The Complete SQL Curriculum for Data Science and AI Interviews

Executive Summary: A Strategic Path to SQL Mastery for Data Science

This document outlines a comprehensive curriculum designed to build expert-level SQL proficiency for Data Scientist, Senior Data Scientist, and AI/ML Engineer roles. Preparation for these interviews requires a specialized approach.[1]
This is not an exhaustive list of every SQL command. Rather, it is a strategically prioritized learning path that reflects the realities of the data science interview. The curriculum de-emphasizes database administration tasks (like permissions or server configuration) and focuses intensely on what interviewers for analytical roles test: advanced data querying, complex data manipulation, analytical pattern recognition, and performance optimization. The objective is to move beyond simple command memorization and develop the ability to "think in SQL" as an analyst.

Module 1: The SQL Landscape and Database Fundamentals

Objective: To build a foundational "map" of the SQL language and the database structures it operates on. This module covers the different "modes" of SQL (defining, manipulating, querying) and the rules that ensure data integrity.

1.1 The Five Categories of SQL Commands

SQL commands are logically grouped into five categories based on their function.[8]
Understanding this structure provides a clear mental model for how to interact with a database.

Category	Full Name	Purpose	Key Commands
DQL	Data Query Language	To fetch and retrieve data from the database.	SELECT [10]

DDL	Data Definition Language	To define and modify the *structure* of database objects.	CREATE, ALTER, DROP, TRUNCATE [10]
DML	Data Manipulation Language	To manipulate the *data* stored within database objects.	INSERT, UPDATE, DELETE [10]
TCL	Transaction Control Language	To manage sets of changes (transactions) safely.	COMMIT, ROLLBACK, SAVEPOINT [10]
DCL	Data Control Language	To manage access permissions and user controls.	GRANT, REVOKE [10]

For data science roles, DQL is the most critical category, followed by DML and DDL. TCL and DCL are important for context but are less frequently tested in analytics interviews.

1.2 DDL: Building the "Container"

DDL commands create and define the database's structure.[12]

- **CREATE**: Used to build new database objects, most commonly a new table. This involves defining the table's columns and their respective data types.[12]
- **ALTER**: Used to modify an existing object's structure, such as adding a new column, deleting a column, or changing a column's data type.[12]
- **DROP**: This command *permanently* deletes a database object (like a table, index, or view) from the database. It removes the structure, all data, and all related objects.[12]
- **TRUNCATE**: This command removes *all records* (rows) from a table but leaves the table's structure intact. It is a DDL operation, not DML, because it is typically faster and uses fewer system resources than DELETE as it does not log individual row deletions.[12]

A classic interview question is to differentiate DELETE, TRUNCATE, and DROP.[1]

- **DROP TABLE employees;**: Deletes the employees table structure, all its data, and any associated indexes or constraints. It's irreversible.
- **TRUNCATE TABLE employees;**: Deletes all rows from the employees table. The empty table structure remains. It's fast but cannot be filtered (it's all or nothing).
- **DELETE FROM employees;**: This is a DML command (see below). It removes all rows one by one and logs each deletion. It is slower but more flexible, as it can be combined with a WHERE clause to remove only *specific* rows.

1.3 DML: Manipulating the Data

DML commands interact with the data *inside* the tables.[12]

- **INSERT**: Adds new rows (records) to a table.[17]

- **UPDATE**: Modifies existing data within one or more rows. It is almost always used with a WHERE clause to specify *which* rows to change.[17]
- **DELETE**: Removes specific rows from a table. It is used with a WHERE clause to filter which rows to remove.[17]

1.4 TCL: Ensuring Safe Changes

TCL commands manage transactions. A transaction is a sequence of one or more SQL statements (like an INSERT followed by an UPDATE) that are treated as a single "all or nothing" unit of work.[18]

- **COMMIT**: Permanently saves all changes made during the current transaction.[14]
- **ROLLBACK**: Undoes all changes made during the current transaction, returning the database to the state it was in before the transaction began.[14]
- **SAVEPOINT**: Sets a named checkpoint *within* a transaction. This allows for a partial ROLLBACK to that specific point, rather than undoing the entire transaction.[14]

1.5 Core Data Types

Choosing the correct data type is essential for data integrity and storage efficiency.

- **Numeric**: Includes INT (and variants like SMALLINT, BIGINT) for whole numbers, NUMERIC/DECIMAL for exact fixed-point precision (ideal for currency), and FLOAT/REAL for approximate floating-point values (common in scientific applications).[24]
- **String**: CHAR(n) is fixed-length, padding with spaces up to n (e.g., for state abbreviations). VARCHAR(n) is variable-length, using only the space needed *up to* n (ideal for names, comments).[1]
- **Date/Time**: DATE stores only the date. TIME stores only the time. DATETIME or TIMESTAMP store both date and time, though TIMESTAMP is often timezone-aware and preferred.[24]
- **Other**: BOOLEAN (or BIT in SQL Server) stores TRUE or FALSE values.[24] JSON is a modern data type for storing semi-structured data within a relational table.[24]

1.6 Data Integrity: Constraints

Constraints are rules applied to columns to enforce data integrity and accuracy.[27]

- **NOT NULL**: Ensures a column cannot contain a NULL (missing) value.[27]
- **UNIQUE**: Ensures all values in a column (or a set of columns) are distinct.[27]
- **PRIMARY KEY (PK)**: A constraint that combines NOT NULL and UNIQUE. It serves as the unique identifier for a row in a table. A table can have only one Primary Key. This

enforces *entity integrity*.[27]

- **FOREIGN KEY (FK)**: A column (or set of columns) in one table that refers to the PRIMARY KEY in another table. This creates the link between tables and enforces *referential integrity*, ensuring that a row in a "child" table cannot be created without a corresponding row in the "parent" table.[27] This PK-FK relationship is the fundamental mechanism that enables relational database joins.[30]
- **CHECK**: A generic constraint that validates data based on a custom boolean expression (e.g., CHECK (salary > 0)).[27]

Module 2: The Core: Data Query Language (DQL)

Objective: To master the SELECT statement, the single most-used command for all data-driven roles.

2.1 The SQL Order of Operations: A Critical Mental Model

A common source of confusion for SQL learners is *why* certain commands work in some places and not others. The reason is the *logical execution order*, which is different from the *written order* of a query.

Written Order (Syntax)	Logical Execution Order	Purpose
SELECT	1. FROM / JOIN	Acquires the tables.
FROM	2. WHERE	Filters individual rows.
WHERE	3. GROUP BY	Aggregates rows into groups.
GROUP BY	4. HAVING	Filters the new groups.
HAVING	5. SELECT	Selects the final columns (and runs functions).
ORDER BY	6. ORDER BY	Sorts the final output.
LIMIT	7. LIMIT / FETCH	Picks the top N rows.

This logical order explains, for example, why a SELECT alias (e.g., SELECT salary / 12 AS monthly_salary) *cannot* be used in a WHERE clause (the SELECT step, #5, hasn't happened when the WHERE step, #2, is executed). This concept will be revisited.

2.2 The SELECT Statement

This is the DQL command used to retrieve data. Its basic syntax involves specifying the columns to retrieve and the table to retrieve them from.[12]

- SELECT column1, column2 FROM table_name;

- SELECT * FROM table_name; (Selects all columns)

2.3 Filtering Rows with WHERE

The WHERE clause filters rows *before* any grouping, based on specific conditions.[31]

- **Comparison Operators**: =, !=, <>, >, <, >=, <=.[31]
- **Logical Operators**: AND, OR, NOT are used to combine multiple conditions.[31]

2.4 Specialized WHERE Operators

These are essential tools for efficient filtering in data analysis.

- **IN**: Used to replace multiple OR conditions (e.g., WHERE city IN ('London', 'Paris', 'Tokyo')).[31]
- **BETWEEN**: Filters for values within an *inclusive* range (e.g., WHERE hire_date BETWEEN '2023-01-01' AND '2023-12-31').[31]
- **LIKE**: Used for pattern matching in strings. It uses two wildcards:
  - %: Matches zero or more characters (e.g., 'Pat%' matches 'Patrick' and 'Pat').[31]
  - _: Matches exactly one character (e.g., 'Patr_ck' matches 'Patrick').[31]
- **IS NULL** / **IS NOT NULL**: The correct way to check for missing data. A common mistake is writing WHERE column = NULL. This will always fail because NULL is not a value; it is a state (the *unknown*). SQL uses three-valued logic (TRUE, FALSE, UNKNOWN), and NULL = NULL evaluates to UNKNOWN, not TRUE. The only correct syntax is WHERE column IS NULL.[31]

2.5 Sorting Results with ORDER BY

The ORDER BY clause sorts the final result set. This is one of the last operations performed.[35]

- It can sort by one or more columns: ORDER BY department, last_name.[33]
- Sorting direction can be specified as ASC (ascending, default) or DESC (descending).[31]
- **Handling NULLs**: In data analysis, NULL values can clutter a sorted list. Most dialects support NULLS FIRST or NULLS LAST to control their placement.[36]
  - ORDER BY hire_date DESC NULLS LAST; (Shows most recent hires first, with non-hires at the bottom).

2.6 Limiting Results

This is a practical necessity in a hands-on environment to prevent fetching millions of rows.

- **LIMIT n**: Used in PostgreSQL and MySQL (e.g., LIMIT 10).[31]
- **TOP n**: Used in SQL Server (e.g., SELECT TOP 10...).[37]
- **FETCH NEXT n ROWS ONLY**: The ANSI SQL standard, often used for pagination.[37]
- **OFFSET**: Used with LIMIT or FETCH to skip a number of rows, enabling pagination (e.g., LIMIT 10 OFFSET 20 retrieves rows 21-30).[38]

Module 3: Aggregation and Grouping

Objective: To move from retrieving individual rows to summarizing and analyzing data—the core of data science.

3.1 Aggregate Functions

Aggregate functions perform a calculation on a set of rows and return a single summary value.[40]

- **COUNT(column)**: Counts the number of non-NULL rows in that column.[40]
  - **COUNT(*)**: Counts the total number of rows in the group.
  - **COUNT(DISTINCT column)**: Counts the number of *unique* non-NULL values.
- **SUM(column)**: Adds all non-NULL values in the column.[40]
- **AVG(column)**: Calculates the average of all non-NULL values.[40]
- **MIN(column)**: Finds the minimum value in the column.[40]
- **MAX(column)**: Finds the maximum value in the column.[40]

3.2 Grouping Data with GROUP BY

The GROUP BY clause is used to "segment" a table into groups *before* aggregate functions are applied.[32] For example, SELECT department, AVG(salary) FROM employees GROUP BY department; will first divide the employees table into groups (one for 'Sales', one for 'Engineering', etc.) and *then* calculate the AVG(salary) for each group.
A critical rule applies: If a GROUP BY clause is used, any non-aggregated column in the SELECT list *must* also be present in the GROUP BY clause. This is because the database cannot know *which* individual employee_name to show for the 'Engineering' group's average salary.

3.3 The Critical Distinction: WHERE vs. HAVING

This is one of the most frequently asked SQL interview questions.[1] The difference is simple but

fundamental.

- **WHERE filters rows**.[43]
- **HAVING filters groups**.[43]

This distinction is a direct consequence of the logical order of operations.[44]

1. The WHERE clause is executed *before* the GROUP BY clause. It filters the raw rows *before* they are grouped and aggregated.
2. The HAVING clause is executed *after* the GROUP BY clause. It filters the new, aggregated "group rows" that were just created.

This also dictates *what* can be in each clause:

- WHERE *cannot* contain aggregate functions (e.g., WHERE COUNT(*) > 10 is invalid) because the aggregation has not happened yet.[44]
- HAVING *must* contain aggregate functions (e.g., HAVING COUNT(*) > 10 is valid) because it operates *on* the aggregated results.[47]

Criterion	WHERE Clause	HAVING Clause
Purpose	Filters individual rows.	Filters aggregated groups.
Execution Order	**Before** GROUP BY.	**After** GROUP BY.[44]
Use of Aggregates	**Cannot** be used (e.g., SUM(), COUNT()).	**Must** be used (or operate on grouped columns).[44]
Example	WHERE country = 'USA'	HAVING SUM(sales) > 1000

Module 4: Essential SQL Functions for Data Analysis

Objective: To master the scalar functions used for data cleaning, transformation, and feature engineering. These functions operate on single values, as opposed to the aggregate functions in the previous module.

4.1 String Functions

String functions are used to manipulate text data, a common task in data cleaning and feature engineering.[129]

- **CONCAT**: Combines (concatenates) two or more strings into a single string.
- **SUBSTRING**: Extracts a portion of a string. You must specify the string, the starting position, and the length.
- **TRIM / LTRIM / RTRIM**: Removes whitespace (or other specified characters) from the beginning (LTRIM), end (RTRIM), or both sides (TRIM) of a string.
- **UPPER / LOWER**: Converts a string to all uppercase or all lowercase. This is critical for standardizing categorical data before grouping.
- **REPLACE**: Replaces all occurrences of a specific substring with another substring.

- **POSITION / CHARINDEX / INSTR**: Finds the starting position of a substring within a larger string.

4.2 Date/Time Functions

These functions are fundamental for any time-series analysis, cohort analysis, or creating time-based features.

- **EXTRACT(part FROM date)**: Retrieves a specific part (like MONTH, YEAR, DAY, HOUR) from a date or timestamp value.
- **DATE_TRUNC(part, date)**: Truncates a date or timestamp to a specific level of precision (like month, week, or day). This is essential for grouping events into consistent time buckets.
- **DATE_ADD / DATE_SUB**: (Syntax varies by dialect, e.g., DATEADD) Adds or subtracts a specified time interval (like 7 days or 1 month) from a date.
- **DATEDIFF(part, start_date, end_date)**: Calculates the number of specified time units (part) between two dates.

4.3 Conversion and Conditional Functions

These functions are used to handle data type issues and NULL values, which are notoriously common in real-world data.

- **CAST(expression AS target_type)**: Explicitly converts a value from one data type to another (e.g., a VARCHAR string '2024-01-01' to a DATE type). This is an explicit conversion, unlike implicit conversions that SQL may try to perform automatically.
- **COALESCE(expr1, expr2,...)**: Returns the *first non-null expression* in its argument list. This is a powerful tool for data cleaning and providing default values (e.g., COALESCE(column_value, 0) replaces all NULLs with 0).[130]
- **NULLIF(expr1, expr2)**: A common interview question. It returns NULL *if* the two expressions are equal. If they are not equal, it returns the first expression. Its primary use case is to prevent division-by-zero errors (e.g., SUM(sales) / NULLIF(SUM(orders), 0)).

Module 5: Combining Disparate Data Sets

Objective: To combine data from multiple tables. Real-world data is almost never in a single, flat file, making this the most essential practical skill.

5.1 Part 1: Relational Joins (Column-based)

JOINs combine tables *horizontally* (side-by-side) based on a logical relationship, or join condition, between their columns.[48] This condition is typically the match between a Foreign Key in one table and a Primary Key in another.[30]

- **INNER JOIN**: The default. Returns only rows that have a matching value in *both* tables.[49]
- **LEFT OUTER JOIN (or LEFT JOIN)**: Returns *all* rows from the *left* table and only the matched rows from the right table. If no match is found, columns from the right table will be NULL.[49]
- **RIGHT OUTER JOIN (or RIGHT JOIN)**: The reverse of a LEFT JOIN. Returns *all* rows from the *right* table and only the matched rows from the left table.[49]
- **FULL OUTER JOIN**: Returns *all* rows from *both* tables, filling in NULLs on either side when a match is not present.[49]
- **CROSS JOIN**: Returns the Cartesian product—every row from the first table combined with every row from the second table. This is rarely used and often an indicator of a missing ON clause.[49]

JOIN Type	Visual (Venn Diagram)	Description
INNER JOIN	Intersection only	Returns only rows that match in both Table A and Table B.
LEFT JOIN	All of Table A	Returns all rows from Table A, plus matching rows from Table B.
RIGHT JOIN	All of Table B	Returns all rows from Table B, plus matching rows from Table A.
FULL OUTER JOIN	All of A and B	Returns all rows from both tables, matching them where possible.

5.2 The SELF JOIN Technique

A SELF JOIN is not a separate SQL command; it is a *technique* where a table is joined to *itself*.[50] This is done by giving the table two different aliases (e.g., FROM employees e JOIN employees m...). It is used to query hierarchical data or compare rows within the same table.[51]

- Hierarchical Example: Finding an employee's manager.
  SELECT e.name AS employee, m.name AS manager FROM employees e INNER JOIN employees m ON e.manager_id = m.employee_id;
- Comparison Example: Finding students who took MTH251 and then MTH252.51
  SELECT c1.student_id, c1.grade AS mth251_grade, c2.grade AS mth252_grade FROM student_course c1 JOIN student_course c2 ON c1.student_id = c2.student_id WHERE

c1.course = 'MTH251' AND c2.course = 'MTH252';

5.3 Part 2: Set Operators (Row-based)

Set operators combine the result sets of two or more SELECT statements *vertically* (stacking them). This requires that the SELECT lists in each query have the same number of columns and compatible data types.[52]

- **UNION**: Combines two result sets and *removes duplicate rows*.[54] This duplicate-removal step (an implicit sort) can be computationally expensive.[56]
- **UNION ALL**: Combines two result sets and *keeps all rows*, including duplicates.[54]
- **UNION vs. UNION ALL**: This is a common interview question.[5] Because UNION ALL skips the expensive duplicate-removal step, it is significantly more performant. The best practice is to *always* use UNION ALL unless there is a specific analytical reason to find only the distinct rows.[55]
- **INTERSECT**: Returns only the rows that appear in *both* result sets.[52]
- **EXCEPT (or MINUS in Oracle)**: Returns the distinct rows from the first result set that are *not* found in the second result set.[52] This is a powerful and clean way to find "users who did action A but not action B."

Module 6: Advanced Query Constructs for Readability and Power

Objective: To master the constructs that separate junior from senior candidates. These tools help write clean, readable, and maintainable code for highly complex analysis.

6.1 Subqueries

A subquery (or inner query) is a SELECT query nested inside another SQL statement (like SELECT, FROM, or WHERE).[57]

- **Scalar Subquery**: A subquery that returns a single, scalar value (one row, one column). It can be used anywhere a single value is expected, such as in the SELECT list or WHERE clause.[58]
    - SELECT e.name, (SELECT AVG(salary) FROM employees) AS avg_company_salary FROM employees e;
- **Multi-row Subquery**: A subquery that returns a list of values (one column, multiple rows). It is typically used with operators like IN, ANY, or ALL.[58]
    - SELECT * FROM products WHERE product_id IN (SELECT product_id FROM order_items WHERE quantity > 100);
- **Correlated Subquery**: A subquery that references columns from the *outer* query. This

type of query is evaluated once *for every row* processed by the outer query, which can be very slow and inefficient.[58]

6.2 Common Table Expressions (CTEs)

A Common Table Expression (CTE) is a temporary, named result set defined using the WITH keyword. It exists only for the duration of a single query.[61]
This construct is a critical differentiator for senior-level candidates. While complex, nested subqueries can get the correct answer, they are notoriously difficult to read and debug. CTEs allow a query to be broken down into logical, sequential "steps," dramatically improving readability and maintainability.[61]
In an interview, a candidate who refactors a 4-level nested subquery into 4 sequential CTEs demonstrates they write code that a *team* can support.
Subquery (Hard to Read):

```
SELECT * FROM (SELECT department, AVG(salary) AS avg_sal FROM employees GROUP BY department) AS dept_sal WHERE avg_sal > (SELECT AVG(salary) FROM employees);
```

CTE (Readable):

```
WITH dept_sal AS (
SELECT department, AVG(salary) AS avg_sal
FROM employees
GROUP BY department
),
company_avg AS (
SELECT AVG(salary) AS avg_sal
FROM employees
)
SELECT * FROM dept_sal
WHERE dept_sal.avg_sal > (SELECT avg_sal FROM company_avg);
```

6.3 Other Database Objects

- **VIEWS**: A virtual table based on the result of a SELECT query.[66] It is stored as a query definition, not as data. Views are used to simplify complex logic (hiding joins from end-users) or for security (e.g., creating a view that only shows non-sensitive columns).[15]
- **Stored Procedures**: A precompiled collection of one or more SQL statements stored in the database.[66] They can accept parameters, contain complex control-of-flow logic (IF, WHILE), and perform transactions. They are executed as a single call (e.g., EXECUTE sp_my_procedure) and *cannot* be used inside a SELECT statement.[15]
- **User-Defined Functions (UDFs)**: Routines that accept parameters and return a value

(either a single scalar value or a table).[66] Unlike procedures, UDFs *can* be used inline within SQL statements, just like built-in functions.[68]

Module 7: The Analyst's Toolkit: Window Functions

Objective: To master window functions. This is arguably the **single most-tested advanced SQL topic** in data science interviews.[5] Their power comes from their ability to perform aggregate-like calculations *without* collapsing rows with a GROUP BY clause.

7.1 Core Concepts: OVER() and PARTITION BY

The OVER() clause is what defines a function as a window function. It defines the "window" or set of rows the function should operate on.[71]
The PARTITION BY clause, used inside OVER(), is the GROUP BY equivalent for window functions. It divides the data set into "partitions" or groups, and the window function is applied and *reset* for each partition.[71]

- AVG(salary) OVER (): Calculates the average salary of the *entire* table.
- AVG(salary) OVER (PARTITION BY department): Calculates the average salary *for each department*, but still returns every employee row.

7.2 Ranking Functions

This group is used to solve the classic "find the Nth highest salary" interview question.[2]

- **ROW_NUMBER()**: Assigns a unique, sequential number to each row within its partition (e.g., 1, 2, 3, 4).[71]
- **RANK()**: Assigns a rank based on the ORDER BY clause. It gives *tied values the same rank* and then *skips subsequent ranks* (e.g., 1, 2, 2, 4).[71]
- **DENSE_RANK()**: Assigns a rank based on the ORDER BY clause. It gives *tied values the same rank* but *does not skip* subsequent ranks (e.g., 1, 2, 2, 3).[71]

Value	ROW_NUMBER()	RANK()	DENSE_RANK()
100	1	1	1
90	2	2	2
90	3	2	2
80	4	4	3

To find the "second highest salary," the correct function is DENSE_RANK().

7.3 Analytic Functions (Time-Series and Comparison)

These functions allow a row to access data from other rows within its window.

- **LEAD()**: Accesses data from a *subsequent* row (peeking into the future).[73]
- **LAG()**: Accesses data from a *previous* row (glancing into the past).[73]

These two functions are the foundation of time-series analysis in SQL. To calculate day-over-day sales growth, one would use:
SELECT date, sales, LAG(sales, 1) OVER (ORDER BY date) AS previous_day_sales FROM....76

7.4 Filtering Window Results with QUALIFY

This topic demonstrates a modern, senior-level understanding of SQL.

- **The Problem**: How to filter the results of a window function, for example, to "find the top-ranked row per group"?
- **The Error**: One *cannot* write WHERE RANK() OVER (...) = 1. This is because, per the logical order of operations, the WHERE clause (Step 2) is executed *long before* window functions (which run with SELECT, Step 5) are computed.[44]
- The "Old Way": The traditional solution is to use a CTE or a subquery to first compute the rank, and then filter on that rank in an outer query:
  WITH RankedData AS (SELECT..., RANK() OVER (...) AS rnk FROM...)
  SELECT * FROM RankedData WHERE rnk = 1;
- The "New Way" (QUALIFY): Some modern data warehouses (like BigQuery, Databricks, Redshift) support the QUALIFY clause. This clause is executed after window functions are computed.46 It allows for a much cleaner and more efficient query:
  SELECT..., RANK() OVER (PARTITION BY... ORDER BY...) FROM... QUALIFY RANK() OVER (PARTITION BY... ORDER BY...) = 1;

Module 8: Advanced Traversal and Data Generation

Objective: To handle complex hierarchical data (like organizational charts) or graph data (like social networks) using Recursive CTEs.

8.1 Recursive CTE Structure

A Recursive CTE is a special type of CTE that references *itself* to perform iterative operations.[81] It consists of two main parts, combined with a UNION ALL:

1. **Anchor Member**: The base query that runs *once* to establish the starting set of the recursion. This is the "base case" (e.g., SELECT employee_id, name, 1 AS level FROM

employees WHERE manager_id IS NULL).[81]
2. **Recursive Member**: The query that runs *iteratively*. It references the CTE itself and joins it to the next level of the hierarchy (e.g., SELECT e.employee_id, e.name, r.level + 1 FROM employees e JOIN recursive_cte r ON e.manager_id = r.employee_id).[81]

8.2 Termination Condition

A common point of confusion is how the recursion stops. There is no explicit WHILE loop. The termination is *implicit*: the recursion automatically stops when the **recursive member** query returns zero new rows.[83] As a safeguard against infinite loops (e.g., cyclical data), most databases provide a MAXRECURSION option.[86]

8.3 Applications

- **Organizational Hierarchies**: Finding all direct and indirect reports for a manager.[81]
- **Graph Traversal**: Finding all possible paths between two nodes in a network.[82]

Module 9: Performance Tuning and Database Internals

Objective: To answer the senior-level interview question: "Your query is running slow. How do you speed it up?".[5] This demonstrates an understanding of *how* the database actually works.

9.1 Database Indexing

An index is a data structure (a "lookup table") that speeds up data retrieval operations on a table, at the cost of slightly slower writes.[90]

- **B-Tree Index**: The default, all-purpose index. It stores data in a sorted, balanced tree. It is excellent for a wide variety of queries, including *range* queries (e.g., WHERE age > 30 or WHERE date BETWEEN...).[34]
- **Hash Index**: Stores a hash of the column value. It is blazing fast for *exact equality* checks (WHERE username = 'test') but is useless for range queries, as the hashed values are not in a logical order.[34]
- **Bitmap Index**: A specialized index for Data Warehousing (OLAP) environments. It is only effective on *low-cardinality* columns (columns with few distinct values, like 'Gender' or 'Region'). It is very fast for complex analytical queries that filter on multiple such columns.[92]

Index Type	Best For (Use Case)	Avoid For (Drawbacks)
B-Tree	General purpose, OLTP. High-cardinality columns. Range queries (>, <, BETWEEN).[34]	Low-cardinality columns in DWH (less efficient than Bitmap).[92]
Hash	Exact equality lookups (=) only. Blazing fast for this one task.[34]	Range queries (useless). Frequent updates.[90]
Bitmap	OLAP / Data Warehouse. Low-cardinality columns (e.g., 'Gender', 'Region').[92]	OLTP / Write-heavy systems. High-cardinality columns (e.g., 'user_id').[92]

9.2 Query Optimization with EXPLAIN

The EXPLAIN (or EXPLAIN ANALYZE in Postgres) command is the primary tool for query optimization. It does not run the query but instead shows the *query execution plan*—the step-by-step "recipe" the database *plans* to use to get the data.[93]

- **How to Read a Plan**: The plan is a *tree* of nodes. To understand the flow, read it from the *inside out* (most indented) to the *outside in* (least indented).[93]
- **Identifying Bottlenecks**: The most common problem to look for is a **Seq Scan (Sequential Scan)** on a large table. This means the database is reading every single row. If this scan is part of a WHERE clause on an indexed column, it means the index is not being used, which is a major performance problem to investigate.[96] The goal is often to replace a Seq Scan with an Index Scan.
- **Understanding Output**: EXPLAIN ANALYZE provides *estimates* vs. *actuals*. Key metrics include "cost" (an arbitrary estimate of work), "actual time" (the real runtime), and "rows" (estimated rows vs. actual rows). A large mismatch between estimated and actual rows can indicate stale statistics, leading the planner to choose a poor plan.[93]

Module 10: Transaction Control and Concurrency

Objective: To understand the theoretical guarantees (ACID) and performance trade-offs (Isolation Levels) that make databases reliable. This is deep technical context expected in senior roles.

10.1 The Four ACID Properties

ACID is an acronym for the four guarantees of a reliable transaction.[100]

- **Atomicity**: "All or nothing." A transaction is an indivisible unit. It either fully succeeds, or

it is fully rolled back as if it never happened.[100]
- **Consistency**: A transaction moves the database from one *valid* state to another. It ensures all constraints and rules are upheld.[100]
- **Isolation**: Ensures that concurrent transactions do not interfere with each other. The effect of one transaction is not visible to others until it is committed.[100]
- **Durability**: Guarantees that once a transaction is committed, its changes are permanent and will survive system failures.[100]

10.2 Concurrency Problems

Isolation (the "I" in ACID) is not absolute; it is a spectrum. Lower levels of isolation are faster but can lead to data anomalies:

- **Dirty Reads**: Transaction A reads data that Transaction B has *modified* but not yet *committed*. If B rolls back, A has read "dirty" data that never existed.[104]
- **Non-Repeatable Reads**: Transaction A reads a row. Transaction B *updates* or *deletes* that row and commits. Transaction A reads the *same row* again and finds it has changed or is gone.[104]
- **Phantom Reads**: Transaction A reads a *range* of rows (e.g., WHERE age > 30). Transaction B *inserts* a new row that fits that range and commits. Transaction A re-runs its query and finds a new "phantom" row has appeared.[104]

10.3 Transaction Isolation Levels

These levels define the trade-off between performance and consistency. Higher levels prevent more anomalies but are "slower" as they require more locking.[101]

Isolation Level	Prevents Dirty Read?	Prevents Non-Repeatable Read?	Prevents Phantom Read?
READ UNCOMMITTED	No	No	No [104]
READ COMMITTED	**Yes**	No	No [104]
REPEATABLE READ	**Yes**	**Yes**	No [104]
SERIALIZABLE	**Yes**	**Yes**	**Yes** [104]

Module 11: Applied SQL for Data Science and AI

Objective: This capstone module applies all previous concepts (CTEs, Joins, Window Functions) to the *exact* types of analytical problems encountered in data science interviews.

11.1 Pattern 1: Funnel Analysis

- **Use Case**: Tracking user conversion through a multi-step process. (e.g., view_homepage $\rightarrow$ view_product $\rightarrow$ add_to_cart $\rightarrow$ purchase).[109]
- **SQL Pattern**: Use a series of sequential CTEs or LEFT JOINs.
    1. CTE_Step1 finds all users who completed step 1.
    2. CTE_Step2 LEFT JOINs CTE_Step1 to the events table for step 2 (often with a time constraint, e.g., event_time > step1_time).
    3. Repeat for all steps.
    4. The final SELECT uses COUNT(DISTINCT step1.user_id), COUNT(DISTINCT step2.user_id), etc., to build the funnel counts.[109]

11.2 Pattern 2: Cohort Analysis

- **Use Case**: Tracking user retention or behavior over time, grouped by their "join date" (the cohort).[3]
- **SQL Pattern**:
    1. CTE_Cohorts: Find the "cohort month" (e.g., MIN(join_date)) for every user.
    2. CTE_Activity: Find all subsequent activity months for each user.
    3. Join CTE_Cohorts to CTE_Activity.
    4. Calculate the "month number" (e.g., activity_month - cohort_month).
    5. GROUP BY cohort_month, month_number and COUNT(DISTINCT user_id) to get a retention matrix. [3] provides a specific example of finding a *reactivation* cohort.

11.3 Pattern 3: Time-Series Analysis

- **Use Case**: Calculating trends, moving averages, and period-over-period growth.[6]
- **SQL Pattern**: This pattern relies almost exclusively on LAG and window-based aggregation.
    - **Period-over-Period Growth**: (current_sales - LAG(sales) OVER (ORDER BY date)) / LAG(sales) OVER (ORDER BY date)
    - **7-Day Moving Average**: AVG(sales) OVER (ORDER BY date ROWS BETWEEN 6 PRECEDING AND CURRENT ROW)

11.4 Pattern 4: Feature Engineering for ML

- **Use Case**: Using SQL to generate features from raw database tables to feed into a machine learning model.[116]
- **SQL Patterns**:
    - Time Features: Extracting components from a timestamp.
      EXTRACT(dayofweek FROM timestamp) AS weekday, EXTRACT(hour FROM timestamp) AS hour_of_day.116
    - Rolling Aggregates: Creating features like "avg_30_day_spend."
      AVG(txn_amount) OVER (PARTITION BY user_id ORDER BY timestamp ROWS BETWEEN 30 PRECEDING AND CURRENT ROW).116
    - Binarization: Creating dummy variables.
      CASE WHEN category = 'Electronics' THEN 1 ELSE 0 END AS is_electronics.123
    - Categorical Encoding:
      COUNT(transaction_id) OVER (PARTITION BY user_id, merchant_category) (e.g., "number of transactions in this category").116

Module 12: Acing the SQL Interview

Objective: To provide a meta-framework for approaching the SQL interview itself.

12.1 Common Interview Question Patterns

The following are canonical questions. A candidate must have immediate, polished answers for them.

- **"Find the Nth highest salary."**
    - **Answer**: Use DENSE_RANK() OVER (ORDER BY salary DESC) AS rnk in a CTE, then filter WHERE rnk = N in the outer query.
- **"Find all duplicate emails in a table."**
    - **Answer**: Use GROUP BY email HAVING COUNT(*) > 1;.
- **"Find users who bought product A but not product B."**
    - Answer: Use SELECT user_id FROM purchases WHERE product = 'A'
      EXCEPT
      SELECT user_id FROM purchases WHERE product = 'B'.
    - (Alternatively: LEFT JOIN... WHERE B.key IS NULL).
- **"What is the difference between WHERE and HAVING?"**
    - **Answer**: (See Module 3.3) WHERE filters rows *before* aggregation, HAVING filters groups *after* aggregation.
- **"What is the difference between UNION and UNION ALL?"**
    - **Answer**: (See Module 5.3) UNION removes duplicates, UNION ALL keeps them. UNION ALL is much faster and should be the default.
- **"What is the difference between DELETE, TRUNCATE, and DROP?"**

    - **Answer**: (See Module 1.2) DROP deletes the table structure. TRUNCATE deletes all data (fast, DDL). DELETE deletes rows (slow, DML) and can be filtered.
- **"What is a window function?"**
    - **Answer**: (See Module 7) It's a function that performs a calculation across a set of rows (a "window") while *preserving* the individual rows, unlike a GROUP BY which collapses them.

12.2 A Framework for SQL Case Studies

A SQL case study is a test of analytical process and communication, not just coding.[124] The worst response is to immediately start writing code. The best response is to be systematic.
A 5-Step Framework for any case study [126]:

1. **Clarify the Objective**: Ask questions *first*. "What is the business goal?" "How do you define 'active user'?" "What is the time frame?" This shows you are thinking about the *problem* before the *query*.[126]
2. **Plan Your Approach**: Verbally state your plan. "To find retention, first, I'll use a CTE to find the join month for all users. Second, I'll use another CTE to find all their monthly activity. Finally, I will join these and group by the cohort month and activity month to build a retention matrix."
3. **Define Metrics and Data**: State the specific metric and data needed. "The metric will be 7-day user retention. I will need the users table for their join date and the logins table for their activity timestamps".[126]
4. **Analyze (Write the Code)**: Write the query. The CTEs should *match the plan* from Step 2. (e.g., WITH user_cohorts AS (...), user_activity AS (...),...). Talk through the query as it is written.
5. **Communicate Impact**: Do not just give the final table or number. Synthesize the result. "The query shows that the 7-day retention for the July cohort was 15%. This is down from 18% in June. A good next step would be to investigate if there was a change in the sign-up flow or a problematic feature launch in early July".