# Order by

Hello Everyone!

Ever wondered how databases can not only store vast amounts of data but also present it in exactly the order you need for analysis or presentation? Let's explore the dynamic capabilities of the 'ORDER BY' clause and uncover how it brings clarity and structure to your data queries!

ORDER BY is a SQL clause used to sort the result set of a query based on specified columns in ascending or descending order.

Syntax:

A black background with white text

Description automatically generated

Explanation:

SELECT:

Specifies the columns to retrieve from the table.

FROM:

Specifies the table from which to retrieve data.

ORDER BY:

Specifies the columns by which to sort the result set. Columns can be sorted in ascending or descending order. By default, ORDER BY sorts the data in ascending order.

Example:

Suppose we have a `students` table with columns `student\_id`, `name`, and `score`. To retrieve student names and their scores sorted by score in descending order.

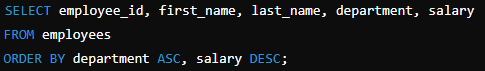
A black background with white text

Description automatically generated

Result:

Example of order by on multiple columns:

Suppose we want to order the employees first by department in ascending order and then by salary in descending order within each department.



Result:

The `ORDER BY` clause in SQL allows you to sort the result set of a query based on specified columns, either in ascending or descending order. Understanding how to use `ORDER BY` is crucial for controlling the presentation and analysis of data retrieved from databases, ensuring that results are displayed in a meaningful and structured way. We can use the keyword DESC to sort the data in descending order and the keyword ASC to sort in ascending order.

I trust you found this video helpful. Thank you!

# Case WHEN

Hello Everyone!

Ever wondered how databases can dynamically categorize and analyze data, making decisions on the fly? Let's dive into the versatile world of 'CASE WHEN' statements and uncover how they empower SQL queries to handle complex conditions and customize data retrieval like never before!

`CASE WHEN` is a SQL expression used for conditional logic, allowing data to be transformed or categorized based on specified conditions.

Syntax:

A black screen with white text

Description automatically generated

Explanation:

CASE: Begins the `CASE WHEN` expression.

WHEN condition: Specifies the condition to be evaluated.

THEN result: Specifies the result if the condition is true.

ELSE result\_default: Specifies the result if none of the preceding conditions are true (optional).

END: Ends the `CASE WHEN` expression.

Example:

Suppose we have a `students` table with columns `student\_id`, `name`, and `score`. We want to categorize students based on their scores into "Excellent", "Good", "Average", and "Needs Improvement":

Input:

Query:

A computer screen with text

Description automatically generated

Result:

A screenshot of a survey

Description automatically generated

The `CASE WHEN` expression in SQL allows you to evaluate conditions and return a value based on those conditions. It's useful for categorizing data, performing calculations, or transforming data based on specific criteria within your queries. Understanding `CASE WHEN` enhances your ability to manipulate and analyze data effectively in database applications.

I trust you found this video helpful. Thank you!

# Union vs union all

Hello Everyone!

Curious about how to combine results from multiple queries into a single, cohesive dataset? Let's dive into the differences between SQL's UNION and UNION ALL, and discover when to use each to maximize your data analysis potential!

`UNION` is an operator that combines the results of two or more SELECT queries, removing duplicate rows from the final result set.

`UNION ALL` is a SQL operator that combines the results of two or more SELECT queries, including all duplicate rows in the final result set.

UNION

Usage: Use `UNION` when you want to merge results from multiple queries but eliminate any duplicates.

Syntax:

A black screen with white text

Description automatically generated

UNION ALL

Usage: Use `UNION ALL` when you want to merge results from multiple queries and retain all duplicates.

Syntax:

A screen shot of a computer

Description automatically generated

UNION

Example:

Suppose we have two tables, `employees\_2023` and `employees\_2024`:

employees\_2023:

A white rectangular object with a black stripe

Description automatically generated with medium confidence

employees\_2024:

A white rectangular object with a black border

Description automatically generated with medium confidence

Query:

A screen shot of a computer

Description automatically generated

Result:

A white rectangular object with a black text

Description automatically generated with medium confidence

UNION ALL

Example:

A screenshot of a computer program

Description automatically generated

Result:

A screenshot of a phone

Description automatically generated

Understanding the difference between `UNION` and `UNION ALL` helps in choosing the right operation for your data merging needs, whether you require a unique list of results or need to retain all duplicate records.

I trust you found this video helpful. Thank you!

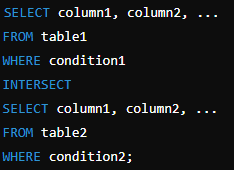
# Intersect

Hello Everyone!

Ever wondered how to find the sweet spot where two datasets overlap perfectly? Imagine being able to pull out just the matching records with a single, powerful SQL command. Curious? Let's dive into the magic of the `INTERSECT` operator!

The `INTERSECT` operator in SQL returns the common rows that exist in the result sets of two or more SELECT queries.

Syntax:



This query returns the rows that are common to both SELECT statements.

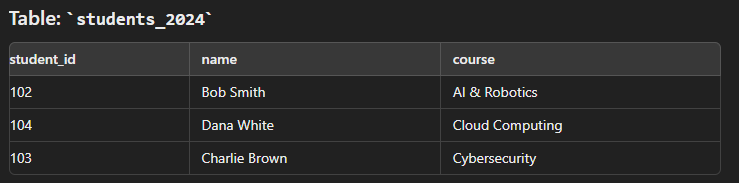
Example:

Let's consider two example tables: students\_2023 and students\_2024. Each table lists the students enrolled in each respective year.

students\_2023:

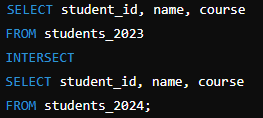


students\_2024:

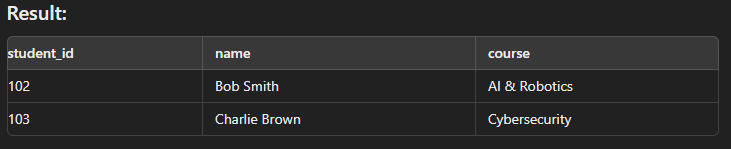


Query:

To find the students who were enrolled in both 2023 and 2024, you can use the INTERSECT operator as follows:



Result:



This query returns the students who are common to both years, along with the courses they were enrolled in.

In summary, the `INTERSECT` operator in SQL is used to return only the rows that are common between two or more `SELECT` queries. It works by comparing the result sets of the queries and retaining only the rows that appear in all of them. This is useful for finding overlapping data between different tables or datasets, ensuring that only the matching records are included in the final result. Each `SELECT` query must have the same number of columns with matching data types.

I trust you found this video helpful. Thank you!

# Join

Hello Everyone!

Ever wondered how you can seamlessly combine data from multiple tables to uncover deeper insights and relationships? Let's dive into the fascinating world of SQL Joins and discover how they bring scattered pieces of information together for powerful data analysis!

A Join is a query that combines rows from two or more tables based on a related column between them.

Here are the main types of joins in SQL:

INNER JOIN

LEFT JOIN or LEFT OUTER JOIN

RIGHT JOIN or RIGHT OUTER JOIN

FULL JOIN or FULL OUTER JOIN

SELF JOIN

INNER JOIN

Definition:

- Returns only the rows that have matching values in both tables.

Syntax:

A black background with white text

Description automatically generated

The default join in SQL is the INNER JOIN. When you use the `JOIN` keyword without specifying the type, it performs an inner join, which returns only the rows with matching values in both tables.

A black screen with white text

Description automatically generated

Example:

Suppose we have two tables, `students` and `courses`:

students:

A screenshot of a computer

Description automatically generated

courses:

A screenshot of a phone number

Description automatically generated

Query:

A black screen with white text

Description automatically generated

Result:

A screenshot of a computer

Description automatically generated

LEFT JOIN or LEFT OUTER JOIN

Definition:

- Returns all rows from the left table and the matched rows from the right table. If no match is found, NULL values are returned for columns from the right table.

Syntax:

A black screen with white text

Description automatically generated

Example:

A black screen with white text

Description automatically generated

Result:

A screenshot of a computer

Description automatically generated

RIGHT JOIN or RIGHT OUTER JOIN

Definition:

- Returns all rows from the right table and the matched rows from the left table. If no match is found, NULL values are returned for columns from the left table.

Syntax:

A black screen with white text

Description automatically generated

Example:

A black background with white text

Description automatically generated

Result:

A screenshot of a computer

Description automatically generated

FULL JOIN or FULL OUTER JOIN

Definition:

- Returns all rows when there is a match in either left or right table. If no match is found, NULL values are returned for columns from the table without a match.

Syntax:

A black screen with white text

Description automatically generated

Example:

A black screen with white text

Description automatically generated

Result:

A screenshot of a computer

Description automatically generated

Self Join

Definition: A self join is a join where a table is joined with itself. This is useful for comparing rows within the same table.

Syntax:

A black background with white text

Description automatically generated

Example:

Suppose we have a single table, `employees`, which includes information about each employee and their manager:

employees:

A screenshot of a computer

Description automatically generated

Query:

A black screen with white text

Description automatically generated

Result:

A screenshot of a computer

Description automatically generated

Understanding these different types of joins helps in combining data from multiple tables effectively, allowing for comprehensive data analysis and reporting.

I trust you found this video helpful. Thank you!

# LIKE operator

Hello Everyone!

Have you ever needed to search for patterns within your data, like finding all email addresses from a specific domain or names that start with a certain letter? Let's explore the power of the SQL LIKE operator and how it makes pattern matching in databases a breeze!

The `LIKE` operator in SQL is used to filter results based on a specified pattern, allowing for partial matches within a column.

Syntax:

Two wildcards are commonly used with the LIKE operator:

% (Percent Sign): This wildcard represents zero, one, or multiple characters. It’s used when you want to match any sequence of characters.

\_ (Underscore): This wildcard represents a single character. It's handy when you need to match a specific character position.

Examples

Let's consider an Employees table, where we have columns Employee ID, Name and Email.

Using the % Wildcard

Example 1: Finding names that start with 'A':

To search for employees whose names start with the letter 'A', you can use the % wildcard after 'A':

Result:

Example 2: Finding email addresses that end with '@domain.com':

To find all employees with email addresses that end in @domain.com, you would use the % wildcard before the domain:

Result:

Using the \_ Wildcard

Example: Finding names that start with 'A' and are five characters long:

To find names that start with 'A' and are exactly five characters long, you can use the \_ wildcard for each character:

Result:

Combining Wildcards

You can also combine % and \_ to create more complex patterns.

Example: Finding names with 'a' as the second character:

To find names where 'a' is the second character, use an underscore for the first character and % for any characters following 'a':

Result:

Understanding how to use the `LIKE` operator and its wildcards enables you to perform powerful and flexible searches within your database.

I trust you found this video helpful. Thank you!

# BETWEEN and IN operators

Hello Everyone!

Ever wondered how to efficiently filter data within a range or from a specific list of values in SQL? Let's dive into the differences between the powerful BETWEEN and IN operators and see how they can streamline your queries!

The `BETWEEN` operator in SQL is used to filter the result set to include only values that fall within a specified range.

The `IN` operator in SQL is used to filter the result set to include only values that match any value in a specified list.

BETWEEN Operator

Use BETWEEN in SQL to filter records within a specific range of values.

Syntax:

A black background with white text

Description automatically generated

IN Operator

Use IN to filter records matching any value within a specified list.

Syntax:

A black background with white text

Description automatically generated

Example:

Suppose we have a table `students` with `student\_id`, name and `age` columns:

students:

A screenshot of a computer

Description automatically generated

BETWEEN

Query:

A black background with white text

Description automatically generated

Result:

A screenshot of a phone

Description automatically generated

IN Operator

Query:

A black background with white text

Description automatically generated

Result:

A screenshot of a phone

Description automatically generated

Difference

A screenshot of a computer

Description automatically generated

Understanding these differences helps in effectively using SQL operators to filter and query data based on different criteria.

I trust you found this video helpful. Thank you!

# Delete vs truncate vs drop

Hello Everyone!

Ever wondered how databases handle data removal and table management differently? Let's explore the distinct roles of DELETE, TRUNCATE, and DROP in SQL operations, and uncover when and why each method is used to maintain and manipulate your database!

`DELETE` is a SQL statement used to remove one or more rows from a table based on specified conditions.

`TRUNCATE` is a SQL statement used to quickly remove all rows from a table, effectively resetting the table without logging individual row deletions.

`DROP` is a SQL statement used to remove an entire table, including its structure and all data it contains, from the database.

Syntax:

A black background with white text

Description automatically generated





Example:

Suppose we have a table `employees`:

employees:

A screenshot of a phone

Description automatically generated

Delete Query:

A black background with white text

Description automatically generated

Result:

A white rectangular object with a black border

Description automatically generated with medium confidence

TRUNCATE

Example:

Using the same `employees` table:

TRUNCATE Query:



Result:

DROP

Example:

Using the same `employees` table:

DROP Query:



Result:



Differences

A screenshot of a computer

Description automatically generated

Summary:

DELETE removes specific rows based on conditions, can be rolled back, and triggers logging.

TRUNCATE removes all rows from a table quickly, resets table size and indexes, and does not log deletions.

DROP removes an entire table, including its structure, indexes, constraints, and all data it contains, immediately and without logging.

Understanding these differences helps in choosing the appropriate operation for manipulating data or managing database structure in SQL.

I trust you found this video helpful. Thank you!

# View

Hello Everyone!

"Ever wondered how you can create a virtual table in SQL that simplifies complex queries and enhances security? Let’s dive into the magic of SQL Views!"

"A SQL View is a virtual table that presents data from one or more tables through a stored query, allowing you to simplify complex queries and enhance data security."

We can create a view by selecting fields from one or more tables present in the database.

A View can either have all the rows of a table or specific rows based on certain conditions.

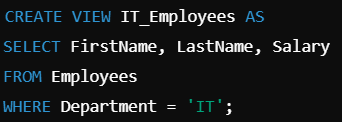
Views are saved SQL queries that act like tables, providing a way to look at data in a specific format or structure.

Unlike actual tables, views don’t store data themselves but pull it from underlying tables whenever accessed.

Let's consider a sample input table named `Employees`:

Creating a Simple View

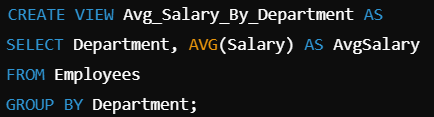
Suppose you want to create a view that shows only IT department employees' names and their salaries.



Querying the View:



Creating a View with Aggregation



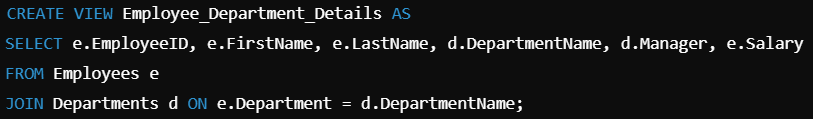
Result of Querying the View:



Creating a Complex View with Joins (if you have another table)

Let’s assume there’s another table `Departments` with department details:

SQL Query to Create a View that Joins the `Employees` and `Departments` Tables:



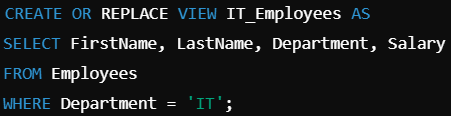
Result of Querying the View:



Modifying an Existing View

Suppose you want to update the `IT\_Employees` view to include the `Department` column as well.

SQL Query to Modify the View:



Result of Querying the Updated View:



Dropping a View

If you no longer need a view, you can remove it from the database.



This will permanently delete the `IT\_Employees` view from the database.

Benefits of Using Views:

Simplicity:

Views allow you to encapsulate complex joins, subqueries, and filters into a single, reusable virtual table.

Example: Instead of writing a complex query every time, you can just select from a view.

Security:

Views can restrict user access to specific columns or rows, enhancing data privacy and security.

Example: Allowing access to employee names and departments without revealing salaries.

Data Abstraction:

Views can hide the complexity of the underlying database schema from end users.

Example: Users interact with a simplified view rather than complex joins across multiple tables.

Consistency:

Views ensure consistency by standardizing data presentation.

Example: Ensuring that all users see data calculated or formatted in the same way.

Common Pitfalls:

Performance Concerns:

Views can impact performance if they involve complex queries or large datasets, as the query is executed each time the view is accessed.

Dependency Issues:

Changes to the underlying tables, like column renaming or dropping, can break views, so careful management is needed.

"SQL Views are a powerful tool for simplifying data access, enhancing security, and ensuring consistency across your database. Whether you're a beginner or an advanced SQL user, mastering views can significantly improve your database management skills."

I trust you found this video helpful. Thank you!

# Stored procedure

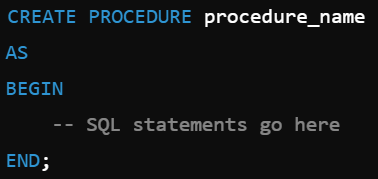
Hello Everyone!

Have you ever wondered how you can supercharge your SQL queries to perform complex tasks, automate routine operations, and boost your database performance? Today, we’re diving into the secret weapon behind it all—stored procedures! Ready to unlock the power?

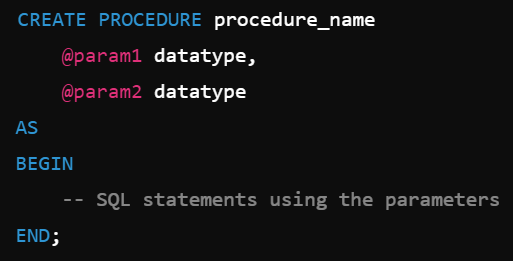
A stored procedure is a precompiled set of SQL statements that can be executed as a single unit to perform a specific task within a database.

Imagine you have a list of instructions you need to follow every day, like making coffee, checking emails, and organizing your desk. Instead of writing down these steps each time, you write them once and save the list. Now, whenever you want to follow these steps, you just pull out your saved list and get started.

Syntax to Create a Stored Procedure



Syntax to Create a Stored Procedure with Parameters



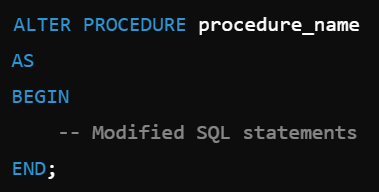
Executing a Stored Procedure



Executing a Stored Procedure with Parameters



Modifying an Existing Stored Procedure

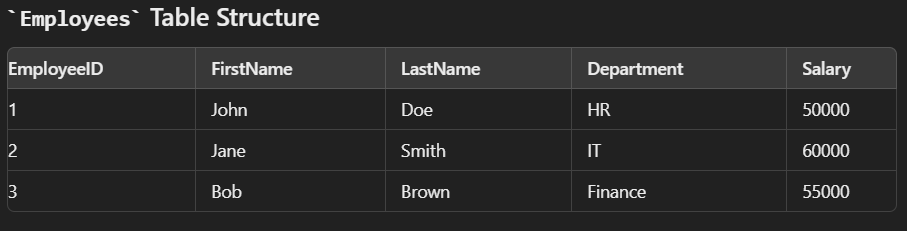


Dropping a Stored Procedure

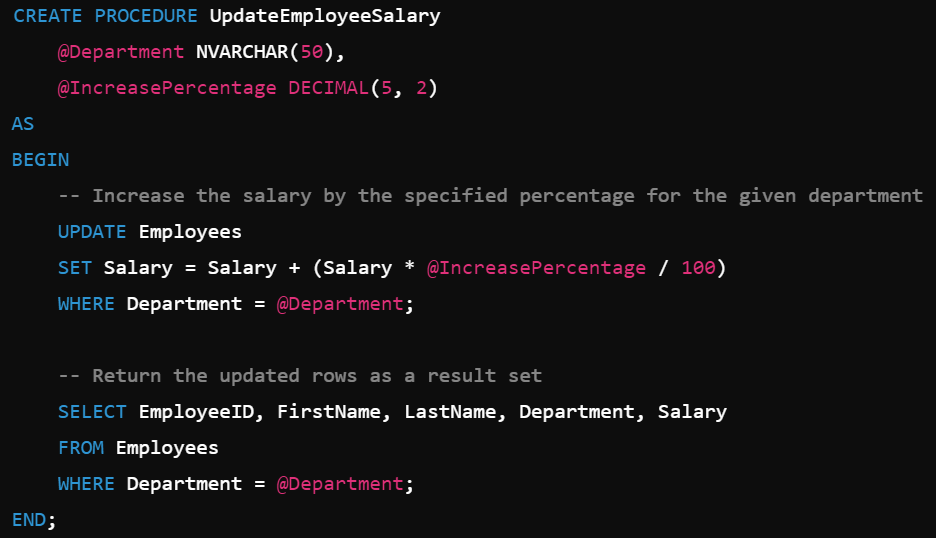


Example

Let's consider a `Employees` table:



Let’s create a Stored Procedure to Increase Salary by a Percentage



Explanation:

Parameters:

- `@Department`: Specifies the department for which the salary increase will be applied.

- `@IncreasePercentage`: Specifies the percentage by which to increase the salary.

UPDATE Statement:

- The `UPDATE` statement increases the salary of all employees in the specified department by the given percentage.

Result Set:

- After updating the salaries, the procedure returns the updated rows for review.

Executing the Stored Procedure

To increase the salary of employees in the IT department by 10%, you would execute the stored procedure like this:



Result:

After execution, the salaries of employees in the IT department will be updated by 10%, and the updated records will be displayed.

Stored procedures are powerful tools in SQL that offer numerous advantages. By encapsulating complex logic and repetitive tasks within the database, they enhance performance by reducing the amount of data transferred between the server and the client. Stored procedures also improve security by controlling access to sensitive operations and data, allowing users to execute predefined tasks without direct access to underlying tables. Additionally, they promote reusability and maintainability, as the logic is centralized and can be easily modified or reused across multiple applications. In essence, stored procedures streamline database management, optimize performance, and ensure consistency across your SQL operations.

I trust you found this video helpful. Thank you!

# Common Table Expression

Hello Everyone

"Did you know you can write SQL queries that reference themselves, turning complex problems into simple, readable code? Let me introduce you to the magic of Common Table Expressions (CTEs)!"

A Common Table Expression (CTE) in SQL is a temporary result set that you can reference within a `SELECT`, `INSERT`, `UPDATE`, or `DELETE` statement to simplify complex queries.

Syntax:

The WITH clause defines the CTE, and it can be referenced in the main query as if it were a temporary table.

Example:

Single CTE Example

Consider an employees table.

A CTE that calculates the average salary per department:

Output

Multiple CTE Example

Two CTEs: one to calculate the total salary per department, and another to calculate the average salary, then combining both results:

Output:

Recursive СТЕ

A Recursive CTE references itself, allowing it to repeatedly execute until a termination condition is met. It’s typically used for hierarchical data, such as organizational charts or tree structures, where each step depends on the previous result.

A recursive CTE consists of two parts:

* Anchor Member: The initial result set that forms the base case.
* Recursive Member: The part that references itself, progressively building new rows.

It continues executing until the termination condition is satisfied.

Example:

Let's consider an example of a recursive CTE that works with hierarchical data representing an organizational structure. In this case, we want to find all the levels of hierarchy in a CompanyHierarchy table.

CompanyHierarchy Table:

Objective:

We want to list all employees along with their reporting hierarchy, including their level in the hierarchy.

Recursive CTE Query:

Output

Explanation:

* Sarah is at level 1 because she has no manager (top-level).
* Robert reports to Sarah, so he is at level 2.
* Kevin and Emily both report to Robert, so they are at level 3.
* Michael reports to Kevin, and Rachel reports to Emily, so both are at level 4.

The recursive CTE helps in traversing the hierarchy and calculating the level of each employee in the organization.

Here are a few advantages of using CTEs in SQL:

Improved Readability and Maintenance: CTEs break down complex queries into smaller, logical steps, making the SQL code easier to read, understand, and maintain.

Reuse of Temporary Result Sets: CTEs allow you to reference the same temporary result set multiple times in the same query, reducing redundancy and improving query efficiency.

Recursive Queries: CTEs support recursion, enabling you to work with hierarchical or tree-structured data (e.g., organizational charts) more effectively.

Encapsulation of Subqueries: By using CTEs, you can encapsulate subqueries and make your main query less cluttered, improving both performance and clarity without creating temporary tables.

I trust you found this video helpful. Thank you!

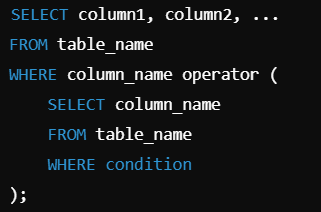
# Subquery

Hello Everyone

"Ever wondered how you can solve complex data problems by nesting queries within queries? Let’s dive into the world of subqueries and unlock the secrets to powerful, efficient SQL!"

A subquery in SQL is a nested query within another query that provides intermediate results used to filter or modify the main query's results.

Syntax



- Main query (SELECT column1, column2, ...):

Selects columns from a table.

- FROM table\_name:

Specifies the main table from which to select.

- WHERE column\_name operator:

Applies a condition based on the result of the subquery.

- Subquery:

A nested query that selects a column based on some condition,

and its result is compared with the main query using the

operator (=, IN, etc.).

- Subqueries can be used in various contexts such as in

WHERE, FROM, or SELECT clauses.

Example:

Let's consider a Orders table:

Find the OrderID and TotalAmount of orders placed by the customer who has the highest total order amount.

Inner Query (Subquery):

This subquery calculates the total order amount for each customer and selects the customer with the highest total amount.

GROUP BY CustomerID groups the orders by customer.

ORDER BY SUM(TotalAmount) DESC sorts the customers by their total order amount in descending order.

LIMIT 1 selects the top customer.

Outer Query:

This query retrieves the OrderID and TotalAmount for orders placed by the customer identified by the subquery.

Output:

I trust you found this video helpful. Thank you!

# Fact and Dimension Table

Hello Everyone

Ever wondered how massive e-commerce websites or your favorite streaming services organize their data to answer questions like *'Which product sold the most last month?'* or *'Who is watching action movies on Friday nights?'* It’s all about fact and dimension tables. Let’s break it down!

Fact and dimension tables are the foundation of data warehousing: facts store measurable data, while dimensions provide descriptive context to analyze that data.

Imagine you're analyzing sales data.

Think of a fact table as the 'what happened' table, it stores numbers like sales revenue or quantities sold.

Dimension tables are the 'who, what, where, and when' tables, they store details like product names, customer information, and sales dates.

Together, they allow you to slice, dice, and analyze your data efficiently.

Think of fact tables as the *verbs* (actions) and dimension tables as the *nouns* (descriptions) in your data story.

Let’s dive into the key differences between fact tables and dimension tables.

Purpose: Fact tables store the measurable data or facts, the 'what happened.' On the other hand, dimension tables store descriptive data or context, the 'who, what, when, and where.'

Data Type: Fact tables deal with numbers, quantitative data like sales amounts, quantities, or profits. Dimension tables, however, focus on textual or categorical data, such as product names, customer regions, or dates.

Granularity: Fact tables are highly granular, recording every specific event, like each sale or transaction. Meanwhile, dimension tables offer a hierarchical structure, categorizing data descriptively.

Examples: Imagine a sales scenario, fact tables would contain data like 'Sales Amount' or 'Quantity Sold,' while dimension tables provide details like 'Product Names,' 'Customer Demographics,' or 'Time Periods.'

Relationships: Fact tables act as the hub, linking to multiple dimension tables to provide context to the raw numbers. Dimension tables connect back to the fact table, forming a web of meaning around the facts.

Updates: Fact tables are dynamic, they grow rapidly with new transactions. Dimension tables, in contrast, are relatively static, updated only when new categories, products, or attributes are introduced.

Now you know how fact and dimension tables work together like peanut butter and jelly to help businesses make data-driven decisions.

Want to learn how to design these tables? Stay tuned for more! Don’t forget to like, subscribe, and hit the bell icon for updates!

I trust you found this video helpful. Thank you!

# LIMIT & OFFSET

Hello Everyone

Have you ever scrolled through a long list of search results on an e-commerce website and wondered, how do they show just 10 items per page? Let’s dive into the secret behind it—LIMIT and OFFSET in SQL!"

"LIMIT is an SQL clause used to specify the maximum number of rows to return in a query result."

"OFFSET is an SQL clause used to specify the number of rows to skip before starting to return query results."

"Imagine you have a giant stack of books. Instead of flipping through all of them to find what you need, you decide to look at only the first 10 books. That’s LIMIT, it tells SQL to fetch only a specified number of rows.

Now, OFFSET is like starting from the 11th book instead of the 1st. Together, LIMIT and OFFSET help you manage which part of your data you want to see at any given time."

Consider a table named Students:

**Query with LIMIT**:

SELECT star FROM Students LIMIT 3

**Output**:

**Query with LIMIT and OFFSET**:

SELECT star FROM Students LIMIT 2 OFFSET 2

**Output**:

**Applications**

**Pagination in Web Applications**: Displaying a fixed number of rows per page in user interfaces, like search results or product listings.

**Data Sampling**: Fetching subsets of data for analysis or testing without overloading the system.

**Scroll-Based Data Loading**: Implementing infinite scrolling by dynamically fetching additional rows as users scroll down.

**Efficient Reporting**: Generating smaller chunks of data for display in dashboards or reports to optimize performance.

"And there you have it—LIMIT and OFFSET in SQL demystified! Next time you browse your favorite website or work with large datasets, you’ll know the SQL magic behind it. Don’t forget to like, share, and subscribe for more SQL tips and tricks!"

I trust you found this video helpful. Thank you!

# ###################

# #################

## Primary key vs foreign key vs composite primary key vs candidate key

**Narration Script for YouTube Video: "Primary Key vs Foreign Key vs Composite Primary Key vs Candidate Key"**

**Exciting Opener**

"Did you know that behind every perfectly working database lies a set of keys that keep everything organized, accessible, and connected? But what exactly are these mysterious keys, and how do they work together? Let's find out!"

**Single-Line Definitions**

1. **Primary Key**: A unique identifier for a record in a table.
2. **Foreign Key**: A field in one table that links to the primary key of another table.
3. **Composite Primary Key**: A primary key made up of two or more columns to uniquely identify a record.
4. **Candidate Key**: A column, or a set of columns, that can uniquely identify any record in a table and is a potential primary key.

**Explanation in Simple Terms**

Imagine a library. Each book has a unique ISBN number—this is like the **primary key**, as it uniquely identifies a book in the database. Now, if the library's borrowing system has a table tracking who borrowed which book, that table might reference the ISBN as a **foreign key** to link borrowed records back to the books.

Sometimes, one field isn’t enough to identify a record uniquely. For example, in a competition, a participant's ID and the event name together might form a **composite primary key** to ensure uniqueness across multiple events.

Finally, every table can have multiple columns that could uniquely identify records—these are **candidate keys**. For instance, a book's ISBN or its title and author combination could both uniquely identify it, but you only choose one as the primary key.

**Example Using Dummy Data**

**Example Using Dummy Data**

**To explain Primary Key, Foreign Key, Composite Primary Key, and Candidate Key, let’s use a hypothetical E-commerce System.**

**Tables**

1. **Customers Table  
   This table stores customer details.**

| **CustomerID (Primary Key)** | **Name** | **Email** | **Phone** |
| --- | --- | --- | --- |
| **101** | **Alice** | [**alice@mail.com**](mailto:alice@mail.com) | **123-456-7890** |
| **102** | **Bob** | [**bob@mail.com**](mailto:bob@mail.com) | **987-654-3210** |
| **103** | **Charlie** | [**charlie@mail.com**](mailto:charlie@mail.com) | **456-789-1234** |

* + **Primary Key: CustomerID uniquely identifies each customer.**
  + **Candidate Key: Both Email and Phone could also uniquely identify a customer, but only one can be chosen as the primary key.**

1. **Orders Table  
   This table records orders placed by customers.**

| **OrderID (Primary Key)** | **CustomerID (Foreign Key)** | **OrderDate** | **Amount** |
| --- | --- | --- | --- |
| **201** | **101** | **2024-12-01** | **100.00** |
| **202** | **102** | **2024-12-02** | **50.00** |
| **203** | **101** | **2024-12-03** | **75.00** |

* + **Primary Key: OrderID uniquely identifies each order.**
  + **Foreign Key: CustomerID links this table to the Customers table, creating a relationship between orders and customers.**

1. **OrderDetails Table  
   This table records items in each order.**

| **OrderID** | **ItemID** | **Quantity** | **Price** |
| --- | --- | --- | --- |
| **201** | **A1** | **2** | **20.00** |
| **201** | **B2** | **1** | **60.00** |
| **202** | **C3** | **1** | **50.00** |

* + **Composite Primary Key: The combination of OrderID and ItemID uniquely identifies each record because one order can contain multiple items.**
  + **Without both columns together, records like 201, A1 and 201, B2 wouldn’t be uniquely identified.**

**Relationships Between Tables**

* **The Customers Table is linked to the Orders Table via the CustomerID (Foreign Key in Orders, Primary Key in Customers).**
* **The Orders Table is linked to the OrderDetails Table via the OrderID (Foreign Key in OrderDetails, Primary Key in Orders).**
* **The OrderDetails Table uses a Composite Primary Key to uniquely identify individual items in an order.**

**This setup demonstrates how:**

* **Primary Keys ensure each row in a table is unique.**
* **Foreign Keys establish relationships between tables.**
* **Composite Primary Keys uniquely identify rows using a combination of columns.**
* **Candidate Keys provide alternative options to uniquely identify records.**

**This design enforces data consistency, avoids duplication, and makes querying easier.**

**Real-Life Analogies**

1. **Primary Key**: Your Social Security Number—unique and identifies you in government records.
2. **Foreign Key**: A student ID on a library card—links the student to their borrowing records.
3. **Composite Primary Key**: Your first and last name combined might uniquely identify you in a small class but not in the entire school.
4. **Candidate Key**: Multiple ways to identify a car—license plate or VIN, but only one is chosen as the official identifier.

**Applications**

1. **Data Integrity**: Ensures no duplicate records exist, maintaining a clean database.
2. **Data Relationships**: Links data across tables for powerful relational queries.
3. **Efficient Queries**: Speeds up searches and lookups with well-defined keys.
4. **Data Validation**: Ensures only valid, related data is entered into tables using foreign keys.

**Interesting Facts**

1. The concept of primary keys originated with early database systems in the 1970s.
2. Foreign keys can enforce "referential integrity," ensuring data consistency between tables.
3. Composite primary keys are often used in many-to-many relationships.
4. A candidate key that isn’t chosen as a primary key is sometimes called an "alternate key."

"That’s it for today! Now you know how these keys unlock the full potential of databases. So, what’s your favorite analogy from this video? Let me know in the comments below. And don’t forget to like, share, and subscribe for more tech insights. Until next time, keep learning!"

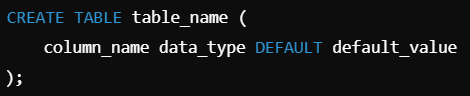
## Alias and Default constraint

Hello Everyone

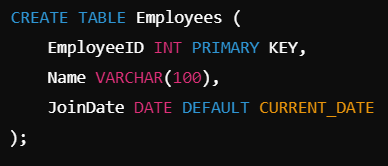
"Ever wished your database could automatically fill in missing values with defaults, just like a safety net? Let’s explore how SQL’s Default Constraint can keep your data clean and reliable with minimal effort!"

A Default Constraint in SQL automatically assigns a default value to a column when no explicit value is provided during an `INSERT` operation.

Syntax



Example



In this example, if no value is provided for the JoinDate column during an INSERT, the current date will be used as the default value.

Here are four key advantages of using a Default Constraint in SQL:

Data Consistency: Default constraints ensure that columns always have a predefined value if no other value is provided, which helps maintain consistency in the data.

Simplified Data Entry: They simplify data entry and reduce the likelihood of errors by automatically providing default values, which means users don’t have to specify values for every column.

Improved Data Integrity: Default constraints prevent null values in columns that should always have meaningful data, thereby enhancing data integrity and reliability.

Reduced Application Complexity: By handling default values at the database level, the complexity of application logic is reduced, as the database ensures that appropriate values are set automatically.

I trust you found this video helpful. Thank you!

## COALESCE() vs ISNULL()

COALESCE(): COALESCE function in SQL returns the first non-NULL expression among its arguments. If all the expressions evaluate to null, then the COALESCE function will return null.

Syntax:

SELECT column(s), CAOLESCE(expression\_1......expression\_n)FROM table\_name;

ISNULL(): The ISNULL function has different uses in SQL Server and MySQL. In SQL Server, ISNULL() function is used to replace NULL values.

Syntax:

SELECT column(s), ISNULL(column\_name, value\_to\_replace) FROM table\_name;

## CONCAT() Function

## ACID properties

A transaction is a single logical unit of work that accesses and possibly modifies the contents of a database. Transactions access data using read-and-write operations. In order to maintain consistency in a database, before and after the transaction, certain properties are followed. These are called ACID properties. ACID (Atomicity, Consistency, Isolation, Durability) is a set of properties that guarantee that database transactions are processed reliably. For more details please read ACID properties in the DBMS article.

# #################

## Window Function

"Ever wondered how to rank, sum, or calculate moving averages in your data without breaking a sweat? Enter window functions! But what makes them so powerful in SQL?"

### Rank vs Dense\_Rank

Provide a use case for each of the functions

### Row\_number

Provide a use case for each of the functions

### Lag

### Lead

### SUM() Over

### Avg() Over

### count() Over

### NTILE()

### PERCENT\_RANK()

### FIRST\_VALUE()

### LAST\_VALUE()

### Nth\_VALUE()

## NVL vs NVL2

These functions work with any data type and pertain to the use of null values in the expression list.

These are all single-row functions i.e. provide one result per row.

NVL(expr1, expr2): In SQL, NVL() converts a null value to an actual value. Data types that can be used are date, character, and number. Data types must match with each other. i.e. expr1 and expr2 must be of the same data type.

Syntax:

NVL (expr1, expr2)

NVL2(expr1, expr2, expr3): The NVL2 function examines the first expression. If the first expression is not null, then the NVL2 function returns the second expression. If the first expression is null, then the third expression is returned i.e. If expr1 is not null, NVL2 returns expr2. If expr1 is null, NVL2 returns expr3. The argument expr1 can have any data type.

Syntax: NVL2 (expr1, expr2, expr3)

## Narrow and Wide format

## Normalization

Ever wondered how databases organize their data efficiently to prevent redundancy and ensure reliability? Let's delve into the world of normalization, where we explore the art of structuring data to optimize database performance and maintain data integrity!

Normalization is the process of organizing data in a database to reduce redundancy and dependency by dividing large tables into smaller, related tables.

Normalization helps organize tables and their relationships to achieve optimal database design.

Why is Normalization Required?

Normalization is required to:

Minimize Redundancy: Reduces data duplication, leading to storage efficiency.

Eliminate Update Anomalies: Ensures data consistency when updating records. Modifications are made in only one place, avoiding inconsistent data.

Prevent Insertion Anomalies: Facilitates adding new records without unnecessary duplication. Ensures that adding new data does not require adding unrelated data.

Avoid Deletion Anomalies: Allows for removal of data without unintended loss of information. Removal of data does not cause unintended loss of information in related tables.

Consider a denormalized table of `Employees`:

A screenshot of a computer

Description automatically generated

### Explanation of Normalization Benefits

1. \*\*Minimize Redundancy\*\*

- \*\*Issue:\*\* Redundant data leads to inefficient storage.

- \*\*Example:\*\* In the table above, both `Manager\_ID` and `Manager\_Name` are repeated for each employee, causing redundancy.

- \*\*Solution:\*\* Normalize by creating separate tables for `Employees`, `Departments`, and `Managers` to store data without repetition.

2. \*\*Eliminate Update Anomalies\*\*

- \*\*Issue:\*\* Modifying data in one place but not in related places can lead to inconsistencies.

- \*\*Example:\*\* If John's name (`Manager\_Name`) changes to "Johnny", every occurrence of "John" in `Manager\_Name` needs updating.

- \*\*Solution:\*\* Normalize by having `Manager\_ID` as a foreign key in the `Employees` table, referencing a `Managers` table where names are stored uniquely.

3. \*\*Prevent Insertion Anomalies\*\*

- \*\*Issue:\*\* Adding new data requires inserting the same information multiple times.

- \*\*Example:\*\* To add a new department with a new manager, one must repeat the manager's information across multiple employee entries.

- \*\*Solution:\*\* Normalize by having separate tables for `Employees` and `Managers`, linked by foreign keys, so adding a new employee or manager does not duplicate unrelated data.

4. \*\*Avoid Deletion Anomalies\*\*

- \*\*Issue:\*\* Removing data unintentionally causes loss of related information.

- \*\*Example:\*\* Deleting an employee might inadvertently delete their department or manager information.

- \*\*Solution:\*\* Normalize by establishing relationships between tables (`Employees` to `Managers` via `Manager\_ID`) so that deleting an employee does not affect unrelated data like department or manager information.

### Normalized Structure

\*\*Employees:\*\*

A screenshot of a computer

Description automatically generated

\*\*Managers:\*\*

A screenshot of a computer

Description automatically generated

\*\*Departments:\*\*

A screenshot of a computer

Description automatically generated

Below are the Different Normal Forms (NF) in SQL:

1. \*\*First Normal Form (1NF):\*\* Ensures each column contains atomic (indivisible) values and there are no repeating groups or arrays.

2. Second Normal Form (2NF):\*\* Meets 1NF requirements and ensures all non-key attributes are fully dependent on the primary key, eliminating partial dependencies.

3. \*\*Third Normal Form (3NF):\*\* Meets 2NF requirements and ensures that no transitive dependencies exist, meaning no non-key column is dependent on another non-key column.

4. \*\*Boyce-Codd Normal Form (BCNF):\*\* A stricter version of 3NF where every determinant must be a candidate key, ensuring all functional dependencies are satisfied.

5. \*\*Fourth Normal Form (4NF):\*\* Addresses multi-valued dependencies where a table has more than one multi-valued dependency, ensuring no independent multi-valued facts within a single table.

6. \*\*Fifth Normal Form (5NF):\*\* Ensures that a database is free from join dependencies and anomalies stemming from join operations.

## First Normal Form (1NF)

Ever wondered how databases ensure data is neatly organized without any messy duplicates or awkward groupings? Let's dive into the First Normal Form (1NF), where we explore the foundational rule that makes sure every piece of data plays nicely in its own space!

First Normal Form (1NF) ensures that each column in a table contains atomic (indivisible) values and there are no repeating groups of columns.

### Explanation of First Normal Form (1NF):

1. \*\*Atomic Values\*\*: Every column in a table must contain atomic (indivisible) values. This means that a column should not have multiple values or repeating groups.

2. \*\*Unique Column Names\*\*: Each column in a table must have a unique name. This ensures clarity and avoids ambiguity in data retrieval and updates.

3. \*\*No Repeating Groups\*\*: Each column should contain data that directly relates to the primary key, and each row should be unique.

### Example Input Table:

Consider a table that stores information about students and the courses they are enrolled in. A non-1NF table might look like this:

A white rectangular object with a black border

Description automatically generated with medium confidence

This table violates 1NF because:

- The `Courses` column contains multiple values (comma-separated list), violating the atomicity rule.

- It's not immediately clear which course belongs to which student without parsing the `Courses` column.

To convert the given example input table into the First Normal Form (1NF), we need to ensure that each column contains atomic values and that there are no repeating groups. Here's how the table would look in 1NF:

\*\*Student Table:\*\*

A screenshot of a computer

Description automatically generated

In this 1NF table:

- Each column (`Student ID`, `Student Name`, `Course`) contains atomic values.

- There are no repeating groups, and each row represents a unique combination of student and course.

This structure adheres to the principles of 1NF by organizing data into a format that supports efficient querying, data integrity, and easier maintenance.

### Importance of First Normal Form (1NF):

1. \*\*Data Integrity\*\*: Ensures each piece of data is stored in a structured, clear format, reducing the risk of inconsistencies and errors.

2. \*\*Efficient Querying\*\*: Allows for efficient querying and indexing of data, as each column contains only single, atomic values.

3. \*\*Flexibility\*\*: Facilitates easier updates and modifications to the database schema without the need for complex data restructuring.

4. \*\*Standardization\*\*: Provides a standardized way of organizing data, making it easier for developers and users to understand and work with the database.

By adhering to 1NF, databases become more robust and easier to maintain, ensuring that data remains accurate and accessible over time.

## Second Normal Form (2NF)

How can a slight tweak in database design turn confusion into clarity? Let's dive into the Second Normal Form (2NF) and uncover its transformative power!

Second Normal Form (2NF) ensures that a table is in First Normal Form (1NF) and that all non-key attributes are fully functionally dependent on the primary key.

### Explanation of Second Normal Form (2NF):

Second Normal Form (2NF) builds upon First Normal Form (1NF) by addressing issues related to partial dependencies within a table.

1. \*\*First Normal Form (1NF) Recap\*\*: Ensures atomicity of columns (no multi-valued attributes) and unique column names.

2. \*\*Second Normal Form (2NF) Requirement\*\*: In addition to 1NF, every non-key column must be fully functionally dependent on the entire primary key.

Example Input Table and Output:

Consider a table that stores information about students and the courses they are enrolled in, including information about the instructors:

\*\*Input Table (Not in 2NF):\*\*

A screenshot of a computer

Description automatically generated

This table is not in 2NF because:

- The `Instructor` column depends only on the `Course`, not the entire primary key (`Student ID`, `Course`).

\*\*Output Table (In 2NF):\*\*

To normalize this table into 2NF, we would split it into two tables:

\*\*Students Table:\*\*

A screenshot of a computer

Description automatically generated

\*\*Courses Table:\*\*

A screenshot of a computer

Description automatically generated

### Importance of Second Normal Form (2NF):

1. \*\*Eliminates Redundancy\*\*: By organizing data into separate tables, each containing a unique set of information, redundancy is minimized, and data consistency is improved.

2. \*\*Maintains Data Integrity\*\*: Ensures that each piece of data is stored in only one place, reducing the risk of anomalies such as update anomalies (where updating one piece of information leads to inconsistencies).

3. \*\*Improves Query Performance\*\*: With well-defined relationships between tables, querying becomes more efficient, as data retrieval does not require scanning unnecessary columns.

4. \*\*Facilitates Scalability\*\*: Allows for easier expansion and modification of the database schema as new requirements arise, without the need for extensive restructuring.

By understanding and implementing Second Normal Form (2NF), database designs become more robust, efficient, and adaptable to changing needs, making data management more effective in both small and large-scale applications.

## Third Normal Form (3NF)

Ever wondered how database design can empower data integrity and streamline information retrieval? Let's unravel the transformative principles of Third Normal Form (3NF) and discover its impact on shaping efficient databases!

Third Normal Form (3NF) ensures that a table is in Second Normal Form (2NF) and eliminates transitive dependencies by requiring that non-key attributes depend only on the primary key.

### Explanation of Third Normal Form (3NF):

Third Normal Form (3NF) further refines database design by addressing transitive dependencies within tables. Here's a beginner-friendly explanation along with an example:

1. \*\*First Normal Form (1NF) and Second Normal Form (2NF) Recap\*\*: Ensures atomicity of columns and eliminates partial dependencies respectively.

2. \*\*Third Normal Form (3NF) Requirement\*\*: In addition to 2NF, every non-key column must depend only on the primary key (no transitive dependencies).

### Example Input Table and Output:

Consider a table that stores information about employees, their departments, and the locations of those departments:

\*\*Input Table (Not in 3NF):\*\*

A screenshot of a computer

Description automatically generated

This table is not in 3NF because:

- There's a transitive dependency where `Department Location` depends on `Department`, which is not part of the primary key (`Employee ID`).

\*\*Output Table (In 3NF):\*\*

To normalize this table into 3NF, we would split it into two tables:

\*\*Employees Table:\*\*

A screenshot of a computer

Description automatically generated

\*\*Departments Table:\*\*

A screenshot of a phone

Description automatically generated

\*\*EmployeeDepartments Table:\*\*

A screenshot of a computer

Description automatically generated

### Importance of Third Normal Form (3NF):

1. \*\*Reduces Redundancy\*\*: By organizing data into separate tables and ensuring each table has a distinct purpose (no transitive dependencies), redundancy is minimized, leading to more efficient storage and maintenance of data.

2. \*\*Enhances Data Integrity\*\*: Prevents anomalies such as insertion, update, and deletion anomalies that can occur when data dependencies are not properly managed.

3. \*\*Simplifies Database Maintenance\*\*: With well-defined relationships and clear dependencies, database administrators can more easily modify and extend the database schema as business requirements evolve.

4. \*\*Improves Query Efficiency\*\*: Queries become more straightforward and efficient as they only need to access relevant tables and columns, reducing the complexity of joins and improving overall performance.

By adhering to Third Normal Form (3NF), databases are structured in a way that supports data integrity, flexibility, and efficiency, making them more robust and suitable for scalable applications.

## Boyce-Codd Normal Form (BCNF)

Did you know that there's a higher level of database normalization that goes beyond Third Normal Form (3NF) to ensure even stricter data integrity? Let's uncover the transformative power of Boyce-Codd Normal Form (BCNF) and how it revolutionizes database design!

Boyce-Codd Normal Form (BCNF) ensures that a table is in Third Normal Form (3NF) and that every determinant in the table is a candidate key.

Explanation of Boyce-Codd Normal Form (BCNF):

Boyce-Codd Normal Form (BCNF) is a stricter form of normalization that ensures every determinant (attributes determining other attributes) in a table is a candidate key. Here's a simplified explanation along with an example:

1. \*\*First Normal Form (1NF), Second Normal Form (2NF), and Third Normal Form (3NF) Recap\*\*: Address atomicity of columns, eliminate partial dependencies, and eliminate transitive dependencies respectively.

2. \*\*Boyce-Codd Normal Form (BCNF) Requirement\*\*: In addition to 3NF, every determinant must be a candidate key, meaning there are no non-trivial functional dependencies of attributes on anything other than a candidate key.

### Example Input Table and Output:

Consider a table that stores information about employees and the projects they are assigned to, including the project managers:

\*\*Input Table (Not in BCNF):\*\*

A screenshot of a project

Description automatically generated

This table is not in BCNF because:

- There's a dependency where `Project Manager` depends on `Project ID`, which is not a candidate key by itself.

\*\*Output Table (In BCNF):\*\*

To normalize this table into BCNF, we would typically break it down into multiple tables:

\*\*Employees Table:\*\*

A screenshot of a computer

Description automatically generated

\*\*Projects Table:\*\*

A screenshot of a project

Description automatically generated

\*\*ProjectAssignments Table:\*\*

A screenshot of a computer

Description automatically generated

\*\*ProjectManagers Table:\*\*

A screenshot of a computer

Description automatically generated

### Importance of Boyce-Codd Normal Form (BCNF):

1. \*\*Eliminates Redundancy and Anomalies\*\*: By ensuring that every determinant is a candidate key, BCNF eliminates the possibility of update, insertion, and deletion anomalies that can occur in less normalized forms.

2. \*\*Maintains Data Integrity\*\*: With clearly defined dependencies and relations between tables, BCNF ensures that data remains accurate and consistent throughout the database.

3. \*\*Improves Query Efficiency\*\*: Queries can be optimized since tables are structured in a way that minimizes unnecessary joins and ensures that only relevant data is accessed.

4. \*\*Supports Scalability and Flexibility\*\*: BCNF facilitates easier database maintenance and schema modifications as the application evolves, ensuring the database remains efficient and adaptable to changing business needs.

By adhering to Boyce-Codd Normal Form (BCNF), databases are not only more robust and reliable but also more efficient in managing and retrieving data, making them suitable for complex and large-scale applications.

## Fourth Normal Form (4NF)

- Addresses multi-valued dependencies.

- Example: Splitting multi-valued attributes into separate tables.

## Fifth Normal Form (5NF)

- Deals with cases where a non-trivial join dependency exists.

- Example: Further decomposing tables to remove join dependencies.

## ###### Interview Important ######

## Partition By

Provide a use case for each of the functions

## EXCEPT

This operation includes tuples that are present in one relationship but should not be present in another relationship. For example: To find customers who have an account but no loan at the bank:

SELECT CustomerName FROM Depositor

EXCEPT

SELECT CustomerName FROM Borrower;

The Except operation automatically eliminates the duplicates. If all the duplicates are supposed to be retained, EXCEPT ALL is used in place of EXCEPT.

## Relationships in SQL

There are various relationships, namely:

• One-to-One Relationship.

• One to Many Relationships.

• Many to One Relationship.

• Self-Referencing Relationship

## Index

## concatenation operator

In SQL for appending two strings, the "concatenation operator" is used and its symbol is " || ".

## ########################################################

## Database Architecture

Database architecture refers to the design and structure of a database system, including the arrangement and interaction of its components. There are several different types of database architectures, each suited to different use cases and performance requirements. Here are some common types:

### 1. \*\*Single-Tier Architecture\*\*

\*\*Description\*\*:

- In a single-tier architecture, the database system consists of a single layer where the database and the application reside on the same system.

\*\*Use Cases\*\*:

- Suitable for small, simple applications with low data volume.

- Often used in standalone applications or small-scale desktop applications.

\*\*Advantages\*\*:

- Simple to design and implement.

- Low latency due to proximity of application and database.

\*\*Disadvantages\*\*:

- Limited scalability.

- Poor performance for larger datasets or higher user loads.

### 2. \*\*Two-Tier Architecture\*\*

\*\*Description\*\*:

- The two-tier architecture separates the database system into two layers: the client layer and the server layer.

- The client layer contains the application and the user interface, while the server layer contains the database management system (DBMS).

\*\*Use Cases\*\*:

- Suitable for small to medium-sized applications.

- Commonly used in client-server applications.

\*\*Advantages\*\*:

- Improved performance compared to single-tier architecture.

- Better separation of concerns between the client and server.

\*\*Disadvantages\*\*:

- Scalability is still limited.

- Increased complexity in managing client-server communication.

### 3. \*\*Three-Tier Architecture\*\*

\*\*Description\*\*:

- The three-tier architecture introduces an additional middle layer between the client and server, often called the application server or business logic layer.

- The three layers are: the presentation layer (client), the application layer (business logic), and the data layer (database).

\*\*Use Cases\*\*:

- Suitable for medium to large-scale applications.

- Common in web applications and enterprise systems.

\*\*Advantages\*\*:

- Better scalability and performance due to the separation of concerns.

- Improved security as the business logic layer can act as a firewall between the client and the database.

- Easier to maintain and update each layer independently.

\*\*Disadvantages\*\*:

- More complex to design and implement.

- Potentially higher latency due to additional layer of communication.

### 4. \*\*N-Tier Architecture\*\*

\*\*Description\*\*:

- N-tier architecture extends the three-tier architecture by adding more layers, such as caching servers, load balancers, and additional application servers.

- The number of tiers can vary based on the complexity and requirements of the application.

\*\*Use Cases\*\*:

- Suitable for large-scale, highly distributed applications.

- Commonly used in complex enterprise systems and cloud-based applications.

\*\*Advantages\*\*:

- Highly scalable and flexible.

- Improved fault tolerance and load balancing.

- Enhanced security and separation of concerns.

\*\*Disadvantages\*\*:

- Very complex to design, implement, and manage.

- Increased latency due to multiple layers of communication.

### 5. \*\*Distributed Database Architecture\*\*

\*\*Description\*\*:

- In distributed database architecture, the database is spread across multiple physical locations, often on different servers or even in different geographical areas.

- Can be either homogenous (same DBMS at all sites) or heterogeneous (different DBMS at different sites).

\*\*Use Cases\*\*:

- Suitable for applications requiring high availability and fault tolerance.

- Commonly used in global applications and large-scale systems.

\*\*Advantages\*\*:

- Improved reliability and availability.

- Enhanced performance through data localization and load distribution.

- Better fault tolerance and disaster recovery.

\*\*Disadvantages\*\*:

- Increased complexity in design and management.

- Challenges in maintaining data consistency and integrity.

### 6. \*\*Cloud Database Architecture\*\*

\*\*Description\*\*:

- Cloud database architecture involves databases hosted on cloud platforms, such as Amazon Web Services (AWS), Google Cloud Platform (GCP), or Microsoft Azure.

- Can be based on various underlying architectures (e.g., two-tier, three-tier, distributed).

\*\*Use Cases\*\*:

- Suitable for applications requiring scalability, flexibility, and reduced infrastructure management.

- Common in modern web applications, mobile apps, and data analytics.

\*\*Advantages\*\*:

- High scalability and flexibility.

- Reduced infrastructure costs and management overhead.

- Built-in redundancy, backup, and disaster recovery features.

\*\*Disadvantages\*\*:

- Dependency on cloud service providers.

- Potential security and compliance concerns.

- Latency and performance variability depending on network conditions.

### Conclusion

Each database architecture has its strengths and weaknesses, making them suitable for different scenarios. The choice of architecture depends on factors such as application requirements, expected load, data volume, and performance needs. Understanding these architectures helps in designing efficient and scalable database systems tailored to specific use cases.

## Database Relationships

One to One

One to Many

Many to Many

Icons/Logos

### ER Diagram

Logos

## SQL vs NoSQL database

Example database structure

## Char vs varchar vs varchar 2

Both of these data types are used for characters, but varchar2 is used for character strings of variable length, whereas char is used for character strings of fixed length. For example, if we specify the type as char(5) then we will not be allowed to store a string of any other length in this variable, but if we specify the type of this variable as varchar2(5) then we will be allowed to store strings of variable length. We can store a string of length 3 or 4 or 2 in this variable.

## WITH clause

The WITH clause provides a way relationship of defining a temporary relationship whose definition is available only to the query in which the with clause occurs. SQL applies predicates in the WITH clause after groups have been formed, so aggregate functions may be used.

## Left & Right



## CURRENT\_DATE

## INTERVAL

## Local and global variables

Global Variable: In contrast, global variables are variables that are defined outside of functions. These variables have global scope, so they can be used by any function without passing them to the function as parameters.

Local Variable: Local variables are variables that are defined within functions. They have local scope, which means that they can only be used within the functions that define them

## NULL values vs zero vs blank space

In SQL, zero or blank space can be compared with another zero or blank space. whereas one null may not be equal to another null. null means data might not be provided or there is no data.

## Important Topics

5. \*Data Modification:\*

- INSERT, UPDATE, DELETE statements

- Transactions and Rollback

7. \*Data Types:\*

- Understanding different data types (e.g., VARCHAR, INT, DATE)

- Type conversions and casting

8. \*Functions:\*

- Scalar functions (e.g., CONCAT, DATE functions)

- CASE expressions

- Window functions (e.g., ROW\_NUMBER, RANK)

11. \*Indexes and Optimization:\*

- Index types (e.g., B-tree, Hash)

- Query optimization techniques

- EXPLAIN statement

12. \*Stored Procedures and Triggers:\*

- Creating and executing stored procedures

- Working with triggers

13. \*Advanced SQL Concepts:\*

- Recursive queries

- Pivot and Unpivot operations

- Working with JSON data

14. \*Database Security:\*

- User management and permissions

- SQL injection prevention

15. \*Working with Multiple Databases:\*

- Connecting to and querying multiple databases

- Cross-database queries

16. \*NoSQL Databases (Optional):\*

- Basics of NoSQL databases (e.g., MongoDB, Cassandra)

17. \*SQL Best Practices:\*

- Writing efficient SQL queries

- Data modeling best practices

18. \*Version Control for SQL:\*

- Managing SQL scripts in version control systems like Git

19. \*Understanding Real-World Use Cases:\*

- Learning how SQL is applied in specific industries (e.g., finance, healthcare)

20. \*Practical Projects:\*

- Building databases and writing SQL queries for real projects to apply your knowledge.