# Short Answer

Answer the following questions with complete sentences in your own words. You are encouraged to conduct your own research online or through other methods before answering the questions. If you research online, please consult multiple sources before you write down your answers. You are expected to be able to explain your answers in detail (Provide examples for each question).

1. What is the aggregate function? How do you use it?

● Aggregate functions are build-in functions that used to perform simple statistics

○ COUNT()

○ AVG()

○ SUM()

○ MAX()

○ MIN()

● The COUNT() function returns the number of rows that matches a specified criterion.

● The AVG() function returns the average value of a numeric column.

● The SUM() function returns the total sum.

● The MIN() and MAX() functions return the smallest and largest values of the selected columns

respectively.

● How does aggregate functions handle Null value?

○ COUNT(\*) and COUNT(1) gives the total number of records in the table including Null values,

there is no difference between these two functions

○ COUNT(column\_name) only considers rows where the column contains a Not-Null value

○ AVG, MIN, MAX, etc. ignore Null values

○ GROUP BY includes a row for null

In SQL, an aggregate function performs a calculation on a set of values and returns a single result. Some common aggregate functions include **SUM**, **AVG**, **MIN**, **MAX**, and **COUNT**.

To use an aggregate function, you can include it in the **SELECT** statement followed by a column name or expression. The result of the aggregate function will be calculated for all rows in the query and returned as a single value.

Here is an example of using the **SUM** aggregate function to calculate the total salary of all employees in a table called **Employees**:

SELECT SUM(salary) AS total\_salary

FROM Employees;

This will return a single row with a column called **total\_salary**, which is the sum of all the **salary** values in the **Employees** table.

You can also use the **GROUP BY** clause to group the results of a query by one or more columns and apply an aggregate function to each group. For example:

SELECT department, SUM(salary) AS total\_salary

FROM Employees

GROUP BY department;

This will return a separate row for each department, with a column called **total\_salary** that shows the sum of the salaries for all employees in that department.

You can also use the **HAVING** clause to filter the results of a query based on an aggregate function. For example:

SELECT department, SUM(salary) AS total\_salary

FROM Employees

GROUP BY department

HAVING SUM(salary) > 100000;

1. Can aggregate functions be used without having and group by? Can it be used with having but not group by? If so, what would the result be? If not, why?

● Having clause is used to filter out grouping records, it’s like the WHERE

clause. The difference is WHERE clause cannot be used with aggregate

functions, whereas Having clause can work with

● Having clause must come after GROUP BY clause and before ORDER BY

clause

● Optional

Graphical user interface, application, table

Description automatically generated

Yes, aggregate functions can be used without the **GROUP BY** or **HAVING** clauses. In this case, the aggregate function will be calculated for all rows in the query and returned as a single value.

For example:

SELECT SUM(salary) AS total\_salary

FROM Employees;

This will return a single row with a column called **total\_salary**, which is the sum of all the **salary** values in the **Employees** table.

However, aggregate functions cannot be used with the **HAVING** clause without the **GROUP BY** clause. The **HAVING** clause is used to filter the results of a **GROUP BY** clause, so it must be used in conjunction with **GROUP BY**. If you try to use **HAVING** without **GROUP BY**, you will get an error.

For example, the following query will produce an error:

SELECT SUM(salary) AS total\_salary

FROM Employees

HAVING SUM(salary) > 100000;

To use the **HAVING** clause in this case, you would need to use the **GROUP BY** clause to group the results by some column, like this:

SELECT department, SUM(salary) AS total\_salary

FROM Employees

GROUP BY department

HAVING SUM(salary) > 100000;

1. What’s the difference between WHERE clause and HAVING clause?

● The SELECT statement is used to select data from a database.

● The WHERE clause is used to filter records

● The data returned is stored in a result table, called the result set.

● Example:

Table

Description automatically generated with medium confidence

Text

Description automatically generated Graphical user interface, application

Description automatically generated

● Having clause is used to filter out grouping records, it’s like the WHERE

clause. The difference is WHERE clause cannot be used with aggregate

functions, whereas Having clause can work with

● Having clause must come after GROUP BY clause and before ORDER BY

clause

● Optional

Graphical user interface, application, table

Description automatically generated

The **WHERE** and **HAVING** clauses are used to filter the results of a SQL query. However, they are used in different ways and at different stages of the query processing.

The **WHERE** clause is used to filter rows before the **GROUP BY** clause is applied. It can be used to filter rows based on any column or expression in the query, and it is applied to each row individually.

For example:

SELECT department, SUM(salary) AS total\_salary

FROM Employees

WHERE salary > 50000

GROUP BY department;

In this example, the **WHERE** clause filters out all rows with a **salary** less than or equal to 50000 before the **GROUP BY** clause is applied. The **GROUP BY** clause then groups the remaining rows by department, and the **SUM** function calculates the total salary for each department.

The **HAVING** clause is used to filter rows after the **GROUP BY** clause is applied. It can be used to filter groups based on an aggregate function, and it is applied to each group individually.

For example:

SELECT department, SUM(salary) AS total\_salary

FROM Employees

GROUP BY department

HAVING SUM(salary) > 100000;

In this example, the **GROUP BY** clause groups the rows by department, and the **SUM** function calculates the total salary for each department. The **HAVING** clause then filters out all groups with a total salary less than or equal to 100000.

In summary, the **WHERE** clause is used to filter rows before grouping, while the **HAVING** clause is used to filter groups after grouping.

4. What are joins? What are the basic types of joins?

Inner Join: Returns records that have matching values in both tables

Left Join: Returns all records from the left table, and the matched records from the right table

Right Join: Returns all records from the right table, and the matched records from the left table

Full Join: Returns all records when there is a match in either left or right table

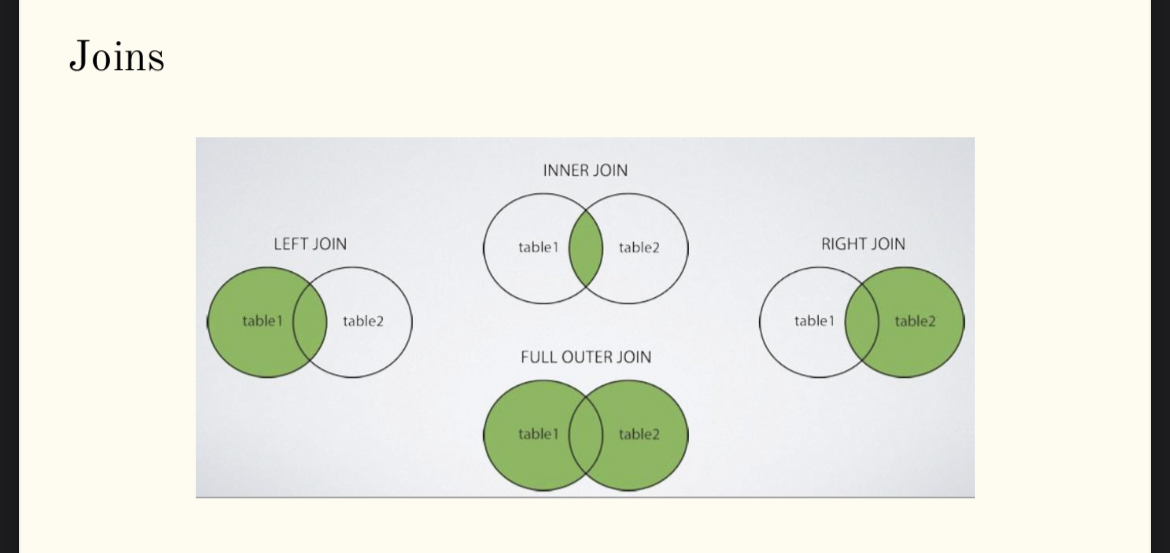
● A JOIN clause is used to combine rows from 2 or more tables, based on a matching column.

● Uses matching data in specified columns to combine or sort data.

● Columns DO NOT have to have the same name.

● Columns DO NOT need to be keys.

● Scope: table to table, table to view, table to synonyms.



Inner Join

● Inner Join returns records that have matching values in both tables.

● Inner Join selects all rows from both tables as long as there is a match between

the columns. If there are records in one table that do not have matches in the other table these records will not be shown.

Graphical user interface, text, application

Description automatically generated

● Query all the students who enroll in the courses and list their enrolled courses

Table

Description automatically generated

Left/Right Join

● The LEFT JOIN keyword returns all records from the left table and the matching records from the right table.

● If there is no matching record for the left table, NULL will be assigned in the result set.

● It’s a good idea to choose one direction (either LEFT or RIGHT) and use it to maintain consistency.

Table

Description automatically generated with low confidence

Full Outer Join

● Used to match up tuples from different relations

● Includes all the relations from both sides

● If there is no matching tuple, shows NULL

Graphical user interface, application

Description automatically generated

In a database, a join is a query that combines rows from two or more tables based on a related column between them.

There are several types of joins:

INNER JOIN: This type of join returns only the rows that match the join condition. If there is no match, the result will not include the rows from either table.

OUTER JOIN: There are three types of outer joins:

LEFT JOIN or LEFT OUTER JOIN: This type of join returns all the rows from the left table and the matching rows from the right table. If there is no match, it will return NULL for the right-side columns.

RIGHT JOIN or RIGHT OUTER JOIN: This type of join returns all the rows from the right table and the matching rows from the left table. If there is no match, it will return NULL for the left-side columns.

FULL OUTER JOIN: This type of join returns all rows from both tables, regardless of whether there is a match. If there is no match, it will return NULL for the non-matching columns.

CROSS JOIN: This type of join returns the Cartesian product of the two tables. It combines every row from the left table with every row from the right table.

Here's an example of an INNER JOIN using SQL:

SELECT \*

FROM table1 INNER JOIN table2

ON table1.id = table2.id;

This will return all rows from table1 and table2 where the id column is the same.

5. What kind of relationship does self-join imply? Can you give an example of self-join?

Self Join

● SELF JOIN is used when a JOIN is used on the same table.

Graphical user interface, text

Description automatically generated

A self-join is a type of join in which a table is joined to itself. It is used to compare values in the same table, especially when the table has a self-referencing foreign key.

Self-joins are useful when you want to compare rows within a table or when you want to create a hierarchical structure with a table that references itself.

Here's an example of a self-join using SQL:

SELECT a.name, b.name

FROM table1 a INNER JOIN table1 b

ON a.manager\_id = b.id

WHERE a.id != b.id;

This query will return the names of all employees and their managers. The table1 table has a self-referencing foreign key called manager\_id that points to the id column of the same table. The INNER JOIN condition compares the manager\_id and id columns to find the matching rows. The WHERE clause filters out the rows where the id and manager\_id are the same, which would be the managers without a manager (they manage themselves).

6. What are sub-query? What are the different types of sub-queries?

● A subquery is a query that is nested inside a SELECT, INSERT, UPDATE, DELETE statement or inside another subquery.

● Subqueries must be written in parentheses.

● Subqueries must include the SELECT clause and the FROM clause.

Text

Description automatically generated

Where to Put a Subquery

● SELECT clause

○ Can be used to retrieve values in a select clause, but only if they return a single result

Text

Description automatically generated

FROM clause

○ Can be used to return an entire table but must have an alias

○ Derived Table

Text

Description automatically generated

● WHERE clause

○ Most common use

○ Used to filter results based on another table

A screenshot of a computer

Description automatically generated with medium confidence

A subquery is a SELECT statement that is embedded in another SELECT, INSERT, UPDATE, or DELETE statement, or in the clause of a CREATE VIEW statement. A subquery is also called an inner query or inner select, while the statement containing the subquery is called an outer query or outer select.

There are several types of subqueries:

Single-row subquery: This type of subquery returns only one row. It is used in the WHERE, HAVING, and SELECT clauses.

Multiple-row subquery: This type of subquery returns multiple rows. It is used in the WHERE, HAVING, and SELECT clauses.

Multiple-column subquery: This type of subquery returns multiple columns. It is used in the SELECT and WHERE clauses.

Correlated subquery: This type of subquery is executed once for each row in the outer query. It is used in the WHERE and SELECT clauses.

Here's an example of a single-row subquery using SQL:

SELECT \*

FROM table1

WHERE value = (SELECT MAX(value) FROM table2);

This query will return all rows from table1 where the value column is equal to the maximum value in the value column of table2. The subquery in the WHERE clause is a single-row subquery because it returns only one row (the maximum value).

Here's an example of a multiple-row subquery using SQL:

SELECT \*

FROM table1

WHERE value IN (SELECT value FROM table2 WHERE column1 = 'A');

This query will return all rows from table1 where the value column is equal to any of the values in the value column of table2 where the column1 is 'A'. The subquery in the WHERE clause is a multiple-row subquery because it returns multiple rows.

7. What are set operators? How is it different from a join?

join combine columns from separate tables; whereas, set operations combine rows from separate tables.

SQL set operators UNION, INTERSECT, and EXCEPT

Chart, bubble chart

Description automatically generated

Set Operations

● Results are unordered. It could be useful to perform operations on these: Union

Intersection Difference

● Different RDBMS provide different levels of support

Intersection

● Intersection Implemented via INTERSECT

Diagram

Description automatically generated

Difference

● Difference Implemented via EXCEPT

○ Display the values in the first select statement MINUS any values found in the second select

statement

Diagram

Description automatically generated

Set operators are used to combine the results of two or more SELECT statements into a single result. Set operators include UNION, UNION ALL, INTERSECT, and MINUS.

Set operators differ from joins in several ways:

Joins combine rows from two or more tables based on a join condition, while set operators combine the results of two or more SELECT statements.

Joins can be INNER, OUTER, or CROSS, while set operators are always INNER.

Joins return a result set with additional columns from the joined tables, while set operators return a result set with only the columns specified in the SELECT statements.

Here's an example of a UNION operator using SQL:

SELECT column1, column2 FROM table1

UNION

SELECT column1, column2 FROM table2;

This query will return the union of the rows returned by the two SELECT statements. The result will include only the rows that appear in either table1 or table2.

Here's an example of a UNION ALL operator using SQL:

SELECT column1, column2 FROM table1

UNION ALL

SELECT column1, column2 FROM table2;

This query will return the union of the rows returned by the two SELECT statements. The result will include all rows from both table1 and table2, even if there are duplicates.

Here's an example of an INTERSECT operator using SQL:

SELECT column1, column2 FROM table1

INTERSECT

SELECT column1, column2 FROM table2;

This query will return the intersection of the rows returned by the two SELECT statements. The result will include only the rows that appear in both table1 and table2.

Here's an example of a MINUS operator using SQL:

SELECT column1, column2 FROM table1

MINUS

SELECT column1, column2 FROM table2;

This query will return the rows from the first SELECT statement that are not in the result of the second SELECT statement. The result will include only the rows that appear in table1 but not in table2.

8. What are the differences between UNION and UNION ALL? What are some rules

you have to follow when you use set operators?

Union command concatenates the results of two queries into a single result set. Union All includes duplicates while Union excludes them.

1st column is sorted automatically for Union, Union All does not

Union cannot be used in recursive CTE while Union All can be used

UNION and UNION ALL

● UNION- eliminates duplicates

● UNION ALL- does NOT eliminate duplicates

● Uses the column names from the first result set

● Data types must match

● Number of attributes must match

Diagram

Description automatically generated

UNION and UNION ALL Example

● R(RName) ● S(SName)

A picture containing graphical user interface

Description automatically generated

UNION and UNION ALL are set operators in SQL that are used to combine the results of two or more SELECT statements.

The main difference between UNION and UNION ALL is that UNION removes duplicates from the final result set, while UNION ALL does not. This means that if a row appears multiple times in the combined results of the SELECT statements, it will only be included once in the final result set when you use UNION, but it will be included multiple times when you use UNION ALL.

Here are some rules to follow when using set operators:

The number and data types of the columns selected by the SELECT statements must be the same.

The ORDER BY clause can only be used with the last SELECT statement in the set.

The LIMIT clause can only be used with the last SELECT statement in the set.

The set operators can only be used with SELECT statements, not with other types of statements such as INSERT, UPDATE, or DELETE.

When using the set operators, you should use parentheses to enclose each SELECT statement and the set operator.

For example:

(SELECT column1, column2 FROM table1 WHERE condition1)

UNION

(SELECT column1, column2 FROM table2 WHERE condition2)

9. What are the views? How is it different from a table? What happens if you modify data on view?

View and Table both are integral parts of a relational database, and both terms are used interchangeably. The view is a result of an SQL query and it is a virtual table, whereas a Table is formed up of rows and columns that store the information of any object and be used to retrieve that data whenever required. In this article, we are going to discuss the difference between View and Table.

A view contains no data of its own but it is like a ‘window’ through which data from tables can be viewed or changed. A view is a query of one or more tables that provides another way of presenting the information. In layman’s terms, a view is a “stored query “.

Graphical user interface, text, application

Description automatically generated

<https://www.geeksforgeeks.org/difference-between-view-and-table/>

View

● A view is often seen as a virtual table

● It displays data that you choose, but does not actually hold any data

● Good for security since you can prevent showing extra data

● DML operations just happen on the table.

○ you can modify data on view level, and the source data will be updated as well.Graphical user interface, text, application

Description automatically generated

In a database, a view is a virtual table that is created based on a SELECT statement. A view is like a saved SELECT statement that can be used to retrieve data from one or more tables.

Views are different from tables in a few ways:

A view does not store data itself. It displays data from the tables that the view is based on.

You cannot add, update, or delete data directly in a view. If you want to modify data, you need to update the underlying tables that the view is based on.

Views do not take up storage space in the database. They are just a way to display data from the tables in a different way.

Views can be used to simplify complex queries by breaking them down into smaller, more manageable pieces.

If you modify data in a view, the changes will be reflected in the underlying tables that the view is based on. For example, if you use an UPDATE statement to modify data in a view, the changes will be made to the rows in the underlying tables that the view is based on. However, you may not be able to modify data in a view if the view is created with the WITH CHECK OPTION clause, which prevents the modification of data that would not be displayed in the view.

10. What is a user-defined stored procedure? Why do you need it?

What is a Stored Procedure?

A SQL stored procedure (SP) is a collection of SQL statements and SQL command logic, which is compiled and stored in the database. Stored procedures in SQL allow us to create SQL queries to be stored and executed on the server. Stored procedures can also be cached and reused. The main purpose of stored procedures is to hide direct SQL queries from the code and improve the performance of database operations such as select, update, and delete data. Here is a detailed article on Stored Procedures in SQL.

SP is used for DML statements while function is used for calculation

SP use Exec to call while function use query and input to call

SP may or may not need any input while function needs at least one

SP may or may not need any output while function must have output

SP can call function but function cannot call SP

Triggers is a stored procedure that runs automatically when some specific event happens(insert| update | delete).

A trigger is a special type of stored procedure that automatically runs when an event occurs in the database server. DML triggers run when a user tries to modify data through a data manipulation language (DML) event. DML events are INSERT, UPDATE, or DELETE statements on a table or view.

<https://learn.microsoft.com/en-us/sql/t-sql/statements/create-trigger-transact-sql?view=sql-server-ver16>

Triggers are special kind of stored procedure. They can be executed after or before data modification happens on a table. There are two types of triggers “Instead of triggers” and “After triggers”.

"Instead of triggers" executes prior to data modification while "after trigger" executes after data modification.

Types of Stored Procedures in SQL Server

There are two types of stored procedures available in SQL Server.

User-defined stored procedures

System stored procedures

User-defined stored procedures

User-defined stored procedures are created by database developers or database administrators. These SPs contain one more SQL statements to select, update, or delete records from database tables. User-defined stored procedures can take input parameters and return output parameters. A user-defined stored procedure is a mixture of DDL (Data Definition Language) and DML (Data Manipulation Language ) commands.

User defined SPs are further classified into two types

T-SQL stored procedures- T-SQL (Transact SQL) SPs receive and return parameters. These SPs process the Insert, Update and Delete queries with or without parameters and return data of rows as output. This is one of the most common ways to write SPs in SQL Server.

CLR stored procedures- CLR (Common Language Runtime) SPs are written in a CLR-based programming language such as C# or VB.NET and are executed by the .NET Framework.

<https://www.c-sharpcorner.com/article/how-to-create-a-stored-procedure-in-sql-server-management-studio/#:~:text=User%2Ddefined%20stored%20procedures%20are,parameters%20and%20return%20output%20parameters>.

Stored Procedure

● In SQL Server, a stored procedure is a set of T-SQL statements that are compiled and stored in the database. The stored procedure accepts input and output parameters, executes the SQL statements, and returns a result set if any.

● System procedures: System procedures are included with SQL Server and are physically stored in the internal, hidden Resource database and logically appear in the sys schema of all the databases. The system-stored procedures start with the sp\_ prefix.

https://learn.microsoft.com/en-us/sql/relational-databases/system-stored- procedures/system-stored-procedures-transact-sql?view=sql-server-ver16

User Defined Stored Procedure

● User Defined Stored Procedures are just Stored Procedures, but created by the user

● Contains statements including calling other stored procedures

● Can have different Input and Output Parameters

● Must be recompiled after time or changes

Graphical user interface, text, application

Description automatically generated

User Defined Stored Procedure

● (basic)

● A stored procedure can have zero or more

INPUT and OUTPUT parameters.

Graphical user interface, application

Description automatically generated

User Defined Stored Procedure

● (SP w/ input parameters)

● Each parameter is assigned a name, and a data type, if no other statement

follows, then this parameter is treated as an INPUT parameter

A picture containing graphical user interface

Description automatically generated

● (SP w/ both INPUT and OUTPUT)

● Stored procedures can return a value to the calling program if the parameter is

specified as OUTPUT.

Graphical user interface, application

Description automatically generated

A user-defined stored procedure is a stored procedure that is created by a user (as opposed to a system-defined stored procedure, which is provided by the database system). User-defined stored procedures allow you to create your own custom functions that can be used to perform specific tasks in the database.

There are several reasons why you might need a user-defined stored procedure:

To encapsulate complex logic or business rules in a reusable form.

To improve the performance of the database by pre-compiling the stored procedure and storing it in memory.

To provide a way to abstract complex queries and make them easier to use and understand.

To enforce data integrity and consistency by centralizing data validation and processing logic in a stored procedure.

To improve security by limiting access to sensitive data and restricting which users can execute certain tasks.

To allow multiple users to share a common set of business logic and database access patterns.

User-defined stored procedures can be created using the CREATE PROCEDURE statement in SQL. They can take input parameters and return output values, and they can be called from within other SQL statements or from within application code.

11. What are indexes? Why are they needed?

Index

● It's used to sort and optimize data fetch time

● Operate similar to index in a book

● When created, an index will create a dynamic Balance Tree (B+ Tree)

● Primary key creates Clustered Index, unique key creates Non-Clustered Index

● Tables without a Clustered Index are called HEAP Tables

● Keys ≠ Indexes

● If a table has no index at all, when new data is inserted, they are added wherever there is free space, and in no particular order.

Diagram

Description automatically generated with medium confidence

In a database, an index is a data structure that allows you to quickly find and retrieve rows from a table based on the values in one or more columns. Indexes are used to improve the performance of SELECT, UPDATE, and DELETE statements by providing a faster way to locate the rows that you want to retrieve or modify.

There are several types of indexes, including:

Clustered indexes: These indexes reorder the rows in the table based on the index key values. There can only be one clustered index per table.

Non-clustered indexes: These indexes create a separate data structure that stores the index key values and a reference to the rows in the table. There can be multiple non-clustered indexes per table.

Unique indexes: These indexes enforce the uniqueness of the index key values, ensuring that no two rows in the table have the same values for the indexed columns.

Full-text indexes: These indexes are used to search for specific words or phrases in large amounts of text data.

You need indexes in a database because they can significantly improve the performance of queries that search for specific rows in a table. Without an index, the database system would have to scan the entire table to find the rows that match the search criteria, which can be time-consuming for large tables. By creating an index, the database system can quickly locate the rows that match the search criteria using the index data structure.

It's important to choose the right columns to index, because creating too many indexes can slow down INSERT, UPDATE, and DELETE statements, as the indexes need to be updated every time the data in the table is modified.

Index is an on-disk structure associated with a table that increases the retrieval speed of rows from a table, to reduce disk I/O, allowing SQL Server to find data in a table without scanning the entire table.

Clustered index: sorts the results, only one per table, associated with PK.

Nonclustered index: does not sort data, more than one per table (max 999), stored separately from clustered index.

Advantages: Increase performance since increase retrieval speed

Disadvantages:

* May slow down other DML: Insert, Update, Delete
* Requires a lot of spaces
* Slow down query if there are too many indexes, ideally is 3-4

12. Is the index always useful? Do you want to create as many indexes as

possible? Why or why not?

Indexes can be useful in improving the performance of SELECT, UPDATE, and DELETE statements, but they may not always be necessary or beneficial.

While creating an index can improve the performance of a SELECT statement, it can also have a negative impact on the performance of INSERT, UPDATE, and DELETE statements. This is because every time you insert, update, or delete a row in a table, the indexes on that table also need to be updated to reflect the changes. This can be time-consuming for large tables with many indexes, and can decrease the overall performance of the database.

In general, you should only create an index if it will be used frequently and will significantly improve the performance of your queries. It is not generally a good idea to create as many indexes as possible, as this can lead to decreased performance and increased overhead.

To determine whether an index is necessary, you should consider the following factors:

The size of the table: If the table is small, an index may not be necessary because the database system can quickly scan the entire table to find the rows you want.

The workload of the database: If the database is heavily used for INSERT, UPDATE, and DELETE operations, creating too many indexes can decrease performance.

The data distribution in the table: If the data in the table is evenly distributed, an index may not be necessary because the database system can quickly locate the rows you want using a table scan.

The columns you are querying: If you are querying a column with a high degree of uniqueness (such as a primary key column), an index may not be necessary because the database system can quickly locate the rows you want using the unique values in the column.

It's important to carefully consider the trade-offs between the benefits of an index and the potential impact on performance before creating an index.

Why shouldn't I create a lot of indexes?

Indexes have overhead for every INSERT/DELETE on the table. Every INSERT must add a row to each/every nonclustered index. Every DELETE must remove a row from each/every nonclustered index.

Indexes require disk space. While nonclustered indexes are generally a lot smaller than the table itself their total size (of all indexes together) can often exceed the size of the table by 2 or 3 times. This can be quite normal. However, a table that is poorly indexed and severely over indexed might have index space which is 6-10 times the size of the base table. This is wasted space on disk and also in cache. While disk space is relatively cheap, memory is not. But, that's not even the most important point. There's a lot more to it… For more insight, check out the post titled: Disk space is cheap… (that's NOT the point!)

Even if you have the disk space to store all of these indexes, it's unlikely that you can fit all of your tables and all of your excessive indexes in cache. Most environments have a hard time fitting all of their data in general in cache – let alone a poorly/overindexed set of data. As a result, you're going to be constantly marshalling data in and out of memory and causing your system to perform excessive IOs.

You can cause the optimizer to waste time in evaluating indexes as well as cause compilation plans to be bigger. This can in turn waste both time and waste even more cache. Especially if you have a lot of plans being generated through adhoc statements. Each statement might waste quite a bit of cache. For more insight, check out the post titled: Clearing the cache – are there other options?

Why is this a common problem?

I'll go into more details in a moment but there are really three reasons:

SHOWPLAN shows missing index recommendations when evaluating a plan. Again, some folks take that to mean that they absolutely should create the index. Often, I've seen multiple developers working on a project all get their index recommendations in a vacuum (per se) and then they just provide scripts to the DBA for the indexes to implement. At this point, someone should evaluate the overall combination but often no one does. And, this leads to another reason why this is such a big problem.

SQL Server 2005 introduced the missing index DMVs and some people believe that all of the indexes it recommends should be created (which is far from the truth).

SQL Server (every version) will allow you to create redundant indexes. Not just similar (and redundant) but completely duplicate. There is a reason and I wrote about it in this SQL Server Magazine Q&A titled: Why SQL Server Lets You Create Redundant Indexes

The end result of this combination is that more and more servers I see – have too many indexes and often redundant or similar indexes. This wastes resources, time, cache and ultimately adds to a servers inability to scale as the data set grows. See, during development when the data sets were smaller, none of this really matters. Almost none is detected until more data and more users (and unfortunately, more code and more applications) exist. And, at that point it's too late to make schema changes BUT, it's not too late to make index changes. Phew. Believe me, many problems are MUCH harder to solve after an application is in production but indexes are one of the easier ones.

<https://www.sqlskills.com/blogs/kimberly/indexes-just-because-you-can-doesnt-mean-you-should/>

13. How is data stored if there is no clustered index? What about when you create a clustered index?

<https://learn.microsoft.com/en-us/sql/relational-databases/indexes/clustered-and-nonclustered-indexes-described?view=sql-server-ver16>

NON-CLUSTERED INDEX

• Since Non-Clustered Indexes do not physically move or store data, there can be many on a single table.

• Currently up to 999 different Indexes

• A Non-Clustered Index on a table with a Clustered Index must now grab data from the B-Tree of the CI.

• So data will come up through the Root of the CI and fall into the Leaf Pages of the NCI

Diagram

Description automatically generated

In a database table, data is stored in a specific physical order unless a clustered index is created on the table.

If there is no clustered index on a table, the data is stored in an unordered heap structure. This means that the rows of the table are stored in no particular order, and there is no logical connection between the physical location of the rows and the values in the rows.

If you create a clustered index on a table, the data is physically reordered based on the index key values. The rows of the table are stored in the order of the index key values, and the physical location of the rows is directly related to the values in the rows.

For example, if you create a clustered index on a table with a column called "LastName," the rows of the table will be physically reordered based on the values in the "LastName" column. This means that all rows with the same last name will be stored together, and rows with different last names will be stored in alphabetical order based on the last name.

Having a clustered index can improve the performance of queries that search for specific rows in the table, because the database system can use the index to quickly locate the rows you want. However, creating a clustered index can also have a negative impact on the performance of INSERT, UPDATE, and DELETE statements, as the index needs to be updated every time the data in the table is modified.

14. What is a B+Tree? What are roots and leaf nodes in a B+Tree? How many children can you have in a B+Tree?

A B+ tree consists of a root, internal nodes and leaves.[1] The root may be either a leaf or a node with two or more children.

A B+ tree can be viewed as a B-tree in which each node contains only keys (not key–value pairs), and to which an additional level is added at the bottom with linked leaves.

<https://en.wikipedia.org/wiki/B%2B_tree#:~:text=A%20B%2B%20tree%20consists%20of,with%20two%20or%20more%20children.&text=A%20B%2B%20tree%20can%20be,the%20bottom%20with%20linked%20leaves>.

The structure of the internal nodes of a B+ tree of order ‘a’ is as follows:

Each internal node is of the form: <P1, K1, P2, K2, ….., Pc-1, Kc-1, Pc> where c <= a and each Pi is a tree pointer (i.e points to another node of the tree) and, each Ki is a key-value (see diagram-I for reference).

Every internal node has : K1 < K2 < …. < Kc-1

For each search field values ‘X’ in the sub-tree pointed at by Pi, the following condition holds : Ki-1 < X <= Ki, for 1 < i < c and, Ki-1 < X, for i = c (See diagram I for reference)

Each internal node has at most ‘a’ tree pointers.

The root node has, at least two tree pointers, while the other internal nodes have at least \ceil(a/2) tree pointers each.

If an internal node has ‘c’ pointers, c <= a, then it has ‘c – 1’ key values.

Diagram

Description automatically generated

Each leaf node is of the form: <<K1, D1>, <K2, D2>, ….., <Kc-1, Dc-1>, Pnext> where c <= b and each Di is a data pointer (i.e points to actual record in the disk whose key value is Ki or to a disk file block containing that record) and, each Ki is a key value and, Pnext points to next leaf node in the B+ tree (see diagram II for reference).

Every leaf node has : K1 < K2 < …. < Kc-1, c <= b

Each leaf node has at least \ceil(b/2) values.

All leaf nodes are at the same level. Diagram

Description automatically generated

Advantages of B+Trees:

A B+ tree with ‘l’ levels can store more entries in its internal nodes compared to a B-tree having the same ‘l’ levels. This accentuates the significant improvement made to the search time for any given key. Having lesser levels and the presence of Pnext pointers imply that the B+ trees is very quick and efficient in accessing records from disks.

Data stored in a B+ tree can be accessed both sequentially and directly.

It takes an equal number of disk accesses to fetch records.

B+trees have redundant search keys, and storing search keys repeatedly is not possible.

Disadvantages of B+Trees:

The major drawback of B-tree is the difficulty of traversing the keys sequentially. The B+ tree retains the rapid random access property of the B-tree while also allowing rapid sequential access.

Application of B+ Trees:

Multilevel Indexing

Faster operations on the tree (insertion, deletion, search)

Database indexing

<https://www.geeksforgeeks.org/introduction-of-b-tree/>

<https://www.geeksforgeeks.org/insertion-in-a-b-tree/>

CLUSTERED INDEX

● Composed of 3 main levels ○ Root Level

 ○ Intermediate Level

○ Leaf Page Level

● Each Node is about 8KB in size ○ 8060B for data

○ 132B for pointers

○ 8192B in Total

Each Index created will have a Balance tree structure to be used, but the type of Index will determine how data is stored in a Balance Tree

Clustered Indexes will store data in Leaf Pages and sort them based on the Key values of the column you choose. Non-Clustered Indexes will NOT store data in the Leaf Pages, instead they’ll point to the rows they’re referencing

Diagram

Description automatically generated

● A clustered index will physically move the data from the table into it’s Balance Tree

● The data is now matching physically and logically

● Data is sorted based on ascending order for the column chosen, this becomes the clustering key

● This is why there can only be 1 Clustered

● Index on a table, data can only be physically sorted and stored once

Diagram

Description automatically generated

A B+Tree is a type of tree-based data structure that is commonly used to implement indexes in databases. It is a self-balancing tree structure that is designed to allow efficient insertions, deletions, and searches.

In a B+Tree, each node in the tree can have multiple children, and the tree is structured such that all leaf nodes are at the same level. This means that there are no data values stored in the interior (non-leaf) nodes of the tree, only in the leaf nodes.

The root node is the top node in the tree, and the leaf nodes are the bottom nodes in the tree. The root node may have one or more child nodes, and each child node may also have one or more child nodes, and so on. The number of children that a node can have depends on the degree of the B+Tree.

For example, in a B+Tree with a degree of 3, each node can have at most 3 child nodes. This means that the root node could have up to 3 child nodes, and each of those child nodes could have up to 3 child nodes, and so on.

The main advantage of a B+Tree is that it allows efficient search, insert, and delete operations, even for large data sets. It does this by keeping the tree balanced, which means that the height of the tree is kept as small as possible. This allows the database system to quickly locate the rows you want, even in large tables with millions of rows.

15. What are the differences between clustered and non-clustered indexes?

A table or view can contain the following types of indexes:

Clustered

Clustered indexes sort and store the data rows in the table or view based on their key values. These are the columns included in the index definition. There can be only one clustered index per table, because the data rows themselves can be stored in only one order.

The only time the data rows in a table are stored in sorted order is when the table contains a clustered index. When a table has a clustered index, the table is called a clustered table. If a table has no clustered index, its data rows are stored in an unordered structure called a heap.

Nonclustered

Nonclustered indexes have a structure separate from the data rows. A nonclustered index contains the nonclustered index key values and each key value entry has a pointer to the data row that contains the key value.

The pointer from an index row in a nonclustered index to a data row is called a row locator. The structure of the row locator depends on whether the data pages are stored in a heap or a clustered table. For a heap, a row locator is a pointer to the row. For a clustered table, the row locator is the clustered index key.

You can add nonkey columns to the leaf level of the nonclustered index to by-pass existing index key limits, and execute fully covered, indexed, queries. For more information, see Create indexes with included columns. For details about index key limits see Maximum capacity specifications for SQL Server.

Both clustered and nonclustered indexes can be unique. This means no two rows can have the same value for the index key. Otherwise, the index is not unique and multiple rows can share the same key value. For more information, see Create unique indexes.

Indexes are automatically maintained for a table or view whenever the table data is modified.

<https://learn.microsoft.com/en-us/sql/relational-databases/indexes/clustered-and-nonclustered-indexes-described?view=sql-server-ver16&viewFallbackFrom=sql-server-ver16What%20are%20the%20differences%20between%20clustered%20and%20non-clustered%20indexes%3F>

In a database, a clustered index is an index that reorders the rows in a table based on the index key values. There can only be one clustered index per table, and the data rows are stored in the physical order of the index key values.

A non-clustered index, on the other hand, is an index that creates a separate data structure that stores the index key values and a reference to the rows in the table. There can be multiple non-clustered indexes per table, and the data rows are not stored in the physical order of the index key values.

Here are some key differences between clustered and non-clustered indexes:

Storage: A clustered index stores the data rows in the physical order of the index key values, while a non-clustered index stores a separate data structure with the index key values and a reference to the rows in the table.

Number per table: A table can only have one clustered index, but it can have multiple non-clustered indexes.

Performance: A clustered index can improve the performance of queries that search for specific rows in the table, because the data is stored in the physical order of the index key values. Non-clustered indexes can also improve the performance of queries, but they may not be as efficient as clustered indexes.

Data modification: Creating or modifying a clustered index can be slower than creating or modifying a non-clustered index, because the data rows in the table need to be physically reordered.

Size: Clustered indexes tend to be larger than non-clustered indexes, because they store the data rows as well as the index key values. Non-clustered indexes are smaller, because they only store the index key values and a reference to the rows in the table.

When deciding which type of index to create, you should consider the size of the table, the workload of the database, and the types of queries that will be run against the table. Clustered indexes are generally more efficient for queries that search for specific rows in the table, while non-clustered indexes are generally more efficient for queries that return a large number of rows.

# Coding Questions:

All questions are based on AdventureWorks data warehouse unless otherwise specified. (We only have internet sales in our data warehouse)(note: I remove some tables in the data warehouse, so it only has internet sales in the database)

1. Find the total Internet sales tax amount.
2. Find the Internet sales amount per order by year and quarter of order date.
3. Find the percentage of sales amount for each order over total sales amount of that month by shipping date.
4. Display the full name, age, address(City, State, Country), and marital status of all customers who live in the US
5. Create a view with the product sold amount by subcategory by category by year and month order date.
6. Use the view created to display product sold amount by category by year of order date.
7. deleted
8. Group the customers into different age groups: < 30, 30-50, and 50+. Get the total number of customers and a total number of Internet sales in each group.
9. Create a stored procedure to input the first or last name of the customer, and return the Internet sale orders the customer(s) have created.
10. Create a stored procedure to display the total number of sales and sales amount for each month by category, with a specified year.
11. Create a function that will calculate the Hypotenuse of a triangle after you give the length of the two other sides of a triangle.

Use sub-queries to complete the following questions

13. Find the name of the customers who have made more than one purchase on the Internet.