

SQL Interview Questions and Answers

Intern to senior level

SQL interview questions and answers from beginner to senior: joins, WHERE vs HAVING, DELETE vs TRUNCATE, indexes, normalization, ACID, second highest salary, duplicate rows, isolation levels and query optimization.

13 questions Read online: <https://www.freedevtemplates.com/interview-questions/sql>

1. What is the difference between INNER JOIN and LEFT JOIN in SQL?

Level: intern SQL

INNER JOIN returns only rows that have a match in both tables, while **LEFT JOIN** returns every row from the left table and fills the right table's columns with NULL when there is no match.

Use INNER JOIN when you only want records that exist on both sides, such as orders that have a customer. Use LEFT JOIN when the left table is the main list and the related data is optional, such as all customers including those who have never placed an order.

A common trick is to find rows with no match by left joining and filtering on WHERE right.id IS NULL. Be careful: putting a condition on the right table in the WHERE clause turns a left join back into an inner join, so put it in the ON clause instead. RIGHT JOIN is the mirror image of LEFT JOIN, and FULL OUTER JOIN keeps unmatched rows from both sides.

Key points

INNER: matches only. LEFT: all left rows plus matches

Unmatched right-side columns are NULL

Filter the right table in ON, not WHERE, to keep left-join behaviour

```
-- every customer, with order count (0 if none)
SELECT c.id, c.name, COUNT(o.id) AS orders
FROM customers c
LEFT JOIN orders o ON o.customer_id = c.id
GROUP BY c.id, c.name;

-- customers who never ordered
SELECT c.*
FROM customers c
LEFT JOIN orders o ON o.customer_id = c.id
WHERE o.id IS NULL;
```

2. What is the difference between WHERE and HAVING in SQL?

Level: intern SQL

WHERE filters individual rows before grouping, while **HAVING** filters groups after GROUP BY, so only **HAVING** can use aggregate functions like COUNT or SUM.

SQL processes a query in a logical order: FROM, WHERE, GROUP BY, HAVING, SELECT, ORDER BY. Because WHERE runs before grouping, the aggregates do not exist yet. HAVING runs after grouping, so it can test them.

Put every condition that does not need an aggregate in WHERE. It reduces the rows before grouping, which is faster than filtering the groups afterwards.

```
SELECT department, COUNT(*) AS staff, AVG(salary) AS avg_salary
FROM employees
WHERE status = 'active' -- filters rows first
GROUP BY department
HAVING COUNT(*) >= 5; -- filters groups after aggregation
```

3. What is the difference between DELETE, TRUNCATE and DROP in SQL?

Level: *intern SQL*

DELETE removes selected rows and can be rolled back, TRUNCATE quickly removes all rows and keeps the table structure, and DROP removes the entire table including its structure.

DELETE is a DML command: it can use a WHERE clause, fires row-level triggers, and is logged row by row, so it is slower on big tables but fully transactional. TRUNCATE is a DDL command that empties the whole table, usually resets identity counters and is much faster because it deallocates data pages instead of logging each row. DROP TABLE deletes the table definition, indexes and permissions as well.

Transaction behaviour differs by database: in PostgreSQL and SQL Server TRUNCATE can be rolled back inside a transaction, while in MySQL and Oracle it causes an implicit commit. Always check your database's documentation before relying on it.

```
DELETE FROM orders WHERE created_at < '2024-01-01'; -- some rows
TRUNCATE TABLE temp_imports; -- all rows, keep table
DROP TABLE temp_imports; -- table is gone
```

4. What is the difference between a primary key and a foreign key?

Level: *intern SQL*

A primary key uniquely identifies each row in its own table and cannot be NULL, while a foreign key is a column that references the primary key of another table to link the two.

A table has one primary key (which may be made of several columns). It guarantees uniqueness and is indexed automatically. A table can have many foreign keys. A foreign key enforces referential integrity: you cannot insert an order for a customer that does not exist, and you cannot delete a customer that still has orders unless you define a cascade rule such as ON DELETE CASCADE.

A foreign key column may contain duplicates and, unless declared NOT NULL, may be NULL. A unique key also enforces uniqueness but, unlike a primary key, allows NULL (the exact number of NULLs allowed depends on the database).

```
CREATE TABLE customers (
  id INT PRIMARY KEY,
  name VARCHAR(100) NOT NULL
);

CREATE TABLE orders (
  id INT PRIMARY KEY,
  customer_id INT NOT NULL,
  FOREIGN KEY (customer_id) REFERENCES customers(id) ON DELETE RESTRICT
);
```

5. How do you find the second highest salary in SQL?

Level: *junior SQL*

Use a subquery that takes the MAX salary below the overall MAX, or use DENSE_RANK() and pick rank 2,

which also handles ties and generalises to the Nth highest.

The subquery version works on every database. The window function version is the one interviewers like best, because changing 2 to N gives you the Nth highest salary, and DENSE_RANK correctly treats equal salaries as the same rank. Use DISTINCT so tied rows return one value.

Another common answer is ORDER BY salary DESC LIMIT 1 OFFSET 1 (MySQL, PostgreSQL) or OFFSET 1 ROWS FETCH NEXT 1 ROWS ONLY (SQL Server), but you need SELECT DISTINCT salary first or ties break it. If there is no second salary, wrap the query so it returns NULL rather than an empty result when the question asks for that.

Key points

Subquery with MAX is the simplest answer

DENSE_RANK handles ties and any N

ROW_NUMBER would be wrong when salaries tie

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

-- Nth highest with window function (here N = 2)
SELECT DISTINCT salary
FROM (
  SELECT salary, DENSE_RANK() OVER (ORDER BY salary DESC) AS rnk
  FROM employees
) ranked
WHERE rnk = 2;
```

6. How do you find and delete duplicate rows in SQL?

Level: junior SQL

Find duplicates with GROUP BY and HAVING COUNT(*) > 1, then delete them by numbering rows within each duplicate group with ROW_NUMBER() and removing every row after the first.

First decide what makes two rows duplicates (for example the same email) and which row you want to keep (usually the oldest, so the lowest id). Always run the matching SELECT before the DELETE, and take a backup or work inside a transaction.

ROW_NUMBER() OVER (PARTITION BY email ORDER BY id) numbers rows inside each group of identical emails, so any row with a number above 1 is a duplicate. In MySQL older than 8.0 window functions are unavailable, and you cannot select from the same table in a DELETE subquery directly, so you wrap the subquery in a derived table or self-join instead. Afterwards, add a unique constraint so the duplicates cannot come back.

```
-- 1. find
SELECT email, COUNT(*) AS copies
FROM users
GROUP BY email
HAVING COUNT(*) > 1;

-- 2. delete, keeping the lowest id (PostgreSQL / SQL Server / MySQL 8+)
WITH ranked AS (
  SELECT id, ROW_NUMBER() OVER (PARTITION BY email ORDER BY id) AS rn
  FROM users
)
DELETE FROM users WHERE id IN (SELECT id FROM ranked WHERE rn > 1);

-- 3. prevent it happening again
ALTER TABLE users ADD CONSTRAINT uq_users_email UNIQUE (email);
```

7. What is the difference between UNION and UNION ALL in SQL?

Level: junior SQL

UNION combines two result sets and removes duplicate rows, while UNION ALL combines them and keeps every row, which makes it faster.

UNION has to sort or hash the combined result to find duplicates, which costs time and memory. If you know the two queries cannot return the same row, or you want the duplicates, use UNION ALL. Both require the same number of columns with compatible types, and the column names come from the first query.

```
SELECT email FROM newsletter_subscribers
UNION -- one row per distinct email
SELECT email FROM customers;
```

```
SELECT email FROM newsletter_subscribers
UNION ALL -- keeps duplicates, faster
SELECT email FROM customers;
```

8. What is database normalization and what are the normal forms?

Level: junior SQL

Normalization is organising tables to reduce redundancy and update anomalies: 1NF means atomic values, 2NF removes partial dependencies on part of a composite key, and 3NF removes dependencies between non-key columns.

In 1NF every column holds a single value (no comma-separated lists) and each row is unique. In 2NF the table is in 1NF and every non-key column depends on the whole primary key, which matters for composite keys. In 3NF the table is in 2NF and non-key columns depend only on the key, not on other non-key columns (a city and zip_code pair stored with an order is an example of what to move out).

Normalisation avoids storing the same fact in many places, so updates and deletes cannot leave inconsistent data. The trade-off is more joins. Reporting databases and read-heavy systems often denormalise on purpose, accepting some redundancy for faster reads.

Key points

1NF: atomic values. 2NF: no partial dependency. 3NF: no transitive dependency

Goal: remove insert, update and delete anomalies

Denormalise deliberately for read performance

9. What are ACID properties in a database?

Level: mid SQL

ACID stands for Atomicity, Consistency, Isolation and Durability: a transaction either fully succeeds or fully fails, keeps the data valid, does not interfere with other transactions, and survives crashes once committed.

Atomicity: all statements in a transaction are applied or none are, so a bank transfer never debits one account without crediting the other. Consistency: a transaction moves the database from one valid state to another, respecting constraints. Isolation: concurrent transactions behave as if they ran in some order, to the degree set by the isolation level. Durability: once committed, data is stored permanently, typically via a write-ahead log.

Relational databases provide ACID by default. Many NoSQL systems relax some of these guarantees to gain availability or scale, which is the trade-off behind the BASE and CAP discussions.

```
BEGIN;
UPDATE accounts SET balance = balance - 500 WHERE id = 1;
UPDATE accounts SET balance = balance + 500 WHERE id = 2;
COMMIT; -- or ROLLBACK; if anything failed
```

10. What is an index in SQL and how does it improve performance?

Level: mid SQL

An index is a separate sorted data structure (usually a B-tree) that lets the database find rows without scanning the whole table, at the cost of extra storage and slower writes.

Without an index, a query like `WHERE email = 'a@b.com'` reads every row (a full table scan). With an index on email the database walks the tree in a handful of steps and jumps straight to the matching rows. Indexes also speed up `JOIN`, `ORDER BY` and `GROUP BY` on the indexed columns.

Every `INSERT`, `UPDATE` and `DELETE` must also update every index on the table, so do not index everything. In a composite index the column order matters: an index on `(last_name, first_name)` helps a search on `last_name` alone but not on `first_name` alone. An index is also skipped when you wrap the column in a function (`WHERE LOWER(email) = ...`) or use a leading wildcard (`LIKE '%smith'`), and it is of little use on low-cardinality columns such as a boolean.

Key points

Faster reads, slower writes, more storage

Column order in composite indexes matters

Functions on the column and leading wildcards prevent index use

```
CREATE INDEX idx_orders_customer_date ON orders (customer_id, created_at);

-- uses the index
SELECT * FROM orders WHERE customer_id = 42 AND created_at >= '2026-01-01';
-- cannot use it efficiently (function on the column)
SELECT * FROM orders WHERE DATE(created_at) = '2026-01-01';
```

11. What is the difference between a clustered and a non-clustered index?

Level: mid SQL

A clustered index defines the physical order in which the table's rows are stored, so there can be only one per table, while a non-clustered index is a separate structure that points back to the rows.

In SQL Server and MySQL InnoDB the primary key is the clustered index by default, so the table data is the leaf level of that tree. Looking up by the clustered key is the fastest possible read, and range scans on it are efficient because neighbouring keys are stored together.

A non-clustered index stores the indexed columns plus a pointer (the clustered key or a row locator). A lookup finds the entry and then does a second lookup to fetch the remaining columns, unless the index covers the query by including all needed columns. Choose a narrow, ever-increasing clustered key, because wide or random keys (such as random GUIDs) are copied into every non-clustered index and cause page splits.

12. What are transaction isolation levels in SQL and what problems do they prevent?

Level: senior SQL

Isolation levels (Read Uncommitted, Read Committed, Repeatable Read, Serializable) control how much concurrent transactions can see of each other's changes, trading consistency against concurrency to prevent dirty reads, non-repeatable reads and phantom reads.

A dirty read means reading data another transaction has not committed. A non-repeatable read means reading the same row twice in one transaction and getting different values because someone committed in between. A phantom read means running the same query twice and getting a different set of rows because rows were inserted or deleted.

Read Uncommitted allows all three. Read Committed prevents dirty reads (the default in PostgreSQL, SQL Server and Oracle). Repeatable Read also prevents non-repeatable reads (the default in MySQL InnoDB, and it also prevents most phantoms there). Serializable prevents all three by making transactions behave as if they ran

one after another, at the highest cost in locking or retries.

Choose the lowest level that is correct for the business rule, and use explicit locking (SELECT ... FOR UPDATE) or optimistic concurrency with a version column for specific hot spots such as inventory or balances.

Key points

Higher isolation = fewer anomalies but less concurrency

Defaults differ by database

Use row locks or version columns for critical updates

13. How do you optimize a slow SQL query?

Level: senior SQL

Read the execution plan with EXPLAIN to find full scans and expensive joins, add or fix indexes for the filter and join columns, select only the columns and rows you need, and rewrite predicates so indexes can be used.

Start by measuring, not guessing: run EXPLAIN (or EXPLAIN ANALYZE in PostgreSQL, or the actual execution plan in SQL Server) and look for sequential scans on large tables, big gaps between estimated and actual row counts, sorts that spill to disk, and nested loops over huge inputs.

Typical fixes: add an index matching the WHERE and JOIN columns (and ORDER BY where possible); avoid SELECT *; make predicates sargable (compare the raw column, not a function of it); replace OFFSET pagination on large offsets with keyset pagination (WHERE id > last_seen_id ORDER BY id LIMIT 50); avoid N+1 query patterns from the application; update table statistics; and archive or partition very large tables.

If the query is still too slow, consider a read replica, caching, a materialised view or precomputed summary table. Confirm every change with the plan and with timings on production-sized data.

Key points

Measure with the execution plan first

Indexes, sargable predicates, fewer columns and rows

Keyset pagination beats large OFFSET

```
EXPLAIN ANALYZE
SELECT o.id, o.total
FROM orders o
WHERE o.customer_id = 42
AND o.created_at >= '2026-01-01'
ORDER BY o.created_at DESC
LIMIT 50;

-- keyset pagination instead of a large OFFSET
SELECT id, total FROM orders
WHERE id < 98765
ORDER BY id DESC
LIMIT 50;
```

These answers are written for real interview conversations. Prefer explaining a trade-off out loud over reciting a definition. Full explanations and related questions: <https://www.freedevtemplates.com/interview-questions/sql>