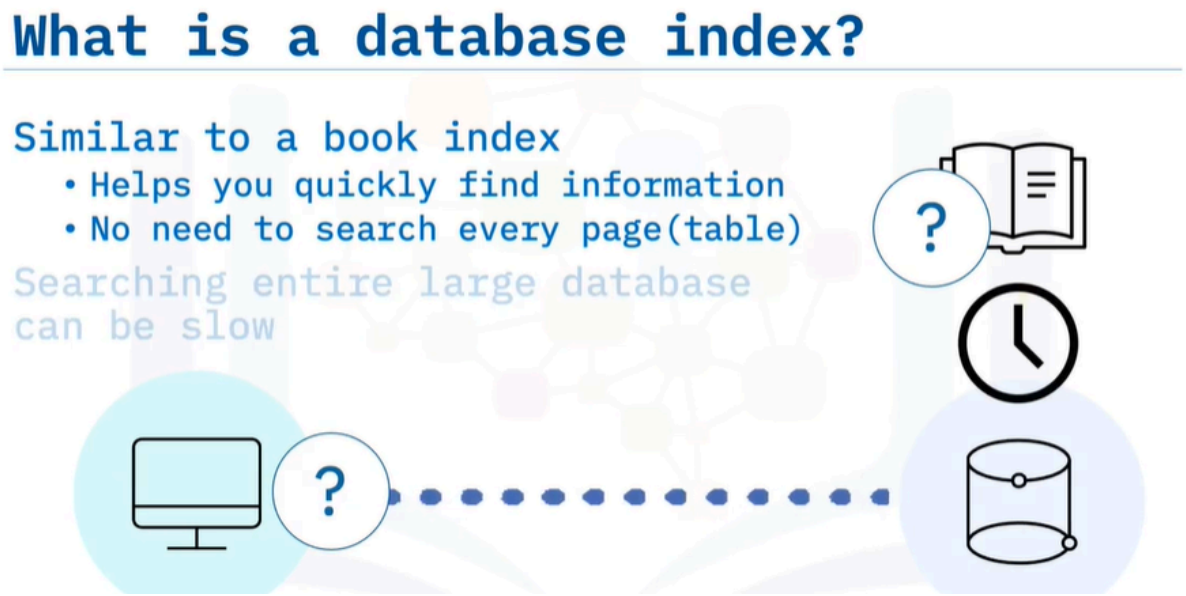
Using Indexes

Sure! Let's talk about database indexes in simple terms.

A database index is like the index at the back of a book. Imagine you have a huge book without an index; if you want to find a specific topic, you would have to flip through every single page, which would take a lot of time. Now, if the book has an index, you can quickly look up the topic and find the page number where it is located. Similarly, a database index helps the computer find information quickly without having to search through every single row in a database table. It organizes the data in a way that makes searching much faster and more efficient.

For example, if you have a database of customers and you often search for them by their customer ID, creating an index on the customer ID column will allow the database to find the information much faster. However, just like a book index takes up space, database indexes also require extra storage and maintenance. So, it's important to create them wisely to balance speed and storage needs.

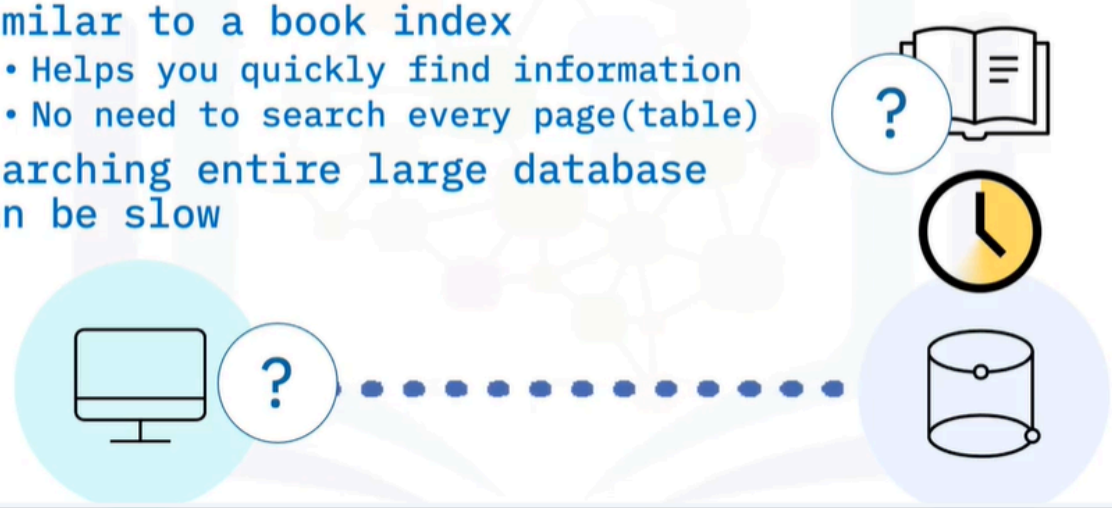


What is a database index?

Similar to a book index

- Helps you quickly find information
- No need to search every page(table)

Searching entire large database can be slow

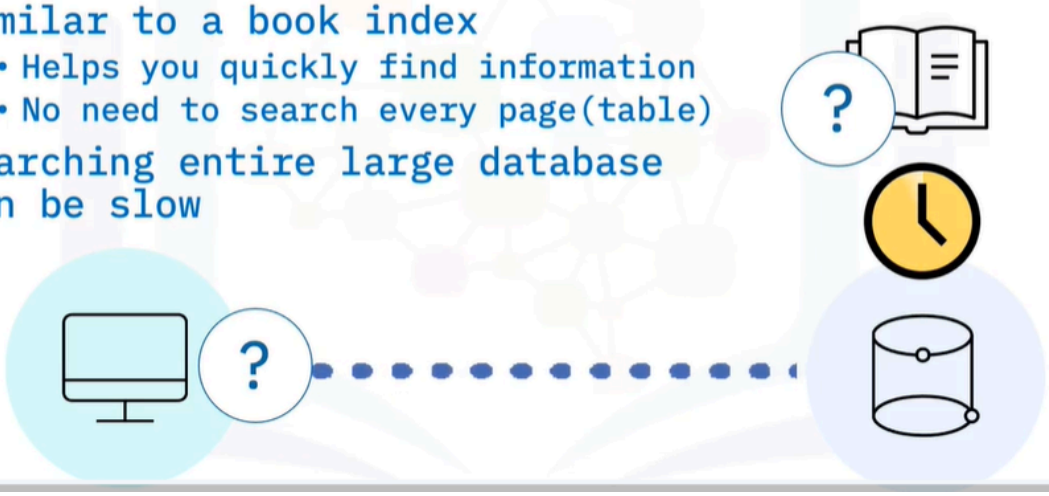


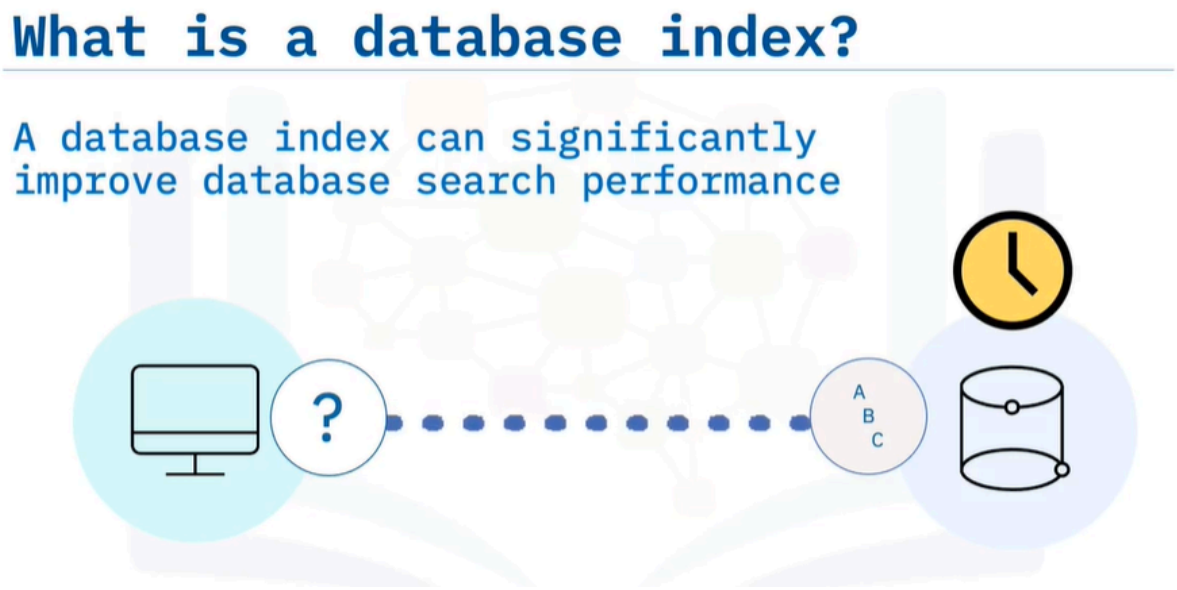
What is a database index?

Similar to a book index

- Helps you quickly find information
- No need to search every page(table)

Searching entire large database can be slow





- So, what is a database index? Much like the index in a book helps you find information in the book faster, an index for a database makes finding information in the database faster. For example, if you search a book without an index, you must search every page in the book to find the information you want. Similarly, if a computer searches a database without an index, it must search the entire database to find the information, which can be very slow—particularly in a very large database. A database index, however, can significantly improve search performance for a database.

What is a database index?

INDEX
11
11
11
12
15
18
18
20
20
20

ORDER_NO	CUST_ID	COST
33	11	45.02
36	15	26.11
39	12	66.26
44	20	15.47
48	11	92.01
49	18	103.50
53	11	89.13
56	20	46.55
61	18	29.17
63	20	40.22

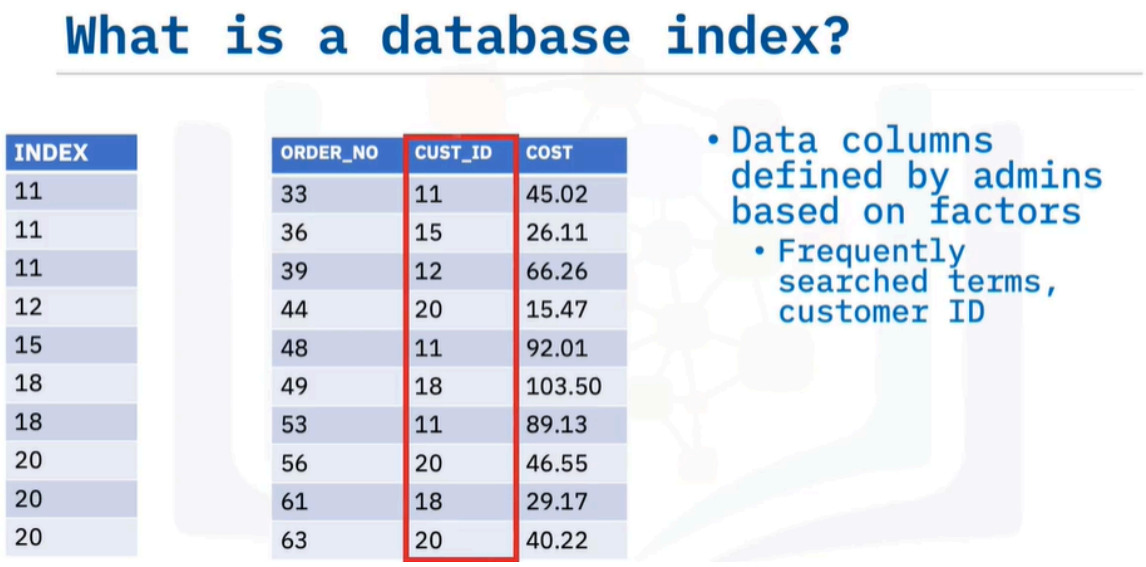
- Ordered copy of selected columns of data
- Enables efficient searches without searching every row

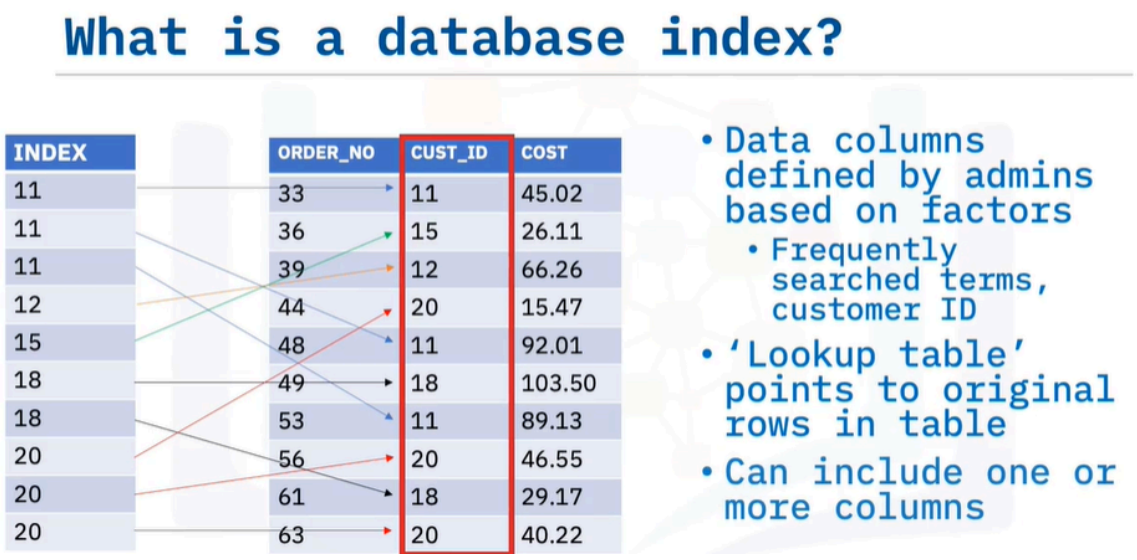
What is a database index?

INDEX
11
11
11
12
15
18
18
20
20
20

ORDER_NO	CUST_ID	COST
33	11	45.02
36	15	26.11
39	12	66.26
44	20	15.47
48	11	92.01
49	18	103.50
53	11	89.13
56	20	46.55
61	18	29.17
63	20	40.22

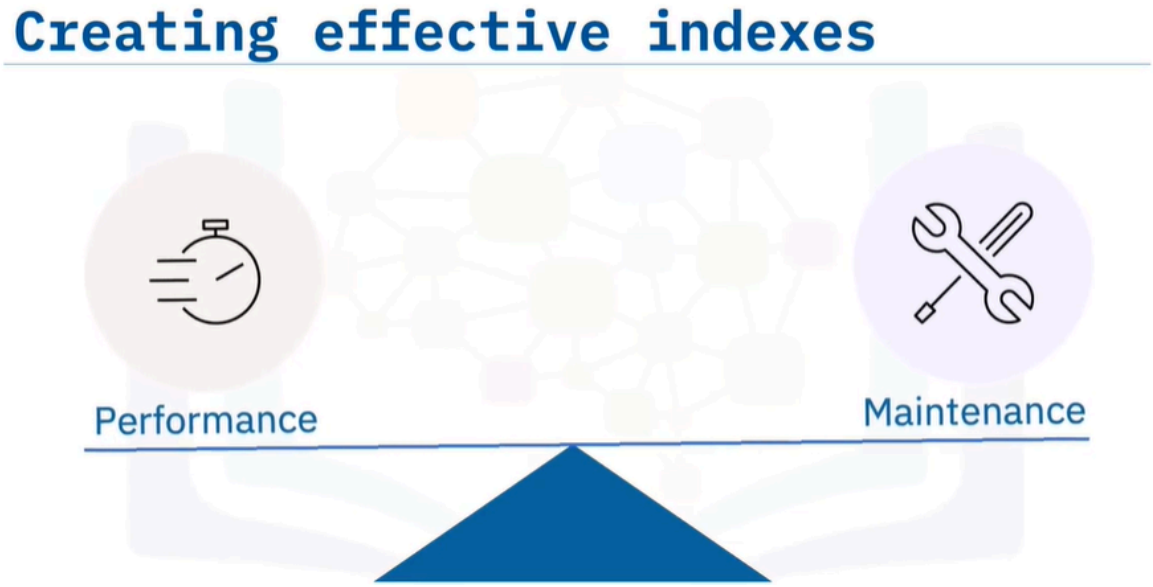
- Data columns defined by admins based on factors
  - Frequently searched terms, customer ID





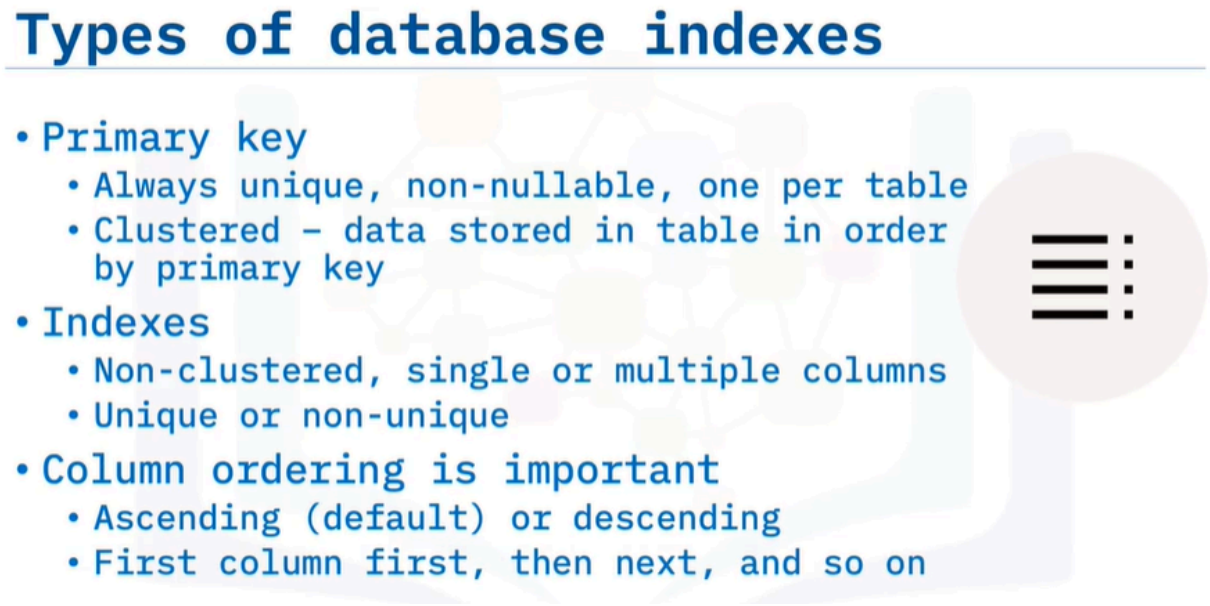
- In a database, the index is essentially an ordered copy of selected columns of table data and is designed to enable efficient searches without having to search every row in a database table every time it is accessed. The columns selected to make an index are typically defined by administrators and can be based on many factors, such as frequently searched terms. In this example, we are selecting the customer ID column. Regardless of how they are constructed, the overall goal of an index is to help users find the information they need quickly and easily. So, an index serves as a 'lookup table' with a

pointer or link to the original rows in the base table for which the index is created. An index can include one or more columns in the table.



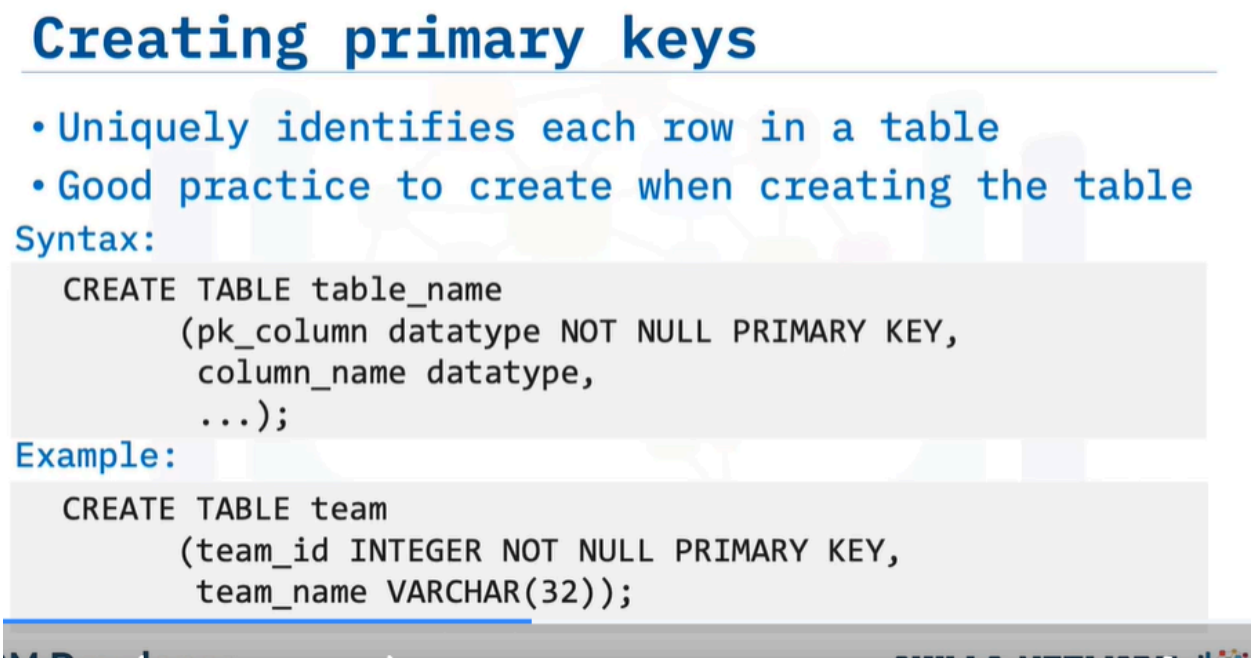
- However, while indexes can improve database performance, they also require additional storage space and maintenance because they must be continually updated.

- For any database, you will likely have to experiment with different types of indexes and indexing options. This will help you find an optimal balance of query performance and index maintenance—such as how much disk space and activity are required to keep the index up to date. For example, narrow indexes, or indexes with relatively few columns in the index key, take up less disk space and require less overhead to maintain, whereas wide indexes, while they may cover a greater number of queries, also require more space and maintenance.



- There are many different types and methods of indexing databases. The main types can generally be described as one of the following: The primary key is a special type of index that is always unique, non-nullable, and there can only be one per table. It is also clustered, meaning that the data is actually stored in the underlying table in the same order as the primary key. Each table can also have any number of additional or secondary indexes other than the primary key. These are non-clustered and can have single or multiple columns. Indexes can be defined as unique or non-unique. Ordering of columns is important because an index will be sorted in the order in which columns are specified when the index is created. Sorting can be ascending (which is the

default) or descending, but it will sort by the first column first, then by the next column, and so on.



- The primary key of a table uniquely identifies each row in a table. While having a primary key is optional in some RDBMSs, it is generally a good practice to create one when creating a table. It may also be possible to add a primary key later by altering the table. The first code block on this screen, shows the general syntax for creating a primary key when creating a table. The second code block shows a simple example: This example would create a table called team with the team_id column as the primary key.

Creating primary keys

Primary key with multiple columns

Syntax:

```
CREATE TABLE table_name
      (column_1_name datatype NOT NULL,
       column_2_name datatype NOT NULL,
       ...
       PRIMARY KEY(column_1_name, column_2_name));
```

Creating primary keys

Ensure uniqueness with auto-incrementing values

Db2: IDENTITY column

```
CREATE TABLE team
       (team_id INT GENERATED BY DEFAULT AS IDENTITY
                   PRIMARY KEY,
        team_name VARCHAR(32));
```

MySQL: AUTO_INCREMENT column

```
CREATE TABLE player
       ( player_id SMALLINT NOT NULL AUTO_INCREMENT,
         player name CHAR(30) NOT NULL,
         PRIMARY KEY (player_id) );
```

- If the primary key consists of multiple columns, you can even specify the primary key constraint after defining all the columns. You simply add the PRIMARY KEY option then specify the column names in parentheses.

Creating indexes

Syntax:

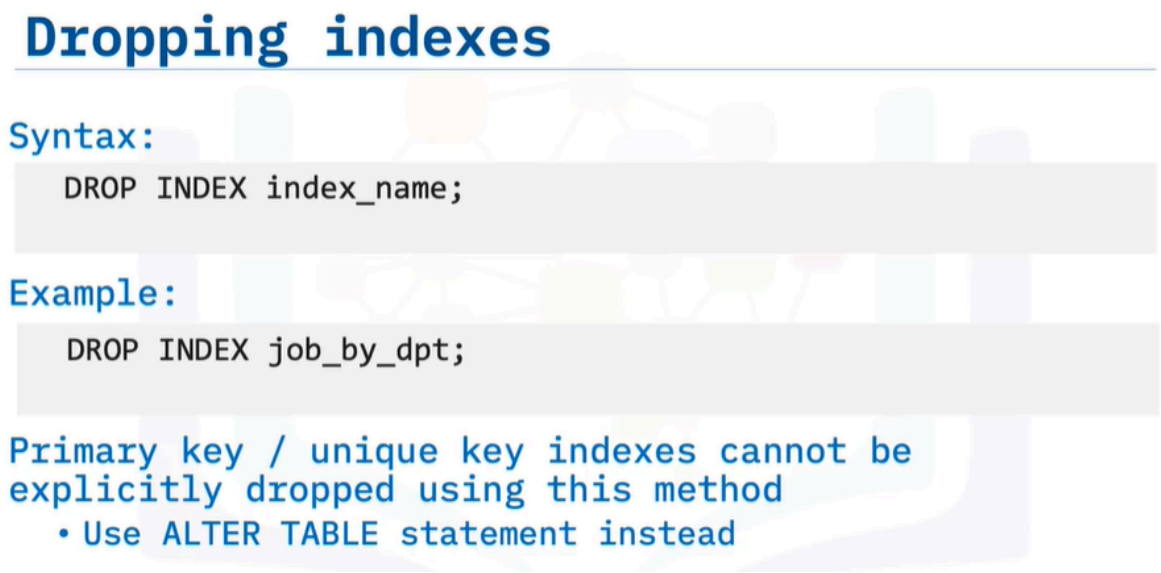
```
CREATE INDEX index_name
      ON table_name (column_1_name, column_2_name, ...);
```

Examples:

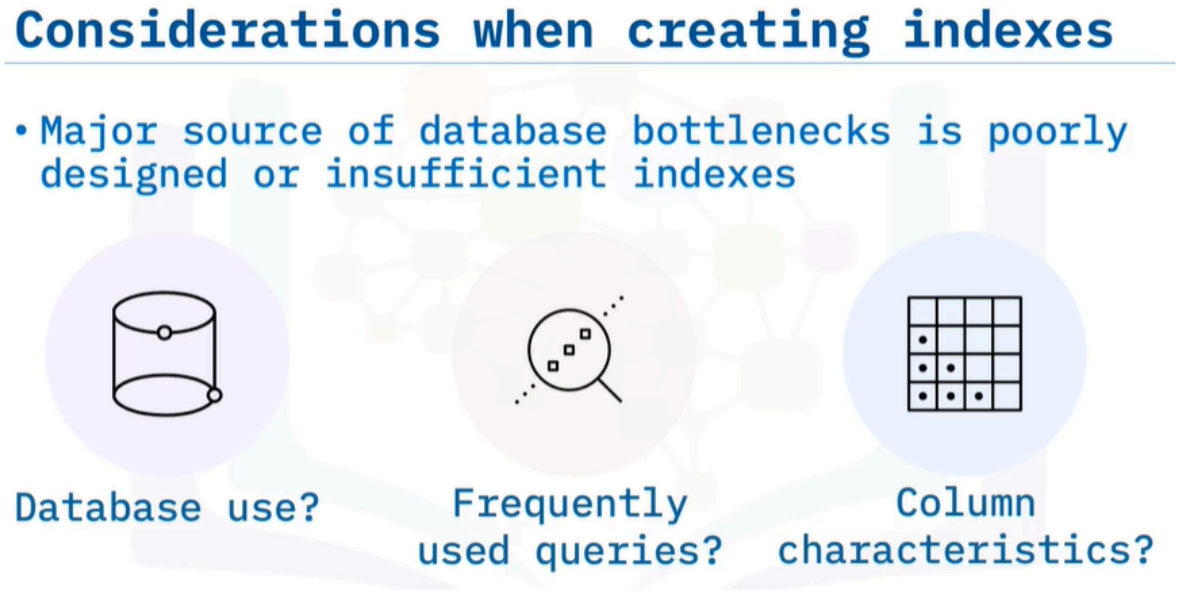
```
CREATE UNIQUE INDEX unique_nam
      ON project (projname);

CREATE INDEX job_by_dpt
      ON employee (workdept, job);
```

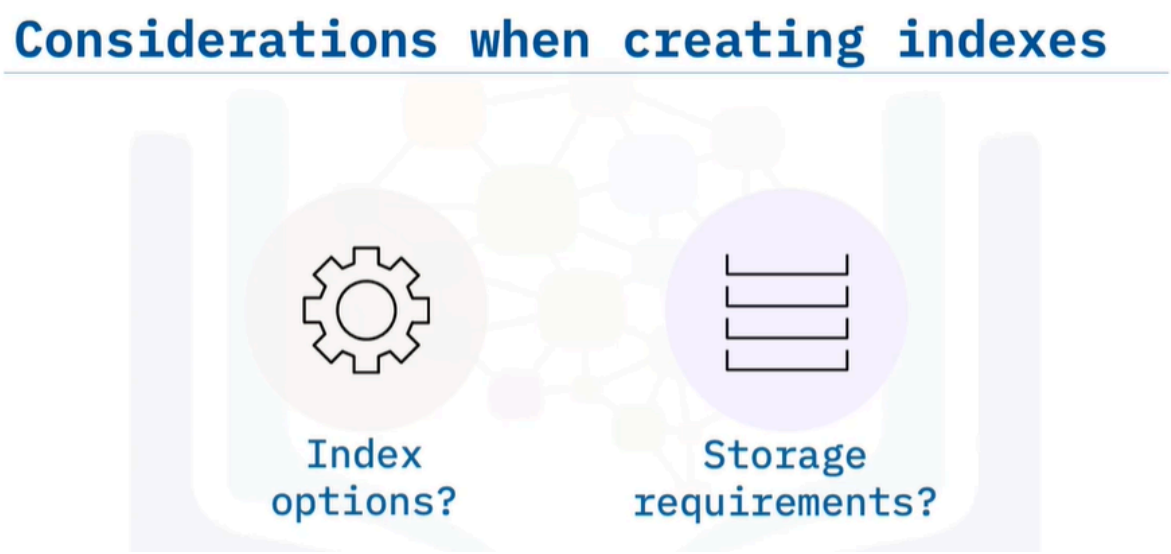
- The basic syntax for creating an index is shown in the first code block. You use the CREATE INDEX statement to define an index, and you use the ON statement to specify on which database table to create it. You specify the column names in parentheses, and the order in which you enter them dictates the index's sort order. And here are a couple of examples. The first one creates a unique index named unique_nam in the project table, with a column named projname, meaning that the project table cannot have multiple rows with the same projname value. When ON table_name is specified, the UNIQUE option prevents the table from containing two or more rows with the same value of the index key. The second example creates an index named job_by_dpt in the employee table, and it includes workdept and job columns in that order. This one is not unique, so the table can have multiple rows with the same workdept and job columns, but the benefit of the index is that query filtering on those fields will be faster than without an index.



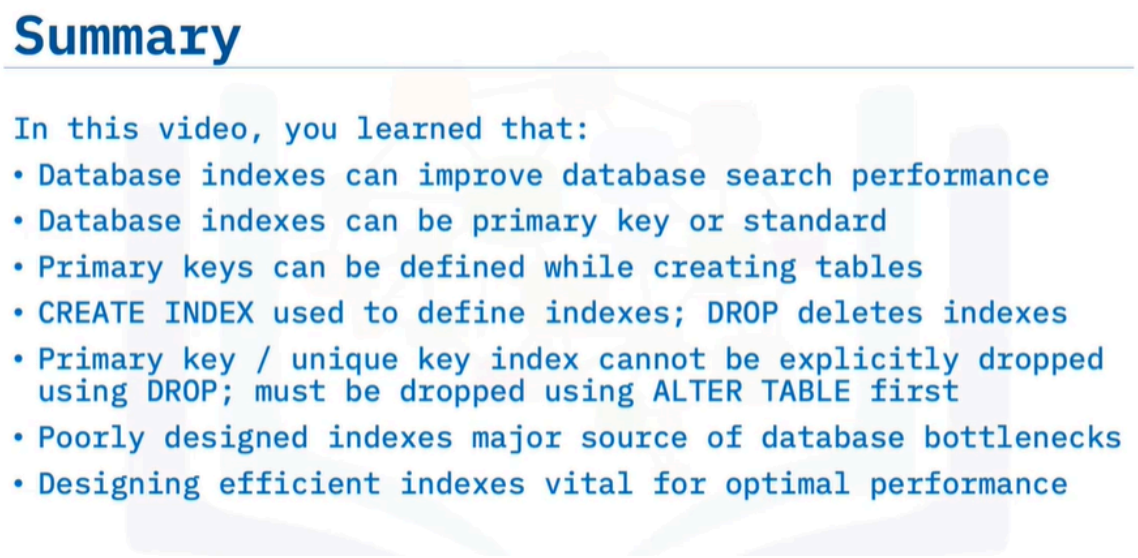
- To delete an index, you use the DROP statement. Dropping an index does not cause any other objects to be dropped but will cause packages that have a dependency on a dropped index or index specification to be invalidated. This example deletes the index named job_by_dpt. A primary key or unique key index cannot be explicitly dropped using the DROP statement. Deleting the primary index or a unique key index is done through the ALTER TABLE statement. After the primary key or unique key is deleted using ALTER TABLE, the index is no longer considered the primary index or unique index, and it can be explicitly dropped using the DROP statement.



- One of the major sources of database application bottlenecks can be poorly designed indexes, or a lack of sufficient, appropriate indexes. Therefore, designing efficient indexes is vital for optimizing database application performance. When designing an index, you should consider the following core principles: Understand how the database will be used. For example, will it be used for heavy online transaction processing or for the storage and retrieval of large data sets? Understand the most frequently used queries. For example, if you know that a frequently used query joins two or more tables, that may assist you in deciding the most appropriate index type to use. Understand the characteristics of the columns. For example, what kind of data will they contain? Will they contain integer values? And will they be unique? Knowing these characteristics from the outset will help you choose an appropriate index type.



- You will also need to consider: When creating or maintaining an index, which indexing options will best improve its performance? For example, if you are creating a clustered index on an existing large table, it would be a good idea to use the ONLINE index option, as it permits concurrent activity on the underlying data while the index is being built. And where is the best place to store the index? Where you decide to store your indexes can improve query performance by increasing the performance of disk input/output operations. For example, if you store a non-clustered index on a storage group located on a different disk than where the table storage group is located, it can improve performance as both disks could be read simultaneously.



How does indexing improve query performance?

Indexing improves query performance in several key ways:

- **Faster Searches**: With an index, the database can quickly locate the rows that match a query without scanning the entire table. This is similar to using a book index to find a topic instead of reading every page.
- **Reduced I/O Operations**: Indexes minimize the amount of data that needs to be read from disk. Since the index is smaller than the full table, fewer disk reads are required, which speeds up the query.
- **Efficient Sorting**: Indexes can help sort data more efficiently. When a query requires sorted results, the database can use the index to retrieve the data in the correct order without additional sorting operations.
- **Improved Join Performance**: When joining multiple tables, indexes on the join columns can significantly speed up the process by quickly locating matching rows in each table.

Overall, by providing a structured way to access data, indexes help reduce the time and resources needed to execute queries, leading to better performance.

What factors influence the choice of index type?

Several factors influence the choice of index type when designing a database. Here are the key considerations:

- **Query Patterns**: Understanding how the database will be queried is crucial. If certain columns are frequently used in search conditions or joins, those columns may benefit from indexing.
- **Data Characteristics**: The type of data in the columns (e.g., integers, strings) and their uniqueness can affect index choice. For example, unique values may benefit from a unique index.
- **Table Size**: Larger tables may require different indexing strategies compared to smaller tables. For large datasets, a clustered index might be more beneficial for performance.
- **Read vs. Write Operations**: If the database is read-heavy, certain index types can improve performance. Conversely, if there are many write operations, maintaining indexes can slow down performance, so a balance is needed.
- **Storage Considerations**: Indexes consume additional storage space. The choice of index type should consider the available storage and the impact on overall database performance.
- **Maintenance Overhead**: Some index types require more maintenance than others. Understanding the trade-offs between performance gains and maintenance costs is essential.

By considering these factors, you can choose the most appropriate index type to optimize database performance.

What would happen if you chose the wrong index type?

Choosing the wrong index type can lead to several negative consequences:

- **Poor Performance**: The primary issue is that queries may run slower. If the index does not match the query patterns, the database may still need to scan the entire table, negating the benefits of indexing.
- **Increased Storage Usage**: Some index types consume more storage space than necessary. This can lead to inefficient use of resources, especially if the index is not effectively supporting query performance.
- **Higher Maintenance Costs**: Maintaining an inappropriate index can increase the overhead during data modifications (inserts, updates, deletes). This can slow down these operations and lead to performance bottlenecks.
- **Complexity in Management**: Using the wrong index type can complicate database management. It may require additional effort to monitor and adjust indexes, leading to increased administrative workload.
- **Potential for Data Inconsistencies**: If an index is not properly maintained or is poorly designed, it can lead to inconsistencies in query results, especially in environments with frequent data changes.

Overall, selecting the right index type is crucial for optimizing database performance and ensuring efficient resource usage.

What factors should you consider when designing an index?

When designing an index, consider the following factors:

- **Query Patterns**: Analyze the most common queries and how they access data. Focus on columns frequently used in WHERE clauses, JOIN conditions, and ORDER BY clauses.

- **Data Characteristics**: Understand the nature of the data in the indexed columns. Consider factors like data type, uniqueness, and distribution of values (e.g., whether values are mostly unique or have many duplicates).
- **Table Size**: The size of the table can influence the choice of index type. Larger tables may benefit from clustered indexes, while smaller tables might not need complex indexing.
- **Read vs. Write Operations**: Determine whether the database will be read-heavy or write-heavy. If there are many write operations, consider the overhead of maintaining indexes and balance it with the need for read performance.
- **Index Type**: Choose the appropriate index type (e.g., clustered, non-clustered, unique) based on the specific use case and performance requirements.
- **Storage Considerations**: Evaluate the storage implications of the index. Indexes consume additional disk space, so consider how much space is available and the impact on overall database performance.
- **Maintenance Overhead**: Consider the maintenance required for the index. Some indexes require more frequent updates, which can affect performance during data modifications.

By carefully considering these factors, you can design effective indexes that optimize database performance and resource usage.

What is the importance of query patterns in index design?

Query patterns are crucial in index design for several reasons:

- **Optimizing Performance**: Understanding how queries access data helps in creating indexes that directly improve the speed of those queries. By focusing on the columns frequently used in WHERE clauses, JOINs, and ORDER BY clauses, you can significantly enhance query performance.

- **Reducing I/O Operations**: By analyzing query patterns, you can design indexes that minimize the amount of data the database needs to read from disk. This reduces input/output operations, leading to faster query execution.
- **Identifying Key Columns**: Query patterns help identify which columns are most important for indexing. This ensures that the indexes created are relevant and beneficial for the most common queries.
- **Balancing Read and Write Operations**: Understanding query patterns allows you to balance the need for fast read operations with the overhead of maintaining indexes during write operations. This is especially important in databases with high transaction volumes.
- **Improving Query Plans**: The database optimizer uses indexes to determine the best execution plan for a query. Well-designed indexes based on query patterns can lead to more efficient query plans, further enhancing performance.

In summary, query patterns guide the index design process, ensuring that the indexes created are effective in improving query performance and overall database efficiency.