**SQL**

Upon hearing the term “data scientist,” buzzwords such as predictive analytics, big data, and deep learning may leap to mind. So, let’s not beat around the bush: data wrangling isn’t the most fun or sexy part of being a data scientist. However, as a data scientist, you will likely spend a great deal of your time working writing SQL queries to retrieve and analyze data. As such, almost every company you interview with will test your ability to write SQL queries. These questions are practically guaranteed if you are interviewing for a data scientist role on a product or analytics team, or if you're after a data science-adjacent role like data analyst or business intelligence analyst. Sometimes, data science interviews may go beyond just writing SQL queries, and cover the basic principles of database design and other big data systems. This focus on data architecture is particularly true at early-stage startups, where data scientists often take an active role in data engineering and data infrastructure development.

**SQL**  
**How SQL Interview Questions Are Asked**

Because most analytics workflows require quick slicing and dicing of data in SQL, interviewers will often present you with hypothetical database tables and a business problem, and then ask you to write SQL on the spot to get to an answer. This is an especially common early interview question.

Conducted via a shared coding environment or through an automated remote assessment tool. Because of the many different flavors of SQL used by industry, these questions aren't usually testing your knowledge of database-specific syntax or obscure commands. Instead, interviews are designed to test your ability to translate reporting requirements into SQL.

For example, at a company like Facebook, you might be given a table on user analytics and asked to calculate the month-to-month retention. Here, it’s relatively straightforward what the query should be, and you're expected to write it. Some companies might make their SQL interview problems more open-ended. For example, Amazon might give you tables about products and purchases and then ask you to list the most popular products in each category. Robinhood may give you a table and ask why users are churning. Here, the tricky part might not be just writing the SQL query, but also figuring out collaboratively with the interviewer what “popular products” or “user churn” means in the first place.

Finally, some companies might ask you about the performance of your SQL query. While these interview questions are rare, and they don’t expect you to be a query optimization expert, knowing how to structure a database for performance, and avoid slow-running queries, can be helpful. This knowledge can come in handy as well when you are asked more conceptual questions about database design and SQL.

**Tips for Solving SQL Interview Questions**

First off, don’t jump into SQL questions without fully understanding the problem. Before you start whiteboarding or typing out a solution, it’s crucial to repeat back the problem so you can be sure you've understood it correctly. Next, try to work backwards, especially if the answer needs multiple joins, subqueries, and common table expressions (CTEs). Don’t overwhelm yourself trying to figure out the multiple parts of the final query at the same time. Instead, imagine you had all the information you needed in a single table, so that your query was just a single SELECT statement. Working backwards slowly from this ideal table, one SQL statement at a time, try to end up with the tables you originally started with.

For more general problem-solving tips, be sure to also read the programming interview tips in the coding chapter. Most of what applies to solving coding questions — like showing your work and asking for help if stuck — applies to solving SQL interview questions too.

**Basic SQL Commands**

Before we cover the must-know SQL commands, a quick note — please don’t be alarmed by minor variations in syntax between your favorite query language and our PostgreSQL snippets:

* **CREATE TABLE**: Creates a table in a relational database and, depending on what database you use (e.g., MySQL), can also be used to define the table's schema.
* **INSERT**: Inserts a row (or a set of rows) into a given table.
* **UPDATE**: Modifies already-existing data.
* **DELETE**: Removes a row (or a group of rows) from a database.
* **SELECT**: Selects certain columns from a table. A common part of most queries.
* **GROUP BY**: Groups/aggregates rows having the contents of a specific column or set of columns.
* **WHERE**: Provides a condition on which to filter before any grouping is applied.
* **HAVING**: Provides a condition on which to filter after any grouping is applied.
* **ORDER BY: Sorts results in ascending or descending order according to the contents of a specific column or set of columns.**
* **DISTINCT: Returns only distinct values.**
* **UNION: Combines results from multiple SELECT statements.**

**Joins**

**Imagine you worked at Reddit and had two separate tables: users and posts.**

**Reddit Users Table**

| **Column Name** | **Type** |
| --- | --- |
| **user\_id** | **integer** |
| **country** | **string** |
| **active\_status** | **boolean** |
| **join\_time** | **datetime** |
|  |  |

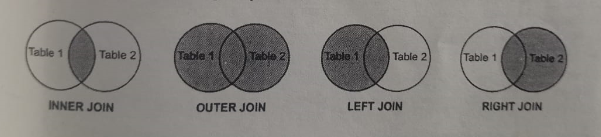
**Reddit Posts Table**

| **Column Name** | **Type** |
| --- | --- |
| **post\_id** | **integer** |
| **user\_id** | **integer** |
| **subreddit\_id** | **integer** |
| **title** | **string** |
| **body** | **string** |
| **active\_status** | **boolean** |
| **post\_time** | **datetime** |
|  |  |

**Joins are used to combine rows from multiple tables based on a common column. As you can see, the user\_id column is the common column between the two tables and links them together; hence it is known as a join key.**

**There are four main types of joins:**

* **Inner Join**
* **Outer Join**
* **Left Join**
* **Right Join**

****

**Inner Join**

**Inner joins combine multiple tables and will preserve the rows where column values match in the tables being combined. The word INNER is optional and is rarely used explicitly because it is the default type of join.**

**Example**

**To find the number of Reddit users who have made a post:**

**SELECT**

**COUNT(DISTINCT u.user\_id)**

**FROM**

**users u**

**JOIN**

**posts p**

**ON**

**u.user\_id = p.user\_id;**

**A self join is a special case of an inner join where a table joins itself. The most common use case for a self join is to look at pairs of rows within the same table.**

**OUTER JOIN**

**Outer joins combine multiple tables by matching on the columns provided, while preserving all rows. As an example of an outer join, we list all inactive users with posts and all inactive posts from any user:**

**SELECT \***

**FROM users u**

**OUTER JOIN posts p ON u.user\_id = p.user\_id**

**WHERE**

**u.active\_status = False**

**OR p.active\_status = False;**

**LEFT JOIN**

**Left joins combine multiple tables by matching on the column names provided, while preserving all the rows from the first table of the join. As an example, we use a left join to find the percentage of users that made a post:**

**SELECT**

**COUNT(**

**DISTINCT CASE**

**WHEN p.post\_id IS NOT NULL THEN u.user\_id**

**END**

**) / COUNT(\*) AS pct\_users**

**FROM**

**users u**

**LEFT JOIN**

**posts p ON u.user\_id = p.user\_id;**

**RIGHT JOIN**

**Right joins combine multiple tables by matching on the column names provided, while preserving all the rows from the second table of the join. For example, we use the right join to find the percentage of posts made where the user is located in the U.S.:**

**SELECT**

**COUNT(**

**DISTINCT CASE**

**WHEN u.country = 'US' THEN p.post\_id**

**END**

**) / COUNT(\*) AS pct\_posts**

**FROM**

**users u**

**RIGHT JOIN**

**posts p ON u.user\_id = p.user\_id;**

**Join Performance**

**Joins are an expensive operation to process and are often bottlenecks in query runtimes. As such, to write efficient SQL, you want to be working with the fewest rows and columns before joining tables together.**

**Some general tips to improve join performance include:**

* **Select specific fields instead of using SELECT \*.**
* **Use LIMIT in your queries.**
* **Filter and aggregate data before joining.**
* **Avoid multiple joins in a single query.**

**Advanced SQL Commands**

**Aggregation**

**For interviews, you need to know how to use the most common aggregation functions like COUNT, SUM, AVG, or MAX.**

**Example:**

**SELECT COUNT(\*)**

**FROM users;**

**Filtering**

**SQL contains various ways to compare rows, the most common of which use:**

* **= (equals)**
* **<> (not equals)**
* **> (greater than)**
* **< (less than)**

**Additionally, logical and filtering clauses such as OR and AND can be used. For example, below is a query to filter active Reddit users from outside the U.S.:**

**SELECT \***

**FROM users**

**WHERE**

**active\_status = True**

**AND country <> 'US';**

**Common Table Expressions and Subqueries**

**Common Table Expressions (CTEs) define a query and then allow it to be referenced later using an alias. They provide a handy way of breaking up large queries into more manageable subsets of data.**

**Example:**

**Below is a CTE that gets the number of posts made by each user, which is then used to calculate the distribution of posts made by users (e.g., 100 users posted 5 times, 80 users posted 6 times, and so on):**

**WITH user\_post\_count AS (**

**SELECT**

**users.user\_id,**

**COUNT(post\_id) AS num\_posts**

**FROM**

**users**

**LEFT JOIN**

**posts ON users.user\_id = posts.user\_id**

**GROUP BY**

**users.user\_id**

**)**

**SELECT**

**num\_posts,**

**COUNT(\*) AS num\_users**

**FROM**

**user\_post\_count**

**GROUP BY**

**1**

**Subqueries serve a similar function to CTEs, but are inline in the query itself and must have a unique alias for the given scope.**

**SELECT**

**num\_posts,**

**COUNT(\*) AS num\_users**

**FROM**

**(**

**SELECT**

**users.user\_id,**

**COUNT(post\_id) AS num\_posts**

**FROM**

**users**

**LEFT JOIN posts ON users.user\_id = posts.user\_id**

**GROUP BY**

**1**

**) u;**

**CTEs and subqueries are mostly similar, with the exception that CTEs can be used recursively. Both concepts are incredibly important to know and practice, since most of the harder SQL interview questions essentially boil down to breaking the problem into smaller chunks of CTEs and subqueries.**

**Window Functions**

**Window functions perform calculations across a set of rows, much like aggregation functions, but do not group those rows as aggregation functions do. Therefore, rows retain their separate identities even with aggregated columns. Thus, window functions are particularly convenient when we want to use both aggregated and non-aggregated values at once. Additionally, the code is often easier to manage than the alternative: using group by statements and then performing joins on the original input table.**

**Syntax-wise, window functions require the OVER clause to specify a particular window. This window has three components:**

* **Partition Specification: separates rows into different partitions, analogous to how GROUP BY operates. This specification is denoted by the clause PARTITION BY.**
* **Ordering Specification: determines the order in which rows are processed, given by the clause ORDER BY.**
* **Window Frame Size Specification: determines which sliding window of rows should be processed for any given row. The window frame defaults to all rows within a partition but can be specified by the clause ROWS BETWEEN (start, end).**

**For instance, below we use a window function to sum up the total Reddit posts per user, and then add each post\_count to each row of the users table:**

**SELECT**

**\*,**

**SUM(posts) OVER (PARTITION BY user\_id) AS post\_count**

**FROM**

**users u**

**LEFT JOIN**

**posts p ON u.user\_id = p.user\_id;**

**Note: A comparable version without using window functions looks like the following:**

**sql**

**CopyEdit**

**SELECT**

**a.\*,**

**b.post\_count**

**FROM**

**users a**

**JOIN (**

**SELECT**

**user\_id,**

**SUM(posts) AS post\_count**

**FROM**

**users u**

**LEFT JOIN**

**posts p ON u.user\_id = p.user\_id**

**GROUP BY**

**1**

**) b ON a.user\_id = b.user\_id;**

**Here is the exact text extracted from the image:**

**As you can see, window functions tend to lead to simpler and more expressive SQL.**

**LAG and LEAD**

**Two popular window functions are (LAG) and (LEAD). These are both positional window functions, meaning they allow you to refer to rows after the current row (LAG), or rows before the current row (LEAD). The below example uses LAG so that for every post, it finds the time difference between the post at hand, and the post made right before it in the same subreddit:**

**SELECT**

**p.\*,**

**LAG(post\_time, 1) OVER (**

**PARTITION BY user\_id,**

**subreddit\_id**

**ORDER BY post\_time ASC**

**) AS prev\_subreddit\_post\_time**

**FROM**

**posts p;**

**RANK**

**Say that for each user, we wanted to rank posts by their length. We can use the window function RANK() to rank the posts by length for each user.**

**SELECT**

**\*,**

**RANK() OVER (**

**PARTITION BY user\_id**

**ORDER BY LENGTH(body) DESC**

**) AS rank**

**FROM**

**users u**

**LEFT JOIN posts p ON u.user\_id = p.user\_id;**