

# FullStack.Cafe - Kill Your Tech Interview

---

## Q1: Define a *Temp Table* ☆

---

Topics: SQL

### Answer:

In a nutshell, a **temp table** is a *temporary storage structure*. Basically, you can use a temp table to store data *temporarily* so you can manipulate and change it before it reaches its destination format.

## Q2: What is a **VIEW** ? ☆

---

Topics: SQL

### Answer:

A **view** is simply a virtual table that is made up of elements of multiple physical or “real” tables. Views are most commonly used to join multiple tables together, or control access to any tables existing in background server processes.

## Q3: What is **PRIMARY KEY** ? ☆

---

Topics: SQL

### Answer:

- A **PRIMARY KEY** constraint is a *unique identifier* for a row within a database table.
- Every table *should have* a primary key constraint to uniquely identify each row and only one primary key constraint can be created for each table.
- The primary key constraints are used to enforce *entity integrity*.

## Q4: What is **FOREIGN KEY** ? ☆☆

---

Topics: SQL

### Answer:

- **FOREIGN KEY** constraint *prevents* any actions that would destroy links between tables with the corresponding data values.
- A foreign key in one table *points* to a primary key in another table.
- Foreign keys prevent actions that would leave rows with foreign key values when there are no primary keys with that value.
- The foreign key constraints are used to **enforce referential integrity**.

## Q5: What is *Normalisation*? ☆☆

---

Topics: SQL Databases

### Answer:

**Normalization** is basically to design a database schema such that **duplicate and redundant data is avoided**. If the same information is repeated in multiple places in the database, there is the risk that it is updated in one place but not the other, leading to data corruption.

There is a number of normalization levels from 1. normal form through 5. normal form. Each normal form describes how to get rid of some specific problem.

By having a database with normalization errors, you open the risk of getting invalid or corrupt data into the database. Since data "lives forever" it is very hard to get rid of corrupt data when first it has entered the database.

### Q6: What is **DEFAULT** ? ☆☆☆

---

Topics: SQL

### Answer:

- **Default** allows to add values to the column *if the value of that column is not set*.
- Default can be defined on number and datetime fields.
- They cannot be defined on timestamp and **IDENTITY** columns.

### Q7: What is the difference between **TRUNCATE** and **DELETE** ? ☆☆☆

---

Topics: MySQL SQL

### Answer:

- **DELETE** is a Data Manipulation Language(DML) command. It can be used for deleting some specified rows from a table. **DELETE** command can be used with **WHERE** clause.
- **TRUNCATE** is a Data Definition Language(DDL) command. It deletes all the records of a particular table. **TRUNCATE** command is faster in comparison to **DELETE**. While **DELETE** command can be rolled back, **TRUNCATE** can not be rolled back in MySQL.

### Q8: What is the difference between *Data Definition Language (DDL)* and *Data Manipulation Language (DML)*? ☆☆☆

---

Topics: MySQL SQL T-SQL Databases

### Answer:

- **Data definition language (DDL)** commands are the commands which are used to define the database. **CREATE**, **ALTER**, **DROP** and **TRUNCATE** are some common DDL commands.
- **Data manipulation language (DML)** commands are commands which are used for manipulation or modification of data. **INSERT**, **UPDATE** and **DELETE** are some common DML commands.

### Q9: What is the difference between **INNER JOIN** , **OUTER JOIN** , **FULL OUTER JOIN** ? ☆☆☆

---

Topics: SQL

**Answer:**

An **inner join** retrieve the matched rows only.

Whereas an **outer join** retrieve the matched rows from one table and all rows in other table ....the result depends on which one you are using:

- **Left:** Matched rows in the right table and all rows in the left table
- **Right:** Matched rows in the left table and all rows in the right table or
- **Full:** All rows in all tables. It doesn't matter if there is a match or not

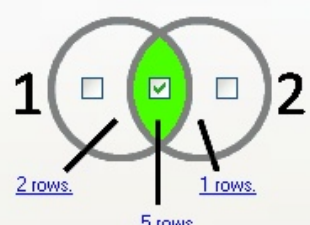
**Inner Join**

Retrieve the matched rows only, that is,  $A \cap B$ .

Block Name: Join Block 2 Join Block

Join Type: Inner Join

Show Output Block output (Inner Join) 5 rows.



Student_Name	Advisor_Name
Student_1	Advisor 1
Student_5	Advisor 3
Student_7	Advisor 3
Student_9	Advisor 1
Student_10	Advisor 3

```
SELECT *
FROM dbo.Students S
INNER JOIN dbo.Advisors A
ON S.Advisor_ID = A.Advisor_ID
```

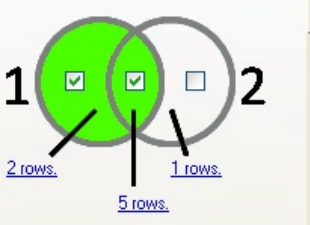
**Left Outer Join**

Select all records from the first table, and any records in the second table that match the joined keys.

Block Name: Join Block 2 Join Block

Join Type: Outer Join on Input 1

Show Output Block output (Outer Join on Input 1) 7 rows.



Student_Name	Advisor_Name
Student_2	_null_
Student_4	_null_
Student_1	Advisor 1
Student_5	Advisor 3
Student_7	Advisor 3
Student_9	Advisor 1
Student_10	Advisor 3

```
SELECT *
FROM dbo.Students S
LEFT JOIN dbo.Advisors A
ON S.Advisor_ID = A.Advisor_ID
```


## Full Outer Join

Select all records from the second table, and any records in the first table that match the joined keys.

Block Name:

Join Type:

Block output (Full Outer Join) 8 rows.



Student_Name	Advisor_Name
Student_2	_null_
Student_4	_null_
Student_1	Advisor 1
Student_5	Advisor 3
Student_7	Advisor 3
Student_9	Advisor 1
Student_10	Advisor 3
_null_	Advisor 5

```
SELECT *
FROM dbo.Students S
FULL JOIN dbo.Advisors A
ON S.Advisor_ID = A.Advisor_ID
```

## Q10: What is the difference between JOIN and UNION ? ☆☆☆

Topics: SQL

### Answer:

UNION puts lines from queries after each other, while JOIN makes a *cartesian* product and subsets it -- completely different operations. Trivial example of UNION :

```
mysql> SELECT 23 AS bah
-> UNION
-> SELECT 45 AS bah;
+-----+
| bah |
+-----+
| 23 |
| 45 |
+-----+
2 rows in set (0.00 sec)
```

similary trivial example of JOIN :

```
mysql> SELECT * FROM
-> (SELECT 23 AS bah) AS foo
-> JOIN
-> (SELECT 45 AS bah) AS bar
-> ON (33=33);
+-----+-----+
| foo | bar |
+-----+-----+
| 23 | 45 |
```

```
+-----+
1 row in set (0.01 sec)
```

## Q11: What is the difference between UNION and UNION ALL ? ☆☆☆

Topics: SQL

### Answer:

UNION removes duplicate records (where all columns in the results are the same), UNION ALL does not.

There is a performance hit when using UNION instead of UNION ALL, since the database server must do additional work to remove the duplicate rows, but usually you do not want the duplicates (especially when developing reports).

### UNION Example:

```
SELECT 'foo' AS bar UNION SELECT 'foo' AS bar
```

### Result:

```
+-----+
| bar |
+-----+
| foo |
+-----+
1 row in set (0.00 sec)
```

### UNION ALL example:

```
SELECT 'foo' AS bar UNION ALL SELECT 'foo' AS bar
```

### Result:

```
+-----+
| bar |
+-----+
| foo |
| foo |
+-----+
2 rows in set (0.00 sec)
```

## Q12: What is the difference between WHERE clause and HAVING clause? ☆☆☆

Topics: SQL

### Answer:

`WHERE` clause introduces a condition on *individual rows*; `HAVING` clause introduces a condition on *aggregations*, i.e. results of selection where a single result, such as count, average, min, max, or sum, has been produced from *multiple* rows. Your query calls for a second kind of condition (i.e. a condition on an aggregation) hence `HAVING` works correctly.

As a rule of thumb, use `WHERE` before `GROUP BY` and `HAVING` after `GROUP BY`. It is a rather primitive rule, but it is useful in more than 90% of the cases.

While you're at it, you may want to re-write your query using ANSI version of the join:

```
SELECT L.LectID, Fname, Lname
FROM Lecturers L
JOIN Lecturers_Specialization S ON L.LectID=S.LectID
GROUP BY L.LectID, Fname, Lname
HAVING COUNT(S.Expertise)>=ALL
(SELECT COUNT(Expertise) FROM Lecturers_Specialization GROUP BY LectID)
```

This would eliminate `WHERE` that was used as a *theta join condition*.

### Q13: Describe the difference between truncate and delete ☆☆☆

Topics: SQL

#### Answer:

- **Delete** command removes the rows from a table based on the condition that we provide with a `WHERE` clause.
- **Truncate** will actually remove all the rows from a table and there will be no data in the table after we run the truncate command.

### Q14: What are the difference between *Clustered* and a *Non-clustered* index? ☆☆☆

Topics: SQL Databases

#### Answer:

- With a **Clustered** index the rows are stored **physically** on the disk in the same order as the index. Therefore, there can be only one clustered index. A clustered index means you are telling the database to store close values actually close to one another on the disk.
- With a **Non Clustered** index there is a second list that has **pointers** to the physical rows. You can have many non clustered indices, although each new index will increase the time it takes to write new records.
- It is generally *faster to read* from a **clustered** index if you want to get back all the columns. You do not have to go first to the index and then to the table.
- *Writing* to a table with a **clustered** index can be *slower*, if there is a need to rearrange the data.

### Q15: What is Denormalization? ☆☆☆

Topics: SQL Databases

#### Answer:

It is the process of improving the performance of the database by *adding* redundant data.

## Q16: Define ACID Properties ☆☆☆

Topics: SQL Databases

### Answer:

- **Atomicity:** It ensures all-or-none rule for database modifications.
- **Consistency:** Data values are consistent across the database.
- **Isolation:** Two transactions are said to be independent of one another.
- **Durability:** Data is not lost even at the time of server failure.

## Q17: What is the difference between INNER JOIN and OUTER JOIN ?

☆☆☆

Topics: SQL

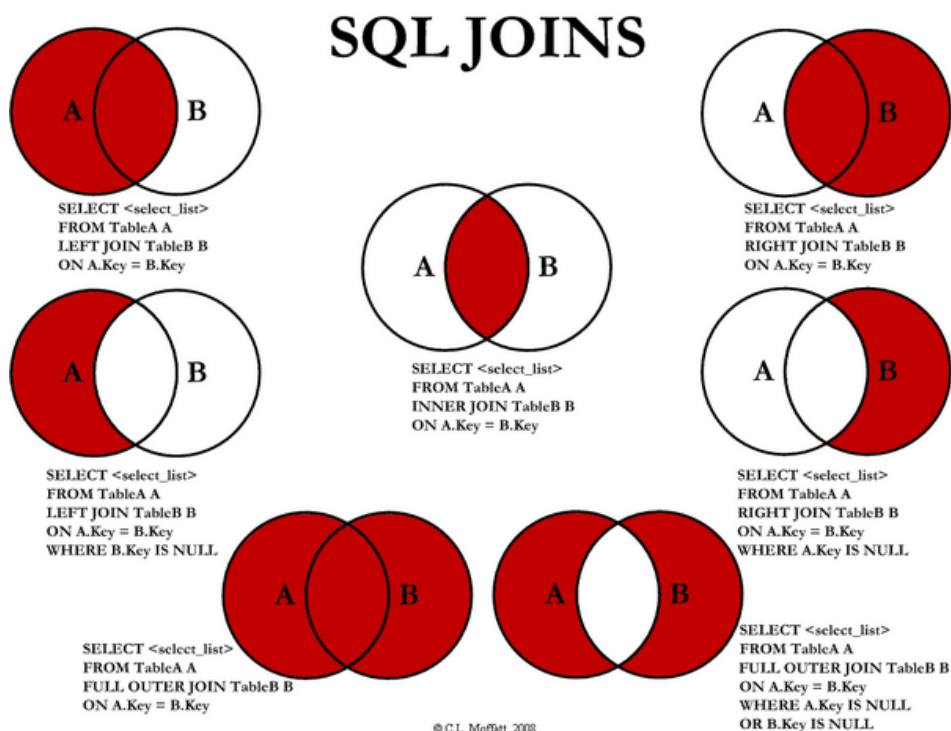
### Problem:

Also, how do LEFT JOIN, RIGHT JOIN and FULL JOIN fit in?

### Solution:

Assuming you're joining on columns with no duplicates, which is a very common case:

- An **INNER JOIN** of A and B gives the result of A intersect B, i.e. the inner part of a Venn diagram intersection.
- An **OUTER JOIN** of A and B gives the results of A union B, i.e. the outer parts of a Venn diagram union.
- A **LEFT OUTER JOIN** will give all rows in A, plus any common rows in B.
- A **RIGHT OUTER JOIN** will give all rows in B, plus any common rows in A.
- A **FULL OUTER JOIN** will give you the **union** of A and B, i.e. all the rows in A and all the rows in B. If something in A doesn't have a corresponding datum in B, then the B portion is null, and vice versa.



## Q18: How a database index can help performance? ☆☆☆

---

**Topics:** SQL Databases

### Answer:

The whole point of having an index is to speed up search queries by essentially cutting down the number of records/rows in a table that need to be examined. An index is a data structure (most commonly a B- tree) that stores the values for a specific column in a table.

## Q19: How does a *Hash* index work? ☆☆☆

---

**Topics:** SQL

### Problem:

Let's say that we want to run a query to find all the details of any employees who are named 'Abc'? How does a hash table index work?

```
SELECT * FROM Employee
WHERE Employee_Name = 'Abc'
```

### Solution:

The reason hash indexes are used is because hash tables are extremely efficient when it comes to just looking up values. So, queries *that compare for equality* to a string can retrieve values very fast if they use a hash index.

For instance, the query in the question could benefit from a hash index created on the `Employee_Name` column. The way a hash index would work is that the column value will be the key into the hash table and the actual value mapped to that key would just be a pointer to the row data in the table. Since a hash table is basically an associative array, a typical entry would look something like "Abc => 0x28939", where `0x28939` is a reference to the table row where `Abc` is stored in memory. Looking up a value like "Abc" in a hash table index and getting back a reference to the row in memory is obviously a lot faster than scanning the table to find all the rows with a value of "Abc" in the `Employee_Name` column.

## Q20: Discuss `INNER JOIN ON` vs `WHERE` clause (with multiple `FROM` tables) ☆☆☆

---

**Topics:** SQL

### Problem:

You can do:

```
SELECT
    table1.this, table2.that, table2.somethingelse
FROM
    table1, table2
WHERE
    table1.foreignkey = table2.primarykey
    AND (some other conditions)
```

Or else:



```
SELECT
    table1.this, table2.that, table2.somethingelse
FROM
    table1 INNER JOIN table2
    ON table1.foreignkey = table2.primarykey
WHERE
    (some other conditions)
```

What syntax would you choose and why?

### Solution:

- `INNER JOIN` is ANSI syntax that you should use. `INNER JOIN` helps human readability, and that's a top priority. It can also be easily replaced with an `OUTER JOIN` whenever a need arises.
- *Implicit* joins (with multiple `FROM` tables) become much much more confusing, hard to read, and hard to maintain once you need to start adding more tables to your query. The old syntax, with just listing the tables, and using the `WHERE` clause to specify the join criteria, is being deprecated in most modern databases.

# FullStack.Cafe - Kill Your Tech Interview

---

## Q1: How to select first 5 records from a table? ☆☆☆

---

Topics: SQL

Answer:

```
-- SQL Server
SELECT TOP 5 * FROM EMP;
```

```
-- Oracle
SELECT * FROM EMP WHERE ROWNUM <= 5;
```

```
-- Generic
SELECT name FROM EMPLOYEE o
WHERE (SELECT count(*) FROM EMPLOYEE i WHERE i.name < o.name) < 5
```

## Q2: What's the difference between a *Primary Key* and a *Unique Key*? ☆☆☆

---

Topics: SQL Databases

Answer:

Primary Key:

- There can only be one primary key in a table
- In some DBMS it cannot be NULL - e.g. MySQL adds NOT NULL
- Primary Key is a *unique* key identifier of the record
- Primary key can be created on multiple columns (composite primary key)

Unique Key:

- Can be more than one *unique* key in one table
- Unique key can have NULL values
- It can be a candidate key
- Unique key can be NULL ; multiple rows can have NULL values and therefore may not be considered "unique"

Difference between Primary Key and Unique key

- **1. Behavior:** Primary Key is used to identify a row (record) in a table, whereas Unique-key is to prevent duplicate values in a column (with the exception of a null entry).
- **2. Indexing:** By default SQL-engine creates Clustered Index on primary-key if not exists and Non-Clustered Index on Unique-key.
- **3. Nullability:** Primary key does not include Null values, whereas Unique-key can.

- **4. Existence:** A table can have at most one primary key, but can have multiple Unique-key.
- **5. Modifiability:** You can't change or delete primary values, but Unique-key values can.

### Q3: What is *Collation*? ☆☆☆

---

**Topics:** SQL

#### **Answer:**

In database systems, Collation specifies how data is sorted and compared in a database. Collation provides the sorting rules, case, and accent sensitivity properties for the data in the database.

For example, when you run a query using the `ORDER BY` clause, collation determines whether or not uppercase letters and lowercase letters are treated the same.

### Q4: Find *duplicate* values in a SQL table ☆☆☆

---

**Topics:** SQL MySQL

#### **Problem:**

We have a table

ID	NAME	EMAIL
1	John	asd@asd.com
2	Sam	asd@asd.com
3	Tom	asd@asd.com
4	Bob	bob@asd.com
5	Tom	asd@asd.com

I want is to get duplicates with the same `email` and `name`.

#### **Solution:**

Simply group on both of the columns:

```
SELECT
    name, email, COUNT(*) as CountOf
FROM
    users
GROUP BY
    name, email
HAVING
    COUNT(*) > 1
```

### Q5: How can `VIEW` be used to provide *security layer* for your app?

☆☆☆

---

**Topics:** MySQL SQL

#### **Answer:**

**Views** can be used to selectively *show limited data* to a user.

For example consider a *Customer* table which has columns including *name*, *location* and *credit card number*. As credit card number is a sensitive customer information, it might be required to hide credit card number from a certain database user. As the user can not be given access to entire *Customer* table, a view with *name* and *location* of the *Customer* can be created. The database user can be given access to that view only. This way view provides a security layer ensuring limited access to the database table.

## Q6: What's the difference between **Azure SQL Database** and **Azure SQL Managed Instance**? ☆☆☆

---

**Topics:** Azure SQL T-SQL

### Answer:

- With **Azure SQL Managed Instance**, you essentially get a full-fledged SQL Server that you can control any way you want, just like you would control a locally configured SQL Server. All the power and access and customization you want. With SQL Managed Instance, auditing happens at the server level, because, you are getting the full database server.
- With, **Azure SQL DB PaaS**, you are essentially getting a database service, so, you give up a lot of control. Resources dedicated to individual DBs like a container. They are grouped under an Azure SQL Server but that SQL Server is shared.

## Q7: What is the difference among **UNION** , **MINUS** and **INTERSECT** ?

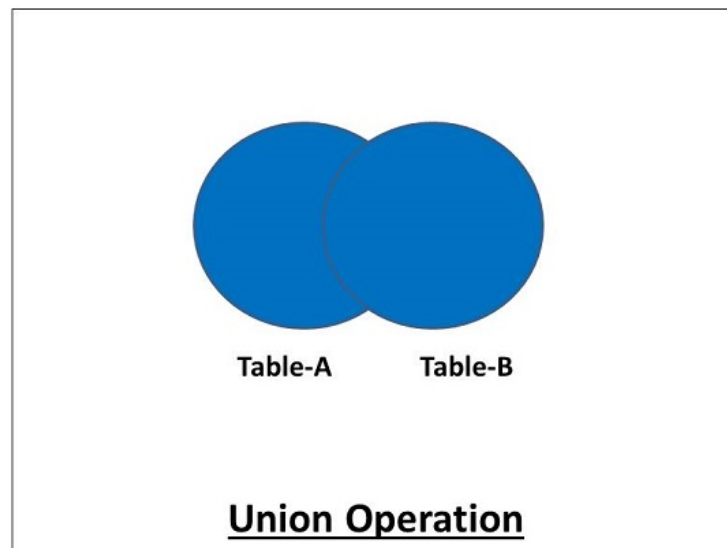
☆☆☆☆

---

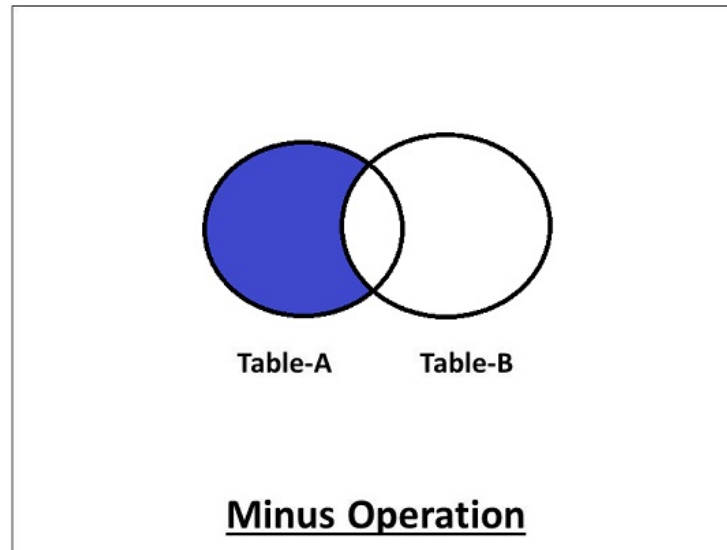
**Topics:** SQL

### Answer:

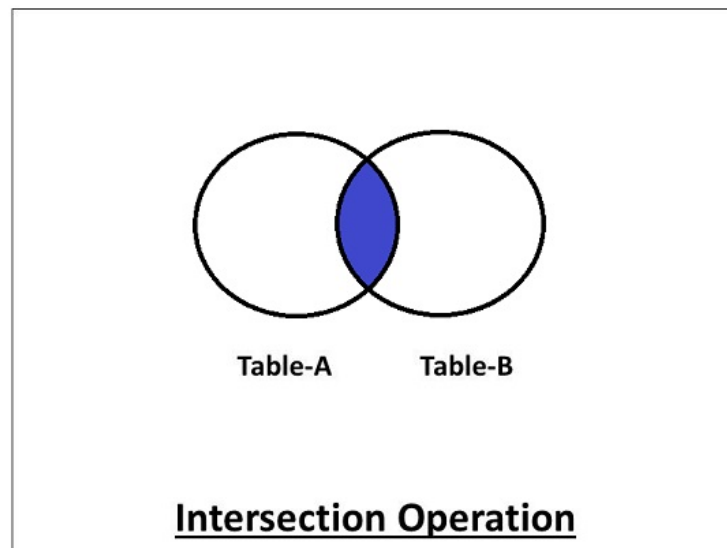
- **UNION** combines the results from 2 tables and eliminates duplicate records from the result set.



- **MINUS** operator when used between 2 tables, gives us all the rows from the first table except the rows which are present in the second table.



- `INTERSECT` operator returns us only the matching or common rows between 2 result sets.



**Q8: How can we *transpose* a table using SQL (changing rows to column or vice-versa)?** ☆☆☆☆

**Topics:** SQL

**Answer:**

A transposition is to rotate information from one row or column to another to change the data layout, for the purpose of making observations from a new perspective.

year	quarter	amount		---->	year	Q1	Q2	Q3	Q4
year2018	Q1	89			year2018	89	93	88	99
year2018	Q2	93			year2019	92	97	90	88
year2018	Q3	88							
year2018	Q4	99							
year2019	Q1	92							
year2019	Q2	97							
year2019	Q3	90							
year2019	Q4	88							

**Method 1:** case when subquery + grouping & aggregation

```
Select year, max(Q1) 'Q1', max(Q2) 'Q2', max (Q3) 'Q3', max (Q4) 'Q4'
from (
  select year,
    case when quarter = 'Q1' then amount end Q1,
    case when quarter = 'Q2' then amount end Q2,
    case when quarter = 'Q3' then amount end Q3,
    case when quarter = 'Q4' then amount end Q4
  from zz11
) t group by year;
```

**Method 2:** sum if + grouping & aggregation

```
SELECT year,
  MAX(IF(quarter = 'Q1', amount, null)) AS 'Q1',
  MAX (IF(quarter = 'Q2', amount, null)) AS 'Q2',
  MAX (IF(quarter = 'Q3', amount, null)) AS 'Q3',
  MAX (IF(quarter = 'Q4', amount, null)) AS 'Q4'
FROM zz11 GROUP BY year;
```

Other methods include WITH ROLLUP + grouping & aggregation and UNION + grouping & aggregation, etc. They are essentially the same: calculate the year value after grouping, and generate new columns Q1-Q4 through enumeration and their values through aggregation.

## Q9: How to generate row number in SQL *without* ROWNUM ☆☆☆☆

**Topics:** SQL

**Answer:**

Consider:

```
SELECT name, sal, (SELECT COUNT(*) FROM EMPLOYEE i WHERE o.name >= i.name) row_num
FROM EMPLOYEE o
order by row_num
```

## Q10: Explain the difference between *Exclusive Lock* and *Update Lock* ☆☆☆☆

---

**Topics:** SQL Databases

### Answer:

- In case of **Exclusive lock**, no other lock can be acquired on that row or table. Every process has to wait until the process which holds the lock releases it.
- In case of **Update lock**, while reading the row or a record, you can have any other lock associated with that row or record. In case of updating the record, update lock changes itself to an exclusive lock and no other process can obtain a lock on that row until the lock is released.

## Q11: How does *B-trees Index* work? ☆☆☆☆

---

**Topics:** Databases SQL

### Answer:

The main reason for the existence of **B-Tree Indexes** is to better utilize the behaviour of devices that read and write large chunks of data. Two properties are important to make the B-Tree better than binary trees when data has to be stored on disk:

- Access to disk is really slow (compared to memory or caches, random access to data on disk is orders of magnitude slower); and
- Every single read causes a whole sector to be loaded from the drive - assuming a sector size of 4K, this means 1000 integers, or tens of some larger objects you're storing.

Hence, we can use the pros of the second fact, while also minimizing the cons - i.e. number of disk accesses.

So, instead of just storing a single number in every node that tells us if we should continue to the left or to the right, we can create a bigger index that tells us if we should continue to the first 1/100, to the second or to the 99-th (imagine books in a library sorted by their first letter, then by the second, and so on). As long as all this data fits on a single sector, it will be loaded anyway, so we might as well use it completely.

This results in roughly  $\log_b N$  lookups, where  $N$  is the number of records. This number, while asymptotically the same as  $\log_2 N$ , is actually a few times smaller with large enough  $N$  and  $b$  - and since we're talking about storing data to disk for use in databases, etc., the amount of data is usually large enough to justify this.

## Q12: What is the *cost* of having a database *index*? ☆☆☆☆

---

**Topics:** Databases SQL

### Answer:

**It takes up space** - and the larger your table, the larger your index. Another performance hit with indexes is the fact that whenever you add, delete, or update rows in the corresponding table, the same operations will have to be done to your index. Remember that an index needs to contain the same up-to-the-minute data as whatever is in the table column(s) that the index covers.

As a general rule, an index should only be created on a table if the **data** in the indexed column will be **queried frequently**.

## Q13: How can I do an `UPDATE` statement with `JOIN` in SQL? ☆☆☆☆

---

**Topics:** SQL

### Problem:

Sale table:

```
id (int)
udid (int)
assid (int)
```

Ud table:

```
id (int)
assid (int)
```

sale.assid contains the correct value to update ud.assid .

What SQL query will do this? Could you do that without JOIN ?

### Solution:

```
update u
set u.assid = s.assid
from ud u
    inner join sale s on
        u.id = s.udid
```

Without JOIN :

```
update ud
set assid = sale.assid
from sale
where sale.udid = id
```

## Q14: What is faster, *one big query* or *many small queries*? ☆☆☆☆

**Topics:** MySQL SQL

### Answer:

What would address your question is the subject JOIN DECOMPOSITION.

You can decompose a join by running multiple single-table queries instead of a multitable join, and then performing the join in the application. For example, instead of this single query:

```
SELECT * FROM tag
JOIN tag_post ON tag_post.tag_id = tag.id
JOIN post ON tag_post.post_id = post.id
WHERE tag.tag = 'mysql';
```

You might run these queries:



```
SELECT * FROM tag WHERE tag = 'mysql';
SELECT * FROM tag_post WHERE tag_id=1234;
SELECT * FROM post WHERE post.id IN (123,456,567,9098,8904);
```

Why on earth would you do this? It looks wasteful at first glance, because you've increased the number of queries without getting anything in return. However, such restructuring can actually give significant performance advantages:

- Caching can be more efficient. Many applications cache "objects" that map directly to tables. In this example, if the object with the tag `mysql` is already cached, the application will skip the first query. If you find posts with an ID of 123, 567, or 908 in the cache, you can remove them from the `IN()` list. The query cache might also benefit from this strategy. If only one of the tables changes frequently, decomposing a join can reduce the number of cache invalidations.
- Executing the queries individually can sometimes reduce lock contention
- Doing joins in the application makes it easier to scale the database by placing tables on different servers.
- The queries themselves can be more efficient. In this example, using an `IN()` list instead of a join lets MySQL sort row IDs and retrieve rows more optimally than might be possible with a join.
- You can reduce redundant row accesses. Doing a join in the application means retrieving each row only once., whereas a join in the query is essentially a denormalization that might repeatedly access the same data. For the same reason, such restructuring might also reduce the total network traffic and memory usage.
- To some extent, you can view this technique as manually implementing a hash join instead of the nested loops algorithm MySQL uses to execute a join. A hash join might be more efficient.

As a result, doing joins in the application can be more efficient when you cache and reuse a lot of data from earlier queries, you distribute data across multiple servers, you replace joins with `IN()` lists, or a join refers to the same table multiple times.

## Q15: What is *Optimistic Locking* and *Pessimistic Locking*? ☆☆☆☆☆

**Topics:** SQL Databases

### Answer:

#### In Short:

- **Optimistic** assumes that nothing's going to change while you're reading it.
- **Pessimistic** assumes that something will and so locks it.

#### Detailed answer:

- **Optimistic Locking** is a strategy where you read a record, take note of a version number (other methods to do this involve dates, timestamps or checksums/hashes) and check that the version *hasn't changed before you write* the record back. When you write the record back you filter the update on the version to make sure it's atomic. (i.e. hasn't been updated between when you check the version and write the record to the disk) and update the version in one hit.

If the record is *dirty* (i.e. different version to yours) you abort the transaction and the user can re-start it.

This strategy is most applicable to high-volume systems and three-tier architectures where you do not necessarily maintain a connection to the database for your session. In this situation, the client cannot actually maintain database locks as the connections are taken from a pool and you may not be using the same connection from one access to the next.

- **Pessimistic Locking** is when you *lock the record* for your exclusive use until you have finished with it. It has much better integrity than optimistic locking but requires you to be careful with your application design to avoid *Deadlocks*. To use pessimistic locking you need either a direct connection to the database (as would

typically be the case in a two tier client server application) or an externally available transaction ID that can be used independently of the connection.

The disadvantage of pessimistic locking is that a resource is locked from the time it is first accessed in a transaction until the transaction is finished, making it inaccessible to other transactions during that time.

## Q16: How does database *Indexing* work? ☆☆☆☆☆

---

**Topics:** SQL Databases

### Answer:

**Indexing** is a way of sorting a number of records on multiple fields. Creating an index on a field in a table creates another data structure (stored on the disc) which holds the field value, and a pointer to the record it relates to. This index structure is then sorted, allowing Binary Searches to be performed on it, which has  $\log_2 N$  block accesses. Also since the data is sorted given a non-key field, the rest of the table doesn't need to be searched for duplicate values, once a higher value is found.

The whole point of having an index is to **speed up search queries** by essentially cutting down the number of records/rows in a table that need to be examined. An index is a data structure (most commonly a B- tree or hash table) that stores the values for a specific column in a table.

Given the nature of a binary search, the **cardinality or uniqueness** of the data is important. Indexing on a field with a cardinality of 2 would split the data in half, whereas a cardinality of 1,000 would return approximately 1,000 records.

## Q17: Name some disadvantages of a *Hash index* ☆☆☆☆☆

---

**Topics:** SQL Databases

### Answer:

- **Hash** tables are *not sorted* data structures, and there are many types of queries that hash indexes can not even help with.

For instance, suppose you want to find out all of the employees who are less than 40 years old. How could you do that with a hash table index? Well, it's not possible because a hash table is only good for looking up key-value pairs – which means queries that check just for *equality*.

## Q18: What are some *other* types of Indexes (vs B-Trees)? ☆☆☆☆☆

---

**Topics:** SQL Databases

### Answer:

- Indexes that use a **R-tree** data structure are commonly used to help with *spatial* problems. For instance, a query like “Find all of the Starbucks within 2 kilometers of me” would be the type of query that could show enhanced performance if the database table uses a R-tree index.
- Another type of index is a **bitmap index**, which works well on columns that contain Boolean values (like true and false), but many instances of those values – basic columns with low selectivity.

## Q19: What is the difference between *B-Tree*, *R-Tree* and *Hash indexing*? ☆☆☆☆☆

---

**Topics:** SQL Databases

### Answer:

**BTree** (in fact B\*Tree) is an efficient ordered key-value map. Meaning:

- given the key, a BTree index can quickly find a record,
- a BTree can be scanned in order.
- it's also easy to fetch all the keys (and records) within a range.

**e.g.** "all events between 9am and 5pm", "last names starting with 'R'"

**RTree** is a **spatial index** which means that it can quickly identify **close** values in 2 or more dimensions. It's used in geographic databases for queries such as:

all points within X meters from (x,y)

**Hash** is an unordered key-value map. It's even more efficient than a BTree:  $O(1)$  instead of  $O(\log n)$ . But it doesn't have any concept of order so it can't be used for sort operations or to fetch ranges.

## Q20: What is *Index Cardinality* and why does it matter? ☆☆☆☆☆

**Topics:** Databases MongoDB SQL

### Answer:

The **Index Cardinality** refers to how many possible values there are for a field.

Imagine we have a collection of all humans on earth with the following field:

```
- "sex" // 99.9% of the time "male" or "female", but string nonetheless (not binary)
- "name"
- "phone"
- "email"
```

The field **sex** only has two possible values. It has a very **low cardinality**. Other fields such as **names**, **usernames**, **phone numbers**, **emails**, etc. will have a more unique value for every document in the collection, which is considered **high cardinality**.

It's a rule of thumb to create indexes on **high-cardinality** keys or put **high-cardinality** keys *first* in the compound index.

### Why?

The greater the cardinality of a field the more helpful an index will be, **because indexes narrow the search space, making it a much smaller set.**

Suppose you have an index on **sex** and are looking for men named John. You would only narrow down the resulting space by approximately **50%** if you indexed by **sex** first. Conversely, if you indexed by **name**, you would immediately narrow down the result set to a minute fraction of users named John, and then you would refer to those documents to check the gender.

# FullStack.Cafe - Kill Your Tech Interview

## Q1: How does TRUNCATE and DELETE operations effect *Identity*?

☆☆☆☆

Topics: SQL T-SQL

### Problem:

Explain based on this table example:

```
USE [TempDB] GO
-- Create Table
CREATE TABLE [dbo].[TestTable](
[ID] [int] IDENTITY(11,1) NOT NULL,
[var] [nchar](10) NULL
) ON [PRIMARY] GO
-- Build sample data
INSERT INTO [TestTable] VALUES ('val')
GO
```

### Solution:

- When the DELETE statement is executed without WHERE clause it will delete all the rows. However, when a new record is inserted the identity value is increased from 11 to 12. *It does not reset identity* but keeps on increasing.
- When the TRUNCATE statement is executed it will remove all the rows. However, when a new record is inserted the identity value is increased from 11 (which is the original value). TRUNCATE *resets* the identity value to the original seed value of the table.

## Q2: What would happen *without* an Index? ☆☆☆☆

Topics: SQL

### Problem:

Let's say that we want to run a query to find all the details of any employees who are named 'Abc'? What would happen without an index?

```
SELECT * FROM Employee
WHERE Employee_Name = 'Abc'
```

### Solution:

Database software would literally have to look at every single row in the Employee table to see if the Employee\_Name for that row is 'Abc'. And, because we want every row with the name 'Abc' inside it, we can not just stop looking once we find just one row with the name 'Abc', because there could be other rows with the name Abc. So, every row up until the last row must be searched – which means thousands of rows in this scenario will have to be examined by the database to find the rows with the name 'Abc'.

This is what is called a **full table scan**.

### Q3: Delete *duplicate* values in a SQL table ☆☆☆☆

Topics: SQL

#### Problem:

We have a table:

ID	NAME	EMAIL
1	John	asd@asd.com
2	Sam	asd@asd.com
3	Tom	asd@asd.com
4	Bob	bob@asd.com
5	Tom	asd@asd.com

How to delete duplicates from it (with the same `name` and `email`)?

#### Solution:

Consider:

```
DELETE FROM users
WHERE id IN (
  SELECT id/*, name, email*/
  FROM users u, users u2
  WHERE u.name = u2.name AND u.email = u2.email AND u.id > u2.id
)
```

Or using `PARTITION BY` :

```
DELETE d
FROM @YourTable d
INNER JOIN (SELECT
  y.id,y.name,y.email,ROW_NUMBER() OVER(PARTITION BY y.name,y.email ORDER BY
y.name,y.email,y.id) AS RowRank
FROM @YourTable y
INNER JOIN (SELECT
  name,email, COUNT(*) AS CountOf
FROM @YourTable
GROUP BY name,email
HAVING COUNT(*)>1
) dt ON y.name=dt.name AND y.email=dt.email
) dt2 ON d.id=dt2.id
WHERE dt2.RowRank!=1
SELECT * FROM @YourTable
```

### Q4: Select first row in each `GROUP BY` group (greatest-n-per-group problem)? ☆☆☆☆☆

Topics: SQL

#### Problem:

Purchases table:

id	customer	total
1	Joe	5
2	Sally	3
3	Joe	2
4	Sally	1

I'd like to query for the id of the largest purchase (total) made by each customer.

### Solution:

Consider:

```
WITH summary AS (  
    SELECT p.id,  
           p.customer,  
           p.total,  
           ROW_NUMBER() OVER(PARTITION BY p.customer  
                               ORDER BY p.total DESC) AS rk  
    FROM PURCHASES p)  
SELECT s.*  
FROM summary s  
WHERE s.rk = 1
```