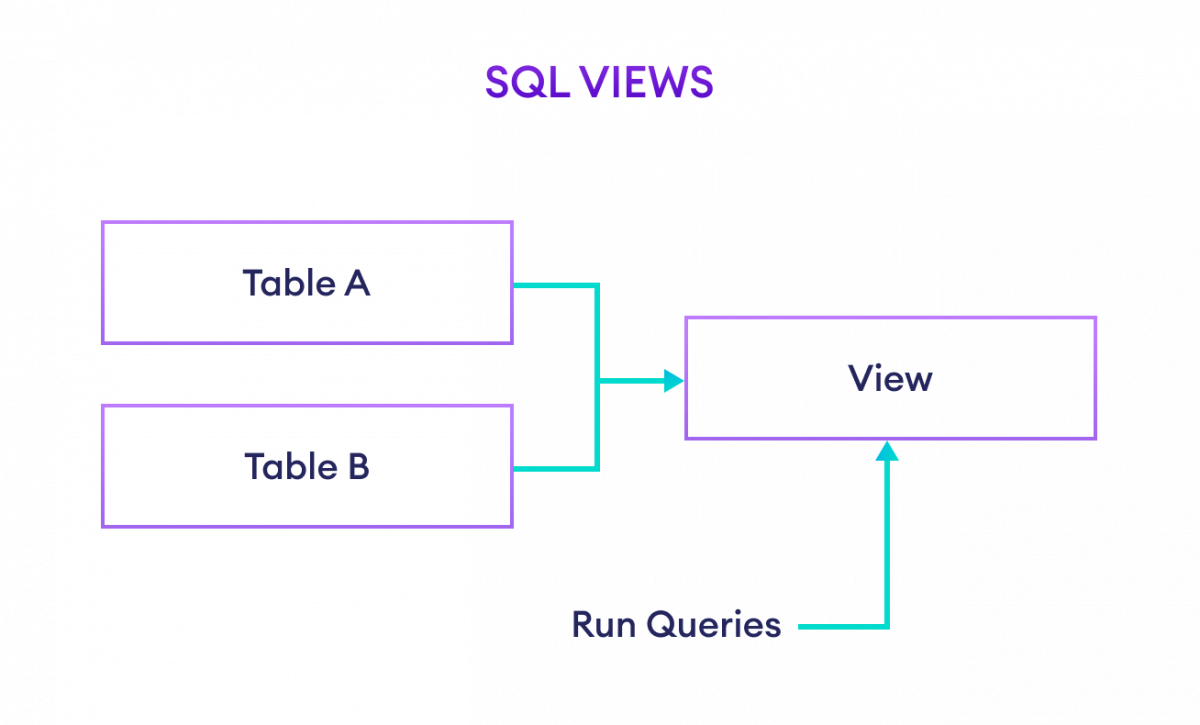
**SQL Views**

In SQL, views contain rows and columns similar to a table, however, views don't hold the actual data.

You can think of a view as a virtual table environment that's created from one or more tables so that it's easier to work with data.

Example: SQL Views

**Creating a View in SQL**

We can create views in SQL by using the CREATE VIEW command. For example,

CREATE VIEW us\_customers AS

SELECT customer\_id, first\_name FROM Customers WHERE Country = 'USA';

[Run Code](https://www.programiz.com/sql/online-compiler)

Here, a view named us\_customers is created from the customers table.

Now to select the customers who lives in **USA**, we can simply run,

SELECT \* FROM us\_customers;

[Run Code](https://www.programiz.com/sql/online-compiler)

**Updating a View**

It's possible to change or update an existing **view** using the CREATE OR REPLACE VIEW command. For example,

CREATE OR REPLACE VIEW us\_customers AS

SELECT \*

FROM Customers

WHERE Country = 'USA';

[Run Code](https://www.programiz.com/sql/online-compiler)

Here, the us\_customers view is updated to show all the fields.

**Deleting a View**

We can delete views using the DROP VIEW command. For example,

DROP VIEW us\_customers;

[Run Code](https://www.programiz.com/sql/online-compiler)

Here, the SQL command deletes the view named us\_customers.

**Note:** If the view is not available, the above command throws an error.

**Views for Complex Queries**

Suppose **A** and **B** are two tables and we wan't to select data from both of the tables. For that, we have to use [SQL JOINS](https://www.programiz.com/sql/join).

However using the JOIN each time could be a tedious task. For that, we can create a view to fetch records easily.

Let's create a view,

CREATE VIEW order\_details AS

SELECT Customers.customer\_id, Customers.first\_name, Orders.amount

FROM Customers

JOIN Orders

ON Customers.customer\_id = Orders.customer\_id;

[Run Code](https://www.programiz.com/sql/online-compiler)

Now, to select the data, we can run

SELECT \*

FROM order\_details;

[Run Code](https://www.programiz.com/sql/online-compiler)

Here, the SQL command selects data from the view order\_details.

**SQL Indexes**

An index in SQL is a schema objec**t** that improves the speed of data retrieval operations on a table. Imagine them like an index in a book instead of flipping through every page (row), the database can jump right to the data it requires.

It works by creating a separate data structure that provides pointers to the rows in a table. Which makes it faster to look up rows based on specific column values. Indexes act as a table of contents for a database, allowing the server to locate data quickly and efficiently, reducing disk I/O operations.

**Benefits of Indexes:**

* **Faster Queries**: Speeds up SELECT and JOIN operations.
* **Lower Disk I/O**: Reduces the load on your database by limiting the amount of data scanned.
* **Better Performance on Large Tables**: Essential when working with millions of records.

**Creating an Index**

Creating an index allows us to define a quick access path to data. SQL indexes can be applied to one or more columns and can be either unique or non-unique.

Unique indexes ensure that no duplicate values are entered in the indexed columns, while non-unique indexes simply speed up queries without enforcing uniqueness. You can create:

* **Single-column indexes**: For basic queries
* **Multi-column indexes**: For queries using multiple filters
* **Unique indexes**: To ensure data uniqueness

**Syntax:**

*CREATE INDEX index  
ON TABLE column;*

**Example**

*CREATE INDEX idx\_product\_id  
ON Sales (product\_id);*

**Explanation:** This creates an index named idx\_product\_id on the product\_id column in the Sales table, improving the speed of queries that filter or join based on this column.

**Why SQL Indexing is Important?**

Indexing in SQL is a critical feature for optimizing query performance, especially for large datasets. Here are some common scenarios where indexing proves beneficial:

1. **Large Data Tables**: SQL queries on tables with millions of rows can significantly slow down due to full table scans. Indexes provide a faster alternative by allowing quick access to relevant rows.
2. **Join Optimization**: Indexes on columns used for joining tables (such as [foreign keys](https://www.geeksforgeeks.org/sql/foreign-key-constraint-in-sql/)) improve the performance of complex joins.
3. **Search Operations**: Queries that search for specific values in a column can be sped up with indexes, reducing the time required to perform lookups.

***Note:*** *It is essential to be mindful of the storage cost and performance tradeoffs associated with indexes. Over-indexing can lead to unnecessary overhead, while under-indexing may slow down data retrieval*

**Multi - Column Indexes**

A multi-column index is an index that involves more than one column in a table. It helps improve query performance when multiple columns are frequently used together in conditions like WHERE, JOIN, or ORDER BY clauses. Instead of creating separate indexes for each column, a multi-column index can increase the speed of search and retrieval process by allowing the database to access data more efficiently.

**Syntax:**

*CREATE INDEX index  
ON TABLE (column1, column2,.....);*

**Example**

*CREATE INDEX idx\_product\_quantity  
ON Sales (product\_id, quantity);*

**Explanation:** This index allows the database to quickly filter or join data based on both product\_id and quantity columns.

**Unique Indexes**

A [unique index](https://www.geeksforgeeks.org/sql/sql-unique-index/) ensures that all values in the indexed column(s) are unique, preventing duplicates. These are useful for maintaining the integrity of the data, ensuring that no two rows have the same values in the indexed columns.

**Syntax:**

*CREATE UNIQUE INDEX index\_name  
ON table\_name (column\_name);*

**Example**

*CREATE UNIQUE INDEX idx\_unique\_employee\_id  
ON Employees (employee\_id);*

**Explanation:** This index ensures that no two rows in the **Employees** table have the same employee\_id, which maintains data integrity and prevents duplicate entries.

**Removing an Index**

If an index is no longer needed, it can be removed to improve write performance or save storage space. As indexes can slow down operations like INSERT, UPDATE, and DELETE due to the overhead of maintaining them, dropping unnecessary indexes can improve overall database efficiency. The [DROP INDEX](https://www.geeksforgeeks.org/sql/sql-drop-index/)command is used for this purpose.

**Syntax**

*DROP INDEX index;*

**Explanation:** This command removes an index from the database schema. It does not affect the underlying data in the table but may slow down future queries that would have benefited from the index.

**Altering an Index**

If an index requires adjustments, such as reorganizing or rebuilding, it can be altered without affecting the data. This is useful for optimizing index performance as tables grow larger.

**Syntax:**

*ALTER INDEX IndexName   
ON TableName REBUILD;*

**Explanation:** This command rebuilds the specified index, which can optimize query performance by reorganizing its structure, especially in large tables.

**Confirming and Viewing Indexes**

We can view all the indexes in a database to understand which ones are in use and confirm their structure. In SQL, the following query helps us see the indexes for a given table:

**Syntax:**

*SELECT \* from USER\_INDEXES;*

**Explanation:** This query retrieves all the indexes in the [database schema](https://www.geeksforgeeks.org/dbms/database-schemas/), showing their names and the columns they are associated with. We can use this information to audit or manage existing indexes.

**Renaming an Index**

In some cases, renaming an index might be necessary for clarity or consistency. While SQL does not directly support renaming indexes, we can use a combination of commands to achieve this.

**Syntax:**

*EXEC sp\_rename 'old\_index\_name', 'new\_index\_name', 'INDEX';*

**Explanation:** This command allows us to rename an existing index, which helps maintain clarity in our database schema.

**When Should Indexes Be Created?**

Indexes can significantly improve query performance, but they should be used judiciously. The following scenarios warrant creating indexes:

1. **Wide Range of Values**: Indexes are helpful when a column has a wide range of values, such as product IDs or customer names, as they speed up search operations.
2. **Non-NULL Values**: Columns that don’t contain many NULL values are ideal candidates for indexing, as NULLs complicate the indexing process.
3. **Frequent Query Conditions**: Indexes should be created on columns frequently used in [WHERE clauses](https://www.geeksforgeeks.org/sql/sql-where-clause/) or as part of a join condition.

**When Should Indexes Be Avoided?**

While indexes enhance performance, they may not always be beneficial, especially in certain situations:

1. **Small Tables**: Indexes are not needed for small tables as queries will likely perform well without them.
2. **Infrequent Query Use**: If a column is rarely used in queries, indexing it will only add overhead.
3. **Frequently Updated Columns**: Avoid indexing columns that are frequently updated, as the index will need to be updated with each change, adding overhead.