Foundational SQL Queries

What is the difference between WHERE and HAVING?

You use **WHERE** for filtering rows before applying any grouping or aggregation. The code snippet below illustrates the use of **WHERE**. It filters the `Users` table for rows where the `Age` is greater than 18.

```
SELECT * FROM Users
WHERE Age > 18;
```

The result of the query is similar to the table below.

userId	firstName	lastName	age
1	John	Doe	30
2	Jane	Don	31
3	Will	Liam	25

userId	firstName	lastName	age
4	Wade	Great	32
5	Peter	Smith	27

On the other hand, you use **HAVING** to filter groups after performing grouping and aggregation. You apply it to the result of aggregate functions, and it is mostly used with the **GROUP BY** clause.

```
SELECT FirstName, Age FROM Users
GROUP BY FirstName, Age
HAVING Age > 30;
```

The code above selects the `FirstName` and `Age` columns, then groups by the `FirstName` and `Age`, and finally gets entries with age greater than 30. The result of the query looks like this:

firstName	age
Wade	32
Jane	31

How do you find duplicates in a table?

To find duplicate records, you must first define the criteria for detecting duplicates. Is it a combination of two or more columns where you want to detect the duplicates, or are you searching for duplicates within a single column?

The following steps will help you find duplicate data in a table.

- Use the **GROUP BY** clause to group all the rows by the column(s) on which you want to check the duplicate values.
- Use the **COUNT** function in the **HAVING** command to check if any groups have more than one entry.

Let's see how to handle single-column duplicates. In a table `Users`, there are three users who are 30 years of age. Let's use the **GROUP BY** clause and **COUNT** function to find the duplicate values.

```
SELECT Age, COUNT(Age)
FROM Users
GROUP BY Age
HAVING COUNT(Age) > 1
```

The result of the query looks like this:

age	count
30	3

Handling multi-column (composite) duplicates is similar to handling single-column duplicates.

```
SELECT FirstName, LastName, COUNT(*) AS dup_count
FROM Users
```

```
GROUP BY FirstName, LastName
HAVING COUNT(*) > 1;
```

After finding duplicates, you might be asked how to delete the duplicates. The query to delete duplicates is shown below using Common Table Expression (CTE) and ROW_NUMBER().

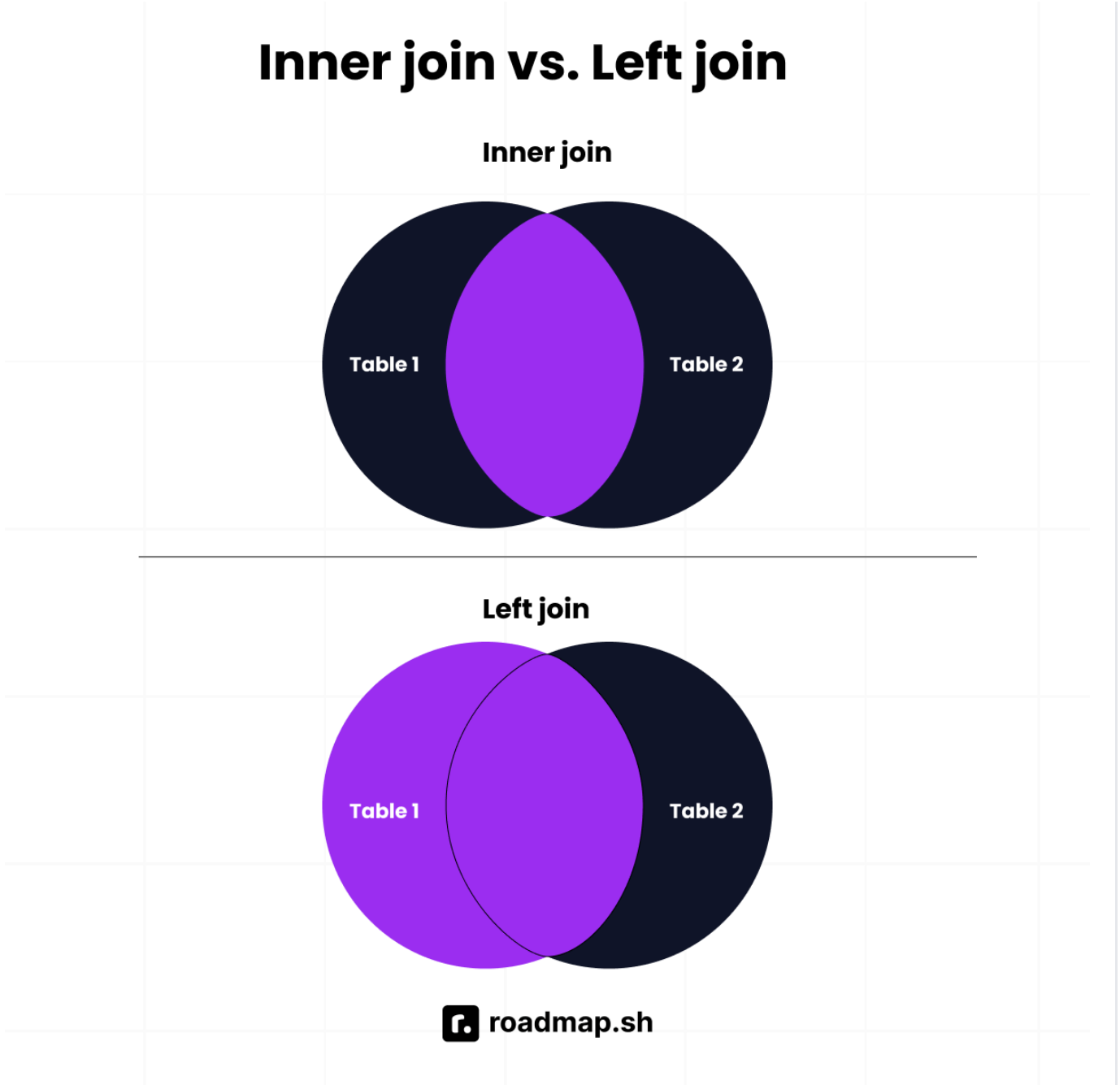
```
WITH ranked AS (
  SELECT *,
         ROW_NUMBER() OVER (PARTITION BY Age ORDER BY id) AS rn
  FROM Users
)
DELETE FROM Users
WHERE id IN (
  SELECT id
  FROM ranked
  WHERE rn > 1
);
```

The query deletes all the duplicates while retaining the first row of data.

What is the difference between INNER JOIN and LEFT JOIN?

A **JOIN** combines data from two or more tables based on a related column between them. It is useful when you need to retrieve data spread across multiple tables in relational database management systems.

An **INNER JOIN** returns only rows with a match in both tables based on the specified join condition. If there are no matching rows, there will be no results. The SQL syntax for an **INNER JOIN** is shown in the code snippet below.


Inner join vs. Left join
Inner join
Table 1
Table 2
Left join
Table 1
Table 2
roadmap.sh

```
SELECT table1.column_name1, table1.column_name2, table2.column_name
INNER JOIN table2
ON table1.column_name = table2.column_name
```

For example, there are two tables `Users` and `Cities` with the following data:

Users table

userId	firstName	lastName	age	cityId
1	John	Doe	30	1
2	Jane	Don	31	1
3	Will	Liam	25	1
4	Wade	Great	32	1
5	Peter	Smith	27	2
6	Rich	Mond	30	2

userId	firstName	lastName	age	cityId
7	Rach	Mane	30	2
8	Zach	Ridge	30	3

Cities table

id	name
1	London
2	Manchester

Let's say you want to retrieve a list of users and their respective city names. You can achieve this using the **INNER JOIN** query.

```
SELECT users.firstName, users.lastName, users.age, cities.name as c
INNER JOIN cities
ON users.cityId = cities.id
```

firstName	lastName	age	cityName
John	Doe	30	London
Jane	Don	31	London
Will	Liam	25	London
Wade	Great	32	London
Peter	Smith	27	Manchester
Rich	Mond	30	Manchester
Rach	Mane	30	Manchester

LEFT JOIN returns all the rows from the left table (table 1) and the matched rows from the right table (table 2). If no matching rows exist in the right table (table 2), then NULL values are returned. The SQL syntax for a Left join is shown in the code snippet below.

```
SELECT table1.column_name1, table1.column_name2, table2.column_name
LEFT JOIN table2
ON table1.column_name = table2.column_name
```

Let's have a look at a practical example with `Users` and `Cities` tables from before.

When you execute the **LEFT JOIN** query, you get the table below.

firstName	lastName	age	cityName
John	Doe	30	London
Jane	Don	31	London
Will	Liam	25	London
Wade	Great	32	London
Peter	Smith	27	Manchester

firstName	lastName	age	cityName
Rich	Mond	30	Manchester
Rach	Mane	30	Manchester
Zach	Ridge	30	null

Write a query to find the second highest salary from a table

Given a table `Salaries`,

id	salary
1	1000
2	2000
3	3000
4	4000

The query to find the second-highest salary is shown in the code snippet below

Master SQL with our new premium course START LEARNING →

```
SELECT DISTINCT Salary
FROM Salaries
ORDER BY Salary DESC
LIMIT 1 OFFSET 1
```

The result of the query is shown below

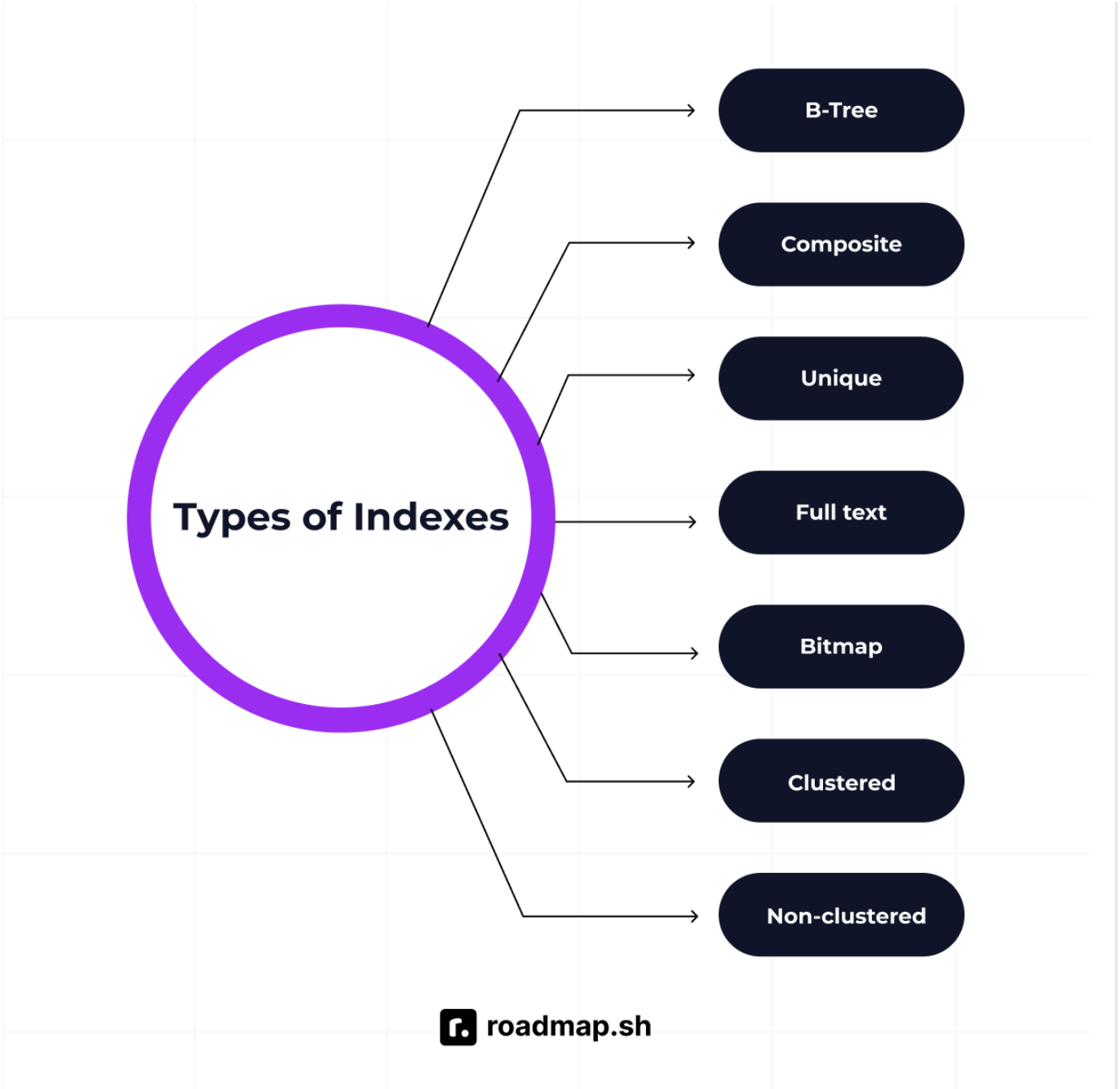
	salary
1	3000

What is the difference between UNION and UNION ALL?

UNION is used for removing duplicates while UNION ALL keeps all duplicates. UNION is slower compared to UNION ALL because of de-duplication. You use UNION when you want to obtain unique records and UNION ALL when you want every row even if they are repeated.

What are indexes and why are they useful?

Indexes in databases are like the indexes in books. They increase the speed of data retrieval from a database. When you want to read data from a table, instead of going through all the rows of the table, indexes help to go straight to the row you are looking for.


Types of Indexes
B-Tree
Composite
Unique
Full text
Bitmap
Clustered
Non-clustered
roadmap.sh

They improve **SELECT** queries, improve performance, and make sorting and filtering faster. They also ensure data integrity. There are different types of indexes, which include:

- B-Tree index
- Composite index
- Unique index
- Full text index
- Bitmap index
- Clustered index
- Non-clustered index

What is a primary key?

A primary key is the unique identifier of a row of data in a table. You use it to identify each row uniquely, and no two rows can have the same primary key. A primary key column cannot be null. In the example below, `user_id` is the primary key.

```
CREATE TABLE users (
   user_id INT PRIMARY KEY,
   name VARCHAR(100),
   phoneNumber VARCHAR(100)
);
```

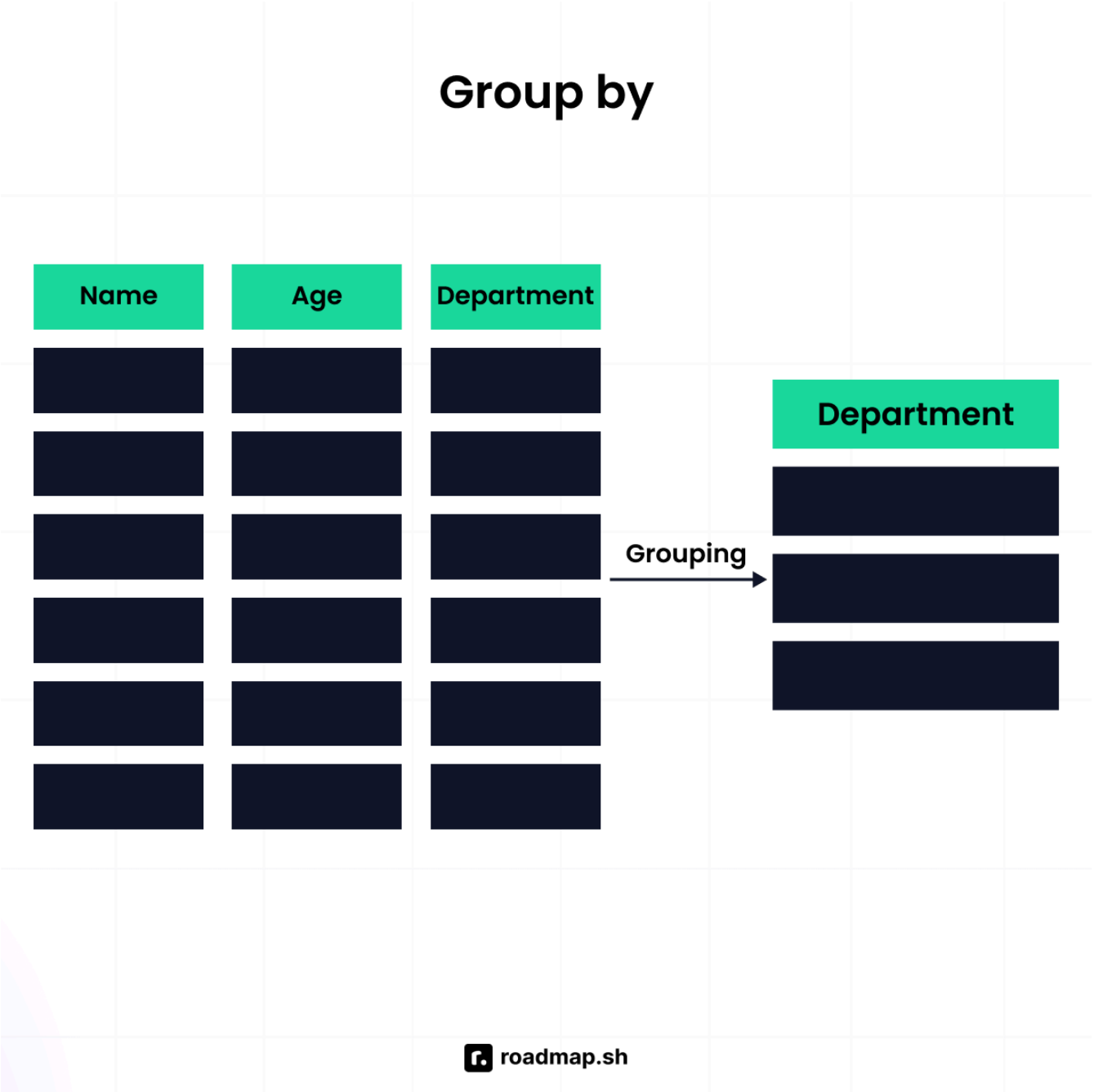
What is a foreign key?

A foreign key is like a bridge between two tables. A foreign key in one table is the primary key in another. It is the connector between the two tables.

Aggregation And Grouping

How does GROUP BY work?

GROUP BY is a standard SQL command that groups rows with the same value in the specified column. You should use with aggregate functions such as **COUNT**, **MIN**, **MAX,** etc.


Group by
Name
Age
Department
Grouping
Department
roadmap.sh

The query below illustrates the **GROUP BY** clause:

```
SELECT columnName FROM Table
GROUP BY columnName
```

What happens if you SELECT a column not in the GROUP BY clause?

If you **SELECT** a column not in the **GROUP BY** clause, it will throw an error stating that the column must be in the **GROUP BY** clause or in an aggregate function. Let's use the table below as an illustration.

firstName	lastName	phoneNumber
John	Doe	+23410910
Jack	Ray	+23410911
Irene	Rotherdam	+23410911

If you run the query below against the database:

```
SELECT firstName, phoneNumber FROM phoneNumbers
GROUP BY phoneNumber
```

The result will be an error because `firstName` is not in the **GROUP BY** clause and not using an aggregate function.

Write a query to COUNT the number of users by country

Given a table `Users` that looks like this:

userId	firstName	lastName	age	country
1	John	Doe	30	Portugal
2	Jane	Don	31	Belgium
3	Will	Liam	25	Argentina
4	Wade	Great	32	Denmark

userId	firstName	lastName	age	country
5	Peter	Smith	27	USA
6	Rich	Mond	30	USA
7	Rach	Mane	30	Argentina
8	Zach	Ridge	30	Portugal

The query to **COUNT** the number of users by country is:

```
SELECT country, COUNT(country) FROM users
GROUP BY country
```

The query uses the **GROUP BY** clause to group the users by country and then shows the count in the next column. The result of the query looks like this:

country	count
USA	2
Portugal	2
Argentina	2
Belgium	1
Denmark	1

What happens if you use GROUP BY without an aggregate function?

If you use the **GROUP BY** clause without an aggregate function, it is equivalent to using the **DISTINCT** command. For example, the command below:

```
SELECT phoneNumber FROM phoneNumbers
GROUP BY phoneNumber
```

is equivalent to:

```
SELECT DISTINCT phoneNumber FROM phoneNumbers
```

What is the difference between COUNT(*) and COUNT(column_name)?

The difference is that **COUNT(*)** counts all the rows of data, including NULL values, while **COUNT(column_name)** counts only non-NULL values in the specified column. Let's illustrate this using a table named `Users`.

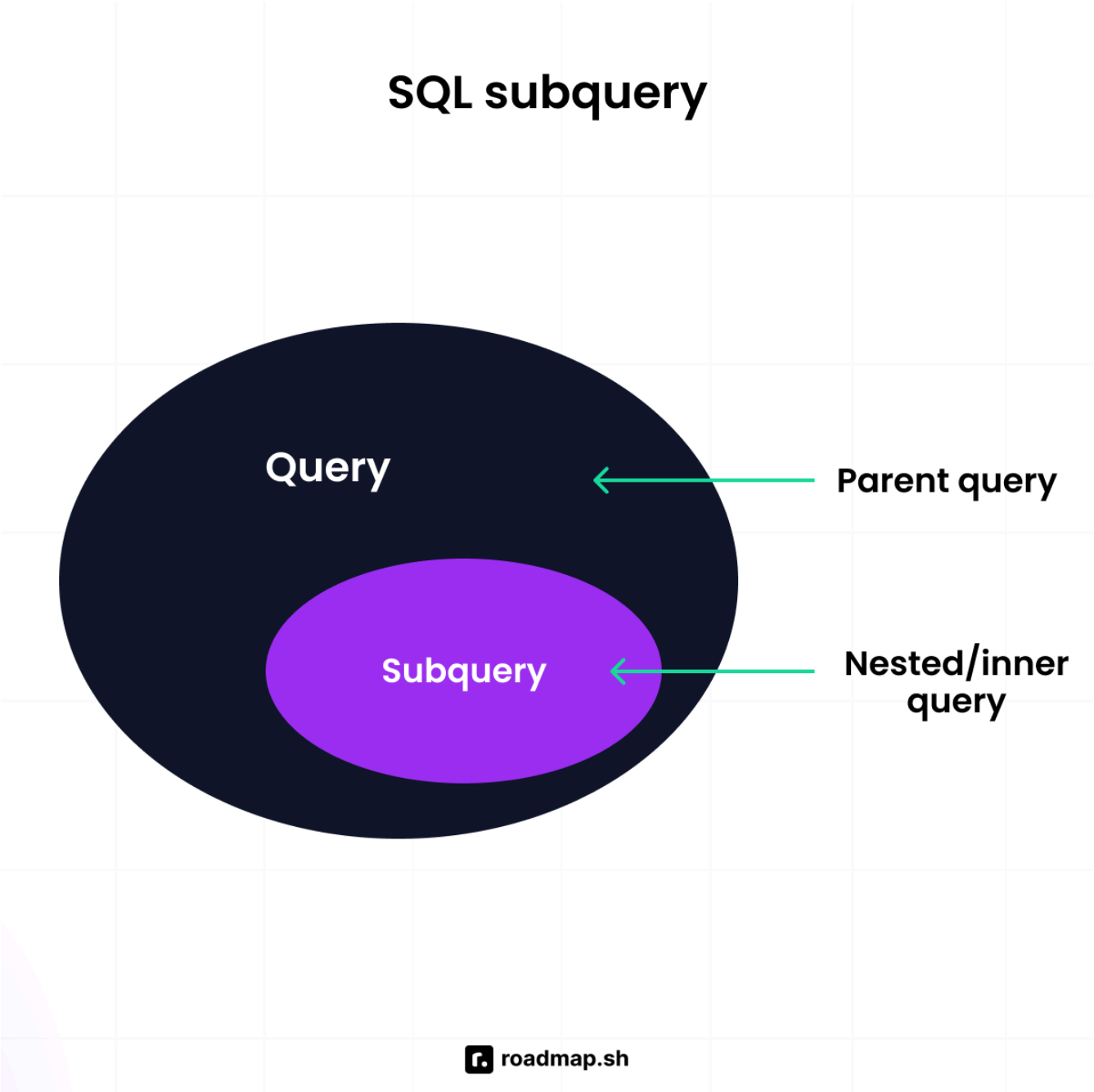
userId	firstName	lastName	age	country
1	John	Doe	30	Portugal
2	Jane	Don	31	Belgium
3	Zach	Ridge	30	Norway
4	null	Tom	25	Denmark

If you use **COUNT(*)**, the result will be 4 but if you use **COUNT(firstName)**, it will return 3, omitting the null value.

Subqueries And Nested Logic

What is the difference between a subquery and a JOIN?

A subquery is a query that is inside another query. You use it for queries that require complex logic. You should use subqueries when you want to use the result of that subquery for another query. In the example below, the subquery is in brackets.


SQL subquery
Query
Parent query
Subquery
Nested/inner query
roadmap.sh

```
SELECT firstName,
  (SELECT COUNT(*)
  FROM cities
  WHERE cities.id = users.city_id) AS cityCount
FROM users;
```

On the other hand, a **JOIN** combines two or more tables based on related columns between them. The related column is usually a foreign key. You should use **JOINS** when you want to pull related data from different tables together. The code below illustrates how to use a **JOIN**.

```
SELECT firstName, COUNT(*) FROM users
JOIN cities ON users.city_id = cities.id
```

A JOIN is faster than a subquery in the following scenarios:

- When you are querying data from multiple tables.
- When you are filtering or joining on index columns.

Write a query to find employees earning more than the average salary

Given an `Employees` table with columns `id`, `name`, and `salary` that looks like this:

id	name	salary
1	Irene	1000
2	Peter	1230
3	Raymond	1450
4	Henry	1790
5	Naomi	2350
6	Bridget	2000
7	Emily	2500
8	Great	3000

id	name	salary
9	Mercedes	2750
10	Zoe	2900

The query to find employees earning more than the average salary is:

```
SELECT * FROM employees
WHERE salary > (SELECT AVG(salary) FROM employees);
```

id	name	salary
5	Naomi	2350
7	Emily	2500
8	Great	3000
9	Mercedes	2750

id	name	salary
10	Zoe	2900

Explain how a correlated subquery works

A correlated subquery is a subquery that depends on a value from the outer query. This means that the query is evaluated for each row that might be selected in the outer query. Below is an example of a correlated subquery.

```
SELECT name, country_id, salary
FROM employees em
WHERE salary > (
  SELECT AVG(salary) FROM employees
  country_id = em.country_id);
```

The code above:

- Runs the outer query through each row of the table.
- Takes the `country_id` from the `employees` table.

- Iterates through the other rows and does the same calculation.

This leads to a degrading performance as the data in the table grows.

You should use a correlated subquery if you want to perform row-specific operations or cannot achieve an operation using JOIN or other aggregate functions.

When should you use EXISTS instead of IN in a subquery?

EXISTS and **IN** are used in subqueries to filter results, but they perform different functions depending on their usage.

You should use **EXISTS** in the following situations:

- When you want to check if a row exists and not the actual values.
- When the subquery is a correlated query.
- When the subquery returns many rows but you want to get the first match.

You should use **IN** in the following scenarios:

- When you are comparing a column to a list of values.
- When the subquery returns a small or static list.

Can you nest subqueries multiple levels deep?

Yes, you can nest subqueries multiple levels deep when you want to perform complex logic. A nested subquery is a subquery inside another subquery, forming layers of subqueries. Many SQL engines allow multiple layers of subqueries, but this causes poor readability and degrades performance.

Window Functions And Advanced Queries

What is a window function?

A window function is a function that allows you to perform operations on a specific set of rows related to the current row. Unlike aggregate functions that perform calculations on an entire data set, window functions can perform operations on a subset of data. These calculations are valid for

aggregates, ranking, and cumulative totals without altering the original dataset.

Write a query to calculate a running total

Let's use a table `Sales` as a reference for this query. It has three columns: `id`, `day` which represents the day of the week, and `amount` which is the amount sold in US Dollars. The table looks like this:

id	day	amount
1	Monday	200
2	Tuesday	300
3	Wednesday	600
4	Thursday	390
5	Friday	900

The query to calculate the running total is:

```
SELECT
  id,
  sale_date,
  amount,
  SUM(amount) OVER (ORDER BY sale_date) AS running_total
FROM
  sales;
```

The query uses a Window function **OVER** to sum the amount for each row of data and saving the running total. It gets the total for each day and adds it to the previous totals. The result of the query looks like this:

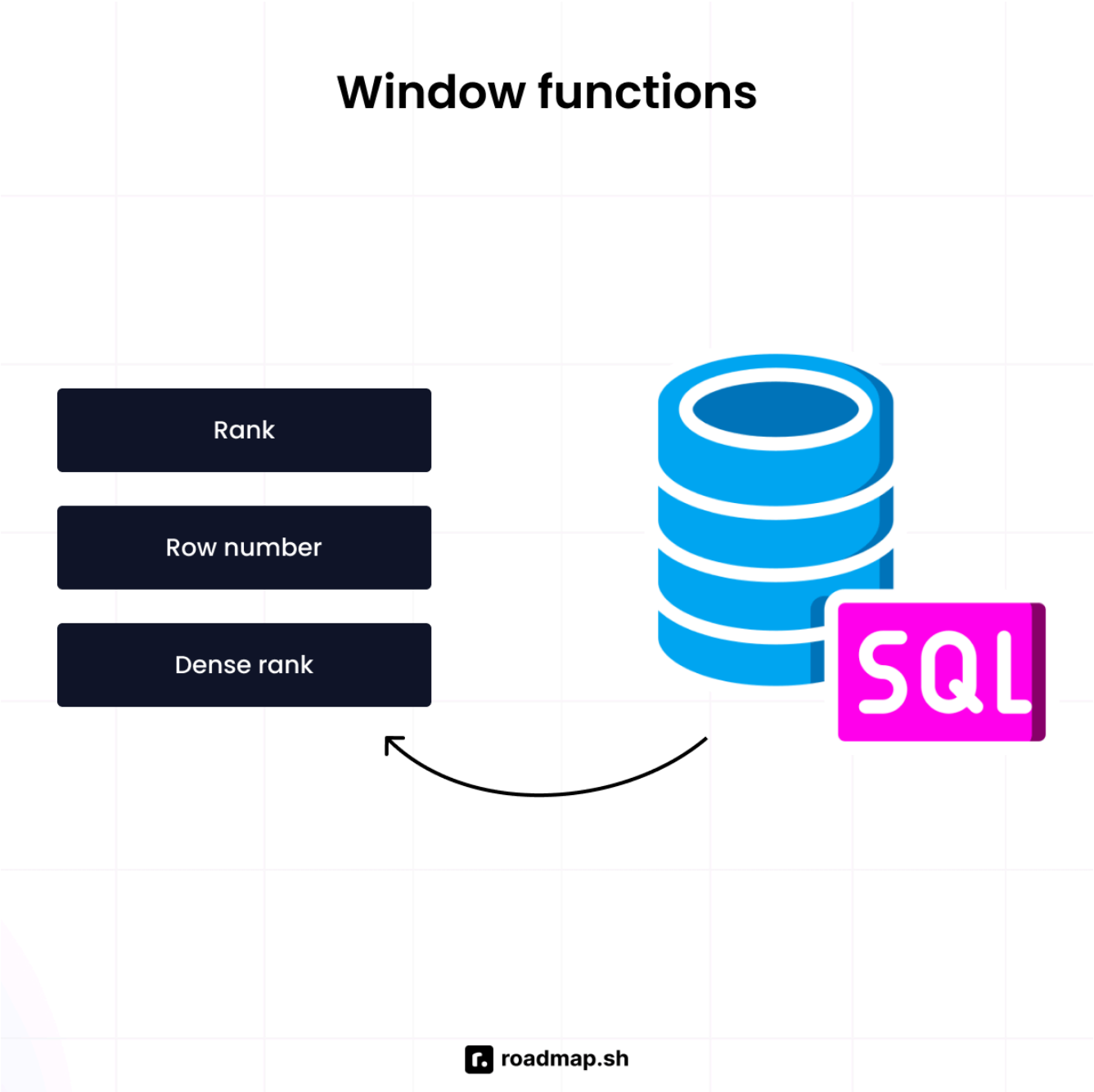
id	day	amount	running_total
1	Monday	200	200
2	Tuesday	300	500
4	Thursday	390	1100

id	day	amount	running_total
3	Wednesday	600	1490
5	Friday	900	2390

You can observe from the image that the last column is `running_total`, which takes the amount for the current day and adds it to its previous value to get its current value.

What is the difference between RANK(), DENSE_RANK(), and ROW_NUMBER()?

The **RANK()** function assigns each row a rank according to an ascending or descending order. If there are matching values, it assigns them the same position and then skips the next number for the next rank. For example, if two rows have equivalent values and are both assigned rank 1, the next rank would be 3 instead of 2.


Window functions
Rank
Row number
Dense rank
SQL
roadmap.sh

Let's use the `Sales` table from the previous question to illustrate the **RANK()** function. The query to rank in order of the amount looks like this:

```
SELECT
  id,
  day,
  amount,
  RANK() OVER (ORDER BY amount DESC) AS amount_rank
FROM
  sales;
```

The result is shown in the image below. You will observe that the amount 900 takes the first rank and 200 the lowest rank. Also, there is a gap between rank 2 and 4 because two values have the same rank. You can also infer that the most sales were on Friday and the least on Monday.

id	day	amount	amount_rank
5	Friday	900	1
3	Wednesday	600	2

id	day	amount	amount_rank
6	Saturday	600	2
4	Thursday	390	4
2	Tuesday	300	5
1	Monday	200	6

DENSE_RANK() function is similar to **RANK()** in that it assigns ranks to rows, but the difference is that **DENSE_RANK** does not leave a gap when there are two or more equivalent values. Let's illustrate it with the `Sales` table from above. The query is shown below.

```
SELECT
  id,
  day,
  amount,
  DENSE_RANK() OVER (ORDER BY amount DESC) AS amount_rank
FROM
  sales;
```

The result is shown below. As you will notice, there is no gap between the ranks like in the **RANK** function.

id	day	amount	amount_rank
5	Friday	900	1
3	Wednesday	600	2
6	Saturday	600	2
4	Thursday	390	3
2	Tuesday	300	4
1	Monday	200	5

ROW_NUMBER assigns a unique number to each row depending on the order you specify. It does not skip numbers; even though there are equivalent values, it assigns them different numbers, unlike **RANK** and **DENSE_RANK** functions that give them the same rank.

Let's use the same `Sales` table to illustrate. The query below shows how to use the **ROW_NUMBER** function.

```
SELECT
  id,
  day,
  amount,
  ROW_NUMBER() OVER (ORDER BY amount DESC) AS rowNumber
FROM
  sales;
```

The result is shown in the image below. You will notice that the `rownumber` column increases, and even though there are matching values, it just assigns a unique row number to each.

id	day	amount	amount_rank
5	Friday	900	1
3	Wednesday	600	2
6	Saturday	600	3

id	day	amount	amount_rank
4	Thursday	390	4
2	Tuesday	300	5
1	Monday	200	6

What is LAG() and LEAD() in SQL? Give an example use case

LAG() and **LEAD()** are window functions used to retrieve data from rows before and after a specified row. You can also refer to them as positional SQL functions.

LAG() allows you to access a value stored in rows before the current row. The row may be directly before or some rows before. Let's take a look at the syntax:

```
LAG(column_name, offset, default_value)
```

It takes three arguments.

- **column_name**: This specifies the column to fetch from the previous row.
- **offset**: This is an optional argument and specifies the number of rows behind to look at. The default is 1.
- **default_value**: This is the value to assign when no previous row exists. It is optional, and the default is NULL.

Using the `Sales` table, let's illustrate the **LAG()** function. The query is used to find the previous day sales. LAG() is useful when you want to create reports of past events.

id	day	amount
1	Monday	200
2	Tuesday	300
3	Wednesday	600
4	Thursday	390

id	day	amount
5	Friday	900
6	Saturday	600

```
SELECT
  id,
  day,
  amount,
  LAG(amount) OVER (ORDER BY id) AS previous_day_sales
FROM
  sales;
```

The result of the query looks like this:

id	day	amount	previous_day_sales
1	Monday	200	null
2	Tuesday	300	200

id	day	amount	previous_day_sales
3	Wednesday	600	300
4	Thursday	390	600
5	Friday	900	390
6	Saturday	600	900

You use the **LEAD()** function to get data from rows after the current row. Its syntax is similar to that of the **LAG()** function. You can use it for forecasting future trends by looking ahead.

The query using the **LEAD()** function is shown below.

```
SELECT
  id,
  day,
  amount,
  LEAD(amount) OVER (ORDER BY id) AS previous_day_sales
```

```
FROM
  sales;
```

id	day	amount	previous_day_sales
1	Monday	200	300
2	Tuesday	300	600
3	Wednesday	600	390
4	Thursday	390	900
5	Friday	900	600
6	Saturday	600	null

How will you detect gaps in a sequence of dates per user?

You will use the **LAG()** function to detect gaps in a sequence of dates per user. You will compare each date with the previous one and check if the

difference is greater than 1.

Let's use a table `ClockIns` to demonstrate how you detect gaps. The table has two columns, `userId` and `clockInDate`, representing the user identification number and the date the user clocked in with an access card into a facility. The table looks like this:

userId	clockInDate
1	2025-01-01
1	2025-01-02
1	2025-01-05
1	2025-01-06
2	2025-01-06
2	2025-01-06
2	2025-01-07

userId	clockInDate
3	2025-01-02
3	2025-01-04
3	2025-01-06
3	2025-01-07

To query to find gaps per user looks like this:

```
WITH clockInGaps AS (
  SELECT
    userid,
    clockInDate,
    LAG(clockInDate) OVER (PARTITION BY userId ORDER BY clockInDate
  FROM
    clockIns
)

SELECT
  userId,
```

```
  previousClockInDate AS gapStart,
  clockInDate AS gapEend,
  clockInDate - previousClockInDate - 1 AS gapDays
FROM clockInGaps
WHERE clockInDate - previousClockInDate > 1
ORDER BY userId, gapStart;
```

The code above starts with creating an expression `clockInGaps` that queries for each user and their `clockInDate` and uses the `LAG` function to get the previous date for each user. Then, the main query filters each row and finds the gaps between the current date and the previous date. The result of the query looks like this:

userId	gapStart	gapEnd	gapDays
1	2025-01-02	2025-01-05	2
2	2025-01-07	2025-01-10	2
3	2025-01-02	2025-01-04	1

userId	gapStart	gapEnd	gapDays
3	2025-01-04	2025-01-06	1

What does the NTILE() function do, and how might it be useful in analyzing data?

NTILE() is a window function that divides rows into a pre-defined number of roughly equal groups. It's like breaking your data into different sets based on your defined criteria. For example, let's say you have some student scores from 1 to 100; you can use the **NTILE()** function to categorize the scores into different groups or buckets.

The syntax of the `NTILE()` function is:

```
NTILE(n) OVER (ORDER BY some_column)
```

- n: represents the number of groups you want to divide your rows into.
- ORDER BY: defines the order of the rows in each group where the function is applied.

Let's see a practical example using a table `Scores`. The table stores students' scores on a test. We will see how to use the **NTILE()** function.

userId	score
1	78
2	70
3	90
4	98
5	60
6	88
7	100
8	66

The query using the **NTILE()** function looks like this:

```
SELECT
  id,
  score,
  NTILE(3) OVER (ORDER BY score DESC) AS category
FROM scores;
```

userId	score	category
7	100	1
4	98	1
3	90	1
6	88	2
1	78	2
2	70	2
8	66	3

userId	score	category
5	60	3

The **NTILE()** function is useful in data analysis because it can detect outliers in a data set and create histograms of data. It can also create percentiles and quartiles for data distribution.

Optimization And Pitfalls

How would you optimize slow-running queries?

To optimize slow-running queries, you need to analyze the query first to know what to optimize. You can perform different optimizations depending on the query. Some of the optimizations include:

- **Using indexes effectively**: Indexes speed up queries by enabling the database to find entries that fit specific criteria quickly. Indexing is the process of mapping the values of one or more columns to a unique value that makes it easy to search for rows that match a search criteria.

You can create indexes on columns used frequently in the **WHERE**, **JOIN**, and **ORDER BY** clauses. However, note that creating too many indexes can slow down inserts, updates, and deletions.

- **Avoid SELECT *** : Using the **SELECT** ***** statement can slow down your query performance because it returns all the columns in a table including the ones not needed for the query. You should select only the columns that you need for a query for optimal performance. So when you see a query that selects all columns, you should check if all the columns are really needed and used further down the query chain.

- **Avoid using subqueries**: Subqueries slow down query performance, especially when you use them in the `WHERE` or `HAVING` clauses. You should avoid using subqueries where possible and use JOINs or other techniques instead.

- **Utilize stored procedures**: Stored procedures are precompiled SQL statements stored in a database, and can be called from an application or directly from a query. Using stored procedures can improve your

query performance by reducing the amount of data that is sent between the database and your application, and also saves time required to compile the SQL statements.

Why should you avoid SELECT * in production code?

You should avoid using **SELECT *** as much as possible in your production code for the following reasons:

- **Increased IO**: Using **SELECT ***, you can return unnecessary data that leads to increased Input/Output cycles at the database level since you will be reading all the data in a table. This effect will be more impactful on a table with a lot of data and even slow down your query.
- **Increased network traffic**: **SELECT *** returns more data than required to the client, which uses more network bandwidth than needed. The increase in network bandwidth causes data to take longer to reach the client application and impacts the application's performance.

- **More application memory**: The return of a lot of data would make your application require more memory to hold the unnecessary data which might you might not use and this impacts application performance.

- **Makes maintenance more difficult**: Using **SELECT *** makes code maintenance more challenging. If the table structure changes by adding, removing, or renaming columns, the queries using **SELECT *** could break unexpectedly. You should explicitly specify the columns from which you want to fetch data to ensure resilience against potential changes in the database schema.

What is the impact of missing indexes?

Missing indexes can affect the performance of queries, especially when the data grows. The major impacts of missing indexes are listed below:

- **Slow queries**: Without indexes, every read query will go through the whole table to find matching rows. This will get worse as the data in the table grows.

- **Locking and concurrency issues**: Scanning a table without indexes takes longer, and locking the table can prevent other queries from running, affecting application performance.
- **Inefficient joins**: Joins on tables without indexes on the join keys are extremely slow and result in bad query performance.
- **Poor user experience**: Missing indexes can lead to poor user experience in your applications. It can result to slower page loads, application hanging when data is being fetched from the database.

What is a SARGable query?

SARGable stands for Search Argumentable query, which uses indexes and leads to efficient queries. If a query is SARGable, the database engine quickly locates rows using indexes, avoids scanning the whole table, and improves query performance.

What are some common mistakes when using GROUP BY?

The common mistakes people encounter when using the **GROUP BY** clause include:

- **Selecting non-aggregated columns not in the GROUP BY clause:** This is a common mistake made my beginners and experts. An example query of this looks like this:

```
SELECT day, amount FROM Sales
GROUP BY day
```

In the query above, the `amount` column is not part of the `GROUP BY` clause and will throw an error that it must appear in the `GROUP BY` clause. To fix this, you should add an aggregate function to the amount column.

```
SELECT day, MAX(amount) FROM Sales
GROUP BY day
```

- **Not using aggregate functions:** It is also a common mistake to use `GROUP BY` without aggregate functions. `GROUP BY` usually goes with aggregate functions like `MAX`, `MIN`, `COUNT`, etc.

- **Grouping by multiple columns**: Grouping by multiple columns can make the query meaningless. It is not common to group by many columns, and when this happens, you should check if you really need to group by those columns.

Why can NOT IN lead to unexpected results with NULLS?

Since NULL is unknown, a `NOT IN` query containing a NULL or NULL in the list of possible values will always return 0 records because of the unknown result introduced by the NULL value. SQL cannot determine for sure whether the value is not in that list.

Let's illustrate this using a table `Sales` that looks like this:

id	day	amount
1	Monday	200
2	Tuesday	300
3	Wednesday	600

id	day	amount
4	Thursday	390
5	Friday	900
6	Saturday	600

If you run the query below, it will return an empty result because SQL cannot determine if the value is not in the list because nothing equals or doesn't equal NULL.

```
SELECT * from sales
WHERE amount NOT IN (200, 300, 600, NULL);
```

Wrapping up

Mastering SQL queries is essential for anyone working with databases, whether you're a beginner just starting out or an experienced developer looking to sharpen your skills. The 30 questions covered in this guide span

from foundational concepts like JOINs and WHERE clauses to advanced topics such as window functions and query optimization.

Remember that SQL proficiency comes with practice. Take time to implement these queries in your own database environment, experiment with different scenarios, and understand how each function behaves with various data sets. The more you practice, the more confident you'll become in handling complex database challenges during interviews and in real-world applications.

Keep this guide handy as a reference, and don't hesitate to revisit the concepts that challenge you most. Good luck with your SQL journey and upcoming interviews!

Roadmaps Best Practices Guides Videos FAQs YouTube

roadmap.sh by @kamrify

THENEWSTACK