SQL & DB Design

CHAPTER 8

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.

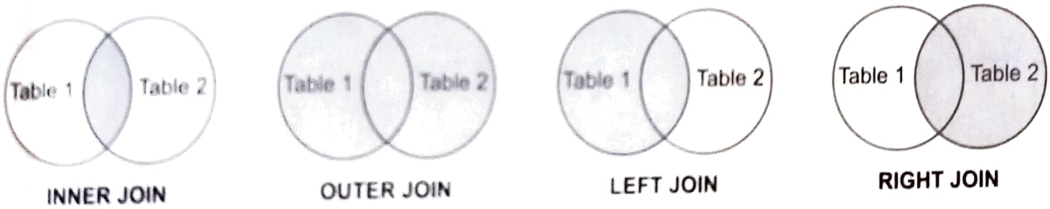
Reddit users

Aa column_name	type
user_id	integer
country	string
active_status	boolean
join_time	datetime

Reddit Posts

Aa 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; hence it is known as a join key. There are four main types of joins



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 because it's the default type of join. As an example, we use an inner join 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 two tables together. Some general tips to improve join performance include the following:

- 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:

```
SELECT COUNT(*) FROM users ...
```

Filtering

SQL contains various ways to compare rows, the most common of which use = and <> (not equal), >, and <, along with regex and other types of logical and filtering clauses such as OR and AND. For example, below we filter to 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. For example, below is a CTE which gets the number of posts made by each user, which is then used to get the distribution of posts made by users (i.e., 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
        1
)

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 that a comparable version without using window functions looks like the following:

```
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
```

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
```

Databases and Systems

Although knowing all of the database's inner workings isn't strictly necessary, having a high-level understanding of basic database and system design concepts is very helpful. Database interview questions typically do not involve minutiae about specific databases but, instead, focus on how databases generally operate and what trade-offs are made during schema design. For example, you might be asked how you'd set up tables to represent a real-world situation, like storing data if you worked at Reddit. You'd need to define the core tables (users, posts, subreddits) and then define the relationships between. For the Reddit example, posts would have a user_id column for the corresponding user that made the post.

You also may be asked to choose which columns should be indexed, which allows for more rapid lookup of data. For the Reddit example, you would want to index the user_id column, since it's likely heavily used as a join key across many important queries.

While data science interviews don't go into system design concepts as deeply or as often as software engineering and data engineering interviews, it can still show up from time to time. This is particularly the case if you are joining a smaller company, where your data science job might involve creating and

managing data pipelines. Besides generic questions about scaling up data infrastructure, you might be asked conceptual questions about popular large-scale processing frameworks (Hadoop, Spark) or orchestration frameworks (Airflow, Luigi) — especially if you happen to list these technologies on your resume.

Keys & Normalization

Primary keys ensure that each entity has its own unique identifier, i.e., no rows in a table are duplicated with respect their primary key. *Foreign keys*, on the other hand, establish mappings between entities. By using a foreign key to link two related tables, we ensure that data is only stored once in the database. For the Reddit example, in the posts table, the post_id column is the primary key, and each post has a user_id which is a foreign key to the users table.

Item	Primary Key	Foreign Key
Consists of One or More Columns	Yes	Yes
Duplicate Values Allowed	No	Yes
NULLs Allowed	No	Yes
Uniquely Identify Rows in a Table	Yes	Maybe
Number Allowed Per Table	One	Zero or More
Indexed	Automatically Indexed	No Index Automatically created

Keys allow us to split data efficiently into separate tables, but still enforce a logical relationship between two tables, rather than having everything duplicated into one table. This process of generally separating out data to prevent redundancy is called normalization. Along with reducing redundancy, normalization helps you enforce database constraints and dependencies, which improves data integrity.

The disadvantage to normalization is that now we need an expensive join operation between the two related tables. As such, in high-performance systems, denormalization is an optimization technique where we keep redundant data to prevent expensive join operations. This speeds up read times, but at the cost of having to duplicate data. At scale, this can be acceptable since storage is cheap, but compute is expensive.

When normalization comes up in interviews, it often concerns the conceptual setup of database tables: why a certain entity should have a foreign key to another entity, what the mapping relationship is between two types of records (one-to-one, one-to-many, or many-to-many), and when it might be advantageous to denormalize a database.

Properties of Distributed Databases

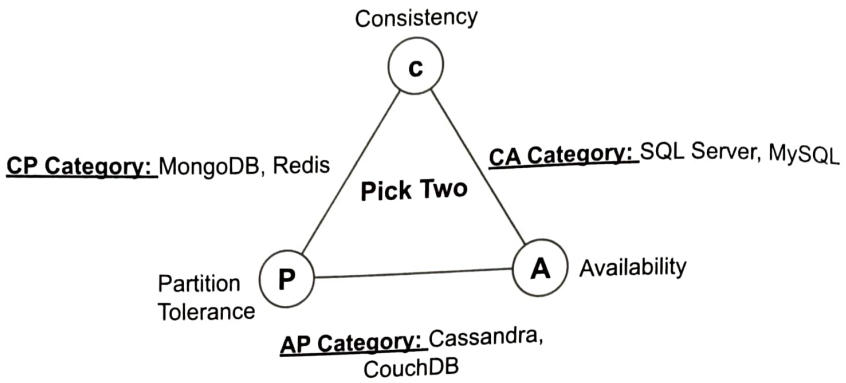
Two concepts, the CAP theorem and the ACID framework, are commonly used to assess theoretical guarantees of databases and are discussed in detail below.

The CAP theorem provides a framework for assessing properties of a distributed database, although only two of the theorem's three specifications can be met simultaneously. The name CAP is an acronym based on the following desirable characteristics of distributed databases:

Consistency: All clients using the database see the same data.

Availability: The system is always available, and each request receives a non-error response, but there's no guarantee that the response contains the latest data.

Partition tolerance: The system functions even if communication between nodes is lost or delayed.



Although the CAP theorem is a theoretical framework, one should consider the real-life trade-offs that need to be made based on the needs of the business and those of the database's users. For example, the Instagram feed focuses on availability and less so on consistency, since what matters is that you get a result instantly when visiting the feed. The penalty for inconsistent results isn't high. It's not going to crush users to see @ChampagnePapi's last post has 57,486 likes (instead of the correct 57,598 likes). In contrast, when designing the service to handle payments on Whatsapp, you'd favor consistency over availability, because you'd want all servers to have a consistent view of how much money a user has to prevent people from sending money they didn't have. The downside is that sometimes sending money takes a minute or a payment fails and you are asked to re-try. Both are reasonable trade-offs in order to prevent double-spend issues.

The second principle for measuring the correctness and completeness of a database transaction is called *the ACID framework*. ACID is an acronym derived from the following desirable characteristics:

- **Atomicity:** an entire transaction occurs as a whole or it does not occur at all (i.e., no partial transactions are allowed). If a transaction aborts before completing, the database does a "rollback" on all such incomplete transactions. This prevents partial updates to a database, which cause data integrity issues.
- **Consistency:** integrity constraints ensure that the database is consistent before and after a given transaction is completed. Appropriate checks handle any referential integrity for primary and foreign keys.
- **Isolation:** transactions occur in isolation so that multiple transactions can occur independently without interference. This characteristic properly maintains concurrency.
- **Durability:** once a transaction is completed, the database is properly updated with the data associated with that transaction, so that even a system failure could not remove that data from it.

The ACID properties are particularly important for online transactional processing (OLTP) systems, where databases handle large volumes of transactions conducted by many users in real time.

Scaling Databases

Traditionally, database scaling was done by using full-copy clusters where multiple database servers (each referred to as a node within the cluster) contained a full copy of the data, and a load balancer would roundrobin incoming requests. Since each database server had a full copy of the data, each node experienced the issues mentioned in the CAP theorem discussed above (especially during high-load periods). With the advent of the cloud, the approach towards scaling databases has evolved rapidly.

Nowadays, the cloud makes two main strategies to scaling feasible: vertical and horizontal scaling.

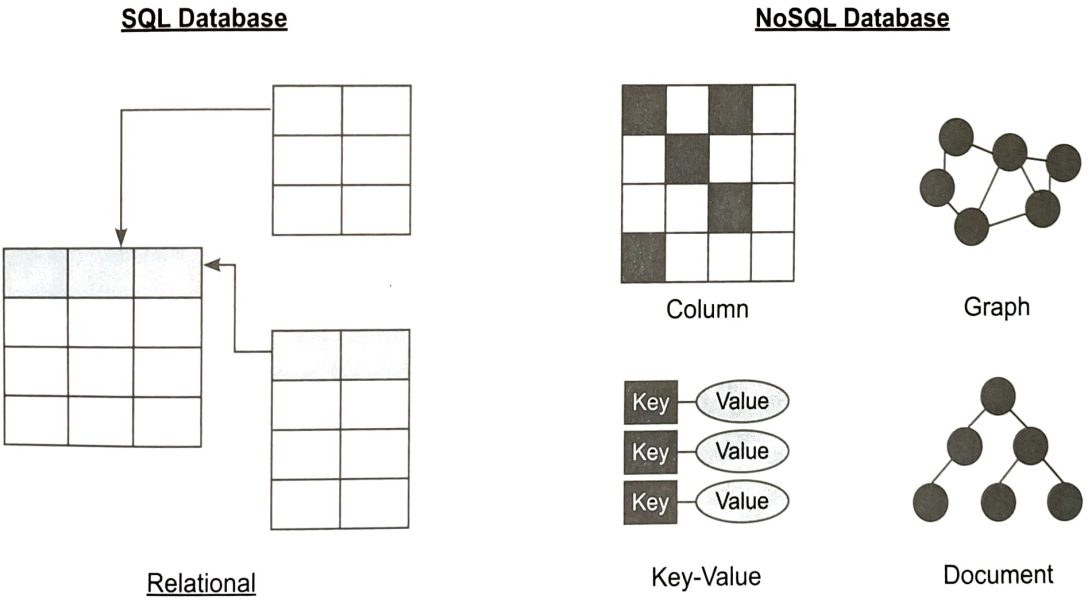
Vertical scaling, also known as scaling up, involves adding CPU and RAM to existing machines. This approach is easy to administer and does not require changing the way a system is architected. However, vertical scaling can quickly become prohibitively expensive, eventually limiting the scope for upgrades. This is because certain machines may be close to their physical limits, making it practically impossible to replace them with more performant servers.

In horizontal scaling, also known as scaling out, more commodity machines (nodes) are added to the resource pool. In comparison to vertical scaling, horizontal scaling has a much cheaper cost structure and has better fault tolerance than vertical scaling. However, as expected, there are trade-offs with this approach. With many more nodes, you have to deal with issues that arise in any distributed system, like handling data consistency between nodes. Therefore, horizontal scaling offers a greater set of challenges in infrastructure management compared to vertical scaling.

Sharding, in which database rows themselves are split across nodes in a cluster, is a common example of horizontal scaling. For all tables, each node has the same schema and columns as the original table, but the data are stored independently of other shards. To split the rows of data, a sharding mechanism determines which node (*shard*) that data for a given key should exist on. This sharding mechanism can be a hash function, a range, or a lookup table. The same operations apply for reading data as well, and so, in this way, each row of data is uniquely mapped to one particular shard.

Relational Databases vs. NoSQL Databases

Relational databases, like MySQL and Postgres, have a table-based structure with a fixed, pre-defined schema. In contrast, NoSQL databases (named because they are "non-SQL" and "non-relational") store data in a variety of forms rather than in a strict table-based structure.



One type of NoSQL database is the document database. MongoDB, the most popular document database, associates each record with a document. The document allows for arbitrarily complex, nested, and varied schemas inside it. This flexibility allows for new fields to be trivially added compared to a relational database, which has to adhere to a pre-defined schema.

Another type of NoSQL database is the graph database. Neo4J is a well-known graph database, which stores each data record along with direct pointers to all the other data records it is connected to.

By making the relationships between the data as important as storing the data itself, graph databases allow for a more natural representation of nodes and edges, when compared to relational databases.

BASE Consistency Model

Analogous to the ACID consistency model for relational databases, the BASE model applies to NoSQL databases:

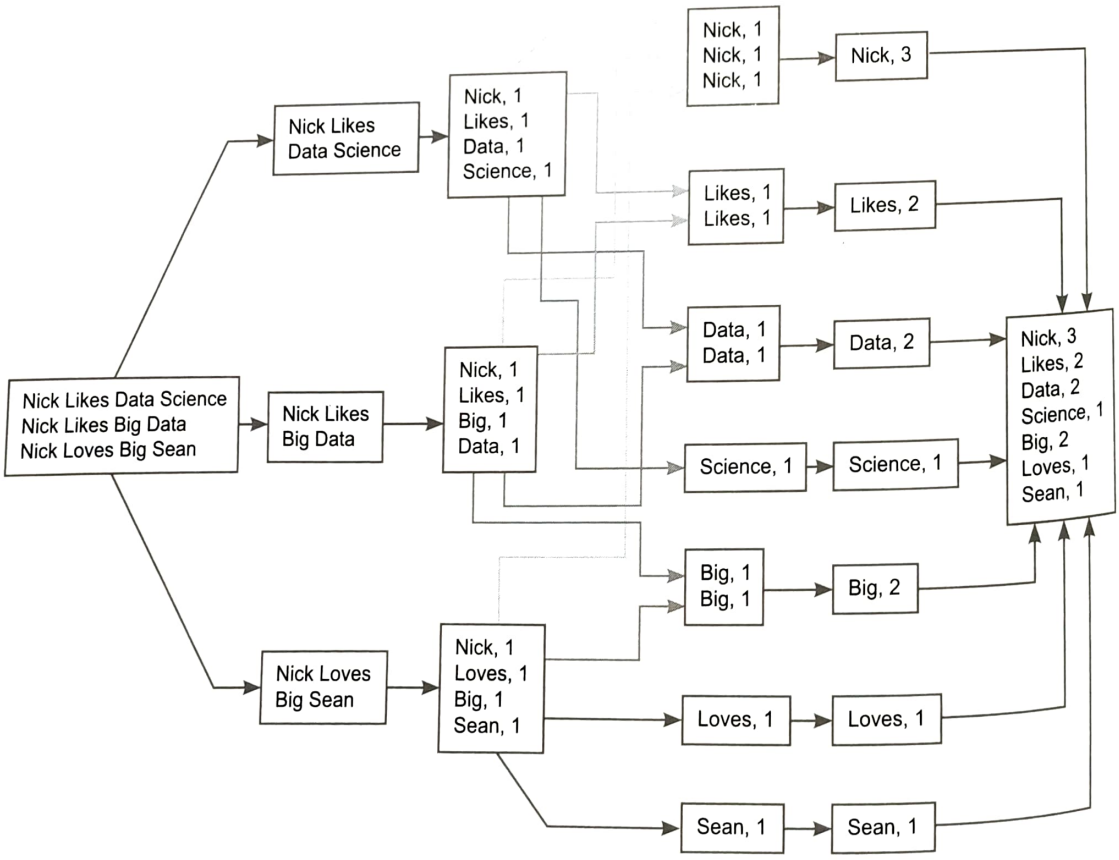
- **Basically Available:** data is guaranteed to be available; there will be a response to any request. This occurs due to the highly distributed approach of NoSQL databases. However, the requested data may be inconsistent and inaccurate.
- **Soft State:** system's state may change over time, even without input. These passive changes occur due to the eventual consistency property.
- **Eventual Consistency:** data will eventually converge to a consistent state, although no guarantees are made on when that will occur.

If you compare and contrast ACID and BASE, you will see that the BASE model puts a stronger focus on availability and scalability but less of an emphasis on data correctness.

MapReduce

MapReduce is a popular data processing framework that allows for the concurrent processing of large volumes of data. MapReduce involves four main steps:

The overall MapReduce word count process



1) **Split step:** splits up the input data and distributes it across different nodes
2) **Map step:** takes the input data and outputs <key, value> pairs
3) **Shuffle step:** moves all the <key, value> pairs with the same key to the same node
4) **Reduce step:** processes the <key, value> pairs and aggregates them into a final output

The secret sauce behind MapReduce's efficiency is the shuffle step; by grouping related data onto the same node, we can take advantage of the locality of data. Said another way, by shuffling the related <key, value> pairs needed by the reduce step to the same node rather than sending them to a different node for reducing, we minimize node-to-node communication, which is often the bottleneck for distributed computing.

For a concrete example of how MapReduce works, assume you want to count the frequency of words in a multi-petabyte corpus of text data. The MapReduce steps are visualized on left:

Here's how each MapReduce step operates in more detail:

1. Split step: We split the large corpus of text into smaller chunks and distribute the pieces to different machines.
2. Map step: Each worker node applies a specific mapping function to the input data and writes the output <key, value> pairs to a memory buffer. In this case, our mapping function simply converts each word into a tuple of the word and its frequency (which is always 1). For example, say we had the phrase "hello world" on a single machine. The map step would convert that input into two key value pairs: <"hello", 1> and <"world", 1>. We do this for the entire corpus, so that if our corpus is words big, we end up with key-value pairs in the map step.
3. Shuffle step: Data is redistributed based on the output keys from the prior step's map function, such that tuples with the same key are located on the same worker node. In this case, it means that all tuples of <"hello", 1> will be located on the same worker node, as will all tuples of <"world", 1>, and so on.
4. Reduce step: Each worker node processes each key in parallel using a specified reducer operation to obtain the required output result. In this case, we just sum up the tuple counts for each key, so if there are 5 tuples for <"hello", 1> then the final output will be <"hello", 5>, meaning that the word "hello" occurred 5 times.

Because the shuffle step moved all the "hello" key-value pairs to the same node, the reducer can operate locally and, hence, efficiently. The reducer doesn't need to communicate with other nodes to ask for their "hello" key-value pairs, which minimizes the amount of precious node-to-node bandwidth consumed.

In practice, since MapReduce is just the processing technique, people rely on Hadoop to manage the steps of the MapReduce algorithm. Hadoop involves:

1) **Hadoop File System (HDFS):** manages data storage, backup, and replication
2) **MapReduce:** as discussed above
3) **YARN:** a resource manager which manages job scheduling and worker node orchestration

Spark is another popular open-source tool that provides batch processing similar to Hadoop, with a focus speed and reduced disk operations. Unlike Hadoop, it uses RAM for computation, enabling faster inmemory performance but higher running costs. Additionally, unlike Hadoop, Spark has built-in resource scheduling and monitoring, whereas MapReduce relies on external resource managers like YARN.

SQL & Database Design Questions

Easy Problems

8.1. Facebook: Assume you have the below events table on app analytics. Write a query to get the clickthrough rate per app in 2019.

events

Aa column_name	type
app_id	integer
event_id	string ("impression", "click")
timestamp	datetime

8.2. Robinhood: Assume you are given the tables below containing information on trades and users. Write a query to list the top three cities that had the most number of completed orders.

trades

Aa column_name	type
order_id	integer
user_id	integer
price	float
quantity	integer
status	string ("complete", "cancelled")
timestamp	datetime

users

Aa column_name	type
user_id	integer
city	string
email	string
signup_date	datetime

8.3. New York Times: Assume that you are given the table below containing information on viewership by device type (where the three types are laptop, tablet, and phone). Define "mobile" as the sum of tablet and phone viewership numbers. Write a query to compare the viewership on laptops versus mobile devices.

viewership

Aa column_name	type
user_id	integer
device_type	string
view_time	datetime

8.4. Amazon: Assume you are given the table below for spending activity by product type. Write a query to calculate the cumulative spend so far by date for each product over time in chronological order.

total_trans

Aa column_name	type
order_id	integer
user_id	integer
product_id	string
spend	float
trans_date	datetime

8.5. eBay: Assume that you are given the table below containing information on various orders made by customers. Write a query to obtain the names of the ten customers who have ordered the highest number of products among those customers who have spent at least $1000 total.

user_transactions

Aa column_name	type
transaction_id	integer
product_id	integer
user_id	integer
spend	float
trans_date	datetime

8.6. Twitter: Assume you are given the table below containing information on tweets. Write a query to obtain a histogram of tweets posted per user in 2020.

tweets

Aa column_name	type
tweet_id	integer
user_id	integer
msg	string
tweet_date	datetime

8.7. Stitch Fix: Assume you are given the table below containing information on user purchases. Write a query to obtain the number of people who purchased at least one or more of the same product on multiple days.

purchases

Aa column_name	type
purchase_id	integer
user_id	integer
product_id	integer
quantity	integer
price	float
purchase_time	datetime

8.8. Linkedin: Assume you are given the table below that shows the job postings for all companies on the platform. Write a query to get the total number of companies that have posted duplicate job listings (two jobs at the same company with the same title and description).

job_listings

Aa column_name	type
job_id	integer
company_id	integer
title	string
description	string
post_date	datetime

8.9. Etsy: Assume you are given the table below on user transactions. Write a query to obtain the list of customers whose first transaction was valued at $50 or more.

user_transactions

Aa column_name	type
transaction_id	integer
product_id	integer
user_id	integer
spend	float
transaction_date	datetime

8.10. Twitter: Assume you are given the table below containing information on each user's tweets over a period of time. Calculate the 7-day rolling average of tweets by each user for every date.

tweets

Aa column_name	type
tweet_id	integer
msg	string
user_id	integer
tweet_date	datetime

8.11. Uber: Assume you are given the table below on transactions made by users. Write a query to obtain the third transaction of every user.

transactions

Aa column_name	type
user_id	integer
spend	float
transaction_date	datetime

8.12. Amazon: Assume you are given the table below containing information on customer spend on products belonging to various categories. Identify the top three highest-grossing items within each category in 2020.

product_spend

Aa column_name	type
transaction_id	integer
category_id	integer
product_id	integer
user_id	integer
spend	float
transaction_date	datetime

8.13. Walmart: Assume you are given the below table on transactions from users. Bucketing users based on their latest transaction date, write a query to obtain the number of users who made a purchase and the total number of products bought for each transaction date.

user_transactions

Aa column_name	type
transaction_id	integer
product_id	integer
user_id	integer
spend	float
transaction_date	datetime

8.14. Facebook: What is a database view? What are some advantages views have over tables?

8.15. Expedia: Say you have a database system where most of the queries made were UPDATEs/INSERTs/DELETEs. How would this affect your decision to create indices? What if the queries made were mostly SELECTs and JOINs instead?

8.16. Microsoft: What is a primary key? What characteristics does a good primary key have?

8.17. Amazon: Describe some advantages and disadvantages of relational databases vs. NoSQL databases.

8.18. Capital One: Say you want to set up a MapReduce job to implement a shuffle operator, whose input is a dataset and whose output is a randomly ordered version of that same dataset. At a high level, describe the steps in the shuffle operator's algorithm.

8.19. Amazon: Name one major similarity and difference between a WHERE clause and a HAVING clause in SQL.

8.20. KPMG: Describe what a foreign key is and how it relates to a primary key.

8.21. Microsoft: Describe what a clustered index and a non-clustered index are. Compare and contrast the two.

Medium Problems

8.22. Twitter: Assume you are given the two tables below containing information on the topics that each Twitter user follows and the ranks of each of these topics. Write a query to obtain all existing users on 2021-01-01 that did not follow any topic in the 100 most popular topics for that day.

user_topics

column_name	type
user_id	integer
topic_id	integer
follow_date	datetime

topic_rankings

column_name	type
topic_id	integer
ranking	integer
ranking_date	datetime

8.23. Facebook: Assume you have the tables below containing information on user actions. Write a query to obtain active user retention by month. An active user is defined as someone who took an action (sign-in, like, or comment) in the current month.

user_actions

column_name	type
user_id	integer
event_id	string ("sign-in", "like", "comment")
timestamp	datetime

8.24. Twitter: Assume you are given the tables below containing information on user session activity. Write a query that ranks users according to their total session durations for each session type between the start date (2021-01-01) and the end date (2021-02-01).

sessions

column_name	type
session_id	integer
user_id	integer
session_type	string
duration	integer
start_time	datetime

8.25. Snapchat: Assume you are given the tables below containing information on Snapchat users and their time spent sending and opening snaps. Write a query to obtain a breakdown of the time spent sending vs. opening snaps (as a percentage of total time spent) for each of the different age groups.

activities

column_name	type
activity_id	integer

age_breakdown

column_name	type
user_id	integer

user_id	integer
type	string ('send', 'open')
time_spent	float
activity_date	datetime

age_bucket	string

8.26. Pinterest: Assume you are given the table below containing information on user sessions, including their start and end times. A session is considered to be concurrent with another user's session if they overlap. Write a query to obtain the user session that is concurrent with the largest number of other user sessions.

sessions

Aa column_name	type
session_id	integer
start_time	datetime
end_time	datetime

8.27. Yelp: Assume you are given the table below containing information on user reviews. Define a top-rated business as one whose reviews contain only 4 or 5 stars. Write a query to obtain the number and percentage of businesses that are top rated.

reviews

Aa column_name	type
business_id	integer
user_id	integer
review_text	string
review_stars	integer
review_date	datetime

8.28. Google: Assume you are given the table below containing measurement values obtained from a sensor over several days. Measurements are taken several times within a given day. Write a query to obtain the sum of the odd-numbered measurements and the sum of the even-numbered measurements by date.

measurements

Aa column_name	type
measurement_id	integer
measurement_value	float
measurement_time	datetime

8.29. Etsy: Assume you are given the two tables below containing information on user signups and user purchases. Of the users who joined within the past week, write a query to obtain the percentage of users that also purchased at least one item.

signups

Aa column_name	type
user_id	integer
signup_date	datetime

user_purchases

Aa column_name	type
user_id	integer
product_id	integer
purchase_amount	float
purchase_date	datetime

8.30. Walmart: Assume you are given the following tables on customer transactions and products. Find the top 10 products that are most frequently bought together (purchased in the same transaction).

transactions

Aa column_name	type
transaction_id	integer
product_id	integer
user_id	integer
quantity	integer
transaction_time	datetime

products

Aa column_name	type
product_id	integer
product_name	string
price	float

8.31. Facebook: Assume you have the table given below containing information on user logins. Write a query to obtain the number of reactivated users (i.e., those who didn't log in the previous month, who then logged in during the current month).

user_logins

Aa column_name	type
user_id	integer
login_date	datetime

8.32. Wayfair: Assume you are given the table below containing information on user transactions for particular products. Write a query to obtain the year-on-year growth rate for the total spend of each product, for each week (assume there is data each week).

user_transactions

Aa column_name	type
transaction_id	integer
product_id	integer
user_id	integer
spend	float
transaction_date	datetime

8.33. Stripe: Assume you are given the table below containing information on user transactions for a particular business using Stripe. Write a query to obtain the account's rolling 7-day earnings.

user_transactions

Aa column_name	type
transaction_id	integer
user_id	integer
amount	float
transaction_date	datetime

8.34. Facebook: Say you had the entire Facebook social graph (users and their friendships). How would you use MapReduce to find the number of mutual friends for every pair of Facebook users?

8.35. Google: Assume you are tasked with designing a large-scale system that tracks a variety of search query strings and their frequencies. How would you design this, and what trade-offs would you need to consider?

SQL & Database Design Solutions

Note: Due to the variety of SQL flavors, don't be alarmed by minor variations in syntax. We've written the SQL snippets in this book in PostgreSQL.

Solution #8.1

To get the click-through rate, we use the following query, which includes a SUM along with a IF to obtain the total number of clicks and impressions, respectively. Lastly, we filter the timestamp to obtain the click-through rate for just the year 2019.

```
SELECT
    app_id,
    SUM(IF(event_id = 'click', 1, 0)) / SUM(IF(event_id = 'impression', 1, 0))
    AS ctr
FROM
    events
WHERE
    timestamp >= '2019-01-01'
    AND timestamp <= '2020-01-01'
GROUP BY
    1
```

Solution #8.2

To find the cities with the top three highest number of completed orders, we first write an inner query to join the trades and user table based on the user_id column and then filter for complete orders. Using COUNT DISTINCT, we obtain the number of orders per city. With that result, we then perform a simple GROUP BY on city and order by the resulting number of orders, as shown below:

```
SELECT
    u.city,
    COUNT(DISTINCT t.order_id) AS num_orders
FROM
    trades t
    JOIN users u ON t.user_id = u.user_id
WHERE
    t.status = 'complete'
GROUP BY
    city
ORDER BY
    num_orders DESC
LIMIT
    3
```

Solution #8.3

To compare the viewership on laptops versus mobile devices, we first can use a IF statement to define the device type according to the specifications. Since the tablet and phone categories form the "mobile" device type, we can set laptop to be its own device type (i.e., "laptop"). We can then simply SUM the counts for each device type:

```
SELECT
    SUM(IF(device_type = 'laptop', 1, 0)) AS laptop_views,
    SUM(IF(device_type IN ('phone', 'tablet'), 1, 0)) AS mobile_views
FROM
    viewership
```

Solution #8.4

Since we don't care about the particular order_id or user_id, we can use a window function to partition by product and order by transaction date. Spending is then summed over every date and product as follows:

```
SELECT
    trans_date,
    product_id,
    SUM(spend) OVER (
        PARTITION BY product_id
        ORDER BY
            trans_date
    ) AS cum_spend
FROM
    total_trans
ORDER BY
```

```
    product_id,
    trans_date ASC
```

Solution #8.5

In order to obtain a count of products by user, we employ COUNT *product_id* for each user; hence, the GROUP BY is performed over *user_id*. To filter on having spent at least $1000, we use a HAVING SUM(*spend*) > 1000 clause. Lastly, we order *user_ids* by *product_id* count and take the top 10.

```
SELECT
    user_id,
    COUNT(product_id) AS num_products
FROM
    user_transactions
GROUP BY
    user_id
HAVING
    SUM(spend) > 1000
ORDER BY
    num_products DESC
LIMIT
    10
```

Solution #8.6

First, we obtain the number of tweets per user in 2020 by using a simple COUNT within an initial subquery. Then, we use that tweet column as the bucket within a new GROUP BY and COUNT as shown below:

```
SELECT
    num_tweets AS tweet_bucket,
    COUNT(*) AS num_users
FROM
    (
        SELECT
            user_id,
            COUNT(*) AS num_tweets
        FROM
            tweets
        WHERE
            tweet_date BETWEEN '2020-01-01'
            AND '2020-12-31'
        GROUP BY
            user_id
    ) total_tweets
```

```
GROUP BY
    num_tweets
ORDER BY
    num_tweets ASC
```

Solution #8.7

We can't simply perform a count since, by definition, the purchases must have been made on different days (and for the same products). To address this issue, we use the window function RANK while partitioning by *user_id* and *product_id* and then order the result by purchase time in order to determine the purchase number. From this inner subquery, we then obtain the count of *user_ids* for which purchase number was 2 (note that we don't need above 2 since any purchase number above 2 denotes multiple products).

```
SELECT
    COUNT(DISTINCT user_id)
FROM
    (
        SELECT
            user_id,
            RANK() OVER (
                PARTITION BY user_id,
                product_id
                ORDER BY
                    CAST(purchase_time AS DATE)
            ) AS purchase_no
        FROM
            purchases
    ) t
WHERE
    purchase_no = 2
```

Solution #8.8

To find all companies with duplicate listings based on title and description, we can use a RANK() window function partitioning on company_id, job_title, and job_description. Then, we can filter for companies where the largest row number based on those partition fields is greater than 1, which indicates duplicated jobs, and then take a count of the number of companies:

```
WITH job_listing_ranks AS (
    SELECT
        company_id,
        title,
        description,
        ROW_NUMBER() OVER (
```

```
            PARTITION BY company_id,
            title,
            description
            ORDER BY
                post_date
            ) AS rank
        FROM
            job_listings
)
SELECT
    COUNT(DISTINCT company_id)
FROM
    (
        SELECT
            company_id
        FROM
            job_listing_ranks
        WHERE
        MAX(rank) > 1
    )
```

Solution #8.9

Although we could use a self join on *transaction_date* = MIN(*transaction_date*) for each user, we could also use the ROW_NUMBER window function to get the ordering of customer purchases. We could then use that subquery to filter on customers whose first purchase (shown in row one) was valued at 50 dollars or more. Note that this would require the subquery to include spend also:

```
WITH purchase_num AS (
    SELECT
        user_id,
        spend,
        ROW_NUMBER() OVER (
            PARTITION BY user_id
            ORDER BY
                transaction_date ASC
        ) as rownum
    FROM
        user_transactions u
)
SELECT
    user_id
FROM
    purchase_num
WHERE
    rownum = 1
    AND spend >= 50.00
```

Solution #8.10

First, we need to obtain the total number of tweets made by each user on each day, which can be gotten in a CTE using GROUP BY with *user_id* and *tweet_date*, while also applying a COUNT DISTINCT to *tweet_id*. Then, we use a window function on the resulting subquery to take an AVG number of tweets over the six prior rows and the current row (thus giving us the 7-day rolling average), while ordering by *user_id* and *tweet_date*:

```
WITH tweet_counts AS (
    SELECT
        user_id,
        CAST(tweet_date AS date) AS tweet_date,
        COUNT(*) AS num_tweets
    FROM
        tweets
    GROUP BY
        user_id,
        CAST(tweet_date AS date)
)
SELECT
    user_id,
    tweet_date,
    AVG(num_tweets) OVER(
        PARTITION BY user_id
        ORDER BY
            user_id,
            tweet_date ROWS BETWEEN 6 preceding
            AND CURRENT ROW
    ) AS rolling_avg_7d
FROM
    tweet_counts
```

Solution #8.11

First, we obtain the transaction numbers for each user. We can do this by using the ROW_NUMBER window function, where we PARTITION by the user_id and ORDER by the transaction_date fields, calling the resulting field a transaction number. From there, we can simply take all transactions having a transaction number equal to 3.

```
WITH nums AS (
    SELECT
        *,
        ROW_NUMBER() OVER (
            PARTITION BY user_id
            ORDER BY
                transaction_date
```

Solution #8.10

First, we need to obtain the total number of tweets made by each user on each day, which can be gotten in a CTE using GROUP BY with *user_id* and *tweet_date*, while also applying a COUNT DISTINCT to *tweet_id*. Then, we use a window function on the resulting subquery to take an AVG number of tweets over the six prior rows and the current row (thus giving us the 7-day rolling average), while ordering by *user_id* and *tweet_date*:

```
WITH tweet_counts AS (
    SELECT
        user_id,
        CAST(tweet_date AS date) AS tweet_date,
        COUNT(*) AS num_tweets
    FROM
        tweets
    GROUP BY
        user_id,
        CAST(tweet_date AS date)
)
SELECT
    user_id,
    tweet_date,
    AVG(num_tweets) OVER(
        PARTITION BY user_id
        ORDER BY
            user_id,
            tweet_date ROWS BETWEEN 6 preceding
            AND CURRENT ROW
    ) AS rolling_avg_7d
FROM
    tweet_counts
```

Solution #8.11

First, we obtain the transaction numbers for each user. We can do this by using the ROW_NUMBER window function, where we PARTITION by the user_id and ORDER by the transaction_date fields, calling the resulting field a transaction number. From there, we can simply take all transactions having a transaction number equal to 3.

```
WITH nums AS (
    SELECT
        *,
        ROW_NUMBER() OVER (
            PARTITION BY user_id
            ORDER BY
                transaction_date
```

```
        ) AS trans_num
    FROM
        transactions
)
SELECT
    user_id,
    spend,
    transaction_date
FROM
    nums
WHERE
    trans_num = 3
```

Solution #8.12

First, we calculate a subquery with total spend by product and category using SUM and GROUP BY. Note that we must filter by a 2020 transaction date. Then, using this subquery, we utilize a window function to calculate the rankings (by spend) for each product category using the RANK window function over the existing sums in the previous subquery. For the window function, we PARTITION by category and ORDER by product spend. Finally, we use this result and then filter for a rank less than or equal to 3 as shown below.

```
WITH product_category_spend AS (
    SELECT
        product_id,
        category_id,
        SUM(spend) AS total_product_spend
    FROM
        product_spend
    WHERE
        transaction_date BETWEEN '2020-01-01'
        AND '2020-12-31'
    GROUP BY
        product_id,
        category_id
),
top_spend AS (
    SELECT
        p.*,
        RANK() OVER (
            PARTITION BY category_id
            ORDER BY
                total_product_spend DESC
        ) AS rnk
    FROM
```

```
        product_category_spend p
)
SELECT
    *
FROM
    top_spend
WHERE
    rnk <= 3
ORDER BY
    category_id,
    rnk DESC
```

Solution #8.13

First, we obtain the latest transaction date for each user. This can be done in a CTE using the RANK window function to get rankings of products purchased per user based on the purchase transaction date. Then, using this CTE, we simply COUNT both the user ids and product ids where the latest rank is 1 while grouping by each transaction date.

```
WITH latest_date AS (
    SELECT
        transaction_date,
        user_id,
        product_id,
        RANK() OVER (
            PARTITION BY user_id
            ORDER BY
                CAST(transaction_date AS DATE) DESC
        ) AS days_rank
    FROM
        user_transactions
)
SELECT
    transaction_date,
    COUNT(DISTINCT user_id) AS num_users,
    COUNT(product_id) AS total_products
FROM
    latest_date
WHERE
    days_rank = 1
GROUP BY
    transaction_date
ORDER BY
    transaction_date desc
```

Solution #8.14

A database view is the result of a particular query within a set of tables. Unlike a normal table, a view does not have a physical schema. Instead, a view is computed dynamically whenever it is requested. If the underlying tables that the views reference are changed, then the views will change accordingly.

Views have several advantages over tables:

1. Views can simplify workflows by aggregating multiple tables, thus abstracting the complexity of underlying data or operations.
2. Since views can represent only a subset of the data, they provide limited exposure of the table's underlying data and hence increase data security.
3. Since views do not store actual data, there is significantly less memory overhead.

Solution #8.15

SQL statements that modify the database, like UPDATE, INSERT, and DELETE, need to change not only the rows of the table but also the underlying indexes. Therefore, the performance of those statements depends on the number of indexes that need to be updated. The larger the number of indexes, the longer it takes those statements to execute. On the flip side, indexing can dramatically speed up row retrieval since no underlying indexes need to be modified. This is important for statements performing full table scans, like SELECTs and JOINs.

Therefore, for databases used in online transaction processing (OLTP) workloads, where database updates and inserts are common, indexes generally lead to slower performance. In situations where databases are used for online analytical processing (OLAP), where database modifications are infrequent but searching and joining the data is common, indexes generally lead to faster performance.

Solution #8.16

A primary key uniquely identifies an entity. It can consist of multiple columns (known as a *composite key*) and cannot be NULL.

Characteristics of a good primary key are:

- **Stability:** a primary key should not change over time.
- **Uniqueness:** having duplicate (non-unique) values for the primary key defeats the purpose of the primary key.
- **Irreducibility:** no subset of columns in a primary key is itself a primary key. Said another way, removing any column from a good primary key means that the key's uniqueness property would be violated.

Solution #8.17

Advantages of Relational Databases: Ensure data integrity through a defined schema and the ACID properties. Easy to get started with and use for small-scale applications. Lends itself well to vertical scaling. Uses an almost standard query language, making learning or switching between different types of relational databases easy.

Advantages of NoSQL Databases: Offers more flexibility in data format and representations, which makes working with unstructured or semistructured data easier. Hence, useful when still iterating on the data schema or adding new features/functionality rapidly like in a startup environment. Convenient to scale with horizontal scaling. Lends itself better to applications that need to be highly available.

Disadvantages of Relational Databases: Data schema needs to be known in advance. Altering schemas is possible, but frequent changes to the schema for large tables can cause performance issues. Horizontal scaling is relatively difficult, leading to eventual performance bottlenecks.

Disadvantages of NoSQL Databases: As outlined by the BASE framework, weaker guarantees on data correctness are made due to the soft-state and eventual consistency property. Managing data consistency can also be difficult due to the lack of a predefined schema that's strictly adhered to. Depending on the type of NoSQL database, it can be challenging for the database to handle some types of complex queries or access patterns.

Solution #8.18

At a high level, to shuffle the data randomly, we need to map each row of the input data to a random key. This ensures that the row of input data is randomly sent to a reducer, where it's simply outputted. More concretely, the steps of the MapReduce algorithm are:

1. Map step: Each row is assigned a random value from 1,...,k, where k is the number of reducer nodes available. Therefore, for every key, the output is the tuple (key, row).
2. Shuffle step: Rows with the same input key go to the same reducer.
3. Reduce step: For each record, the row is simply outputted.

Since the reducer only has rows that were filtered randomly for a given value of i, where i is from 1,...,k, the resulting output will be ordered randomly.

Solution #8.19

A couple of answers are possible, but here are some examples:

Similarities:

1. Both clauses are used to limit/filter a given query's results.
2. Both clauses are optional within a query.
3. Usually, queries utilizing one of the two can be transformed to use the other.

Differences:

1. A HAVING clause can follow a GROUP BY statement, but WHERE cannot.
2. A WHERE clause evaluates per row, whereas a HAVING clause evaluates per group.
3. Aggregate functions can be referred to in a logical expression if a HAVING clause is used.

Solution #8.20

Foreign keys are a set of attributes that aid in joining tables by referencing primary keys (although joins can occur without them). Primarily, they exist to ensure data integrity. The table with the primary key is called the parent table, whereas the table with the foreign key is called the child table. Since foreign keys create a link between the two tables, having foreign keys ensures that these links are valid and prevents data from being inserted that would otherwise violate these conditions. Foreign keys can be created during CREATE commands, and it is possible to DROP or ALTER foreign keys.

When designating foreign keys, it is important to think about the *cardinality* — the relationship between parent and child tables. Cardinality can take on four forms: one-to-one (one row in the parent table maps to one row in the child table), one-to-many (one row in the parent table maps to many rows in the child table), many-to-one (many rows in the parent table map to one row in the

child table), and many-to-many (many rows in the parent table map to many rows in the child table). The particular type of relationship between the parent and child table determines the specific syntax used when setting up foreign keys.

Solution #8.21

Both clustered indexes and non-clustered indexes help speed up queries in a database. With a clustered index, database rows are stored physically on the disk in the same exact order as the index. This arrangement allows you to rapidly retrieve all rows that fall into a range of clustered index values. However, there can only be one clustered index per table since data can only be sorted physically on the disk in one particular way at a time.

In contrast, a non-clustered index does not match the physical layout of the rows on the disk on which the data are stored. Instead, it duplicates data from the indexed column(s) and contains a pointer to the rest of data. A non-clustered index is stored separately from the table data, and hence, unlike a clustered index, multiple non-clustered indexes can exist per table. Therefore, insert and update operations on a non-clustered index are faster since data on the disk doesn't need to match the physical layout as in the case of a clustered index. However, this makes the storage requirement for a non-clustered index higher than for a clustered index. Additionally, lookup operations for a non-clustered index may be slower than that of a clustered index since all queries must go through an additional layer of indirection.

Solution #8.22

First, we need to obtain the top 100 most popular topics for the given date by employing a simple subquery. Then, we need to identify all users who followed no topic included within these top 100 for the date specified. Equivalently, we could identify those that did follow one of these topics and then filter them out of this list of users that existed on 2021-01-01.

Two approaches are as follows:

1. use the MINUS (or EXCEPT) operator and subtract those following a top 100 topic (via an inner join) from the entire user universe
2. use a WHERE NOT EXISTS clause in a similar fashion.

For simplicity, the solution below uses the MINUS operator. Note that we need to filter for date in the *user_topics* table so that we capture only existing users as of 2020-01-01:

```
WITH top_topics AS (
    SELECT
        *
    FROM
        topic_rankings
    WHERE
        ranking_date = '2021-01-01'
        AND rank <= 100
)
SELECT
    DISTINCT user_id
FROM
```

```
    user_topics
WHERE
    follow_date <= '2021-01-01'
MINUS
SELECT
    u.user_id
FROM
    user_topics u
    JOIN top_topics t ON u.topic_id = t.topic_id
```

Solution #8.23

In order to calculate user retention, we need to check for each user whether they were active this month versus last month. To bucket days into each month, we need to obtain the first day of the month for the specified date by using DATE_TRUNC. We use a COUNT DISTINCT over *user_id* to obtain the monthly active user (MAU) count for the month. This can be put into a subquery called *curr_month*, and then EXISTS can be used to check it against another subquery for the previous month, *last_month*. In that subquery, ADD_MONTHS can be used with an argument of 1 to get the previous month, thereby allowing us to check for user actions from the previous month (since that would mean they were logged in), as shown below:

```
SELECT
    DATE_TRUNC('month', curr_month.timestamp) AS month,
    COUNT(DISTINCT curr_month.user_id) AS mau
FROM
    user_actions curr_month
WHERE
    EXISTS (
        SELECT
            *
        FROM
            user_actions last_month
        WHERE
            add_months(DATE_TRUNC('month', last_month.timestamp), -1) =
            DATE_TRUNC('month', curr_month.timestamp)
    )
GROUP BY
    DATE_TRUNC('month', curr_month.timestamp)
ORDER BY
    month ASC;
```

Solution #8.24

First, we can perform a CTE to obtain the total session duration by user and session type between the start and end dates. Then, we can use RANK to obtain the rank, making sure to partition by session type and then order by duration as in the query below:

```
WITH user_duration AS (
    SELECT
        user_id,
        session_type,
        SUM(duration) AS duration
    FROM
        sessions
    WHERE
        start_time BETWEEN '2021-01-01'
        AND '2021-02-01'
    GROUP BY
        user_id,
        session_type
)
SELECT
    user_id,
    session_type,
    RANK() OVER (
        partition by user_id
        session_type
        ORDER BY
        user_id
            duration DESC
    ) AS rank
FROM
    user_duration
ORDER BY
    session_type,
    rank DESC
```

Solution #8.25

We can obtain the total time spent on sending and opening using conditional IF statements for each activity type while getting the amount of time_spent in a CTE. We can also obtain the total time spent in the same CTE. Next, we take that result and JOIN by the corresponding *user_id* with activities. We filter for just send and open activity types and group by age bucket. Then, using this CTE, we can calculate the percentages of send and open time spent versus overall time spent as follows:

```
WITH time_stats AS (
    SELECT
        age_breakdown.age_bucket,
        SUM(IF(type = 'send', time_spent, 0)) AS send_timespent,
        SUM(IF(type = 'open', time_spent, 0)) AS open_timespent,
        SUM(time_spent) AS total_timespent
    FROM
```

```
        age_breakdown
        JOIN activities on age_breakdown.user_id = activities.user_id
    WHERE
        activities.type IN ('send', 'open')
    GROUP BY
        age_breakdown.age_bucket
)
SELECT
    age_bucket,
    send_timespent / total_timespent AS pct_send,
    open_timespent / total_timespent AS pct_open
FROM
    time_stats
```

Solution #8.26

The first step is to determine the query logic for when two sessions are concurrent. Say we have two sessions, session 1 and session 2. Note that there are two cases in which they overlap:

1. If session 1 starts first, then the start time for session 2 is less than or equal to session 1's end time
2. If session 2 starts first, then session 1's end time for session 1 is greater than or equal to session 2's start time

In total, this simplifies to session 2's start time falling between session 1's start time and session 1's end time.

With this in mind, we can calculate the number of sessions that started during the time another session was running by using an inner join and using BETWEEN to check the concurrency case as follows:

```
SELECT
    s1.session_id,
    COUNT(s2.session_id) AS concurrents
FROM
    sessions s1
    JOIN sessions s2 ON s1.session_id != s2.session_id
    AND s2.start_time BETWEEN s1.start_time
    AND s1.end_time
GROUP BY
    s1.session_id
ORDER BY
    concurrents DESC
LIMIT
    1
```

Solution #8.27

First, we need to identify businesses having reviews consisting of only 4 or 5 stars. We can do so by using a CTE to find the lowest number of stars given to a business across all its reviews. Then, we

can use a SUM and IF statement to filter across businesses with a minimum review of 4 or 5 stars to get the total number of top-rated businesses, and then divide this by the total number of businesses to find the percentage of top-rated businesses.

```
WITH min_review AS (
    SELECT
        business_id,
        min(review_stars) AS min_stars
    FROM
        reviews
    GROUP BY
        business_id
)
SELECT
    (1.0 * SUM(IF(min_stars >= 4, 1, 0)) / COUNT(*)) * 100.0 AS top_places_pct
FROM
    min_review
```

Solution #8.28

First, we need to establish which measurements are odd numbered and which are even numbered. We can do so by using the ROW_NUMBER window function over the *measurement_time* to obtain the measurement number during a day. Then, we filter for odd numbers by checking if a measurement's mod 2 is 1 for odds or is 0 for evens. Finally, we sum by date using a conditional IF statement, summing over the corresponding *measurement_value*:

```
WITH measurements_by_count AS (
    SELECT
        CAST(measurement_time AS date) measurement_day,
        measurement_value,
        ROW_NUMBER() OVER (
            PARTITION BY CAST(measurement_time AS date)
            ORDER BY
                measurement_time ASC
        ) AS measurement_count
    FROM
        measurements
)
SELECT
    measurement_day,
    SUM(
        IF(measurement_count % 2 != 0, 0, measurement_value)
    ) AS odd_sum,
    SUM(
        IF(measurement_count % 2 = 0, measurement_value, 0)
    ) AS even_sum
```

```
FROM
    measurements_by_count
GROUP BY
    measurement_day
ORDER BY
    measurement_day ASC
```

Solution #8.29

First, we obtain the latest week's users. To do this, we use NOW for the current time and subtract an INTERVAL of 7 days, thus providing the relevant user IDs to look at. By using LEFT JOIN, we have all signed-in users, and whether they made a purchase or not. Now we take the COUNT of DISTINCT users from the purchase table, divide it by the COUNT of DISTINCT users from the signup table, and then multiply the results by 100 to obtain a percentage:

```
SELECT
    COUNT(DISTINCT p.user_id) / COUNT(DISTINCT s.user_id) * 100 AS
    last_week_pct
FROM
    signups s
    LEFT JOIN user_purchases p ON p.user_id = s.user_id
WHERE
    s.signup_date > NOW() - INTERVAL 7 DAY
```

Solution #8.30

First, we can join the transactions and product tables together based on *product_id* to get the *user_id*, *product_name*, and *transaction_time* for the transactions. With the CTE at hand, we can do a self join to fetch products that were purchased together by a single user by joining on *transaction_id*. Note that we want all pairs of products, but we don't want to overcount, i.e., if user A purchased products X and Y in the same transaction, then we only want to count the (X, Y) transaction once, and not also (Y, X). To handle this, we can use a condition within the inner join that the *product_id* of A is less than that of B (where A and B are the CTE results from before). Lastly, we use a GROUP BY clause for each pair of products and sort by the resulting count, taking the top 10:

```
WITH (
    SELECT
        t.user_id,
        p.product_name,
        t.transaction_id
    FROM
        transactions t
        JOIN product p ON t.product_id = p.product_id
) AS purchase_info
```

```
SELECT
    p1.product_name AS product1,
    p2.product_name AS product2,
    COUNT(*) AS count
FROM
    purchase_info p1
    JOIN purchase_info p2 ON p1.transaction_id = p2.transaction_id
    AND p1.product_id < p2.product_id
GROUP BY
    1,
    2
ORDER BY
    3 DESC
LIMIT
    10
```

Solution #8.31

First, we look at all users who did not log in during the previous month. To obtain the last month's data, we subtract an INTERVAL of 1 month from the current month's login date. Then, we use a WHERE EXISTS against the previous month's interval to check whether there was a login in the previous month. Finally, we COUNT the number of users satisfying this condition.

```
SELECT
    DATE_TRUNC('month', current_month.login_date) AS current_month,
    COUNT(*) AS num_reactivated_users
FROM
    user_logins current_month
WHERE
    NOT EXISTS (
        SELECT
            *
        FROM
            user_logins last_month
        WHERE
            DATE_TRUNC('month',    last_month.login_date)    BETWEEN    DATE_
            TRUNC('month', current_month.login_date) AND DATE_TRUNC('month',
            current_month.login_date) - INTERVAL '1 month'
    )
```

Solution #8.32

First, we need to obtain the total weekly spend by product using SUM and GROUP BY operations and use DATE_TRUNC on the transaction date to specify a particular week. Using this information, we then calculate the prior year's weekly spend for each product. In particular, we want to take a LAG for 52 weeks, and PARTITION BY product, to calculate that week's prior year spend for the

given product. Lastly, we divide the current total spend by the corresponding previous 52-week lag value:

```
WITH weekly_spend AS (
    SELECT
        DATE_TRUNC('week', transaction_date :: DATE) AS week,
        product_id,
        SUM(spend) AS total_spend
    FROM
        user_transactions
    GROUP BY
        week,
        product_id
),
total_weekly_spend AS (
    SELECT
        w.*,
        LAG(total_spend, 52) OVER (
            PARTITION BY product_id
            ORDER BY
              week ASC
        ) as prev_total_spend
    FROM
       weekly_spend w
)
SELECT
   product_id,
   total_spend,
   prev_total_spend,
   total_spend / prev_total_spend AS spend_yoy
FROM
   total_weekly_spend
```

Solution #8.33

First, we need to obtain the total daily transactions using a simple SUM and GROUP BY operation. Having the daily transactions, we then perform a self join on the table using the condition that the transaction date for one transaction occurs within 7 days of the other, which we can check by using the DATE_ADD function along with the condition that the earlier date doesn't precede the later date:

```
WITH daily_transactions AS (
   SELECT
      CAST(transaction_date AS DATE),
      SUM(amount) AS total_amount
   FROM
      user_transactions
```

```
    GROUP BY
        transaction_date
)
SELECT
    t2.transaction_date,
    SUM(t1.amount) AS weekly_rolling_total
FROM
    daily_transactions t1
    INNER JOIN daily_transactions t2 ON t1.transaction_date > DATE_ADD('DAY',
    -7, t2.transaction_date) AND t1.transaction_date <= t2.transaction_date
GROUP BY
    t2.transaction_date
ORDER BY
    t2.transaction_date ASC
```

Solution #8.34

To use MapReduce to find the number of mutual friends for all pairs of Facebook users, we can think about what the end output needs to be and then work backward. Concretely, for all given pairs of users X and Y, we want to identify which friends they have in common, from which we'll derive the mutual friend count. The core of this algorithm is finding the intersection between the friends list for X and the friends list for Y. This operation can be delegated to the reducer. Therefore, it is sensible that the key for our reduce step should be the tuple (X, Y) and that the value to be reduced is the combination of the friends list of X and the friends list of Y. Thus, in the map step, we want to output the tuple (X, Z) for each friend Z that X has.

As an example, assume that X is friends with [W, Y, Z] and Y is friends with [X, Z].

1. Map step: For X, we want to output the following tuples: 1) ((X, W), [W, Y, Z]), 2) ((X, Y), [W, Y, Z]), and 3) ((X, Z), [W, Y, Z]). For Y we want to output the following tuples: 1) ((X, Y), [X, Z]), and 2) ((Y, Z), [X, Z]). Note that the key is sorted, so that (Y, X) → (X, Y).
2. Shuffle step: Each machine is delegated data based on the keys from the map step, i.e., each tuple (X, Y). So, in the previous example, note that the map step outputs the key (X, Y) for both X and Y, and therefore both of the keys are on the same machine. That machine will therefore have the tuple (X, Y) as the key, and will store [W, Y, Z] and [X, Z] to be used in the reduce step.
3. Reduce step: We group by keys and take the intersection of the resulting lists. For the example of (X, Y) → [W, Y, Z], [X, Z], we take the intersection of [W, Y, Z] and [X, Z], which is [Z]. Thus, we return the length of the set (1) for the input (X, Y).

Therefore, we are able to identify Z as the common friend of X and Y, and can return 1 as the number of mutual friends. The process outlined above is repeated in parallel for every pair of Facebook users in order to find the final mutual friend counts between each pair of users.

Solution #8.35

To design a system that tracks search query strings and their frequencies, we can start with a basic keyvalue store. For each search query string, we store the corresponding frequency in a database table containing only those two fields. To build the system at scale, we have two options: vertical scaling or horizontal scaling. For vertical scaling, we would add more CPU and RAM to existing

machines, an action that is not likely to work well at Google's scale. Instead, we should consider horizontal scaling, in which more machines (nodes) are added to a cluster. We would then store search query strings across a large set of nodes and be able to quickly find which node contains a given search query string.

For the actual sharding logic, consisting of mapping query strings to particular shards, several approaches are possible. One way is to use a range of values; for example, we could have 26 shards and map query strings beginning with A to shard 1, B to shard 2, and so on. While this approach is simple to implement, its primary drawback is that the data obtained could be unevenly distributed, meaning that certain shards would need to deal with much more data than others. For example, the shard containing strings starting with the letter 'x' will have much less load than the shard containing strings starting with the letter 'a.'

An alternative sharding scheme could be to use a hash function that maps the query string to a particular shard number. This is another simple solution and would help reduce the problem of all data being mapped to the same shard. However, adding new nodes is troublesome since the hash function must be re-run across all nodes and the data rebalanced. However, those problems can be addressed through a method called "consistent hashing," which aids in data rebalancing when new servers are added.