Structured Query Language (SQL)

Structured Query Language is a standard Database language which is used to create, maintain and retrieve the relational database. Following are some interesting facts about SQL.

SQL is case insensitive. But it is a recommended practice to use keywords (like SELECT, UPDATE, CREATE, etc) in capital letters and use user defined things (liked table name, column name, etc) in small letters.

We can write comments in SQL using "–" (double hyphen) at the beginning of any line.

SQL is the programming language for relational databases (explained below) like MySQL, Oracle, Sybase, SQL Server etc. Other non-relational databases (also called NoSQL) databases like MongoDB, DynamoDB, etc do not use SQL

Although there is an ISO standard for SQL, most of the implementations slightly vary in syntax. So, we may encounter queries that work in SQL Server but do not work in MySQL.

What is Relational Database?

Relational database means the data is stored as well as retrieved in the form of relations (tables). Table 1 shows the relational database with only one relation called STUDENT which stores ROLL_NO, NAME, ADDRESS, PHONE and AGE of students.

ROLL_NO	**NAME**	**ADDRESS**	**PHONE**	**AGE**
1	RAM	DELHI	9455123451	18
2	RAMESH	GURGAON	9652431543	18
3	SUJIT	ROHTAK	9156253131	20
4	SURESH	DELHI	9156768971	18

TABLE 1

These are some important terminologies that are used in terms of relation.

Attribute: Attributes are the properties that define a relation. e.g.; **ROLL_NO**, **NAME** etc.

Tuple: Each row in the relation is known as tuple. The above relation contains 4 tuples, one of which is shown as:

1	RAM	DELHI	9455123451	18

Degree: The number of attributes in the relation is known as degree of the relation.
The **STUDENT** relation defined above has degree 5.

Cardinality: The number of tuples in a relation is known as cardinality. The **STUDENT** relation defined above has cardinality 4.

Column: Column represents the set of values for a particular attribute. The column **ROLL_NO** is extracted from relation STUDENT.

ROLL_NO
1
2
3

4

The queries to deal with relational database can be categories as:

Data Definition Language: It is used to define the structure of the database. e.g; CREATE TABLE, ADD COLUMN, DROP COLUMN and so on.

Data Manipulation Language: It is used to manipulate data in the relations. e.g.; INSERT, DELETE, UPDATE and so on.

Data Query Language: It is used to extract the data from the relations. e.g.; SELECT

So first we will consider the Data Query Language. A generic query to retrieve from a relational database is:

1. **SELECT** [**DISTINCT**] Attribute_List **FROM** R1,R2….RM
2. [**WHERE** condition]
3. [**GROUP BY** (Attributes)[**HAVING** condition]]
4. [**ORDER BY**(Attributes)[**DESC**]];

Part of the query represented by statement 1 is compulsory if you want to retrieve from a relational database. The statements written inside [] are optional. We will look at the possible query combination on relation shown in Table 1.

Case 1: If we want to retrieve attributes **ROLL_NO** and **NAME** of all students, the query will be:

SELECT ROLL_NO, NAME **FROM** STUDENT;

ROLL_NO	**NAME**
1	RAM
2	RAMESH
3	SUJIT
4	SURESH

Case 2: If we want to retrieve **ROLL_NO** and **NAME** of the students whose **ROLL_NO** is greater than 2, the query will be:

SELECT ROLL_NO, NAME **FROM** STUDENT

WHERE ROLL_NO>2;

ROLL_NO	**NAME**
3	SUJIT
4	SURESH

CASE 3: If we want to retrieve all attributes of students, we can write * in place of writing all attributes as:

SELECT * FROM STUDENT

WHERE ROLL_NO>2;

ROLL_NO	**NAME**	**ADDRESS**	**PHONE**	**AGE**

3	SUJIT	ROHTAK	9156253131	20
4	SURESH	DELHI	9156768971	18

CASE 4: If we want to represent the relation in ascending order by **AGE**, we can use ORDER BY clause as:

SELECT * FROM STUDENT **ORDER BY** AGE;

ROLL_NO	**NAME**	**ADDRESS**	**PHONE**	**AGE**
1	RAM	DELHI	9455123451	18
2	RAMESH	GURGAON	9652431543	18
4	SURESH	DELHI	9156768971	18
3	SUJIT	ROHTAK	9156253131	20

Note: ORDER BY **AGE** is equivalent to ORDER BY **AGE** ASC. If we want to retrieve the results in descending order of **AGE**, we can use ORDER BY **AGE** DESC.

CASE 5: If we want to retrieve distinct values of an attribute or group of attribute, DISTINCT is used as in:

SELECT DISTINCT ADDRESS **FROM** STUDENT;

ADDRESS
DELHI
GURGAON
ROHTAK

If DISTINCT is not used, DELHI will be repeated twice in result set. Before understanding GROUP BY and HAVING, we need to understand aggregations functions in SQL.

AGGRATION FUNCTIONS: Aggregation functions are used to perform mathematical operations on data values of a relation. Some of the common aggregation functions used in SQL are:

- **COUNT:** Count function is used to count the number of rows in a relation. e.g;

SELECT COUNT (PHONE) **FROM** STUDENT;

COUNT(PHONE)
4

- **SUM:** SUM function is used to add the values of an attribute in a relation. e.g;

SELECT SUM (AGE) **FROM** STUDENT;

SUM(AGE)
74

In the same way, MIN, MAX and AVG can be used. As we have seen above, all aggregation functions return only 1 row.

AVERAGE: It gives the average values of the tupples. It is also defined as sum divided by count values.
Syntax:AVG(attributename)
OR
Syntax:SUM(attributename)/COUNT(attributename)
The above mentioned syntax also retrieves the average value of tupples.

MAXIMUM:It extracts the maximum value among the set of tupples.
Syntax:MAX(attributename)

MINIMUM:It extracts the minimum value amongst the set of all the tupples.
Syntax:MIN(attributename)

GROUP BY: Group by is used to group the tuples of a relation based on an attribute or group of attribute. It is always combined with aggregation function which is computed on group. e.g.;

SELECT ADDRESS, **SUM**(AGE) **FROM** STUDENT

GROUP BY (ADDRESS);

In this query, SUM(**AGE**) will be computed but not for entire table but for each address. i.e.; sum of AGE for address DELHI(18+18=36) and similarly for other address as well. The output is:

ADDRESS	**SUM(AGE)**
DELHI	36
GURGAON	18
ROHTAK	20

If we try to execute the query given below, it will result in error because although we have computed SUM(AGE) for each address, there are more than 1 ROLL_NO for each address we have grouped. So it can't be displayed in result set. We need to use aggregate functions on columns after SELECT statement to make sense of the resulting set whenever we are using GROUP BY.

SELECT ROLL_NO, ADDRESS, **SUM**(AGE) **FROM** STUDENT

GROUP BY (ADDRESS);

NOTE: An attribute which is not a part of GROUP BY clause can't be used for selection. Any attribute which is part of GROUP BY CLAUSE can be used for selection but it is not mandatory. But we could use attributes which are not a part of the GROUP BY clause in an aggregate function.

Inner Join vs Outer Join

An SQL Join is used to combine data from two or more tables based on a common field between them. For example, consider the following two tables.

Student Table

EnrollNo	**StudentName**	**Address**
1001	geek1	geeksquiz1
1002	geek2	geeksquiz2
1003	geek3	geeksquiz3
1004	geek4	geeksquiz4

StudentCourse Table

CourseID	EnrollNo
1	1001
2	1001

3	1001
1	1002
2	1003

Inner Join / Simple join:

In an INNER join, it allows retrieving data from two tables with the same ID.

Syntax:

SELECT COLUMN1, COLUMN2 FROM

[TABLE 1] INNER JOIN [TABLE 2]

ON Condition;

The following is a join query that shows the names of students enrolled in different courseIDs.

```
SELECT StudentCourse.CourseID,Student.StudentName

FROM Student

INNER JOIN StudentCourse

ON StudentCourse.EnrollNo = Student.EnrollNo

ORDER BY StudentCourse.CourseID;
```

Note: INNER is optional above. Simple JOIN is also considered as INNER JOIN The above query would produce following result.

CourseID	StudentName
1	geek1
1	geek2
2	geek1
2	geek3
3	geek1

What is the difference between inner join and outer join?

Outer Join is of three types:

1. Left outer join
2. Right outer join
3. Full Join

1. Left outer join returns all rows of a table on the left side of the join. For the rows for which there is no matching row on the right side, the result contains NULL on the right side.

Syntax:

SELECT T1.C1, T2.C2

FROM TABLE T1

LEFT JOIN TABLE T2

ON T1.C1= T2.C1;

SELECT Student.StudentName,StudentCourse.CourseID

FROM Student

LEFT OUTER JOIN StudentCourse

ON StudentCourse.EnrollNo = Student.EnrollNo

ORDER BY StudentCourse.CourseID;

Note: OUTER is optional above. Simple LEFT JOIN is also considered as LEFT OUTER JOIN

StudentName	CourseID
geek4	*NULL*
geek2	1
geek1	1
geek1	2
geek3	2
geek1	3

2. Right Outer Join is similar to Left Outer Join (Right replaces Left everywhere).

Syntax:

SELECT T1.C1, T2.C2

FROM TABLE T1

RIGHT JOIN TABLE T2

ON T1.C1= T2.C1;

Example:

SELECT Student.StudentName, StudentCourse.CourseID

FROM Student

RIGHT OUTER JOIN StudentCourse

ON StudentCourse.EnrollNo = Student.EnrollNo

ORDER BY StudentCourse.CourseID;

3. Full Outer Join contains the results of both the Left and Right outer joins. It is also known as cross join. It will provide a mixture of two tables.

Syntax:

SELECT * FROM T1

CROSS JOIN T2;

Having vs Where Clause in SQL

The difference between the having and where clause in SQL is that the where clause cann*ot* be used with aggregates, but the having clause can.

The **where** clause works on row's data, not on aggregated data. Let us consider below table 'Marks'.

Student	Course	Score
a	c1	40
a	c2	50
b	c3	60
d	c1	70
e	c2	80

Consider the query

SELECT Student, Score **FROM** Marks **WHERE** Score >=40

This would select data row by row basis.

The **having** clause works on aggregated data.

For example, output of below query

SELECT Student, SUM(score) AS total **FROM** Marks **GROUP BY** Student

Student	Total
a	90
b	60
d	70
e	80

When we apply having in above query, we get

SELECT Student, SUM(score) AS total **FROM** Marks **GROUP BY** Student **HAVING** total > 70

Student	Total
a	90
e	80

Note: It is not a predefined rule but in a good number of the SQL queries, we use WHERE prior to GROUP BY and HAVING after GROUP BY. The Where clause acts as a **pre filter** where as Having as a **post filter.**

A **database object** is any defined object in a database that is used to store or reference data.Anything which we make from **create command** is known as Database Object.It can be used to hold and manipulate the data.Some of the examples of database objects are : view, sequence, indexes, etc.

- **Table –** Basic unit of storage; composed rows and columns

- **View –** Logically represents subsets of data from one or more tables
- **Sequence –** Generates primary key values
- **Index –** Improves the performance of some queries
- **Synonym –** Alternative name for an object

Different database Objects :

1. **Table –** This database object is used to create a table in database.

Syntax :

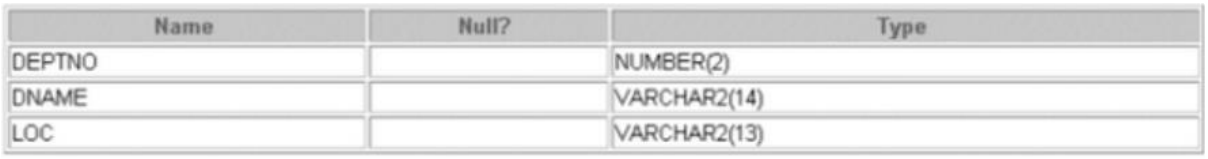
```
CREATE TABLE [schema.]table
          (column datatype [DEFAULT expr][, ...]);
```

Example :

```
CREATE TABLE dept
        (deptno NUMBER(2),
         dname VARCHAR2(14),
         loc VARCHAR2(13));
```

Output :

DESCRIBE dept;



Name	Null?	Type
DEPTNO		NUMBER(2)
DNAME		VARCHAR2(14)
LOC		VARCHAR2(13)

2. **View** – This database object is used to create a view in database.A view is a logical table based on a table or another view. A view contains no data of its own but is like a window through which data from tables can be viewed or changed. The tables on which a view is based are called base tables. The view is stored as a SELECT statement in the data dictionary.

Syntax :

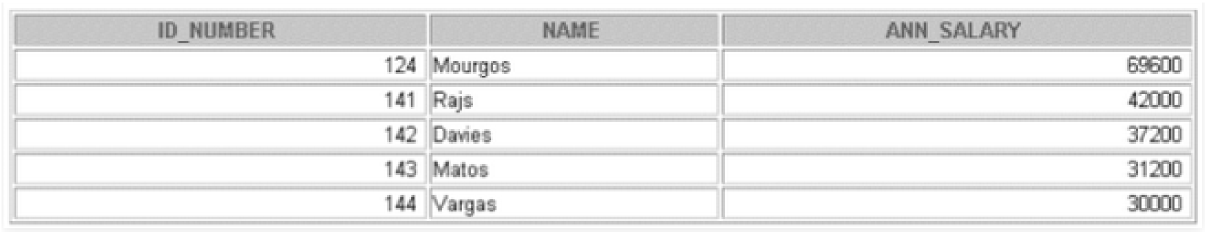
```
CREATE [OR REPLACE] [FORCE|NOFORCE] VIEW view
               [(alias[, alias]...)]
               AS subquery
               [WITH CHECK OPTION [CONSTRAINT constraint]]
               [WITH READ ONLY [CONSTRAINT constraint]];
```

Example :

```
i.    CREATE VIEW salvu50
ii.           AS SELECT employee_id ID_NUMBER, last_name NAME,
iii.          salary*12 ANN_SALARY
iv.           FROM employees
v.            WHERE department_id = 50;
```

Output :

```
SELECT *
FROM salvu50;
```



ID_NUMBER	NAME	ANN_SALARY
124	Mourgos	69600
141	Rajs	42000
142	Davies	37200
143	Matos	31200
144	Vargas	30000

3. **Sequence –** This database object is used to create a sequence in database.A sequence is a user created database object that can be shared by multiple users to generate unique integers. A typical usage for sequences is to create a primary key value, which must be unique for each row.The sequence is generated and incremented (or decremented) by an internal Oracle routine.

Syntax :

```
CREATE SEQUENCE sequence
        [INCREMENT BY n]
        [START WITH n]
        [{MAXVALUE n | NOMAXVALUE}]
        [{MINVALUE n | NOMINVALUE}]
        [{CYCLE | NOCYCLE}]
        [{CACHE n | NOCACHE}];
```

Example :

```
CREATE SEQUENCE dept_deptid_seq
          INCREMENT BY 10
          START WITH 120
          MAXVALUE 9999
          NOCACHE
```

```
                NOCYCLE;
```

Check if sequence is created by :

```
SELECT sequence_name, min_value, max_value,
               increment_by, last_number
               FROM   user_sequences;
```

4. **Index –** This database object is used to create a indexes in database.An Oracle server index is a schema object that can speed up the retrieval of rows by using a pointer.Indexes can be created explicitly or automatically. If you do not have an index on the column, then a full table scan occurs.

An index provides direct and fast access to rows in a table. Its purpose is to reduce the necessity of disk I/O by using an indexed path to locate data quickly. The index is used and maintained automatically by the Oracle server. Once an index is created, no direct activity is required by the user.Indexes are logically and physically independent of the table they index. This means that they can be created or dropped at any time and have no effect on the base tables or other indexes.

Syntax :

1. CREATE INDEX index
2. ON table (column[, column]...);

Example :

```
CREATE INDEX emp_last_name_idx
          ON  employees(last_name);
```

5. **Synonym –** This database object is used to create a indexes in database.It simplify access to objects by creating a synonym(another name for an object). With synonyms, you can Ease referring to a table owned by another user and shorten lengthy object names.To refer to a table owned by another user, you need to prefix the table name with the name of the user who created it followed by a period. Creating a synonym eliminates the need to qualify the object name with the schema and provides you with an alternative name for a table, view, sequence,procedure, or other objects. This method can be especially useful with lengthy object names, such as views.

In the syntax:
PUBLIC : creates a synonym accessible to all users
synonym : is the name of the synonym to be created
object : identifies the object for which the synonym is created

Syntax :

```
CREATE [PUBLIC] SYNONYM synonym FOR  object;
```

Example :

```
CREATE SYNONYM d_sum FOR dept_sum_vu;
```

Nested Queries in SQL

In nested queries, a query is written inside a query. The result of inner query is used in execution of outer query. We will use **STUDENT, COURSE, STUDENT_COURSE** tables for understanding nested queries.

STUDENT

S_ID	**S_NAME**	**S_ADDRESS**	**S_PHONE**	**S_AGE**
S1	RAM	DELHI	9455123451	18
S2	RAMESH	GURGAON	9652431543	18
S3	SUJIT	ROHTAK	9156253131	20
S4	SURESH	DELHI	9156768971	18

COURSE

C_ID	**C_NAME**
C1	DSA
C2	Programming
C3	DBMS

STUDENT_COURSE

S_ID	**C_ID**
S1	C1
S1	C3
S2	C1
S3	C2
S4	C2
S4	C3

There are mainly two types of nested queries:

- **Independent Nested Queries:** In independent nested queries, query execution starts from innermost query to outermost queries. The execution of inner query is independent of outer query, but the result of inner query is used in execution of outer query. Various operators like IN, NOT IN, ANY, ALL etc are used in writing independent nested queries.
- **IN:** If we want to find out **S_ID** who are enrolled in **C_NAME** 'DSA' or 'DBMS', we can write it with the help of independent nested query and IN operator. From **COURSE** table, we can find

out **C_ID** for **C_NAME** 'DSA' or DBMS' and we can use these **C_ID**s for finding **S_ID**s from **STUDENT_COURSE** TABLE.

STEP 1: Finding **C_ID** for **C_NAME** ='DSA' or 'DBMS'

Select **C_ID** from **COURSE** where **C_NAME** = 'DSA' or **C_NAME** = 'DBMS'

STEP 2: Using **C_ID** of step 1 for finding **S_ID**

Select **S_ID** from **STUDENT_COURSE** where **C_ID** IN

(SELECT **C_ID** from **COURSE** where **C_NAME** = 'DSA' or **C_NAME**='DBMS');

The inner query will return a set with members C1 and C3 and outer query will return those **S_ID**s for which **C_ID** is equal to any member of set (C1 and C3 in this case). So, it will return S1, S2 and S4.

Note: If we want to find out names of **STUDENT**s who have either enrolled in 'DSA' or 'DBMS', it can be done as:

Select S_NAME from **STUDENT** where **S_ID** IN

(Select **S_ID** from **STUDENT_COURSE** where **C_ID** IN

(SELECT **C_ID** from **COURSE** where **C_NAME**='DSA' or **C_NAME**='DBMS'));

NOT IN: If we want to find out **S_ID**s of **STUDENT**s who have neither enrolled in 'DSA' nor in 'DBMS', it can be done as:

Select **S_ID** from **STUDENT** where **S_ID** NOT IN

(Select **S_ID** from **STUDENT_COURSE** where **C_ID** IN

(SELECT **C_ID** from **COURSE** where **C_NAME**='DSA' or **C_NAME**='DBMS'));

The innermost query will return a set with members C1 and C3. Second inner query will return those **S_ID**s for which **C_ID** is equal to any member of set (C1 and C3 in this case) which are S1, S2 and S4. The outermost query will return those **S_ID**s where **S_ID** is not a member of set (S1, S2 and S4). So it will return S3.

- **Co-related Nested Queries:** In co-related nested queries, the output of inner query depends on the row which is being currently executed in outer query. e.g.; If we want to find out **S_NAME** of **STUDENT**s who are enrolled in **C_ID** 'C1', it can be done with the help of co-related nested query as:

Select S_NAME from **STUDENT** S where EXISTS(select * from **STUDENT_COURSE** SC where S.**S_ID**=SC.**S_ID** and SC.**C_ID**='C1');

For each row of **STUDENT** S, it will find the rows from **STUDENT_COURSE** where S.**S_ID** = SC.**S_ID** and SC.**C_ID**='C1'. If for a **S_ID** from **STUDENT** S, atleast a row exists in **STUDENT_COURSE** SC with **C_ID**='C1', then inner query will return true and corresponding **S_ID** will be returned as output.

Join operation Vs Nested query in DBMS

The growth of technology and automation coupled with exponential amounts of data has led to the importance and omnipresence of databases which, simply put, are organized collections of data. Considering a naive approach, one can theoretically keep all the data in one large table, however that increases the access time in searching for a record, security issues if the master table is destroyed, redundant storage of information and other issues. So tables are decomposed into multiple smaller tables.

For retrieving information from multiple tables, we need to extract selected data from different records, using operations called join(inner join, outer join and most importantly natural join). Consider 2 table schemas employee(employee_name, street, city)with n rows and works(employee_name, branch_name, salary) with m rows. A cartesian product of these 2 tables creates a table with n*m rows. A natural join selects from this n*m rows all rows with same values for employee_name. To avoid loss of information(some tuples in employee have no corresponding tuples in works) we use left outer join or right outer join.

A join operation or a nested query is better subject to conditions:

Suppose our 2 tables are stored on a local system. Performing a join or a nested query will make little difference. Now let tables be stored across a distributed databases. For a nested query, we only extract the relevant information from each table, located on different computers, then merge the tuples obtained to obtain the result. For a join, we would be required to fetch the whole table from each site and create a large table from which the filtering will occur, hence more time will be required. So for distributed databases, nested queries are better.

RDBMS optimizer is concerned with performance related to the subquery or join written by the programmer. Joins are universally understood hence no optimization issues can arise. If portability across multiple platforms is called for, avoid subqueries as it may run into bugs(SQL server more adept with joins as its usually used with Microsoft's graphical query editors that use joins).

Implementation specific: Suppose we have queries where a few of the nested queries are constant. In MySQL, every constant subquery would be evaluated as many times as encountered, there being no cache facility. This is an obvious problem if the constant subquery involves large tuples.

Subqueries return a set of data. Joins return a dataset which is necessarily indexed. Working on indexed data is faster so if the dataset returned by subqueries is large, joins are a better idea.

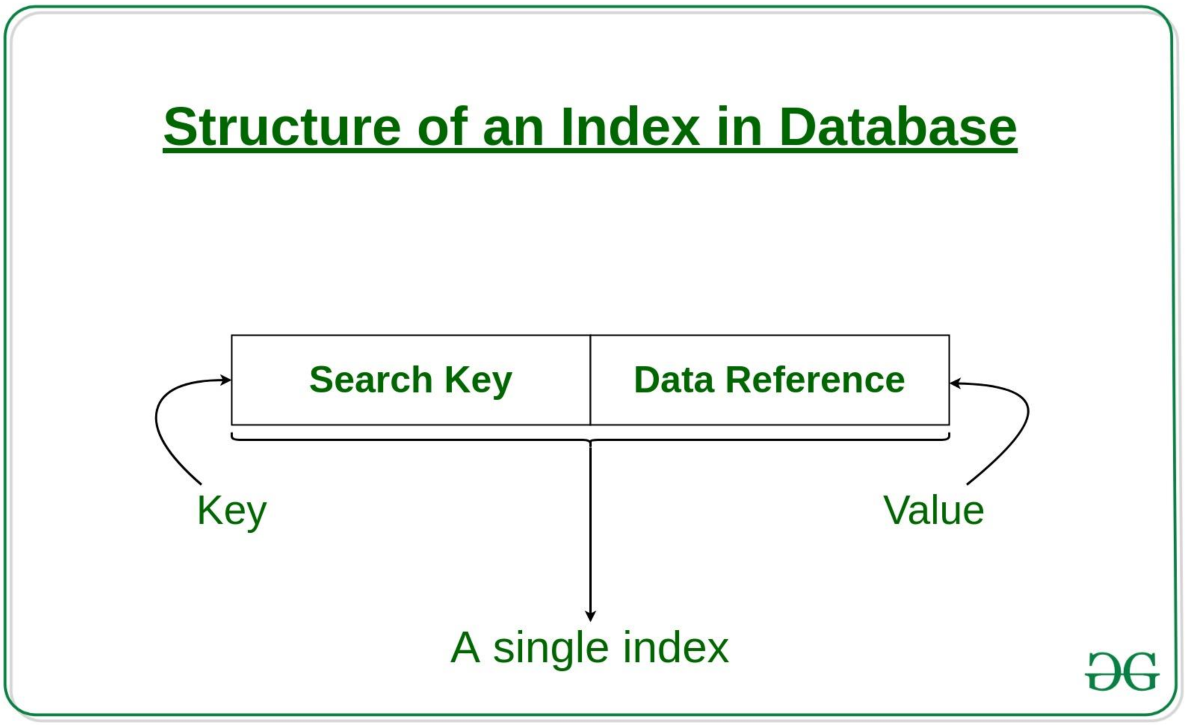
Subqueries may take longer to execute than joins depending on how the database optimizer treats them(may be converted to joins). Subqueries are easier to read, understand and evaluate than cryptic joins. They allow a bottom-up approach, isolating and completing each task sequentially.

Indexing in Databases | Set 1

Indexing is a way to optimize the performance of a database by minimizing the number of disk accesses required when a query is processed. It is a data structure technique which is used to quickly locate and access the data in a database.

Indexes are created using a few database columns.

- The first column is the **Search key** that contains a copy of the primary key or candidate key of the table. These values are stored in sorted order so that the corresponding data can be accessed quickly.
  Note: The data may or may not be stored in sorted order.
- The second column is the **Data Reference** or **Pointer** which contains a set of pointers holding the address of the disk block where that particular key value can be found.



The indexing has various attributes:

- **Access Types**: This refers to the type of access such as value based search, range access, etc.
- **Access Time**: It refers to the time needed to find particular data element or set of elements.
- **Insertion Time**: It refers to the time taken to find the appropriate space and insert a new data.

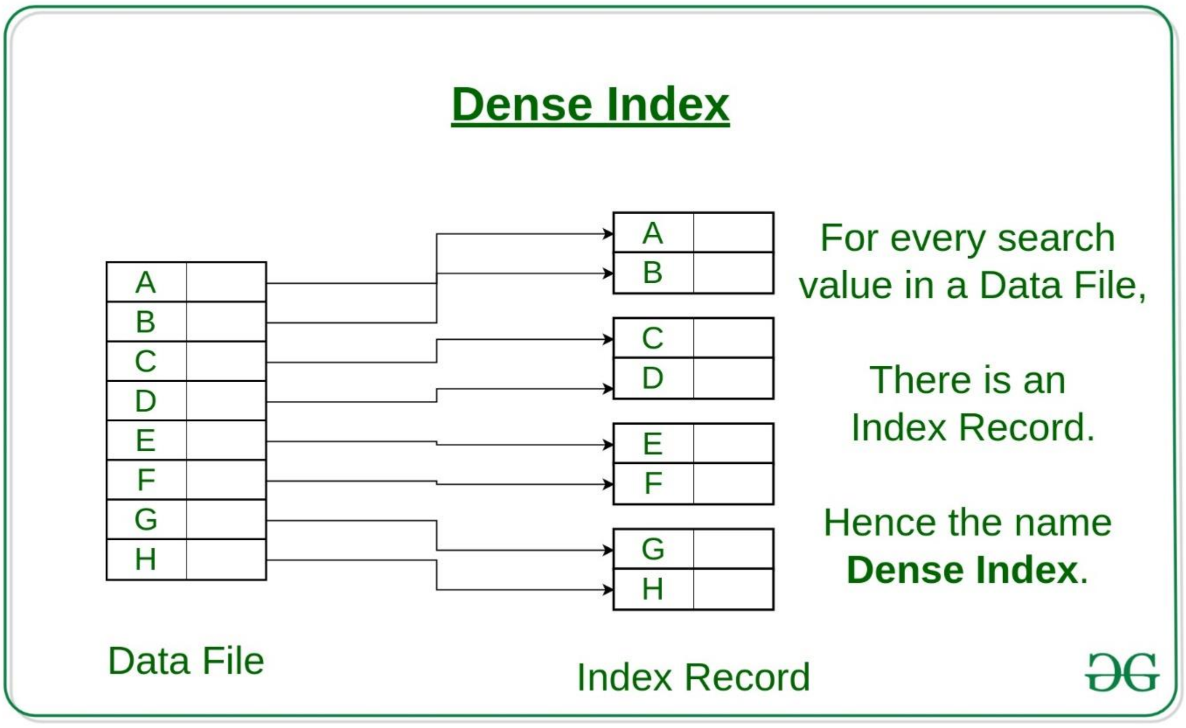
- **Deletion Time**: Time taken to find an item and delete it as well as update the index structure.
- **Space Overhead**: It refers to the additional space required by the index.

In general, there are two types of file organization mechanism which are followed by the indexing methods to store the data:

1. Sequential File Organization or Ordered Index File: In this, the indices are based on a sorted ordering of the values. These are generally fast and a more traditional type of storing mechanism. These Ordered or Sequential file organization might store the data in a dense or sparse format:

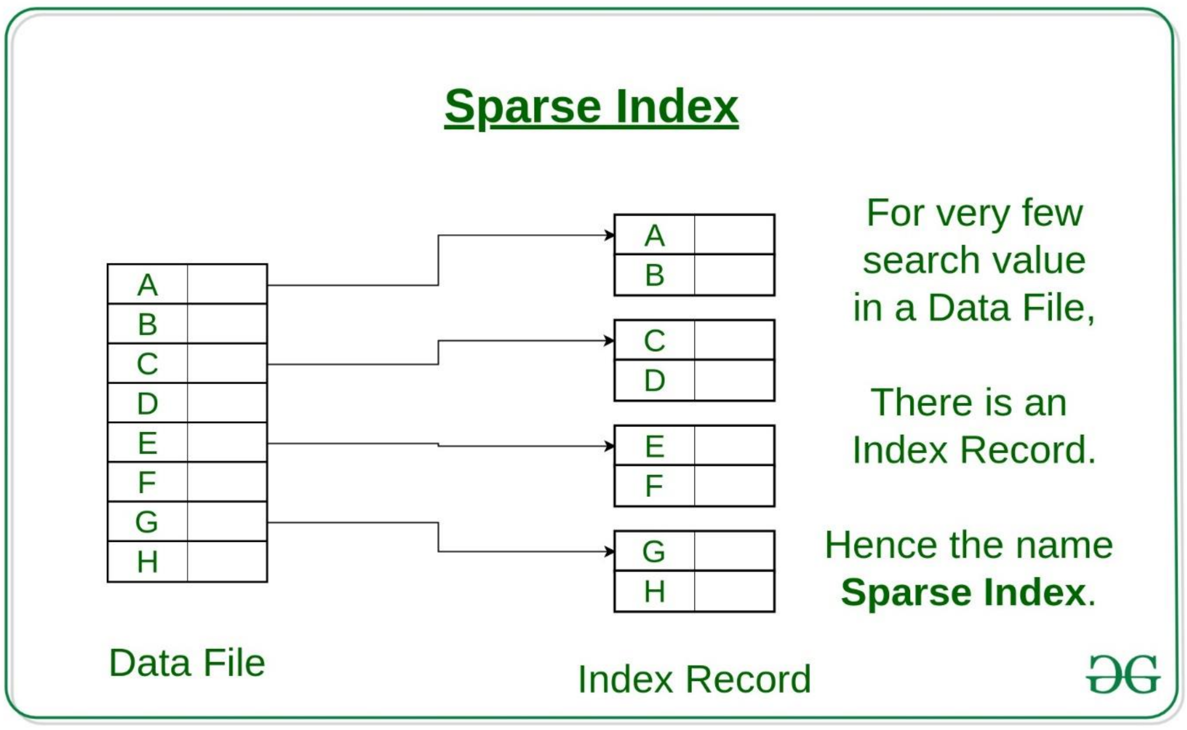
(i) Dense Index:

- 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.


Dense Index
A
B
C
D
E
F
G
H
Data File
Index Record
For every search value in a Data File,
There is an Index Record.
Hence the name Dense Index.

(ii) Sparse Index:

- The index record appears only for a few items in the data file. Each item points to a block as shown.
- To locate a record, we find the index record with the largest search key value less than or equal to the search key value we are looking for.
- We start at that record pointed to by the index record, and proceed along with the pointers in the file (that is, sequentially) until we find the desired record.
- Number of Accesses required=$\log_2(n)+1$, (here n=number of blocks acquired by index file)

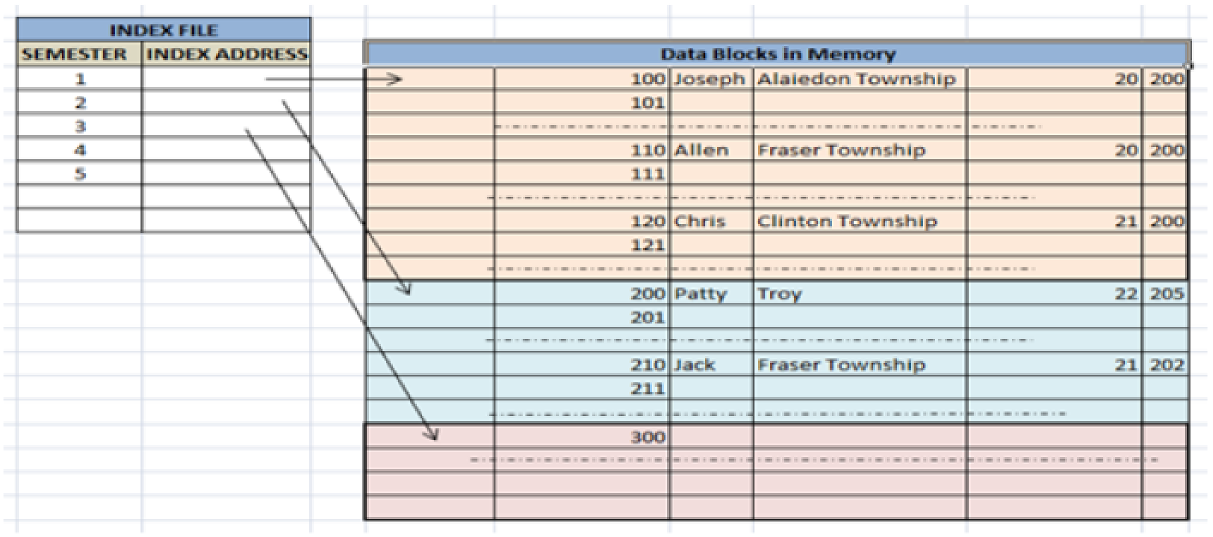

Sparse Index
A
B
C
D
E
F
G
H
For very few search value in a Data File,
There is an Index Record.
Hence the name **Sparse Index**.
Data File
Index Record

2. Hash File organization: Indices are based on the values being distributed uniformly across a range of buckets. The buckets to which a value is assigned is determined by a function called a hash function.

There are primarily three methods of indexing:

- Clustered Indexing
- Non-Clustered or Secondary Indexing
- Multilevel Indexing

1. Clustered Indexing
When more than two records are stored in the same file these types of storing known as cluster indexing. By using the cluster indexing we can reduce the cost of searching reason being multiple records related to the same thing are stored at one place and it also gives the frequent joining of more than two tables (records).
Clustering index is defined on an ordered data file. The data file is ordered on a non-key field. 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 the 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.



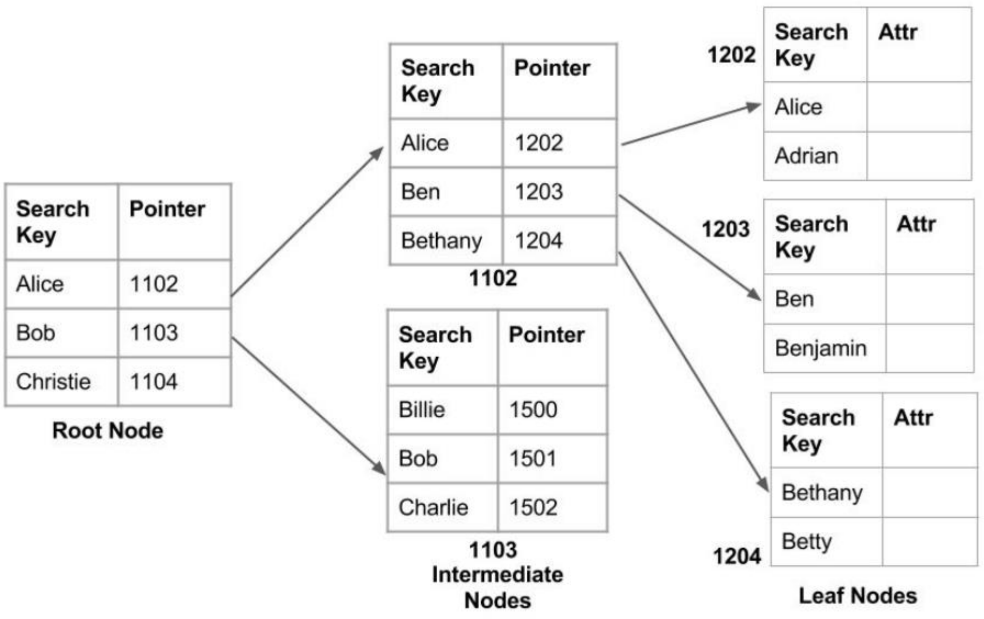
INDEX FILE	
SEMESTER	INDEX ADDRESS
1	
2	
3	
4	
5	

Data Blocks in Memory					
	100	Joseph	Alaiedon Township	20	200
	101				
	110	Allen	Fraser Township	20	200
	111				
	120	Chris	Clinton Township	21	200
	121				
	200	Patty	Troy	22	205
	201				
	210	Jack	Fraser Township	21	202
	211				
	300				

Clustered index sorted according to first name (Search key)

Primary Indexing:
This is a type of Clustered Indexing wherein the data is sorted according to the search key and the primary key of the database table is used to create the index. It is a default format of indexing where it induces sequential file organization. As primary keys are unique and are stored in a sorted manner, the performance of the searching operation is quite efficient.

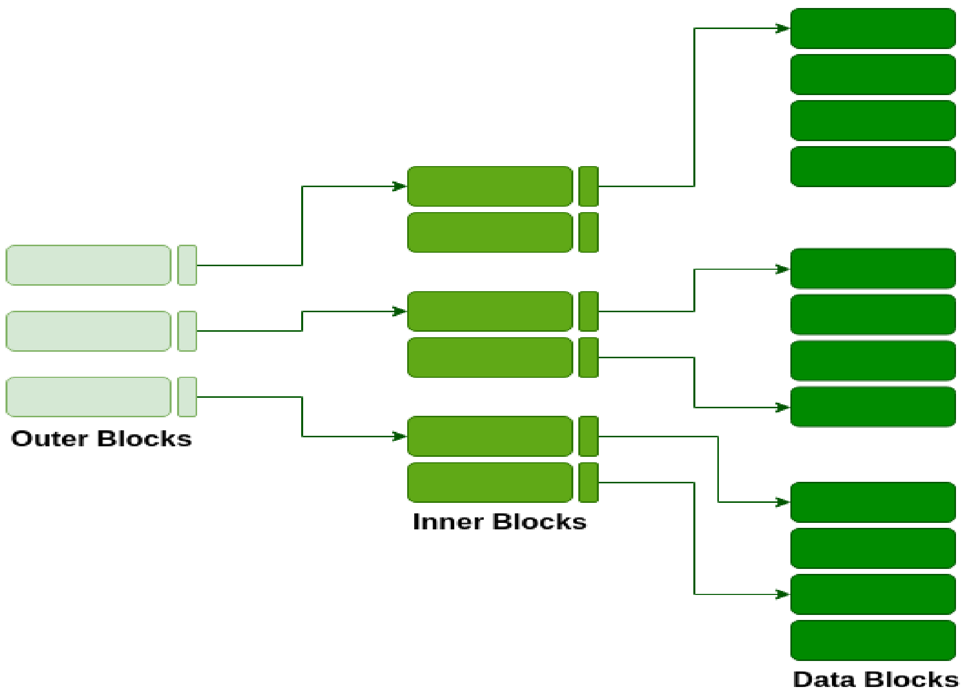
2. Non-clustered or Secondary Indexing
A non clustered index just tells us where the data lies, i.e. it gives us a list of virtual pointers or references to the location where the data is actually stored. Data is not physically stored in the order of the index. Instead, data is present in leaf nodes. For eg. the contents page of a book. Each entry gives us the page number or location of the information stored. The actual data here(information on each page of the book) is not organized but we have an ordered reference(contents page) to where the data points actually lie. We can have only dense ordering in the non-clustered index as sparse ordering is not possible because data is not physically organized accordingly.
It requires more time as compared to the clustered index because some amount of extra work is done in order to extract the data by further following the pointer. In the case of a clustered index, data is directly present in front of the index.



Non clustered index

3. Multilevel Indexing

With the growth of the size of the database, indices also grow. As the index is stored in the main memory, a single-level index might become too large a size to store with multiple disk accesses. The multilevel indexing segregates the main block into various smaller blocks so that the same can stored in a single block. The outer blocks are divided into inner blocks which in turn are pointed to the data blocks. This can be easily stored in the main memory with fewer overheads.



SQL queries on clustered and non-clustered Indexes

Prerequisite – Indexing in Databases Indexing is a procedure that returns your requested data faster from the defined table. Without indexing, the SQL server has to scan the whole table for your data. By indexing, SQL server will do the exact same thing you do when searching for content in a book by checking the index page. In the same way, a table's index allows us to locate the exact data without scanning the whole table. There are two types of indexing in SQL.

1. Clustered index
2. Non-clustered index

1. **Clustered** – Clustered index is the type of indexing that establishes a physical sorting order of rows. Suppose you have a table *Student_info* which contains *ROLL_NO* as a primary key, then Clustered index which is self-created on that primary key will sort the *Student_info* table as per *ROLL_NO*. Clustered index is like Dictionary; in the dictionary, sorting order is alphabetical and there is no separate index page.

Examples:

Input:

CREATE TABLE Student_info(ROLL_NO int(10) primary key,NAME varchar(20),DEPARTMENT varchar(20),);

insert into Student_info values(1410110405, 'H Agarwal', 'CSE')

insert into Student_info values(1410110404, 'S Samadder', 'CSE')

insert into Student_info values(1410110403, 'MD Irfan', 'CSE')

SELECT * FROM Student_info

Output:

ROLL_NO	NAME	DEPARTMENT
1410110403	MD Irfan	CSE
1410110404	S Samadder	CSE
1410110405	H Agarwal	CSE

If we want to create a Clustered index on another column, first we have to remove the primary key, and then we can remove the previous index. Note that defining a column as a primary key makes that column the Clustered Index of that table. To make any other column, the clustered index, first we have to remove the previous one as follows below.

Syntax:

//Drop index

drop index table_name.index_name

//Create Clustered index index

create Clustered index IX_table_name_column_name

on table_name (column_name ASC)

Note: We can create only one clustered index in a table.

2. Non-clustered: Non-Clustered index is an index structure separate from the data stored in a table that reorders one or more selected columns. The non-clustered index is created to improve the performance of frequently used queries not covered by a clustered index. It's like a textbook; the index page is created separately at the beginning of that book. **Examples:**

Input:

```
CREATE TABLE Student_info

(

ROLL_NO int(10),

NAME varchar(20),

DEPARTMENT varchar(20),

);

insert into Student_info values(1410110405, 'H Agarwal', 'CSE')

insert into Student_info values(1410110404, 'S Samadder', 'CSE')

insert into Student_info values(1410110403, 'MD Irfan', 'CSE')


SELECT * FROM Student_info
```

Output:

ROLL_NO	NAME	DEPARTMENT
1410110405	H Agarwal	CSE
1410110404	S Samadder	CSE
1410110403	MD Irfan	CSE

Note: We can create one or more Non_Clustered index in a table.

Syntax:

```
//Create Non-Clustered index

create NonClustered index IX_table_name_column_name  on table_name (column_name ASC)
```

Table: Student_info

ROLL_NO	NAME	DEPARTMENT
1410110405	H Agarwal	CSE

1410110404	S Samadder	CSE
1410110403	MD Irfan	CSE

Input: create NonClustered index IX_Student_info_NAME on Student_info (NAME ASC)
Output: Index

NAME	ROW_ADDRESS
H Agarwal	1
MD Irfan	3
S Samadder	2

Clustered vs Non-Clustered index:

- In a table, there can be only one clustered index or one or more than one non_clustered index.
- In Clustered index, there is no separate index storage but in Non-Clustered index, there is separate index storage for the index.
- Clustered index offers faster data access, on the other hand, the Non-clustered index is slower.

Reference: https://www.geeksforgeeks.org