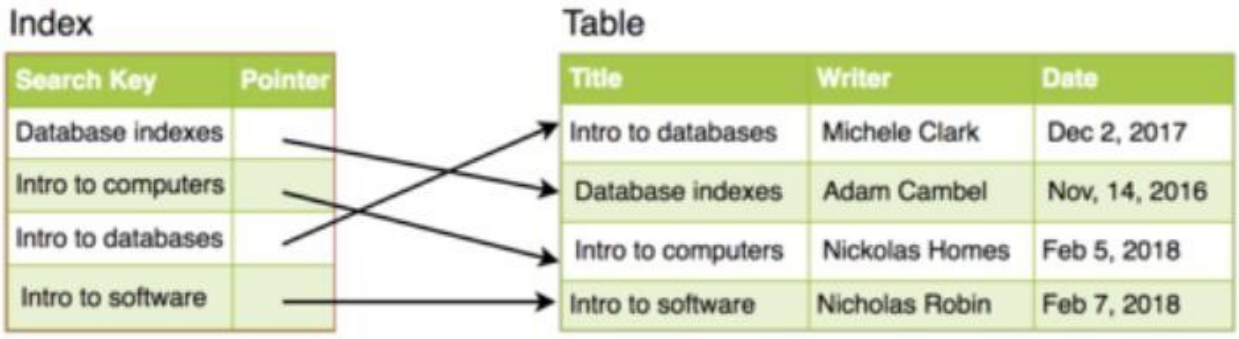
- Indexing is used to optimize the performance of a database by minimizing the number of disk accesses required when a query is processed.
- The index is a type of data structure. It is used to locate and access the data in a database table quickly.
- Concept is similar to index in book. An index in a database is very similar to an index in the back of a book.
- Indexing is a data structure technique to efficiently retrieve records from the database files based on some attributes on which the indexing has been done.
- An index is a copy of selected columns of data, from a table, that is designed to enable efficient search. An index normally includes a "key" or direct link to the original row of data from which it was copied, to allow the complete row to be retrieved efficiently.

Structure of an index

Search key	Data Reference

- Indexes can be created using some database columns. The first column of the database is the search key that contains a copy of the primary key or candidate key of the table.

- The second column of the database is the data reference. It contains a set of pointers holding the address of the disk block where the value of the particular key can be found.
- The keys are a fancy term for the values we want to look up in the index. The keys are based on the tables' columns. By comparing keys to the index it is possible to find one or more database records with the same value.



Ordered Indices

- The indices are usually sorted to make searching faster. The indices which are sorted are known as ordered indices.

  If the indexes are sorted, then it is called as ordered indices.

- **Example:** Suppose we have an *student* table with thousands of record and each of which is 10 bytes long. If their IDs start with 1, 2, 3. and so on and we have to search student with ID-543.
  - In the case of a database with no index, we have to search the disk block from starting till it reaches 543. The DBMS will read the record after reading 542*10=5420 bytes.
  - In the case of an index, we will search using indexes and the DBMS will read the record after reading 542*2= 1084 bytes which are very less compared to the previous case.
- Ordered Indexing is of two types −
  - Dense Index
  - Sparse Index

Dense Index

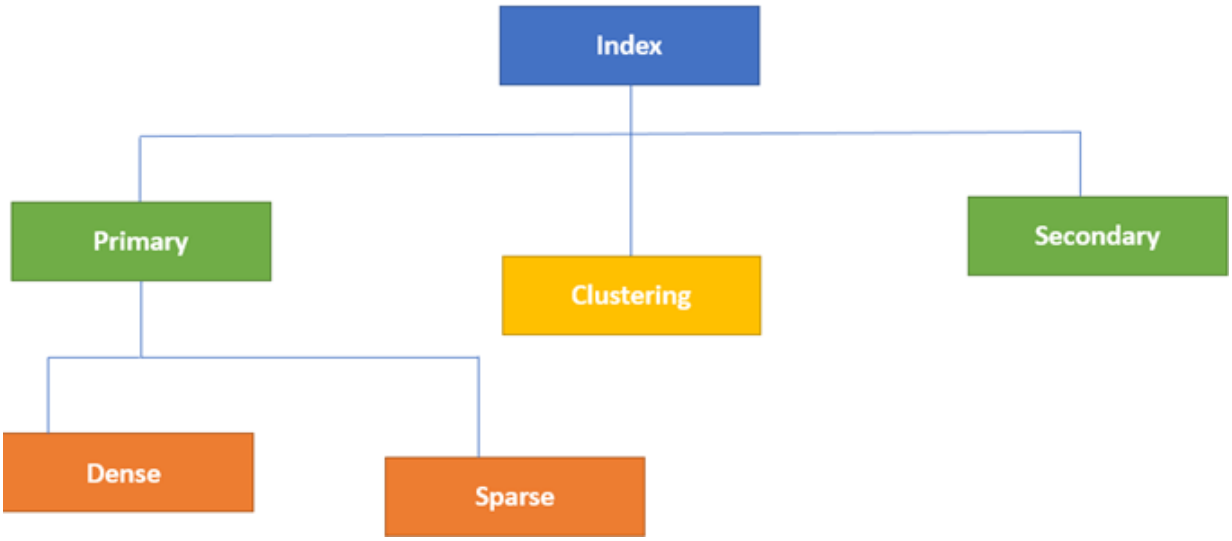
- For every search key value in the data file, there is an index record.
- This record contains the search key and also a reference to the first data record with that search key value
- In this, the number of records in the index table is same as the number of records in the main table.
- It needs more space to store index record itself. The index records have the search key and a pointer to the actual record on the disk.

- For every search values in a data file, there is an index record, hence the name is Dense Index.

Sparse Index

- In sparse index, index records are not created for every search key.
- An index record here contains a search key and an actual pointer to the data on the disk.
- To search a record, we first proceed by index record and reach at the actual location of the data. If the data we are looking for is not where we directly reach by following the index, then the system starts sequential search until the desired data is found.
- For few search values in a data file, there are index records, hence the name is Sparse Index.

Types of Indexing



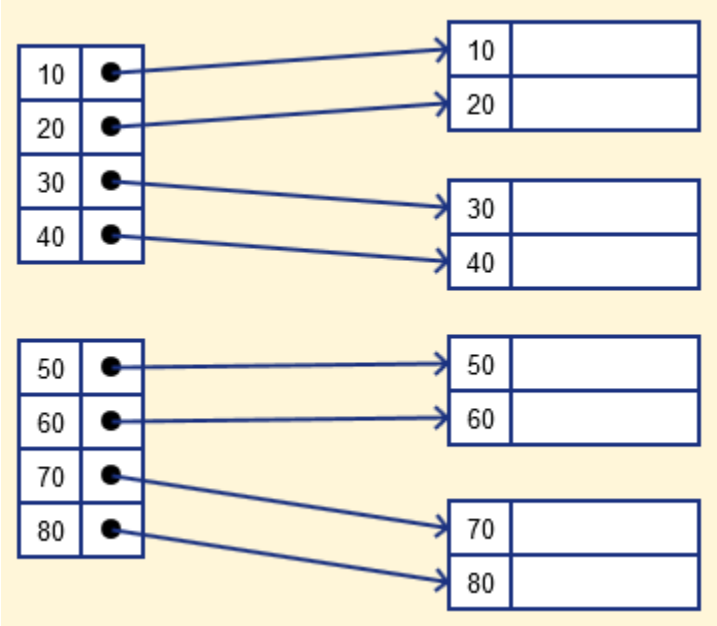
- Indexing is defined based on its indexing attributes. Indexing can be of the following types
  - **Primary Index** − Primary index is defined on an ordered data file. The data file is ordered on a key field. The key field is generally the primary key of the relation.
  - **Secondary Index** − Secondary index may be generated from a field which is a candidate key and has a unique value in every record, or a non-key with duplicate values. In secondary indexing, to reduce the size of mapping, another level of indexing is introduced.
  - **Clustering Index** − Clustering index is defined on an ordered data file. The data file is ordered on a non-key field.

Primary Index

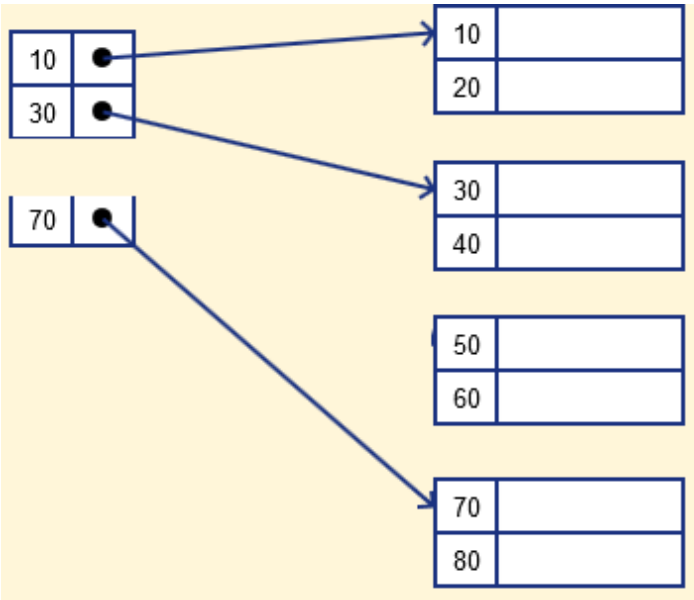
- If the index is created on the basis of the primary key of the table, then it is known as primary indexing.
- Primary Index is an ordered file which is fixed length size with two fields. The first field is the same a primary key and second, filed is pointed to that specific data block.
- In the Primary Index, there is always one to one relationship between the entries in the index table.
- The primary index can be classified into two types: Dense index and Sparse index.

- 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.

Dense Index Example

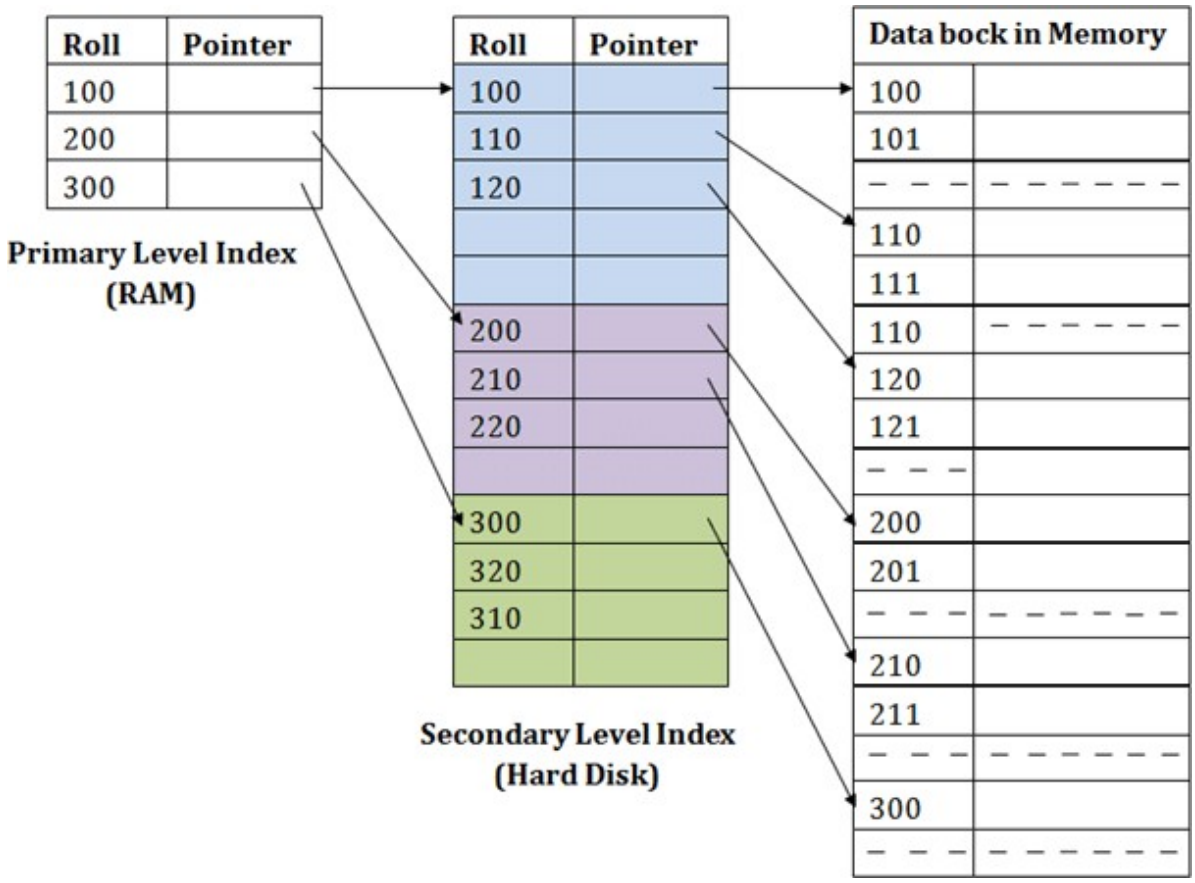


Sparse Index Example



Secondary Index

- In the sparse indexing, as the size of the table grows, the size of mapping also grows. These mappings are usually kept in the primary memory so that address fetch should be faster. Then the secondary memory searches the actual data based on the address got from mapping. If the mapping size grows then fetching the address itself becomes slower. In this case, the sparse index will not be efficient. To overcome this problem, secondary indexing is introduced.
- In secondary indexing, to reduce the size of mapping, another level of indexing is introduced. In this method, the huge range for the columns is selected initially so that the mapping size of the first level becomes small. Then each range is further divided into smaller ranges. The mapping of the first level is stored in the primary memory, so that address fetch is faster. The mapping of the second level and actual data are stored in the secondary memory (hard disk).
- For example, If you want to find the record of roll 111 in the following diagram,
  - It will search the highest entry which is smaller than or equal to 111 in the first level index. It will get 100 at this level.
  - Then in the second index level, again it does max (111) <= 111 and gets 110. Now using the address 110, it goes to the data block and starts searching each record till it gets 111.

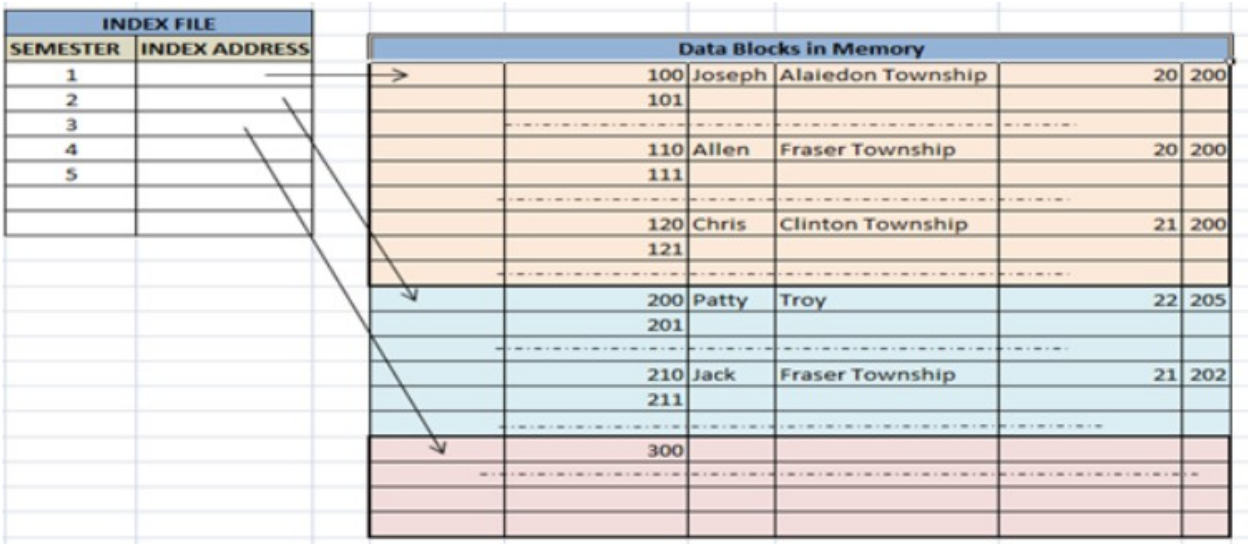


Clustering Index

- In some cases, the index is created on non-primary key columns which may not be unique for each record. In such cases, in order to identify the records faster, we will group two or more columns together to get the unique values and create index out of them. This method is known as clustering index. Basically, records with similar characteristics are grouped together and indexes are created for these groups.
- For example, students studying in each semester are grouped together. i.e.; 1st Semester students, 2nd semester students, 3rd semester students etc are grouped.
- In the diagram below we can see that, indexes are created for each semester in the id file. In the data block, the students of each semester are grouped together to form the cluster. The address in the index file points to the beginning of each cluster. In the data blocks, requested student ID is then search in sequentially.

——New records are inserted into the clusters based on their group. In above case, if a new student joins 3rd semester, then his record is inserted into the semester 3 cluster in the secondary memory. Same is done with update and delete.

——This method of file organization is better compared to other methods as it provides better distribution of records, and hence making search easier and faster. But in each cluster, there would be unused space left. Hence it will take more memory compared to other methods.



Multiple key access

——We have assumed implicitly that only one index is used to process a query on a relation. However, for certain types of queries, it is advantageous to use multiple indices if they exist.

——Index can be created with combined / composite search keys (i.e. having more than one column).

Creating and Dropping index

——The CREATE INDEX statement is used to create indexes in tables.

——To create a new index for a table, you use the `CREATE INDEX` statement as follows:

```
CREATE INDEX index_name
ON table_name(column1[,column2,...])
```

In this syntax:

- First, specify the name of the index. The index name should be meaningful and includes table alias and column name(s) where possible, along with the suffix _I such as:
- <table_name>_<column_name>_I
- Second, specify the name of the table followed by one or more indexed columns surrounded by parentheses.

Creating an index on one column example

Suppose, you often want to look up members by the last name and you find that the query is quite slow. To speed up the lookup, you create an index for the last_name column:

```
CREATE INDEX members_last_name_i
ON members(last_name);
```

Now, showing the indexe,

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

Creating an index on multiple columns example

The following example creates an index on both last name and first name columns:

```
CREATE INDEX members_name_i

ON members(last_name,first_name);
```

Removing an index

——To remove an index, you use the DROP INDEX statement:

```
DROP INDEX index_name;
```

For example, to drop the members_last_name_i index, you use the following statement:

```
DROP INDEX members_last_name_i;
```