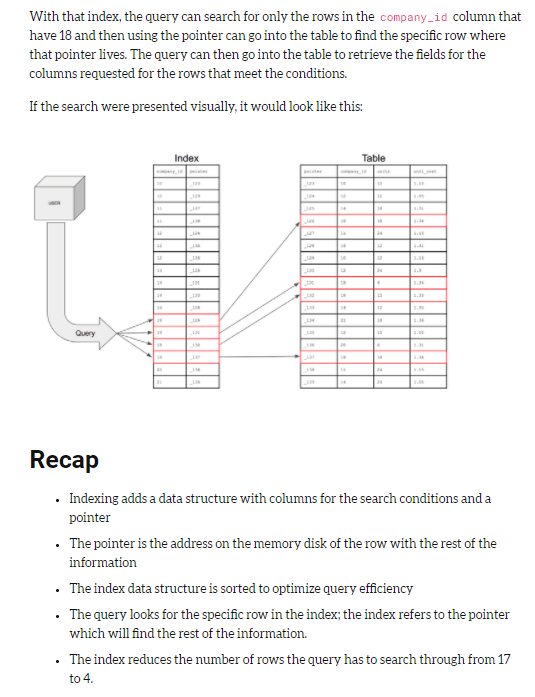
## Index:

Indexes are used by queries to find data from tables quickly. Indexes are created on tables and views. Index on a table or a view, is very similar to an index that we find in a book. If there is no index to help the query, then the query engine, checks every row in the table from the beginning to the end. This is called as Table Scan. Table scan is bad for performance.

**Clustered Index:**  
A clustered index determines the physical order of data in a table. For this reason, a table can have **only one** clustered index.

 Primary key constraint create **clustered indexes automatically** if no clustered index already exists on the table and a nonclustered index is not specified when you create the PRIMARY KEY constraint.

* InnoDB is the default storage engine for MySQL, and it does not support true clustered indexes where the data is physically stored in the order of the index key.
* Instead, InnoDB uses the **PRIMARY KEY** as a clustered index. In summary, while MySQL's InnoDB does not support explicit clustered indexes, the **PRIMARY KEY** serves as a clustered index by default. The **CREATE INDEX** statement is used for creating non-clustered indexes on specific columns to optimize query performance.

MySQL, the terms "CREATE INDEX" and "CREATE NONCLUSTERED INDEX" are often used interchangeably. When you issue a **CREATE INDEX** statement in MySQL, you are creating a non-clustered index by default. In MySQL's default storage engine, InnoDB, true clustered indexes are not supported. Therefore, all indexes created using the **CREATE INDEX** statement are effectively non-clustered.  
*Create Clustered Index IX\_tblEmployee\_Name ON tblEmployee\_Table(Column.Name)*

**Non-Clustered Index:**  
A nonclustered index is analogous to an index in a textbook. The data is stored in one place, the index in another place. The index will have pointers to the storage location of the data. Since, the nonclustered index is stored separately from the actual data, a table can have more than one non clustered index, just like how a book can have an index by Chapters at the beginning and another index by common terms at the end.

1. **Only one clustered index per table**, where as you can have more than one non clustered index  
2. **Clustered index is faster than a non-clustered index**, because, the non-clustered index has to refer back to the table, if the selected column is not present in the index.  
3. **Clustered index determines the storage order of rows in the table**, and hence doesn't require additional disk space, but whereas a non-Clustered index is stored separately from the table, additional storage space is required.

1. **Atomicity:**
   * **Easy Understanding:** Think of atomicity like a single, indivisible action. In a transaction, either all the changes happen, or none happen at all.
   * **Example:** Imagine transferring money between bank accounts. Atomicity ensures that if you withdraw money from one account, the corresponding deposit into the other account also happens. If any part fails, the entire transaction is rolled back.
2. **Consistency:**
   * **Easy Understanding:** After a transaction, the database should be in a consistent state. This means that the data follows predefined rules and constraints, maintaining the integrity of the database.
   * **Example:** If you have a rule that every person's age must be 18 or older, a consistency violation would occur if a transaction tried to set someone's age to 16.

CREATE TABLE Person (

PersonID INT PRIMARY KEY,

FirstName VARCHAR(50),

LastName VARCHAR(50),

Age INT CHECK (Age >= 18)

);

1. **Isolation:**
   * **Easy Understanding:** Transactions should operate independently of each other. The outcome of one transaction should not affect the outcome of another.
   * **Example:** If two people are booking flights simultaneously, the isolation principle ensures that the bookings are processed independently. One person's booking doesn't interfere with the other person's.
2. **Durability:**
   * **Easy Understanding:** Once a transaction is committed, its changes are permanent, surviving any subsequent failures or system crashes.
   * **Example:** If you update your profile information on a website and receive a confirmation message, durability ensures that your updated profile information is still there even if the server crashes immediately after.

Explain the difference between eventual consistency & strong consistency

**Find the duplicate name with count from table**

SELECT NAME, COUNT(NAME) FROM USER GROUP BY NAME HAVING COUNT(NAME)>1

SELECT max(salary)

FROM emptable WHERE salary < (SELECT max(salary) FROM emptable);

**select** **min**(salary) **from** (**select** **distinct** salary **from** emp **order** **by** salary **desc**)

**where** rownum < 3;

select \* from Employee ORDER BY `sal` ASC limit 5,1; // will return 6th highest

Aggregate Function: Used to group multiple rows together as input to form a single value output.

Eg: Min, Max, Avg, Count, Sum, count(\*)

**Min, Max, Avg, Count, Sum** -> These are works on **column**

Eg1) Select Avg(Salary) from Employee;

Eg2) Select Sum(Salary) from Employee;

**count(\*)** -> This will works on **rows**

Eg1) Select count(\*) from Employee; -Return no of rows in Employee table

## Group by:

i) Aggregate functions are calculated for each group.

ii) Groups are formed on the basis of certain attributes.

iii) It’s used group the data.

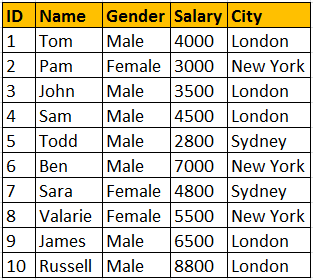
iv) It’s **always** used in conjunction with one or more **Aggregate functions**.

**Eg) Aggregate functions are calculated for each group:**

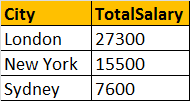
Determine the maximum salary of employee in each department.

SELECT Department, Max(Salary)as Max\_Salary from Employee Group by Department.

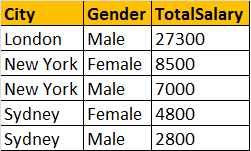
**Examples**: https://csharp-video-tutorials.blogspot.com/2012/08/group-by-part-11.html



**Eg1) Query for retrieving total salaries by city**:  
We are applying SUM() aggregate function on Salary column, and grouping by city column. This effectively adds, all salaries of employees with in the same city.  
**Select City, SUM(Salary) as TotalSalary**  
**from tblEmployee**  
**Group by City**



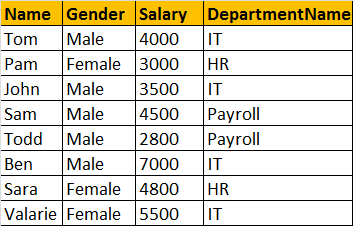
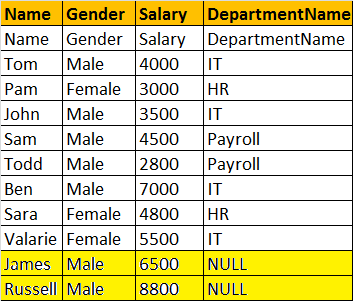
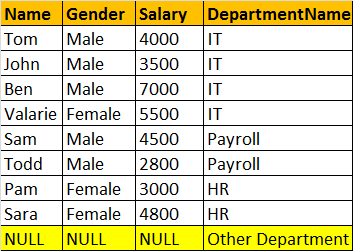
**Eg2) Query for retrieving total salaries by city and by gender**: It's possible to group by multiple columns. In this query, we are grouping first by city and then by gender.   
**Select City, Gender, SUM(Salary) as TotalSalary**  
**from tblEmployee**  
**group by City, Gender**

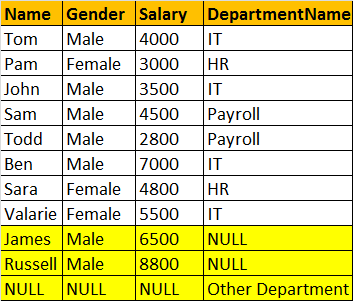
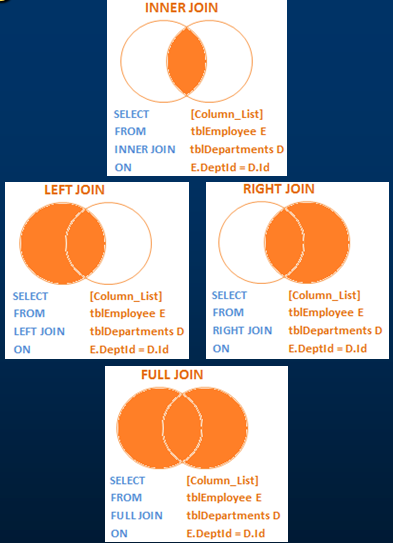
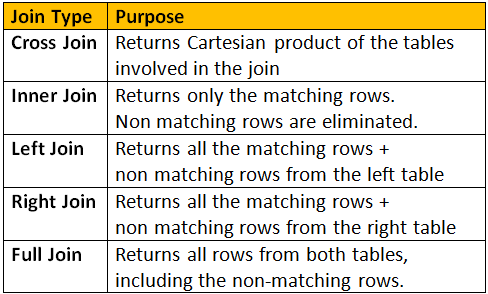


**Filtering** rows using WHERE clause, before aggregations take place:  
**Select City, SUM(Salary) as TotalSalary**  
**from tblEmployee**  
**Where City = 'London'**  
**group by City**  
  
Filtering groups using HAVING clause, after all aggrgations take place:  
**Select City, SUM(Salary) as TotalSalary**  
**from tblEmployee**  
**group by City**  
**Having City = 'London'**

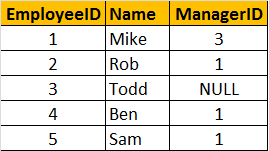
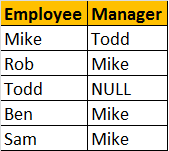
**Difference between WHERE and HAVING clause:**  
1. WHERE clause can be used with - Select, Insert, and Update statements, where as HAVING clause can only be used with the Select statement.  
2. WHERE filters rows before aggregation (GROUPING), whereas, HAVING filters groups, after the aggregations are performed.  
3. Aggregate functions cannot be used in the WHERE clause, unless it is in a sub query contained in a HAVING clause, whereas, aggregate functions can be used in Having clause.

Join:

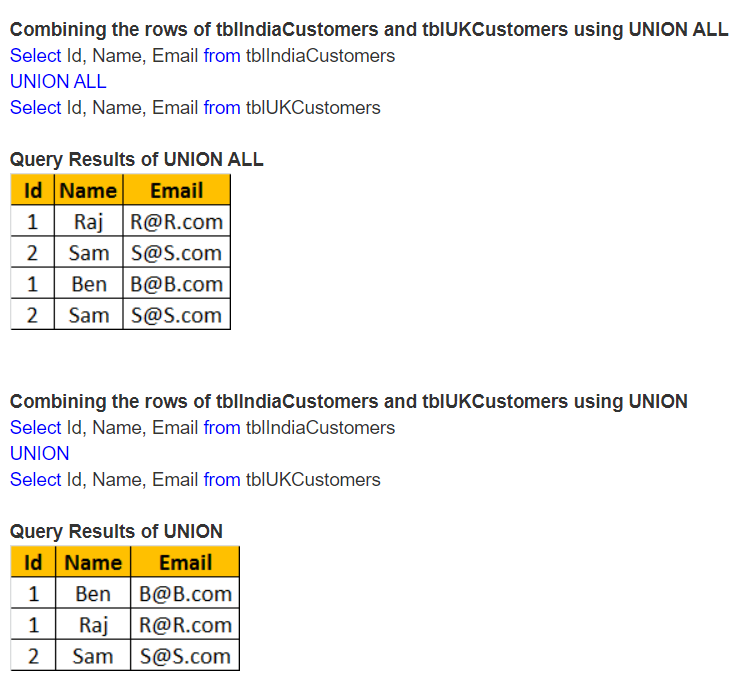
**General Formula for Joins**  
SELECT      ColumnList  
FROM           LeftTableName  
JOIN\_TYPE  RightTableName  
ON                 JoinCondition  
  
**CROSS JOIN**  
CROSS JOIN, produces the **cartesian product of the 2 tables** involved in the join. For example, in the Employees table we have 10 rows and in the Departments table we have 4 rows. So, a cross join between these 2 tables produces 40 rows. Cross Join shouldn't have ON clause.   
  
**CROSS JOIN Query:**  
SELECT Name, Gender, Salary, DepartmentName  
FROM tblEmployee  
CROSS JOIN tblDepartment  
  
**JOIN or INNER JOIN**  
Write a query, to retrieve Name, Gender, Salary and DepartmentName from Employees and Departments table. The output of the query should be as shown below.  
  
  
SELECT Name, Gender, Salary, DepartmentName  
FROM tblEmployee  
INNER JOIN tblDepartment  
ON tblEmployee.DepartmentId = tblDepartment.Id  
  
**OR**  
  
SELECT Name, Gender, Salary, DepartmentName  
FROM tblEmployee  
JOIN tblDepartment  
ON tblEmployee.DepartmentId = tblDepartment.Id  
  
**Note:** JOIN or INNER JOIN means the same. It's always better to use INNER JOIN, as this explicitly specifies your intention.  
  
If you look at the output, we got only 8 rows, but in the Employees table, we have 10 rows. We didn't get JAMES and RUSSELL records. This is because the DEPARTMENTID, in Employees table is NULL for these two employees and doesn't match with ID column in Departments table.  
  
So, in summary, INNER JOIN, returns only the matching rows between both the tables. Non matching rows are eliminated.  
  
**LEFT JOIN or LEFT OUTER JOIN**  
Now, let's say, I want all the rows from the Employees table, including JAMES and RUSSELL records. I want the output, as shown below.  
  
  
SELECT Name, Gender, Salary, DepartmentName  
FROM tblEmployee  
LEFT OUTER JOIN tblDepartment  
ON tblEmployee.DepartmentId = tblDepartment.Id  
  
**OR**  
  
SELECT Name, Gender, Salary, DepartmentName  
FROM tblEmployee  
LEFT JOIN tblDepartment  
ON tblEmployee.DepartmentId = tblDepartment.Id  
  
**Note:** You can use, LEFT JOIN or LEFT OUTER JOIN. OUTER keyowrd is optional  
  
**LEFT JOIN**, returns all the matching rows + non matching rows from the left table. In reality, INNER JOIN and LEFT JOIN are extensively used.  
  
**RIGHT JOIN or RIGHT OUTER JOIN**  
I want, all the rows from the right table. The query output should be, as shown below.  


SELECT Name, Gender, Salary, DepartmentName  
FROM tblEmployee  
RIGHT OUTER JOIN tblDepartment  
ON tblEmployee.DepartmentId = tblDepartment.Id  
  
**OR**  
  
SELECT Name, Gender, Salary, DepartmentName  
FROM tblEmployee  
RIGHT JOIN tblDepartment  
ON tblEmployee.DepartmentId = tblDepartment.Id  
  
**Note:** You can use, RIGHT JOIN or RIGHT OUTER JOIN. OUTER keyowrd is optional  
  
**RIGHT JOIN**, returns all the matching rows + non matching rows from the right table.  
  
**FULL JOIN or FULL OUTER JOIN**  
I want all the rows from both the tables involved in the join. The query output should be, as shown below.  
  
  
  
SELECT Name, Gender, Salary, DepartmentName  
FROM tblEmployee  
FULL OUTER JOIN tblDepartment  
ON tblEmployee.DepartmentId = tblDepartment.Id  
  
OR  
  
SELECT Name, Gender, Salary, DepartmentName  
FROM tblEmployee  
FULL JOIN tblDepartment  
ON tblEmployee.DepartmentId = tblDepartment.Id  
  
**Note:** You can use, FULLJOIN or FULL OUTER JOIN. OUTER keyowrd is optional  
  
**FULL JOIN**, returns all rows from both the left and right tables, including the non-matching rows.  
  
**Joins Summary**  
  
  


## **Self-Join**

Consider tblEmployees table shown below.  
  
  
Write a query which gives the following result.  
  
  
  
**Self Join Query:**  
A MANAGER is also an EMPLOYEE. Both the, EMPLOYEE and MANAGER rows, are present in the same table. Here we are joining tblEmployee with itself using different alias names, E for Employee and M for Manager. We are using LEFT JOIN, to get the rows with ManagerId NULL. You can see in the output TODD's record is also retrieved, but the MANAGER is NULL. If you replace LEFT JOIN with INNER JOIN, you will not get TODD's record.  
Select E.Name as Employee, M.Name as Manager  
from tblEmployee E  
Left Join tblEmployee M  
On E.ManagerId = M.EmployeeId  
  
  
In short, joining a table with itself is called as **SELF JOIN**. SELF JOIN is not a different type of JOIN. It can be classified under any type of JOIN - INNER, OUTER or CROSS Joins. The above query is, LEFT OUTER SELF Join.  
  
**Inner Self Join tblEmployee table:**  
Select E.Name as Employee, M.Name as Manager  
from tblEmployee E  
Inner Join tblEmployee M  
On E.ManagerId = M.EmployeeId  
  
**Cross Self Join tblEmployee table:**  
Select E.Name as Employee, M.Name as Manager  
from tblEmployee  
Cross Join tblEmployee

**Union and UnionAll**



**Query to get 3rd Highest salary from Employee table**

SELECT DISTINCT salary

FROM Employee

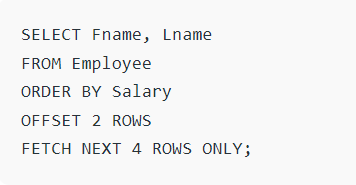
ORDER BY salary DESC

LIMIT 1 OFFSET 2;

**LIMIT 1 OFFSET 2** skips the first two rows (the highest and second-highest salaries) and retrieves the third row.

**Fetch 1Lack record from table effectively in sql**-> Offset ->https://www.youtube.com/watch?v=7JINCqgT\_cQ

<https://www.geeksforgeeks.org/sql-offset-fetch-clause/>



## **Add Foreign Key:**

Can a foreign key ever be null?

Yes, it can be NULL. If we don’t pass any value to the foreign key column, it will take NULL value. But we can add **default constraint** also, will see in the next part,

Graphical user interface, text, application, email

Description automatically generated

## **Default constraint**

##### Graphical user interface, text, application, email Description automatically generated

# **Cascading referential integrity constraint**

If we delete parent table record, we will get exception as its used in child table as an FK. In SQL, we can set delete rule and update rule for **foreign key (Child table)**. We can set as **null/default value** in the delete rule of foreign key (Child table), the child record got deleted and parent record key will be updated to null/default value

There is another option called **cascade**, if this rule was added to FK, when we delete the parent table record, all the child table who are using that particular row will be deleted.

Graphical user interface, text, application, email

Description automatically generated

**Check Constraints:**

A screenshot of a computer

Description automatically generated with medium confidence

**A screenshot of a computer

Description automatically generated with medium confidence**

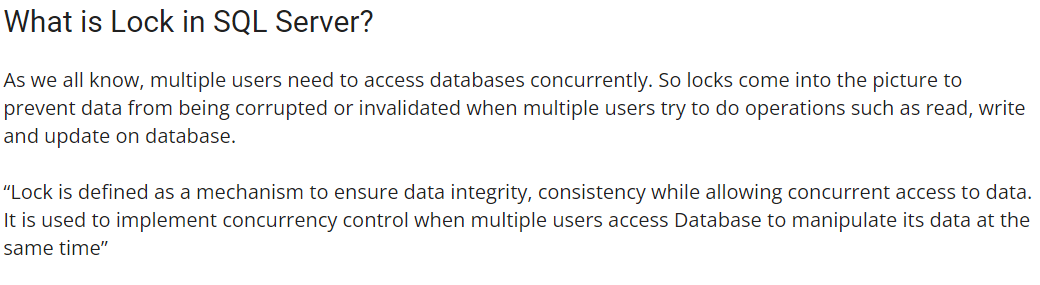
**Unique Key:**

**Text

Description automatically generated**

**Normalization in sql:**

**Offset query**

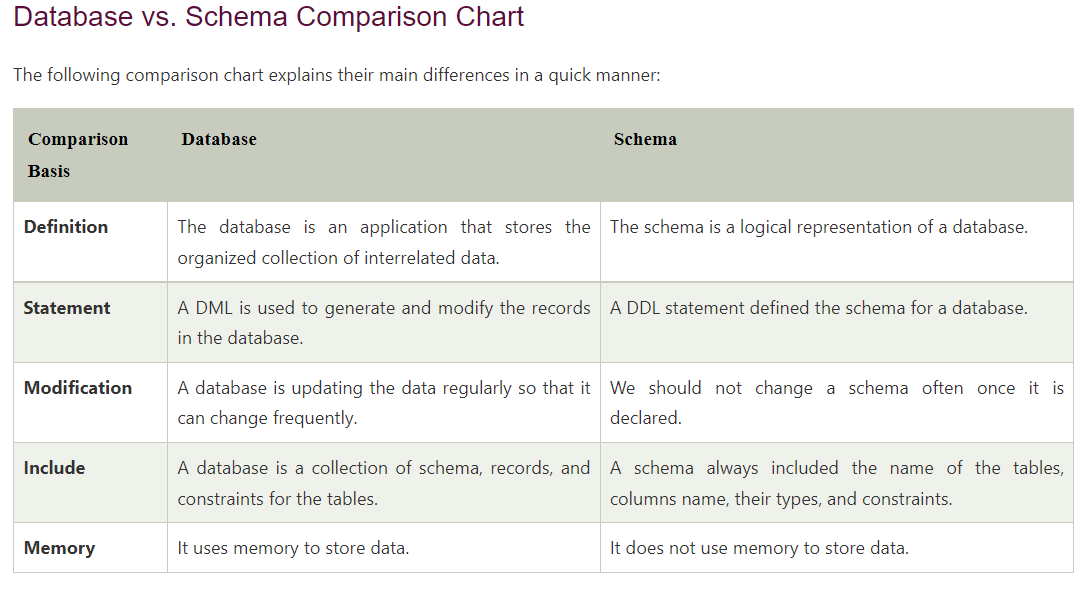
****

**Difference between shared and exclusive locks in sql:**

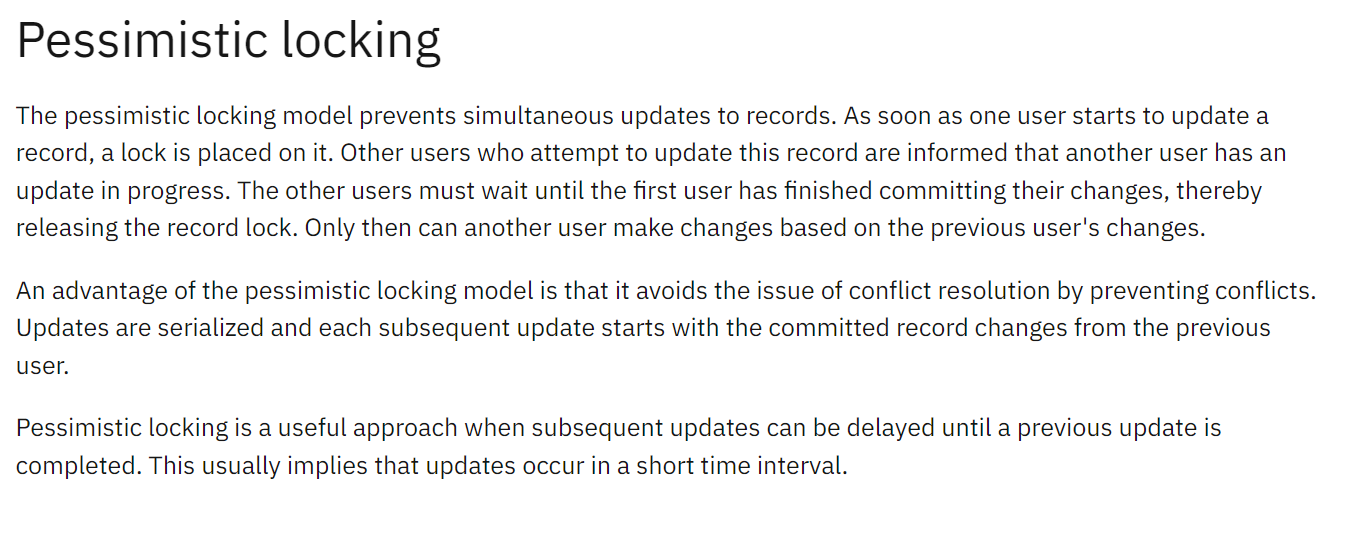
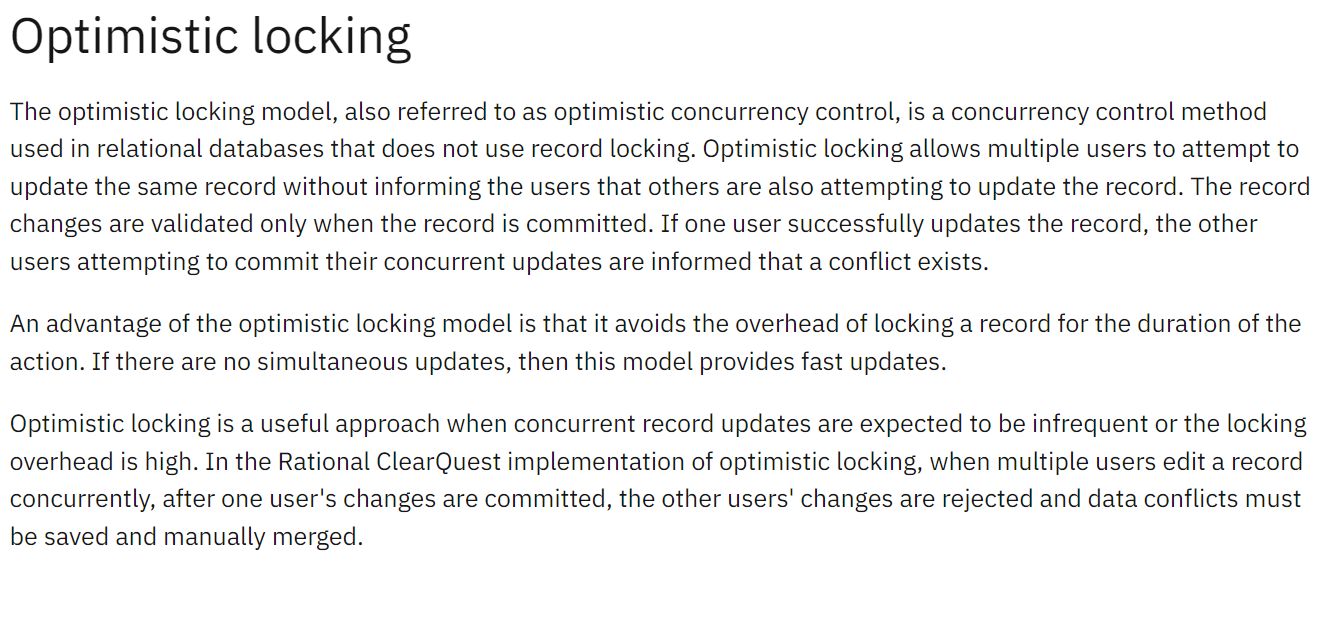
Shared and exclusive lock, both are going to allow only read not write.

But the difference is that, one table got exclusively locked already in one session, we can’t able to lock again in another session.

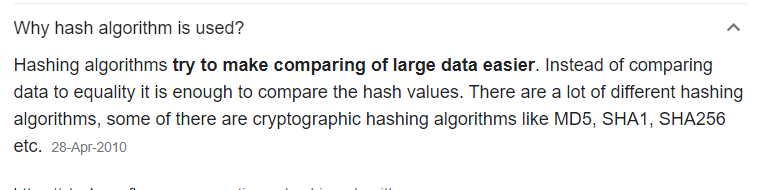
whereas, if one table got shared locked already in one session, we can do share lock again in another session also.



##### Pessimistic and Optimistic locks

****

**Hashing algorithm**

****

**CAP,**

**Schema in sql**

**Isolation - Dirtyread, RepeatRead**