Interview References

1. SQL & Metrics Interview (45 min)

This technical interview will be 45 minutes, conducted via HackerRank CodePair with a member of the team. The call will consist of both technical and case study portions.

At Airbnb, SQL-type languages (Hive, Presto, etc.) are the standard for data analysis. This interview will test your ability with basic and intermediate SQL concepts such as joins, aggregations, filtering, subqueries, self-joins, and window functions.

You're encouraged to use online resources during the interview—our goal is to simulate a real work environment.

The case study assesses your understanding of product measurement and your ability to structure an analytical approach to product metrics. You may be asked to dig deeper or clarify your methods.

SQL Query Reference: Basic SELECT Examples

```
SELECT *
   FROM listings
   WHERE city = 'San Francisco';

   SELECT listing_id, price, room_type
   FROM listings
   WHERE city = 'New York' AND price > 200 AND is_active = 1;

   SELECT listing_id, city, price
   FROM listings
   WHERE city = 'Los Angeles' OR city = 'Miami';

   SELECT *
   FROM listings
   WHERE city IN ('San Francisco', 'Seattle', 'Portland');

   SELECT listing_id, city, price
   FROM listings
   WHERE price BETWEEN 100 AND 300;

   SELECT listing_id, city
   FROM listings
   WHERE listing_name LIKE '%Beach%';
```

```sql
SELECT listing_id, city
FROM listings
WHERE description IS NULL;

SELECT DISTINCT listing_id
FROM reviews
WHERE review_text IS NOT NULL;

SELECT listing_id, city, price
FROM listings
ORDER BY price ASC;

SELECT listing_id, city, price
FROM listings
ORDER BY price DESC
LIMIT 10;

SELECT *
FROM bookings
LIMIT 20;

SELECT DISTINCT city
FROM listings;

SELECT COUNT(*) AS total_listings
FROM listings;

SELECT COUNT(*) AS la_listings
FROM listings
WHERE city = 'Los Angeles';

SELECT SUM(total_price) AS total_revenue
FROM bookings
WHERE status = 'completed';

SELECT AVG(price) AS avg_price
FROM listings;
```

```sql
SELECT MIN(price) AS min_price
FROM listings
WHERE is_active = 1;

SELECT MAX(price) AS max_price
FROM listings;

SELECT *
FROM bookings
WHERE booking_date >= CURRENT_DATE - INTERVAL '30 days';

SELECT *
FROM bookings
WHERE YEAR(booking_date) = 2024;

SELECT listing_id, price,
    CASE
        WHEN price < 100 THEN 'Budget'
        WHEN price BETWEEN 100 AND 300 THEN 'Mid-Range'
        ELSE 'Luxury'
    END AS price_category
FROM listings;

SELECT listing_id,
    CONCAT(city, ', ', country) AS location
FROM listings;

SELECT listing_id, UPPER(city) AS city_upper
FROM listings;

SELECT guest_id, LOWER(email) AS email_normalized
FROM guests;

SELECT listing_id, description
FROM listings
WHERE LENGTH(description) < 50;

SELECT listing_id, ROUND(price, 2) AS price_rounded
```

```
FROM listings;

SELECT listing_id, room_type
FROM listings
WHERE room_type != 'Entire home/apt';

SELECT listing_id, city, price
FROM listings
ORDER BY city ASC, price DESC;

SELECT listing_id, COALESCE(price, 0) AS price
FROM listings;
```

Business Summary

Summary of Airbnb experimentation in the marketplace context.

Marketplace Snapshot

Experimentation System and Practices

Marketing Context and Levers to Test

Metric Wiring for Readouts

Design Patterns to Reference

Quick talk track (60–90 seconds):

[Prepare a brief summary or elevator pitch for the interview.]

If you'd like, I can help convert this to a one-pager with example interview questions for each design.

This technical interview will be 45 minutes and will be conducted via HackerRank CodePair with a member of the team. The phone interview will consist of a technical portion and a case study.

At Airbnb we use a varied set of tools for doing data analysis, but the one constant in almost all pathways is the use of SQL-type languages (Hive, Presto etc.). With that in mind, this interview will test your knowledge of basic and intermediate SQL and may include concepts such as joining, aggregating, filtering, subqueries, self-joins, windowing functions etc.

We do not mandate that you memorize anything and you are encouraged to use any and all tools available to you (including looking up syntax on the internet). We endeavor to simulate the working environment to the best of our ability.

The case study portion of the interview will test your understanding of how to measure product development, and your ability to structure an analytical approach to solve a problem involving product metrics. The questions will be at a high level, but the team might ask you to dig a bit deeper to clearly define your approach.

1.

i. **Q10: ORDER BY - Ascending**

ii. : Find all listings that have guest reviews.

a.
```
SELECT DISTINCT
listing_id
FROM reviews
WHERE review_text IS NOT NULL;
```

b. : Find all listings that don't have a description.

2.
```
SELECT listing_id, city
FROM listings
WHERE description IS NULL;
```

2. : Find all listings with names containing "Beach".

ii.
```
SELECT listing_id, city
FROM listings
WHERE listing_name LIKE '%Beach%';
```

ii. : Find listings with prices between $100 and $300 per night.

b.
```
SELECT listing_id, city, price
FROM listings
WHERE price BETWEEN 100 AND 300;
```

b. : Find all listings in San Francisco, Seattle, or Portland.

2.
```
SELECT *
FROM listings
WHERE city IN ('San Francisco', 'Seattle', 'Portland');
```

2. : Find listings in either Los Angeles or Miami.

ii.

```
SELECT listing_id, city, price
FROM listings
WHERE city = 'Los Angeles' OR city = 'Miami';
```

ii. : Find all active listings in New York with price greater than $200 per night.

b.

```
SELECT listing_id, price, room_type
FROM listings
WHERE city = 'New York' AND price > 200 AND is_active = 1;
```

b. : Find all listings located in San Francisco.

2.

```
SELECT *
FROM listings
WHERE city = 'San Francisco';
```

SQL Query Reference: Basic SELECT Examples

Business Summary

Summary of Airbnb experimentation with marketplace context.

Marketplace snapshot

Topic	Key facts	Why it matters for experiments
Two-sided market	Hosts supply listings; guests demand stays and experiences. >5M hosts and >8M active listings as of Q4-2024.	Supply constraints and host behavior create interference and inventory effects. Split metrics by side and market. (The Airbnb Tech Blog)
Core metrics	Nights & Experiences Booked (N&E) and Gross Booking Value (GBV). GBV is the dollar value of	Use N&E as leading indicator; GBV leads revenue. Tie

Topic	Key facts	Why it matters for experiments
	bookings incl. host earnings, fees, cleaning, taxes; N&E counts nights and seats net of cancels. (SEC)	experiment targets to these. (SEC)
Revenue model	Service fees to guests and hosts; revenue recognized at check-in. 2024 revenue ≈ $11.1B. (SEC)	Readouts on “revenue” lag; prefer bookings/N&E for timely calls. (SEC)
Geography	Growth accelerating in APAC and LATAM; brand pushes and local payments (e.g., Pix in Brazil) increased first-time bookers.	Stratify or randomize by market; expect heterogeneous effects and adoption frictions.

Experimentation system and practices

Area	What Airbnb published	Takeaway for interview
Philosophy & pitfalls	Classic A/B pitfalls amplified by marketplace: long booking lags, cross-device identity, host responsiveness, multiple booking flows, early-significance traps; use A/A tests and bias checks. (The Airbnb Tech Blog)	Plan fixed horizons or sequential controls; monitor booking-lag cohorts; validate assignment and logging with A/A. (The Airbnb Tech Blog)
Platform (ERF)	Internal A/B platform with Airflow/Presto; ~2.5k metrics/day and ~50k experiment-metric combos; metric hierarchy: Core, Target, Certified; precomputed dimensional cuts. (Medium)	Align analyses to Core metrics; slice by subject-level (guest/host, geo, device) and event-level dimensions. (Medium)
Ranking eval	Interleaving for search ranking: team-draft pairs, booking-based attribution, ~50× speed vs A/B	Use interleaving to triage models, then confirm with A/B on business KPIs. (Airbnb Tech)

Area	What Airbnb published	Takeaway for interview
	for ranker screening; ~82% agreement with A/B. (Airbnb Tech)	
Non-standard designs	SEO measurement at market/URL level with randomized canonical URLs and diff-in-diff with cluster-robust SEs. (Medium)	When user-level randomization is impossible, switch the unit (market/URL), use DiD or synthetic controls. (Medium)

Marketing context and levers to test

Lever	Examples they cite	Experiment/causal design notes
Acquisition mix	Brand campaigns expanded in LATAM; local rails like Pix improved conversion and first-time bookers.	Geo-level holdouts or staggered rollouts; measure first-time bookers, assisted bookings, CAC/LTV.
Product-led growth	Search ranking, pricing, payments, onboarding; 535+ features/UX upgrades driving conversion. (Airbnb Newsroom)	Use ERF; prioritize conversion-proximate KPIs that predict N&E and GBV.
Monetization	Added guest travel insurance and an extra fee for cross-currency bookings. (Airbnb Newsroom)	Price/fee tests require guardrails on conversion, cancellations, NPS, and supply elasticity.
Supply growth	Co-Host Network to unlock hosting capacity; launched across 10 countries. (Airbnb Newsroom)	Supply-side experiments: split by city or host cohorts; watch acceptance rate, availability, calendar density.

Metric wiring for readouts

Stage	Primary KPI	Guardrails / cuts
Discovery → Search	Search CTR, listing views, save rate	Device, market, new vs repeat, Rooms vs entire home. (Medium)
Consideration → Request	Message sends, request rate, IB share	Host response/accept rate, time to first response. (The Airbnb Tech Blog)
Booking	N&E, GBV, ADR	Cancellation rate, coupon/fee leakage, fraud. (SEC)
Post-booking	CS contacts, ratings, repeat rate	Cross-border vs domestic, length of stay.

Design patterns to reference

Scenario	Recommended approach
Ranking/model iteration	Interleaving → confirm via A/B on Core metrics. (Airbnb Tech)
SEO landing pages	Randomize at canonical-URL level; diff-in-diff with clustered SEs; power via simulation. (Medium)
Geo marketing	Country/region switchbacks or stepped-wedge rollouts; CUPED/DiD for variance reduction; monitor spillovers. (Medium)
Marketplace interference	Segment by side (host/guest) and market; check capacity and acceptance bottlenecks before launch decisions. (The Airbnb Tech Blog)

Quick talk track (60–90 seconds)

- Airbnb runs an ERF-based experimentation program tied to Core, Target, and Certified metrics, with dimensional cuts by geo/device/subject. (Medium)
- For ranking, they interleave to get ~50× faster read, then A/B on conversion KPIs. (Airbnb Tech)
- For SEO and other externally mediated surfaces, they randomize at the market/URL level and estimate effects via diff-in-diff with clustered errors. (Medium)
- Top-line company metrics are N&E and GBV; revenue is fee-based and recognized at check-in, so near-term reads use bookings, not revenue. (SEC)
- Current growth focus includes expansion markets, brand investment, and localized payments that raise first-time bookers.

If you want, I can convert this into a one-pager with example interview prompts tied to each design.

2. : Filtering with WHERE
3. : AND operator
4. : OR logic
5. : IN clause
6. : Range filtering
7. : Pattern matching
8. : IS NULL
9. : IS NOT NULL
10. : Sorting

Question 2	Concept	Difficulty	Business Purpose
Find all listings located in San Francisco.	Filtering with WHERE	Easy	Find listings in specific market for regional analysis

```
SELECT *
    FROM listings
    WHERE city = 'San Francisco';
```

Question 3	Concept	Difficulty	Business Purpose

Find all active listings in New York with price > $200.	AND operator	Easy	Identify premium active listings for marketing campaigns

```
SELECT listing_id, price, room_type
   FROM listings
   WHERE city = 'New York' AND price > 200 AND is_active = 1;
```

Question 4	Concept	Difficulty	Business Purpose
Find listings in either Los Angeles or Miami.	OR logic	Easy	Analyze multi-market presence

```
SELECT listing_id, city, price
   FROM listings
   WHERE city = 'Los Angeles' OR city = 'Miami';
```

Question 5	Concept	Difficulty	Business Purpose
Find all listings in San Francisco, Seattle, or Portland.	IN clause	Easy	Multi-city market analysis

```
SELECT *
   FROM listings
   WHERE city IN ('San Francisco', 'Seattle', 'Portland');
```

Question 6	Concept	Difficulty	Business Purpose
Find listings with prices between $100 and $300.	Range filtering	Easy	Identify mid-range pricing segments

```
SELECT listing_id, city, price
   FROM listings
   WHERE price BETWEEN 100 AND 300;
```

Question 7	Concept	Difficulty	Business Purpose
Find all listings with names containing “Beach”.	Pattern match	Easy	Search listings by name patterns for category

```
SELECT listing_id, city
   FROM listings
   WHERE listing_name LIKE '%Beach%';
```

Question 8	Concept	Difficulty	Business Purpose
Find all listings that don’t have a description.	IS NULL	Easy	Data quality check for incomplete profiles

```
SELECT listing_id, city
   FROM listings
   WHERE description IS NULL;
```

Question 9	Concept	Difficulty	Business Purpose
Find all listings that have guest reviews.	IS NOT NULL	Easy	Identify complete listings for quality scoring

```
SELECT DISTINCT listing_id
   FROM reviews
   WHERE review_text IS NOT NULL;
```

Question 10	Concept	Difficulty	Business Purpose
List all listings ordered by price from	Sorting	Easy	Price analysis from low to high

low to high.			

```
SELECT listing_id, city, price
   FROM listings
   ORDER BY price ASC;
```

Question 11	Concept	Difficulty	Business Purpose
List the top 10 most expensive listings.	Sorting DESC	Easy	Identify premium inventory

```
SELECT listing_id, city, price
   FROM listings
   ORDER BY price DESC
   LIMIT 10;
```

Question 12	Concept	Difficulty	Business Purpose
Retrieve the first 20 bookings from the bookings table.	Result limiting	Easy	Sample data for quick analysis

```
SELECT *
   FROM bookings
   LIMIT 20;
```

Question 13	Concept	Difficulty	Business Purpose
Find all unique cities where Airbnb has listings.	Unique values	Easy	Identify all markets we operate in

```
SELECT DISTINCT city
   FROM listings;
```

Question 14	Concept	Difficulty	Business Purpose

Count the total number of listings.	COUNT function	Easy	Total inventory metrics

```
SELECT COUNT(*) AS total_listings
   FROM listings;
```

Question 15	Concept	Difficulty	Business Purpose
Count how many listings are in Los Angeles.	COUNT with WHERE	Easy	Market-specific inventory

```
SELECT COUNT(*) AS la_listings
   FROM listings
   WHERE city = 'Los Angeles';
```

Question 16	Concept	Difficulty	Business Purpose
Calculate the total revenue from all completed bookings.	Aggregation - SUM	Easy	Calculate total revenue

```
SELECT SUM(total_price) AS total_revenue
   FROM bookings
   WHERE status = 'completed';
```

Question 17	Concept	Difficulty	Business Purpose
Find the average nightly price of all listings.	Aggregation - AVG	Easy	Average pricing analysis

```
SELECT AVG(price) AS avg_price
   FROM listings;
```

Question 18	Concept	Difficulty	Business Purpose
Find the minimum price among all listings.	Aggregation - MIN	Easy	Identify entry-level pricing

```
SELECT MIN(price) AS min_price
   FROM listings
   WHERE is_active = 1;
```

Question 19	Concept	Difficulty	Business Purpose
Find the maximum price among all listings.	Aggregation - MAX	Easy	Identify luxury segment pricing

```
SELECT MAX(price) AS max_price
   FROM listings;
```

Question 20	Concept	Difficulty	Business Purpose
Find all bookings made in the last 30 days.	Date comparison	Easy	Recent bookings analysis

```
SELECT *
   FROM bookings
   WHERE booking_date >= CURRENT_DATE - INTERVAL '30 days';
```

Question 21	Concept	Difficulty	Business Purpose
Find all bookings made in 2024.	Date extraction	Easy	Year-over-year analysis

```
SELECT *
```

```
   FROM bookings
   WHERE YEAR(booking_date) = 2024;
```

Question 22	Concept	Difficulty	Business Purpose
Categorize listings as ‘Budget’, ‘Mid-Range’, or ‘Luxury’ by price.	Conditional logic	Easy	Categorize listings by price tier

```
SELECT listing_id, price,
       CASE
           WHEN price < 100 THEN 'Budget'
           WHEN price BETWEEN 100 AND 300 THEN 'Mid-Range'
           ELSE 'Luxury'
       END AS price_category
   FROM listings;
```

Question 23	Concept	Difficulty	Business Purpose
Concatenate city and country for each listing.	String concatenation	Easy	Create display names for reporting

```
SELECT listing_id,
       CONCAT(city, ', ', country) AS location
   FROM listings;
```

Question 24	Concept	Difficulty	Business Purpose
Convert all city names to uppercase.	String manipulation	Easy	Standardize city names for grouping

```
SELECT listing_id, UPPER(city) AS city_upper
   FROM listings;
```

Question 25	Concept	Difficulty	Business Purpose

Convert all guest emails to lowercase.	String manipulation	Easy	Normalize email addresses

```
SELECT guest_id, LOWER(email) AS email_normalized
   FROM guests;
```

Question 26	Concept	Difficulty	Business Purpose
Find listings with description length less than 50 characters.	String length	Easy	Identify incomplete descriptions

```
SELECT listing_id, description
   FROM listings
   WHERE LENGTH(description) < 50;
```

Question 27	Concept	Difficulty	Business Purpose
Round all listing prices to 2 decimal places.	Number formatting	Easy	Display prices with 2 decimals

```
SELECT listing_id, ROUND(price, 2) AS price_rounded
   FROM listings;
```

Question 28	Concept	Difficulty	Business Purpose
Find all listings that are NOT entire home/apt.	Negation	Easy	Exclude specific segments

```
SELECT listing_id, room_type
   FROM listings
   WHERE room_type != 'Entire home/apt';
```

Question 29	Concept	Difficulty	Business Purpose
List listings ordered by city, then by price within each city.	Multi-column sort	Easy	Organize listings by location/price

```
SELECT listing_id, city, price
   FROM listings
   ORDER BY city ASC, price DESC;
```

Question 30	Concept	Difficulty	Business Purpose
Replace NULL prices with 0.	Null handling	Easy	Provide default values for missing data

```
SELECT listing_id, COALESCE(price, 0) AS price
   FROM listings;
```

SQL Topology

0. SQL FUNDAMENTALS

0.1 SQL Syntax and Commands

0.1.1 SELECT and FROM Statements

0.1.2 Simple Filtering with WHERE

0.1.3 Sorting with ORDER BY

0.1.4 Limiting Results with LIMIT/OFFSET

0.1.5 INSERT, UPDATE, and DELETE Statements

0.1.6 TRUNCATE and COPY Statements

0.2 Introduction to Databases

0.2.1 What is SQL and Relational Databases?

0.2.2 Database Management Systems (DBMS) Overview

0.2.3 Query Execution Basics

0.2.4 Creating, Dropping, and Renaming Databases

0.3 Introduction to Tables

0.3.1 Creating, Dropping, Renaming, and Altering Tables

0.3.2 Temporary Tables

0.3.3 Sequences and Auto-Increment

1. Database Architecture and Objects

1.1.1 Schemas, Tables, Views, and Functions

1.1.2 Primary Keys, Foreign Keys, and Relationships

1.1.3 Indexes and Performance Optimization

1.1.4 Stored Procedures, Triggers, and Cursors

1.2 SQL Data Types and Constraints

1.2.1 Numeric, String, Date/Time, and Boolean Types

1.2.2 Data Type Conversions and Casting

15.2 Transactions and Concurrency

15.2.1 SQL Transactions

15.2.2 ACID Properties and Management

SQL Theory

SQL THEORY

Core Query Structure

CLAUSE	PURPOSE	INTERVIEW
`SELECT`	Specifies columns, aggregates, and calculated metrics. The backbone of all data retrieval.	...
`FROM`	Identifies the data source. Uses the structure: catalog.schema.table (e.g., hive.marketing.campaigns).	...
`WHERE`	Filters rows before aggregation or grouping.	...
`GROUP BY`	Aggregates rows based on non-aggregated columns.	Used for standard reporting metrics (sum of sales per region). Can be extended with ROLLUP or CUBE for multi-level aggregates.
`HAVING`	Filters the results after aggregation/grouping.	Used to filter groups (e.g., HAVING COUNT(user_id) > 100) **Rule**: WHERE filters rows, HAVING filters groups.
`WITH (CTE)`	Creates Common Table Expressions (CTEs) to name and define subqueries.	Mandatory for complex queries Improves readability and allows you to break down logic into reusable steps

LIMIT / OFFSET	Restricts the number of output rows.	Used for efficient testing, prototyping, and implementing pagination logic in applications.

Analytical Window Functions

FUNCTION	CATEGORY	EXAMPLE	ANALYTICS USE CASE
ROW_NUMBER()	Ranking	ROW_NUMBER() OVER (PARTITION BY user_id ORDER BY transaction_date DESC)	Finding the most recent (or first) record per group. No ties possible.
RANK()	Ranking	RANK() OVER (PARTITION BY dept ORDER BY salary DESC)	Assigns a rank. Ties get the same rank, and the next rank number is skipped (e.g., 1, 2, 2, 4).
DENSE_RANK()	Ranking	DENSE_RANK() OVER (PARTITION BY dept ORDER BY salary DESC)	Assigns a rank. Ties get the same rank, and the next rank number is not skipped (e.g., 1, 2, 2, 3).
LAG()	Value	LAG(sales_amount, 1) OVER (ORDER BY date)	Calculating period-over-period change, such as sales from the previous day, month, or quarter.
LEAD()	Value	LEAD(price, 1) OVER (PARTITION BY stock_id ORDER BY timestamp)	Looking ahead to the next value in a sequence (e.g., finding the next price increase).
SUM() / AVG()	Aggregate	SUM(sales) OVER (ORDER BY week ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)	Calculating running totals or cumulative sums over time.

Join Types

FUNCTION	CATEGORY	EXAMPLE
`INNER JOIN`	Returns only rows where the join key exists in both tables	Used for merging core data tables where a match is necessary. Generally the fastest join type
`LEFT JOIN`	Returns all rows from the left table and matched rows from the right table (filling in NULL for unmatched rows)	Used for data enrichment, such as joining a master list of products to a sparse table of reviews
`FULL OUTER JOIN`	Returns all rows from both tables, filling in NULLs for any non-matches	Used to find and analyze data discrepancies or missing records between two datasets
`CROSS JOIN`	Returns the Cartesian product (every row from the first table matched with every row from the second table)	Extremely rare in analytics; typically only used for generating scaffolding data (e.g., a complete date-product pair combination)

•••

...

```
-- Single condition
SELECT * FROM bookings
```

```
WHERE booking_date >= DATE '2024-01-01';

-- Multiple conditions
SELECT * FROM bookings
WHERE status = 'confirmed'
  AND total_price > 500
  AND guest_country = 'US';

-- IN operator
SELECT * FROM listings
WHERE city IN ('San Francisco', 'New York', 'Los Angeles');

-- LIKE pattern matching
SELECT * FROM listings
WHERE property_type LIKE '%Apartment%';

-- IS NULL / IS NOT NULL
SELECT * FROM reviews
WHERE response_date IS NULL;
```

Comprehensive SQL Guide for Analytics

SQL FUNDAMENTALS

SQL Syntax and Commands

SELECT and FROM Statements

SELECT retrieves data from tables, FROM specifies the source table(s).

```sql
SELECT column1, column2, ...
FROM table_name;
```

0.1.2 Simple Filtering with WHERE

Purpose: Filter rows based on conditions before any grouping occurs.

-- Basic WHERE clause

SELECT * FROM customers

WHERE country = 'Germany';

```
-- Multiple conditions with operators
SELECT product_name, price
FROM products
WHERE price > 10 AND category_id = 2;

-- Pattern matching with LIKE
SELECT * FROM customers
WHERE customer_name LIKE 'A%';  -- Starts with 'A'
```

Important Notes:

- Presto uses single quotes for string literals
- String comparisons are case-sensitive in Presto
- WHERE filters individual rows before GROUP BY

0.1.3 Sorting with ORDER BY

```
-- Single column sorting
SELECT * FROM products
ORDER BY price DESC;

-- Multiple column sorting
SELECT customer_name, country, city
FROM customers
ORDER BY country ASC, city DESC;

-- Presto supports NULLS FIRST/LAST
SELECT * FROM products
ORDER BY discount DESC NULLS LAST;
```

0.1.4 Limiting Results with LIMIT/OFFSET

```
-- Get top 10 results (Presto syntax)
SELECT * FROM orders
LIMIT 10;

-- Pagination: Skip 20, get next 10
SELECT * FROM orders
LIMIT 10 OFFSET 20;

-- Alternative syntax
SELECT * FROM orders
OFFSET 20 LIMIT 10;
```

0.1.5 INSERT, UPDATE, and DELETE Statements

Note: Presto is primarily for analytics (read-only). These operations are typically done in Hive or the underlying data store.

```
-- INSERT: Add new data (Hive syntax)
INSERT INTO customers VALUES ('Cardinal', 'Tom B. Erichsen', 'Norway');

-- INSERT with specific columns
INSERT INTO customers (customer_name, contact_name, country)
VALUES ('Cardinal', 'Tom B. Erichsen', 'Norway');

-- UPDATE and DELETE are not directly supported in Presto
-- Use Hive or recreate tables with CREATE TABLE AS
```

0.1.6 TRUNCATE and COPY Statements

```
-- In Presto, recreate table instead of TRUNCATE

DROP TABLE IF EXISTS temp_customers;

CREATE TABLE temp_customers AS SELECT * FROM customers WHERE 1=0;


-- Copy data between tables

CREATE TABLE new_table AS

SELECT * FROM old_table;
```

0.2 Introduction to Databases

0.2.1 What is SQL and Relational Databases?

- **SQL**: Structured Query Language for managing relational databases
- **Relational Database**: Stores data in structured tables with relationships
- **Key Benefits**:
    - Widely used across all organizations
    - Skills transferable between database systems
    - ACID properties ensure data consistency

0.2.2 Database Management Systems (DBMS) Overview

- **Presto/Trino**: Distributed SQL query engine for big data
- **Hive**: Data warehouse infrastructure built on Hadoop
- **OLTP vs OLAP**:
    - **OLTP**: High volume of short transactions, normalized schemas
    - **OLAP**: Complex queries on large datasets, denormalized schemas (Presto's domain)

0.2.3 Query Execution Basics

Logical Execution Order (Critical for understanding):

1. FROM / JOIN - Identifies and combines source tables
2. WHERE - Filters individual rows
3. GROUP BY - Aggregates rows into groups
4. HAVING - Filters aggregated groups
5. SELECT - Selects final columns and calculations
6. DISTINCT - Removes duplicates
7. ORDER BY - Sorts result set
8. LIMIT/OFFSET - Restricts rows returned

0.2.4 Creating, Dropping, and Renaming Databases

```
-- In Presto, work with schemas instead of databases
CREATE SCHEMA IF NOT EXISTS analytics;
DROP SCHEMA IF EXISTS test_schema;

-- Switch schema
USE analytics;
```

0.3 Introduction to Tables

0.3.1 Creating, Dropping, Renaming, and Altering Tables

```
-- Create table (Hive/Presto syntax)
CREATE TABLE IF NOT EXISTS employees (
    employee_id BIGINT,
    name VARCHAR,
    email VARCHAR,
    salary DECIMAL(10,2),
    hire_date DATE
) WITH (
    format = 'ORC',
    partitioned_by = ARRAY['hire_date']
);

-- Drop table
DROP TABLE IF EXISTS temp_employees;

-- Create table as select (CTAS)
CREATE TABLE high_value_customers AS
SELECT customer_id, SUM(amount) as total_spent
FROM orders
GROUP BY customer_id
HAVING SUM(amount) > 10000;

-- Presto doesn't support ALTER TABLE for structure changes
-- Recreate table with new structure instead
```

0.3.2 Temporary Tables

```
-- Presto uses WITH clauses (CTEs) instead of temp tables
WITH temp_calculations AS (
    SELECT customer_id, SUM(amount) as total_spent
    FROM orders
    GROUP BY customer_id
)
SELECT * FROM temp_calculations;
```

0.3.3 Sequences and Auto-Increment

```
-- Presto doesn't have auto-increment
-- Generate sequences using ROW_NUMBER()
SELECT
    ROW_NUMBER() OVER (ORDER BY created_at) as id,
    product_name
FROM products;
```

1. DATABASE ARCHITECTURE AND OBJECTS

1.1 Database Structure

1.1.1 Schemas, Tables, Views, and Functions

```
-- Create schema
CREATE SCHEMA IF NOT EXISTS marketing;

-- Create view
CREATE OR REPLACE VIEW customer_orders AS
SELECT c.customer_name, o.order_date, o.total_amount
FROM customers c
JOIN orders o ON c.customer_id = o.customer_id;

-- Presto doesn't support stored functions
-- Use inline calculations or CTEs instead
```

1.1.2 Primary Keys, Foreign Keys, and Relationships

```
-- Presto doesn't enforce constraints
-- Document relationships in table comments
COMMENT ON TABLE orders IS 'Orders table with customer_id as FK to customers.customer_id';
```

1.1.3 Indexes and Performance Optimization

```
-- Presto doesn't create indexes
-- Performance optimization through:
-- 1. Partitioning
-- 2. Bucketing
-- 3. File formats (ORC, Parquet)
-- 4. Statistics

-- Create partitioned table
CREATE TABLE events (
   event_id BIGINT,
   user_id BIGINT,
   event_type VARCHAR,
   event_time TIMESTAMP
) WITH (
   format = 'ORC',
   partitioned_by = ARRAY['event_date'],
   bucketed_by = ARRAY['user_id'],
   bucket_count = 50
);
```

1.2 SQL Data Types and Constraints

1.2.1 Numeric, String, Date/Time, and Boolean Types

Presto Data Types:

- **Numeric**: TINYINT, SMALLINT, INTEGER, BIGINT, REAL, DOUBLE, DECIMAL(p,s)
- **String**: VARCHAR, CHAR(n), VARBINARY, JSON
- **Date/Time**: DATE, TIME, TIMESTAMP, INTERVAL
- **Boolean**: BOOLEAN
- **Complex**: ARRAY, MAP, ROW

1.2.2 Data Type Conversions and Casting

```
-- Explicit casting in Presto
SELECT
    CAST('123' AS INTEGER) as int_value,
    CAST(order_date AS VARCHAR) as date_string,
    CAST(price AS DECIMAL(10,2)) as formatted_price,
    TRY_CAST('invalid' AS INTEGER) as safe_cast  -- Returns NULL instead of error
FROM orders;
```

2. DATA PREPARATION AND PROFILING

2.1 Exploratory Data Analysis with SQL

2.1.1 Data Profiling Techniques

```
-- Basic profiling query
SELECT
    COUNT(*) as total_rows,
    COUNT(DISTINCT customer_id) as unique_customers,
    MIN(order_date) as earliest_order,
    MAX(order_date) as latest_order,
    AVG(total_amount) as avg_order_value,
    APPROX_PERCENTILE(total_amount, 0.5) as median_order_value
FROM orders;

-- Check for nulls
SELECT
    COUNT(*) - COUNT(customer_id) as null_customers,
    COUNT(*) - COUNT(order_date) as null_dates
FROM orders;
```

2.1.2 Frequency Analysis and Distributions

```
-- Frequency distribution
SELECT
    status,
    COUNT(*) as frequency,
    COUNT(*) * 100.0 / SUM(COUNT(*)) OVER () as percentage
FROM orders
GROUP BY status
ORDER BY frequency DESC;

-- Histogram bins using WIDTH_BUCKET
SELECT
    WIDTH_BUCKET(price, 0, 100, 10) as price_bucket,
    COUNT(*) as count
```

```
FROM products

GROUP BY 1

ORDER BY 1;
```

2.1.3 Identifying Data Quality Issues

```
-- Find duplicates

SELECT customer_email, COUNT(*) as count

FROM customers

GROUP BY customer_email

HAVING COUNT(*) > 1;


-- Find orphaned records (anti-join pattern)

SELECT o.*

FROM orders o

LEFT JOIN customers c ON o.customer_id = c.customer_id

WHERE c.customer_id IS NULL;
```

2.2 Data Cleaning and Transformation

2.2.1 Handling Null Values and Missing Data

```
-- Replace nulls with default value using COALESCE

SELECT

    customer_id,

    COALESCE(phone, 'No Phone') as phone,

    COALESCE(email, 'No Email') as email,

    IF(address IS NULL, 'No Address', address) as address
```

```
FROM customers;


-- Filter out nulls
SELECT * FROM orders
WHERE ship_date IS NOT NULL;


-- Use NULLIF to handle specific values as NULL
SELECT
    revenue,
    expenses,
    revenue / NULLIF(expenses, 0) as profit_ratio  -- Prevents division by zero
FROM financial_data;
```

2.2.2 String Manipulation and Text Processing

```
-- Common string functions in Presto
SELECT
    UPPER(customer_name) as uppercase_name,
    LOWER(email) as lowercase_email,
    TRIM(address) as trimmed_address,
    SUBSTR(phone, 1, 3) as area_code,
    CONCAT(first_name, ' ', last_name) as full_name,
    first_name || ' ' || last_name as full_name_alt,
    REPLACE(description, 'old', 'new') as updated_desc,
    SPLIT(tags, ',') as tag_array,
    LENGTH(description) as desc_length
```

```
FROM customers;


-- Regular expressions in Presto

SELECT * FROM customers

WHERE REGEXP_LIKE(email, '^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}$');


-- Extract with regex

SELECT

    REGEXP_EXTRACT(url, 'utm_source=([^&]+)') as utm_source,

    REGEXP_EXTRACT_ALL(text, '\d+') as all_numbers

FROM marketing_data;
```

2.2.3 Date and Time Manipulations

```
-- Date functions in Presto

SELECT

    CURRENT_DATE as today,

    CURRENT_TIMESTAMP as right_now,

    DATE_ADD('day', 7, order_date) as week_later,

    DATE_DIFF('day', order_date, ship_date) as days_to_ship,

    EXTRACT(YEAR FROM order_date) as order_year,

    EXTRACT(MONTH FROM order_date) as order_month,

    DATE_TRUNC('month', order_date) as month_start,

    DATE_FORMAT(order_date, '%Y-%m') as year_month,

    order_date + INTERVAL '30' DAY as payment_due

FROM orders;
```

```
-- Working with timestamps
SELECT
    FROM_UNIXTIME(unix_timestamp) as timestamp_value,
    TO_UNIXTIME(timestamp_col) as unix_value
FROM events;
```

2.3 Data Shaping and Restructuring

2.3.1 Pivoting and Unpivoting Data

```
-- Pivot using conditional aggregation (Presto doesn't have PIVOT)
SELECT
    customer_id,
    SUM(CASE WHEN YEAR(order_date) = 2023 THEN revenue END) as revenue_2023,
    SUM(CASE WHEN YEAR(order_date) = 2024 THEN revenue END) as revenue_2024,
    SUM(CASE WHEN YEAR(order_date) = 2025 THEN revenue END) as revenue_2025
FROM sales
GROUP BY customer_id;

-- Unpivot using CROSS JOIN with VALUES
SELECT customer_id, year, revenue
FROM sales
CROSS JOIN UNNEST(
    ARRAY[2023, 2024, 2025],
    ARRAY[revenue_2023, revenue_2024, revenue_2025]
) AS t(year, revenue);
```

3. CORE SQL FOR ANALYTICS

3.1 Advanced SELECT Operations

3.1.1 Complex WHERE Clauses and Filtering

```
-- Complex conditions

SELECT * FROM orders

WHERE (status = 'Shipped' OR status = 'Delivered')

  AND order_date >= DATE '2024-01-01'

  AND total_amount > 100

  AND customer_id IN (

      SELECT customer_id FROM customers

      WHERE country = 'USA'

  );
```

3.1.2 Pattern Matching with LIKE and Regular Expressions

```
-- LIKE patterns

SELECT * FROM products

WHERE product_name LIKE 'Ch%'      -- Starts with Ch

   OR product_name LIKE '%ese'     -- Ends with ese

   OR product_name LIKE '%ee%';    -- Contains ee


-- Regular expressions in Presto

SELECT * FROM emails

WHERE REGEXP_LIKE(email_address, '^[a-zA-Z0-9._%+-]+@airbnb\.com$');
```

3.1.3 Subqueries and Correlated Subqueries

```
-- Non-correlated subquery

SELECT * FROM employees

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


-- Correlated subquery

SELECT e1.name, e1.salary

FROM employees e1

WHERE e1.salary > (

   SELECT AVG(e2.salary)

   FROM employees e2

   WHERE e2.department = e1.department

);
```

3.1.5 Operators (IN, BETWEEN, EXISTS, ALL, ANY, NOT, IS NULL)

```
-- Various operators in Presto

SELECT * FROM products

WHERE price BETWEEN 10 AND 50

  AND category_id IN (1, 2, 3)

  AND EXISTS (

      SELECT 1 FROM order_details

      WHERE order_details.product_id = products.product_id

  )

  AND price > ALL (SELECT price FROM products WHERE category_id = 4)
  AND discontinued IS NOT NULL;

-- ANY/SOME operator
SELECT * FROM products
WHERE price > ANY (SELECT price FROM products WHERE category_id = 2);
```

3.1.6 Set Operations (UNION, UNION ALL, INTERSECT, EXCEPT)

```
-- UNION removes duplicates
SELECT country FROM customers
UNION
SELECT country FROM suppliers;

-- UNION ALL keeps duplicates (faster)
SELECT city FROM customers
UNION ALL
SELECT city FROM suppliers;

-- INTERSECT: rows in both
SELECT product_id FROM orders_2023
INTERSECT
SELECT product_id FROM orders_2024;

-- EXCEPT: rows in first but not second
SELECT customer_id FROM all_customers
EXCEPT
```

```
SELECT customer_id FROM inactive_customers;
```

3.2 JOIN Strategies for Analysis

3.2.1 Inner, Left, Right, and Full Outer Joins

```
-- INNER JOIN: Only matching records
SELECT c.customer_name, o.order_date
FROM customers c
INNER JOIN orders o ON c.customer_id = o.customer_id;

-- LEFT JOIN: All from left, matching from right
SELECT c.customer_name, o.order_date
FROM customers c
LEFT JOIN orders o ON c.customer_id = o.customer_id;

-- RIGHT JOIN: All from right, matching from left
SELECT c.customer_name, o.order_date
FROM customers c
RIGHT JOIN orders o ON c.customer_id = o.customer_id;

-- FULL OUTER JOIN: All from both
SELECT c.customer_name, o.order_date
FROM customers c
FULL OUTER JOIN orders o ON c.customer_id = o.customer_id;
```

3.2.2 Self-Joins and Cross Joins

```
-- Self-join: Find employees with same salary
SELECT e1.name as employee1, e2.name as employee2, e1.salary
FROM employees e1
JOIN employees e2 ON e1.salary = e2.salary
WHERE e1.employee_id < e2.employee_id;

-- Cross join: Cartesian product
SELECT p.product_name, c.category_name
FROM products p
CROSS JOIN categories c;

-- Alternative syntax
SELECT p.product_name, c.category_name
FROM products p, categories c;
```

3.2.3 Complex Multi-Table Joins

```
-- Join multiple tables
SELECT
    c.customer_name,
    o.order_date,
    od.quantity,
    p.product_name,
    s.company_name as supplier
FROM customers c
JOIN orders o ON c.customer_id = o.customer_id
JOIN order_details od ON o.order_id = od.order_id
JOIN products p ON od.product_id = p.product_id
JOIN suppliers s ON p.supplier_id = s.supplier_id
WHERE o.order_date >= DATE '2024-01-01';
```

3.2.4 Semi-Joins and Anti-Joins

```
-- Semi-join using EXISTS
SELECT * FROM customers c
WHERE EXISTS (
    SELECT 1 FROM orders o
    WHERE o.customer_id = c.customer_id
);

-- Semi-join using IN
SELECT * FROM customers
WHERE customer_id IN (SELECT customer_id FROM orders);

-- Anti-join: Customers with no orders
SELECT c.*
FROM customers c
LEFT JOIN orders o ON c.customer_id = o.customer_id
WHERE o.customer_id IS NULL;

-- Anti-join using NOT EXISTS
SELECT * FROM customers c
```

```
WHERE NOT EXISTS (
    SELECT 1 FROM orders o
    WHERE o.customer_id = c.customer_id
);
```

3.3 Aggregation and Grouping

3.3.1 GROUP BY and HAVING Clauses

```
-- GROUP BY with HAVING
SELECT
    department,
    COUNT(*) as employee_count,
    AVG(salary) as avg_salary
FROM employees
WHERE status = 'Active'
GROUP BY department
HAVING COUNT(*) > 5
   AND AVG(salary) > 50000
ORDER BY avg_salary DESC;
```

Key Difference: WHERE filters rows before grouping, HAVING filters groups after aggregation

3.3.2 Aggregate Functions (COUNT, SUM, AVG, MIN, MAX)

```
SELECT
    COUNT(*) as total_orders,
    COUNT(DISTINCT customer_id) as unique_customers,
    SUM(total_amount) as total_revenue,
    AVG(total_amount) as avg_order_value,
    MIN(order_date) as first_order,
    MAX(order_date) as last_order,
    ARBITRARY(status) as sample_status,  -- Presto-specific
    BOOL_AND(is_shipped) as all_shipped,  -- Presto-specific
    BOOL_OR(is_expedited) as any_expedited  -- Presto-specific
FROM orders
WHERE status = 'Completed';
```

3.3.3 Grouping Sets, Rollup, and Cube Operations

```
-- GROUPING SETS in Presto
SELECT
    product,
    region,
    year,
    SUM(sales) as total,
    GROUPING(product, region, year) as grouping_id
FROM sales_data
GROUP BY GROUPING SETS (
    (product, region),
    (product, year),
    (region, year),
    ()  -- Grand total
);
```

```
-- ROLLUP
SELECT
    year,
    quarter,
    month,
    SUM(revenue) as total
FROM sales
GROUP BY ROLLUP (year, quarter, month);

-- CUBE
SELECT
    product,
    region,
    SUM(sales) as total
FROM sales_data
GROUP BY CUBE (product, region);
```

3.3.4 Median and Percentiles

```
-- Percentiles in Presto
SELECT
    APPROX_PERCENTILE(salary, 0.5) as median_salary,
    APPROX_PERCENTILE(salary, 0.25) as q1,
    APPROX_PERCENTILE(salary, 0.75) as q3,
    APPROX_PERCENTILE(salary, ARRAY[0.25, 0.5, 0.75, 0.95]) as percentiles
FROM employees;
```

4. ADVANCED ANALYTICAL FUNCTIONS

4.1 Window Functions for Analytics

4.1.1 ROW_NUMBER, RANK, and DENSE_RANK

```
-- Ranking functions in Presto
SELECT
    department,
    employee_name,
    salary,
    ROW_NUMBER() OVER (PARTITION BY department ORDER BY salary DESC) as row_num,
    RANK() OVER (PARTITION BY department ORDER BY salary DESC) as rank_num,
    DENSE_RANK() OVER (PARTITION BY department ORDER BY salary DESC) as dense_rank,
    PERCENT_RANK() OVER (PARTITION BY department ORDER BY salary DESC) as percent_rank
FROM employees;
```

4.1.2 LAG and LEAD for Time-Based Analysis

```
-- Period-over-period comparison
SELECT
    month,
    revenue,
    LAG(revenue, 1) OVER (ORDER BY month) as prev_month,
    LAG(revenue, 12) OVER (ORDER BY month) as same_month_last_year,
    LEAD(revenue, 1) OVER (ORDER BY month) as next_month,
    revenue - LAG(revenue, 1) OVER (ORDER BY month) as mom_change,
    (revenue - LAG(revenue, 1) OVER (ORDER BY month)) * 100.0 /
        NULLIF(LAG(revenue, 1) OVER (ORDER BY month), 0) as mom_growth_pct
FROM monthly_revenue;

-- Sessionization with LAG
SELECT
    user_id,
    event_time,
    LAG(event_time) OVER (PARTITION BY user_id ORDER BY event_time) as prev_event,
    CASE
        WHEN DATE_DIFF('minute',
            LAG(event_time) OVER (PARTITION BY user_id ORDER BY event_time),
            event_time) > 30
        THEN 1
        ELSE 0
    END as new_session
FROM user_events;
```

4.1.3 Cumulative and Moving Aggregations

```
-- Running total in Presto
SELECT
    order_date,
    revenue,
    SUM(revenue) OVER (
        ORDER BY order_date
        ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
    ) as running_total,
    SUM(revenue) OVER (
        ORDER BY order_date
        RANGE BETWEEN INTERVAL '30' DAY PRECEDING AND CURRENT ROW
    ) as rolling_30_day
FROM daily_revenue;

-- Moving average
SELECT
    date,
    sales,
    AVG(sales) OVER (
        ORDER BY date
        ROWS BETWEEN 6 PRECEDING AND CURRENT ROW
    ) as moving_avg_7d,
    AVG(sales) OVER (
        ORDER BY date
        ROWS BETWEEN 29 PRECEDING AND CURRENT ROW
    ) as moving_avg_30d
FROM daily_sales;
```

4.1.4 NTILE for Bucketing

```
-- Divide customers into quartiles
SELECT
```

```
    customer_id,
    total_purchases,
    NTILE(4) OVER (ORDER BY total_purchases DESC) as quartile,
    NTILE(10) OVER (ORDER BY total_purchases DESC) as decile,
    NTILE(100) OVER (ORDER BY total_purchases DESC) as percentile
FROM customer_spending;
```

4.2 Common Table Expressions (CTEs)

4.2.1 Basic CTEs for Query Organization

```
-- Simple CTE
WITH high_value_customers AS (
    SELECT customer_id, SUM(total_amount) as total_spent
    FROM orders
    GROUP BY customer_id
    HAVING SUM(total_amount) > 10000
)
SELECT c.customer_name, h.total_spent
FROM high_value_customers h
JOIN customers c ON h.customer_id = c.customer_id;

-- Multiple CTEs
WITH
customer_totals AS (
    SELECT customer_id, SUM(amount) as total
    FROM orders
    GROUP BY customer_id
),
customer_ranks AS (
    SELECT
        customer_id,
        total,
        DENSE_RANK() OVER (ORDER BY total DESC) as rank
    FROM customer_totals
)
SELECT * FROM customer_ranks WHERE rank <= 10;
```

4.2.2 Recursive CTEs

```
-- Note: Presto has limited recursive CTE support
-- Alternative: Use iterative approach or external processing
-- Example of hierarchical data traversal without recursion
WITH employee_hierarchy AS (
    SELECT
        e1.employee_id,
        e1.name,
        e1.manager_id,
        e2.name as manager_name,
        e3.name as skip_manager_name
    FROM employees e1
    LEFT JOIN employees e2 ON e1.manager_id = e2.employee_id
    LEFT JOIN employees e3 ON e2.manager_id = e3.employee_id
)
SELECT * FROM employee_hierarchy;
```

4.3 CASE Statements and Conditional Logic

4.3.1 Simple and Searched CASE Expressions

```
-- Simple CASE
SELECT
    order_id,
    status,
    CASE status
        WHEN 'Pending' THEN 'Awaiting Processing'
        WHEN 'Shipped' THEN 'In Transit'
        WHEN 'Delivered' THEN 'Completed'
        ELSE 'Unknown'
    END as status_description
FROM orders;

-- Searched CASE
SELECT
    customer_id,
    total_purchases,
    CASE
        WHEN total_purchases >= 10000 THEN 'Platinum'
        WHEN total_purchases >= 5000 THEN 'Gold'
        WHEN total_purchases >= 1000 THEN 'Silver'
        ELSE 'Bronze'
    END as customer_tier
FROM customer_summary;
```

4.3.2 Data Categorization and Binning

```
-- Age group binning
SELECT
    customer_id,
    age,
    CASE
        WHEN age < 18 THEN 'Under 18'
        WHEN age BETWEEN 18 AND 24 THEN '18-24'
        WHEN age BETWEEN 25 AND 34 THEN '25-34'
        WHEN age BETWEEN 35 AND 44 THEN '35-44'
        WHEN age BETWEEN 45 AND 54 THEN '45-54'
        WHEN age >= 55 THEN '55+'
    END as age_group
FROM customers;

-- Using WIDTH_BUCKET for automatic binning
SELECT
    customer_id,
    purchase_amount,
    WIDTH_BUCKET(purchase_amount, 0, 1000, 10) as amount_bucket
FROM purchases;
```

4.3.4 Conditional Functions (IF, COALESCE, NULLIF)

```
-- COALESCE: First non-null value
SELECT
    customer_id,
    COALESCE(email, phone, 'No Contact') as primary_contact
FROM customers;


-- NULLIF: Returns NULL if expressions equal
SELECT
    revenue,
    expenses,
    revenue / NULLIF(expenses, 0) as profit_ratio  -- Prevents division by zero
FROM financial_data;


-- IF function in Presto
SELECT
    customer_id,
    IF(total_purchases > 5000, 'VIP', 'Regular') as customer_type,
    IF(age >= 18, 'Adult', 'Minor') as age_category
FROM customer_summary;


-- TRY for safe operations
SELECT
    TRY(CAST(string_col AS INTEGER)) as safe_int,
    TRY(date_col / amount) as safe_division
FROM data_table;
```

5. TIME SERIES ANALYSIS WITH SQL

5.1 Date and Time Functions

5.1.1 Date Arithmetic and Calculations

```
-- Date arithmetic in Presto
SELECT
    order_date,
    ship_date,
    DATE_DIFF('day', order_date, ship_date) as days_to_ship,
    order_date + INTERVAL '30' DAY as payment_due,
    order_date - INTERVAL '1' YEAR as year_ago,
    DATE_ADD('month', 3, order_date) as quarter_later,
    DATE_ADD('week', -2, order_date) as two_weeks_before
FROM orders;
```

5.1.2 Date Parts Extraction and Formatting

```
-- Extract date parts in Presto
SELECT
    order_date,
    YEAR(order_date) as year,
    MONTH(order_date) as month,
    DAY(order_date) as day,
    DAY_OF_WEEK(order_date) as day_of_week,  -- 1=Monday, 7=Sunday
    DAY_OF_YEAR(order_date) as day_of_year,
    WEEK(order_date) as week_number,
    QUARTER(order_date) as quarter,
    DATE_FORMAT(order_date, '%Y-%m') as year_month,
    DATE_FORMAT(order_date, '%Y-W%v') as year_week_iso,
    DATE_TRUNC('month', order_date) as month_start,
    DATE_TRUNC('week', order_date) as week_start
FROM orders;
```

5.1.3 Time Zone Handling

```
-- Time zone operations in Presto
SELECT
    AT_TIMEZONE(event_timestamp, 'America/Los_Angeles') as la_time,
    AT_TIMEZONE(event_timestamp, 'UTC') as utc_time,
    WITH_TIMEZONE(event_timestamp, 'America/New_York') as ny_time,
    CURRENT_TIMEZONE() as session_timezone
FROM events;
```

5.2 Temporal Analytics

5.2.1 Period-over-Period Comparisons

```
-- Year-over-Year comparison
WITH current_year AS (
    SELECT
        DATE_TRUNC('month', order_date) as month,
        SUM(revenue) as revenue
    FROM orders
    WHERE YEAR(order_date) = 2024
    GROUP BY 1
),
previous_year AS (
    SELECT
        DATE_TRUNC('month', order_date + INTERVAL '1' YEAR) as month,
        SUM(revenue) as revenue
    FROM orders
    WHERE YEAR(order_date) = 2023
    GROUP BY 1
)
SELECT
    c.month,
    c.revenue as current_revenue,
    p.revenue as previous_revenue,
    (c.revenue - p.revenue) * 100.0 / NULLIF(p.revenue, 0) as yoy_growth
FROM current_year c
LEFT JOIN previous_year p ON c.month = p.month
ORDER BY c.month;
```

5.2.2 Rolling Time Windows and Moving Averages

```
-- 30-day rolling revenue
SELECT
```

```
    date,
    daily_revenue,
    SUM(daily_revenue) OVER (
        ORDER BY date
        RANGE BETWEEN INTERVAL '29' DAY PRECEDING AND CURRENT ROW
    ) as rolling_30_day
FROM daily_revenue;


-- Week-over-week with proper date handling
WITH weekly_metrics AS (
    SELECT
        DATE_TRUNC('week', date) as week,
        SUM(revenue) as weekly_revenue
    FROM daily_revenue
    GROUP BY 1
)
SELECT
    week,
    weekly_revenue,
    LAG(weekly_revenue, 1) OVER (ORDER BY week) as prev_week,
    (weekly_revenue - LAG(weekly_revenue, 1) OVER (ORDER BY week)) * 100.0 /
        NULLIF(LAG(weekly_revenue, 1) OVER (ORDER BY week), 0) as wow_growth
FROM weekly_metrics;
```

6. COHORT AND RETENTION ANALYSIS

6.1 Cohort Analysis Framework

6.1.1 Defining Cohorts and Time Series

```
-- Define user cohorts by signup month
WITH user_cohorts AS (
    SELECT
        user_id,
        DATE_TRUNC('month', MIN(signup_date)) as cohort_month
    FROM users
    GROUP BY user_id
)
SELECT
    cohort_month,
    COUNT(DISTINCT user_id) as cohort_size
FROM user_cohorts
GROUP BY cohort_month
ORDER BY cohort_month;
```

6.2 Retention Analytics

6.2.1 Basic Retention Curves

```
-- Classic retention analysis in Presto
WITH cohorts AS (
    SELECT
        user_id,
        DATE_TRUNC('month', MIN(event_date)) as cohort_month
    FROM user_events
    GROUP BY user_id
),
activities AS (
    SELECT
        c.cohort_month,
        DATE_DIFF('month', c.cohort_month, DATE_TRUNC('month', e.event_date)) as months_since,
        COUNT(DISTINCT e.user_id) as active_users
    FROM user_events e
    JOIN cohorts c ON e.user_id = c.user_id
    GROUP BY 1, 2
),
cohort_sizes AS (
    SELECT
        cohort_month,
        COUNT(DISTINCT user_id) as cohort_size
    FROM cohorts
    GROUP BY cohort_month
)
SELECT
    a.cohort_month,
    a.months_since,
    a.active_users,
    cs.cohort_size,
    a.active_users * 100.0 / cs.cohort_size as retention_rate
FROM activities a
```

```
JOIN cohort_sizes cs ON a.cohort_month = cs.cohort_month
ORDER BY a.cohort_month, a.months_since;
```

6.3 Advanced Cohort Metrics

6.3.3 Customer Lifetime Value (CLTV) Calculations

```
-- Simple LTV calculation
WITH customer_revenue AS (
    SELECT
        customer_id,
        SUM(revenue) as total_revenue,
        COUNT(DISTINCT DATE_TRUNC('month', order_date)) as active_months,
        MIN(order_date) as first_order,
        MAX(order_date) as last_order,
        DATE_DIFF('day', MIN(order_date), MAX(order_date)) as customer_lifespan_days
    FROM orders
    GROUP BY customer_id
)
SELECT
    AVG(total_revenue) as avg_ltv,
    AVG(total_revenue / NULLIF(active_months, 0)) as avg_monthly_value,
    AVG(customer_lifespan_days) as avg_customer_lifespan,
    APPROX_PERCENTILE(total_revenue, 0.5) as median_ltv
FROM customer_revenue;
```

7. BUSINESS INTELLIGENCE AND KPI DEVELOPMENT

7.1 KPI Framework Design

7.1.1 Metric Definitions and Calculations

```
-- Core business metrics with Presto
WITH metrics AS (
    SELECT
        DATE_TRUNC('day', date) as day,
        COUNT(DISTINCT user_id) as dau,
        COUNT(DISTINCT CASE WHEN event_type = 'purchase' THEN user_id END) as purchasers,
        SUM(CASE WHEN event_type = 'purchase' THEN revenue ELSE 0 END) as revenue,
        COUNT(CASE WHEN event_type = 'purchase' THEN 1 END) as transactions
    FROM events
    GROUP BY 1
)
SELECT
    day,
    dau,
    purchasers,
    revenue,
    transactions,
    purchasers * 100.0 / NULLIF(dau, 0) as conversion_rate,
    revenue / NULLIF(transactions, 0) as aov,
    revenue / NULLIF(purchasers, 0) as arpu
FROM metrics;
```

7.2 Business Metrics and Analytics

7.2.1 Sales and Revenue Analysis

```
-- Revenue breakdown with Presto

SELECT

    product_category,

    region,

    COUNT(DISTINCT order_id) as orders,

    SUM(quantity) as units_sold,

    SUM(revenue) as total_revenue,

    AVG(revenue) as avg_order_value,

    APPROX_PERCENTILE(revenue, 0.5) as median_order_value

FROM sales

WHERE order_date >= CURRENT_DATE - INTERVAL '30' DAY

GROUP BY product_category, region

ORDER BY total_revenue DESC;
```

7.2.2 Customer Analytics and Segmentation

```
-- RFM (Recency, Frequency, Monetary) Analysis in Presto

WITH rfm AS (

    SELECT

        customer_id,

        DATE_DIFF('day', MAX(order_date), CURRENT_DATE) as recency,

        COUNT(DISTINCT order_id) as frequency,

        SUM(revenue) as monetary
    FROM orders
    WHERE order_date >= CURRENT_DATE - INTERVAL '1' YEAR
    GROUP BY customer_id
),
rfm_scores AS (
    SELECT
        customer_id,
        NTILE(5) OVER (ORDER BY recency) as r_score,
        NTILE(5) OVER (ORDER BY frequency DESC) as f_score,
        NTILE(5) OVER (ORDER BY monetary DESC) as m_score
    FROM rfm
)
SELECT
    customer_id,
    CAST(r_score AS VARCHAR) || CAST(f_score AS VARCHAR) || CAST(m_score AS VARCHAR) as rfm_segment,
    CASE
        WHEN r_score >= 4 AND f_score >= 4 AND m_score >= 4 THEN 'Champions'
        WHEN r_score >= 3 AND f_score >= 3 AND m_score >= 3 THEN 'Loyal Customers'
        WHEN r_score >= 4 AND f_score <= 2 THEN 'New Customers'
        WHEN r_score <= 2 AND f_score >= 3 THEN 'At Risk'
        ELSE 'Other'
    END as segment
FROM rfm_scores;
```

7.2.3 Marketing Attribution and Campaign Analysis

```sql
-- Multi-touch attribution (first-click within 7 days) in Presto
WITH first_touch AS (
    SELECT
        user_id,
        campaign_id,
        MIN(touch_time) as first_touch_time
    FROM marketing_touches
    GROUP BY user_id, campaign_id
),
conversions AS (
    SELECT
        c.user_id,
        c.conversion_time,
        c.revenue,
        ft.campaign_id,
        ft.first_touch_time
    FROM conversions c
    JOIN first_touch ft ON c.user_id = ft.user_id
    WHERE c.conversion_time BETWEEN ft.first_touch_time
        AND ft.first_touch_time + INTERVAL '7' DAY
)
SELECT
    campaign_id,
    COUNT(DISTINCT user_id) as attributed_conversions,
    SUM(revenue) as attributed_revenue,
    AVG(revenue) as avg_conversion_value
```

```
FROM conversions

GROUP BY campaign_id

ORDER BY attributed_revenue DESC;
```

8. ADVANCED ANALYTICAL TECHNIQUES

8.1 Statistical Analysis with SQL

8.1.1 Descriptive Statistics and Percentiles

```
-- Comprehensive statistics in Presto

SELECT

    COUNT(*) as count,

    AVG(value) as mean,

    STDDEV(value) as std_dev,

    VARIANCE(value) as variance,

    MIN(value) as min,

    MAX(value) as max,

    APPROX_PERCENTILE(value, 0.25) as q1,

    APPROX_PERCENTILE(value, 0.50) as median,

    APPROX_PERCENTILE(value, 0.75) as q3,

    APPROX_PERCENTILE(value, ARRAY[0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]) as deciles

FROM data_points;
```

8.1.3 Outlier Detection and Anomaly Analysis

```
-- Z-score based outlier detection in Presto

WITH stats AS (
    SELECT
        AVG(value) as mean,
        STDDEV(value) as std_dev
    FROM transactions
),
z_scores AS (
    SELECT
        transaction_id,
        value,
        (value - s.mean) / NULLIF(s.std_dev, 0) as z_score
    FROM transactions t
    CROSS JOIN stats s
)
SELECT * FROM z_scores
WHERE ABS(z_score) > 3;  -- Outliers beyond 3 standard deviations
```

8.3 Experiment Analysis and A/B Testing

8.3.2 Control and Treatment Group Analysis

```
-- A/B test analysis with Presto
WITH experiment_results AS (
    SELECT
        variant_group,
        COUNT(DISTINCT user_id) as users,
        COUNT(DISTINCT CASE WHEN converted = 1 THEN user_id END) as converters,
        SUM(revenue) as total_revenue
    FROM ab_test_data
    WHERE experiment_id = 'exp_123'
    GROUP BY variant_group
)
SELECT
    variant_group,
    users,
    converters,
    converters * 100.0 / NULLIF(users, 0) as conversion_rate,
    total_revenue / NULLIF(users, 0) as revenue_per_user,
    total_revenue / NULLIF(converters, 0) as revenue_per_converter
FROM experiment_results
ORDER BY variant_group;
```

9. DATA MODELING FOR ANALYTICS

9.1 Dimensional Modeling Concepts

9.1.1 Star Schema and Snowflake Schema

```
-- Creating fact and dimension tables in Presto/Hive
CREATE TABLE IF NOT EXISTS fact_sales (
    sale_id BIGINT,
    date_key INTEGER,
    product_key INTEGER,
    customer_key INTEGER,
    quantity INTEGER,
    revenue DECIMAL(10,2)
) WITH (
    format = 'ORC',
    partitioned_by = ARRAY['date_key']
);

CREATE TABLE IF NOT EXISTS dim_product (
    product_key INTEGER,
    product_id VARCHAR,
    product_name VARCHAR,
    category VARCHAR,
    subcategory VARCHAR
) WITH (
    format = 'ORC'
);
```

10. PERFORMANCE OPTIMIZATION AND SCALABILITY

10.1 Query Performance Tuning

10.1.1 Execution Plan Analysis

```
-- View execution plan in Presto
EXPLAIN (TYPE DISTRIBUTED)
SELECT c.customer_name, SUM(o.total_amount)
FROM customers c
JOIN orders o ON c.customer_id = o.customer_id
GROUP BY c.customer_name;
```

```
-- Analyze query with statistics

EXPLAIN ANALYZE

SELECT * FROM large_table

WHERE partition_column = '2024-01-01';
```

10.1.2 Query Optimization Techniques

Presto-Specific Best Practices:

1. Use partition predicates to limit data scanned
2. Filter before joining when possible
3. Use appropriate file formats (ORC, Parquet)
4. Leverage bucketing for joins
5. Use APPROX functions for large aggregations

```
-- Optimized query with partition pruning

SELECT *

FROM events

WHERE event_date = DATE '2024-01-01'  -- Partition predicate

  AND user_id IN (SELECT user_id FROM active_users);


-- Use APPROX functions for performance

SELECT

    APPROX_DISTINCT(user_id) as unique_users,

    APPROX_PERCENTILE(revenue, 0.5) as median_revenue

FROM large_table;
```

10.2 Big Data SQL Techniques

10.2.2 Partitioning and Bucketing Strategies

```
-- Create partitioned and bucketed table
CREATE TABLE events_optimized
WITH (
    format = 'ORC',
    partitioned_by = ARRAY['event_date'],
    bucketed_by = ARRAY['user_id'],
    bucket_count = 100
)
AS SELECT * FROM events;

-- Query with partition pruning
SELECT COUNT(*)
FROM events_optimized
WHERE event_date BETWEEN DATE '2024-01-01' AND DATE '2024-01-31';
```

11. MACHINE LEARNING DATASET PREPARATION

11.1 Feature Engineering with SQL

11.1.1 Creating Model-Ready Datasets

```
-- Create features for ML model in Presto
WITH user_features AS (
    SELECT
        user_id,
        COUNT(DISTINCT order_id) as total_orders,
        AVG(order_value) as avg_order_value,
        MAX(order_date) as last_order_date,
        DATE_DIFF('day', MIN(order_date), MAX(order_date)) as customer_lifespan,
        SUM(CASE WHEN product_category = 'Electronics' THEN 1 ELSE 0 END) as electronics_orders,
        STDDEV(order_value) as order_value_variance
    FROM orders
    GROUP BY user_id
)
SELECT
    uf.*,
    CASE WHEN total_orders > 10 THEN 1 ELSE 0 END as high_value_customer  -- Target
FROM user_features uf;
```

12. ADVANCED SQL PATTERNS FOR ANALYSTS

12.2 Funnel Analysis and User Journey

12.2.1 Multi-step Process Analysis

```
-- E-commerce funnel analysis with Presto
WITH funnel_steps AS (
    SELECT
        user_id,
        MAX(CASE WHEN event_type = 'view_product' THEN 1 ELSE 0 END) as view_product,
        MAX(CASE WHEN event_type = 'add_to_cart' THEN 1 ELSE 0 END) as add_to_cart,
        MAX(CASE WHEN event_type = 'checkout' THEN 1 ELSE 0 END) as checkout,
        MAX(CASE WHEN event_type = 'purchase' THEN 1 ELSE 0 END) as purchase
    FROM user_events
    WHERE event_date >= CURRENT_DATE - INTERVAL '30' DAY
    GROUP BY user_id
)
SELECT
    COUNT(*) as total_users,
    SUM(view_product) as viewed_product,
    SUM(view_product * add_to_cart) as added_to_cart,
    SUM(view_product * add_to_cart * checkout) as checked_out,
    SUM(view_product * add_to_cart * checkout * purchase) as purchased,
    -- Conversion rates
    SUM(view_product * add_to_cart) * 100.0 / NULLIF(SUM(view_product), 0) as view_to_cart_rate,
    SUM(view_product * add_to_cart * checkout) * 100.0 /
        NULLIF(SUM(view_product * add_to_cart), 0) as cart_to_checkout_rate,
    SUM(view_product * add_to_cart * checkout * purchase) * 100.0 /
        NULLIF(SUM(view_product * add_to_cart * checkout), 0) as checkout_to_purchase_rate
FROM funnel_steps;
```

12.4 Top-N Queries and Rankings

```
-- Top 3 products per category with tie handling
WITH ranked_products AS (
    SELECT
        category,
        product_name,
        revenue,
        DENSE_RANK() OVER (PARTITION BY category ORDER BY revenue DESC) as rank
    FROM product_sales
)
SELECT * FROM ranked_products
WHERE rank <= 3;

-- Using array_agg for top-n collection
SELECT
    category,
    SLICE(ARRAY_AGG(product_name ORDER BY revenue DESC), 1, 3) as top_3_products
FROM product_sales
GROUP BY category;
```

Key Presto/Trino SQL Patterns and Solutions

Pattern 1: Finding Duplicates

```
-- Method 1: GROUP BY with HAVING
SELECT email, COUNT(*) as duplicate_count
FROM users
GROUP BY email
HAVING COUNT(*) > 1;

-- Method 2: Window Function
WITH duplicates AS (
    SELECT *, ROW_NUMBER() OVER (PARTITION BY email ORDER BY user_id) as rn
    FROM users
)
SELECT * FROM duplicates WHERE rn > 1;
```

Pattern 2: Nth Highest Value

```
-- Method 1: DENSE_RANK
WITH ranked_salaries AS (
    SELECT salary, DENSE_RANK() OVER (ORDER BY salary DESC) as rank
    FROM employees
)
SELECT DISTINCT salary FROM ranked_salaries WHERE rank = 3;

-- Method 2: Using OFFSET
SELECT DISTINCT salary
FROM employees
ORDER BY salary DESC
LIMIT 1 OFFSET 2;
```

Pattern 3: Consecutive Events

```
-- Find users with 3+ consecutive days of activity using Presto
WITH daily_activity AS (
    SELECT
        user_id,
        activity_date,
        activity_date - INTERVAL '1' DAY * ROW_NUMBER() OVER (
            PARTITION BY user_id ORDER BY activity_date
        ) as group_id
    FROM user_activities
),
consecutive_groups AS (
    SELECT
        user_id,
        MIN(activity_date) as start_date,
        MAX(activity_date) as end_date,
        COUNT(*) as consecutive_days
    FROM daily_activity
    GROUP BY user_id, group_id
)
SELECT DISTINCT user_id
FROM consecutive_groups
WHERE consecutive_days >= 3;
```

Pattern 4: Sessionization (Time-based grouping)

```
-- 30-minute session windows in Presto
WITH session_markers AS (
    SELECT
        user_id,
        event_time,
        LAG(event_time) OVER (PARTITION BY user_id ORDER BY event_time) as
prev_event_time,
        CASE
            WHEN DATE_DIFF('minute',
                LAG(event_time) OVER (PARTITION BY user_id ORDER BY event_time),
                event_time) > 30
                OR LAG(event_time) OVER (PARTITION BY user_id ORDER BY event_time) IS NULL
            THEN 1
            ELSE 0
        END as new_session
    FROM events
),
session_labels AS (
    SELECT
        user_id,
        event_time,
        SUM(new_session) OVER (
            PARTITION BY user_id
            ORDER BY event_time
            ROWS UNBOUNDED PRECEDING
        ) as session_id
    FROM session_markers
)
SELECT
    user_id,
    session_id,
    MIN(event_time) as session_start,
    MAX(event_time) as session_end,
    COUNT(*) as event_count,
    DATE_DIFF('minute', MIN(event_time), MAX(event_time)) as session_duration_minutes
FROM session_labels
GROUP BY user_id, session_id;
```

Pattern 5: Working with Arrays and JSON (Presto-specific)

```
-- Array operations
SELECT
    user_id,
    ARRAY_AGG(DISTINCT product_id) as products_purchased,
    CARDINALITY(ARRAY_AGG(DISTINCT product_id)) as unique_products,
    ARRAY_JOIN(ARRAY_AGG(product_name ORDER BY purchase_date), ', ') as product_list
FROM purchases
GROUP BY user_id;

-- JSON operations
SELECT
    JSON_EXTRACT_SCALAR(metadata, '$.utm_source') as utm_source,
    JSON_EXTRACT(metadata, '$.properties') as properties,
    CAST(JSON_EXTRACT(metadata, '$.user_id') AS BIGINT) as user_id
FROM events
WHERE JSON_EXTRACT_SCALAR(metadata, '$.event_type') = 'purchase';
```

Performance Best Practices for Presto/Trino

1. **Query Optimization Checklist**:
    - Use partition predicates to limit data scanned
    - Push filters down as early as possible
    - Use columnar formats (ORC, Parquet)
    - Leverage bucketing for large joins

    - Use APPROX functions for large-scale aggregations
    - Avoid SELECT * in production queries

2. **Join Optimization**:

    - Put smaller table on the right side of the join
    - Use broadcast joins for small dimension tables
    - Ensure join keys have matching data types
    - Consider bucketing tables on join keys

3. **Aggregation Optimization**:

    - Use APPROX_DISTINCT instead of COUNT(DISTINCT) for large datasets
    - Pre-aggregate data where possible
    - Use partitioning to reduce data scanned

4. **Presto-Specific Tips**:

    - Use `TRY` functions to handle potential errors gracefully
    - Leverage `UNNEST` for array operations
    - Use `VALUES` for creating inline tables
    - Take advantage of lambda functions for complex array operations

Common Interview Tips for Presto/Trino Environments

1. **Key Differences to Remember**:

    - No stored procedures or triggers
    - Limited support for UPDATE/DELETE
    - CTEs instead of temp tables
    - APPROX functions for better performance
    - Strong array and JSON support

2. **Performance Considerations**:

    - Always mention partition pruning
    - Discuss file format choices (ORC vs Parquet)
    - Consider memory limits for large sorts/aggregations
    - Understand distributed query execution

3. **Business Context at Scale**:

  - DAU/MAU calculations with APPROX_DISTINCT
  - Sessionization at scale with proper windowing
  - Funnel analysis with large event datasets
  - Cohort retention with partitioned data

SQL Applied

SQL APPLIED

FIELD CASES

@Airbnb

Realistic Database Schema:

- **Users table**: Tracks both hosts and guests with superhost status
- **Listings**: Includes property types, neighborhoods, instant booking, and business travel readiness
- **Bookings**: Captures guest stays, cancellations, and business travel flags
- **Reviews**: Two-way review system with detailed rating categories (cleanliness, communication, value, etc.)
- **Search logs**: Tracks the user journey from search to booking
- **Calendar**: Dynamic pricing and availability management

Airbnb Business Problems Covered:

1. **Marketplace Analytics**: Supply-demand balance, market saturation, geographic expansion opportunities
2. **Host Performance**: Superhost impact analysis, revenue optimization, pricing strategies
3. **Guest Behavior**: Cohort retention, booking patterns, lifetime value segmentation
4. **Conversion Funnel**: Search to booking conversion, instant booking impact
5. **Trust & Safety**: Review analysis, rating distributions, quality metrics
6. **Revenue Optimization**: Dynamic pricing, seasonal patterns, occupancy rates

Real Airbnb Metrics:

- Booking conversion rates and lead times
- Host lifetime value and performance rankings
- Guest retention by signup cohort

- Search effectiveness and click-through rates
- Pricing optimization based on occupancy
- Superhost vs regular host performance comparison

The questions progress from basic queries about listings and availability to complex problems Airbnb's data team actually faces, such as:

- Identifying high-opportunity markets with unmet demand
- Analyzing the business impact of instant booking
- Measuring seasonal pricing effectiveness
- Building guest loyalty segmentation
- Tracking conversion through the entire user journey

SQL Question Types Source 1

Term Definition Questions

Trigger: a procedure stored within a database, which automatically happens whenever a specific event occurs.

Index: a special lookup table within a database to increase data retrieval speed.

Cursor: a pointer, or identifier, associated with a single or group of rows.

Constraints: rules used to limit the type of data allowed within a table. Common constraints include primary key, foreign key, unique key, and NOT NULL.

ETL (Extract, transform, and load): a data integration process used to combine multiple data sources into one data store, such as a data warehouse.

Primary key, foreign key, and unique key: constraints used to identify records within a table.

Normalization vs. denormalization: techniques used to either divide data into multiple tables to achieve integrity ("normalization") or combine data into a table to increase the speed of data retrieval ("denormalization").

RDBMS vs. DBMS: two types of database management systems. Within a relational database management system (RDBMS) data is stored as a table, while in a database management system (DBMS) its stored as a file.

Clustered vs. non-clustered index: two types of indices used to sort and store data. A clustered index sorts data based on their key values, while a non-clustered index stores data and their records in separate locations.

Comprehension Questions Source 1

What is the purpose of an index in a table? Explain the different types.

What are the types of joins in SQL?

What is the difference between DROP, TRUNCATE, and DELETE statements?

How do you use a cursor?

What is the difference between a HAVING clause and a WHERE clause?

Read more: SQL vs. MySQL: Differences, Similarities, Uses, and Benefits

Questions about a query

This second category gives you an SQL query and asks you a question about it. This tests your ability to read, interpret, analyze, and debug code written by others.

Forms query analysis questions may take:

Given a query,

Put the clauses in order by how SQL would run them.

Identify the error and correct it.

Predict what the query will return.

Explain what problem the query is meant to solve.

Learn more: SQL vs. NoSQL: The Differences Explained + When to Use Each

Write a query

Categorization, aggregation, and ratio (CASE, COUNT, or SUM, numerator and denominator)

Joining two tables (JOIN inner vs. left or right)

Modifying a database (INSERT, UPDATE, and DELETE)

Comparison operators (Less than, greater than, equal to)

Organizing data (ORDER BY, GROUP BY, HAVING)

Subqueries

Forms query-writing questions may take:

Given a table or tables with a few sample rows,

List the three stores with the highest number of customer transactions.

Extract employee IDs for all employees who earned a three or higher on their last performance review.

Calculate the average monthly sales by product displayed in descending order.

Find and remove duplicates in the table without creating another table.

Identify the common records between two tables.

SQL Question Types Source 2

Joins: Master inner, left, right, and full joins.

Aggregations: Know GROUP BY , HAVING , and functions like SUM() , COUNT() , etc.

Window Functions: Focus on ROW_NUMBER() , RANK() , LAG() , LEAD() .

Subqueries: Learn how to handle subqueries within SELECT, WHERE, and FROM.

Common Table Expressions (CTEs): Understand how and when to use them.

Indexes and Performance: Learn indexing strategies and how to optimize query performance.

Data Modeling: Understand normalization, denormalization, and keys.

Complex Queries: Be able to write complex queries combining multiple concepts.

Real-world Scenarios: Be prepared to solve business problems with SQL.

Error Handling: Learn how to debug and fix common SQL issues.

SQL Question Types Source 3

1 Joins (inner, left, self joins) and understanding data relationships

2 Aggregations with GROUP BY , COUNT() , SUM() , AVG()

3 Filtering using WHERE , HAVING , and subconditions

4 Window functions like RANK() , ROW_NUMBER() , LAG() and LEAD()

5 Subqueries and CTEs, especially for nested logic or transformations

6 Indexes and performance optimization, especially on large datasets

---

SQL Questions Source 1

1.) Aggregation (sum vs. count, avg, etc....)

2.) How would Select data from table A that is not in table B (they are looking for NOT EXISTS or a LEFT JOIN scenario here)

3.) Union vs. Union all

4.) Difference in JOINS (usually a real world example is asked here such as "You have a customers table and order table. What JOIN would you use to find all customers that had orders?"

5.) Date manipulation (this is tricky, because each of these companies have asked varying levels of complexity. One question was asked "how to get the previous 6 months worth of data", another asked "How would you convert a DATETIME field to just DATE"

6.) Inserting data into an already created table

7.) Case statements (the questions were always a bit ambiguous here, but I was asked a case statement question in each interview)

8.) Subquery or CTE related questions. They cared less about the answer, but more about how these are actually used

9.) How to identify duplicates in a table? What about multiple tables?

10.) Difference between WHERE and HAVING.

11.) Windows Functions (LAG / LEAD here).

BONUS QUESTIONS (this is a good way to stand out as a Data Analyst): How would you improve query performance / what would you do if a query is running slow? How would you improve Data Quality in this scenario?

SQL Questions Source 2

1: Counting duplicate rows

2: What are Joins and types of join and how do they work

3. Aggregate functions

4. Window functions [100% certain], even though at the job you'll Google the syntax but not for live coding round. Even theoretical questions like which window function to solve a particular problem.

5. More windows function: Difference between Rank and Dense Rank

6. On more advanced levels can be questions can be about subqueries, how SQL works behind the scenes, how we store data.

SQL Questions Source 3

Easy

Generate a report that shows employees who their manager is. This was a SELF-JOIN and I'm so sick of this interview question lmao

Show the latest used product. This was simply using MAX on a datetime field.

Medium:

Find customers with the highest orders between a date span. This involved CTEs, converting a datetime to date, and a JOIN.

Calculate the change over time of products for a date span. This involved some aggregation, a case statement, CTE, and window functions.

Hard:

Find users who were active for 4 consecutive days on our app. Again, this was more CTES, windows functions, and aggregations. Also using HAVING a lot.

Other:

They asked a bit about my experiences with queries running slow and solutions. They asked a bit about indexing and working with "big data." They asked about how I would ensure results are correct with large sets of data. Finally, they asked a bit about data visualization experience via Tableau.

Leetcode premium SQL top 40 (sorted by hard first)

SCHEMA

listings (listing_id, host_id, city, country, room_type, bedrooms, price, created_at, is_active)

hosts (host_id, name, joined_date, superhost, country, response_rate)

guests (guest_id, name, signup_date, country, verified)

bookings (booking_id, listing_id, guest_id, host_id, checkin_date, checkout_date, booking_date, nights, guests_count, total_price, status)

reviews (review_id, booking_id, listing_id, guest_id, host_id, review_date, rating, review_text)

payments (payment_id, booking_id, amount, payment_date, payment_method, status)

amenities (amenity_id, listing_id, amenity_name)

calendar (listing_id, date, available, price)

search_sessions (session_id, guest_id, search_date, filters, results_count)

messages (message_id, sender_id, receiver_id, booking_id, sent_date, message_text)

PRACTICE PROBLEMS

SET 1

QUESTION INDEX OVERVIEW

150 Easy questions (30%): Basic SELECT, filtering, simple joins, basic aggregations

250 Medium questions (50%): Window functions, complex joins, subqueries, date/time analysis, business metrics

100 Difficult questions (20%): Advanced analytics, multi-step analysis, optimization, segmentation, funnel analysis

Basic SELECT & Filtering

Basic SELECT

Concept: SELECT statement **Difficulty:** Easy

Business Purpose: Retrieve basic listing information for inventory review

Question: Retrieve all listing IDs, cities, and prices from the listings table.

```
SELECT DATE(session_start) AS day, COUNT(DISTINCT user_id) AS dau
FROM sessions
WHERE DATE(session_start) = DATE '2025-08-01'
GROUP BY 1;
```

WHERE Clause - Single Condition

Concept: Filtering with WHERE **Difficulty:** Easy

Business Purpose: Find listings in specific market for regional analysis

Question: Find all listings located in San Francisco.

```
SELECT *
FROM listings
WHERE city = 'San Francisco';
```

WHERE Clause - Multiple Conditions

Concept: AND operator **Difficulty:** Easy

Business Purpose: Identify premium active listings for marketing campaigns

Question: Find all active listings in New York with prices greater than $200 per night.

```
SELECT listing_id, price, room_type
FROM listings
WHERE city = 'New York' AND price > 200 AND is_active = 1;
```

...

Concept: ... **Difficulty:** Easy

Business Purpose: ...

Question: ...

```
...
```

...

Concept: ... **Difficulty:** Easy

Business Purpose: ...

Question: ...

```
...
```

...

Concept: ... **Difficulty:** Easy

Business Purpose: ...

Question: ...

```
...
```

...

Concept: ... **Difficulty:** Easy

Business Purpose: ...

Question: ...

```
...
```

...

Concept: ... **Difficulty:** Easy

Business Purpose: ...

Question: ...

```
...
```

PRACTICE PROBLEMS

SET 2

SQL Applied (Unsorted)

Two-Sided Marketplace Challenge

Scenario: "Host sign-ups have increased 30% quarter-over-quarter, but guest bookings have only grown 10%. What problems could this create, and how would you investigate?"

Structure your approach:

Clarify the Problem: This is a supply-demand imbalance - classic two-sided marketplace challenge.

Potential Issues:

- Lower host occupancy rates → frustrated hosts → churn
- Pricing pressure (more supply = lower prices?)
- Quality dilution if we're accepting lower-quality hosts
- Search experience degradation (too many results?)

Key Metrics to Track:

Host Side:

- Host occupancy rate (% of available nights booked)
- Host earnings per month
- Time to first booking
- Host retention/churn rate
- Host satisfaction scores

Guest Side:

- Search results returned per query
- Booking conversion rate
- Guest satisfaction with listings

Marketplace Health:

- Supply utilization rate
- Take rate (Airbnb's revenue %)
- Market concentration (are bookings concentrated among top hosts?)

Investigative Questions:

Geographic Analysis:

- Is the imbalance uniform across markets?
- Are new hosts in areas with existing high demand or oversupplied markets?

Host Quality:

- What's the quality distribution of new hosts?
- Are they competitive on price/amenities?
- Photo quality, descriptions, response rates?

Guest Demand:

- Has guest search volume increased?
- Are guests finding what they're looking for?
- Has time-to-book changed?

Segment Analysis:

- What types of properties are new hosts adding?
- Is it matching guest search patterns?

Recommendations:

Short-term:

- Pause host acquisition in oversupplied markets
- Improve search ranking for new hosts (cold-start problem)
- Dynamic pricing recommendations for new hosts

Medium-term:

- Focus guest acquisition in undersupplied markets
- Host quality scoring and selective acceptance
- Better host onboarding (photos, pricing, calendars)

Long-term:

- Predictive model: Match host acquisition to demand forecasts
- Market-by-market growth strategies
- Product features to improve utilization (flexible search, last-minute bookings)

Metrics

1. Acquisition Metrics

Metrics focused on attracting new users (guests or hosts) and evaluating channel effectiveness.

- New User Signups: Number of new guests or hosts registering per period (e.g., monthly).
- Customer Acquisition Cost (CAC): Total marketing spend divided by new users acquired.
- Acquisition Channel Efficiency: Performance by channel (e.g., email, social, referrals), measured by cost per acquisition or conversions.
- Referral Rate: Percentage of new signups from referrals.

2. Engagement Metrics

Metrics tracking user interaction with the platform post-acquisition.

- User Engagement Rate: Sessions, searches, or views per active user.
- Messaging Response Rate: Percentage of inquiries responded to by hosts within a timeframe (e.g., 24 hours).
- App/Website Traffic: Unique visitors or page views, segmented by source.
- Active Users: Daily/Monthly Active Users (DAU/MAU) for guests and hosts.

3. Conversion Metrics

Metrics for turning interest into actions, like bookings.

- Conversion Rate: Percentage of searches or inquiries leading to bookings.
- Inquiry-to-Booking Rate: Ratio of host inquiries to completed bookings.
- First Booking Rate: Percentage of new users who book within a set period (e.g., 7 days).

4. Retention Metrics

Metrics assessing user loyalty and repeat behavior.

- Retention Rate: Percentage of users active in subsequent periods (e.g., 30-day or cohort-based).
- Churn Rate: Percentage of users who stop engaging (e.g., no bookings in 90 days).
- Repeat Booking Rate: Percentage of users with multiple bookings.

5. Revenue Metrics

Metrics tied to financial outcomes from marketing efforts.

- Total Revenue: Aggregate booking value or fees collected.
- Average Daily Rate (ADR): Average price per night for booked listings.
- Revenue Per Available Rental (RevPAR): Revenue divided by available listing-nights.
- Average Booking Value: Mean value of bookings, segmented by user type or channel.

6. Cost Metrics

Metrics for efficiency of spending.

- Marketing Spend: Total costs on campaigns, ads, etc.
- Cost Per Acquisition (CPA): Similar to CAC but per channel or campaign.
- Return on Investment (ROI): Revenue generated minus costs, divided by costs.

7. Customer Value Metrics

Long-term value derived from users.

- Lifetime Value (LTV): Projected revenue from a user over their lifetime.
- Customer Lifetime Value (CLV): Similar to LTV, often including host contributions.

8. Funnel Metrics

End-to-end user journey analysis.

- Funnel Drop-off Rates: Percentage lost at each stage (e.g., signup -> search -> booking -> review).
- Booking Lead Time: Average days from search/inquiry to booking.
- Cohort Analysis: Performance by user groups (e.g., signup month) across funnel stages.

9. User Experience Metrics

Metrics reflecting satisfaction and quality.

- Net Promoter Score (NPS): Likelihood of recommendation from surveys.
- Review Scores: Average ratings for listings or hosts.
- Guest/Host Satisfaction Score: Composite from reviews and feedback.
- Superhost Ratio: Percentage of hosts meeting elite criteria (e.g., high ratings, low cancellations).

10. Operational Metrics (Marketing-Adjacent)

Metrics influencing marketing strategies, like supply-demand balance.

- Occupancy Rate: Percentage of available nights booked.
- Average Length of Stay (ALOS): Mean days per booking.
- Year-over-Year Growth: Increases in bookings, revenue, or users.
- Market Penetration: Listings or bookings per city/region.

SQL Applied 1 (Syntax)

SQL Applied 1 (Syntax)

Interview-Length SQL Combinations for Marketing Analytics

@Claude

1. Month-over-Month Growth by Channel

Combination: DATE_TRUNC + LAG + CASE

```sql
-- Calculate MoM growth and flag declining channels
SELECT
    DATE_TRUNC('month', booking_date) as month,
    marketing_channel,
    COUNT(*) as bookings,
    LAG(COUNT(*)) OVER (PARTITION BY marketing_channel ORDER BY DATE_TRUNC('month', booking_date)) as prev_month,
    CASE
        WHEN LAG(COUNT(*)) OVER (PARTITION BY marketing_channel ORDER BY DATE_TRUNC('month', booking_date)) IS NULL THEN NULL
        ELSE (COUNT(*) - LAG(COUNT(*)) OVER (PARTITION BY marketing_channel ORDER BY DATE_TRUNC('month', booking_date))) * 100.0 /
             LAG(COUNT(*)) OVER (PARTITION BY marketing_channel ORDER BY DATE_TRUNC('month', booking_date))
    END as mom_growth_pct
FROM bookings
WHERE booking_date >= '2024-01-01'
GROUP BY month, marketing_channel
ORDER BY marketing_channel, month;
```

2. Find Returning Users Within Time Window

Combination: SELF JOIN + DATE_DIFF + WHERE

```sql
-- Users who booked again within 60 days
SELECT
    b1.user_id,
    b1.booking_date as first_booking,
    b2.booking_date as second_booking,
    DATE_DIFF('day', b1.booking_date, b2.booking_date) as days_between,
    b1.marketing_channel as first_channel,
    b2.marketing_channel as second_channel
FROM bookings b1
JOIN bookings b2
    ON b1.user_id = b2.user_id
    AND b2.booking_date > b1.booking_date
    AND DATE_DIFF('day', b1.booking_date, b2.booking_date) <= 60
WHERE b1.booking_date >= '2024-01-01';
```

3. Top Performers by Segment

Combination: CTE + RANK + HAVING

```sql
-- Top 3 campaigns per channel with minimum spend
WITH campaign_performance AS (
    SELECT
        marketing_channel,
        campaign_id,
        SUM(spend) as total_spend,
        SUM(revenue) as total_revenue,
        SUM(revenue) / NULLIF(SUM(spend), 0) as roas
    FROM marketing_data
    WHERE campaign_date >= '2024-01-01'
    GROUP BY marketing_channel, campaign_id
    HAVING SUM(spend) >= 5000
)
SELECT
    marketing_channel,
    campaign_id,
    total_spend,
    total_revenue,
    roas,
    RANK() OVER (PARTITION BY marketing_channel ORDER BY roas DESC) as rank
FROM campaign_performance
QUALIFY rank <= 3;
```

4. Cohort Week 1 Retention

Combination: CTE + DATE_TRUNC + DATE_DIFF + CASE

```
-- Week 1 retention rate by signup channel
WITH signups AS (
    SELECT
        user_id,
        DATE_TRUNC('week', signup_date) as cohort_week,
        marketing_channel
    FROM users
    WHERE signup_date >= '2024-01-01'
)
SELECT
    s.cohort_week,
    s.marketing_channel,
    COUNT(DISTINCT s.user_id) as signups,
    COUNT(DISTINCT CASE
        WHEN DATE_DIFF('day', s.cohort_week, b.booking_date) <= 7
        THEN b.user_id
    END) as week1_bookers,
    COUNT(DISTINCT CASE
        WHEN DATE_DIFF('day', s.cohort_week, b.booking_date) <= 7
        THEN b.user_id
    END) * 100.0 / COUNT(DISTINCT s.user_id) as week1_retention_pct
FROM signups s
LEFT JOIN bookings b ON s.user_id = b.user_id
GROUP BY s.cohort_week, s.marketing_channel;
```

5. Revenue Percentiles by Channel

Combination: PERCENTILE_CONT + CASE + WHERE IN

```
...
```

6. Attribution Window Analysis

Combination: CTE + DATE_DIFF + BETWEEN + SUM with CASE

```
-- Revenue attributed in different time windows
WITH conversions AS (
    SELECT
        t.marketing_channel,
        t.touch_date,
        b.booking_date,
        b.revenue,
        DATE_DIFF('day', t.touch_date, b.booking_date) as
days_to_conversion
    FROM marketing_touches t
    JOIN bookings b
        ON t.user_id = b.user_id
        AND b.booking_date >= t.touch_date
        AND DATE_DIFF('day', t.touch_date, b.booking_date) <= 30
)
SELECT
    marketing_channel,
    COUNT(*) as total_conversions,
    SUM(CASE WHEN days_to_conversion <= 1 THEN revenue ELSE 0 END) as
day1_revenue,
    SUM(CASE WHEN days_to_conversion <= 7 THEN revenue ELSE 0 END) as
day7_revenue,
    SUM(CASE WHEN days_to_conversion <= 14 THEN revenue ELSE 0 END) as
day14_revenue,
    SUM(revenue) as day30_revenue
FROM conversions
GROUP BY marketing_channel;
```

7. Running Totals with Reset

Combination: SUM OVER + PARTITION BY + DATE_TRUNC

```sql
-- Cumulative spend per campaign, resetting each quarter
SELECT
    campaign_id,
    DATE_TRUNC('quarter', spend_date) as quarter,
    spend_date,
    daily_spend,
    SUM(daily_spend) OVER (
        PARTITION BY campaign_id, DATE_TRUNC('quarter', spend_date)
        ORDER BY spend_date
    ) as qtd_spend
FROM campaign_spend
WHERE spend_date >= '2024-01-01'
ORDER BY campaign_id, spend_date;
```

8. First and Last Touch Attribution

Combination: CTE + ROW_NUMBER + CASE + WHERE

```sql
-- Compare first vs last touch channel
WITH ranked_touches AS (
    SELECT
        user_id,
        marketing_channel,
        touch_date,
        ROW_NUMBER() OVER (PARTITION BY user_id ORDER BY touch_date ASC) as first_touch,
        ROW_NUMBER() OVER (PARTITION BY user_id ORDER BY touch_date DESC) as last_touch
    FROM marketing_touches
)
SELECT
    first_channel,
    last_channel,
    COUNT(DISTINCT user_id) as users,
    SUM(revenue) as total_revenue
FROM (
    SELECT
```

```
        t1.user_id,
        t1.marketing_channel as first_channel,
        t2.marketing_channel as last_channel,
        b.revenue
    FROM ranked_touches t1
    JOIN ranked_touches t2 ON t1.user_id = t2.user_id
    JOIN bookings b ON t1.user_id = b.user_id
    WHERE t1.first_touch = 1 AND t2.last_touch = 1
) attribution
GROUP BY first_channel, last_channel
ORDER BY total_revenue DESC;
```

9. Active Users by Recency

Combination: MAX + DATE_DIFF + CASE + GROUP BY

```
-- Segment users by last activity
SELECT
    marketing_channel,
    CASE
        WHEN DATE_DIFF('day', MAX(booking_date), CURRENT_DATE) <= 30 THEN
'active'
        WHEN DATE_DIFF('day', MAX(booking_date), CURRENT_DATE) <= 90 THEN
'at_risk'
        WHEN DATE_DIFF('day', MAX(booking_date), CURRENT_DATE) <= 180 THEN
'dormant'
        ELSE 'churned'
    END as user_status,
    COUNT(DISTINCT user_id) as users,
    AVG(DATE_DIFF('day', MAX(booking_date), CURRENT_DATE)) as
avg_days_since_booking
FROM bookings
WHERE marketing_channel IS NOT NULL
GROUP BY marketing_channel, user_status
ORDER BY marketing_channel,
    CASE user_status
        WHEN 'active' THEN 1
        WHEN 'at_risk' THEN 2
        WHEN 'dormant' THEN 3
        ELSE 4
    END;
```

10. Conversion Funnel with Drop-off

Combination: CTE + LEFT JOIN + COUNT DISTINCT + COALESCE

```
-- Simple funnel analysis
WITH funnel AS (
    SELECT DISTINCT user_id, 'impression' as stage FROM impressions WHERE date >= '2024-09-01'
    UNION
    SELECT DISTINCT user_id, 'click' as stage FROM clicks WHERE date >= '2024-09-01'
    UNION
    SELECT DISTINCT user_id, 'signup' as stage FROM signups WHERE date >= '2024-09-01'
    UNION
    SELECT DISTINCT user_id, 'booking' as stage FROM bookings WHERE date >= '2024-09-01'
)
SELECT
    'impression' as stage,
    COUNT(DISTINCT CASE WHEN stage = 'impression' THEN user_id END) as users,
    COUNT(DISTINCT CASE WHEN stage = 'click' THEN user_id END) * 100.0 /
        COUNT(DISTINCT CASE WHEN stage = 'impression' THEN user_id END) as conversion_to_next
FROM funnel
UNION ALL
SELECT
    'click' as stage,
    COUNT(DISTINCT CASE WHEN stage = 'click' THEN user_id END) as users,
    COUNT(DISTINCT CASE WHEN stage = 'signup' THEN user_id END) * 100.0 /
        COUNT(DISTINCT CASE WHEN stage = 'click' THEN user_id END) as conversion_to_next
FROM funnel
UNION ALL
SELECT
```

```
    'signup' as stage,
    COUNT(DISTINCT CASE WHEN stage = 'signup' THEN user_id END) as users,
    COUNT(DISTINCT CASE WHEN stage = 'booking' THEN user_id END) * 100.0 /
        COUNT(DISTINCT CASE WHEN stage = 'signup' THEN user_id END) as
conversion_to_next
FROM funnel;
```

11. Weekend vs Weekday Performance

Combination: EXTRACT + CASE + DATE_TRUNC + WHERE

```
-- Compare weekend vs weekday bookings
SELECT
    marketing_channel,
    CASE
        WHEN EXTRACT(DOW FROM booking_date) IN (0, 6) THEN 'weekend'
        ELSE 'weekday'
    END as day_type,
    COUNT(*) as bookings,
    AVG(revenue) as avg_revenue,
    SUM(revenue) as total_revenue
FROM bookings
WHERE booking_date >= DATE_TRUNC('month', CURRENT_DATE - INTERVAL '3
months')
GROUP BY marketing_channel, day_type
ORDER BY marketing_channel, total_revenue DESC;
```

12. Users with Multiple Channels

Combination: HAVING + COUNT DISTINCT + STRING_AGG

```
-- Find users exposed to multiple marketing channels
SELECT
    user_id,
    COUNT(DISTINCT marketing_channel) as num_channels,
    STRING_AGG(DISTINCT marketing_channel, ', ' ORDER BY marketing_channel)
```

```sql
as channels,
    MIN(touch_date) as first_touch,
    MAX(touch_date) as last_touch
FROM marketing_touches
WHERE touch_date >= '2024-01-01'
GROUP BY user_id
HAVING COUNT(DISTINCT marketing_channel) >= 3
ORDER BY num_channels DESC;
```

SQL Applied 2 (Syntax)

SQL Applied 2 (Syntax)

Common SQL Patterns

w/ Problem-Solving

@Grok

Problems require combining patterns like joins, window functions, CTEs, subqueries, and date handling. Provide your SQL query and explain the business insight it derives.

Note: Dates are based on data up to October 05, 2025.

Aggregation

w/ GROUP BY & HAVING

Description: Summarize by categories and filter aggregates post-grouping; identifies high-performing campaign types for reallocating ad budgets in Q4 2025.

Syntax

```
SELECT
  col1,
  AGG(col2) AS agg_col2
FROM table
GROUP BY col1
HAVING <aggregate_condition>;
```

Airbnb

```
SELECT
  campaign_type,
  AVG(conversion_rate) AS avg_cr
FROM campaigns
WHERE start_date >= DATE '2025-01-01'
GROUP BY campaign_type
HAVING AVG(conversion_rate) > 0.05;
```

Joins w/ Aggregations

Description: Combine tables and aggregate results; calculates revenue by user country to prioritize international marketing expansions in 2025.

Syntax

```
SELECT
  t1.col,
  AGG(t2.col) AS agg_val
FROM table1 AS t1
JOIN table2 AS t2
  ON t1.id = t2.id
GROUP BY t1.col;
```

Airbnb

```
SELECT
  u.country,
  SUM(b.booking_value) AS total_revenue
FROM users AS u
JOIN bookings AS b
  ON u.user_id = b.user_id
WHERE b.booking_date >= DATE '2025-01-01'
GROUP BY u.country;
```

Subqueries

Filtering or Derived Metrics

Description: Use nested queries for thresholds or comparisons; finds above-average bookers among 2025 signups for targeted retention.

Syntax

```
SELECT
  col
FROM table
WHERE col >
  (SELECT AGG(col) FROM table);
```

GENERAL SUBQUERY TYPES

Scalar Aggregates

Add metrics like total spend, average rating, or max purchase per user.

```
SELECT
  user_id,
  (SELECT COUNT(*) FROM orders o WHERE o.user_id = u.user_id) AS
order_count
FROM users u;
```

Correlated Lookups

Pull in related data from another table based on the current row.

```
SELECT
 product_id,
  (SELECT category_name FROM categories c WHERE c.category_id =
p.category_id) AS category
FROM products p;
```

SCALAR + CORRELATED SUBQUERY TYPES

w/ SELF JOINS

SCALAR

Table: campaign_performance

campaign_id	channel	spend	impressions	clicks	conversions
1	Search	500	10,000	800	40
2	Social	300	8,000	400	20
3	Display	200	12,000	300	10
4	Email	100	2,000	150	15

Scalar *w/ SELECT*

Use Case: Attach a global benchmark to each row.

```
SELECT
  campaign_id,
  channel,
  spend,
  (SELECT AVG(spend) FROM campaign_performance) AS avg_spend_all
FROM campaign_performance;
```

Result:

campaign_id	channel	spend	avg_spend_all
1	Search	500	275
2	Social	300	275
3	Display	200	275

4	Email	100	275

Every row is enriched with the **same scalar value** (global average spend).

Scalar w/ *WHERE*

Use Case: Filter rows against a single-value condition.

```
SELECT campaign_id, channel, spend
FROM campaign_performance
WHERE spend > (SELECT AVG(spend) FROM campaign_performance);
```

Result:

campaign_id	channel	spend
1	Search	500
2	Social	300

Only campaigns with spend above the **global average (275)** are returned.

Scalar w/ *HAVING*

Use Case: Filter groups against a global benchmark.

```
SELECT channel, AVG(conversions) AS avg_conversions
FROM campaign_performance
GROUP BY channel
HAVING AVG(conversions) > (
  SELECT AVG(conversions) FROM campaign_performance
);
```

Result:

channel	avg_conversions
Search	40
Social	20

Only channels with average conversions above the **global average (21.25)** are kept.

Scalar Subqueries Utility

- **SELECT** → Enrich rows with a global constant (benchmarking).
- **WHERE** → Filter individual rows against a global constant.
- **HAVING** → Filter aggregated groups against a global constant.

If you're modeling attribution or running CUPED adjustments, `SELECT` subqueries can help attach pre-period metrics, while `WHERE` subqueries can isolate treatment groups or filter out noisy data.

CORRELATED

Correlated *w/ SELECT*

Query: Attach each campaign's **channel-level average spend**.

```sql
SELECT
  c1.campaign_id,
  c1.spend,
  (SELECT AVG(c2.spend)
   FROM campaign_performance c2
   WHERE c2.channel = c1.channel) AS avg_spend_channel
FROM campaign_performance c1;
```

Result:

campaign_id	channel	spend	avg_spend_channel
1	Search	500	550
5	Search	600	550
2	Social	300	275
6	Social	250	275
3	Display	200	200

4	Email	100	100

Each row gets the **average spend of its own channel** (row-dependent).

Correlated *w/ WHERE*

Query: Keep only campaigns whose spend is **above their channel's average spend**.

```
SELECT c1.campaign_id, c1.channel, c1.spend
FROM campaign_performance c1
WHERE c1.spend > (
  SELECT AVG(c2.spend)
  FROM campaign_performance c2
  WHERE c2.channel = c1.channel
);
```

Result:

campaign_id	channel	spend
5	Search	600
2	Social	300

Keeps campaigns above the **average spend of their own channel**.

Correlated *w/ HAVING*

Use Case: Find channels whose average conversions are above the average conversions of campaigns in the *same spend tier*.

```
SELECT c1.channel, AVG(c1.conversions) AS avg_conversions
FROM campaign_performance c1
GROUP BY c1.channel
HAVING AVG(c1.conversions) > (
  SELECT AVG(c2.conversions)
  FROM campaign_performance c2
  WHERE (CASE WHEN c2.spend >= 300 THEN 'High' ELSE 'Low' END) =
        (CASE WHEN AVG(c1.spend) >= 300 THEN 'High' ELSE 'Low' END)
);
```

Result:

channel	avg_conversions
Search	50
Social	19

SUMMARY

- Search (avg 50) beats the **High-spend tier average** (~40).
- Social (avg 19) beats the **Low-spend tier average** (~14).
- Display and Email don't exceed their tier averages.

Correlated Subqueries Utility

- **SELECT** → enrich each row with a context-specific benchmark (e.g., channel average).
- **WHERE** → filter rows relative to their peer group (e.g., above channel average).
- **HAVING** → Filter groups relative to a peer group benchmark (e.g., channel vs. spend-tier average).

w/ NON-SELF JOINS

SCALAR

They can pull a single value from any table, not just the one in the outer query.

Use Case: Attach the global average spend from the campaign_performance table to each row in a completely different users table.

```
SELECT
  u.user_id,
  u.country,
  (SELECT AVG(spend) FROM campaign_performance) AS avg_campaign_spend
FROM users u;
```

Here, the subquery references a different table (campaign_performance), not a self-join.

CORRELATED

They just need to reference a column from the **outer query**.

Use Case: For each user, check if they've booked more than the average spend of their country (outer query = users, subquery = bookings).

```
SELECT u.user_id, u.country
FROM users u
WHERE u.total_spend > (
  SELECT AVG(b.amount)
  FROM bookings b
  WHERE b.country = u.country
);
```

Here, the subquery references a different table (bookings), but it's still correlated because it depends on u.country from the outer query.

Airbnb

- **Scalar** is great for attaching *global KPIs* (e.g., "global average CTR").
- **Correlated** is great for *contextual comparisons* (e.g., "this campaign vs. its channel average" or "this user's spend vs. their own historical average").

Table: monthly_campaign_performance

campaign_id	channel	month	spend	conversions
1	Search	2025-07-01	400	30
1	Search	2025-08-01	500	40
2	Social	2025-07-01	200	15
2	Social	2025-08-01	300	20
3	Display	2025-07-01	250	12
3	Display	2025-08-01	200	10

Query: Compare each campaign's spend this month vs. last month

We'll use a correlated subquery that looks up the previous month's spend for the same campaign.

```sql
SELECT
  curr.campaign_id,
  curr.channel,
  curr.month,
  curr.spend AS current_spend,
  (
    SELECT prev.spend
    FROM monthly_campaign_performance prev
    WHERE prev.campaign_id = curr.campaign_id
      AND prev.month = DATEADD('month', -1, curr.month)
  ) AS previous_spend,
  curr.spend - COALESCE((
    SELECT prev.spend
    FROM monthly_campaign_performance prev
    WHERE prev.campaign_id = curr.campaign_id
      AND prev.month = DATEADD('month', -1, curr.month)
  ), 0) AS spend_delta
FROM monthly_campaign_performance curr
WHERE curr.month = '2025-08-01';
```

Result:

campaign_id	channel	month	current_spend	previous_spend	spend_delta
1	Search	2025-08-01	500	400	+100
2	Social	2025-08-01	300	200	+100
3	Display	2025-08-01	200	250	-50

Sample Use Cases

- **Experimentation**: Compare treatment vs. pre-period metrics (CUPED-style adjustments).
- **Retention/Churn**: Track whether a user’s or campaign’s activity is increasing or declining month-over-month.
- **Budget Optimization**: Spot campaigns that are scaling up or down in spend and conversions.

Common Table Expressions (CTEs)

Readable Complex Queries

Description: Define temporary result sets for multi-step logic; tracks monthly first-bookings for cohort analysis in 2025.

Syntax

```
WITH cte AS (
  SELECT ...
  FROM ...
)
SELECT ...
FROM cte;
```

Airbnb

```
WITH new_users AS (
  SELECT
    user_id,
    MIN(booking_date) AS first_booking
  FROM bookings
  WHERE booking_date >= DATE '2025-01-01'
  GROUP BY user_id
)
SELECT
```

```
  DATE_TRUNC('month', first_booking) AS cohort_month,
  COUNT(user_id) AS new_bookers
FROM new_users
GROUP BY cohort_month;
```

Window Functions

Ranking or Running Totals

Description: Calculate across row sets without collapsing rows; computes cumulative conversions to evaluate YTD momentum.

Syntax

```
SELECT
  col,
  WINDOW_FUNC(expr) OVER (
    PARTITION BY part_col
    ORDER BY order_col
  ) AS win_val
FROM table;
```

Airbnb

```
SELECT
  campaign_id,
  start_date,
  conversions,
  SUM(conversions) OVER (
    ORDER BY start_date
    ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
  ) AS running_total
FROM campaigns
WHERE start_date >= DATE '2025-01-01'
  AND start_date <= DATE '2025-10-05'
ORDER BY start_date;
```

Self-Joins

Comparisons Over Time or Hierarchies

Description: Compare rows within the same table, e.g., MoM.

Syntax

```sql
SELECT
  t1.col,
  t2.col
FROM table AS t1
JOIN table AS t2
  ON t1.id = t2.id
 AND <relationship_condition>;
```

Airbnb

Compare monthly revenue MoM from Jan to Oct 2025.

```sql
SELECT
  m1.month,
  m1.revenue,
  m2.revenue AS prev_revenue,
  (m1.revenue - m2.revenue) / NULLIF(m2.revenue, 0) AS growth_rate
FROM monthly_metrics AS m1
LEFT JOIN monthly_metrics AS m2
  ON m1.month = date_add('month', 1, m2.month)
WHERE m1.month BETWEEN DATE '2025-01-01' AND DATE '2025-10-01';
```

Comparing Booking Behavior Over Time

Use Case: Track how a *guest's booking frequency changes month-over-month*; helps identify users with increasing or decreasing engagement—useful for churn modeling or targeting re-engagement campaigns.

```sql
SELECT
  curr.user_id,
  curr.month AS current_month,
  curr.booking_count AS current_bookings,
  prev.booking_count AS previous_bookings,
  curr.booking_count - COALESCE(prev.booking_count, 0) AS booking_delta
FROM
  monthly_bookings curr
LEFT JOIN
  monthly_bookings prev
  ON curr.user_id = prev.user_id AND curr.month = DATE_ADD('month', 1,
prev.month);
```

Hierarchical Comparison: Campaign vs. Sub-Campaign

Use Case: Compare performance of sub-campaigns to their parent campaign; useful for evaluating budget allocation efficiency across nested campaign structures.

```
SELECT
  child.campaign_id,
  child.name AS sub_campaign,
  parent.name AS parent_campaign,
  child.spend,
  parent.spend AS parent_spend,
  ROUND(child.spend / parent.spend, 2) AS spend_ratio
FROM
  campaigns child
JOIN
  campaigns parent
  ON child.parent_campaign_id = parent.campaign_id;
```

Experimentation: CUPED Pre-Period Matching

Use Case: Match each user’s pre-period metric to their post-period performance; enables CUPED adjustment or direct lift analysis for experimentation frameworks.

```
SELECT
  post.user_id,
  post.conversion_rate AS post_conversion,
  pre.conversion_rate AS pre_conversion,
  post.conversion_rate - pre.conversion_rate AS lift
```

```
FROM
  experiment_results post
JOIN
  experiment_results pre
  ON post.user_id = pre.user_id AND pre.period = 'pre' AND post.period =
'post';
```

Pricing Dynamics: Host Price Changes

Use Case: Detect when a host changes their listing price; useful for dynamic pricing analysis or alerting systems.

```
SELECT
  curr.listing_id,
  curr.date AS current_date,
  curr.price AS current_price,
  prev.price AS previous_price,
  curr.price - prev.price AS price_change
FROM
  listing_prices curr
JOIN
  listing_prices prev
  ON curr.listing_id = prev.listing_id AND curr.date = DATE_ADD('day', 1,
prev.date);
```

Dates

Cohort or Time-Based Analysis

Description: Group by periods and compute period metrics; builds signup cohorts for 2025.

Syntax

```
SELECT
  DATE_TRUNC('month', date_col) AS month,
  AGG(col) AS agg_value
FROM table
GROUP BY month;
```

Airbnb

```
SELECT
  DATE_TRUNC('month', signup_date) AS cohort_month,
  COUNT(user_id) AS signups
FROM users
WHERE signup_date BETWEEN DATE '2025-01-01' AND DATE '2025-10-05'
GROUP BY cohort_month
ORDER BY cohort_month;
```

Top-N Queries
w/ LIMIT and ORDER BY

Description: Retrieve top or bottom results; top 10 listings by reviews for featured promotions.

Syntax

```
SELECT
  col
FROM table
ORDER BY sort_col DESC
LIMIT n;
```

Airbnb

```
SELECT
  listing_id,
  AVG(review_score) AS avg_score
FROM reviews
WHERE review_date >= DATE '2025-01-01'
GROUP BY listing_id
ORDER BY avg_score DESC
LIMIT 10;
```

Conditional Aggregations
w/ CASE

Description: Segment metrics inside aggregates; segments conversions by source for 2025 optimization.

Syntax

```
SELECT
  SUM(CASE WHEN condition THEN measure ELSE 0 END) AS seg_val
FROM table;
```

Airbnb

```
SELECT
  campaign_id,
  SUM(CASE WHEN source = 'email' THEN conversions ELSE 0 END)
    AS email_conversions,
  SUM(CASE WHEN source = 'social' THEN conversions ELSE 0 END)
    AS social_conversions
FROM campaigns
WHERE start_date >= DATE '2025-01-01'
GROUP BY campaign_id;
```

Pivot/Unpivot
Reshaping Data

Description: Pivot via CASE for portability; pivots monthly bookings for cross-city comparison.

Syntax

```
SELECT
  col,
  SUM(CASE WHEN pivot_col = val1 THEN agg_unit ELSE 0 END) AS val1,
  SUM(CASE WHEN pivot_col = val2 THEN agg_unit ELSE 0 END) AS val2
FROM table
```

```sql
GROUP BY col;
```

Airbnb

```sql
SELECT
  city,
  SUM(CASE
        WHEN DATE_TRUNC('month', booking_date) = DATE '2025-01-01'
        THEN 1 ELSE 0
      END) AS jan_bookings,
  SUM(CASE
        WHEN DATE_TRUNC('month', booking_date) = DATE '2025-02-01'
        THEN 1 ELSE 0
      END) AS feb_bookings
FROM bookings
GROUP BY city;
```

NULLs & COALESCE Management

Description: Provide defaults for missing values; defaults unknown sources to 'Organic' for attribution.

Syntax

```sql
SELECT
  COALESCE(col1, default_val) AS col1_filled
FROM table
WHERE col2 IS NOT NULL;
```

Airbnb

```sql
SELECT
  user_id,
  COALESCE(referral_source, 'Organic') AS source,
  signup_date
FROM users
```

```sql
WHERE signup_date >= DATE '2025-01-01';
```

String Manipulation

Cleaning or Extraction

Description: Normalize and extract patterns; normalizes emails and surfaces potential duplicates.

Syntax

```sql
SELECT
  substr(col, start, len) AS part,
  replace(col, find, replace) AS replaced
FROM table;
```

Airbnb

```sql
SELECT
  user_id,
  LOWER(TRIM(email)) AS cleaned_email,
  COUNT(*) OVER (
    PARTITION BY LOWER(TRIM(email))
  ) AS dup_count
FROM users
WHERE email LIKE '%airbnb%'
  AND signup_date <= DATE '2025-10-05'
ORDER BY cleaned_email;
```

SQL Problem Sets for Airbnb Marketing Analytics Interviews

Below is a set of 10 medium-to-hard SQL problems focused on Airbnb's marketing metrics.

These are designed to mimic interview questions, emphasizing analytical thinking, query optimization, and business insights (e.g., user acquisition, retention, campaign ROI, segmentation, and funnel analysis).

Schema

- **users** (user_id, signup_date, country, referral_source, is_host)
- **bookings** (booking_id, user_id, listing_id, booking_date, booking_value, stay_length_days)
- **campaigns** (campaign_id, start_date, end_date, source, conversions, cost)
- **reviews** (review_id, listing_id, user_id, review_date, review_score)
- **listings** (listing_id, host_id, city, price_per_night)
- **monthly_metrics** (month DATE, revenue, bookings_count, active_users)

Problem Set

Problems require combining patterns like joins, window functions, CTEs, subqueries, and date handling. Provide your SQL query and explain the business insight it derives.

Note: Dates are based on data up to October 05, 2025.

Problem 1 (Medium): Cohort Retention Rate

Calculate the 30-day retention rate for users who signed up in each month of 2025. Retention is defined as users who made at least one booking within 30 days of signup. Output: cohort_month, signups, retained_users, retention_rate (as percentage).

Problem 2 (Medium): Campaign ROI by Source

For campaigns running in 2025, compute ROI (revenue generated minus cost, divided by cost) per source. Join with bookings to attribute revenue where booking_date is between campaign start/end and referral_source matches campaign source. Filter for ROI > 1.5. Output: source, total_cost, total_revenue, roi.

Problem 3 (Hard): User Lifetime Value (LTV) Segmentation

Using a CTE, segment users into 'High', 'Medium', 'Low' LTV based on total booking_value (High: > $5000, Medium: $1000-5000, Low: < $1000). Then, calculate average review_score per segment for bookings in 2025. Output: ltv_segment, user_count, avg_review_score.

Problem 4 (Medium): Month-over-Month Growth in Active Hosts

Identify active hosts (those with at least one booking in the month) and compute MoM growth percentage in active hosts from January to September 2025. Use self-join on monthly aggregated data. Output: month, active_hosts, prev_active_hosts, mom_growth_pct.

Problem 5 (Hard): Funnel Conversion Analysis

Build a funnel for user journey in Q3 2025: signups -> first booking (within 7 days) -> repeat booking (within 30 days of first). Use window functions to track per user. Output: total_signups, first_bookers, repeat_bookers, first_booking_rate, repeat_rate.

Problem 6 (Medium): Top Cities by Booking Growth

Rank the top 5 cities by year-over-year booking value growth from 2024 to 2025 (assume prior year data available). Use subqueries for YoY comparison. Output: city, bookings_2024, bookings_2025, growth_pct, rank.

Problem 7 (Hard): Attribution Modeling

Attribute bookings to the last campaign a user interacted with before booking (assume a interactions table with user_id, campaign_id, interaction_date). Use ROW_NUMBER() to get the last interaction per booking. Output: campaign_id, attributed_bookings, total_value.

Problem 8 (Medium): Churn Rate by Referral Source

Calculate monthly churn rate for users signed up in 2025: churned if no activity (booking or review) in the last 90 days as of October 05, 2025. Group by referral_source. Output: referral_source, total_users, churned_users, churn_rate.

Problem 9 (Hard): Pivot Table for Seasonal Metrics

Pivot average stay_length_days by city and quarter in 2025 (Q1-Q3). Use CASE for pivoting. Output: city, q1_avg_stay, q2_avg_stay, q3_avg_stay.

Problem 10 (Medium): Anomaly Detection in Conversions

Flag campaigns in 2025 where daily conversions dropped more than 20% from the 7-day moving average. Use LAG and AVG over window. Output: campaign_id, date, conversions, moving_avg, drop_pct (for flagged days).

SQL Applied 1 (PS)

SQL Applied 1 (PS)

@ChatGPT

Schema

- **users**: user_id, signup_ts, signup_channel, country.
  - Note: one row per user.
- **sessions**: session_id, user_id, start_ts, landing_url, channel, geo
  - Note: channel is first-touch for session.
- **events**: user_id, session_id, event, ts, listing_id.
  - Note: event in {view, wishlist, start_checkout, book}.
- **bookings**: booking_id, user_id, session_id, ts, payout.
  - Note: ts is booking time.
- **ad_impressions**: impression_id, user_id, campaign_id, channel, ts.
  - Note: ad logs.
- **ad_clicks**: click_id, user_id, campaign_id, channel, ts.
  - Note: click logs.
- **channel_spend**: dt, channel, campaign_id, spend.
  - Note: daily spend.
- **emails**: email_id, user_id, campaign, sent_ts, open_ts, click_ts.
  - Note: lifecycle.
- **experiment_assignments**: user_id, variant.

Problem & Solution Sets

1. Theme: CTR/CVR

Question: Daily CTR and CVR by campaign

Key : `date_trunc, LEFT JOIN, BETWEEN, NULLIF, INTERVAL`

Pattern: Attribution window join. CTR/CVR by day and campaign

SQL:

~~~
```sql
SELECT
  date_trunc('day', i.ts) AS d,
  i.campaign_id,
  count(*) AS impressions,
  count(c.click_id) AS clicks,
  count(b.booking_id) AS bookings,
  1.0 * count(c.click_id) / count(*) AS ctr,
  1.0 * count(b.booking_id) /
  NULLIF(count(c.click_id), 0) AS cvr
FROM ad_impressions i
LEFT JOIN ad_clicks c
  ON c.user_id = i.user_id
 AND c.campaign_id = i.campaign_id
 AND c.ts BETWEEN i.ts AND i.ts + INTERVAL '1' day
LEFT JOIN bookings b
  ON b.user_id = i.user_id
 AND b.ts BETWEEN i.ts AND i.ts + INTERVAL '7' day
GROUP BY 1, 2;
```
~~~

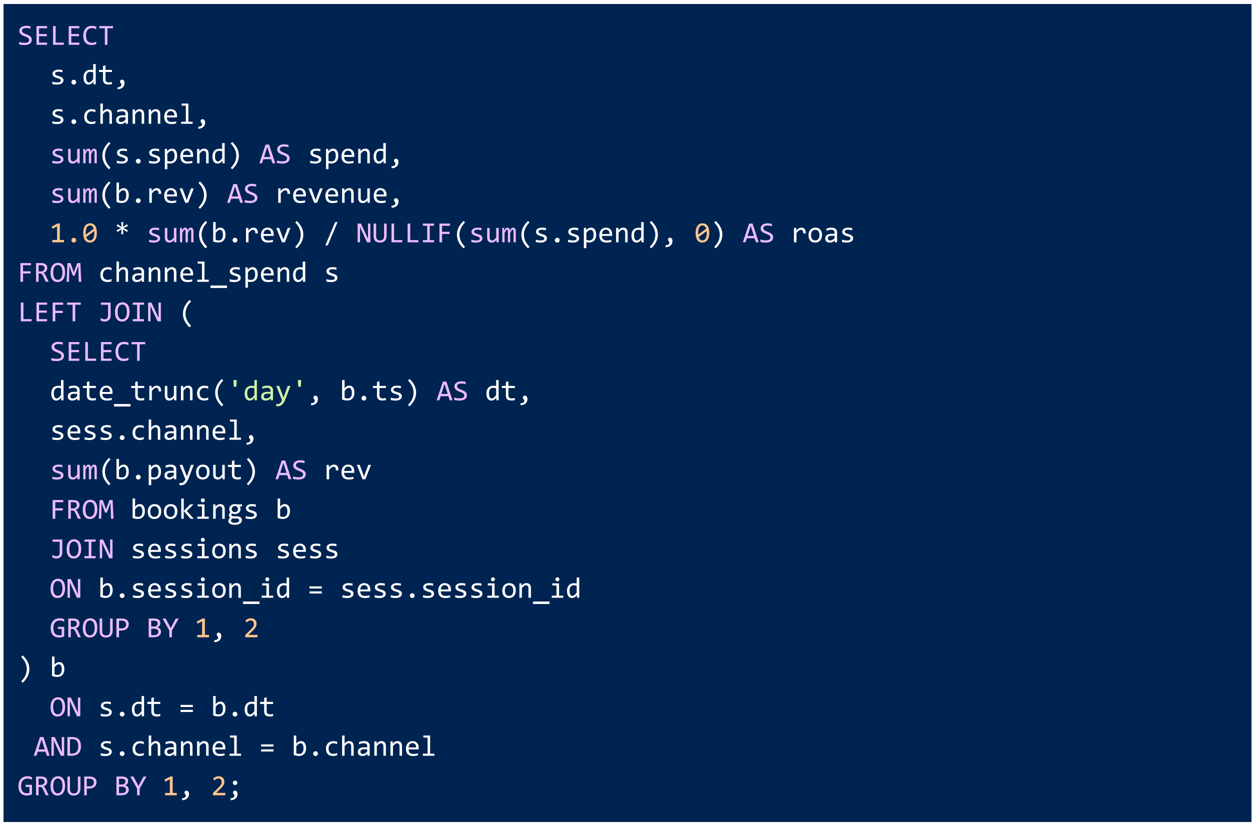
2. Theme: ROAS

Question: ROAS by channel per day.

Keys: date_trunc, LEFT JOIN, GROUP BY, NULLIF.

Pattern: Join daily spend to booked revenue.

SQL:



```
SELECT
  s.dt,
  s.channel,
  sum(s.spend) AS spend,
  sum(b.rev) AS revenue,
  1.0 * sum(b.rev) / NULLIF(sum(s.spend), 0) AS roas
FROM channel_spend s
LEFT JOIN (
  SELECT
  date_trunc('day', b.ts) AS dt,
  sess.channel,
  sum(b.payout) AS rev
  FROM bookings b
  JOIN sessions sess
  ON b.session_id = sess.session_id
  GROUP BY 1, 2
) b
  ON s.dt = b.dt
 AND s.channel = b.channel
GROUP BY 1, 2;
```

3. Theme: Last-touch

Question: Last channel before booking (per booking).

Key : max_by, inequality join.

Pattern: Last-touch attribution.

SQL:

```
SELECT
  b.booking_id,
  max_by(sess.channel, sess.start_ts) AS last_touch
FROM bookings b
JOIN sessions sess
  ON b.user_id = sess.user_id
 AND sess.start_ts <= b.ts
```

```
GROUP BY 1;
```

4. Theme: LTV 90d

Question: 90-day LTV by signup cohort month.

Key : WITH, date_trunc, INTERVAL, SUM.

Pattern: Cohort LTV with fixed horizon.

SQL:

```sql
WITH u AS (
  SELECT
  user_id,
  date_trunc('month', signup_ts) AS cohort
  FROM users
),
r AS (
  SELECT
  u.cohort,
  b.user_id,
  sum(b.payout) AS rev
  FROM u
  JOIN bookings b
  ON u.user_id = b.user_id
 AND b.ts <= u.cohort + INTERVAL '90' day
  GROUP BY 1, 2
)
SELECT
  cohort,
  count(DISTINCT user_id) AS users,
  1.0 * sum(rev) / NULLIF(count(DISTINCT user_id), 0) AS ltv_90d
FROM r
GROUP BY 1
ORDER BY 1;
```

5. Theme: Funnel

Question: Session-level funnel rates (view→checkout→book).

Key : CASE, MAX, SUM, NULLIF.

Pattern: Binary step flags then ratios.

SQL:

```sql
WITH step AS (
  SELECT
  e.session_id,
  max(CASE WHEN event = 'view' THEN 1 END) AS v,
  max(CASE WHEN event = 'start_checkout' THEN 1 END) AS sc,
  max(CASE WHEN event = 'book' THEN 1 END) AS bk
  FROM events e
  GROUP BY 1
)
```

```
SELECT
  count(*) AS sessions,
  1.0 * sum(v) / count(*) AS p_view,
  1.0 * sum(sc) / NULLIF(sum(v), 0) AS p_sc_given_view,
  1.0 * sum(bk) / NULLIF(sum(sc), 0) AS p_book_given_sc
FROM step;
```

6. Theme: Rolling

Question: 7-day rolling bookings per channel.

Key : window SUM with ROWS frame.

Pattern: Rolling aggregates.

SQL:

```sql
SELECT
  date_trunc('day', b.ts) AS d,
  s.channel,
  sum(b.payout) AS rev,
  sum(sum(b.payout)) OVER (
  PARTITION BY s.channel
  ORDER BY date_trunc('day', b.ts)
  ROWS BETWEEN 6 PRECEDING AND CURRENT ROW
  ) AS rev_7d
FROM bookings b
JOIN sessions s
```

```
  ON b.session_id = s.session_id
GROUP BY 1, 2;
```

7. Theme: Retention

Question: Month-N retention from signup.

Key : WITH, date_trunc, DISTINCT, date_diff.

Pattern: Retention matrix by cohort and month N.

SQL:

```sql
WITH a AS (
  SELECT
  user_id,
  date_trunc('month', signup_ts) AS cohort
  FROM users
),
m AS (
  SELECT DISTINCT
  e.user_id,
  date_trunc('month', e.ts) AS active_m
  FROM events e
)
SELECT
  cohort,
  date_diff('month', cohort, active_m) AS m_n,
  count(DISTINCT m.user_id) AS active_users
FROM a
JOIN m
  ON a.user_id = m.user_id
GROUP BY 1, 2
ORDER BY 1, 2;
```

8. Theme: A/B mean

Question: A/B diff in booking rate with stats.

Key : SUM(CASE), proportions, sqrt.

Pattern: Diff-in-proportions with z-score.

SQL:

```sql
WITH base AS (
  SELECT
  t.variant,
  count(*) AS users,
  sum(CASE WHEN b.booking_id IS NOT NULL THEN 1 ELSE 0 END)
  AS booked
  FROM experiment_assignments t
  LEFT JOIN bookings b
  ON b.user_id = t.user_id
  GROUP BY 1
),
stats AS (
  SELECT
  variant,
  users,
  booked,
  1.0 * booked / users AS cr
  FROM base
)
SELECT
  sA.cr AS cr_A,
  sB.cr AS cr_B,
  (sB.cr - sA.cr) AS diff,
  ( sA.cr * (1 - sA.cr) / sA.users +
  sB.cr * (1 - sB.cr) / sB.users ) AS var,
  (sB.cr - sA.cr) / sqrt(var) AS z
FROM stats sA
CROSS JOIN stats sB
WHERE sA.variant = 'A'
  AND sB.variant = 'B';
```

9. Theme: CUPED

Question: CUPED-style adjusted mean for A/B.

Key : covar_samp, var_samp, window avg.

Pattern: Variance reduction on revenue.

SQL:

```sql
WITH d AS (
  SELECT
  t.variant,
  u.user_id,
  COALESCE(pre.pre_rev, 0.0) AS x,
  COALESCE(post.post_rev, 0.0) AS y
  FROM experiment_assignments t
  JOIN users u
  ON t.user_id = u.user_id
  LEFT JOIN (
  SELECT user_id, sum(payout) AS pre_rev
  FROM bookings
  WHERE ts < date '2025-01-01'
  GROUP BY 1
  ) pre USING (user_id)
  LEFT JOIN (
  SELECT user_id, sum(payout) AS post_rev
  FROM bookings
  WHERE ts >= date '2025-01-01'
  GROUP BY 1
  ) post USING (user_id)
),
theta AS (
  SELECT covar_samp(y, x) / NULLIF(var_samp(x), 0) AS th
  FROM d
)
SELECT
  variant,
  avg(y - th * (x - avg(x) OVER ())) AS cuped_mean
FROM d
CROSS JOIN theta
GROUP BY 1;
```

10. Theme: DiD

  Question: Geo DiD on bookings before and after launch.

  Key : WITH, CASE, GROUP BY.

  Pattern: 2×2 cells, compute DiD from aggregates.

  SQL:

```sql
WITH g AS (
  SELECT
  geo,
  CASE WHEN geo IN ('US_TX', 'US_FL') THEN 1 ELSE 0 END AS trt
  FROM sessions
  GROUP BY 1, 2
),
ba AS (
  SELECT
  CASE WHEN b.ts < date '2025-06-01' THEN 0 ELSE 1 END AS post,
  s.geo,
  count(*) AS books
  FROM bookings b
  JOIN sessions s
  ON b.session_id = s.session_id
  GROUP BY 1, 2
)
SELECT
  post, trt, sum(books) AS n
FROM ba
JOIN g USING (geo)
GROUP BY 1, 2;
```

11. Theme: Sessionize

Question: Build 30-min sessions from events.

Key : LAG, date_diff, running SUM.

Pattern: Gaps-and-islands.

SQL:

```sql
WITH e AS (
  SELECT
  user_id,
  ts,
  CASE
  WHEN lag(ts) OVER (
    PARTITION BY user_id ORDER BY ts
    ) IS NULL
   OR date_diff(
    'minute',
    lag(ts) OVER (PARTITION BY user_id ORDER BY ts),
    ts
    ) > 30
  THEN 1 ELSE 0
  END AS new_s
  FROM events
),
s AS (
  SELECT
  user_id,
  ts,
  sum(new_s) OVER (
  PARTITION BY user_id ORDER BY ts
  ) AS sid
  FROM e
)
SELECT * FROM s;
```

12. Theme: MTA simple

Question: Equal split across channels in 7-day lookback.

Key : array_agg, array_distinct, UNNEST, cardinality.

Pattern: Simple equal-weight attribution.

SQL:

```sql
WITH hist AS (
  SELECT
  b.booking_id,
  b.payout,
  array_distinct(array_agg(s.channel)) AS chs
  FROM bookings b
  JOIN sessions s
  ON b.user_id = s.user_id
 AND s.start_ts BETWEEN b.ts - INTERVAL '7' day AND b.ts
  GROUP BY 1, 2
),
explode AS (
  SELECT
  booking_id,
  payout,
  ch,
  cardinality(chs) AS k
  FROM hist
  CROSS JOIN UNNEST(chs) AS t(ch)
)
```

```
SELECT
  ch AS channel,
  sum(payout * 1.0 / k) AS attrib_rev
FROM explode
GROUP BY 1;
```

13. Theme: Percentiles

  Question: Time-to-book percentiles.

  Key : min_by, date_diff, approx_percentile.

  Pattern: Distribution summary.

  SQL:

```sql
WITH t AS (
  SELECT
  b.user_id,
  min_by(s.start_ts, s.start_ts) AS first_s,
  min_by(b.ts, b.ts) AS first_b
  FROM bookings b
  JOIN sessions s
  ON b.session_id = s.session_id
  GROUP BY 1
)
SELECT
  approx_percentile(
  date_diff('hour', first_s, first_b),
  ARRAY[0.5, 0.9, 0.95]
  ) AS p50_p90_p95
FROM t;
```

14. Theme: UTM parse

  Question: Extract UTM params from landing_url.

  Key : url_extract_parameter.

  Pattern: URL parsing for attribution.

  SQL:

```sql
SELECT
  session_id,
  url_extract_parameter(landing_url, 'utm_source') AS utm_source,
  url_extract_parameter(landing_url, 'utm_campaign') AS utm_campaign
FROM sessions;
```

15. Theme: De-dupe

  Question: Pick one device per user by latest seen.

  Key : max_by, GROUP BY.

  Pattern: Latest per key.

  SQL:

```sql
SELECT
  user_id,
  max_by(device, start_ts) AS primary_device
FROM sessions
GROUP BY 1;
```

16. Theme: Pivot

  Question: Pivot revenue by channel (wide).

  Key : conditional SUM.

  Pattern: Pivot-like conditional aggregation.

  SQL:

```sql
SELECT
  date_trunc('day', b.ts) AS d,
  sum(CASE WHEN s.channel = 'Paid Search' THEN b.payout END)
  AS paid_search,
  sum(CASE WHEN s.channel = 'Paid Social' THEN b.payout END)
  AS paid_social,
  sum(CASE WHEN s.channel = 'Affiliates' THEN b.payout END)
  AS affiliates
FROM bookings b
JOIN sessions s
  ON b.session_id = s.session_id
```

```
GROUP BY 1
ORDER BY 1;
```

17. Theme: Top paths

  Question: Top 10 channel paths in 7 days pre-booking.

  Key : array_agg ORDER BY, array_join.

  Pattern: Path sequencing.

  SQL:

```sql
WITH seq AS (
  SELECT
  b.booking_id,
  array_join(
  array_agg(s.channel ORDER BY s.start_ts),
  ' > '
  ) AS path
  FROM bookings b
  JOIN sessions s
  ON b.user_id = s.user_id
 AND s.start_ts BETWEEN b.ts - INTERVAL '7' day AND b.ts
  GROUP BY 1
)
SELECT
  path,
  count(*) AS n
FROM seq
GROUP BY 1
ORDER BY n DESC
LIMIT 10;
```

18. Theme: Anomaly

  Question: Z-score for daily spend by channel.

  Key : window AVG, STDDEV_SAMP, ROWS frame.

  Pattern: Rolling z-score.

  SQL:

```sql
WITH x AS (
  SELECT
  dt,
  channel,
  spend,
  avg(spend) OVER (
  PARTITION BY channel
  ORDER BY dt
  ROWS BETWEEN 14 PRECEDING AND CURRENT ROW
  ) AS mu,
  stddev_samp(spend) OVER (
  PARTITION BY channel
  ORDER BY dt
  ROWS BETWEEN 14 PRECEDING AND CURRENT ROW
  ) AS sd
  FROM channel_spend
)
SELECT
  dt, channel, spend,
  (spend - mu) / NULLIF(sd, 0) AS z
FROM x;
```

19. Theme: Dedup clicks

  Question: First click after impression within 24h.

  Key : LEFT JOIN, BETWEEN, min_by.

  Pattern: First-event within window.

  SQL:

```sql
SELECT
  i.impression_id,
  min_by(c.click_id, c.ts) AS first_click
FROM ad_impressions i
LEFT JOIN ad_clicks c
  ON c.user_id = i.user_id
 AND c.campaign_id = i.campaign_id
 AND c.ts BETWEEN i.ts AND i.ts + INTERVAL '24' hour
```

```
GROUP BY 1;
```

20. Theme: CPA

  Question: CPA by campaign using first conversion in 7 days.

  Key : min_by, CASE, NULLIF.

  Pattern: Horizon-trimmed first conversion.

  SQL:

```sql
WITH conv AS (
  SELECT
  c.campaign_id,
  c.click_id,
  min_by(b.booking_id, b.ts) AS first_book
  FROM ad_clicks c
  LEFT JOIN bookings b
  ON b.user_id = c.user_id
 AND b.ts BETWEEN c.ts AND c.ts + INTERVAL '7' day
  GROUP BY 1, 2
),
agg AS (
  SELECT
  campaign_id,
  sum(CASE WHEN first_book IS NOT NULL THEN 1 ELSE 0 END)
  AS convs
  FROM conv
  GROUP BY 1
)
SELECT
  s.campaign_id,
  sum(s.spend) / NULLIF(a.convs, 0) AS cpa
FROM channel_spend s
JOIN agg a
  ON s.campaign_id = a.campaign_id
GROUP BY 1;
```

21. Theme: HLL

  Question: Approx unique users per channel.

  Key : approx_distinct.

  Pattern: Scale-aware uniques.

  SQL:

```sql
SELECT
  s.channel,
  approx_distinct(b.user_id) AS users
FROM bookings b
JOIN sessions s
  ON b.session_id = s.session_id
GROUP BY 1;
```

~~~

22. Theme: Cohort ROAS

  Question: Signup-month cohort ROAS in 30 days.

  Key : WITH, date_trunc, INTERVAL, NULLIF.

  Pattern: Cohort revenue vs spend month.

  SQL:

```sql
WITH c AS (
  SELECT
  user_id,
  date_trunc('month', signup_ts) AS cohort
  FROM users
),
r AS (
  SELECT
  c.cohort,
  sum(b.payout) AS rev
  FROM c
  JOIN bookings b
  ON b.user_id = c.user_id
 AND b.ts <= c.cohort + INTERVAL '30' day
  GROUP BY 1
),
spend AS (
~~~

```
  SELECT
  date_trunc('month', dt) AS m,
  sum(spend) AS spend
  FROM channel_spend
  GROUP BY 1
)
SELECT
  r.cohort,
  r.rev,
  s.spend,
  1.0 * r.rev / NULLIF(s.spend, 0) AS roas
FROM r
JOIN spend s
  ON r.cohort = s.m;
```

23. Theme: Country mix

  Question: Share of revenue by country per week.

  Key : window SUM, ratio of sums.

  Pattern: Mix share by period.

  SQL:

```sql
SELECT
  date_trunc('week', b.ts) AS wk,
  u.country,
  sum(b.payout) AS rev,
  1.0 * sum(b.payout)
  / sum(sum(b.payout)) OVER (
    PARTITION BY date_trunc('week', b.ts)
  ) AS share
FROM bookings b
JOIN users u
  ON b.user_id = u.user_id
GROUP BY 1, 2;
```

24. Theme: Email lift

Question: Open→book conversion within 3 days of send.

Key : MAX(CASE), AVG, BETWEEN.

Pattern: Post-exposure outcome window.

SQL:

```sql
WITH got_email AS (
  SELECT user_id, sent_ts FROM emails
),
outcome AS (
  SELECT
  g.user_id,
  max(
  CASE
```

```
    WHEN b.ts BETWEEN g.sent_ts
     AND g.sent_ts + INTERVAL '3' day
    THEN 1
  END
  ) AS booked
  FROM got_email g
  LEFT JOIN bookings b
  ON b.user_id = g.user_id
  GROUP BY 1
)
SELECT avg(booked) AS conv_rate
FROM outcome;
```

SQL Applied 2 (PS)

SQL Applied 2 (PS)

@Perplexity

Common Analytics SQL Interview Questions & Solution Patterns

1. Aggregation

Typical Questions:

- "What's the total number of bookings per city per month?"
- "Find the average revenue per booking by campaign."

Pattern:

```
SELECT city, MONTH(booking_date) AS month, COUNT(*) AS bookings
FROM bookings
GROUP BY city, MONTH(booking_date)
```

Use GROUP BY for breakdowns, with aggregate functions like COUNT, SUM, AVG.

2. Filtering and Sorting

Typical Questions:

- "List bookings for Paris in January, sorted by revenue."

- "Show the top 10 campaigns with the highest conversion rate."

Pattern:

```
SELECT *
FROM bookings
WHERE city = 'Paris' AND booking_date BETWEEN '2025-01-01' AND '2025-01-31'
ORDER BY revenue DESC
LIMIT 10
```

Use WHERE for filters, ORDER BY and LIMIT for rankings.

3. Joins and Attribution

Typical Questions:

- "Total bookings per marketing campaign."
- "Attribute bookings to the last campaign click."

Pattern:

```
SELECT mc.campaign_name, COUNT(*) AS total_bookings
FROM bookings b
JOIN marketing_campaigns mc ON b.campaign_id = mc.campaign_id
GROUP BY mc.campaign_name
```

4. Window Functions (Rolling/Funnel/Ranking)

Typical Questions:

- "7-day rolling average of bookings per city."
- "Rank campaigns by conversions."

Pattern:

```
SELECT city, booking_date, bookings,

    AVG(bookings) OVER (PARTITION BY city ORDER BY booking_date ROWS BETWEEN 6 PRECEDING AND CURRENT ROW) AS avg_7d

FROM daily_bookings
```

Use OVER (...) for running, ranking, or partitioned calculations.

5. Funnel Analysis/Retention Cohorts

Typical Questions:

- "What percent of users made bookings after their first exposure to a campaign?"
- "What is user retention after 30/60/90 days?"

Pattern:

- Create first-touch and subsequent event subqueries using MIN(...) or LAG(...) functions, join or filter on intervals.
- Calculate percentages by dividing cohort outcomes by cohort size.

6. Case Statements for Segmentation

Typical Questions:

- "Classify bookings as 'High', 'Medium', or 'Low' revenue."

Pattern:

```
SELECT booking_id,

  CASE

    WHEN revenue >= 300 THEN 'High'

    WHEN revenue >= 150 THEN 'Medium'

    ELSE 'Low'
  END AS revenue_segment
FROM bookings
```

7. Missing Data & Time Series Completion

Typical Questions:

- "How to calculate moving averages if some dates are missing?"

Pattern:

- Generate a calendar table (sequence() in Presto), left join it to your data, fill NULLs with defaults.
- This ensures accurate window functions by date, not just by row count.

8. Self-Joins or CTEs for Sequential Analysis

Typical Questions:

- "Calculate days between campaign click and booking for each user."
- "Find month-over-month changes."

Pattern:

- Use WITH clauses (CTEs) to create intermediate result sets.
- Use LAG() or date arithmetic for calculating differences.

Schema

Bookings

booking_id	user_id	city	booking_date	campaign_id	revenue	device_type	rating

1	101	Paris	2025-01-15	201	250	mobile	5
2	102	London	2025-01-16	202	300	desktop	4
3	103	New York	2025-02-01	NULL	400	mobile	4
4	104	Paris	2025-02-17	203	225	tablet	3
5	101	Paris	2025-03-05	202	350	mobile	5

Marketing Campaigns

campaign_id	**campaign_name**	**channel**	**launch_date**	**promo_code**
201	Winter Blast	email	2025-01-10	WB2025
202	London Calling	social	2025-01-12	LC2025
203	City Escape	search	2025-02-10	CE2025
204	Global Spring	referral	2025-03-01	GS2025

Users

user_id	**user_email**	**signup_date**
101	alice@sample.com	2025-01-05
102	bob@sample.com	2025-01-06
103	carol@sample.com	2025-01-20
104	dave@sample.com	2025-02-14

Listings

listing_id	host_id	city	active	avg_rating
L10	H201	Paris	TRUE	4.7
L11	H202	London	TRUE	4.8
L12	H203	New York	FALSE	4.3
L13	H204	Paris	TRUE	4.6

Campaign Clicks

click_id	campaign_id	user_id	click_time	device_type
C101	201	101	2025-01-11	mobile
C102	202	102	2025-01-13	desktop
C103	203	104	2025-02-15	tablet
C104	201	103	2025-01-12	mobile

Problem Sets

Easy

Foundations

1. List all Airbnb bookings in January 2025, sorted by booking date.

```sql
SELECT booking_id
FROM bookings b
WHERE booking_date BETWEEN '2025-01-01' AND '2025-01-31'
ORDER BY booking_date DESC
```

2. Count the total number of listings in each city.

3. Find the average star rating for each listing.

4. Which users signed up in the past 30 days?

5. How many marketing campaigns ran last quarter?

6. Find the unique number of guests per booking.

7. Count users who have never made a booking.

8. What is the total revenue generated in February 2025?

9. List distinct marketing channels (e.g., email, social, referral) used in the last year.

10. What percentage of bookings were attributed to paid campaigns?

Medium

Aggregations, Joins & More

11. **For each campaign, show total bookings and average booking value.**
12. **List the top 5 listings by bookings, including average star rating.**
13. **Retrieve the number of active listings per host, filtered by a minimum average rating of 4.5.**
14. **Find the daily number of bookings for each city, showing the 7-day moving average (fill in missing dates!).**
15. **Calculate click-through rate (CTR) by campaign: ratio of users clicking to users exposed.**

16. **Show the percentage of first-time bookings coming from each marketing channel.**
17. **Find users who made bookings from more than one marketing channel.**
18. **Identify the most common device type used to make bookings.**
19. **Find the ratio of bookings made using promotions vs. regular price.**
20. **Which cities have the highest average booking duration?**

Hard

Window Functions, Cohorts, Attribution, Advanced Analytics

21. **For each listing, find the month-over-month growth rate in bookings this year.**
22. **List the top 3 campaigns by conversions and compute their conversion funnel drop-off at each stage (impressions → clicks → bookings).**
23. **Attribute bookings to the marketing campaign that had the last touch before the booking event, for each user.**
24. **Compute retention rates: What percentage of users booked more than once within 90 days of their first booking?**
25. **For each campaign, calculate the median time between first exposure and first booking.**
26. **Identify properties that saw an increase in bookings after a major event or campaign launch—use window functions.**
27. **Which listings saw a decline in ratings after a price change promoted through a campaign?**
28. **Segment Airbnb users into cohorts by signup month and calculate booking rates per cohort over the following 6 months.**
29. **Analyze the effectiveness of a recent A/B test: Compare booking conversions between test and control groups.**
30. **Given a table of bookings, campaigns, and user actions, determine which sequence of marketing activities most strongly predicts a booking, using CTEs and aggregation.**

Solution Sets

1. List all Airbnb bookings in January 2025, sorted by booking date.

```
SELECT *
FROM bookings
WHERE booking_date BETWEEN DATE '2025-01-01' AND DATE '2025-01-31'
ORDER BY booking_date;
```

2. Count the total number of listings in each city.

```
SELECT city, COUNT(*) AS num_listings
FROM listings
GROUP BY city;
```

3. Find the average star rating for each listing.

```
SELECT listing_id, AVG(rating) AS avg_rating
FROM bookings
GROUP BY listing_id;
```

(Assumes the bookings table references each listing_id.)

4. Which users signed up in the past 30 days?

```
SELECT user_id, user_email
FROM users
WHERE signup_date >= CURRENT_DATE - INTERVAL '30' DAY;
```

5. How many marketing campaigns ran last quarter?

```
SELECT COUNT(*) AS num_campaigns

FROM marketing_campaigns

WHERE launch_date BETWEEN DATE '2025-04-01' AND DATE '2025-06-30';
```

(Adjust dates according to the "last quarter" at your interview date.)

6. Find the unique number of guests per booking.

```
SELECT booking_id, COUNT(DISTINCT user_id) AS unique_guests
FROM bookings
GROUP BY booking_id;
```

7. Count users who have never made a booking.

```
SELECT COUNT(*) AS users_no_booking
FROM users u
LEFT JOIN bookings b ON u.user_id = b.user_id
WHERE b.booking_id IS NULL;
```

8. What is the total revenue generated in February 2025?

```
SELECT SUM(revenue) AS feb_revenue
FROM bookings
WHERE booking_date BETWEEN DATE '2025-02-01' AND DATE '2025-02-28';
```

9. List distinct marketing channels used in the last year.

```
SELECT DISTINCT channel
FROM marketing_campaigns
WHERE launch_date >= CURRENT_DATE - INTERVAL '1' YEAR;
```

10. What percentage of bookings were attributed to paid campaigns?

```
SELECT
  100.0 * COUNT(DISTINCT booking_id) FILTER (WHERE campaign_id IS NOT NULL) / COUNT(*) AS percent_paid
FROM bookings;
```

11. For each campaign, show total bookings and average booking value.

```
SELECT c.campaign_id, c.campaign_name, COUNT(b.booking_id) AS total_bookings, AVG(b.revenue) AS avg_booking_value
FROM marketing_campaigns c
LEFT JOIN bookings b ON c.campaign_id = b.campaign_id
GROUP BY c.campaign_id, c.campaign_name;
```

12. List the top 5 listings by bookings, including average star rating.

```
SELECT listing_id, COUNT(*) AS total_bookings, AVG(rating) AS avg_rating
FROM bookings
GROUP BY listing_id
ORDER BY total_bookings DESC
LIMIT 5;
```

13. Retrieve the number of active listings per host, filtered by a minimum average rating of 4.5.

```
SELECT host_id, COUNT(listing_id) AS active_listings
FROM listings
WHERE active = TRUE AND avg_rating >= 4.5
GROUP BY host_id;
```

14. Find the daily number of bookings for each city, showing the 7-day moving average (fill in missing dates).

(Requires generating a date calendar; see previous instructions.)

15. Calculate click-through rate (CTR) by campaign.

sql

```
SELECT
  c.campaign_id,
  COUNT(DISTINCT clk.user_id) AS unique_clicks,
  -- Suppose we have an 'impressions' table or can estimate
  COUNT(DISTINCT clk.user_id) * 1.0 / NULLIF(SUM(impressions), 0) AS ctr
FROM marketing_campaigns c
```

```
LEFT JOIN clicks clk ON c.campaign_id = clk.campaign_id
GROUP BY c.campaign_id;
```

16. Show the percentage of first-time bookings coming from each marketing channel.

```
WITH ft_bookings AS (
  SELECT user_id, MIN(booking_date) AS first_booking
  FROM bookings
  GROUP BY user_id
)
SELECT
  mc.channel,
  COUNT(*) AS num_first_time,
  100.0 * COUNT(*) / (SELECT COUNT(*) FROM ft_bookings) AS pct_first_time
FROM bookings b
JOIN ft_bookings ft ON b.user_id = ft.user_id AND b.booking_date = ft.first_booking
JOIN marketing_campaigns mc ON b.campaign_id = mc.campaign_id
GROUP BY mc.channel;
```