SQL FOR QA AND SDET INTERVIEWS

The SQL Starter for Testers

Every query runs. Every result on these pages came out of the database that ships inside this file.

FREE STARTER EDITION · SHARE IT FREELY

24 questions · 14 sections · one runnable practice database

MySQL · PostgreSQL · SQL Server · SQLite

by Sreenidhi Rajakrishnan
QA Automation Architect · QA and SDET Educator

Edition 1.0 · 2026

Index

Every section, the questions it holds, and the page it starts on. Page numbers are printed in the bottom right corner.

HOW TO USE THIS GUIDE

Read it with the database open

This guide has one rule behind it. Every query on every page was run against the practice database that ships with this file, and the result printed underneath it is the result that came back. Nothing is typed from memory. If a page says a query returns two rows, it returns two rows, and you can prove it in a minute.

Start by loading the database

The appendix at the back is the whole script. Save it as `sql_seed.sql`, load it once, and every page becomes something you can try rather than something you have to trust. Section 1 shows you how in about a minute, with no server to install.

How a question is laid out

The question, then the answer written the way you would say it out loud in the room. Then the query on a light panel, and its real result on a dark one. Some questions add two boxes at the end. **Where the databases differ** is the dialect note, because Indian interviews ask across MySQL, PostgreSQL and SQL Server and the panel often uses a different one from you. **In the interview** is what to actually say, which is usually the part that decides the outcome.

The level badges

Junior is asked of everyone, whatever your experience. **Mid** is the working band, roughly two to five years, and it is where most SQL rounds are decided. **Senior** questions are the ones where the interviewer is listening for a trade off rather than a syntax. Aim to be fluent through Mid and comfortable talking through Senior at a whiteboard.

> ONE HABIT THAT PAYS OFF IN EVERY SQL ROUND
>
> Say what you are doing while you type. "I will start with the orders table, join the customers so I can show the name, then group it." A candidate who narrates gets helped when they take a wrong turn. A candidate who types in silence gets marked on the final query alone, which is a much harder way to pass.

What this file is

This is the free starter, and it is meant to be useful on its own rather than to tease you. It carries a real question from every one of the fourteen sections of the complete guide, the same practice database in the appendix, and the same rule that nothing is printed unless the database produced it. The complete guide holds 140 questions and goes considerably deeper. The last page says exactly what is in it, so you can decide for yourself.

SECTION 1

Your Practice Database

Everything in this guide runs. Load the seed script once and every query on every page will give you the same result printed beside it. This first section is a short tour of the eleven tables you will be working with, so the rest of the guide reads like something you can try rather than something you have to trust.

Q 1 LEVEL · Junior

How do I load the practice database and start running these queries?

The guide ships with one file, sql_seed.sql. It creates the tables and inserts the data. You do not need MySQL, PostgreSQL or SQL Server installed to follow along, because SQLite comes built into Python, and every query in this guide is written to run on all four.

Save sql_seed.sql next to a new folder, open a terminal there, and run the Python below. It creates practice.db and prints the table list. After that you can run any query in this guide against it.

PYTHON

```
import sqlite3

con = sqlite3.connect("practice.db")
con.executescript(open("sql_seed.sql").read())
con.commit()

for row in con.execute("SELECT name FROM sqlite_master "
                       "WHERE type='table' ORDER BY name"):
    print(row[0])
```

QUERY

```
SELECT name FROM sqlite_master WHERE type = 'table' ORDER BY name;
```

RESULT, FROM THE PRACTICE DATABASE

```
name
----------------
customers
customers_staging
departments
employees
login_audit
order_items
orders
payments
products
test_results
test_runs
```

WHERE THE DATABASES DIFFER

The sqlite_master query is the SQLite way to list tables. MySQL uses SHOW TABLES. PostgreSQL uses \dt in psql. SQL Server uses SELECT name FROM sys.tables. All four also support the standard SELECT table_name FROM information_schema.tables, except SQLite.

IN THE INTERVIEW

If an interviewer gives you a live database and no documentation, listing the tables is the correct first move. Say it out loud: "first I would look at what tables exist and how they join".

Q 2 LEVEL · Junior

What is in each table, and how do the tables connect?

Eleven tables, in three groups.

The interview classics: departments and employees. Almost every SQL interview in India uses a version of these two, so the puzzles later in the guide use them too.

The application under test: customers, products, orders, order_items and payments. This is the shape of nearly every e-commerce or booking system you will be asked to test.

The QA angle: test_runs, test_results and login_audit, plus customers_staging for practising a data migration check. No generic SQL course gives you these, and they are exactly what your own job looks like.

- employees.dept_id points at departments.dept_id, and employees.manager_id points at another row in employees.
- orders.customer_id points at customers.customer_id.
- order_items.order_id points at orders.order_id, and order_items.product_id points at products.product_id.
- payments.order_id points at orders.order_id.
- test_results.run_id points at test_runs.run_id.
- login_audit.customer_id points at customers.customer_id.

QUERY

```
SELECT 'customers' AS table_name, COUNT(*) AS row_count FROM customers
UNION ALL SELECT 'products',          COUNT(*) FROM products
UNION ALL SELECT 'orders',            COUNT(*) FROM orders
UNION ALL SELECT 'order_items',       COUNT(*) FROM order_items
UNION ALL SELECT 'payments',          COUNT(*) FROM payments
UNION ALL SELECT 'employees',         COUNT(*) FROM employees
UNION ALL SELECT 'departments',       COUNT(*) FROM departments
UNION ALL SELECT 'test_runs',         COUNT(*) FROM test_runs
UNION ALL SELECT 'test_results',      COUNT(*) FROM test_results
UNION ALL SELECT 'login_audit',       COUNT(*) FROM login_audit
UNION ALL SELECT 'customers_staging', COUNT(*) FROM customers_staging;
```

RESULT, FROM THE PRACTICE DATABASE

```
table_name         row_count
-----------------  ---------
customers                  8
products                   7
orders                    10
order_items               11
payments                   8
employees                 12
departments                5
test_runs                  5
test_results              25
login_audit               10
customers_staging          9
```

SECTION 2

Reading Data With Confidence

SELECT, WHERE and ORDER BY are where every interview starts, and where a surprising number of candidates lose easy marks. The trap is not the syntax, it is the order the database evaluates things in, and knowing that order is what lets you explain your answer instead of just typing it.

Q 3 LEVEL · Junior

What order does a database actually run the parts of a SELECT in?

You write SELECT first, but the database runs it almost last. The real order is FROM, then JOIN, then WHERE, then GROUP BY, then HAVING, then SELECT, then DISTINCT, then ORDER BY, then LIMIT.

This single fact explains a lot of everyday errors. You cannot use a column alias you created in SELECT inside your WHERE clause, because WHERE ran before SELECT existed. You can use that alias in ORDER BY, because ORDER BY runs after SELECT. And a filter that belongs in WHERE should not be put in HAVING, because HAVING runs after grouping and is far slower.

- FROM and JOIN decide which rows exist at all.
- WHERE filters individual rows, before any grouping.
- GROUP BY collapses rows into groups, and HAVING filters those groups.
- SELECT builds the output columns, which is why aliases are born here.
- ORDER BY and LIMIT run last, on the finished result.

QUERY

```
SELECT city, COUNT(*) AS staff
FROM employees
WHERE salary > 70000
GROUP BY city
HAVING COUNT(*) > 1
ORDER BY staff DESC, city;
```

RESULT, FROM THE PRACTICE DATABASE

```
city       staff
---------  -----
Hyderabad      4
Bengaluru      3
Pune           2
```

IN THE INTERVIEW

If you are asked "why can I not use my alias in WHERE", the answer is one sentence: WHERE runs before SELECT, so the alias does not exist yet.

Q 4 LEVEL · Junior

What is the difference between WHERE and HAVING?

WHERE filters rows before they are grouped. HAVING filters groups after they are formed, which is why HAVING is the only place an aggregate like COUNT or SUM can appear in a filter.

The practical rule: if your condition is about one row, use WHERE. If it is about a whole group, use HAVING. Putting a plain row condition in HAVING still works on most databases, but it makes the query slower,

because you grouped rows you were going to throw away anyway.

QUERY

```
SELECT d.dept_name,
       COUNT(*)    AS employees,
       AVG(e.salary) AS avg_salary
FROM employees e
JOIN departments d ON d.dept_id = e.dept_id
WHERE e.hire_date < '2023-01-01'
GROUP BY d.dept_name
HAVING COUNT(*) >= 2
ORDER BY employees DESC, d.dept_name;
```

RESULT, FROM THE PRACTICE DATABASE

```
dept_name    employees  avg_salary
-----------  ---------  ----------
Engineering          3   155000.00
QA                   3    83666.67
Support              2    66500.00
```

WHERE THE DATABASES DIFFER

Portable across all four. MySQL is the odd one out in that it lets you reference a SELECT alias inside HAVING; SQL Server and PostgreSQL want the full expression or a subquery.

SECTION 3

NULL And The Three Valued Logic

NULL is the single biggest source of wrong test results in SQL, and the topic most likely to separate a candidate who has used SQL from one who has read about it. NULL does not mean zero and it does not mean an empty string. It means unknown, and once you accept that, every strange behaviour in this section becomes obvious.

Q 5 LEVEL · Junior

Why does WHERE email = NULL return nothing?

Because NULL means unknown, and comparing anything to an unknown gives an unknown answer, not true. WHERE only keeps rows where the condition is definitely true, so a row with an unknown result is dropped. That includes NULL = NULL, which is also unknown rather than true.

The correct test is IS NULL or IS NOT NULL. These are special operators precisely because the equals sign cannot do the job.

QUERY

```
SELECT customer_id, full_name, email
FROM customers
WHERE email IS NULL;
```

RESULT, FROM THE PRACTICE DATABASE

```
customer_id  full_name    email
-----------  -----------  -----
          7  Nisha Verma  NULL
```

WHERE THE DATABASES DIFFER

Identical on all four. SQL Server has a legacy setting, SET ANSI_NULLS OFF, that makes = NULL behave like IS NULL, but it is deprecated and you should never rely on it.

IN THE INTERVIEW

This is the most asked NULL question in Indian QA interviews. Answer it in one line: "NULL means unknown, so equals cannot work, you use IS NULL".

Q 6 LEVEL · Mid

What is three valued logic?

Most languages have true and false. SQL has true, false and unknown. Any comparison involving NULL produces unknown, and unknown then spreads through your AND and OR conditions.

The rules that catch people: unknown AND false is false, but unknown AND true is unknown. unknown OR true is true, but unknown OR false is unknown. And NOT unknown is still unknown, which is why negating a condition does not always give you the rows you expected.

- WHERE keeps a row only when the condition is true, never when it is unknown.
- So a row and its negation can both be excluded, which feels wrong until you know why.
- That is the source of the NOT IN trap in the next question.

QUERY

```
SELECT 'salary > 100000'      AS test,
       COUNT(*)               AS matched
FROM employees WHERE salary > 100000
UNION ALL
SELECT 'NOT (salary > 100000)', COUNT(*)
FROM employees WHERE NOT (salary > 100000)
UNION ALL
SELECT 'total rows',            COUNT(*) FROM employees;
```

RESULT, FROM THE PRACTICE DATABASE

```
test                  matched
--------------------  -------
salary > 100000             4
NOT (salary > 100000)       8
total rows                 12
```

Q 7 LEVEL · Mid

Why did my NOT IN query return zero rows?

Because the list you gave NOT IN contained a NULL. NOT IN is expanded into a chain of not equals joined by AND, and the moment one of those comparisons is against NULL the whole chain becomes unknown, so no row qualifies. It does not error. It just returns nothing, which in a test reads as "no mismatches found" and passes.

This is the most dangerous NULL behaviour for a tester, because the failure mode is a false pass. Use NOT EXISTS instead, which is not affected, or filter the NULLs out of the inner query first.

QUERY

```
SELECT 'NOT IN with a NULL in the list' AS approach,
       COUNT(*) AS rows_returned
FROM departments d
WHERE d.dept_id NOT IN (SELECT e.dept_id FROM employees e)
UNION ALL
SELECT 'NOT EXISTS', COUNT(*)
FROM departments d
WHERE NOT EXISTS (SELECT 1 FROM employees e WHERE e.dept_id = d.dept_id)
UNION ALL
SELECT 'NOT IN with NULLs removed', COUNT(*)
FROM departments d
WHERE d.dept_id NOT IN (SELECT e.dept_id FROM employees e WHERE e.dept_id IS NOT NULL);
```

RESULT, FROM THE PRACTICE DATABASE

```
approach                       rows_returned
-----------------------------  -------------
NOT IN with a NULL in the list             0
NOT EXISTS                                 1
NOT IN with NULLs removed                  1
```

WHERE THE DATABASES DIFFER

The same on all four databases. This is standard SQL behaviour, not a quirk of any one engine.

IN THE INTERVIEW

If you can explain this one clearly you will stand out. The Data department really does have no employees, but NOT IN cannot tell you that, because Ganesh Iyer has a NULL department.

SECTION 4

Joins

Joins are the heart of every SQL interview, and the heart of every real verification you will ever write. A test does not usually live in one table: a customer places an order, the order has items, the items have products, and the order should have a payment. Checking that chain end to end is a join, and finding the link that is missing is a join too.

Q 8 LEVEL · Junior

What is the difference between an INNER JOIN and a LEFT JOIN?

An INNER JOIN keeps only the rows that matched on both sides. A LEFT JOIN keeps every row from the left table, and fills the right hand columns with NULL where there was no match.

For a tester the difference is not academic. INNER JOIN answers "show me the orders that have a payment". LEFT JOIN answers "show me every order, and tell me which ones have no payment". The second question is the one that finds bugs, because a missing row cannot show up in an inner join by definition.

QUERY

```
SELECT o.order_id,
       o.status,
       o.amount,
       p.payment_id,
       p.status AS payment_status
FROM orders o
LEFT JOIN payments p ON p.order_id = o.order_id
ORDER BY o.order_id, p.payment_id;
```

RESULT, FROM THE PRACTICE DATABASE

```
order_id  status     amount  payment_id  payment_status
--------  ---------  ------  ----------  --------------
    1001  delivered    5398           1  success
    1002  delivered   18999           2  success
    1003  cancelled     899        NULL  NULL
    1004  delivered    1299           3  success
    1005  shipped      3299           4  success
    1006  delivered     899        NULL  NULL
    1007  pending      4999        NULL  NULL
    1008  delivered    2499           5  failed
    1008  delivered    2499           6  success
    1009  returned    18999           7  success
    1010  delivered    1798           8  success
```

IN THE INTERVIEW

Notice order 1006: delivered, and no payment row at all. That is a real bug, found in one query. Notice order 1008 too: it appears twice, because the customer paid twice.

Q 9 LEVEL · Senior

Why does putting a condition in WHERE break my LEFT JOIN?

Because of the order things run in. The join happens first and fills the unmatched right hand columns with NULL. Then WHERE runs, and a condition like p.status = 'success' is unknown for those NULL rows, so they are thrown away. You have just turned your LEFT JOIN into an INNER JOIN without noticing.

The rule: a condition about the right hand table belongs in the ON clause. A condition about the left hand table belongs in WHERE. The one exception is IS NULL on the right hand key, which is the anti-join and must be in WHERE.

QUERY

```
SELECT 'condition in ON, stays a LEFT JOIN'   AS variant, COUNT(*) AS rows_returned
FROM orders o
LEFT JOIN payments p ON p.order_id = o.order_id AND p.status = 'success'
UNION ALL
SELECT 'condition in WHERE, becomes an INNER JOIN', COUNT(*)
FROM orders o
LEFT JOIN payments p ON p.order_id = o.order_id
WHERE p.status = 'success';
```

RESULT, FROM THE PRACTICE DATABASE

```
variant                                 rows_returned
--------------------------------------  -------------
condition in ON, stays a LEFT JOIN                 10
condition in WHERE, becomes an INNER JOIN           7
```

IN THE INTERVIEW

This is a senior level question and a very common review comment. If you can explain it in the interview you are ahead of most candidates.

Q 10 LEVEL · Mid

What is a self join, and what is it used for?

A self join is a table joined to itself, using two different aliases so the database can tell the two copies apart. You need it whenever a table points at its own rows, which is far more common than it sounds: an employee has a manager, a comment has a parent comment, a category has a parent category.

The alias is not optional decoration. Without it the database cannot tell which copy of the table each column belongs to.

QUERY

```
SELECT e.emp_id,
       e.emp_name  AS employee,
       e.salary    AS employee_salary,
       m.emp_name  AS manager,
       m.salary    AS manager_salary
FROM employees e
LEFT JOIN employees m ON m.emp_id = e.manager_id
ORDER BY e.emp_id;
```

RESULT, FROM THE PRACTICE DATABASE

```
emp_id  employee       employee_salary  manager        manager_salary
------  -------------  ---------------  -------------  --------------
     1  Asha Nair               210000  NULL                     NULL
     2  Rahul Menon              95000  Asha Nair              210000
     3  Divya Iyer               78000  Rahul Menon             95000
     4  Karthik Rao             120000  Asha Nair              210000
     5  Meera Pillai            135000  Karthik Rao            120000
     6  Sandeep Kumar            88000  Asha Nair              210000
     7  Neha Sharma              78000  Rahul Menon             95000
     8  Vikram Reddy             62000  Sandeep Kumar           88000
     9  Priya Das                71000  Sandeep Kumar           88000
    10  Arjun Nair               99000  Sandeep Kumar           88000
    11  Fatima Sheikh           120000  Karthik Rao            120000
    12  Ganesh Iyer              54000  Asha Nair              210000
```

IN THE INTERVIEW

Use LEFT JOIN, not INNER JOIN, for the manager lookup. Asha has no manager, and an inner join would silently drop the head of the company from your report.

SECTION 5

Grouping And Aggregation

GROUP BY is the moment SQL stops listing rows and starts answering questions. How many orders per status. How many tests passed in each run. Which customer spends the most. It is also where the interview gets its first real signal about you, because the mistakes here are predictable and an interviewer knows exactly which ones to look for.

Q 11 LEVEL · Junior

What does GROUP BY actually do?

It collapses many rows into one row per group. You choose the column that defines the group, and every other column in your SELECT has to be an aggregate, because a group of five rows cannot show five different prices in one cell.

Read the query below as a sentence: for each category, count the products and find the cheapest and the dearest. That phrasing, "for each X", is the tell. Whenever a requirement contains it, you need a GROUP BY.

- COUNT(*) counts the rows in the group.
- SUM, AVG, MIN and MAX work on one column inside the group.
- One row comes out per distinct value of the grouping column.
- ORDER BY runs after the grouping, so it can sort by an aggregate.

QUERY

```
SELECT category,
       COUNT(*)   AS products,
       MIN(price) AS cheapest,
       MAX(price) AS dearest
FROM products
GROUP BY category
ORDER BY category;
```

RESULT, FROM THE PRACTICE DATABASE

```
category     products  cheapest  dearest
-----------  --------  --------  -------
Accessories         4       899     4499
Displays            1     18999    18999
Peripherals         2      3299     7999
```

IN THE INTERVIEW

If you can rephrase the requirement as "for each … give me …" then say that out loud in the interview before you type. It shows you are choosing GROUP BY on purpose, not by reflex.

Q 12 LEVEL · Mid

How do I count passes and failures in the same query, without running it twice?

Conditional aggregation. Put a CASE inside the SUM, so each row contributes 1 to the counter you want and 0 to the others. One pass over the data gives you every column of the report.

This is the single most useful pattern in this guide for a QA engineer. It turns a test_results table into the run summary that a reporting dashboard would show you, and you can run it the moment a pipeline finishes instead of waiting for the report to render.

QUERY

```
SELECT r.run_id,
       r.suite_name,
       COUNT(*) AS total,
       SUM(CASE WHEN t.status = 'passed'  THEN 1 ELSE 0 END) AS passed,
       SUM(CASE WHEN t.status = 'failed'  THEN 1 ELSE 0 END) AS failed,
       SUM(CASE WHEN t.status = 'skipped' THEN 1 ELSE 0 END) AS skipped
FROM test_runs r
JOIN test_results t ON t.run_id = r.run_id
GROUP BY r.run_id, r.suite_name
ORDER BY r.run_id;
```

RESULT, FROM THE PRACTICE DATABASE

```
run_id  suite_name  total  passed  failed  skipped
------  ----------  -----  ------  ------  -------
     1  regression      6       4       2        0
     2  smoke           3       3       0        0
     3  regression      7       4       2        1
     4  regression      7       6       1        0
     5  smoke           2       1       0        0
```

WHERE THE DATABASES DIFFER

CASE works on all four databases and is the answer to give. PostgreSQL also has the shorter COUNT(*) FILTER (WHERE status = 'passed'), and MySQL lets you write SUM(status = 'passed') because it treats true as 1. Both are neat, neither is portable.

IN THE INTERVIEW

Bring this one to the interview as your own example: "I use conditional aggregation to get a pass and fail breakdown per suite straight out of the results table." It answers a SQL question and shows how you work at the same time.

SECTION 6

Subqueries And EXISTS

A subquery is just a query inside a query. Once you can see where the brackets go, the interview questions become easy, because almost all of them are the same three shapes reused: a value, a list, and a check for existence. The part that separates candidates is knowing when a join would have been the better answer.

Q 13 LEVEL · Mid

How do I find rows with no match, using NOT EXISTS?

NOT EXISTS reads as a sentence: give me the orders for which no payment row exists. It is the safest of the three ways to ask this question, because unlike NOT IN it is not affected by NULLs.

The other two ways are a LEFT JOIN with IS NULL on the right side, and NOT IN. Know all three, lead with NOT EXISTS, and mention that the LEFT JOIN version is equally correct and sometimes faster depending on the engine.

QUERY

```
SELECT o.order_id, o.customer_id, o.status, o.amount
FROM orders o
WHERE NOT EXISTS (SELECT 1 FROM payments p
                  WHERE p.order_id = o.order_id
                    AND p.status = 'success')
ORDER BY o.order_id;
```

RESULT, FROM THE PRACTICE DATABASE

```
order_id  customer_id  status     amount
--------  -----------  ---------  ------
    1003            3  cancelled     899
    1006            2  delivered     899
    1007            6  pending      4999
```

IN THE INTERVIEW

This exact query is a real bug report. Order 1006 is marked delivered and has no successful payment against it. Goods shipped, money not taken. If you can find that in an interview and explain why it matters, you have shown testing skill, not just SQL.

SECTION 7

Common Table Expressions

A CTE is a named result you define at the top of a query and use underneath. Nothing you can do with a CTE is impossible with a subquery, but a query built out of named steps is a query you can explain in an interview, hand to a reviewer, and debug one piece at a time. That is why they get asked about.

Q 14 LEVEL · Mid

What is a CTE and why would I use one instead of a subquery?

A common table expression is a named query defined with WITH, used by the query that follows it. It exists only for that one statement.

The reason to use one is readability, and readability is not a small thing in an interview. A nested subquery makes the reader start in the middle and work outwards. A CTE lets them read top to bottom, in the order you thought of it. You can also test each step on its own by selecting from it, which is exactly how you debug a query that is giving the wrong number.

QUERY

```
WITH order_totals AS (
    SELECT order_id, SUM(qty * unit_price) AS line_total
    FROM order_items
    GROUP BY order_id
)
SELECT o.order_id, o.amount AS order_amount, t.line_total
FROM orders o
JOIN order_totals t ON t.order_id = o.order_id
WHERE o.amount <> t.line_total
ORDER BY o.order_id;
```

RESULT, FROM THE PRACTICE DATABASE

```
order_id  order_amount  line_total
--------  ------------  ----------
    1007          4999        4499
```

WHERE THE DATABASES DIFFER

Supported by PostgreSQL, SQL Server 2005 and later, SQLite 3.8.3 and later, and MySQL 8.0 and later. MySQL 5.7 has no CTE at all, so if the company is on 5.7 you write it as a derived table instead. Worth asking which version they run.

IN THE INTERVIEW

This query finds the order whose stored total does not match its own line items. It is one of the six faults planted in the practice data, and it is the single most common real bug in an e-commerce system: the header and the lines disagree.

SECTION 8

Window Functions

Window functions are the line between a candidate who learned SQL for an interview and one who has used it. They let you keep every row and still see a calculation across a group of rows: a rank, a running total, the previous row's value. Almost every interview puzzle in the next section is two lines long once you know them.

Q 15 LEVEL · Mid

What is the difference between ROW_NUMBER, RANK and DENSE_RANK?

They only differ when there is a tie, and the tie is the entire point of the question.

ROW_NUMBER always gives 1, 2, 3, even for equal values, so two identical salaries get different numbers and which one gets 1 is arbitrary unless you add a tie breaker. RANK gives equal values the same number and then skips: 1, 2, 2, 4. DENSE_RANK gives equal values the same number and does not skip: 1, 2, 2, 3.

The practice data has two employees on 78000 and two on 120000, so the difference is visible in the output rather than something you have to take on trust.

QUERY

```
SELECT emp_name,
       salary,
       ROW_NUMBER() OVER (ORDER BY salary DESC) AS row_num,
       RANK()       OVER (ORDER BY salary DESC) AS rank_val,
       DENSE_RANK() OVER (ORDER BY salary DESC) AS dense_val
FROM employees
ORDER BY salary DESC, emp_name;
```

RESULT, FROM THE PRACTICE DATABASE

```
emp_name       salary  row_num  rank_val  dense_val
-------------  ------  -------  --------  ---------
Asha Nair      210000        1         1          1
Meera Pillai   135000        2         2          2
Fatima Sheikh  120000        4         3          3
Karthik Rao    120000        3         3          3
Arjun Nair      99000        5         5          4
Rahul Menon     95000        6         6          5
Sandeep Kumar   88000        7         7          6
Divya Iyer      78000        8         8          7
Neha Sharma     78000        9         8          7
Priya Das       71000       10        10          8
Vikram Reddy    62000       11        11          9
Ganesh Iyer     54000       12        12         10
```

IN THE INTERVIEW

For "the second highest salary" the correct choice is DENSE_RANK, because with RANK a tie at the top means nobody is ranked 2 at all and your query returns nothing. That is the trap in the classic puzzle, and the next section works through it.

SECTION 9

The Classic Interview Puzzles

There is a short list of SQL problems that Indian interview panels have been asking for fifteen years, and they still ask them. Second highest salary. Duplicates. Employees who earn more than their manager. They are not asked because the answers are useful at work. They are asked because each one has an edge case, and the edge case is what is being marked.

Q 16 LEVEL · Mid

How do I find the second highest salary?

The most asked SQL interview question in India, and the one where candidates most often give an answer that works on the sample data and breaks on real data.

The simple version below needs no window functions and runs on every database including MySQL 5.7: find the highest salary that is below the maximum. It handles ties correctly, because if two people share the top salary they are both excluded and you still get the genuine second value.

Be ready for the follow up, which is always the same: what if there is no second highest? This version returns NULL rather than an error, which is usually what you want.

QUERY

```
SELECT MAX(salary) AS second_highest
FROM employees
WHERE salary < (SELECT MAX(salary) FROM employees);
```

RESULT, FROM THE PRACTICE DATABASE

```
second_highest
--------------
        135000
```

WHERE THE DATABASES DIFFER

The window version, on MySQL 8, PostgreSQL, SQL Server 2012 and SQLite 3.25 and later: SELECT DISTINCT salary FROM (SELECT salary, DENSE_RANK() OVER (ORDER BY salary DESC) AS r FROM employees) t WHERE r = 2. Use DENSE_RANK, not RANK: with two people tied at the top, RANK skips 2 entirely and the query returns nothing.

IN THE INTERVIEW

The answer that gets marked down is LIMIT 1 OFFSET 1. It returns the second row, not the second distinct salary, so with a tie at the top it hands back the top salary again. Say that out loud even if the interviewer does not ask.

Q 17 LEVEL · Mid

How do I find duplicate rows?

Group by the columns that define a duplicate, then keep the groups with more than one row. The hard part is not the SQL, it is deciding what counts as a duplicate, and that is a question worth asking out loud in the interview.

The same person entered twice with two different ids is a duplicate even though the rows are not identical, because the id was generated by the system. So the grouping columns are the business columns: name and email, not the primary key.

QUERY

```
SELECT full_name,
       email,
       COUNT(*)          AS copies,
       MIN(customer_id)  AS keep_this_id,
       MAX(customer_id)  AS extra_id
FROM customers
GROUP BY full_name, email
HAVING COUNT(*) > 1
ORDER BY full_name;
```

RESULT, FROM THE PRACTICE DATABASE

```
full_name       email               copies  keep_this_id  extra_id
--------------  ------------------  ------  ------------  --------
Aarti Deshpande  aarti.d@example.com       2             1         8
```

IN THE INTERVIEW

Selecting MIN and MAX of the id in the same query is what turns a finding into a fix. You now know which row to keep and which to remove, without a second query.

SECTION 10

Finding Bad Data

This is the section that no generic SQL course gives a tester. Every query here answers a question you will genuinely be asked at work: did the data survive the migration, does the total match the parts, is anything pointing at a row that no longer exists. The practice database has six faults planted in it, and by the end of this section you will have found all six.

Q 18 LEVEL · Mid

How do I check that an order total matches its own line items?

Add up the lines, join the sum back to the header, and compare. This is the most common real defect in any system that stores a total: the header and the detail rows drift apart, usually after a partial cancellation or a discount that was applied in only one place.

The report below shows every order rather than only the broken one. That is deliberate. A check that returns nothing tells you nothing about whether it ran, while a report with one non-zero line tells you the check worked and what it found.

QUERY

```
WITH lines AS (
    SELECT order_id, SUM(qty * unit_price) AS line_total
    FROM order_items
    GROUP BY order_id
)
SELECT o.order_id,
       o.amount                              AS header_amount,
       COALESCE(l.line_total, 0)             AS line_items_total,
       o.amount - COALESCE(l.line_total, 0)  AS difference
FROM orders o
LEFT JOIN lines l ON l.order_id = o.order_id
ORDER BY o.order_id;
```

RESULT, FROM THE PRACTICE DATABASE

```
order_id  header_amount  line_items_total  difference
--------  -------------  ----------------  ----------
    1001           5398              5398           0
    1002          18999             18999           0
    1003            899               899           0
    1004           1299              1299           0
    1005           3299              3299           0
    1006            899               899           0
    1007           4999              4499         500
    1008           2499              2499           0
    1009          18999             18999           0
    1010           1798              1798           0
```

IN THE INTERVIEW

Order 1007 is the planted fault: the header says 4999 and the one line item comes to 4499. In a real system that is 500 rupees that either the customer was overcharged or the company never collected. That is how you write the bug report.

SECTION 11

Dates And Times

Dates cause more wrong answers than any other data type, and almost always for the same reason: the column holds a time as well as a date, and the query forgot. This section fixes that, then covers the functions you need for run durations, monthly reports and the time zone question that comes up in every team that has a server abroad.

Q 19 LEVEL · Mid

Why does my BETWEEN filter miss the last day?

Because the column holds a time as well as a date. BETWEEN '2026-08-11' AND '2026-08-12' means from midnight on the eleventh to midnight on the twelfth, so anything that happened at nine in the morning on the twelfth is after your upper bound and is dropped.

The output below proves it. Four runs exist across those two days and only two come back. Both runs on the twelfth are missing, and nothing in the result tells you they are missing, which is what makes this bug so expensive.

QUERY

```
SELECT run_id, suite_name, environment, started_at
FROM test_runs
WHERE started_at BETWEEN '2026-08-11' AND '2026-08-12'
ORDER BY run_id;
```

RESULT, FROM THE PRACTICE DATABASE

```
run_id  suite_name  environment  started_at
------  ----------  -----------  -------------------
     2  smoke       staging      2026-08-11 09:15:00
     3  regression  staging      2026-08-11 21:00:00
```

IN THE INTERVIEW

This is the single most common date bug in reporting, and every tester should have it in their head. Any report with a date range filter is worth checking for it on day one, and it is a great answer when an interviewer asks what bugs you have found.

SECTION 12

Changing Data Safely

Reading data cannot break anything. Writing it can, and a tester writes far more than people expect: setting up a scenario, resetting state between runs, correcting a record so a workflow can continue. This section is about doing that without becoming the story of the week.

Q 20 LEVEL · Mid

What is a transaction, and how do I undo a mistake?

A transaction groups several statements so they all take effect or none of them do. BEGIN starts it, COMMIT makes it permanent, ROLLBACK throws it away.

The output below is the whole idea in one query. Inside the transaction the payments table is emptied and the count really does read zero. After the rollback, all eight rows are back. Nothing was lost, because nothing was ever committed.

STATEMENT

```
BEGIN;
DELETE FROM payments;
SELECT COUNT(*) AS rows_seen_inside_the_transaction FROM payments;
ROLLBACK;

-- then look at what changed:
SELECT COUNT(*) AS rows_after_the_rollback FROM payments;
```

RESULT, FROM THE PRACTICE DATABASE

```
(BEGIN OK)

(8 rows affected)

rows_seen_inside_the_transaction
--------------------------------
                               0

(ROLLBACK OK)

rows_after_the_rollback
-----------------------
                      8
```

WHERE THE DATABASES DIFFER

MySQL with InnoDB, PostgreSQL, SQL Server and SQLite all support this. MySQL with the older MyISAM engine does not, and a rollback there does nothing at all. SQL Server writes BEGIN TRANSACTION. Data definition statements such as CREATE and ALTER cannot be rolled back on MySQL or Oracle, but can be on PostgreSQL, SQL Server and SQLite.

IN THE INTERVIEW

Make this your habit on any shared database: type BEGIN before you type UPDATE. Run the change, look at the result, and only then decide between COMMIT and ROLLBACK. It costs six characters and it has saved a great many careers.

SECTION 13

Indexes And Why A Query Is Slow

Sooner or later an interviewer asks why a query is slow, and the answer they are waiting for is almost always about indexes. This section shows the database making that decision in front of you. Every plan printed here is the real plan SQLite produced for that exact query, so you can watch a full table read turn into a direct lookup the moment an index exists.

Q 21 LEVEL · Junior

What is an index, and why does it make a query faster?

An index is a second, sorted copy of one or more columns, kept beside the table, with a pointer back to the full row. The book analogy is the one to use in an interview: to find every mention of a word you can read all four hundred pages, or you can turn to the index at the back, which is sorted, and jump straight to the three pages that matter.

Without an index the database has no choice but to look at every row and check it. That is called a full table scan. On ten rows it is instant. On ten million it is the reason your page is still loading.

The query below has no index to help it, and the plan says so. Read only the last column. The word SCAN means every row was examined.

QUERY

```
EXPLAIN QUERY PLAN
SELECT order_id, order_date, amount
FROM orders
WHERE customer_id = 1;
```

RESULT, FROM THE PRACTICE DATABASE

```
id  parent  notused  detail
--  ------  -------  -----------
 2       0      216  SCAN orders
```

WHERE THE DATABASES DIFFER

EXPLAIN QUERY PLAN is the SQLite spelling. MySQL and SQL Server both use EXPLAIN, and SQL Server also offers SET SHOWPLAN_ALL ON or the graphical plan in SSMS. PostgreSQL uses EXPLAIN for the estimate and EXPLAIN ANALYZE to run the query and report what really happened, which is the more honest of the two.

IN THE INTERVIEW

Say the sentence "an index is a sorted lookup structure that lets the database find rows without reading all of them" and you have answered the question properly. Then add the cost: it has to be kept up to date on every insert, update and delete.

Q 22 LEVEL · Mid

How do I prove that a query actually used an index?

You ask the database for its plan and you read one word. SCAN means it read the whole table. SEARCH means it jumped straight to the rows it needed, and the plan names the index it used to do it.

This is the same query as the previous question, with an index added first. Nothing else changed. The plan flips from SCAN to SEARCH, and it tells you which index earned it.

- SCAN means every row was read. On a large table that is the problem.
- SEARCH means the database went straight to the matching rows.
- The index name in the plan tells you which index was chosen, which matters when a table has several.
- The (customer_id=?) part shows which condition the index was able to satisfy.

QUERY

```
CREATE INDEX ix_orders_customer ON orders (customer_id);

EXPLAIN QUERY PLAN
SELECT order_id, order_date, amount
FROM orders
WHERE customer_id = 1;
```

RESULT, FROM THE PRACTICE DATABASE

```
(CREATE OK)

id  parent  notused  detail
--  ------  -------  ------------------------------------------------------------
 3       0       61  SEARCH orders USING INDEX ix_orders_customer (customer_id=?)
```

WHERE THE DATABASES DIFFER

The words differ by database but the idea does not. MySQL shows type ALL for a full scan and ref or range when an index is used, with the chosen index in the key column. PostgreSQL prints Seq Scan against Index Scan. SQL Server prints Table Scan or Clustered Index Scan against Index Seek. Scan is the slow word in all four.

IN THE INTERVIEW

If you are asked how you would investigate a slow query, the first sentence out of your mouth should be "I would look at the execution plan". Candidates who start guessing at the query instead of asking the database lose the point immediately.

SECTION 14

The Questions You Get Because You Are A Tester

Everything so far applies to anyone who writes SQL. This section is the part that is yours. These are the questions asked specifically because the role is QA or SDET, and the answers that separate a candidate who has validated real data from one who has finished a SQL course. They are also, conveniently, the things you will do most often once you have the job.

Q 23 LEVEL · Mid

My test passed on screen. How do I prove the record was really written?

By going and looking. A user interface can show a success message from a response body that was never persisted, and that class of bug is invisible to a test that only checks the screen. Reading the row back is what turns a shallow test into a real one.

Check more than the one field you were watching. A record written correctly means the right values, in the right row, exactly once, with its related rows and its defaults populated.

Below, a row is inserted the way an application would, and then read back to confirm what actually landed.

- The values you sent, unchanged. Trailing spaces and case changes are real bugs and easy to miss.
- Exactly one row. A retry that wrote twice still looks correct if you only check the first row.
- The columns you did not send: created date, status, a default flag. These are where silent defects live.
- The related rows. An order without its order_items is a half written order.

STATEMENT

```
INSERT INTO customers (customer_id, full_name, email, city, signup_date, status)
VALUES (9, 'Test User 8842', 'qa.test.8842@example.com', 'Pune', '2026-08-12', 'active');

-- then look at what changed:
SELECT customer_id, full_name, email, city, signup_date, status
FROM customers
WHERE email = 'qa.test.8842@example.com';
```

RESULT, FROM THE PRACTICE DATABASE

```
(1 row affected)

customer_id  full_name       email                    city  signup_date  status
-----------  --------------  -----------------------  ----  -----------  ------
          9  Test User 8842  qa.test.8842@example.com  Pune  2026-08-12   active
```

IN THE INTERVIEW

Use a unique value you generated for the test, such as an email with a timestamp in it, and your verification query can never accidentally match somebody else's row. It also makes your test data easy to clean up afterwards.

Q 24 LEVEL · Senior

How do I prove a test is flaky rather than genuinely broken?

A broken test fails every time. A flaky test passes and fails on the same code, in the same environment, with nothing changed between runs. That is a claim you can prove with one query against your run history, and proving it is far more persuasive than saying a test feels unreliable.

Group the results by test and by environment, count the passes and the failures, and keep only the tests that have both. Grouping by environment matters: a test that passes on staging and fails on production every single time is not flaky, it is telling you something true about production.

This practice data has one genuinely flaky test planted in it, and the query finds it.

QUERY

```
SELECT t.test_name, r.environment,
       COUNT(*) AS runs,
       SUM(CASE WHEN t.status = 'passed' THEN 1 ELSE 0 END) AS passed,
       SUM(CASE WHEN t.status = 'failed' THEN 1 ELSE 0 END) AS failed
FROM test_results t
JOIN test_runs r ON r.run_id = t.run_id
GROUP BY t.test_name, r.environment
HAVING passed > 0 AND failed > 0
ORDER BY t.test_name;
```

RESULT, FROM THE PRACTICE DATABASE

```
test_name            environment  runs  passed  failed
-------------------  -----------  ----  ------  ------
checkout_place_order  staging         3       1       2
profile_update_email  staging         2       1       1
```

WHERE THE DATABASES DIFFER

Referring to the alias passed inside HAVING works in SQLite and MySQL. PostgreSQL and SQL Server do not allow an alias there, so repeat the expression: HAVING SUM(CASE WHEN t.status = 'passed' THEN 1 ELSE 0 END) > 0. Writing it out in full is the portable habit.

IN THE INTERVIEW

Take this query to a retrospective and you change the conversation. "These two tests failed and passed on identical code this week" is evidence. It also gives you the ranked list to fix, which is the difference between complaining about flakiness and managing it.

APPENDIX

The practice database

This is the whole script, so the guide is self contained. Copy it into a file called `sql_seed.sql`, load it, and every result in this book is reproducible on your own machine. It loads as written on MySQL, PostgreSQL and SQLite, and the last block of comments in the script says what two lines to change for SQL Server.

The fastest way in is Python, because SQLite is built into it and needs no server. Section 1 has the four lines.

SQL_SEED.SQL · PART 1

```
-- ============================================================================
-- SQL FOR QA AND SDET INTERVIEWS 2026
-- Practice database seed script
-- by Sreenidhi Rajakrishnan  |  sreenidhirajakrishnan.com
-- ============================================================================
--
-- Run this file once, then every query in the guide will work on your machine
-- and give you exactly the output printed in the guide.
--
-- Easiest way (no install needed, Python already has SQLite built in):
--
--     sqlite3 practice.db < sql_seed.sql
--
-- or from Python:
--
--     import sqlite3
--     con = sqlite3.connect("practice.db")
--     con.executescript(open("sql_seed.sql").read())
--
-- MySQL:       mysql -u root -p practice < sql_seed.sql
-- PostgreSQL:  psql -d practice -f sql_seed.sql
-- SQL Server:  see the notes at the bottom of this file
--
-- The script is written in plain, portable SQL so it loads on all four.
-- Dates are stored as text in ISO format (YYYY-MM-DD), which is what SQLite
-- uses and what MySQL and PostgreSQL both accept for a DATE column.
--
-- The data is deliberately imperfect. There are missing values, a duplicate
-- customer, an order whose total does not match its line items, a delivered
-- order with no payment, a flaky test, and a migration that dropped a row.
-- Those are the things a QA engineer is paid to find, so they are the things
-- you should be able to find with SQL.
-- ============================================================================
```

SQL_SEED.SQL · PART 2

```
DROP TABLE IF EXISTS test_results;
DROP TABLE IF EXISTS test_runs;
DROP TABLE IF EXISTS login_audit;
DROP TABLE IF EXISTS payments;
DROP TABLE IF EXISTS order_items;
DROP TABLE IF EXISTS orders;
DROP TABLE IF EXISTS products;
DROP TABLE IF EXISTS customers_staging;
DROP TABLE IF EXISTS customers;
DROP TABLE IF EXISTS employees;
DROP TABLE IF EXISTS departments;

-- ---------------------------------------------------------------------------
-- departments and employees: the classic interview tables
-- ---------------------------------------------------------------------------
CREATE TABLE departments (
    dept_id    INTEGER      NOT NULL,
    dept_name  VARCHAR(50)  NOT NULL,
    location   VARCHAR(50),
    PRIMARY KEY (dept_id)
);

CREATE TABLE employees (
    emp_id      INTEGER      NOT NULL,
    emp_name    VARCHAR(60)  NOT NULL,
    dept_id     INTEGER,
    manager_id  INTEGER,
    salary      INTEGER,
    hire_date   DATE,
    city        VARCHAR(40),
    email       VARCHAR(80),
    PRIMARY KEY (emp_id)
);
```

SQL_SEED.SQL · PART 3

```
INSERT INTO departments (dept_id, dept_name, location) VALUES
  (1, 'QA',          'Bengaluru'),
  (2, 'Engineering', 'Hyderabad'),
  (3, 'Product',     'Pune'),
  (4, 'Support',     'Chennai'),
  (5, 'Data',        'Bengaluru');

-- Note: Asha has no manager, Ganesh has no department, two pairs of employees
-- share a salary, and two employees out-earn their own manager.
INSERT INTO employees (emp_id, emp_name, dept_id, manager_id, salary, hire_date, city, email) VALUES
  ( 1, 'Asha Nair',     2, NULL, 210000, '2018-03-12', 'Hyderabad', 'asha.nair@example.com'),
  ( 2, 'Rahul Menon',   1,    1,  95000, '2019-07-01', 'Bengaluru', 'rahul.menon@example.com'),
  ( 3, 'Divya Iyer',    1,    2,  78000, '2020-01-15', 'Bengaluru', 'divya.iyer@example.com'),
  ( 4, 'Karthik Rao',   2,    1, 120000, '2019-11-05', 'Hyderabad', 'karthik.rao@example.com'),
  ( 5, 'Meera Pillai',  2,    4, 135000, '2021-02-20', 'Hyderabad', 'meera.pillai@example.com'),
  ( 6, 'Sandeep Kumar', 3,    1,  88000, '2020-06-10', 'Pune',      'sandeep.kumar@example.com'),
  ( 7, 'Neha Sharma',   1,    2,  78000, '2021-09-01', 'Bengaluru', 'neha.sharma@example.com'),
  ( 8, 'Vikram Reddy',  4,    6,  62000, '2022-04-18', 'Chennai',   'vikram.reddy@example.com'),
  ( 9, 'Priya Das',     4,    6,  71000, '2022-08-25', 'Chennai',   'priya.das@example.com'),
  (10, 'Arjun Nair',    3,    6,  99000, '2023-01-09', 'Pune',      'arjun.nair@example.com'),
  (11, 'Fatima Sheikh', 2,    4, 120000, '2023-05-14', 'Hyderabad', 'fatima.sheikh@example.com'),
  (12, 'Ganesh Iyer', NULL,   1,  54000, '2024-02-02', 'Bengaluru', NULL);

-- ---------------------------------------------------------------------------
-- the application under test: customers, products, orders, payments
-- ---------------------------------------------------------------------------
CREATE TABLE customers (
    customer_id  INTEGER      NOT NULL,
    full_name    VARCHAR(60)  NOT NULL,
    email        VARCHAR(80),
    city         VARCHAR(40),
    signup_date  DATE,
    status       VARCHAR(20),
    PRIMARY KEY (customer_id)
);
```

SQL_SEED.SQL · PART 4

```
CREATE TABLE products (
    product_id    INTEGER       NOT NULL,
    product_name  VARCHAR(60)   NOT NULL,
    category      VARCHAR(30),
    price         DECIMAL(10,2),
    is_active     INTEGER,
    PRIMARY KEY (product_id)
);

CREATE TABLE orders (
    order_id     INTEGER       NOT NULL,
    customer_id  INTEGER,
    order_date   DATE,
    status       VARCHAR(20),
    amount       DECIMAL(10,2),
    PRIMARY KEY (order_id)
);

CREATE TABLE order_items (
    item_id     INTEGER       NOT NULL,
    order_id    INTEGER,
    product_id  INTEGER,
    qty         INTEGER,
    unit_price  DECIMAL(10,2),
    PRIMARY KEY (item_id)
);

CREATE TABLE payments (
    payment_id  INTEGER       NOT NULL,
    order_id    INTEGER,
    paid_on     DATE,
    amount      DECIMAL(10,2),
    method      VARCHAR(20),
    status      VARCHAR(20),
    PRIMARY KEY (payment_id)
);
```

SQL_SEED.SQL · PART 5

```
-- Note: Nisha has no email, and Aarti was created twice under two ids.
INSERT INTO customers (customer_id, full_name, email, city, signup_date, status) VALUES
  (1, 'Aarti Deshpande', 'aarti.d@example.com',   'Pune',      '2025-01-12', 'active'),
  (2, 'Rohit Sharma',    'rohit.s@example.com',   'Delhi',     '2025-02-03', 'active'),
  (3, 'Sneha Kulkarni',  'sneha.k@example.com',   'Mumbai',    '2025-02-19', 'active'),
  (4, 'Imran Qureshi',   'imran.q@example.com',   'Hyderabad', '2025-03-08', 'inactive'),
  (5, 'Lakshmi Menon',   'lakshmi.m@example.com', 'Kochi',     '2025-03-22', 'active'),
  (6, 'Tarun Gupta',     'tarun.g@example.com',   'Delhi',     '2025-04-01', 'active'),
  (7, 'Nisha Verma',     NULL,                    'Jaipur',    '2025-04-15', 'active'),
  (8, 'Aarti Deshpande', 'aarti.d@example.com',   'Pune',      '2025-05-02', 'active');

-- Note: the headset has never been ordered, and the hub is switched off.
INSERT INTO products (product_id, product_name, category, price, is_active) VALUES
  (101, 'Wireless Mouse',           'Accessories', 899.00,   1),
  (102, 'Mechanical Keyboard',      'Accessories', 4499.00,  1),
  (103, '27 inch Monitor',          'Displays',    18999.00, 1),
  (104, 'Laptop Stand',             'Accessories', 1299.00,  1),
  (105, 'USB C Hub',                'Accessories', 2499.00,  0),
  (106, 'Webcam 1080p',             'Peripherals', 3299.00,  1),
  (107, 'Noise Cancelling Headset', 'Peripherals', 7999.00,  1);

-- Note: order 1007 carries an amount that does not match its line items.
INSERT INTO orders (order_id, customer_id, order_date, status, amount) VALUES
  (1001, 1, '2025-05-04', 'delivered', 5398.00),
  (1002, 2, '2025-05-06', 'delivered', 18999.00),
  (1003, 3, '2025-05-06', 'cancelled', 899.00),
  (1004, 1, '2025-05-11', 'delivered', 1299.00),
  (1005, 5, '2025-05-15', 'shipped',   3299.00),
  (1006, 2, '2025-06-02', 'delivered', 899.00),
  (1007, 6, '2025-06-09', 'pending',   4999.00),
  (1008, 4, '2025-06-14', 'delivered', 2499.00),
  (1009, 1, '2025-06-21', 'returned',  18999.00),
  (1010, 5, '2025-07-02', 'delivered', 1798.00);
```

SQL_SEED.SQL · PART 6

```
INSERT INTO order_items (item_id, order_id, product_id, qty, unit_price) VALUES
  ( 1, 1001, 102, 1, 4499.00),
  ( 2, 1001, 101, 1, 899.00),
  ( 3, 1002, 103, 1, 18999.00),
  ( 4, 1003, 101, 1, 899.00),
  ( 5, 1004, 104, 1, 1299.00),
  ( 6, 1005, 106, 1, 3299.00),
  ( 7, 1006, 101, 1, 899.00),
  ( 8, 1007, 102, 1, 4499.00),
  ( 9, 1008, 105, 1, 2499.00),
  (10, 1009, 103, 1, 18999.00),
  (11, 1010, 101, 2, 899.00);

-- Note: order 1006 was delivered but never paid for, and order 1008 was paid
-- twice because the first attempt failed and the customer retried.
INSERT INTO payments (payment_id, order_id, paid_on, amount, method, status) VALUES
  (1, 1001, '2025-05-04', 5398.00,  'upi',        'success'),
  (2, 1002, '2025-05-06', 18999.00, 'card',       'success'),
  (3, 1004, '2025-05-11', 1299.00,  'upi',        'success'),
  (4, 1005, '2025-05-15', 3299.00,  'netbanking', 'success'),
  (5, 1008, '2025-06-14', 2499.00,  'card',       'failed'),
  (6, 1008, '2025-06-15', 2499.00,  'upi',        'success'),
  (7, 1009, '2025-06-21', 18999.00, 'card',       'success'),
  (8, 1010, '2025-07-02', 1798.00,  'upi',        'success');
```

SQL_SEED.SQL · PART 7

```
-- ------------------------------------------------------------------------
-- the QA angle: automation run history
-- ------------------------------------------------------------------------
CREATE TABLE test_runs (
    run_id        INTEGER      NOT NULL,
    suite_name    VARCHAR(40),
    environment   VARCHAR(20),
    started_at    VARCHAR(20),
    finished_at   VARCHAR(20),
    triggered_by  VARCHAR(20),
    PRIMARY KEY (run_id)
);

CREATE TABLE test_results (
    result_id      INTEGER       NOT NULL,
    run_id         INTEGER,
    test_name      VARCHAR(60),
    status         VARCHAR(20),
    duration_ms    INTEGER,
    error_message  VARCHAR(120),
    PRIMARY KEY (result_id)
);

-- Note: run 5 is still going, so it has no finish time yet.
INSERT INTO test_runs (run_id, suite_name, environment, started_at, finished_at, triggered_by) VALUES
  (1, 'regression', 'staging',    '2026-08-10 21:00:00', '2026-08-10 21:42:00', 'nightly'),
  (2, 'smoke',      'staging',    '2026-08-11 09:15:00', '2026-08-11 09:21:00', 'pipeline'),
  (3, 'regression', 'staging',    '2026-08-11 21:00:00', '2026-08-11 21:55:00', 'nightly'),
  (4, 'regression', 'production', '2026-08-12 06:00:00', '2026-08-12 06:38:00', 'release'),
  (5, 'smoke',      'production', '2026-08-12 07:00:00', NULL,                  'manual');
```

SQL_SEED.SQL · PART 8

```
-- Note: checkout_place_order fails and passes on the same suite and the same
-- environment. That is the definition of a flaky test, and you can prove it.
INSERT INTO test_results (result_id, run_id, test_name, status, duration_ms, error_message) VALUES
  ( 1, 1, 'login_valid_user',      'passed',  1240, NULL),
  ( 2, 1, 'login_invalid_password','passed',   980, NULL),
  ( 3, 1, 'checkout_place_order',  'failed',  3120, 'TimeoutError: waiting for selector #pay'),
  ( 4, 1, 'search_by_category',    'passed',   760, NULL),
  ( 5, 1, 'profile_update_email',  'failed',  2210, 'AssertionError: expected 200 got 500'),
  ( 6, 1, 'cart_add_remove',       'passed',  1450, NULL),
  ( 7, 2, 'login_valid_user',      'passed',  1180, NULL),
  ( 8, 2, 'checkout_place_order',  'passed',  2890, NULL),
  ( 9, 2, 'search_by_category',    'passed',   700, NULL),
  (10, 3, 'login_valid_user',      'passed',  1290, NULL),
  (11, 3, 'login_invalid_password','passed',  1010, NULL),
  (12, 3, 'checkout_place_order',  'failed',  3340, 'TimeoutError: waiting for selector #pay'),
  (13, 3, 'search_by_category',    'passed',   815, NULL),
  (14, 3, 'profile_update_email',  'passed',  1980, NULL),
  (15, 3, 'cart_add_remove',       'skipped', NULL, NULL),
  (16, 3, 'apply_coupon',          'failed',  1670, 'AssertionError: discount not applied'),
  (17, 4, 'login_valid_user',      'passed',  1120, NULL),
  (18, 4, 'login_invalid_password','passed',   940, NULL),
  (19, 4, 'checkout_place_order',  'passed',  2750, NULL),
  (20, 4, 'search_by_category',    'passed',   690, NULL),
  (21, 4, 'profile_update_email',  'passed',  1890, NULL),
  (22, 4, 'cart_add_remove',       'passed',  1390, NULL),
  (23, 4, 'apply_coupon',          'failed',  1720, 'AssertionError: discount not applied'),
  (24, 5, 'login_valid_user',      'passed',  1150, NULL),
  (25, 5, 'checkout_place_order',  'running', NULL, NULL);
```

SQL_SEED.SQL · PART 9

```
-- ------------------------------------------------------------------------
-- login_audit: rows that only make sense in order (for LAG, LEAD and gaps)
-- ------------------------------------------------------------------------
CREATE TABLE login_audit (
    audit_id     INTEGER      NOT NULL,
    customer_id  INTEGER,
    login_at     VARCHAR(20),
    ip_address   VARCHAR(20),
    outcome      VARCHAR(20),
    PRIMARY KEY (audit_id)
);

-- Note: customer 4 fails three times in a row and then gets in.
INSERT INTO login_audit (audit_id, customer_id, login_at, ip_address, outcome) VALUES
  ( 1, 2, '2026-08-12 08:01:00', '203.0.113.10',  'success'),
  ( 2, 4, '2026-08-12 08:05:00', '203.0.113.44',  'failed'),
  ( 3, 4, '2026-08-12 08:06:00', '203.0.113.44',  'failed'),
  ( 4, 4, '2026-08-12 08:07:00', '203.0.113.44',  'failed'),
  ( 5, 4, '2026-08-12 08:09:00', '203.0.113.44',  'success'),
  ( 6, 1, '2026-08-12 09:12:00', '198.51.100.7',  'success'),
  ( 7, 6, '2026-08-12 09:40:00', '198.51.100.23', 'failed'),
  ( 8, 6, '2026-08-12 09:41:00', '198.51.100.23', 'success'),
  ( 9, 3, '2026-08-12 10:02:00', '203.0.113.99',  'success'),
  (10, 1, '2026-08-12 11:30:00', '198.51.100.7',  'failed');
```

SQL_SEED.SQL · PART 10

```
-- ------------------------------------------------------------------------
-- customers_staging: the source side of a data migration, so you can practise
-- comparing expected against actual. Two things went wrong in this migration.
-- ------------------------------------------------------------------------
CREATE TABLE customers_staging (
    customer_id  INTEGER      NOT NULL,
    full_name    VARCHAR(60),
    email        VARCHAR(80),
    city         VARCHAR(40),
    signup_date  DATE,
    status       VARCHAR(20),
    PRIMARY KEY (customer_id)
);

INSERT INTO customers_staging (customer_id, full_name, email, city, signup_date, status) VALUES
  (1, 'Aarti Deshpande', 'aarti.d@example.com',   'Pune',      '2025-01-12', 'active'),
  (2, 'Rohit Sharma',    'rohit.s@example.com',   'Delhi',     '2025-02-03', 'active'),
  (3, 'Sneha Kulkarni',  'sneha.k@example.com',   'Mumbai',    '2025-02-19', 'active'),
  (4, 'Imran Qureshi',   'imran.q@example.com',   'Hyderabad', '2025-03-08', 'inactive'),
  (5, 'Lakshmi Menon',   'lakshmi.m@example.com', 'Cochin',    '2025-03-22', 'active'),
  (6, 'Tarun Gupta',     'tarun.g@example.com',   'Delhi',     '2025-04-01', 'active'),
  (7, 'Nisha Verma',     NULL,                    'Jaipur',    '2025-04-15', 'active'),
  (8, 'Aarti Deshpande', 'aarti.d@example.com',   'Pune',      '2025-05-02', 'active'),
  (9, 'Deepak Joshi',    'deepak.j@example.com',  'Nagpur',    '2025-05-19', 'active');
```

SQL_SEED.SQL · PART 11

```
-- ========================================================================
-- Running this on SQL Server
-- ------------------------------------------------------------------------
-- SQL Server does not understand DROP TABLE IF EXISTS on versions before 2016,
-- and it wants GO between batches in SSMS. Two small edits make it load:
--   1. Replace each DROP TABLE IF EXISTS x; with
--      IF OBJECT_ID('x','U') IS NOT NULL DROP TABLE x;
--   2. Add a GO line after the last DROP and after each CREATE TABLE block.
-- Everything else in this file is standard SQL and loads unchanged.
--
-- Running this on MySQL
-- ------------------------------------------------------------------------
-- Loads as written. If your server runs in strict mode and complains about the
-- DATE columns, the ISO format used here (YYYY-MM-DD) is already the format
-- MySQL wants, so no change is needed.
--
-- Running this on PostgreSQL
-- ------------------------------------------------------------------------
-- Loads as written. PostgreSQL folds unquoted names to lower case, which is
-- what this file uses throughout, so nothing needs quoting.
-- ========================================================================
```

IF THIS WAS USEFUL

What the complete guide adds

The file you are holding is a real slice of a bigger book, not a preview of one. Same sections, same practice database, same rule that every result is a real result. The complete guide carries **140 questions** instead of the 24 here, and the table below is its actual contents.

SECTION	QUESTIONS
1. Your Practice Database	5
2. Reading Data With Confidence	9
3. NULL And The Three Valued Logic	8
4. Joins	12
5. Grouping And Aggregation	12
6. Subqueries And EXISTS	11
7. Common Table Expressions	6
8. Window Functions	10
9. The Classic Interview Puzzles	12
10. Finding Bad Data	10
11. Dates And Times	9
12. Changing Data Safely	11
13. Indexes And Why A Query Is Slow	12
14. The Questions You Get Because You Are A Tester	13

It goes deeper everywhere it matters: the full set of join behaviours, subqueries against joins with the reasoning for choosing one, common table expressions including the recursive ones, the complete window function section, every classic puzzle with its edge case, deleting duplicates safely, the median, gaps in a sequence, isolation levels and deadlocks, and the whole data quality toolkit for checking a release.

You will find it at **sreenidhirajakrishnan.com**, along with the interview kits for Playwright, Selenium, API testing, Robot Framework and manual testing.

> **PLEASE PASS THIS ON**
> If a friend is preparing for interviews, send them this file. It is free, it is complete in itself, and the practice database in the appendix is worth having on its own.