

Database Performance

Platform, SRE & DevOps interview guide

Questions	97 questions · spoken answers included
Online (always latest)	interviewhub.eknathalabs.com/database_performance_interview_guide.html
Printed	2026-10-21

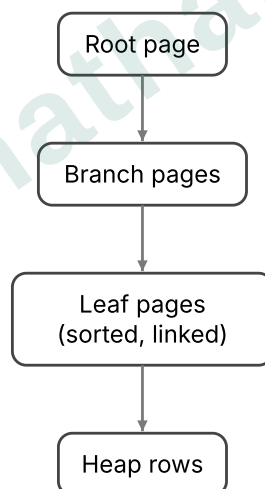
Free to use and share for personal study. Answers are written for interview practice, so check command flags and versions against the official documentation before running anything in production.

Spoken answer · ~89 s

An index is a separate data structure that lets the database find rows without scanning the whole table. The default type in most relational databases is the B-tree, a balanced tree where each node is a page holding sorted keys and pointers. The root points to internal nodes, which point to leaf nodes; in a B+ tree variant, all entries live in the leaves, which are linked together in order. A lookup starts at the root, compares the search key to keys in the page to pick the right child, and descends a few levels, usually three or four even for hundreds of millions of rows, because each page holds hundreds of entries. That makes lookups logarithmic. The leaf entry gives the row location, such as a heap tuple pointer in PostgreSQL or the primary key in an InnoDB secondary index. Because leaves are sorted and linked, B-trees also support range scans, ordering, and prefix matching efficiently. Inserts find the right leaf and add an entry; when a page is full, it splits, which keeps the tree balanced but can fragment it. The costs: indexes use disk and memory, and slow writes since every insert, update, or delete must maintain them. So index columns used in filters, joins, and sorts, and don't over-index.

Likely follow-ups

- Why are B-tree lookups logarithmic?
- What does an index cost on writes?

B-tree index

An index is a separate data structure that allows the database engine to find rows without scanning every row in the table. Without an index, every query does a full sequential scan — $O(n)$. With a B-tree index, lookups are $O(\log n)$.

B-TREE MECHANICS

B-tree: balanced tree where each node holds sorted keys + pointers to children

Leaf nodes: hold actual index entries → pointers to heap (table) rows

Height: typically 3-4 levels for millions of rows ($\log_{base\ 100}\text{ of }10M \approx 3.5$)

-- PostgreSQL: inspect index usage

```
SELECT schemaname, tablename, indexname, idx_scan, idx_tup_read FROM pg_stat_user_indexes
ORDER BY idx_scan;
```

B-tree supports: equality (=), range (<, >, BETWEEN), ORDER BY, IS NULL. It does NOT efficiently support: leading wildcard LIKE ('%foo'), full-text search, geometric queries, or array containment. For those, use specialized index types: GIN (full-text, arrays), GiST (geometric, range types), BRIN (large sequential tables like timestamps), Hash (equality-only, tiny).

Interview tip: Always explain the trade-off — indexes speed up reads but slow down writes (INSERT/UPDATE/DELETE must update all indexes on that table). Every additional index has a write-amplification cost. In write-heavy tables, too many indexes hurt overall throughput.

Official docs

- [PostgreSQL: Indexes](#)
- [Use The Index, Luke!](#)

Q2 When does a query NOT use an index even though one exists — what are the common pitfalls?

Indexing

Spoken answer · ~89 s

Several situations cause the optimiser to ignore an existing index. A function or expression on the column, such as WHERE LOWER(email) = 'x' or WHERE date(created_at) = ..., prevents using a plain index; fix it by rewriting the predicate or creating an expression index. Implicit type conversions, like comparing a string column to a number, defeat the index. A leading wildcard, LIKE '%abc', can't use a B-tree because it can't seek by prefix. For composite indexes, skipping the leftmost column means the index isn't usable. Low selectivity: if the query returns a large fraction of the table, say more than ten to twenty percent, a sequential scan is cheaper than many random lookups, so the planner rightly chooses it. Stale statistics lead to wrong row estimates and bad plans; run ANALYZE. Small tables are scanned anyway. OR conditions across different columns, negations like != or NOT IN, and large IN lists can prevent index use. Different collation or data types in joins break matching. Functions in ORDER BY, or ordering that doesn't match the index, force sorts. Also, parameter sniffing in some databases and planner cost settings such as random_page_cost. Diagnose with EXPLAIN, check statistics and the query's predicates, and test whether forcing the index actually helps before assuming the planner is wrong.

Likely follow-ups

- Why might a sequential scan beat an index?
- How do expression indexes help?

```
-- Function wrapping prevents index use:
```

```
WHERE UPPER(email) = 'FOO@BAR.COM'      -- scans table (function on column)
```

```
WHERE email = UPPER('foo@bar.com')      -- uses index (function on literal)
```

```
-- Type mismatch:
```

```
WHERE user_id = '123' -- user_id is INT, implicit cast kills index
```

```
-- Leading wildcard:
```

```
WHERE name LIKE '%smith' -- full scan (no prefix to match in B-tree)
```

```
WHERE name LIKE 'smith%' -- uses index (prefix match)
```

```
-- Low cardinality: planner may prefer seq scan if index selectivity is poor
```

```
WHERE status = 'active' -- if 90% of rows are 'active', seq scan wins
```

The planner may also skip an index if: the table is small (sequential scan is faster for tiny tables), statistics are stale (ANALYZE hasn't run), or the cost estimate is wrong (adjust `random_page_cost` for SSD). Force index use for debugging with `SET enable_seqscan = off` (PostgreSQL) or `FORCE INDEX` (MySQL) — but never use these in production as permanent fixes.

Implicit casts: Passing a string where an integer is expected causes a cast on every row — the index is useless. Always match parameter type to column type. In ORMs, inspect the generated SQL to catch type mismatches.

Official docs

- [PostgreSQL: Index usage \(why an index is not used\)](#)

Q3

What is a composite index — how does column order affect its usefulness?

Indexing

Spoken answer · ~83 s

A composite index covers several columns in a defined order, and it behaves like a phone book sorted by last name, then first name. The index is sorted by the first column, then by the second within equal first values, and so on. Therefore column order determines which queries it can serve. It supports equality or range on the leftmost prefix: an index on (a, b, c) helps queries filtering on a; on a and b; on a, b, and c; but not on b alone or c alone, and only partially on a and c. A range condition on a column stops later columns from being used for seeking; so equality columns should come before the range column. Rules of thumb: put equality-filtered columns first, then the range or sort column; among equality columns, put the most selective first, although query patterns matter more than selectivity; and put columns that serve ORDER BY in the right place so the sort can be avoided. One composite index can replace several single-column ones, saving write overhead, and it can be a covering index if it includes all needed columns. Avoid redundant indexes: (a, b) makes a separate index on (a) unnecessary.

Likely follow-ups

- Why should equality columns come before range columns?
- Does an index on (a, b) help a query on b alone?

```
CREATE INDEX idx_user_status ON orders (user_id, status, created_at);
```

```
-- This index supports these queries efficiently:
```

```

WHERE user_id = 5                                -- uses index (leftmost prefix)

WHERE user_id = 5 AND status = 'shipped'         -- uses index

WHERE user_id = 5 AND status = 'shipped' AND created_at > '2024-01-01' -- full use

-- These are NOT efficiently supported:

WHERE status = 'shipped'                         -- skips leftmost column, full scan

WHERE created_at > '2024-01-01'                 -- skips two leftmost columns

```

The leftmost prefix rule: a composite index on (A, B, C) can be used for queries filtering on A, A+B, or A+B+C — but not B alone or C alone. Column order rule: put equality columns first, then range columns last (range condition stops further index use for subsequent columns). Put the most selective column first within the equality group to minimize the rows scanned.

Design heuristic: For a query `WHERE user_id = ? AND status = ? ORDER BY created_at`, the ideal index is `(user_id, status, created_at)` — the `ORDER BY` is satisfied by the index, eliminating a sort step. This pattern (equality + range/sort) is one of the most common index optimization patterns.

Official docs

- [PostgreSQL: Indexes](#)
- [Use The Index, Luke!](#)

Q4 What is an index covering query — what is a covering index?

Indexing

Spoken answer · ~88 s

A covering index contains all the columns a query needs, so the database can answer it from the index alone without visiting the table. In PostgreSQL this appears as an Index Only Scan, and in MySQL as Using index in EXPLAIN. It avoids the extra random I/O of fetching rows from the heap or clustered index, which can speed up queries significantly, especially for large tables or queries returning many rows. You build one by including the filtered and selected columns in the index. In PostgreSQL 11 and later, the `INCLUDE` clause adds non-key payload columns that are stored in the leaf entries but not used for ordering or uniqueness, keeping the key small: for example `CREATE INDEX ON orders (customer_id, created_at) INCLUDE (total)`. In MySQL InnoDB, secondary indexes automatically contain the primary key, so queries needing only indexed columns plus the primary key are covered. Caveats: wide indexes use more space and slow writes; in PostgreSQL, index-only scans rely on the visibility map, so a table with many recent updates or insufficient vacuuming still must check the heap; covering everything for every query defeats the purpose, so reserve it for hot, frequent queries. Verify with EXPLAIN that heap fetches are low, and compare the benefits to the write overhead.

Likely follow-ups

- What does `INCLUDE` do in PostgreSQL?
- Why does vacuum matter for index-only scans?

Covering index

Regular index	Covering index
Index lookup then heap fetch	All columns in index
Extra I/O	Index-only scan

```
-- Without covering index: index lookup → heap fetch for each row
```

```
SELECT name FROM users WHERE email = 'x@y.com'; -- index on email, heap fetch for name
```

```
-- With covering index: no heap fetch needed
```

```
CREATE INDEX idx_email_name ON users (email) INCLUDE (name); -- PostgreSQL 11+
```

```
CREATE INDEX idx_email_name ON users (email, name); -- MySQL covering index
```

```
-- EXPLAIN shows "Index Only Scan" in PostgreSQL if covered
```

```
EXPLAIN SELECT name FROM users WHERE email = 'x@y.com';
```

A covering index contains all columns needed by the query — the database never needs to go back to the heap (table) for additional data. This eliminates random I/O for heap fetches, which is often the most expensive part of an index lookup. In PostgreSQL, use `INCLUDE` to add non-key columns to the index without affecting sort order. In MySQL, all columns in the `SELECT` and `WHERE` must be in the index for an Index Only Scan (key-only lookup).

When to use: Covering indexes are most valuable for high-frequency, read-heavy queries on large tables. The trade-off is larger index size and slower writes. Profile first — don't preemptively add `INCLUDE` columns to every index.

Official docs

- [PostgreSQL: Indexes](#)
- [Use The Index, Luke!](#)

Spoken answer · ~84 s

Index bloat is wasted space inside an index: pages that are mostly empty or filled with dead entries. It arises because updates and deletes in MVCC databases leave dead tuples, and index entries pointing to them remain until vacuum cleans them; pages may then be only partly reused, and B-tree pages are rarely merged, so the index grows larger than the live data needs. Heavy update or delete workloads, long-running transactions that block vacuum cleanup, and random key insertion patterns worsen it. Effects: larger indexes mean more disk and fewer pages fitting in memory, slower scans, and more I/O. Detection in PostgreSQL: compare index size against expected size, use the pgstattuple extension's pgstatindex for leaf density and fragmentation, use bloat estimation queries, or monitor index size growth over time. In MySQL, check data_free and fragmentation in information_schema. Fixes: REINDEX rebuilds the index, and REINDEX CONCURRENTLY in PostgreSQL 12 and later does so without blocking writes; pg_repack can rebuild tables and indexes online; in MySQL, OPTIMIZE TABLE or ALTER TABLE rebuilds. Prevention: tune autovacuum to run frequently on busy tables, avoid long transactions, use HOT updates by leaving fillfactor free and not indexing frequently changing columns, drop unused indexes, and monitor bloat regularly.

Likely follow-ups

- How do long transactions cause bloat?
- How do you rebuild an index without blocking writes?

```
-- PostgreSQL: check index bloat
```

```
SELECT relname, pg_size_pretty(pg_relation_size(oid)) AS size FROM pg_class WHERE relkind = 'i' ORDER BY pg_relation_size(oid) DESC LIMIT 20;
```

```
-- Extension for detailed bloat estimate:
```

```
SELECT * FROM pgstattuple('idx_name'); -- requires pgstattuple extension
```

```
-- Fix: rebuild the index (non-blocking in PG 12+)
```

```
REINDEX INDEX CONCURRENTLY idx_name;
```

```
-- MySQL: check index fragmentation
```

```
SELECT TABLE_NAME, DATA_FREE FROM information_schema.TABLES WHERE TABLE_SCHEMA = 'mydb';
```

```
OPTIMIZE TABLE tablename; -- rebuilds and defragments (locks table in older MySQL)
```

Index bloat occurs when deleted/updated rows leave dead entries in the index. B-tree pages become sparse — more disk reads per lookup, more cache pressure. In PostgreSQL, VACUUM reclaims heap space but index bloat accumulates over time and requires REINDEX. Triggers for rebuilding: index is significantly larger than it should be, slow queries despite the index being used, or after bulk deletes/updates of a large fraction of the table.

REINDEX vs REINDEX CONCURRENTLY: Plain REINDEX locks the table — do not use in production. REINDEX CONCURRENTLY (PostgreSQL 12+) builds a new index alongside the old one and swaps atomically. Takes 2x space temporarily but causes zero downtime.

Official docs

- [PostgreSQL: Routine vacuuming](#)

Q6 How do you read and interpret a query execution plan (EXPLAIN ANALYZE)?

Query optimization

Spoken answer · ~88 s

EXPLAIN shows the plan the optimiser chose, and EXPLAIN ANALYZE actually runs the query and adds real timings and row counts, so you can compare estimates to reality. The plan is a tree; read it from the innermost, most indented nodes outward, since data flows upward. Each node shows the operation, such as Seq Scan, Index Scan, Index Only Scan, Bitmap Heap Scan, Nested Loop, Hash Join, Merge Join, Sort, or Aggregate, with estimated cost, estimated rows, and with ANALYZE the actual time, actual rows, and loops. Key things I look for: large gaps between estimated and actual rows, which signal stale statistics or correlated columns and often explain bad plan choices; sequential scans on big tables where an index should apply; expensive sorts that spill to disk; nested loops with high loop counts; the node where most actual time is spent, remembering to multiply by loops; filters removing many rows after the scan, shown as rows removed by filter, suggesting a missing or better index; and in PostgreSQL, using BUFFERS to see cache hits versus disk reads. Remember that ANALYZE executes the statement, so wrap data-modifying queries in a transaction and roll back. Then act: add or adjust indexes, rewrite the query, update statistics, or change settings, and re-measure.

Likely follow-ups

- Why compare estimated and actual rows?
- Why wrap EXPLAIN ANALYZE of an UPDATE in a transaction?

```
EXPLAIN (ANALYZE, BUFFERS, FORMAT TEXT) SELECT ...; -- PostgreSQL full analysis
```

```
EXPLAIN FORMAT=JSON SELECT ...; -- MySQL (8.0+)
```

```
-- Key nodes to understand (PostgreSQL):
```

```
Seq Scan      -- full table scan (look for large row_count × cost)
```

```
Index Scan    -- uses index, then fetches heap rows
```

```
Index Only Scan -- covers query without heap fetch (best case)
```

```
Bitmap Heap Scan -- combines multiple indexes, batches heap fetches
```

```
Hash Join     -- builds hash table on smaller relation (good for large joins)
```

```
Nested Loop   -- for each outer row, probe inner (good when inner is small/indexed)
```

Merge Join -- both sides sorted, merge in order (good for large sorted relations)

Key metrics: actual rows vs estimated rows (big discrepancy = stale statistics → run ANALYZE), actual time per node (find the expensive node), Buffers hit vs read (cache hit rate — more hits = better), and loops × time (nested loop with many loops can be very expensive). The plan reads bottom-up — the innermost node executes first.

Rows estimate vs actual: If estimated rows = 1 but actual rows = 100,000, the planner underestimated selectivity and may have chosen a bad join order or wrong index. Fix: `ANALYZE tablename` to refresh statistics, or increase `default_statistics_target` for skewed columns.

Official docs

- [PostgreSQL: Using EXPLAIN](#)
- [PostgreSQL: Performance tips](#)

Q7 What is the N+1 query problem — how do you detect and fix it?

Query optimization

Spoken answer · ~88 s

The N+1 query problem happens when code runs one query to fetch a list of N parent records, then runs an additional query per parent to fetch related data, resulting in N+1 round trips. For example, loading 100 orders and then querying each order's customer separately gives 101 queries. It often comes from ORMs with lazy loading, where accessing a relationship in a loop triggers a hidden query. Each query is fast, so it's invisible in slow query logs, but the total latency and database load multiply with the data size, and network round trips dominate. Detection: enable query logging or counting in development and tests; use APM and tracing tools showing many repeated similar queries within one request; look at `pg_stat_statements` for queries with huge call counts; and libraries such as Bullet for Rails or django-debug-toolbar, or assertions on query counts in tests. Fixes: eager loading, such as `select_related` or `prefetch_related` in Django, `includes` in Rails, or `JOIN FETCH` in JPA, which uses a join or one extra query with an IN list for all parents; write an explicit join; batch lookups with a data loader pattern, common in GraphQL; or cache lookups. Be careful that eager loading everything can fetch too much data, and large joins can duplicate rows.

Likely follow-ups

- Why does N+1 not show up in the slow query log?
- How does a data loader solve it in GraphQL?

N+1 queries

N+1	Fixed
1 query for list + N per row	JOIN or batched IN query
Slow under load	Constant query count

-- N+1 pattern: 1 query to get list + N queries to get related data

```

SELECT * FROM orders LIMIT 100;           -- 1 query

SELECT * FROM users WHERE id = 1;         -- query per order (N=100)

SELECT * FROM users WHERE id = 2;         -- ... repeated 100 times

-- Fix: JOIN or IN clause

SELECT o.*, u.name FROM orders o JOIN users u ON o.user_id = u.id LIMIT 100;

-- Or: batch load with IN

SELECT * FROM users WHERE id IN (1,2,3,...,100); -- 1 query for all users

-- Detection in PostgreSQL: log slow queries

SET log_min_duration_statement = 10; -- log queries taking > 10ms

```

N+1 queries are invisible in development (small datasets) but catastrophic in production (thousands of records). An ORM loading relationships lazily is the primary cause. Detection: `pg_stat_statements` (shows query frequency — same query called 10,000 times/minute), slow query log, or APM tools showing many identical fast queries. Fix at the ORM level: eager loading (Hibernate's `JOIN FETCH`, Rails' `includes`, Sequelize's `include`).

DataLoader pattern: For GraphQL APIs, use DataLoader to batch and deduplicate queries within a single request — collects all IDs needed, fires one IN query. This solves N+1 at the API layer without changing the data model.



Official docs

- [Use The Index, Luke: N+1 / ORM pitfalls](#)

Spoken answer · ~85 s

When a query was fast and became slow, something changed, so I look for what. Start by measuring: is it slow now consistently, or intermittently, for everyone or specific parameters? Compare the current EXPLAIN ANALYZE against the old plan if available, from logs or auto_explain. The commonest causes: data growth, where a table crossed a size or selectivity threshold; stale or inaccurate statistics, causing the planner to choose a poor plan, so run ANALYZE; a plan flip, for example from index scan to sequential scan or a different join order; bloat from a lack of vacuuming; a dropped or invalid index, or an index that changed after a migration; changes in the query or the ORM, a new deployment, or different parameter values hitting skewed data. Resource contention: locks and blocked sessions, long-running transactions, connection saturation, CPU, memory pressure causing cache evictions, disk I/O saturation, or noisy neighbours on shared infrastructure; checkpoints and autovacuum activity at the same time. Also configuration changes, version upgrades, and failover to a replica with a cold cache. Use pg_stat_statements to see when mean time changed, check locks in pg_locks and pg_stat_activity, and review monitoring dashboards and deployment history. Fix the cause, then add alerting on query latency regression.

Likely follow-ups

- What is a plan flip?
- How do you compare with the old plan?

```
-- Step 1: Run EXPLAIN ANALYZE and compare to expected plan
```

```
EXPLAIN (ANALYZE, BUFFERS) SELECT ...;
```

```
-- Step 2: Check if statistics are stale
```

```
SELECT last_analyze, last_autoanalyze, n_live_tup, n_dead_tup FROM pg_stat_user_tables
WHERE relname = 'tablename';
```

```
ANALYZE tablename; -- refresh statistics manually
```

```
-- Step 3: Check data growth
```

```
SELECT pg_size_pretty(pg_total_relation_size('tablename'));
```

```
SELECT COUNT(*) FROM tablename; -- has table grown 10x?
```

```
-- Step 4: Check if index is being used (or was dropped)
```

```
SELECT * FROM pg_indexes WHERE tablename = 'tablename';
```

Common causes: table grew past the threshold where the planner switches from index scan to seq scan (statistics are stale, planner doesn't know it grew), an index was accidentally dropped during a migration, autovacuum is behind and dead tuple bloat is skewing statistics, data distribution changed (a value that was rare is now 50% of rows — index no longer selective), or a schema migration changed a column type breaking index compatibility.

Plan regression after deploy: A schema migration (adding a column, changing a type, adding a constraint) triggers a statistics reset. The planner may choose a different plan for a few minutes until autovacuum/autoanalyze catches up. This causes a classic "deploy → slow queries → recovers on its own in 5 minutes" pattern.

Official docs

- [PostgreSQL: Performance tips](#)
- [PostgreSQL: Routine vacuuming](#)

Q9 What are window functions and when do they outperform subqueries or self-joins?

Query optimization

Spoken answer · ~69 s

Window functions compute a value for each row using a set of related rows, defined by OVER with optional PARTITION BY, ORDER BY, and a frame, without collapsing rows as GROUP BY does. Examples: ROW_NUMBER, RANK, and DENSE_RANK for ranking; LAG and LEAD to access previous and next rows; running totals and moving averages with SUM or AVG over a frame; FIRST_VALUE and LAST_VALUE; and NTILE for buckets. They outperform alternatives for several reasons. Compared to correlated subqueries, which may re-execute for each row, a window function makes a single pass over the data, after one sort or hash by the partition. Compared to self-joins, such as joining a table to itself to get the previous row or a running total, which can be quadratic, windows avoid the join and read the table once. They're ideal for top N per group, using ROW_NUMBER filtered in an outer query; calculating differences between consecutive rows; percentage of total; running balances; and deduplication keeping the latest row per key.

Likely follow-ups

- How do you get the top N rows per group?
- Why can a window function beat a correlated subquery?

```
-- Rank users by order count per region – window function
```

```
SELECT user_id, region, order_count,  
       RANK() OVER (PARTITION BY region ORDER BY order_count DESC) AS rank  
FROM user_stats;
```

```
-- Running total
```

```
SELECT date, revenue,  
       SUM(revenue) OVER (ORDER BY date ROWS UNBOUNDED PRECEDING) AS cumulative  
FROM daily_revenue;
```

```
-- Lag/Lead: compare to previous row
```

```
SELECT date, revenue,  
       LAG(revenue, 1) OVER (ORDER BY date) AS prev_day_revenue  
FROM daily_revenue;
```

Window functions operate over a "window" of rows related to the current row without collapsing them (unlike GROUP BY). They are evaluated after WHERE, GROUP BY, and HAVING but before ORDER BY and LIMIT. Performance advantage: the database scans the result set once and computes the window in-memory — a self-join or correlated subquery to achieve the same result would require multiple scans. Common functions: RANK, DENSE_RANK, ROW_NUMBER, LAG, LEAD, FIRST_VALUE, LAST_VALUE, NTILE, SUM/AVG/COUNT with OVER.

Frame specification matters: ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW (running total) vs ROWS BETWEEN 6 PRECEDING AND CURRENT ROW (7-day rolling average). Getting the frame wrong produces silently incorrect results — always verify with sample data.

Official docs

- [PostgreSQL: Window functions tutorial](#)

Q10 How do you optimize a slow JOIN query — what strategies do you apply?

Query optimization

Spoken answer · ~88 s

To optimise a slow join, I start with EXPLAIN ANALYZE to see the join algorithm, row estimates, and where time goes. Strategies. Indexes: make sure the join columns are indexed on the side being looked up, especially foreign keys, and that data types and collations match on both sides so indexes are usable. Reduce the input before joining: filter early with WHERE conditions on each table, select only needed columns, and use subqueries or CTEs to pre-aggregate large tables before joining. Choose the right join method: nested loop with an index suits small outer sets; hash join suits large unsorted inputs and needs enough work_mem to avoid spilling; merge join suits inputs already sorted by the join key. Fix bad row estimates by updating statistics, increasing the statistics target, or creating extended statistics for correlated columns, since misestimates lead to the wrong algorithm or join order. Check join order and the number of tables joined; in complex queries, break them into steps. Avoid joining on expressions, and avoid unnecessary joins or DISTINCT used to hide duplication from many-to-many joins; use EXISTS for semi-joins instead of joining just to filter. Consider denormalisation, materialised views, or pre-computed aggregates for hot reporting queries. Partition large tables so that partition pruning limits the data.

Likely follow-ups

- When is a hash join better than a nested loop?
- Why can bad row estimates ruin a join?

```
-- Ensure join columns are indexed on both sides:
```

```
CREATE INDEX ON orders (user_id); -- foreign key columns often missing indexes
```

```
-- Filter early – push WHERE conditions to reduce join input size:
```

```
-- Bad: join everything then filter
```

```
SELECT * FROM orders o JOIN users u ON o.user_id = u.id WHERE o.status = 'active';
```

```
-- Better (if planner doesn't auto-push): use subquery to pre-filter
```

```
SELECT * FROM (SELECT * FROM orders WHERE status = 'active') o JOIN users u ON o.user_id = u.id;
```

```
-- Check join order: smaller result set as outer loop in nested loop join
```

```
SET join_collapse_limit = 1; -- disable PostgreSQL join reorder (for testing only)
```

Join optimization checklist: index join columns (especially FK columns — databases don't auto-create FK indexes), ensure statistics are fresh (planner needs accurate row counts for join order), avoid functions on join columns, avoid `SELECT *` (retrieve only needed columns to reduce data movement), and consider materialized CTEs for complex multi-step transformations (reused intermediate results). For very complex multi-table joins, break into steps using temp tables or CTEs with `MATERIALIZED` hint.

Join algorithm choice: Hash Join is best for large unsorted datasets. Nested Loop is best when the inner table is small or well-indexed. Merge Join requires both sides sorted. If the planner picks the wrong algorithm, stale statistics are usually the cause.

Official docs

- [PostgreSQL: Controlling the planner with explicit JOIN clauses](#)

Q11 What is VACUUM in PostgreSQL — why is it critical for performance?

PostgreSQL

Spoken answer · ~87 s

PostgreSQL uses MVCC, so an update or delete doesn't remove the old row version; it marks it dead, and readers with older snapshots may still need it. `VACUUM` reclaims space from dead tuples so it can be reused, updates the visibility map, which enables index-only scans, updates the free space map, and freezes old transaction IDs to prevent transaction ID wraparound, which would otherwise force the database to stop accepting writes to protect data. Without vacuuming, tables and indexes bloat, scans slow down, disk fills, and eventually wraparound becomes an emergency. Plain `VACUUM` runs concurrently with normal operations and makes space reusable within the table but typically doesn't shrink the file; `VACUUM FULL` rewrites the table to return space to the OS but takes an exclusive lock, so avoid it on busy tables, using `pg_repack` as an alternative. Autovacuum runs it automatically, triggered when dead tuples exceed a threshold plus a scale factor of the table size; for large, busy tables, the default scale factor of twenty percent is too lax, so tune per table with lower thresholds, higher cost limits, and enough workers. `ANALYZE`, often run along with it, updates planner statistics. Common problems are long-running transactions or abandoned replication slots, which prevent cleanup of dead rows.

Likely follow-ups

- What is transaction ID wraparound?
- Why avoid `VACUUM FULL` on a busy table?

```
-- Check autovacuum status per table:
```

```
SELECT relname, n_live_tup, n_dead_tup, last_vacuum, last_autovacuum, last_autoanalyze FROM pg_stat_user_tables ORDER BY n_dead_tup DESC LIMIT 20;
```

```
-- Manual vacuum (non-blocking):
```

```
VACUUM tablename;
```

```
-- Full vacuum (rewrites, reclaims disk – locks table):
```

```
VACUUM FULL tablename; -- never in production without planning
```

```
-- Check transaction ID wraparound risk:
```

```
SELECT datname, age(datfrozenxid) FROM pg_database ORDER BY age DESC;
```

PostgreSQL uses MVCC (Multi-Version Concurrency Control) — UPDATE and DELETE create new row versions, leaving old "dead tuples" in place. VACUUM reclaims this space for reuse. Without regular vacuuming: table bloat (dead tuples waste disk and slow sequential scans), statistics become inaccurate (planner makes bad decisions), and transaction ID wraparound risk grows (a PostgreSQL-specific catastrophic failure that forces the DB into read-only mode to prevent data corruption).

Transaction ID wraparound: PostgreSQL uses 32-bit transaction IDs. When age approaches 2 billion, PostgreSQL forces a VACUUM FREEZE. If you ignore autovacuum and this limit is exceeded, the DB shuts down in read-only mode. Monitor `age(datfrozenxid)` and alert above 1.5 billion. This has caused real production outages.

Official docs

- [PostgreSQL: Routine vacuuming](#)

Q12 How do you identify and kill long-running queries in PostgreSQL?

PostgreSQL

Spoken answer · ~83 s

In PostgreSQL, `pg_stat_activity` shows current sessions: query text, state, start times, wait events, and which backend is blocking which. To find long-running queries, filter where state is active and `now()` minus `query_start` exceeds a threshold, and also look for sessions idle in transaction, which can hold locks and block vacuum. Before killing, understand what the query is: who runs it, whether it's blocking others, found via `pg_blocking_pids` or the `pg_locks` views, and whether it's part of an important job, because killing a long write transaction triggers a rollback that may itself take a long time. The graceful action is `pg_cancel_backend(pid)`, which sends a cancel to the current query and leaves the connection open. If that doesn't work, `pg_terminate_backend(pid)` terminates the whole session. Avoid killing the postmaster or using operating system `kill -9`, which forces a crash recovery restart. Prevention is better: set `statement_timeout` to cap query runtime, `idle_in_transaction_session_timeout` to drop abandoned transactions, and `lock_timeout` to avoid long waits on locks; use role-level or application-level settings. Log slow queries with `log_min_duration_statement`, and use `auto_explain` for plans. Then investigate why it was slow, adding indexes or rewriting. In MySQL the equivalents are `SHOW PROCESSLIST` and `KILL QUERY` or `KILL connection id`.

Likely follow-ups

- What is the difference between cancel and terminate?
- Which timeouts prevent runaway queries?

```
-- Find long-running queries:
```

```
SELECT pid, now() - pg_stat_activity.query_start AS duration, query, state FROM  
pg_stat_activity WHERE state != 'idle' AND query_start < now() - interval '5 minutes' ORDER  
BY duration DESC;
```

```
-- Find blocking locks:
```

```
SELECT blocked.pid, blocking.pid AS blocking_pid, blocked.query AS blocked_query,  
blocking.query AS blocking_query FROM pg_stat_activity blocked JOIN pg_stat_activity  
blocking ON blocking.pid = ANY(pg_blocking_pids(blocked.pid)) WHERE  
cardinality(pg_blocking_pids(blocked.pid)) > 0;
```

```
-- Cancel query (graceful):
```

```
SELECT pg_cancel_backend(pid);
```

```
-- Kill connection (immediate):
```

```
SELECT pg_terminate_backend(pid);
```

Long-running queries hold locks that block other queries. An idle transaction (`BEGIN` without `COMMIT/ROLLBACK`) is especially dangerous — it holds locks indefinitely even though the query is done. Set `idle_in_transaction_session_timeout = '5min'` to auto-terminate idle-in-transaction sessions. Set `statement_timeout` per role to prevent runaway queries. Always investigate before terminating — a long-running `VACUUM ANALYZE` is legitimate.

Lock hierarchy: `ALTER TABLE` takes an `AccessExclusiveLock` — it blocks all reads and writes. Plan schema migrations carefully: use tools like `pg_repack` or `pgroll` for zero-downtime migrations that avoid heavy locking. Never run `ALTER TABLE` in production during peak hours.

Official docs

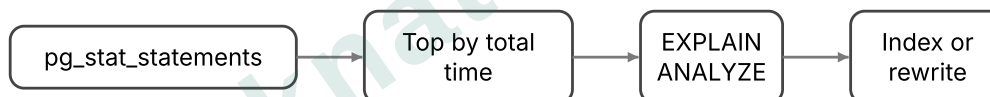
- [PostgreSQL: pg_stat statements](#)

Spoken answer · ~85 s

pg_stat_statements is a PostgreSQL extension that tracks execution statistics for every distinct normalised query, with constants replaced by parameters: number of calls, total, mean, min, max and standard deviation of execution time, rows returned, shared buffer hits and reads, temporary file usage, WAL generated, and more. Enable it by adding it to shared_preload_libraries, restarting, and running CREATE EXTENSION pg_stat_statements. To find the worst queries, sort in different ways, because the worst depends on the question. By total_exec_time to find where the database spends most time overall, usually the best starting point, since a moderately slow query called millions of times matters more than a rare slow one. By mean_exec_time for individually slow queries. By calls to spot chatty patterns like N+1. By rows for large result sets. By shared_blks_read for I/O-heavy queries, and by temp_blks_written for queries spilling to disk. Also look at stddev for inconsistent performance. Then take the top candidates, run EXPLAIN ANALYZE with realistic parameters, and optimise. Reset the statistics with pg_stat_statements_reset after a change to measure the impact, or take periodic snapshots to track trends. Combine with the slow query log, and use dashboards from tools like pgwatch or monitoring agents. Watch the query text length and the memory overhead.

Likely follow-ups

- Why sort by total time rather than mean time?
- How do you measure the impact of a fix?

🔗 Finding slow queries

```
-- Enable (requires postgresql.conf restart):
```

```
shared_preload_libraries = 'pg_stat_statements'
```

```
CREATE EXTENSION IF NOT EXISTS pg_stat_statements;
```

```
-- Top 10 queries by total time:
```

```
SELECT query, calls, total_exec_time/1000 AS total_sec, mean_exec_time AS avg_ms, rows FROM  
pg_stat_statements ORDER BY total_exec_time DESC LIMIT 10;
```

```
-- Top 10 by average latency (worst per-call performance):
```

```
SELECT query, calls, mean_exec_time AS avg_ms FROM pg_stat_statements WHERE calls > 100  
ORDER BY mean_exec_time DESC LIMIT 10;
```

```
-- Reset stats:
```

```
SELECT pg_stat_statements_reset();
```

`pg_stat_statements` normalizes queries (replaces literals with \$1, \$2 placeholders) and tracks total execution time, call count, rows, shared blocks hit/read, and I/O timing. Use it to find: the query with highest total time (highest impact to optimize), queries called millions of times with even small latency (N+1 detection), and queries with low cache hit rate (high `blks_read` vs `blks_hit` — missing from `shared_buffers`). This is the first tool to reach for during a performance investigation.

Production standard: Always have `pg_stat_statements` enabled. The overhead is minimal (<5% CPU) and the visibility is invaluable. Cloud SQL, RDS, and Cloud Spanner all have equivalent query insights dashboards built on the same principle.



Official docs

- [PostgreSQL: pg_stat_statements](#)

Q14 How do you tune PostgreSQL memory settings — what are the key parameters?

PostgreSQL

Spoken answer · ~83 s

PostgreSQL memory tuning centres on a few parameters. `shared_buffers` is PostgreSQL's own cache of data pages; a common starting point is around twenty-five percent of RAM, as the operating system's page cache also helps, with more only if testing shows benefit. `work_mem` is the memory per sort or hash operation, per query node, per connection, so total usage can be many multiples; raise it carefully, globally to modest values, or per session or role for heavy reports, since too high a value across many connections causes out-of-memory kills, while too low causes spills to disk, visible in EXPLAIN as external sorts or temp files. `maintenance_work_mem` is used for VACUUM, CREATE INDEX, and ALTER TABLE, and can be much higher, a few hundred megabytes to gigabytes. `effective_cache_size` is not an allocation but a hint to the planner about how much memory is available for caching in total, including the OS cache, typically fifty to seventy-five percent of RAM, influencing index versus sequential scan decisions. Others: `wal_buffers`, `huge_pages` to reduce page table overhead, and `max_connections`, which multiplies per-connection memory, so use a pooler instead of raising it. Also keep an eye on overcommit settings in Linux to avoid the OOM killer.

Likely follow-ups

- Why is `work_mem` dangerous to raise globally?
- What does `effective_cache_size` actually do?

```
shared_buffers = 25% of RAM          # PostgreSQL's own page cache

effective_cache_size = 75% of RAM     # hints to planner about OS cache

work_mem = 4MB–256MB                 # per-sort / per-hash operation

maintenance_work_mem = 256MB–2GB     # for VACUUM, CREATE INDEX, ALTER TABLE

wal_buffers = 64MB                   # WAL write buffer (auto-tuned in PG 12+)

-- Check buffer hit rate (should be > 99%):
```

```
SELECT sum(blks_hit)*100.0 / (sum(blks_hit)+sum(blks_read)) AS cache_hit_rate FROM
pg_stat_database;
```

work_mem is the most commonly misunderstood parameter. It's allocated per sort/hash operation per query — a complex query with multiple sort nodes and 100 parallel connections can use work_mem × nodes × connections of RAM simultaneously. Setting it too high causes OOM; too low causes disk sorts (look for "Disk: true" in EXPLAIN output). Set a low default and use SET work_mem = '256MB' per session for analytics queries. shared_buffers should be 25% of RAM for dedicated PostgreSQL hosts.

work_mem × connections math: 100 connections × 10 sort nodes × 256MB work_mem = 256GB potential peak RAM. Always calculate the worst-case memory footprint before setting work_mem globally high.



Official docs

- [PostgreSQL: Performance tips](#)
- [PostgreSQL: Resource consumption settings](#)

Q15 How does the InnoDB buffer pool work — how do you tune it?

MySQL

Spoken answer · ~89 s

The InnoDB buffer pool is MySQL's main memory cache for table data and index pages. Reads check the pool first; a miss reads the page from disk into the pool. Writes modify pages in memory, marking them dirty, and they're flushed to disk later in the background, while durability comes from the redo log. The pool is managed with a modified LRU list, divided into a young sublist and an old sublist, so a large table scan doesn't evict the hot working set: new pages enter at the midpoint. Tuning. innodb_buffer_pool_size is the most important setting: on a dedicated database server, typically sixty to eighty percent of RAM, sized to hold the working set, leaving room for the operating system, connections, and other buffers. Check effectiveness with the buffer pool hit ratio from Innodb_buffer_pool_reads versus read requests, though what matters more is whether disk reads hurt latency. Use multiple buffer pool instances for concurrency on large pools. Resize online in recent versions in chunks. Enable the dump and load at shutdown and startup, so the cache is warm after a restart. Tune flushing with innodb_io_capacity to match the storage's IOPS, and the redo log size for write-heavy workloads, as a small log forces frequent flushing. Monitor free pages, dirty page percentage, and evictions.

Likely follow-ups

- Why does the buffer pool use a midpoint insertion?
- How big should the buffer pool be?

```
-- Check buffer pool hit rate (should be > 99%):
```

```
SHOW GLOBAL STATUS LIKE 'Innodb_buffer_pool_%';
```

```
-- Calculate hit rate:
```

```
SELECT (1 - (Innodb_buffer_pool_reads / Innodb_buffer_pool_read_requests)) * 100 AS
hit_rate;
```

```
-- Set buffer pool size (my.cnf):
```

```
innodb_buffer_pool_size = 70-80% of RAM # for dedicated MySQL server
```

```
innodb_buffer_pool_instances = 8 # reduce mutex contention (1 per GB)
```

```
-- Warm up buffer pool after restart:
```

```
innodb_buffer_pool_dump_at_shutdown = ON
```

```
innodb_buffer_pool_load_at_startup = ON
```

InnoDB buffer pool is MySQL's combined page cache and index cache — it's the most impactful single parameter for MySQL performance. It caches both data pages and index pages. A warm buffer pool serves reads from memory; a cold pool (after restart) has high I/O until pages are loaded. Enable dump/load to persist the "hot set" across restarts, dramatically reducing the post-restart performance dip.

Multiple instances: With a large buffer pool (> 8GB), split into 8 instances (`innodb_buffer_pool_instances=8`). Each instance has its own mutex, reducing lock contention for concurrent access. Without this, a single mutex on a 64GB buffer pool becomes a bottleneck.

Official docs

- [MySQL: InnoDB buffer pool](#)

Q16 What is the MySQL slow query log — how do you use it and what do you look for? MySQL

Spoken answer · ~87 s

The MySQL slow query log records statements that take longer than `long_query_time` seconds, optionally also those not using indexes, with details such as query time, lock time, rows sent, and rows examined. Enable it with `slow_query_log` on, set the threshold, for instance one second at first and lower later, such as 100 milliseconds, set the log file or table output, and optionally `log_queries_not_using_indexes` with a rate limit to avoid noise. In production, enable it carefully and rotate logs. To analyse, don't read it by hand; aggregate with `pt-query-digest` from Percona Toolkit, or `mysqldumpslow`, which groups similar queries and ranks them by total time, calls, and average. What I look for: queries with a high total time, which is the best optimisation target; a large ratio of rows examined to rows sent, indicating missing or ineffective indexes, since the query scans many rows to return few; high lock time, suggesting contention; queries creating temporary tables on disk or sorting with `filesort`; and sudden new queries after a deployment. Next, run `EXPLAIN` on the top offenders, add or adjust indexes, rewrite queries, and measure again. Performance Schema and `sys` schema views provide similar information with less overhead. Review the log regularly, not only during incidents, and combine it with application tracing.

Likely follow-ups

- What does a high rows examined to rows sent ratio mean?
- Why aggregate with `pt-query-digest`?

```
-- Enable slow query log:
```

```
slow_query_log = ON
```

```
slow_query_log_file = /var/log/mysql/slow.log
```

```
long_query_time = 1          # log queries > 1 second
```

```
log_queries_not_using_indexes = ON # catch full scans regardless of time
```

```
-- Analyze with mysqldumpslow:
```

```
mysqldumpslow -s t -t 10 /var/log/mysql/slow.log # top 10 by total time
```

```
mysqldumpslow -s c -t 10 /var/log/mysql/slow.log # top 10 by count
```

```
-- Or with pt-query-digest (more detailed):
```

```
pt-query-digest /var/log/mysql/slow.log | head -100
```

The slow query log captures queries exceeding `long_query_time`. Enable `log_queries_not_using_indexes` to catch full scans even on fast small tables — they will degrade as data grows. Use `pt-query-digest` from Percona Toolkit for structured analysis: fingerprinted query patterns, sorted by total execution time, with response time distribution. This is the equivalent of PostgreSQL's `pg_stat_statements` but file-based.

Performance Schema: MySQL 5.6+ has Performance Schema — enables `events_statements_summary_by_digest` for real-time query analytics without a log file. Use this in production for live query profiling without the disk I/O overhead of the slow log.

Official docs

- [PostgreSQL: Using EXPLAIN](#)
- [PostgreSQL: pg_stat_statements](#)
- [MySQL: Optimization](#)

Spoken answer · ~88 s

MySQL replication lag is the delay between a transaction committing on the primary and being applied on a replica, which causes stale reads. Detect it with `SHOW REPLICA STATUS`, looking at `Seconds_Behind_Source`, though this metric can be misleading, for instance showing zero when the I/O thread is stalled; better to use a heartbeat table, as `pt-heartbeat` does, which writes timestamps on the primary and compares on the replica for accurate measurement. Compare the I/O thread, which fetches binary logs, and the SQL thread, which applies them, to see where it's lagging. Causes: single-threaded apply of a high write volume, since historically the replica applied transactions serially; large transactions or bulk updates and deletes that take a long time to apply; weaker replica hardware or disk I/O contention from heavy reads; missing indexes on the replica for row-based updates, leading to full table scans per row; long-running queries or locks on the replica; network latency; and DDL operations. Reduce lag: enable multi-threaded replication with `replica_parallel_workers` and logical-clock parallelism; break large batch operations into small chunks; ensure primary keys exist on all tables; use faster storage and adequate hardware; tune durability settings on replicas carefully; move heavy reporting elsewhere; and route latency-sensitive reads to the primary or use semi-synchronous replication. Alert on lag thresholds.

Likely follow-ups

- Why is `Seconds_Behind_Source` sometimes misleading?
- How does a missing primary key worsen lag?

```
-- Check replication lag on replica:
```

```
SHOW SLAVE STATUSG -- "Seconds_Behind_Master" field
```

```
SHOW REPLICA STATUSG -- MySQL 8.0+ syntax
```

```
-- Enable parallel replication (MySQL 5.7+):
```

```
replica_parallel_workers = 8      # number of parallel apply threads
```

```
replica_parallel_type = LOGICAL_CLOCK # dependency-based parallelism
```

```
-- Reduce binlog event size:
```

```
binlog_row_image = MINIMAL # only log changed columns, not full row
```

Replication lag sources: single-threaded replica applying multi-threaded primary writes (solve with parallel replication), long-running transactions on primary blocking replica, large bulk operations, or network latency between primary and replica. Monitor `Seconds_Behind_Master` but also compare GTID positions for more accurate lag measurement. In cloud: Cloud SQL and RDS replicas have built-in parallel apply.

Reading stale data: If your application reads from a replica, it may read data that hasn't been replicated yet. Implement read-your-writes consistency by routing writes AND the subsequent reads for the same user session to the primary (or wait for replica to catch up using `WAIT_FOR_EXECUTED_GTID_SET`).

Official docs

- [PostgreSQL: High availability, load balancing, replication](#)
- [MySQL: Replication](#)

Q18 What is connection pooling — why is it essential for database performance?

Connections

Spoken answer · ~85 s

Connection pooling maintains a set of reusable database connections that applications borrow and return, instead of opening a new connection for each request. It's essential because connections are expensive: establishing one involves a TCP and TLS handshake and authentication, and in PostgreSQL each connection is a separate process using several megabytes of memory; maintaining thousands of them harms performance through memory use, context switching, and lock contention. Databases also have a hard `max_connections` limit. A pool bounds the number of connections, so the database operates at a healthy concurrency level, typically a small multiple of CPU cores, while many more application requests queue for a connection. It reduces latency by eliminating connection setup. Types: in-application pools, such as HikariCP, which is per process; and external poolers, like PgBouncer or ProxySQL, or RDS Proxy and Cloud SQL managed pooling, which multiplex many client connections onto fewer server connections and matter when there are many application instances, as in Kubernetes or serverless. PgBouncer modes: session, transaction, and statement; transaction mode gives the best multiplexing but breaks session-level features such as prepared statements in some setups, advisory locks, and temporary tables. Size pools carefully: bigger isn't better, set timeouts, validate connections, and monitor pool wait times and usage.

Likely follow-ups

- Why doesn't a bigger pool mean better performance?
- What breaks in PgBouncer transaction mode?

```
-- PostgreSQL: each connection is a forked process (~5-10MB RAM each)
```

```
-- 1000 direct connections = 5-10GB RAM overhead on the DB server
```

```
-- PgBouncer config (pgbouncer.ini):
```

```
pool_mode = transaction # most efficient – connection returned after each transaction
```

```
max_client_conn = 10000 # clients can connect to PgBouncer
```

```
default_pool_size = 25 # actual connections to PostgreSQL
```

```
-- Check PgBouncer stats:
```

```
psql -p 6432 pgbouncer -c "SHOW POOLS;"
```

```
psql -p 6432 pgbouncer -c "SHOW STATS;"
```

PostgreSQL opens a new OS process per connection — expensive at scale. A 500-connection PostgreSQL instance with 8 cores is wasting 492 connections waiting (CPU can only run 8 at once). PgBouncer multiplexes thousands of application connections onto a small pool of actual DB connections. Pool modes: session (connection held for full session), transaction (returned after each transaction — most efficient), statement (returned after each statement — incompatible with multi-statement transactions).

Cloud SQL / RDS: Cloud SQL has built-in connection pooling via the Auth Proxy. For high-concurrency apps, still use PgBouncer in front — Cloud SQL's `max_connections` is often the bottleneck, not the Auth Proxy. Run PgBouncer as a sidecar in Kubernetes or as a dedicated service.

Official docs

- [PgBouncer](#)

Q19 Database is running out of connections — how do you diagnose and fix the connection exhaustion?

Connections

Spoken answer · ~85 s

When a database runs out of connections, applications see errors like too many connections. First stabilise: identify who holds the connections. In PostgreSQL, query `pg_stat_activity` grouped by application, user, client address, and state; in MySQL, `SHOW PROCESSLIST`. Look for lots of idle connections, which suggests oversized pools or leaks; idle in transaction sessions, where the application opened a transaction and never finished; many active sessions all waiting on a lock or a slow query, causing requests to pile up and take more connections; or a sudden traffic spike or retry storm. Immediate relief: terminate idle-in-transaction or runaway sessions, kill the blocking query, restart or scale down the misbehaving application, and temporarily raise the limit if memory allows, though that's only a stopgap. Root causes and fixes: connection leaks in code, fixed by always releasing connections, using context managers, and setting max lifetime and leak detection; oversized pools times many replicas; missing pooling, where each request opens a connection; slow queries holding connections longer; and lack of timeouts. Long-term: add an external pooler like PgBouncer, right-size pools so total across instances stays below the limit, set `idle_in_transaction_session_timeout` and `statement_timeout`, add retry with backoff and circuit breakers, and alert on connection usage percentage before exhaustion.

Likely follow-ups

- How do you find leaked connections?
- What is the danger of retry storms?

Connection exhaustion



```
-- PostgreSQL: check current connections
```

```
SELECT count(*), state, wait_event_type, wait_event FROM pg_stat_activity GROUP BY state, wait_event_type, wait_event ORDER BY count DESC;
```

```
SELECT setting FROM pg_settings WHERE name = 'max_connections';
```

```
-- Find connections by application:
```

```
SELECT application_name, count(*) FROM pg_stat_activity GROUP BY application_name ORDER BY count DESC;
```

```
-- MySQL:
```

```
SHOW STATUS LIKE 'Threads_connected';
```

```
SHOW PROCESSLIST; -- active queries + sleeping connections
```

Root causes: application not returning connections to pool (connection leak — missing finally/close() blocks), pool size set too large (each app instance claims N connections, $N \times \text{pods}$ exceeds max_connections), connection pool not configured (every request opens a new connection), or idle connections not timed out (sleeping connections accumulating). Fix immediately: restart the leaking service, reduce pool size, or increase max_connections temporarily. Fix permanently: add connection pooling layer (PgBouncer), fix the connection leak in code, set idle connection timeouts.

Kubernetes scale-out: When HPA scales from 5 to 50 pods, a pool of 20 connections per pod = 1000 connections to PostgreSQL. Plan for $\text{max_pod_replicas} \times \text{pool_size}$ when sizing PostgreSQL max_connections. PgBouncer decouples this — all 50 pods connect to PgBouncer's 1000-connection capacity, which maps to 25 real DB connections.

Official docs

- [PgBouncer](#)

Spoken answer · ~88 s

A deadlock happens when two or more transactions each hold a lock the other needs, so none can proceed. For example, transaction A locks row 1 then wants row 2, while transaction B locked row 2 and wants row 1. The database detects the cycle, using a wait-for graph, picks a victim, aborts it with an error such as PostgreSQL's deadlock detected or MySQL's error 1213, and the other continues. Detection: check the database logs, where PostgreSQL logs the involved queries after `deadlock_timeout`, and in MySQL use `SHOW ENGINE INNODB STATUS` to see the latest deadlock, or enable `innodb_print_all_deadlocks`. Monitor deadlock counts in metrics. Prevention: access resources in a consistent order across all code paths, for example always update rows sorted by ID; keep transactions short and avoid user interaction or network calls inside them; lock only what's needed, acquiring locks early with `SELECT FOR UPDATE` where appropriate, possibly with `NOWAIT` or `SKIP LOCKED` for queues; use appropriate isolation levels; add indexes so updates lock fewer rows, since without indexes InnoDB may lock many rows or gaps; and avoid large batch updates in one transaction by chunking. Recovery: applications must catch the deadlock error and retry the transaction, with backoff, since the aborted transaction rolled back completely. Make operations idempotent.

Likely follow-ups

- Why must the application retry after a deadlock?
- How does consistent lock ordering prevent deadlocks?

```
-- PostgreSQL deadlock detection (automatic, logged):
```

```
tail -f /var/log/postgresql/postgresql.log | grep deadlock
```

```
-- Or check pg_locks for current lock waits:
```

```
SELECT l.pid, l.locktype, l.relation::regclass, l.mode, l.granted FROM pg_locks l WHERE NOT l.granted;
```

```
-- MySQL: show last deadlock
```

```
SHOW ENGINE INNODB STATUSG -- LATEST DETECTED DEADLOCK section
```

```
-- Prevention: always acquire locks in the same order
```

```
-- Transaction 1: lock table A, then table B
```

```
-- Transaction 2: lock table A, then table B (not B then A)
```

A deadlock occurs when two transactions each hold a lock the other needs — they wait forever. PostgreSQL detects deadlocks automatically and kills one transaction (the victim) with an error. MySQL InnoDB does the same. Both log the deadlock details. Prevention: acquire locks in a consistent global order across all transactions, keep transactions short (reduce lock hold time), and use `SELECT FOR UPDATE` to explicitly lock rows in the order you'll update them.

Application handling: Deadlock victims receive an error (error code 40P01 in PostgreSQL). Application code MUST handle this with retry logic — deadlocks are normal under high concurrency. A deadlock-prone query that isn't retried will surface as a user-visible error under load.

Official docs

- [PostgreSQL: Explicit locking](#)

Q21 What is WAL in PostgreSQL — how does WAL enable replication and point-in-time recovery?

Replication & HA

Spoken answer · ~88 s

The Write-Ahead Log is PostgreSQL's mechanism for durability and recovery: before any change to data pages is written to disk, a record describing the change is appended and flushed to the WAL. On commit, only the WAL needs to be flushed, which is sequential and fast, while the dirty data pages are written later at checkpoints. After a crash, PostgreSQL replays the WAL from the last checkpoint to restore consistency. The WAL also underpins replication: in streaming replication, a standby connects to the primary and receives WAL records in real time, then replays them to stay in sync, which is physical replication of the byte-level changes. Replication slots make sure the primary keeps WAL until the standby has consumed it, though an abandoned slot makes WAL accumulate and can fill the disk. For point-in-time recovery, you take a base backup, a consistent copy of the data directory, and continuously archive WAL segments to durable storage with `archive_command` or tools like `pgBackRest` and `WAL-G`. To restore, you copy back the base backup, then replay archived WAL up to a chosen target time, transaction ID, or named restore point, recovering the database to a moment just before an accidental delete or bad deploy. `wal_level` controls the detail, and logical replication needs `wal_level` logical.

Likely follow-ups

- Why can an abandoned replication slot fill the disk?
- What do you need for point-in-time recovery?

```
-- WAL (Write-Ahead Log): all changes written to WAL before modifying data pages
```

```
-- Check WAL directory size:
```

```
SELECT pg_size_pretty(sum(size)) FROM pg_ls_waldir();
```

```
-- WAL archiving (for PITR):
```

```
archive_mode = on
```

```
archive_command = 'gsutil cp %p gs://my-wal-archive/%f' # copy WAL to GCS
```

```
-- Streaming replication settings:
```

```
wal_level = replica          # minimum for replication
```

```
max_wal_senders = 10      # max replica connections
```

```
wal_keep_size = 1GB      # keep WAL on primary for slow replicas
```

WAL ensures durability (data is never lost if fsync is on) and enables both streaming replication (replica receives WAL stream continuously) and PITR (replay archived WAL segments to any point in time). Replicas in streaming replication continuously apply WAL — they are always slightly behind the primary (replication lag). If a replica falls too far behind and the primary discards WAL it hasn't received, the replica must be rebuilt from a base backup.

PITR for Cloud SQL: Cloud SQL automatically archives WAL to GCS. Enable PITR in Cloud SQL config — it retains 7 days of WAL by default. For compliance workloads, this is your RTO/RPO safety net: restore to any second within the retention window.

Official docs

- [PostgreSQL: Write-Ahead Logging](#)
- [PostgreSQL: High availability, load balancing, replication](#)
- [PostgreSQL: Continuous archiving and PITR](#)

Q22 **What is the difference between synchronous and asynchronous replication — when do you choose each?** Replication & HA

Spoken answer · ~89 s

In synchronous replication, the primary waits for one or more replicas to confirm they have received, or applied, the transaction before acknowledging the commit to the client. That gives zero data loss on primary failure, an RPO of zero, because committed data exists on at least two nodes, at the cost of higher commit latency, bounded by the replica round trip, and an availability risk: if the synchronous replica goes down or becomes unreachable, writes stall unless you configure fallback, such as quorum commit with multiple candidates. In asynchronous replication, the primary commits locally and sends changes to replicas afterwards, without waiting. It offers the lowest write latency, no dependence on replica health, and works well across regions, but replicas lag, so reads from them can be stale, and if the primary fails, recent transactions that hadn't been shipped can be lost. Choose synchronous for financial or critical data where losing committed transactions is unacceptable, and where replicas are close, in the same region or nearby zones. Choose asynchronous for read scaling, cross-region disaster recovery, and workloads where small potential data loss is tolerable and latency matters. Intermediate options: semi-synchronous replication in MySQL, PostgreSQL's `remote_write` versus `remote_apply` levels, per-transaction `synchronous_commit` settings, and quorum-based configurations. Align the choice with the business RPO and RTO.

Likely follow-ups

- What is the availability risk of synchronous replication?
- How can you make durability a per-transaction choice?

Replication

Synchronous	Asynchronous
Waits for replica	Returns immediately
No loss, slower writes	Lag, possible loss

```
-- PostgreSQL: configure synchronous replication
```

```
synchronous_standby_names = 'replica1' # in postgresql.conf
```

```
-- Per transaction override:
```

```
SET synchronous_commit = local; # don't wait for replica (lower durability)
```

```
SET synchronous_commit = remote_write; # wait until replica has WAL in OS buffer
```

```
SET synchronous_commit = on; # wait until replica has WAL fsynced (full sync)
```

```
-- Check replication lag in bytes:
```

```
SELECT client_addr, state, sent_lsn, write_lsn, flush_lsn, replay_lsn, (sent_lsn -  
replay_lsn) AS lag_bytes FROM pg_stat_replication;
```

Async replication: primary commits as soon as WAL is locally fsynced. Replica applies WAL asynchronously. Pros: zero write latency overhead. Cons: data loss risk on primary failure (replica may be seconds behind). Sync replication: primary waits for replica to acknowledge WAL before returning to client. Pros: zero data loss (RPO=0). Cons: every write waits for a network round-trip to the replica — adds latency. Use sync for financial transactions and compliance workloads; async for high-throughput workloads that can tolerate seconds of potential data loss.

Quorum commit: PostgreSQL supports ANY 1 (replica1, replica2) — commit acknowledged when any 1 of 2 replicas confirm. This combines sync durability with high availability — if one replica goes down, the other still satisfies the quorum.

Official docs

- [PostgreSQL: High availability, load balancing, replication](#)

Spoken answer · ~88 s

A lagging read replica means it's applying changes slower than the primary generates them. Start by measuring what kind of lag it is: receive lag, where WAL or binlog isn't arriving fast enough, versus replay lag, where it's arriving but not applied quickly. In PostgreSQL, compare `pg_stat_replication` sent, write, flush, and replay positions on the primary; in MySQL, compare I/O and SQL thread status. Causes: write bursts or large batch jobs and big transactions that take long to replay; a single-threaded replay unable to keep up with highly concurrent writes; replica hardware or storage weaker than the primary; heavy read queries on the replica, which in PostgreSQL can conflict with replay, causing it to wait or cancel queries, controlled by `max_standby_streaming_delay` and `hot_standby_feedback`; missing indexes or primary keys on the replica; network bandwidth or latency, especially cross-region; vacuum or checkpoint activity; and long locks. Fixes: provision replicas with equal or better resources and fast disks; enable parallel apply where supported; break large writes into smaller batches; move heavy analytics to a dedicated replica; add the right indexes; tune the conflict-handling settings knowingly; improve network paths and compression; and add more replicas to spread read load. For consistency, route reads that need fresh data to the primary, or use read-your-writes techniques.

Likely follow-ups

- How can replica queries conflict with replay in PostgreSQL?
- How do you handle reads needing fresh data?

```
-- PostgreSQL: check replication lag
```

```
SELECT now() - pg_last_xact_replay_timestamp() AS replication_lag;
```

```
-- Check if replica is in recovery:
```

```
SELECT pg_is_in_recovery();
```

```
-- Check for replication slot bloat (replica not consuming WAL):
```

```
SELECT slot_name, active, pg_size_pretty(pg_wal_lsn_diff(pg_current_wal_lsn(),
restart_lsn)) AS lag FROM pg_replication_slots;
```

```
-- MySQL: check replica threads
```

```
SHOW REPLICA STATUSG -- check SQL_Delay, Seconds_Behind_Source
```

Lag causes: single-threaded WAL apply unable to keep up with parallel primary writes (enable parallel apply), long-running transactions on replica blocking WAL application, heavy read queries on the replica consuming I/O needed for WAL apply (dedicated replica for analytics vs serving reads), or replication slot not being consumed (slot accumulates WAL — can fill disk on primary). A replication slot with a lagging consumer is a disk bomb — always set `max_slot_wal_keep_size` as a safety limit.

Replication slot danger: An inactive replication slot causes the primary to retain ALL WAL since the slot's last confirmed LSN. If a logical replication consumer (like Debezium CDC) goes down for hours, the primary's `/pg_wal` directory can fill the disk, causing a

primary outage. Monitor slot lag closely.

Official docs

- [Amazon RDS: Read replicas](#)
- [PostgreSQL: High availability, load balancing, replication](#)

Q24 How does schema design impact database performance — what are the key design decisions?

Schema design

Spoken answer · ~88 s

Schema design determines how much data the database must touch, how efficiently it can use indexes, and how well it scales. Key decisions: normalisation level, which reduces redundancy and update anomalies but needs joins, versus selective denormalisation for read-heavy paths. Data types: choose the smallest appropriate types, since smaller rows mean more rows per page, better cache use, and smaller indexes; use proper types like integers for IDs, timestamps instead of strings, and numeric versus floating point for money; avoid oversized text columns in hot tables, and be careful with JSON blobs that can't be indexed or constrained as easily. Primary keys: sequential integers or time-ordered IDs keep inserts efficient and indexes compact, while random UUIDs cause page splits and fragmentation, particularly in InnoDB where the primary key is the clustered index. Indexes aligned to access patterns, with foreign keys indexed. Constraints, such as NOT NULL, unique, and foreign keys, which help the optimiser and integrity. Partitioning for large time-based or tenant-based tables, aiding pruning and retention. Avoiding wide tables with many rarely used columns, and moving large or infrequently used data elsewhere. Handling of NULLs, and soft deletes that bloat tables and complicate indexes. Design for the query workload, not just the entities; consider read versus write ratios, growth, and tenancy.

Likely follow-ups

- Why do random UUID primary keys hurt InnoDB?
- How do data types affect performance?

```
-- Choose appropriate data types (smaller = faster):
```

```
INT (4 bytes) vs BIGINT (8 bytes) – use INT until you need > 2 billion rows
```

```
VARCHAR(255) vs TEXT – TEXT is fine in PostgreSQL (same storage), avoid in MySQL for indexing
```

```
TIMESTAMPTZ vs TIMESTAMP – always TIMESTAMPTZ in PostgreSQL (stores in UTC)
```

```
UUID vs BIGINT – UUID as PK causes index fragmentation (random inserts); use BIGSERIAL or ULIDs
```

```
-- Normalization vs denormalization trade-off:
```

```
3NF: minimize redundancy, maximize consistency (OLTP workloads)
```

Denormalized: pre-compute joins for read performance (OLAP / reporting)

Key decisions: primary key type (UUID v4 causes B-tree fragmentation on insert-heavy tables — use sequential UUIDs/ULIDs/BIGSERIAL), column ordering (PostgreSQL aligns types — grouping 8-byte columns together reduces padding/wasted space), NOT NULL constraints (NULLable columns require extra storage bits and complicate query logic), and partitioning (range/list/hash partitioning for large tables improves query performance and maintenance operations like archival).

ULID over UUID: ULID (Universally Unique Lexicographically Sortable Identifier) is time-ordered and random-suffix — it's globally unique like UUID but sequential like BIGSERIAL. New inserts always go to the rightmost B-tree leaf page instead of randomly scattered across the tree. Dramatically less index fragmentation and better cache locality for recent-data queries.



Official docs

- [PostgreSQL: Database design basics \(normalization\)](#)

Q25

What is table partitioning — when should you use it and what are the pitfalls?

Schema design

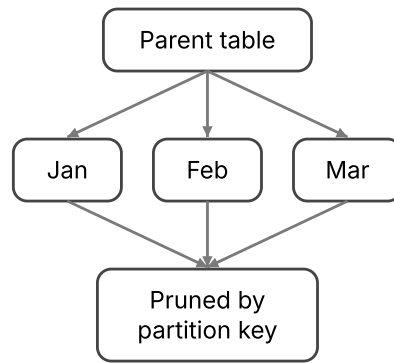
Spoken answer · ~86 s

Table partitioning splits one large logical table into smaller physical pieces, called partitions, by a key. Methods: range, commonly by date; list, by discrete values such as region or tenant; and hash, to spread rows evenly. In PostgreSQL, declarative partitioning handles this natively. Benefits: partition pruning, where queries filtering on the partition key scan only relevant partitions; cheap data retention, since dropping or detaching an old partition is instant compared with a huge DELETE that bloats the table and needs vacuuming; smaller indexes per partition that fit memory better; faster maintenance, such as vacuum and reindex per partition; and the ability to put partitions on different storage. Use it for very large tables, hundreds of gigabytes or more, time-series or log-like data with retention policies, and multi-tenant data. Pitfalls: queries that don't filter on the partition key scan all partitions and can be slower than a single table; primary keys and unique constraints must include the partition key; too many partitions increase planning time and memory use; foreign keys to and from partitioned tables have restrictions; a skewed key creates hot partitions; creating future partitions must be automated, using `pg_partman` or scheduled jobs, otherwise inserts fail; and migrating an existing table to partitioned form needs planning.

Likely follow-ups

- Why must unique keys include the partition key?
- How does partitioning make retention cheap?

🚧 Partitioning



```
-- PostgreSQL declarative partitioning:
```

```
CREATE TABLE events (id BIGINT, created_at TIMESTAMPTZ, data JSONB) PARTITION BY RANGE (created_at);
```

```
CREATE TABLE events_2024_q1 PARTITION OF events FOR VALUES FROM ('2024-01-01') TO ('2024-04-01');
```

```
CREATE TABLE events_2024_q2 PARTITION OF events FOR VALUES FROM ('2024-04-01') TO ('2024-07-01');
```

```
-- Partition pruning: check if query prunes partitions
```

```
EXPLAIN SELECT * FROM events WHERE created_at > '2024-06-01'; -- should show only q2+
```

```
-- MySQL: check partition info
```

```
SELECT PARTITION_NAME, TABLE_ROWS FROM information_schema.PARTITIONS WHERE TABLE_NAME = 'events';
```

Partitioning benefits: partition pruning (query only scans relevant partitions), faster archival (DROP old partition instead of DELETE millions of rows — near-instant), parallel query across partitions, and index sizes per partition stay manageable. Use for: time-series data (partition by month/quarter), multi-tenant data (partition by tenant_id), or any table growing beyond 50-100M rows. Pitfall: queries without a partition key in WHERE clause scan ALL partitions — worse than an unpartitioned table if the key is missing.

Partition key in every query: A partitioned table is only beneficial if your hot queries filter on the partition key. If your most common query does `WHERE user_id = ?` but the table is partitioned by `created_at`, you get no pruning — and the overhead of partition routing adds cost. Partition key must match your query patterns.

📖 Official docs

- [PostgreSQL: Table partitioning](#)

Spoken answer · ~85 s

Normalisation organises data so each fact is stored once, removing redundancy and update anomalies, typically up to third normal form: every non-key column depends on the key, the whole key, and nothing but the key. Benefits: data integrity, smaller storage, easier writes, since you update one place. The cost is that reads need joins across tables. Denormalisation deliberately duplicates or precomputes data to make reads faster and simpler, such as storing a customer name on an order, keeping counters, or flattening tables for reporting. Benefits: fewer joins, lower read latency; costs: duplicated data to keep consistent, more complex writes, and risk of inconsistency. In practice, I start normalised, because it's correct and flexible, and measure. Denormalise only where profiling shows a join or aggregation is a real bottleneck, and where reads greatly outnumber writes. Good techniques include materialised views, summary or rollup tables updated by triggers or jobs, cached columns such as order totals, and read models separate from the write model, as in CQRS. Analytics systems often use star schemas, which are intentionally denormalised. Keep the source of truth normalised, and make the duplicate data derivable and rebuildable, with the update logic in one place, such as in triggers or a defined pipeline.

Likely follow-ups

- How do you keep denormalised data consistent?
- When is a star schema appropriate?

```
-- Normalized (3NF): separate tables, joins at query time
```

```
users(id, name, email) + orders(id, user_id, total) → JOIN on user_id
```

```
-- Denormalized: store user_name in orders to avoid join
```

```
orders(id, user_id, user_name, total) → no join needed
```

```
-- Hybrid: JSONB column for flexible attributes without schema changes
```

```
ALTER TABLE products ADD COLUMN attributes JSONB;
```

```
CREATE INDEX idx_attributes_gin ON products USING GIN (attributes); -- queryable
```

Normalized (3NF): eliminates data redundancy, simplifies writes (update one place), enforces consistency. Good for OLTP where writes are frequent. Denormalized: precomputes data, eliminates join overhead, simplifies read queries. Good for read-heavy analytics and reporting. In practice: start normalized, denormalize only where profiling shows join cost is a bottleneck. Materialized views are a middle ground — normalized source, precomputed denormalized view refreshed on schedule.

JSONB for flexible attributes: Store semi-structured or highly variable data in PostgreSQL JSONB columns with GIN indexes. You get schema flexibility without an EAV (Entity-Attribute-Value) anti-pattern, and JSONB is fully queryable with operators like @> (containment) and ->> (field extraction).

- [PostgreSQL: Database design basics \(normalization\)](#).

Q27

When do you choose a NoSQL database over a relational database?

NoSQL

Spoken answer · ~90 s

I choose NoSQL when the data model, scale, or access pattern doesn't fit a relational database well, not because relational can't scale; modern relational systems scale very far. Relational is the default when you need complex queries and joins, strong transactional guarantees across multiple entities, flexible ad hoc querying, and a well-understood schema. NoSQL families suit specific cases. Key-value stores like Redis or DynamoDB suit very high-throughput lookups by key, caches, sessions, and counters, with predictable low latency at scale. Document stores such as MongoDB or Firestore suit hierarchical, evolving, or varied records read as a unit, like catalogues or profiles, where schema flexibility helps. Wide-column stores like Bigtable or Cassandra suit massive write throughput and time-series or event data at petabyte scale with known access patterns. Graph databases suit highly connected data and relationship traversals, like social networks or fraud detection. Search engines handle full-text search. Trade-offs to weigh: weaker or limited transactions and joins, often needing denormalisation and designing the data model around known queries, eventual consistency in some systems, less flexible querying, and operational differences. Consider the team's skills, managed service availability, consistency requirements, and the cost of getting it wrong. Many systems use polyglot persistence: a relational database as the system of record, with specialised stores for caching, search, or analytics.

Likely follow-ups

- Why isn't scale alone a reason to leave relational?
- What is polyglot persistence?

Use cases for NoSQL:

Document (MongoDB, Firestore): variable schema, nested objects, rapid iteration

Key-Value (Redis, Memystore): caching, sessions, counters, pub/sub

Wide-Column (Bigtable, Cassandra): time-series, IoT, 100M+ rows, write-heavy

Graph (Neo4j, Neptune): relationship traversal, social networks, fraud detection

Search (Elasticsearch, OpenSearch): full-text search, log analytics, faceted search

Choose NoSQL when: data doesn't fit a relational model (variable schema, deeply nested), write throughput exceeds what a single PostgreSQL instance can handle horizontally (Cassandra/Bigtable scale horizontally), or you need a specific access pattern that relational databases serve poorly (graph traversal, full-text search). Don't choose NoSQL to avoid learning SQL or as a premature optimization — most apps scale to tens of millions of rows fine on PostgreSQL with proper indexing.

Multi-document transactions: Most NoSQL databases have weak or no cross-document transactions. If your data model requires atomic updates across multiple "records" (like transferring money between accounts), a relational database with ACID transactions is

the correct choice. NoSQL consistency models (eventual, causal) require careful application-level handling.

Official docs

- [Azure: Choose a data store \(relational vs NoSQL\)](#)

Q28 How do you optimize Redis for production — what are the key performance and reliability concerns?

NoSQL

Spoken answer · ~87 s

Production Redis needs attention to performance, memory, persistence, and availability. Performance: Redis is single-threaded for command execution, so avoid slow commands like KEYS, large SMEMBERS, or big range operations; use SCAN instead, and watch the slowlog and latency monitoring. Use pipelining to cut round trips, and connection pooling. Avoid big keys and hot keys. Memory: set maxmemory and an eviction policy that fits the use, such as allkeys-lru for caches, or noeviction for primary data stores; set TTLs on cache keys; watch fragmentation ratio, and use memory-efficient data structures and encodings. Leave headroom for fork copy-on-write during snapshots. Persistence: RDB snapshots give compact point-in-time backups but can lose recent writes; AOF logs every write with fsync options like everysec, offering better durability at some cost; use both for important data, and put persistence on suitable disks. Availability: use replicas with Redis Sentinel for automatic failover, or Redis Cluster for sharding and scale; managed services like ElastiCache or Memorystore handle much of this. Be aware of replication being asynchronous, so failover can lose writes. Security: require authentication, ACLs, TLS, and keep it off the public internet. Operations: disable transparent huge pages, tune the OS settings, monitor memory, evictions, hit rate, connected clients, replication lag, and latency, and plan capacity.

Likely follow-ups

- Why avoid the KEYS command in production?
- What is the trade-off between RDB and AOF?

```
-- Check memory usage and eviction policy:
```

```
redis-cli INFO memory | grep -E "used_memory_human|maxmemory|mem_fragmentation"
```

```
redis-cli CONFIG GET maxmemory-policy # eviction when memory full
```

```
-- Set for cache workloads:
```

```
CONFIG SET maxmemory-policy allkeys-lru # evict least recently used
```

```
-- Find memory hogs:
```

```
redis-cli --bigkeys # scan for large keys
```

```
redis-cli MEMORY USAGE key_name # bytes for specific key
```

```
-- Check slow log:
```

```
redis-cli SLOWLOG GET 10          # last 10 slow commands
```

```
redis-cli CONFIG SET slowlog-log-slower-than 10000 # log commands > 10ms
```

Redis is single-threaded for command processing — one slow command (KEYS *, LRange with huge list, SORT without index) blocks all other clients. Never run KEYS * in production — use SCAN with COUNT for iteration. Large values (>1MB per key) slow serialization and network transfer. Pipeline multiple commands to reduce round-trips. Use Redis Cluster for horizontal scaling or Sentinel for HA without sharding. For GCP: Cloud Memorystore (managed Redis) handles replication and failover automatically.

KEYS * in production: KEYS * is $O(n)$ and blocks the entire Redis instance while it scans. On an instance with 10 million keys, it can block for seconds. Use `SCAN 0 COUNT 100` for iterating keys — it returns a cursor and is non-blocking.



Official docs

- [Redis documentation](#)

Q29

What is Bigtable/HBase — when is wide-column the right choice and how do you model data for it?

NoSQL

Spoken answer · ~86 s

Bigtable, and its open-source cousin HBase, is a wide-column NoSQL store: a sparse, distributed, sorted map. Data is organised in rows identified by a row key, with columns grouped into column families, and each cell can hold multiple timestamped versions. Rows are stored sorted lexicographically by row key and split into tablets spread across nodes, giving very high write and read throughput with single-digit millisecond latency at huge scale. It has no joins or secondary indexes, and offers atomic operations only at the single-row level. It's the right choice for very large datasets, terabytes to petabytes, with high-throughput writes and known access patterns: time-series and IoT data, monitoring metrics, event and clickstream data, financial ticks, personalisation, and analytics serving. Data modelling revolves around the row key, since it determines both locality and performance. Design the key for the main query pattern: lookups by prefix or range scans; include the entity ID and then a timestamp for time-series. Avoid hotspots from sequential keys, such as a timestamp prefix, which sends all writes to one node; use field promotion, salting, or hashing, or reversed timestamps for most-recent-first scans. Keep the number of column families small, put data accessed together in the same row, and keep rows reasonably sized.

Likely follow-ups

- Why do sequential row keys cause hotspots?
- What is field promotion?

```
-- Bigtable row key design (all access is by row key prefix scan):
```

```
-- Bad: monotonic timestamp as row key → hot spot on latest tablet
```

```
row_key = "2024-01-15T10:30:00#user123" -- all writes go to same server
```

```
-- Good: salted / reversed timestamp
```

```
row_key = "user123#~2024-01-15T10:30:00" -- spread across tablets
```

```
-- Or: hash prefix for write distribution
```

```
row_key = "hash(user_id)[0:4]#user123#timestamp"
```

```
gcloud bigtable instances list
```

```
cbt read <table> prefix="user123" count=10
```

Bigtable is designed for: massive scale (petabytes), high write throughput (millions of writes/second), and time-series or event data. Data model: one big sorted table — all queries are row key lookups or prefix scans. No secondary indexes, no joins, no SQL. Row key design IS your data model — you must design rows to support your access patterns. Hot spots (all writes going to one tablet) are the #1 performance problem — salt or hash the row key to distribute writes.

When NOT to use Bigtable: If your queries need secondary indexes, ad-hoc filters, or SQL-like flexibility, use Cloud Spanner or PostgreSQL. Bigtable's power comes from its access pattern constraints — brute-forcing relational patterns onto it produces worse results than just using a relational DB.

Official docs

- [Apache Cassandra documentation](#)

Spoken answer · ~84 s

Tuning Cloud SQL for PostgreSQL combines instance sizing, database flags, and good practices. Sizing: choose machine type with enough memory for the working set, SSD storage with enough capacity because IOPS and throughput scale with disk size, and enable automatic storage increase. Availability: enable the high availability configuration with a standby in another zone, and read replicas for scaling reads or cross-region disaster recovery. Configuration through database flags: tune `shared_buffers`, `work_mem`, `maintenance_work_mem`, and `effective_cache_size` sensibly, noting that Cloud SQL sets reasonable defaults based on instance size; set `max_connections` according to memory but rely on pooling; configure autovacuum aggressively for busy tables; enable `pg_stat_statements` and `log_min_duration_statement` to find slow queries; and set timeouts such as `statement_timeout` and `idle_in_transaction_session_timeout`. Connections: use the Cloud SQL Auth Proxy or connectors with private IP, application-side pooling, or managed connection pooling to avoid exhaustion. Backups: enable automated backups with point-in-time recovery, test restores, and pick a retention policy. Maintenance: set maintenance windows and deny periods. Security: private IP, IAM database authentication, SSL enforcement, and CMEK if required. Observability: Query Insights for top queries, plans, and wait events, plus Cloud Monitoring alerts on CPU, memory, disk, connections, and replication lag. Regularly review index usage and bloat, and load test.

Likely follow-ups

- What does Query Insights show you?
- Why does disk size affect performance on Cloud SQL?

```
-- Key Cloud SQL flags to configure:
```

```
max_connections = 200          # size based on pool config, not raw app connections
```

```
shared_buffers = 25% of RAM    # Cloud SQL sets this automatically based on tier
```

```
work_mem = 16MB                # adjust for analytics queries
```

```
pg_stat_statements.track = all # enable query insights
```

```
log_min_duration_statement = 1000 # log queries > 1s
```

```
-- Check Cloud SQL metrics in Cloud Monitoring:
```

```
cloudsql.googleapis.com/database/cpu/utilization
```

```
cloudsql.googleapis.com/database/memory/utilization
```

```
cloudsql.googleapis.com/database/disk/read_ops_count
```

```
cloudsql.googleapis.com/database/postgresql/num_backends
```

Cloud SQL-specific: use SSD storage for production (HDD is 3x cheaper but 10x slower IOPS). Enable automatic storage increase to avoid disk-full outages. Use private IP (no public IP) and connect via Cloud SQL Auth Proxy. Enable Query Insights (built-in pg_stat_statements visualization). Choose the right machine tier — CPU is usually not the bottleneck; RAM (for shared_buffers + OS cache) and IOPS matter most. For read scaling, add read replicas in the same region as the consumers.

Read replica for analytics: Route OLAP/reporting queries to a read replica to protect the primary from analytical workload interference. Use a larger replica tier if needed — you can size primary and replicas independently in Cloud SQL.

Official docs

- [Cloud SQL](#)

Q31 What is Cloud Spanner — when should you choose it over Cloud SQL?

Cloud databases

Spoken answer · ~89 s

Cloud Spanner is Google's fully managed, horizontally scalable, strongly consistent relational database. It offers SQL, schemas, and ACID transactions with external consistency across rows, tables, and even regions, using TrueTime, with automatic sharding and replication, and availability SLAs up to 99.999 percent in multi-region configurations. You scale by adding nodes or processing units, with online schema changes and no downtime for maintenance. Choose it over Cloud SQL when you need one or more of: scale beyond what a single primary can handle, for write throughput or storage in the many terabytes, without manual sharding; global distribution with consistent transactions across regions; very high availability without managing failover; or continuous growth where you want to avoid re-architecting later. Cloud SQL, with PostgreSQL or MySQL, is the better choice for conventional applications of moderate size: it's cheaper, fully compatible with standard engines and extensions, easier to migrate to, and supports familiar tooling. Trade-offs of Spanner: higher cost, especially at small scale, since it has a minimum capacity; schema design requirements such as avoiding monotonically increasing primary keys to prevent hotspots, and using interleaved tables or foreign keys with locality in mind; some differences from standard SQL dialects, though a PostgreSQL interface is available; and migration effort. Evaluate the actual scale and availability requirements before committing.

Likely follow-ups

- Why does Spanner cost more at small scale?
- What is TrueTime used for?

```
-- Spanner: globally distributed, horizontally scalable, ACID relational DB
```

```
-- When to choose Spanner:
```

```
✓ Need > 30,000 QPS on a single table (beyond single Cloud SQL node)
```

```
✓ Multi-region writes with external consistency (banking, booking systems)
```

```
✓ Zero-downtime horizontal scaling without sharding logic in app
```

```
✓ 99.999% SLA requirement
```

-- When NOT to choose Spanner:

x Small to medium workloads (< \$500/month budget)

x Complex analytical queries (use BigQuery instead)

x Requires stored procedures or PL/pgSQL (use Cloud SQL)

Spanner uses TrueTime (GPS + atomic clock synchronization) to achieve external consistency across regions — a property no other globally distributed database provides. Primary key design is critical (avoid monotonic keys — causes hot splits). Interleaved tables (Spanner's parent-child relationship) co-locate related rows on the same Spanner node, eliminating cross-node joins for the most common access pattern. Cost: ~\$0.90/node-hour for a single processing unit — significantly more than Cloud SQL. Justify with the SLA and scale requirements.

Cloud SQL → Spanner migration trigger: When you start sharding Cloud SQL manually (splitting users across multiple DB instances by ID range), that's the signal you should have moved to Spanner. Spanner eliminates application-level sharding entirely.

Official docs

- [Cloud SQL](#)
- [Cloud Spanner](#)

Q32 How do you perform a zero-downtime schema migration on a large table in production?

Ops & maintenance

Spoken answer · ~87 s

A zero-downtime schema migration on a large table means never holding long locks or breaking running application code. Principles: use the expand and contract pattern in separate steps, so old and new application versions both work. Expand: add new columns as nullable or with a default that doesn't rewrite the table, which modern PostgreSQL supports instantly for constants; add new tables; create indexes concurrently with CREATE INDEX CONCURRENTLY in PostgreSQL, or use online DDL in MySQL with ALGORITHM=INPLACE. Add constraints in two steps, such as NOT VALID then VALIDATE CONSTRAINT, to avoid long locks. Migrate: deploy code that writes to both old and new structures, then backfill existing rows in small batches with pauses, monitoring replication lag and load, rather than one huge UPDATE. Switch reads to the new column, verify, then stop writing the old one. Contract: remove old columns or tables in a later release. For changes that can't be done in place, such as changing a column type, use online schema change tools: gh-ost or pt-online-schema-change for MySQL, pg_repack or a shadow-table approach in PostgreSQL, which copy data to a new table while tracking changes, then swap. Always set lock_timeout so the migration fails fast instead of queuing behind a long transaction and blocking everything.

Likely follow-ups

- Why set a lock_timeout during migrations?
- How do online schema change tools work?

```
-- Problem: ALTER TABLE on a 500M row table takes hours and locks the table
```

```
-- Solution: online schema change tools
```

```
-- PostgreSQL: pg_repack or pgroll
```

```
pg_repack --table tablename --jobs 4 dbname # online table repack
```

```
-- MySQL: pt-online-schema-change (Percona) or gh-ost (GitHub)
```

```
pt-online-schema-change --alter "ADD COLUMN flag TINYINT DEFAULT 0" D=mydb,t=mytable --execute
```

```
gh-ost --alter "ADD COLUMN flag TINYINT DEFAULT 0" --database mydb --table mytable --execute
```

```
-- Manual expand-contract pattern:
```

```
Step 1: Add new nullable column (fast, no data rewrite)
```

```
Step 2: Deploy code to write to both old and new column
```

```
Step 3: Backfill new column in batches (UPDATE WHERE id BETWEEN x AND y)
```

```
Step 4: Deploy code to read from new column
```

```
Step 5: Drop old column
```

The expand-contract (blue-green schema) pattern is the safest for zero-downtime migrations. Never add NOT NULL columns without defaults in a single step — it locks the table for the full rewrite. Add as nullable first, backfill, then add the constraint. Use batched backfills with rate limiting to avoid overwhelming the database during migration.

Adding NOT NULL without default: ALTER TABLE ADD COLUMN status VARCHAR NOT NULL on a 100M row table rewrites every row immediately under an exclusive lock. Always add nullable first, backfill, then add NOT NULL constraint. PostgreSQL 11+ allows NOT NULL with a constant default without rewrite — but complex defaults still rewrite.

Official docs

- [gh-ost: GitHub's online schema migration](#)
- [Flyway documentation](#)

Spoken answer · ~88 s

Backups protect against data loss, and a backup you haven't restored is only a hope, so I treat restore testing as part of the job. Types: logical backups, like `pg_dump` or `mysqldump`, which are portable but slow for big databases and give a point-in-time snapshot only; physical backups, copies of the data files, such as `pg_basebackup`, `pgBackRest`, `Percona XtraBackup`, or cloud snapshots, which are faster to take and restore; and continuous archiving of WAL or binary logs for point-in-time recovery. Do it safely: take backups from a replica to avoid loading the primary, make sure they are consistent, encrypt them, store copies off-site and in a different account or region with immutability to resist ransomware, monitor success and size, and keep retention aligned with requirements. Restore regularly into a separate environment, verify data integrity, and time it. RPO, the recovery point objective, is the maximum acceptable data loss measured in time; it determines backup frequency and whether you need continuous WAL archiving or synchronous replication. RTO, the recovery time objective, is the maximum acceptable downtime to restore service; it determines the approach: restoring a large backup could take hours, so low RTO needs standby replicas, faster snapshots, or automated failover. Define them with the business, document runbooks, and run disaster recovery drills.

Likely follow-ups

- Why is replication not a backup?
- How do RPO and RTO drive the design?

```
-- PostgreSQL: logical backup (portable, slow for large DBs)
```

```
pg_dump -Fc -j 4 dbname > backup.dump      # parallel custom format
```

```
pg_restore -Fc -j 4 -d targetdb backup.dump
```

```
-- Physical backup (faster, same PG version required)
```

```
pg_basebackup -D /backup/base -Ft -z -P    # full base backup
```

```
-- PITR: restore base backup + replay WAL to target time
```

```
recovery_target_time = '2024-01-15 14:30:00' # in recovery.conf / postgresql.conf
```

```
-- Test your restore regularly:
```

```
pg_restore ... && psql -c "SELECT COUNT(*) FROM critical_table;" # verify
```

RTO (Recovery Time Objective): how long can the system be down? Drives backup restore speed requirements. RPO (Recovery Point Objective): how much data loss is acceptable? Drives backup frequency and replication lag tolerance. A daily backup gives RPO of 24 hours (you lose up to a day of data). WAL archiving gives RPO of minutes (replay WAL up to the last archived segment). Streaming replication with sync commit gives RPO of zero. Test restores monthly — an untested backup is an untested failure.

Cloud SQL PITR: Enable in Cloud SQL with one flag. GCP retains base backups + WAL for up to 7 days. Restore to any second within that window. This makes Cloud SQL's RTO/RPO significantly better than self-managed PostgreSQL with manual WAL archiving.

Official docs

- [AWS Disaster Recovery of workloads \(whitepaper\)](#).
- [PostgreSQL: Continuous archiving and PITR](#)
- [Google SRE Book: Data Integrity](#).

Q34 Database disk is filling up — how do you diagnose and resolve it without downtime?

Ops & maintenance

Spoken answer · ~86 s

When a database disk is filling up, first assess how fast and what is growing, and buy time. Check space usage by component: data files, indexes, WAL or binary logs, temporary files, logs, and backups on the same volume. In PostgreSQL, find the largest tables and indexes using `pg_total_relation_size` and check `pg_wal`; a stuck replication slot or failing `archive_command` makes WAL pile up, and a long-running transaction can prevent vacuum, causing bloat. In MySQL, check binary log retention, relay logs, temp tables, and the slow log. Immediate relief without downtime: expand the volume online if the platform supports it, as cloud disks usually do, or enable automatic storage growth; drop an abandoned replication slot, fix archiving, purge old binary logs safely after confirming replicas have consumed them, rotate or compress logs, and delete unneeded temp files or old backups. Don't delete WAL or data files manually. Then address root causes: bloat, by tuning autovacuum and rebuilding with `pg_repack`; unneeded indexes; large tables where you can archive or purge old data, using partitioning to drop old partitions quickly; data growth that needs capacity planning; and runaway queries generating temp files. Prevent recurrence with alerts at 70, 80, and 90 percent, growth-rate forecasting, retention policies, and regular capacity reviews.

Likely follow-ups

- Why can a replication slot fill the disk?
- What would you never delete manually?

```
-- PostgreSQL: find largest objects
```

```
SELECT schemaname, tablename,
pg_size_pretty(pg_total_relation_size(schemaname||'.'||tablename)) AS size FROM
information_schema.tables ORDER BY pg_total_relation_size(schemaname||'.'||tablename) DESC
LIMIT 20;
```

```
-- Check for WAL accumulation:
```

```
SELECT pg_size_pretty(sum(size)) FROM pg_ls_waldir();
```

```
-- Check for bloated tables vs actual live data:
```

```
SELECT relname, n_dead_tup, pg_size_pretty(pg_relation_size(oid)) FROM pg_stat_user_tables
ORDER BY n_dead_tup DESC LIMIT 10;
```

```
-- Run VACUUM to reclaim dead tuple space:
```

```
VACUUM VERBOSE ANALYZE tablename;
```

```
-- Emergency: drop old partitions, truncate log tables
```

```
DROP TABLE events_2022_q1; -- instant, no VACUUM needed
```

Disk space culprits: table bloat from dead tuples (VACUUM to reclaim), WAL accumulation from inactive replication slots or archive failure, index bloat, and log data growth. Immediate actions: run VACUUM on bloated tables, drop old partitions (instant disk recovery), check and fix WAL archiving, drop unused indexes. Long-term: implement automated archival/deletion of old data, monitor disk usage with alerts at 70% and 85% full.

Replication slot disk bomb: An inactive replication slot silently accumulates WAL — primary cannot remove WAL that the slot hasn't consumed. This is the most common cause of unexpected disk exhaustion in production PostgreSQL. Set `max_slot_wal_keep_size = 10GB` in `postgresql.conf` as a safety cap.

Official docs

- [PostgreSQL: Routine vacuuming](#)
- [PostgreSQL: Write-Ahead Logging](#)

Spoken answer · ~89 s

In Kubernetes, connection exhaustion is amplified by scale: each pod has its own pool, so total connections equal pods times pool size, and autoscaling or rolling updates multiply that, since surge pods briefly double connections. First, do the arithmetic: compare the maximum replicas times pool size, with extra for rollouts, against the database's max_connections. Diagnose with `pg_stat_activity` grouped by client address to see which workloads hold connections, and look for idle and idle-in-transaction sessions. Solutions: reduce per-pod pool sizes to what each pod needs, as it's usually much smaller than the default; use an external pooler such as PgBouncer, deployed as a Deployment or sidecar, or a managed one like Cloud SQL's pooling or RDS Proxy, so thousands of client connections multiplex onto a small number of server connections; set pool timeouts, max lifetime, and idle timeouts so connections are recycled and leaks detected. Ensure graceful shutdown closes connections on SIGTERM. Use connection retries with exponential backoff and jitter to avoid thundering herds at restart. Limit HPA maximum replicas relative to database capacity, and use PodDisruptionBudgets and slower rollout settings. Consider separating read traffic to replicas. Add application-level backpressure, such as queueing, circuit breakers and timeouts, rather than letting requests pile up. Monitor connection counts per service, pool wait times, and alert on utilisation.

Likely follow-ups

- How do rolling updates multiply connections?
- Where would you place PgBouncer in Kubernetes?

```
-- Architecture: pods → PgBouncer → PostgreSQL
```

```
-- PgBouncer as a Kubernetes Deployment:
```

```
kubectl get pods -l app=pgbouncer -n db
```

```
kubectl exec -it pgbouncer-pod -- psql -p 6432 pgbouncer -c "SHOW POOLS;"
```

```
-- Key metrics to watch:
```

```
cl_waiting: clients waiting for a connection (> 0 = pool exhausted)
```

```
sv_active: active server connections
```

```
sv_idle: idle server connections available
```

```
-- Scale response: increase pool_size in PgBouncer config
```

```
-- Or: scale PgBouncer horizontally (multiple replicas with shared backend)
```

In Kubernetes, connection exhaustion is amplified by HPA scaling — $50 \text{ pods} \times 20 \text{ pool connections} = 1000 \text{ connections}$. Architecture: each app pod connects to PgBouncer (ClusterIP Service), PgBouncer maintains a small pool of actual PostgreSQL connections. PgBouncer itself can be scaled horizontally — multiple PgBouncer replicas share the backend PostgreSQL connection budget. Use transaction-mode pooling for highest efficiency. Monitor `cl_waiting` in PgBouncer — any clients waiting means the pool is undersized.

Pgpool vs PgBouncer: PgBouncer is lightweight and fast — best for connection pooling only. Pgpool-II adds load balancing, query caching, and connection pooling but is heavier. For most Kubernetes deployments, PgBouncer is the right choice. Pgpool only if you need its additional features.

Official docs

- [PgBouncer](#)

Q36

What is a partial index — when does it dramatically outperform a full index?

Indexing

Spoken answer · ~88 s

A partial index covers only the rows that satisfy a WHERE condition, instead of every row in the table. For example, `CREATE INDEX ON orders (customer_id) WHERE status = 'pending'`. The index is smaller, faster to scan, cheaper to maintain, since updates to rows outside the condition don't touch it, and it fits more easily in memory. It dramatically outperforms a full index when queries target a small, well-defined subset of a large table. Classic cases: a queue or job table where most rows are completed and queries look only for the unprocessed ones; soft-deleted data, indexing only where `deleted_at` IS NULL; a flag with skewed distribution, such as indexing only the rare true values; recent or active data; and enforcing uniqueness on a subset, such as unique email among active users, which a full unique index can't express. The query must include a predicate the planner can prove implies the index condition, so the condition in the query must match or be a stricter form of the index's WHERE clause; parameterised queries can prevent the proof if the predicate isn't a constant in the query. Downsides: it helps only the specific query shapes, adds an index to maintain and to reason about, and a mismatched predicate means it won't be used.

Likely follow-ups

- What must a query contain for the planner to use a partial index?
- How can a partial index enforce conditional uniqueness?

```
-- Full index on status column: indexes ALL rows including completed orders
```

```
CREATE INDEX idx_orders_status ON orders (status);
```

```
-- Partial index: only indexes rows where status = 'pending'
```

```
CREATE INDEX idx_orders_pending ON orders (user_id, created_at) WHERE status = 'pending';
```

```
-- This query uses the partial index efficiently:
```

```
SELECT * FROM orders WHERE status = 'pending' AND user_id = 123 ORDER BY created_at;
```

```
-- Check index size comparison:
```

```
SELECT pg_size_pretty(pg_relation_size('idx_orders_pending'));
```

Partial indexes index only a subset of rows that match a WHERE condition. If your table has 100M orders but only 10,000 are 'pending' (the rows you query 99% of the time), a partial index on pending orders is 10,000x smaller, fits in cache, and is dramatically faster than a full index. Classic use cases: active/inactive flags (index only active), soft deletes (index only non-deleted), unprocessed queue items (index only unprocessed status).

Real-world impact: A table with 500M rows where 0.01% are "active" — a full index is 500M entries; a partial index on active rows is 50K entries. The partial index fits entirely in shared_buffers; the full index is a cold disk read. This single change can reduce query time from seconds to microseconds.



Official docs

- [PostgreSQL: Indexes](#)

Q37

What are CTEs (WITH clauses) — when do they help vs hurt performance?

Query optimization

Spoken answer · ~89 s

A Common Table Expression, introduced with WITH, defines a named temporary result set that a query can reference, improving readability by breaking complex logic into steps. They also enable recursive queries, such as traversing hierarchies or graphs, and let you reuse a subquery result several times in a statement, and chain data-modifying statements in PostgreSQL. Performance depends on the engine. Historically in PostgreSQL before version 12, CTEs were always materialised, acting as optimisation fences: the planner couldn't push predicates into them, which could make queries much slower, since the entire CTE was computed first. From version 12, non-recursive, side-effect-free CTEs referenced once are inlined by default, behaving like subqueries; you can force the old behaviour with MATERIALIZED or force inlining with NOT MATERIALIZED. Materialisation helps when the CTE is expensive and referenced multiple times, since it's computed once, or when you want to deliberately control the plan. It hurts when it materialises a huge set that a filter could have reduced, or prevents index use. MySQL 8 and others generally merge CTEs like derived tables. Guidelines: use CTEs for clarity, check the execution plan, avoid assuming they're free, and for hot paths test alternatives such as subqueries, temporary tables with indexes, or joins. Recursive CTEs need a termination condition to avoid endless loops.

Likely follow-ups

- What changed for CTEs in PostgreSQL 12?
- When is materialising a CTE beneficial?

```
-- PostgreSQL 12+: CTEs are NOT optimization fences by default
```

```
WITH active_users AS (  
  SELECT id FROM users WHERE status = 'active'  
)  
SELECT o.* FROM orders o JOIN active_users u ON o.user_id = u.id;
```

```
-- Force materialization (pre-12 behavior, optimization fence):
```

```
WITH active_users AS MATERIALIZED ( SELECT id FROM users WHERE status = 'active' )
```

```
-- Prevent materialization (inline the CTE):
```

```
WITH active_users AS NOT MATERIALIZED ( SELECT id FROM users WHERE status = 'active' )
```

```
-- Recursive CTE: tree traversal
```

```
WITH RECURSIVE org_tree AS (  
  SELECT id, name, parent_id FROM employees WHERE parent_id IS NULL  
  UNION ALL  
  SELECT e.id, e.name, e.parent_id FROM employees e JOIN org_tree o ON e.parent_id = o.id  
) SELECT * FROM org_tree;
```

In PostgreSQL before version 12, CTEs were optimization fences — the planner couldn't push filters through them, and they were always materialized (computed once, stored in memory). In PG 12+, simple CTEs are inlined by default. When to use **MATERIALIZED**: when the CTE is referenced multiple times and computing it once saves repeated scans. When to avoid: CTE on a large table with a filter that could be pushed to an index — use a subquery instead so the planner can optimize.

Recursive CTEs: Recursive CTEs are the correct tool for hierarchical data (org charts, category trees, graph traversal). They outperform application-level recursion by orders of magnitude. Learn the pattern — it comes up in any job involving hierarchical data.

Official docs

- [PostgreSQL: WITH queries \(CTEs\)](#)

Spoken answer · ~88 s

Physical replication, PostgreSQL's streaming replication, ships the WAL, the byte-level changes to data pages, so the standby is an exact binary copy of the entire cluster. It replicates everything, all databases, tables, and indexes, requires the same major version and architecture, and the standby is read-only. It's ideal for high availability, failover, and read replicas. Logical replication decodes the WAL into logical changes, inserts, updates, and deletes at the row level, and publishes them for chosen tables using a publication-subscription model: the subscriber applies the changes as ordinary SQL operations. Differences and benefits: you can replicate selected tables rather than the whole cluster; the subscriber is a normal writable database, so it can have additional tables, different indexes, or even a different schema layout; it works across major PostgreSQL versions, enabling near-zero-downtime upgrades; it supports consolidating data from several sources and feeding other systems through logical decoding, as in change data capture with Debezium. Limitations: it doesn't replicate DDL or schema changes automatically, sequences, or large objects by default; tables generally need a primary key or replica identity; conflicts can occur on writable subscribers; performance overhead is higher; and it requires `wal_level` logical. Use physical replication for HA and read scaling, and logical replication for migrations, upgrades, selective replication, and data integration.

Likely follow-ups

- How does logical replication enable major-version upgrades?
- Why does logical replication need a replica identity?

```
-- Physical replication: streams WAL byte-for-byte (same PG version required)
```

```
-- Logical replication: streams row-level changes (flexible – cross-version, cross-DB)
```

```
-- Set up logical replication:
```

```
wal_level = logical # in postgresql.conf (requires restart)
```

```
-- On publisher:
```

```
CREATE PUBLICATION my_pub FOR TABLE users, orders;
```

```
-- On subscriber:
```

```
CREATE SUBSCRIPTION my_sub CONNECTION 'host=primary ...' PUBLICATION my_pub;
```

```
-- Check replication status:
```

```
SELECT * FROM pg_stat_subscription;
```

Logical replication use cases: zero-downtime major version upgrades (logical rep from old to new version), selective table replication (replicate only specific tables to analytics DB), cross-database aggregation, and CDC (Change Data Capture) with tools like Debezium. Physical replication is better for HA replicas (faster, lower overhead, full DB copy). Logical replication allows schema differences between publisher and subscriber — the subscriber can have additional indexes, different column ordering, or even a different DB system (via Debezium → Kafka).

Replication slot management: Logical replication uses replication slots. A slot for a Debezium consumer that goes offline for 24 hours will accumulate 24 hours of WAL on the primary. Always monitor slot lag and set `max_slot_wal_keep_size` to prevent disk exhaustion.

Official docs

- [PostgreSQL: High availability, load balancing, replication](#)
- [PostgreSQL: Logical replication](#)

Q39 How do you efficiently store and query time-series data in a relational database?

Schema design

Spoken answer · ~85 s

Time-series data is append-heavy, queried by time ranges, and often aged out. In a relational database, the key techniques are as follows. Partition by time, for example daily or monthly range partitions, so queries prune to relevant partitions, inserts hit small hot partitions, and old data is dropped instantly by detaching or dropping partitions. Index carefully: a B-tree on timestamp plus the entity ID, or a BRIN index for naturally ordered data, which is tiny and effective for large append-only tables. Keep rows narrow with compact types. Cluster or order data physically by time so range scans are sequential. Batch inserts or use COPY for high ingest rates. Pre-aggregate: maintain rollup tables or continuous aggregates for hourly and daily summaries, so dashboards don't scan raw data, with raw data retained for a shorter period. Use retention policies, and compress or move older partitions to cheaper storage. Extensions like TimescaleDB automate this with hypertables, automatic chunking, native compression, continuous aggregates, and retention policies, while staying within PostgreSQL. Query tips: always filter on the time column, use `time_bucket` or `date_trunc` for aggregation, and avoid functions on the timestamp column in predicates. At very large scale or high cardinality, consider purpose-built stores like InfluxDB, ClickHouse, Prometheus, or Bigtable.

Likely follow-ups

- Why is a BRIN index suited to time-series data?
- How do rollups protect dashboard performance?

```
-- Partition by time range (monthly or quarterly):
```

```
CREATE TABLE metrics (ts TIMESTAMPTZ NOT NULL, host TEXT, value FLOAT) PARTITION BY RANGE (ts);
```

```
CREATE TABLE metrics_2024_01 PARTITION OF metrics FOR VALUES FROM ('2024-01-01') TO ('2024-02-01');
```

```
-- Composite index: (host, ts) for common access pattern
```

```
CREATE INDEX ON metrics_2024_01 (host, ts DESC);
```

```
-- BRIN index: very cheap for sequential time data
```

```
CREATE INDEX ON metrics_2024_01 USING BRIN (ts); -- tiny index, good for seq scans
```

```
-- Or: TimescaleDB extension for automated time-series management
```

```
SELECT create_hypertable('metrics', 'ts', chunk_time_interval => INTERVAL '1 week');
```

Time-series patterns: always include timestamp in indexes as the trailing sort column (most queries are "latest N for entity X"), use DESC sort on timestamp index to serve "latest" queries from the leftmost pages, partition by time for fast archival (drop old partitions instead of DELETE), and use BRIN indexes for append-only time-ordered data (BRIN is 1000x smaller than B-tree for sequential data). For high-cardinality metrics (Prometheus-style), use TimescaleDB or a purpose-built TSDB (Victoria Metrics, InfluxDB).

TimescaleDB: TimescaleDB is a PostgreSQL extension that adds automatic time-based partitioning (hypertables), continuous aggregates, and data retention policies. It's fully compatible with standard PostgreSQL — same SQL, same tools, same connection pooling. For serious time-series workloads on PostgreSQL, it's the right tool.

Official docs

- [Azure: Choose a data store \(relational vs NoSQL\)](#)

Spoken answer · ~86 s

Database observability combines metrics, logs, and query-level insight. Metrics to watch: availability and connection counts relative to the limit; query throughput and latency percentiles; CPU, memory, disk I/O, IOPS and latency, and disk space; cache hit ratio and buffer pool behaviour; replication lag and status; lock waits, deadlocks, and long-running or idle-in-transaction sessions; autovacuum activity, dead tuples, bloat, and transaction ID age in PostgreSQL; checkpoint frequency, WAL generation, and temp file usage; and in MySQL, InnoDB metrics and slow query counts. Logs: slow query logs with a sensible threshold, error logs, deadlock logs, lock wait logging, autovacuum logs, and connection logs; use `auto_explain` to capture plans for slow queries. Query-level: `pg_stat_statements`, Performance Schema, or cloud tools like Query Insights and Performance Insights to find top queries by total time, with wait events. Tracing in the application connects slow requests to specific queries. Alerts should be actionable: connections above eighty percent, replication lag over a threshold, disk space and growth forecasts, rising query latency compared with baseline or SLO, failed backups, deadlock spikes, long transactions, and XID wraparound risk. Dashboards show the golden signals per database, with deployment markers. Add synthetic checks and regularly test restore and failover. Review dashboards and alerts after incidents and capacity-plan using trends.

Likely follow-ups

- Which alerts would page someone?
- How do wait events help diagnosis?

```
-- Critical PostgreSQL metrics to monitor:
```

```
SELECT sum(blks_hit)*100.0/(sum(blks_hit)+sum(blks_read)) FROM pg_stat_database; -- cache hit rate > 99%
```

```
SELECT count(*) FROM pg_stat_activity WHERE state = 'active'; -- active connections
```

```
SELECT count(*) FROM pg_stat_activity WHERE wait_event_type = 'Lock'; -- lock waits
```

```
SELECT now() - pg_last_xact_replay_timestamp() FROM pg_stat_replication; -- replica lag
```

```
SELECT age(datfrozenxid) FROM pg_database ORDER BY age DESC LIMIT 1; -- XID age
```

```
-- Export to Prometheus:
```

```
postgres_exporter --web.listen-address=:9187 --config.file=config.yml
```

Alert thresholds: cache hit rate <99% (increase `shared_buffers` or add RAM), connection count >80% of `max_connections` (add PgBouncer or scale), replication lag >30s (investigate replica), transaction ID age >1.5B (urgent VACUUM needed), disk >80% full (archival or resize), lock wait time >5s (investigate blocking query). Use `postgres_exporter` with Prometheus + Grafana for the full metrics pipeline, or Cloud SQL's built-in Query Insights for managed deployments.

Principal-level practice: Build SLO-based DB alerting — not just "query_time > 1s" but "99th percentile query latency error budget burn rate". This separates platform observability maturity from ad-hoc monitoring. Treat the database as a service with its own SLO, not just infrastructure to watch.

Official docs

- [PostgreSQL: High availability, load balancing, replication](#)
- [PostgreSQL: pg_stat statements](#)

Q41 What is a materialized view — how do you use it for performance and when does it go stale?

Query optimization

Spoken answer · ~88 s

A materialised view stores the result of a query physically on disk, unlike a regular view, which is just a saved query that runs each time. Querying a materialised view is fast because the expensive joins and aggregations are already computed, and you can index it. It suits reporting, dashboards, and summary data where the underlying query is costly and slightly stale results are acceptable. In PostgreSQL, you create it with `CREATE MATERIALIZED VIEW` and update it with `REFRESH MATERIALIZED VIEW`, which recomputes everything and locks reads during the refresh; `REFRESH MATERIALIZED VIEW CONCURRENTLY` avoids blocking readers but requires a unique index and takes longer. PostgreSQL doesn't refresh them automatically or incrementally; you schedule refreshes with `cron`, `pg_cron`, or a job, or trigger them on data changes. Some databases, such as Oracle, Snowflake, and BigQuery, offer automatic or incremental refresh. The view goes stale between refreshes: it shows data as of the last refresh, so decide on acceptable staleness, set the refresh frequency to match, and expose the refresh time to users. Considerations: refresh cost grows with data size; refreshing too frequently can overload the database; storage overhead; and consistency, since the view may disagree with base tables. Alternatives include summary tables maintained incrementally by triggers or streaming jobs, and caching.

Likely follow-ups

- What does `REFRESH CONCURRENTLY` require?
- How do you handle acceptable staleness?

```
-- Create a materialized view (stores results on disk):
```

```
CREATE MATERIALIZED VIEW monthly_revenue AS
  SELECT DATE_TRUNC('month', created_at) AS month, SUM(total) AS revenue FROM orders GROUP
 BY 1 ORDER BY 1;
WITH DATA;
```

```
-- Refresh (blocks reads in PostgreSQL by default):
```

```
REFRESH MATERIALIZED VIEW monthly_revenue;
```

```
-- Non-blocking refresh (PostgreSQL 9.4+, requires unique index on matview):
```

```
CREATE UNIQUE INDEX ON monthly_revenue (month);
```

```
REFRESH MATERIALIZED VIEW CONCURRENTLY monthly_revenue;
```

```
-- Query the matview like a regular table:
```

```
SELECT * FROM monthly_revenue WHERE month >= '2024-01-01';
```

Materialized views precompute expensive aggregations and store the result. Great for: dashboards and reporting queries that aggregate millions of rows (run once, query many times), denormalizing normalized data for read performance, and caching the result of complex multi-table joins. Stale data: the matview reflects the data at the last REFRESH time — it does not auto-update. Schedule REFRESH CONCURRENTLY via pg_cron or external scheduler. For near-real-time data, TimescaleDB Continuous Aggregates incrementally update materialized views as new data arrives.

pg_cron for scheduled refresh: `SELECT cron.schedule('refresh-revenue', '0 * * * *', 'REFRESH MATERIALIZED VIEW CONCURRENTLY monthly_revenue');` — refreshes hourly with no application code change needed. pg_cron runs inside PostgreSQL, survives container restarts, and has no external dependency.

Official docs

- [PostgreSQL: Materialized views](#)

Q42 What is EXPLAIN in MySQL — how do you interpret the output to find performance problems?

MySQL

Spoken answer · ~80 s

In MySQL, EXPLAIN shows how the optimiser plans to execute a query, with one row per table or subquery in the plan. Key columns. type describes the access method, from best to worst: system, const, eq_ref, ref, range, index, and ALL; ALL means a full table scan, which is the red flag on large tables, and index means a full index scan. possible_keys lists candidate indexes, while key shows the one actually chosen; a NULL key means none is used. key_len shows how much of a composite index is used, revealing whether all columns are being used. rows estimates how many rows will be examined, and filtered the percentage kept after conditions; multiplying across joined tables shows the total work. Extra gives crucial hints: Using index means a covering index; Using where; Using temporary and Using filesort signal temporary tables and extra sorting, which are costly on big data; Using index condition means index condition pushdown. Use EXPLAIN ANALYZE in MySQL 8 to get actual timings and row counts, and EXPLAIN FORMAT=JSON or JSON for more detail. Problems to find: full scans, no usable index, filesorts, large row estimates, and poor join order.

Likely follow-ups

- What does type ALL indicate?
- How does key_len help with composite indexes?

```
EXPLAIN SELECT * FROM orders WHERE user_id = 123 AND status = 'pending';
```

```
EXPLAIN FORMAT=JSON SELECT ...; -- detailed JSON output (MySQL 8+)
```

```
EXPLAIN ANALYZE SELECT ...; -- actually executes + shows real times (MySQL 8.0.18+)
```

-- Key columns in EXPLAIN output:

type: ALL=full scan(bad), index=index scan, range=range scan, ref=non-unique index, eq_ref=unique, const=best

key: which index was chosen (NULL = no index used)

rows: estimated rows examined (multiply across joins for total cost)

Extra: "Using filesort"=expensive sort, "Using temporary"=temp table, "Using index"=covering index(good)

type column priority (best to worst): system > const > eq_ref > ref > range > index > ALL. Seeing type=ALL on a large table is a red flag — add an index. "Using filesort" means MySQL cannot satisfy ORDER BY with an index — add the sort column to the index. "Using temporary" means a temp table was created for GROUP BY or ORDER BY — restructure the query or add a covering index. rows × loops (for nested queries) gives the total row examination cost.

Index hints: If MySQL picks the wrong index, use `FORCE INDEX (idx_name)` temporarily to confirm it's an optimizer issue, then fix by updating statistics (`ANALYZE TABLE`) rather than permanently forcing an index. Forced indexes break silently when the data distribution changes.

Official docs

- [MySQL: EXPLAIN output](#)

Q43 What is optimistic vs pessimistic locking — when do you choose each?

Connections

Spoken answer · ~87 s

Both strategies handle concurrent modification of the same data. Pessimistic locking assumes conflicts are likely: it locks the row up front, for example `SELECT ... FOR UPDATE`, so other transactions block until the lock is released, which prevents conflicting updates entirely. It's simple to reason about and guarantees no lost updates, but holds locks, reducing concurrency, risks deadlocks, and hurts throughput, particularly if transactions are long or span user interaction. Optimistic locking assumes conflicts are rare: it reads without locks, then at update time checks that the data hasn't changed since it was read, typically using a version number or timestamp column: `UPDATE items SET qty = ?, version = version + 1 WHERE id = ? AND version = ?`. If zero rows are affected, someone else changed it, and the application retries or reports a conflict. It avoids holding locks, scales well for read-heavy workloads and long user think-times, and works across stateless services, but conflicts cause retries and wasted work, and under high contention it can thrash. Choose pessimistic when contention on the same rows is high, the cost of retry or failure is high, or operations are short and critical, like inventory decrement or ticket booking with strict guarantees, or queue processing with `SKIP LOCKED`.

Likely follow-ups

- How does a version column detect conflicts?
- When is `SKIP LOCKED` useful?

🔒 Locking styles

Pessimistic	Optimistic
Lock rows up front	Version check at commit
Safe for high contention	Best for low contention

```
-- Pessimistic locking: lock the row before reading (prevent concurrent updates)
```

```
BEGIN;  
SELECT * FROM inventory WHERE product_id = 1 FOR UPDATE; -- row is now locked  
UPDATE inventory SET quantity = quantity - 1 WHERE product_id = 1;  
COMMIT;
```

```
-- Optimistic locking: no lock, use version column to detect conflict
```

```
SELECT id, quantity, version FROM inventory WHERE product_id = 1;  
-- In application: compute new_quantity = quantity - 1  
UPDATE inventory SET quantity = ?, version = version + 1  
WHERE product_id = 1 AND version = ?; -- if 0 rows updated: conflict, retry
```

Pessimistic locking: acquire a lock before reading. Safe but reduces concurrency — all readers of the same row must queue. Use when: conflict probability is high (hot inventory items, bank account balances), the operation must not fail (you can't retry).

Optimistic locking: no locks, detect conflicts at write time using a version counter. Use when: conflict probability is low (most reads are followed by a write that doesn't conflict), read-heavy workloads (no lock contention). ORM frameworks (Hibernate, ActiveRecord) support optimistic locking natively via a version column.

SELECT FOR UPDATE scope: A lock acquired with SELECT FOR UPDATE is released only at COMMIT or ROLLBACK. A slow application-level operation between the lock acquisition and the commit holds the lock for that entire duration — blocking all other writers. Keep critical section (SELECT FOR UPDATE → UPDATE → COMMIT) as short as possible.

📖 Official docs

- [PostgreSQL: Explicit locking \(optimistic vs pessimistic\)](#)

Spoken answer · ~78 s

Database migrations in a CI/CD pipeline must be automated, versioned, tested, and safe for rolling deployments. Keep migration scripts in the repository with a tool such as Flyway, Liquibase, Alembic, or the framework's migrator, applied in order and tracked in a schema history table; never edit an applied migration. In CI, validate by running all migrations against a fresh database and against a copy or snapshot of production-like data, to catch errors and measure duration and locking; lint migrations for dangerous operations, using tools such as squawk or strong_migrations. Deploy as a distinct pipeline step that runs once, before the application rollout, using a Job or pipeline stage with a lock, not on each replica's startup. Design every change as backward compatible using expand and contract, since old and new application versions run simultaneously: add before use, backfill in batches, and remove only in a later release. Use lock_timeout, concurrent index creation, and online schema change tools for large tables. Use credentials with minimal necessary privilege. Take a backup or snapshot before risky releases, and plan rollback as roll forward or application rollback, since reverse migrations are risky.

Likely follow-ups

- Why avoid editing an already applied migration?
- What would a migration linter catch?

```
-- Tools: Flyway, Liquibase, Alembic (Python), golang-migrate, sqitch
```

```
-- Flyway example migration versioning:
```

```
V1__Create_users_table.sql
```

```
V2__Add_email_index.sql
```

```
V3__Add_status_column.sql
```

```
-- Run in CI/CD:
```

```
flyway migrate -url=jdbc:postgresql://... -user=app -password=... -  
locations=filesystem:./migrations
```

```
-- Safe migration checklist:
```

```
✓ Add columns as nullable (no NOT NULL without default)
```

```
✓ Add indexes CONCURRENTLY (non-blocking)
```

```
✓ Never rename columns (use add + backfill + drop)
```

```
✓ Never change column types directly (add new column, migrate data)
```

```
✓ Use expand-contract for breaking schema changes
```

Schema migrations in CI/CD: migrations run automatically on deploy, schema and code changes must be backward-compatible for the duration of a rolling deployment (both old and new code run simultaneously against the new schema). Add columns before deploying code that uses them; drop columns only after all code references to them are removed. Use migration tools that track applied versions and are idempotent. Separate DDL migrations (fast, schema only) from data migrations (slow, batched) to avoid long deployment windows.

GitOps for migrations: Commit migration files to Git alongside application code. The deployment pipeline applies pending migrations before starting the new app version. Migration history is version-controlled and auditable. Failed migrations should be caught in staging — never let an untested migration near production schema.

Official docs

- [Flyway: Database migrations in CI/CD](#)

Q45

Walk through your methodology for diagnosing a production database performance incident.

Ops & maintenance

Spoken answer · ~90 s

My methodology for a production database performance incident is structured: stabilise, observe, localise, fix, learn. First, confirm and scope the impact: which services and users, since when, and what changed recently, such as deployments, configuration, traffic, or failovers. If a recent change is a likely cause, roll it back early. Then look at database health in layers. Resources: CPU, memory, disk I/O and latency, network, and disk space; saturation indicates the bottleneck type. Connections: counts against limits, queued requests, idle-in-transaction sessions. Activity: what's running right now, from `pg_stat_activity` or the process list, sorted by duration; look for long-running queries, lock waits and blocking chains, and deadlocks. Query level: use `pg_stat_statements` or slow logs for the top queries by total time and any new or regressed ones, then `EXPLAIN ANALYZE` for the culprit to find missing indexes, plan changes, or stale statistics. Background work: autovacuum, checkpoints, backups, replication lag, and batch jobs. Mitigate fast with safe actions: kill the runaway or blocking queries, add or enable throttling, scale up or add replicas, redirect traffic, disable a heavy feature, or fix with an index created concurrently. Communicate status regularly. After recovery, verify against SLOs, write a blameless postmortem covering root cause and detection gaps, and add alerts, query limits, load tests, and capacity plans to prevent recurrence.

Likely follow-ups

- What is the first question you ask about recent changes?
- Which safe mitigations can you apply immediately?

This is the Principal-level systems thinking question. The interviewer wants structure, prioritization, and depth — not a random list of commands.

INCIDENT TRIAGE METHODOLOGY

Phase 1 – Quantify: what's actually slow? Which query/table/operation? Since when?

Phase 2 – Immediate relief: kill blocking queries, kill long-running idle transactions

Phase 3 – Identify hottest queries: `pg_stat_statements ORDER BY total_exec_time DESC`

Phase 4 – Explain the worst query: `EXPLAIN (ANALYZE, BUFFERS)` – look for seq scans, bad estimates

Phase 5 – Check system resources: CPU, I/O wait, memory pressure, connection count

Phase 6 – Check for lock contention: `pg_stat_activity WHERE wait_event_type = 'Lock'`

Phase 7 – Check for bloat: `n_dead_tup`, `last_autovacuum` – is VACUUM behind?

Phase 8 – Implement fix: index, query rewrite, or VACUUM

Phase 9 – Verify and document: before/after metrics, root cause, prevention

Common root causes by symptom: sudden slowdown with high I/O wait → autovacuum running or index bloat. Sudden slowdown after deploy → plan regression (stale stats after schema change, run ANALYZE). Gradual slowdown over weeks → data growth past index threshold, table bloat. Spike-and-recover pattern → lock contention from long transactions. Steady increase in query time → N+1 queries introduced in new code or missing index on growing table.

Principal-level deliverable: After any performance incident, write a postmortem with: the query/queries involved, the root cause (index missing, stats stale, connection exhaustion), timeline, the fix applied, and what will prevent recurrence (alert added, index created, VACUUM frequency tuned). This turns a reactive incident into a proactive improvement — that is the Principal Platform Engineer's job.



Official docs

- [PostgreSQL: Routine vacuuming](#)

Q46

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

Indexing

Spoken answer · ~58 s

A clustered index stores the table rows themselves in the order of the index key, so there can be only one per table. In MySQL InnoDB the primary key is the clustered index, and secondary indexes store the primary key value as their row pointer, which means a long primary key makes every secondary index larger and lookups need a second traversal. In SQL Server the clustered index is chosen explicitly. PostgreSQL has heap tables with all indexes non-clustered; the CLUSTER command reorders the table once but does not maintain it. Clustered ordering makes range scans on the key very fast because rows are adjacent, but random primary keys such as UUIDs cause page splits and fragmentation on inserts. I prefer compact, monotonically increasing keys for clustered storage, and I use secondary indexes, ideally covering ones, for other access paths.

Likely follow-ups

- Why do long primary keys hurt InnoDB?
- Why are random UUID keys a problem?

🚧 Clustered index

Clustered	Non-clustered
Rows stored in index order	Separate structure + pointer
One per table	Many per table

A clustered index defines the physical row order (the table is the index); non-clustered indexes are separate structures pointing back to rows.

```
CREATE CLUSTERED INDEX ix ON orders(id); -- SQL Server
```

```
CLUSTER orders USING orders_pkey; -- PostgreSQL one-time
```

Interview tip: InnoDB primary key is the clustered index.



Start here (official docs)

- [PostgreSQL documentation](#)

Q47

What is index selectivity and cardinality, and why do they matter?

Indexing

🗣 Spoken answer · ~64 s

Cardinality is the number of distinct values in a column, and selectivity is how few rows a typical value returns relative to the table. An index on a high-selectivity column, like an email or order ID, lets the database jump to a handful of rows. An index on a boolean or a status column with three values returns a large part of the table, so the planner often prefers a sequential scan because random index lookups would cost more. The optimiser relies on statistics, histograms and most common values to estimate this, so stale statistics lead to bad plans; I run `ANALYZE` after bulk changes. Low-cardinality columns can still be useful as the leading column of a composite index paired with a selective one, or in a partial index that covers only the rare value, such as unprocessed rows. I judge an index by its effect on real query plans, not by theory.

Likely follow-ups

- When is a low-cardinality index useful?
- How do statistics affect the plan?

Selectivity is the fraction of rows a value matches; high-cardinality columns make better index candidates because they narrow results.

```
SELECT count(DISTINCT status)::float/count(*) FROM orders;
```

```
SELECT n_distinct FROM pg_stats WHERE tablename='orders';
```

Interview tip: Low-selectivity indexes are often ignored.

 Official docs

- [Prometheus: Metric and label naming](#)
- [Use The Index, Luke!](#)

Q48

How do you find unused and duplicate indexes and safely remove them?

Indexing

 Spoken answer · ~60 s

Every index slows writes, consumes storage and memory, and adds vacuum work, so unused ones are pure cost. In PostgreSQL I query `pg_stat_user_indexes` for indexes with zero or very few scans, and in MySQL the `sys` schema shows unused indexes. Statistics reset on restart or manual reset, so I observe over a complete business cycle, including month-end or seasonal jobs, before deciding. Duplicates and redundant indexes appear when one index is a prefix of another, for example an index on column A when another covers A and B. Unique and constraint-backing indexes must stay. Before dropping, I save the definition so it can be recreated, check replicas since usage may differ there, and drop with the `concurrent` option to avoid locks. In MySQL, I can make an index invisible first to test the effect without deleting it. Then I monitor query performance after the change.

Likely follow-ups

- Why watch a full business cycle?
- What is an invisible index in MySQL?

Check index usage statistics over a full business cycle, find overlapping definitions, then drop in a controlled change with a rollback plan.

```
SELECT relname, indexrelname, idx_scan FROM pg_stat_user_indexes ORDER BY idx_scan;
```

```
DROP INDEX CONCURRENTLY ix_old;
```

Interview tip: Reset windows can hide seasonal use.

 Official docs

- [PostgreSQL: Indexes](#)

 Spoken answer · ~55 s

B-tree handles equality, range queries and ordering, and covers most needs. Hash indexes support equality only and are rarely preferred. GIN, the generalised inverted index, maps elements to rows and excels at containment queries on arrays, JSONB and full-text search, at the cost of slower writes. GiST supports overlap, geometry, range types and nearest-neighbour searches. SP-GiST helps with partitioned search spaces like prefixes. BRIN stores summaries of block ranges and is extremely small, ideal for huge append-only tables where values correlate with physical order, such as timestamps in a log table, but is useless if the data is randomly ordered. Choosing the right type depends on the operators in the query, so I verify with EXPLAIN that the index is used. Extensions add more, such as trigram indexes for fuzzy text matching.

Likely follow-ups

- When does BRIN work well?
- Why use GIN for JSONB?

B-tree is the default for equality and ranges; GIN suits arrays, JSONB and full text; GiST handles geometric and nearest-neighbour; BRIN is tiny for naturally ordered huge tables; hash is equality only.

```
CREATE INDEX ON docs USING gin (payload jsonb_path_ops);
```

```
CREATE INDEX ON events USING brin (created_at);
```

Interview tip: BRIN needs physical correlation.

 Official docs

- [PostgreSQL: Indexes](#)
- [Use The Index, Luke!](#)

 Spoken answer · ~61 s

An index-only scan returns results straight from the index when it contains every column the query needs, avoiding heap lookups. In PostgreSQL there is a catch from MVCC: the index does not hold row visibility information, so the executor must consult the visibility map to know whether all rows on a heap page are visible to everyone. If pages are not marked all-visible, because of recent updates or because vacuum has not run, it must still visit the heap, and the plan shows many heap fetches. To enable index-only scans I design covering indexes using the INCLUDE clause for payload columns, and I make sure autovacuum keeps up on frequently updated tables. The benefits are less I/O and better cache use, particularly for large tables, while the costs are bigger indexes and slower writes. I check the Heap Fetches figure in EXPLAIN ANALYZE to confirm the benefit.

Likely follow-ups

- What does the visibility map record?
- What does INCLUDE do?

The query is answered from the index alone without visiting the table, but only if the visibility map shows pages as all-visible.

```
CREATE INDEX ON orders (customer_id) INCLUDE (total);
```

```
EXPLAIN (ANALYZE, BUFFERS) SELECT total FROM orders WHERE customer_id=7;
```

Interview tip: VACUUM keeps the visibility map fresh.

 Start here (official docs)

- [PostgreSQL documentation](#)

 Spoken answer · ~62 s

With OFFSET, the database must produce and discard all skipped rows, so page one hundred thousand is slow and the cost grows linearly with depth. Results can also shift between requests when rows are inserted. Keyset pagination, also called the seek method, remembers the last row's sort key from the previous page and asks for rows after it using a where clause, which an index can satisfy directly, so every page costs about the same. The sort must be on a unique, indexed combination of columns, so I include a tie-breaker such as the primary key in both the order by and the comparison, using row-value comparison where supported. The trade-offs are that you cannot jump to an arbitrary page number and the cursor must be passed to clients. For APIs and infinite scroll it is the better choice; OFFSET remains acceptable for small tables or shallow pages.

Likely follow-ups

- Why do you need a unique tie-breaker?
- What do you lose with keyset pagination?

OFFSET scans and discards rows so deep pages get slower; keyset (seek) pagination uses an indexed last-seen value and stays fast.

```
SELECT * FROM items ORDER BY id LIMIT 50 OFFSET 100000;
```

```
SELECT * FROM items WHERE id > :last_id ORDER BY id LIMIT 50;
```

Interview tip: Order by a unique indexed key.



Official docs

- [REST API pagination \(Google AIP-158\)](#)

Q52 How do you speed up COUNT(*) on large tables?

Query optimization

Spoken answer · ~60 s

In MVCC databases like PostgreSQL, a count must check row visibility, so it scans the table or an index; on a table with hundreds of millions of rows that is expensive. I begin by asking whether an exact number is needed. For approximate totals such as a UI showing about 1.2 million results, the planner statistics give a cheap estimate. If an exact figure is needed, options include maintaining a counter table updated by triggers or application logic, though that adds write contention; using materialised or summary tables refreshed periodically; counting over a narrow indexed range such as the last day; or using an index-only scan on a small index with a well-vacuumed table. InnoDB also scans, though it can pick the smallest secondary index. For analytics, a columnar store or a cached aggregate is often a better design than counting on the transactional database.

Likely follow-ups

- Why is count slow in PostgreSQL?
- When are estimates acceptable?

Use estimates from statistics, maintain counters, count with index-only scans, or cache; exact counts require scanning visible rows.

```
SELECT reltuples::bigint FROM pg_class WHERE relname='orders';
```

```
SELECT count(*) FROM orders WHERE created_at >= now() - interval '1 day';
```

Interview tip: Ask whether exactness is really needed.



Start here (official docs)

- [PostgreSQL documentation](#)

Spoken answer · ~57 s

UNION combines results and removes duplicates, which requires a sort or hash over the whole result and costs time and memory; UNION ALL simply concatenates, so when I know the sets are disjoint or duplicates are acceptable I use ALL. For IN versus EXISTS, modern optimisers often rewrite both into semi-joins and produce the same plan, but the semantic differences matter. EXISTS short-circuits at the first match and handles NULLs predictably. NOT IN behaves unexpectedly when the subquery returns NULL, because the comparison becomes unknown and no rows are returned, so I prefer NOT EXISTS or an anti-join. For large lists of constants, IN with a values list or a join against a temporary table can be better. As always, I check the actual plan with EXPLAIN on realistic data instead of assuming one form is faster.

Likely follow-ups

- Why is NOT IN dangerous with NULLs?
- When does UNION ALL matter for speed?

UNION removes duplicates by sorting or hashing; UNION ALL just appends. EXISTS can stop at the first match and is often better than IN for correlated checks.

```
SELECT id FROM a UNION ALL SELECT id FROM b;
```

```
SELECT 1 FROM orders o WHERE EXISTS (SELECT 1 FROM items i WHERE i.order_id=o.id);
```

Interview tip: Default to UNION ALL if duplicates are impossible.

Start here (official docs)

- [PostgreSQL documentation](#)

 Spoken answer · ~60 s

A pattern like name LIKE 'abc%' can use a B-tree index because it is a range scan on a prefix, although in PostgreSQL with non-C collations I need the text_pattern_ops operator class. A pattern with a leading wildcard, like '%abc', cannot use a normal index because there is no ordered starting point, so the database scans the entire table. For infix matching and fuzzy search, PostgreSQL's pg_trgm extension with a GIN or GiST index breaks text into trigrams and accelerates LIKE, ILIKE and similarity queries. For natural-language search with ranking, stemming and stop words, I use built-in full-text search with a tsvector column and a GIN index, and keep it updated with a generated column or trigger. MySQL offers FULLTEXT indexes. When requirements include relevance tuning, typo tolerance and facets at scale, a dedicated search engine such as Elasticsearch or OpenSearch is the right tool.

Likely follow-ups

- Why can't '%abc' use a B-tree?
- What is a trigram index?

Leading wildcards cannot use B-tree indexes; use trigram indexes, full-text search indexes or a search engine.

```
CREATE EXTENSION pg_trgm; CREATE INDEX ON users USING gin (name gin_trgm_ops);
```

```
SELECT * FROM docs WHERE tsv @@ to_tsquery('english','deploy & rollback');
```

Interview tip: Prefix patterns can use B-tree with the right operator class.

 Official docs

- [PostgreSQL: Indexes](#)

 Spoken answer · ~61 s

Row-by-row inserts from an application pay a network round trip, a parse and often a commit for each row. The fastest approach in PostgreSQL is COPY, which streams data in bulk; MySQL has LOAD DATA. Where those are not possible, multi-row inserts or prepared statements batched in a single transaction cut overhead dramatically. For very large loads, I load into an unlogged or staging table, then transform and merge into the target, and consider dropping or deferring secondary indexes and constraints until after the load, rebuilding afterwards, which is faster than maintaining them for each row. Commit in sensible chunks, perhaps tens of thousands of rows, to limit lock time, transaction log size and replication lag. For bulk updates, I batch by primary key ranges with short transactions to avoid long locks and bloat. After large loads I run ANALYZE so the planner has fresh statistics.

Likely follow-ups

- Why is COPY faster than inserts?
- Why commit in chunks?

Batch rows, use COPY or multi-row inserts, disable unnecessary indexes temporarily, commit in chunks and avoid per-row round trips.

```
COPY orders FROM '/data/orders.csv' CSV;
```

```
INSERT INTO t VALUES (1, 'a'), (2, 'b'), (3, 'c');
```

Interview tip: Round trips dominate small inserts.

 Start here (official docs)

- [PostgreSQL documentation](#)

 Spoken answer · ~58 s

A prepared statement is parsed once and executed many times with bound parameters. This saves parse and sometimes planning time, and crucially separates code from data, which is the primary defence against SQL injection, so I always parameterise queries rather than concatenating strings. The performance subtlety is plan reuse. After several executions, PostgreSQL may switch to a generic plan that does not depend on the parameter value; when data is skewed, for example one customer has millions of rows and others few, the generic plan may be poor for some values. The setting `plan_cache_mode` lets me force custom plans per execution. With connection poolers in transaction mode, session-level prepared statements can break unless the pooler supports protocol-level prepared statements. In the application, ORMs and drivers often prepare automatically, so I check their configuration when diagnosing odd plan behaviour.

Likely follow-ups

- What is a generic plan?
- Why can transaction pooling break prepared statements?

Prepared statements parse once and bind parameters, saving planning time and preventing SQL injection; generic plans can misestimate skewed data.

```
PREPARE q(int) AS SELECT * FROM orders WHERE customer_id=$1; EXECUTE q(42);
```

```
SET plan_cache_mode = force_custom_plan;
```

Interview tip: Always parameterise, never concatenate.

 Start here (official docs)

- [PostgreSQL documentation](#)

 Spoken answer · ~58 s

Multi-version concurrency control lets readers and writers avoid blocking each other, because each transaction sees a snapshot. In PostgreSQL an update writes a new row version and marks the old one as dead for later transactions, and a delete only marks it. Dead versions stay in the table until vacuum can prove that no active transaction still needs them. If updates are frequent and vacuum cannot keep up, or a long-running transaction or a lagging replica with feedback holds back the cleanup horizon, the table and its indexes accumulate dead space, called bloat, which makes scans slower and wastes cache. Mitigation includes tuning autovacuum to be more aggressive for hot tables, avoiding long idle-in-transaction sessions, using fillfactor to enable heap-only tuple updates, and occasionally rebuilding with `pg_repack` or a concurrent reindex. Monitoring dead tuple ratios helps catch it early.

Likely follow-ups

- What blocks vacuum from cleaning?
- What is a HOT update?

Each update creates a new row version; old versions remain until vacuum removes them, so heavy updates need vacuum capacity.

```
SELECT n_dead_tup, n_live_tup FROM pg_stat_user_tables WHERE relname='orders';
```

```
SELECT xmin, xmax, * FROM orders LIMIT 1;
```

Interview tip: Long transactions block cleanup.

 Official docs

- [PostgreSQL: Routine vacuuming](#)
- [PostgreSQL: Concurrency control \(MVCC\)](#)

 Spoken answer · ~58 s

PostgreSQL uses 32-bit transaction identifiers, compared in a circular fashion, so each row's age relative to the current transaction must stay below about two billion. Vacuum freezes old rows, marking them as visible to all future transactions. If freezing does not happen, the system eventually stops accepting writes to protect data from appearing to vanish, which is a severe outage that requires single-user-mode vacuuming. Causes include autovacuum being disabled or too slow for large tables, long-running transactions or abandoned replication slots holding the horizon back, and prepared transactions left open. Prevention is to monitor the age of `datfrozenxid` and per-table `relfrozenxid` and alert at moderate thresholds, keep autovacuum healthy with sufficient workers and cost limits, kill stuck old transactions and drop unused replication slots. Large tables may need tuned freeze settings and scheduled manual vacuums in quiet hours.

Likely follow-ups

- What do abandoned replication slots cause?
- What does freezing do?

Transaction IDs are 32-bit and wrap; vacuum freezes old rows to avoid data becoming invisible, and failing to vacuum can force the database into read-only shutdown.

```
SELECT datname, age(datfrozenxid) FROM pg_database ORDER BY 2 DESC;
```

```
VACUUM (FREEZE, VERBOSE) big_table;
```

Interview tip: Alert well before 1.5 billion.

 Official docs

- [PostgreSQL: Routine vacuuming](#)

Spoken answer · ~59 s

The default trigger is a fraction of the table, twenty percent dead rows, which for a table with a billion rows means hundreds of millions of dead tuples before vacuum starts. For large or hot tables I set per-table storage parameters: a much smaller scale factor and a fixed threshold, so vacuum triggers earlier and does smaller, steadier work. If vacuum is running but too slowly, I raise the cost limit or reduce the cost delay so it does more work per cycle, and increase the number of autovacuum workers and maintenance memory when many tables need attention. I watch `pg_stat_user_tables` for dead tuple counts, last autovacuum times, and `pg_stat_progress_vacuum` for running jobs, and check that long transactions are not preventing cleanup. For insert-only tables, the insert threshold setting matters for freezing and visibility map updates. Tuning is iterative, guided by metrics.

Likely follow-ups

- Why is the default scale factor bad for big tables?
- How do you see vacuum progress?

Lower scale factors per table, raise cost limit or workers, and monitor dead tuples and vacuum duration.

```
ALTER TABLE orders SET (autovacuum_vacuum_scale_factor = 0.01, autovacuum_vacuum_cost_limit = 2000);
```

```
SELECT * FROM pg_stat_progress_vacuum;
```

Interview tip: Defaults suit small tables, not huge ones.

 Official docs

- [PostgreSQL: Routine vacuuming](#)

What is table and index bloat repair with `pg_repack`, `REINDEX CONCURRENTLY` and `VACUUM FULL`?

Spoken answer · ~58 s

Plain `VACUUM` marks space reusable but rarely returns it to the operating system, so bloated tables stay large. `VACUUM FULL` rewrites the entire table into a compact file but holds an exclusive lock for the duration, blocking reads and writes, which is unacceptable on live production tables. Alternatives work online: `REINDEX CONCURRENTLY` rebuilds an index while allowing normal operations, taking extra disk space and time; `pg_repack` creates a new copy of the table with triggers capturing changes, then swaps it in briefly under lock. These need free disk space about the size of the table and can add load and replication lag, so I run them in quiet periods and watch. Before repair, I confirm that bloat is real, using estimation queries or `pgstattuple`, and fix the cause, such as stuck transactions or insufficient autovacuum, otherwise it returns.

Likely follow-ups

- Why avoid `VACUUM FULL` in production?
- How do you measure bloat?

`VACUUM FULL` rewrites the table with an exclusive lock; `pg_repack` and `REINDEX CONCURRENTLY` rebuild online with minimal locking.

```
REINDEX INDEX CONCURRENTLY ix_orders_created;
```

```
pg_repack -d appdb -t orders
```

Interview tip: Avoid `VACUUM FULL` on live systems.

Official docs

- [PostgreSQL: Routine vacuuming](#)

 Spoken answer · ~64 s

EXPLAIN with ANALYZE executes the query and shows actual timings and row counts, and the BUFFERS option adds how many pages were found in shared buffers, as hits, versus read from the operating system or disk. A plan node with a high read count shows where I/O is spent; with track_io_timing enabled I also see time waiting on I/O. I first compare estimated rows with actual rows at each node: a large mismatch means stale or insufficient statistics, correlated columns or a misestimate, which leads the planner into bad join orders or scan choices. I look for sequential scans on large tables where an index should work, nested loops with huge outer row counts, sorts or hashes spilling to disk, shown as temp blocks, and repeated loops. Because ANALYZE runs the statement, I wrap data-changing queries in a transaction that I roll back. Fixes include indexes, statistics targets, query rewrites and work memory changes.

Likely follow-ups

- What do shared hit and read mean?
- Why wrap DML in a rollback?

Buffers show shared hits versus reads; many reads mean cache misses and disk I/O. Compare estimated and actual rows to spot bad statistics.

```
EXPLAIN (ANALYZE, BUFFERS, VERBOSE) SELECT ...;
```

```
SET track_io_timing = on;
```

Interview tip: Actual versus estimated rows is the first check.

 Official docs

- [PostgreSQL: Performance tips](#)

 Spoken answer · ~60 s

InnoDB writes changes to the redo log before the data pages are flushed, so after a crash it can replay committed changes; the setting `innodb_flush_log_at_trx_commit` decides whether each commit forces the log to disk, with the default of one giving full durability, and zero or two trading some durability for speed. The undo log stores previous versions of rows, which lets transactions roll back and lets consistent reads see older snapshots, implementing MVCC; long transactions make undo grow and slow purge. The doublewrite buffer protects against partial page writes during a crash by writing pages first to a safe area before their final location. Understanding these explains common behaviours: commit latency is dominated by log fsync, large transactions inflate undo and redo, and a redo log that is too small causes aggressive flushing and stalls. Tuning involves log file size, flush method and I/O capacity.

Likely follow-ups

- What does `innodb_flush_log_at_trx_commit=1` mean?
- Why do long transactions hurt?

Redo ensures durability, undo supports rollback and MVCC, and doublewrite protects against torn pages.

```
SHOW ENGINE INNODB STATUS\G
```

```
SHOW VARIABLES LIKE 'innodb_flush_log_at_trx_commit';
```

Interview tip: Flush setting trades durability for speed.

**Official docs**

- [MySQL: InnoDB buffer pool](#)

How do gap locks and next-key locks cause deadlocks in InnoDB at REPEATABLE READ?

Spoken answer · ~61 s

At the default REPEATABLE READ level, InnoDB uses next-key locks, a record lock plus a lock on the gap before it, when scanning an index range with locking reads or updates, to prevent phantom rows. Two transactions can each lock a gap and then both try to insert into it; because gap locks do not conflict with each other but do block inserts, they wait on each other and deadlock. These situations appear around unique-key checks, range updates and delete-then-insert patterns. I diagnose with the latest detected deadlock section of the engine status or the deadlock logging option, which shows the statements and locks. Mitigations include keeping transactions short, accessing rows in a consistent order, using more selective indexes so fewer gaps are locked, switching to READ COMMITTED where phantom protection is not required, and adding retry logic in the application, since deadlocks are normal and recoverable.

Likely follow-ups

- What does a gap lock protect against?
- Why should apps retry on deadlock?

Range scans lock gaps between index records to prevent phantoms; concurrent inserts into the same gap can deadlock.

```
SELECT * FROM t WHERE id BETWEEN 10 AND 20 FOR UPDATE;
```

```
SHOW ENGINE INNODB STATUS\G -- LATEST DETECTED DEADLOCK
```

Interview tip: READ COMMITTED reduces gap locking.

Official docs

- [PostgreSQL: Explicit locking](#)
- [MySQL: InnoDB buffer pool](#)
- [MySQL: InnoDB locking](#)

 Spoken answer · ~56 s

Statement-based replication logs SQL statements and is compact, but nondeterministic functions and some statements can produce different results on replicas. Row-based replication logs actual row changes, which is deterministic and safe, with larger logs for bulk updates. Mixed switches automatically. Today row format is the standard. GTIDs assign every transaction a globally unique identifier, so a replica tracks which transactions it has executed instead of binary log file names and positions. This makes failover and topology changes much easier: when promoting a replica or repointing others, auto-position figures out where to continue, avoiding manual position arithmetic and the errors it invites. GTIDs also help detect errant transactions that exist on a replica but not the source. I combine GTID with semi-synchronous or group replication for stronger durability and monitor replica lag and errors continuously.

Likely follow-ups

- Why is row-based replication safer?
- What is an errant transaction?

Row-based replication is the safest; GTIDs give each transaction a global ID, making failover and replica repointing much simpler.

```
SHOW VARIABLES LIKE 'binlog_format';
```

```
CHANGE REPLICATION SOURCE TO SOURCE_AUTO_POSITION=1;
```

Interview tip: Use ROW format and GTIDs.

 Official docs

- [PostgreSQL: High availability, load balancing, replication](#)
- [MySQL: Replication](#)

How do you perform online schema changes in MySQL with gh-ost or pt-online-schema-change?

Spoken answer · ~58 s

Altering a large table can lock it or take hours. Online schema change tools avoid that. They create a copy of the table with the new structure, copy rows in small chunks to limit load, and keep the copy in sync with ongoing changes, pt-online-schema-change using triggers and gh-ost reading the binary log, which avoids trigger overhead and allows throttling and pausing. At the end they swap the tables with a quick atomic rename. I run a dry-run first, check disk space for the copy, set throttling based on replica lag and server load, and schedule the cutover at a quiet time. Considerations include foreign keys, which complicate things, tables without primary keys, and a plan to abort cleanly. Newer MySQL versions support many in-place and instant alters natively, so I check the algorithm available before reaching for a tool.

Likely follow-ups

- How does gh-ost avoid triggers?
- When is a native instant ALTER enough?

These tools create a shadow table, copy data in chunks while capturing changes, and swap tables atomically with minimal blocking.

```
gh-ost --alter='ADD COLUMN notes TEXT' --database=app --table=orders --execute
```

```
pt-online-schema-change --alter 'ADD INDEX ix(c)' D=app,t=orders --execute
```

Interview tip: Always test with dry-run first.

Official docs

- [gh-ost: GitHub's online schema migration](#)

 Spoken answer · ~57 s

Lag appears when the replica cannot apply changes as fast as the source produces them. Common causes are single-threaded apply of a very write-heavy workload, which I address with parallel replication using multiple workers; large transactions such as bulk deletes or alters that take as long to apply as to run and block everything behind them, which I fix by batching work into smaller transactions; replicas with slower disks or less memory than the source; heavy analytical reads on the replica competing for resources; and tables lacking a primary key or suitable index, which forces full scans for each row event under row-based replication. I measure lag with replica status and heartbeat tables, since seconds-behind-source can be misleading. Applications that read from replicas must tolerate lag or use read-your-writes routing, and I alert on lag thresholds.

Likely follow-ups

- Why is a missing primary key harmful for replicas?
- How do you handle read-after-write needs?

Single-threaded apply, large transactions, heavy replica reads, I/O limits and missing indexes on replicas; use parallel replication and smaller transactions.

```
SHOW REPLICA STATUS\G
```

```
SET GLOBAL replica_parallel_workers=8;
```

Interview tip: Large transactions serialise apply.

 Official docs

- [PostgreSQL: High availability, load balancing, replication](#)
- [MySQL: Replication](#)

Spoken answer · ~61 s

A database can only make progress on as many queries as it has CPU cores and I/O capacity; extra concurrent connections just add context switching, lock contention and memory use, so throughput drops when the pool is too large. A common starting formula is about twice the number of cores plus the number of effective disks, then I load test and adjust. The total across all application instances matters, so with many pods each configured with a pool of twenty, the database sees their sum. Therefore I size per-instance pools from a global budget, and use a server-side pooler such as PgBouncer or ProxySQL to multiplex many clients onto a few database connections. I set timeouts for acquiring connections, idle timeout, and maximum lifetime to recycle connections, and monitor wait time for connections in the application, which signals real saturation better than the pool size does.

Likely follow-ups

- Why can a bigger pool be slower?
- How does the total across instances matter?

Pool sizing

Pool size should match what the database can run concurrently, often cores times two plus disks, not the number of clients; more connections increase contention.

```
SELECT count(*), state FROM pg_stat_activity GROUP BY state;
```

```
pool_size = ((core_count * 2) + effective_spindle_count)
```

Interview tip: Smaller pools are often faster.

Official docs

- [PgBouncer](#)

 Spoken answer · ~58 s

PgBouncer sits between applications and PostgreSQL, accepting many client connections and mapping them to a small number of server connections. In session mode a client keeps a server connection until it disconnects, which is compatible with all features but gives limited multiplexing. In transaction mode the server connection is returned to the pool when the transaction ends, so thousands of mostly idle clients can share a few dozen server connections, which is the usual choice for web applications. The cost is that anything relying on session state does not work reliably: session-level advisory locks, temporary tables, SET commands that persist, LISTEN and NOTIFY, and in older versions protocol-level prepared statements. Statement mode releases after every statement and forbids multi-statement transactions. I monitor with SHOW POOLS and SHOW STATS, watch client wait times, and size default_pool_size to the database's capacity.

Likely follow-ups

- What breaks in transaction mode?
- How do you monitor PgBouncer?

Session mode holds a server connection per client; transaction mode releases after each transaction (best multiplexing, restricts session features); statement mode is rarely used.

```
pool_mode = transaction
default_pool_size = 20
max_client_conn = 2000
```

```
SHOW POOLS;
```

Interview tip: Transaction mode breaks session state.

 Official docs

- [PgBouncer](#)

 Spoken answer · ~58 s

A session that issued BEGIN and then stopped, perhaps because the application crashed, waited on an external call, or forgot to commit, stays idle in transaction. It keeps its locks, blocking other writers, and holds an old snapshot, which prevents vacuum from removing dead rows anywhere in the database, leading to bloat and eventually wraparound risk. I find these by querying pg_stat_activity for that state ordered by transaction age, and by alerting on the maximum age. The preventive control is idle_in_transaction_session_timeout, which terminates such sessions after a limit; I set a value appropriate to the application, maybe tens of seconds. In the application, I make transactions short, avoid making HTTP calls or waiting on users inside them, and make sure the framework releases connections in finally blocks. For immediate relief I terminate the oldest offender after confirming it is safe.

Likely follow-ups

- Why does it block vacuum?
- How do you prevent it?

They hold locks and block vacuum; find them in pg_stat_activity and set idle_in_transaction_session_timeout.

```
SELECT pid, now()-xact_start AS age, query FROM pg_stat_activity WHERE state='idle in transaction' ORDER BY age DESC;
```

```
ALTER SYSTEM SET idle_in_transaction_session_timeout='60s';
```

Interview tip: A forgotten BEGIN can bloat the whole database.

 Start here (official docs)

- [PostgreSQL documentation](#)

 Spoken answer · ~60 s

When a database restarts or fails over, every application instance tries to reconnect at once, often with identical retry intervals, creating a thundering herd that can exhaust connection limits and CPU, slowing recovery or causing another failure. I mitigate it in layers. Clients retry with exponential backoff and random jitter, and their pools have bounded sizes and acquisition timeouts. A server-side pooler or proxy absorbs the reconnection burst by queuing clients while keeping the number of real server connections low. The database's maximum connections limit and reserved connections for administrators ensure I can still log in. Applications should fail fast and shed load or serve degraded responses instead of piling up requests. After failover, DNS or proxy endpoints should point to the new primary quickly, and caches and prepared statements should be re-established gradually. I rehearse failovers to see how the whole system behaves.

Likely follow-ups

- Why add jitter to retries?
- What reserved connections are for?

Use jittered retries, connection limits, pooler queues and gradual ramp-up so clients do not overwhelm the recovering database.

```
max_connections = 500
```

```
retry delay = base * 2^n + random(0, base)
```

Interview tip: Thundering herd needs jitter.

**Start here (official docs)**

- [PostgreSQL documentation](#)

 Spoken answer · ~56 s

In PostgreSQL each client connection is a separate backend process with its own memory for caches and state, and queries can allocate work_mem for each sort or hash node, so worst-case memory is connections times work_mem times operations per query. Raising max_connections to several thousand without thought can cause out-of-memory kills, heavy context switching and slower snapshot computation. Typical advice is to keep max_connections moderate, a few hundred, and introduce a pooler for scale. I also keep a few connections reserved for superusers. If applications really need many concurrent sessions, I check whether they hold connections idle, which a pooler can fix. MySQL uses threads with lighter per-connection overhead but similar principles apply with per-thread buffers. When capacity is an issue, adding read replicas or vertical scale is better than allowing more simultaneous connections.

Likely follow-ups

- How is worst-case memory computed?
- Why use a pooler instead?

Each connection uses memory for backend state and work_mem per operation; raising limits without pooling risks out-of-memory and slowdowns.

```
SHOW max_connections;
```

```
SELECT pg_size_pretty(pg_backend_memory_contexts_total) -- per-backend memory
```

Interview tip: Pool instead of raising the limit.

 Official docs

- [PgBouncer](#)

 Spoken answer · ~55 s

A replication slot guarantees that the primary keeps all WAL segments a consumer, such as a standby or a logical decoding client like a CDC tool, has not yet received. That prevents the replica from falling irrecoverably behind. The danger is that if the consumer disconnects or stops, the slot stays and the primary keeps accumulating WAL indefinitely, which can fill the disk and take the primary down. It can also hold back the cleanup horizon for logical slots. I monitor each slot's active status and retained WAL size, alert when they grow, and drop slots that belong to decommissioned consumers. Newer versions allow `max_slot_wal_keep_size` to cap retention, trading safety for the consumer by invalidating the slot rather than endangering the primary. Slots should be named and documented so owners are known.

Likely follow-ups

- How can a slot fill the disk?
- What does `max_slot_wal_keep_size` do?

A slot makes the primary retain WAL until the consumer confirms it; an inactive slot makes WAL pile up and can fill the disk.

```
SELECT slot_name, active, pg_size_pretty(pg_wal_lsn_diff(pg_current_wal_lsn(),
restart_lsn)) AS lag FROM pg_replication_slots;
```

```
SELECT pg_drop_replication_slot('old_slot');
```

Interview tip: Alert on retained WAL size.

 Official docs

- [PostgreSQL: High availability, load balancing, replication](#)

Spoken answer · ~60 s

Failover has to detect that the primary is really down, which is hard because network partitions can look like failures. A coordination service or a tool such as Patroni uses a distributed lock so only one node can be primary. Steps are: ensure the old primary is fenced so it cannot accept writes, promote the most up-to-date replica, repoint clients through a virtual IP, DNS, proxy or service endpoint, and reconfigure the other replicas to follow the new primary. Risks include split brain, where two nodes accept writes and data diverges; data loss with asynchronous replication, as committed transactions not yet shipped are lost, which synchronous replication or quorum commit reduces at a latency cost; and application behaviour, like cached connections or DNS TTLs delaying cutover. Afterwards the old primary must be rebuilt or rewound. I rehearse failovers regularly and measure recovery time.

Likely follow-ups

- What is split brain?
- How does synchronous replication help?

Zero-downtime failover

Detect failure reliably, fence the old primary, promote the most caught-up replica, redirect traffic and rebuild; risks are split brain and data loss with async replication.

```
pg_ctl promote -D /var/lib/postgresql/data
```

```
patronictl failover cluster
```

Interview tip: Fence the old primary.

Start here (official docs)

- [PostgreSQL documentation](#)

 Spoken answer · ~60 s

Replicas offload read traffic but reflect the primary with a delay, so correctness depends on which reads can tolerate staleness. Reports, search pages and dashboards usually can; a user who just changed a setting and then reloads the page usually cannot. Strategies include routing a user's reads to the primary for a short window after they write, tracking the log position of the last write and only reading from replicas that have caught up to it, or using sticky sessions. Monitoring replica lag in time and bytes lets the router stop sending traffic to replicas beyond a threshold and fall back to the primary. Proxies like ProxySQL or application-level routing implement this. Also consider that replicas can have query conflicts with replay, and that heavy analytical queries on a replica can cause lag, so I separate reporting replicas from those serving user traffic.

Likely follow-ups

- What is read-your-writes?
- Why separate analytics replicas?

Send only lag-tolerant reads to replicas, use read-your-writes strategies for recent writers, and monitor lag with fallbacks to the primary.

```
SELECT now() - pg_last_xact_replay_timestamp() AS lag;
```

```
SELECT pg_wal_lsn_diff(pg_current_wal_lsn(), replay_lsn) FROM pg_stat_replication;
```

Interview tip: Not all reads are equal.

 Official docs

- [PostgreSQL: High availability, load balancing, replication](#)

What is the difference between logical and physical backups and when would you use each?

 Spoken answer · ~57 s

A physical backup copies the database's files, as a base backup, together with the write-ahead log, enabling fast restore of the whole cluster and point-in-time recovery to a chosen moment. It is version- and architecture-specific. A logical backup exports schema and data as SQL or a custom format, like `pg_dump` or `mysqldump`; it is portable across versions and platforms, supports restoring individual tables, and is useful for migrations and upgrades, but is slower to restore on large databases and gives a single point in time. In production I use physical backups with continuous WAL or binary log archiving as the primary disaster recovery method, using tools like `pgBackRest` or managed snapshots, and add periodic logical dumps for portability and small-scope recovery. All backups need encryption, offsite copies and regularly tested restores that measure real recovery time.

Likely follow-ups

- What does PITR need?
- Why test restores?

Physical backups copy data files and WAL for fast full restore and PITR; logical backups dump SQL or rows, which are portable but slower.

```
pg_basebackup -D /backup -X stream -P
```

```
pg_dump -Fc appdb > appdb.dump
```

Interview tip: Test the restore, not just the backup.

Start here (official docs)

- [PostgreSQL documentation](https://www.postgresql.org/docs/)

What is change data capture and how do logical decoding or binlog readers support it?

Spoken answer · ~56 s

Change data capture reads the database's transaction log to emit every insert, update and delete as an ordered event stream. Instead of polling tables or having the application write to both the database and a message queue, which risks inconsistency, CDC derives events from the committed log. In PostgreSQL, logical decoding exposes changes through a replication slot, and tools like Debezium turn them into messages on Kafka; MySQL tools read the binary log. Use cases are keeping search indexes and caches in sync, feeding analytics warehouses, building event-driven integrations and migrations with minimal downtime. Operational concerns include the slot retaining WAL if the consumer stalls, schema evolution needing coordinated handling, ordering and exactly-once semantics at the consumer, initial snapshots of existing data, and handling of large transactions. Monitoring consumer lag and slot size is essential.

Likely follow-ups

- Why is CDC better than dual writes?
- What are the operational risks?

Change data capture



CDC streams row-level changes from the database log to downstream systems with low overhead, avoiding polling and dual writes.

```
CREATE PUBLICATION app_pub FOR TABLE orders;
```

debezium connector reading WAL via a replication slot

Interview tip: Handle schema changes and slot lag.

Official docs

- [PostgreSQL: Logical replication](#)
- [MySQL: Replication](#)

 Spoken answer · ~60 s

Data types affect storage size, memory use, index size and speed. Smaller types mean more rows per page and better cache efficiency, so I choose integer sizes to fit the realistic range but with growth in mind, because changing an integer primary key to bigint later on a huge table is painful. I store numbers as numeric types, not strings, dates as proper date or timestamp types, and use fixed-point decimals for money rather than floating point. Join and foreign key columns must have identical types, otherwise implicit casts can prevent index usage. Text columns of unbounded size should avoid being in frequently scanned wide tables; large values are stored out of line. Enumerations or small lookup tables can replace repeated long strings. UUIDs are convenient but larger and random, so ordered variants help index locality. Good type discipline early prevents expensive migrations.

Likely follow-ups

- Why must join columns have identical types?
- Why use numeric for money?

Use the smallest suitable types, avoid text for numbers, use timestamps with time zone appropriately and consistent types for join keys.

```
ALTER TABLE events ALTER COLUMN id TYPE bigint;
```

```
CREATE TABLE t (id bigint, created_at timestamptz, amount numeric(12,2));
```

Interview tip: Mismatched types prevent index use.

 Start here (official docs)

- [PostgreSQL documentation](#)

 Spoken answer · ~60 s

Auto-increment integers are small, fast to compare, ordered so inserts append to the right edge of the B-tree, and friendly to caches. Their downsides are exposure of record counts and guessable identifiers if exposed in URLs, and coordination difficulties when generating IDs across multiple writers or shards. UUIDs can be generated anywhere without coordination, are hard to guess and simplify merging data from different systems, but at sixteen bytes they enlarge every index referencing them, and random version-four values scatter inserts across the index, causing page splits, poor cache locality and write amplification. Time-ordered variants such as UUIDv7 or ULID combine uniqueness with roughly sequential inserts and mitigate most of that cost. In clustered storage like InnoDB the effect of random keys is especially strong. I choose integers for internal single-database tables and time-ordered UUIDs when distributed generation or opaque identifiers are required.

Likely follow-ups

- Why are random UUIDs slow to insert?
- What is UUIDv7?

Integers are compact and ordered; UUIDs are globally unique and safe to generate anywhere but larger and random unless time-ordered.

```
id bigint GENERATED ALWAYS AS IDENTITY
```

```
id uuid DEFAULT gen_random_uuid()
```

Interview tip: Consider UUIDv7 or ULID for ordering.

 Start here (official docs)

- [PostgreSQL documentation](#)

 Spoken answer · ~56 s

There are three common models. A shared schema stores all tenants in the same tables with a tenant identifier column: it is the most efficient and simplest to operate, but needs strict query discipline, often enforced with row-level security, and noisy neighbours can affect each other. Schema-per-tenant gives stronger separation and per-tenant customisation but becomes hard to manage with thousands of tenants because of catalog size and migrations. Database-per-tenant gives maximum isolation and easy per-tenant backup or placement, at the highest cost. For shared schema performance, I make tenant_id the leading column of primary keys and relevant indexes so queries touch only one tenant's contiguous data, and consider partitioning or sharding by tenant for very large ones. Large tenants may be moved to dedicated infrastructure. Rate limits and resource quotas protect against noisy neighbours.

Likely follow-ups

- Why lead with tenant_id?
- How does row-level security help?

Include tenant_id in keys and indexes, choose shared schema, schema-per-tenant or database-per-tenant by isolation needs, and guard against noisy neighbours.

```
CREATE INDEX ON orders (tenant_id, created_at);
```

```
ALTER TABLE orders ENABLE ROW LEVEL SECURITY;
```

Interview tip: Lead with tenant_id in composite indexes.

 Official docs

- [Kubernetes: Multi-tenancy](#)
- [PostgreSQL: Indexes](#)

 Spoken answer · ~58 s

JSONB offers flexibility to store attributes that vary between rows or change often without schema migrations, and PostgreSQL can index it with GIN and query it with containment and path operators. But overusing it loses relational advantages: constraints, foreign keys, type checking, straightforward joins, efficient statistics for the planner, and compact storage. Updating a small field rewrites the whole JSON value. My rule is that core entities and fields used in joins, filtering, sorting or aggregation belong in proper columns, while optional metadata, third-party payloads and rarely queried attributes can live in JSONB. If a JSON field becomes hot, I promote it to a real column, possibly a generated column from the JSON with its own index. Validation can be enforced with check constraints or application schemas. The hybrid approach gives flexibility where it is valuable and rigor elsewhere.

Likely follow-ups

- What do you lose with JSONB?
- What is a generated column?

Use JSONB for sparse, evolving or document-like attributes; use relational columns for core, frequently filtered fields needing constraints and joins.

```
CREATE INDEX ON events USING gin (payload);
```

```
SELECT payload->>'plan' FROM accounts WHERE payload @> '{"active": true}';
```

Interview tip: Promote hot JSON fields to columns.

 Start here (official docs)

- [PostgreSQL documentation](#)

 Spoken answer · ~58 s

Foreign keys enforce that referenced rows exist, preventing orphans, which is far cheaper than discovering corrupt data later. The cost is that inserts and updates to the child table must verify the parent, and deletes or key updates on the parent must check or cascade to children. PostgreSQL does not automatically index the referencing column, so without an index those parent-side checks scan the child table and can cause locking and slow deletes; I create indexes on foreign key columns. On extremely high-throughput systems, some teams drop foreign keys for sharded or very large tables and enforce integrity in application code, accepting the risk; in distributed databases cross-shard constraints may not be available. My default is to keep them, because integrity bugs are expensive, and remove them only where measurements show a real problem and compensating controls exist.

Likely follow-ups

- Why index the referencing column?
- When might you drop foreign keys?

They guarantee integrity but add checks on writes and need indexes on the referencing column; use them by default and remove only with a clear reason.

```
CREATE INDEX ON order_items (order_id);
```

```
ALTER TABLE order_items ADD CONSTRAINT fk FOREIGN KEY (order_id) REFERENCES orders(id) ON DELETE CASCADE;
```

Interview tip: Index the child column.

 Start here (official docs)

- [PostgreSQL documentation](#)

 Spoken answer · ~61 s

DynamoDB distributes items across partitions by hashing the partition key, and each partition has throughput limits. If many requests target the same key value, for example a popular product or a date used as the key for all current writes, that partition throttles even if the table has spare capacity overall. Avoiding hot partitions starts with choosing a high-cardinality partition key that spreads access evenly, like user or device identifiers. For unavoidable hot keys, I use write sharding, appending a random or calculated suffix to the key and reading with a scatter-gather, or aggregating in a buffer before writing. Adaptive capacity and on-demand mode help absorb uneven loads but are not a fix for extreme skew. Caching hot reads in DAX or a cache layer also helps. Data modelling in DynamoDB begins with the access patterns, then keys and secondary indexes are designed to serve them.

Likely follow-ups

- What is write sharding?
- Why start from access patterns?

Items are spread by partition key hash; uneven access concentrates load on one partition. Use high-cardinality keys, write sharding and adaptive capacity.

```
pk = f"{user_id}#{random.randint(0,9)}"
```

```
aws dynamodb describe-table --table-name events
```

Interview tip: Design keys from access patterns first.

 Official docs

- [PostgreSQL: Table partitioning](#)
- [Microsoft: Sharding pattern](#)
- [Amazon DynamoDB Developer Guide](#)

 Spoken answer · ~58 s

Once Redis reaches maxmemory, its eviction policy decides what happens. Noeviction rejects writes with errors, appropriate when Redis is a primary data store and losing keys is unacceptable. Allkeys-lru evicts the least recently used keys among all keys, a good default for a cache, and allkeys-lfu favours frequency of use. The volatile policies only consider keys with an expiry set, so if few keys have TTLs they may find nothing to evict and fail writes. Volatile-ttl evicts those expiring soonest. Random policies exist but are rarely best. I choose based on role: cache means allkeys-lru or lfu with TTLs on entries, while queues or session stores need careful capacity planning and noeviction with alerts. I monitor memory usage, evicted keys, hit ratio and fragmentation, and size maxmemory below the machine's RAM to leave headroom for forks and buffers.

Likely follow-ups

- Why can volatile policies fail?
- What should you monitor in Redis?

When memory is full Redis evicts by policy such as allkeys-lru, volatile-ttl or noeviction; choose by whether the data is a cache or a store.

```
CONFIG SET maxmemory 2gb
```

```
CONFIG SET maxmemory-policy allkeys-lru
```

Interview tip: Never use noeviction for a pure cache.

 Official docs

- [Kubernetes: Node-pressure eviction](#)
- [Redis: Key eviction](#)
- [Redis documentation](#)

 Spoken answer · ~60 s

A cache stampede occurs when a popular key expires and many requests simultaneously miss and hit the database to rebuild it. Defences include a single-flight lock so only one request recomputes while others wait or serve the stale value, adding random jitter to expiry times so keys do not all expire together, and refreshing early or serving stale data while revalidating in the background. For consistency, the common pattern is cache-aside: read from the cache, on a miss load from the database and populate; on writes update the database and then delete the cache entry, which is safer than updating it because of races. There is always a short window of inconsistency, so I define acceptable staleness, set TTLs as a safety net, and for strict needs use write-through or event-based invalidation from change data capture. I monitor hit ratio and database load.

Likely follow-ups

- What is cache-aside?
- Why delete instead of update on write?

Use request coalescing, jittered TTLs, stale-while-revalidate and explicit invalidation on writes; accept eventual consistency where possible.

```
SET lock:key 1 NX EX 10
```

```
TTL = base + random(0, 60)
```

Interview tip: Two hard problems: invalidation and naming.

 Official docs

- [Google SRE Book: Addressing Cascading Failures](#)
- [AWS Builders' Library: Caching challenges and strategies](#)

 Spoken answer · ~59 s

MongoDB queries without a supporting index perform collection scans, so I create indexes matching query patterns. For compound indexes the guideline is equality fields first, then sort fields, then range fields, so the index serves filtering and ordering together. Explain with execution statistics shows documents examined versus returned; a large ratio indicates inefficiency. Covered queries, where the projection is satisfied by the index alone, avoid fetching documents. In aggregation pipelines, stage order matters: put match and sort stages early so they can use indexes, project only required fields, and limit before expensive stages like lookup or group, because later stages cannot use indexes in most cases and large groups may spill to disk. I avoid unbounded array growth in documents, watch the working set against RAM, and use the profiler to find slow operations. Shard keys need high cardinality and even distribution.

Likely follow-ups

- What is the equality-sort-range rule?
- What is a covered query?

Compound indexes follow the equality-sort-range rule; the pipeline should filter and project early so stages can use indexes.

```
db.orders.createIndex({customerId:1, createdAt:-1})
```

```
db.orders.find({customerId:7}).explain('executionStats')
```

Interview tip: Equality, sort, range.

 Official docs

- [PostgreSQL: Indexes](#)
- [MongoDB manual](#)

 Spoken answer · ~60 s

Distributed stores replicate data across nodes, and consistency describes what a read can see after a write. Eventual consistency guarantees replicas converge but a read may return stale data briefly; it offers the best availability and latency. Strong consistency ensures reads reflect the latest committed write, at the cost of higher latency and reduced availability during partitions. Many systems let me tune per operation. In quorum-based systems like Cassandra with N replicas, writing to W and reading from R replicas with R plus W greater than N guarantees overlap and so the read sees the latest write. DynamoDB offers eventually consistent reads by default and strongly consistent reads at higher cost, and transactions when needed. I choose strong consistency for things like balances and inventory, and eventual for feeds and counters, always designing around the CAP trade-offs and the business impact of stale reads.

Likely follow-ups

- What does $R + W > N$ guarantee?
- When is eventual consistency fine?

Eventual, read-your-writes, strong and tunable quorum consistency; choose per operation based on correctness needs and latency.

```
CONSISTENCY QUORUM; -- Cassandra cqlsh
```

```
aws dynamodb get-item --consistent-read
```

Interview tip: $R + W > N$ gives strong reads in quorum systems.

 Official docs

- [Jepsen: Consistency models](#)

How do you use RDS or Cloud SQL read replicas, Multi-AZ and high availability correctly?

Spoken answer · ~57 s

People often confuse high availability with read scaling. A Multi-AZ or regional HA configuration keeps a synchronously replicated standby in another zone and promotes it on failure; in the classic model the standby does not serve traffic, so it adds resilience but not capacity. Read replicas replicate asynchronously and serve read queries, which scales reads but can lag and are not a guaranteed failover target unless promoted deliberately. Many managed services now offer variants that combine both. I plan for failover time, typically one to two minutes, with DNS endpoint changes, so applications must retry and reconnect, and I test by forcing a failover. Maintenance windows, minor version upgrades and storage scaling also can cause brief interruptions. I monitor replica lag, CPU, free storage and connections, and put cross-region replicas or backups in place for disaster recovery.

Likely follow-ups

- Does Multi-AZ increase read capacity?
- How do you test failover?

HA standby is for failover durability, not read scaling; read replicas scale reads asynchronously; know the failover time and test it.

```
aws rds describe-db-instances --query 'DBInstances[].[DBInstanceIdentifier,MultiAZ]'
```

```
aws rds reboot-db-instance --db-instance-identifier db --force-failover
```

Interview tip: Standbys do not serve reads in classic Multi-AZ.

Official docs

- [Amazon RDS: Multi-AZ deployments](#)
- [Amazon RDS: Read replicas](#)
- [Cloud SQL](#)

 Spoken answer · ~60 s

Database performance on cloud storage depends on IOPS, throughput and latency limits, which vary by volume type and size. General-purpose volumes often provide a baseline proportional to size plus burst credits; once credits are exhausted, performance falls to the baseline and latency spikes, a common cause of mysterious slowdowns after a busy period. I watch metrics such as read and write latency, disk queue depth, IOPS against the provisioned limit and burst balance. If the workload sustains high I/O, I move to provisioned IOPS storage or increase volume size to raise the baseline. Memory matters too: a larger buffer cache reduces physical reads. I also check for instance-level throughput caps, which can be the limit rather than the disk, and enable storage autoscaling with a maximum to avoid running out of space. Query and index optimisation usually reduces I/O more cheaply than buying it.

Likely follow-ups

- What happens when burst credits run out?
- Why can the instance limit I/O?

Match provisioned or burst IOPS and throughput to the workload; watch queue depth, burst balance and storage autoscaling limits.

```
aws cloudwatch get-metric-statistics --metric-name DiskQueueDepth ...
```

```
gcloud sql instances patch db --storage-auto-increase
```

Interview tip: Burst credits run out.

 Start here (official docs)

- [PostgreSQL documentation](#)

What is Amazon Aurora and how does its architecture differ from standard PostgreSQL or MySQL?

Spoken answer · ~60 s

In standard engines each replica has its own copy of the data and replays the log. Aurora instead separates the database engine from a purpose-built distributed storage service that keeps six copies of the data across three availability zones and handles log application itself. Writers send log records, not full pages, to storage, reducing I/O. Read replicas attach to the same shared storage volume, so they need not replay data themselves; replica lag is typically milliseconds, and failover to a replica takes tens of seconds because there is no data copying. Storage grows automatically, and backups and snapshots are continuous and low impact. Trade-offs include cost, some vendor lock-in, and behaviour differences from stock engines, for example in extensions or tuning parameters. I still need good schema and query design, and I use cluster endpoints for writes and reader endpoints for load-balanced reads.

Likely follow-ups

- Why is Aurora replica lag so low?
- What are the trade-offs of Aurora?

Aurora separates compute from a shared, distributed storage layer replicated six ways across three zones; replicas read the same storage, giving fast failover and low replica lag.

```
aws rds describe-db-clusters
```

```
aws rds failover-db-cluster --db-cluster-identifier prod
```

Interview tip: Storage is shared, not copied per replica.

Official docs

- [Amazon Aurora User Guide](#)

 Spoken answer · ~58 s

Managed databases hide the operating system but expose rich telemetry. I track the basics: CPU, memory, free storage, connections, IOPS, throughput and latency, plus replication lag and replica health. Performance Insights or Query Insights show database load broken down by wait events and top SQL, which is the fastest way to see what is consuming time. Enhanced monitoring gives operating system metrics at fine granularity. Logs, including slow query logs and error logs, are exported to the cloud logging service for search and alerting. Alerts focus on conditions that affect users or will soon: storage nearly full, connection count near the limit, sustained high load, replica lag, failed backups, and failover events. I also review recommendations from the provider, schedule maintenance deliberately, and keep dashboards that combine database metrics with application latency so correlation is quick during incidents.

Likely follow-ups

- What does Performance Insights show?
- Which alerts matter most?

Combine instance metrics, enhanced monitoring, performance insights and logs; alert on saturation and replication, not just CPU.

```
aws pi get-resource-metrics ...
```

```
gcloud sql operations list --instance db
```

Interview tip: Use Performance Insights for top SQL.

 Start here (official docs)

- [PostgreSQL documentation](#)

How do you plan a major version upgrade of a production database with minimal downtime?

Spoken answer · ~62 s

I begin by reading release notes for breaking changes, and testing the application against the new version in a staging environment restored from a recent production backup, including extensions, drivers, query plans and performance. In-place upgrades with `pg_upgrade` in link mode are fast but need downtime for the process and a tested rollback, such as keeping the old data directory or a snapshot. For near-zero downtime I build a new cluster on the target version, keep it in sync by logical replication, or use the cloud provider's blue-green deployment feature, then switch traffic after verifying data and performance, and keep the old cluster for a while as a fallback, with reverse replication if feasible. After the upgrade I run `ANALYZE` because statistics may not carry over, check query regressions, and monitor closely. Communication, a maintenance window and a clear go or no-go checklist are part of the plan.

Likely follow-ups

- Why run `ANALYZE` after upgrading?
- How does blue-green help?

Test on a clone, check extension and application compatibility, use logical replication or blue-green deployments for near-zero downtime, and keep a rollback path.

```
pg_upgrade --check
```

```
aws rds create-blue-green-deployment ...
```

Interview tip: Rehearse with production-like data.

Start here (official docs)

- [PostgreSQL documentation](#)

How do you investigate sudden latency spikes on a database that otherwise looks healthy?

Spoken answer · ~59 s

I start by establishing the pattern: are spikes periodic, tied to deploys, batch jobs, or specific queries? Then I look at what sessions are waiting for, using wait events in `pg_stat_activity` or the Performance Schema in MySQL, which immediately separates lock waits, I/O waits and CPU. Periodic spikes often come from checkpoints flushing many dirty pages, autovacuum or autoanalyze jobs, backups, cron jobs or cache eviction after a deploy. Sudden plan changes after statistics updates or data growth can turn a fast query into a slow one; I compare plans with the `pg_stat_statements` history. Lock contention from long transactions or schema changes causes queues. I also check infrastructure: burst credits, throttling by the cloud provider, noisy neighbours, and connection storms. With the cause identified, I fix it specifically, for example spreading checkpoints, adjusting vacuum, adding an index, or limiting batch concurrency.

Likely follow-ups

- What are wait events?
- What causes periodic spikes?

Correlate with locks, checkpoints, autovacuum, cache misses, plan changes, noisy neighbours and I/O limits using wait events and time-series.

```
SELECT wait_event_type, wait_event, count(*) FROM pg_stat_activity GROUP BY 1,2 ORDER BY 3 DESC;
```

```
SELECT * FROM pg_stat_bgwriter;
```

Interview tip: Wait events show where time goes.

Start here (official docs)

- [PostgreSQL documentation](https://www.postgresql.org/docs/)

 Spoken answer · ~60 s

I enable slow query logging with a sensible threshold such as a few hundred milliseconds, and include enough context like duration, rows and the statement, being careful about sensitive literals in logs. Individual log lines are useful but aggregation is better. PostgreSQL's `pg_stat_statements` normalises queries and tracks calls, total and mean time, rows and buffer usage; `pgBadger` analyses logs into reports; MySQL has the slow log with `pt-query-digest` or the Performance Schema. I rank queries by total time consumed, calls times mean time, because a moderately slow query called a million times hurts more than one very slow report. For each top query I examine the plan, index usage, row estimates and call patterns, such as N+1 queries from the application, and fix by indexing, rewriting, caching or batching. After changes, I compare before and after metrics and keep the monitoring so regressions are visible.

Likely follow-ups

- Why rank by total time?
- What does `pg_stat_statements` normalise?

Log queries over a threshold, aggregate with tools like `pg_stat_statements`, `pgBadger` or `pt-query-digest`, and fix the top offenders by total time.

```
ALTER SYSTEM SET log_min_duration_statement = '500ms';
```

```
pt-query-digest slow.log | head -50
```

Interview tip: Rank by total time, not single slowest.

 Official docs

- [Google SRE Workbook: Implementing SLOs](#)
- [PostgreSQL: Using EXPLAIN](#)
- [PostgreSQL: pg_stat_statements](#)

 Spoken answer · ~60 s

I collect time-series for database size, growth by table, queries per second, latency percentiles, CPU, memory, I/O and connections, and look at daily and weekly peaks as well as the growth trend. Memory sizing aims for the hot working set and indexes to fit in cache, since misses go to disk. Storage planning includes data, indexes, WAL or binary logs, temporary files, backups and bloat, plus headroom for maintenance operations like rebuilding indexes that need extra space. I project growth with business forecasts, such as expected user numbers or new features, add a safety margin, and decide in advance triggers for action: when CPU stays above sixty percent at peak, or storage reaches seventy percent. Scaling options are vertical increases, read replicas, caching, archiving old data, partitioning, or sharding, each with different effort. I load test before big events and review capacity quarterly.

Likely follow-ups

- What should fit in memory?
- What triggers a scaling action?

Track trends in data size, working set, QPS and latency, project growth, and size for peak plus headroom with a plan for scaling up or out.

```
SELECT pg_size_pretty(pg_database_size(current_database()));
```

```
SELECT relname, pg_size_pretty(pg_total_relation_size(relid)) FROM  
pg_catalog.pg_statio_user_tables ORDER BY pg_total_relation_size(relid) DESC LIMIT 10;
```

Interview tip: The working set should fit in memory.

**Start here (official docs)**

- [PostgreSQL documentation](https://www.postgresql.org/docs/)

How do you archive or purge old data from huge tables without hurting production?

Spoken answer · ~60 s

Large deletes create huge transactions, lock contention, bloat, long replication lag and heavy log volume. The best design is time-based partitioning: old data lives in its own partition, which I detach and archive or drop instantly with negligible overhead. If the table is not partitioned, I delete in small batches of a few thousand rows ordered by an indexed key, committing each batch, pausing between them to let replicas keep up and vacuum clean, and monitoring lock waits and lag, with the ability to stop at any time. Before deleting, I archive to cheaper storage like object storage in a columnar format, so compliance and analytical needs are met. Afterwards, I check for bloat and reclaim space if needed. Retention rules should be defined with the business and automated, and verified by tests, so data growth is controlled continuously and not through emergency cleanups.

Likely follow-ups

- Why is dropping a partition better than deleting?
- How do you throttle batch deletes?

Partition by time and drop partitions, or delete in small batches with indexes, throttling and monitoring; archive first to cheaper storage.

```
ALTER TABLE events DETACH PARTITION events_2024_01;
```

```
DELETE FROM logs WHERE id IN (SELECT id FROM logs WHERE created_at < now()-interval '1 year' LIMIT 5000);
```

Interview tip: Dropping a partition is instant; deleting is not.

Official docs

- [PostgreSQL: Table partitioning](#)

 Spoken answer · ~64 s

When many sessions are waiting, the real problem is usually one blocker at the head of the chain. I find it using `pg_blocking_pids` or the `sys` schema's lock waits in MySQL, and look at what the blocker is doing, who owns it and how long it has run. If it is a long idle transaction or an ad hoc query, I try a gentle cancel of the query first, and terminate the session only if needed, aware that rolling back a large transaction can itself take time and keep locks. If it is a schema change waiting for a lock and queuing everything behind it, cancelling it clears the queue. Afterwards I address causes: set statement timeout, lock timeout and idle-in-transaction timeout; run DDL with a short lock timeout and retries; make application transactions short; and review the offending query for indexing or batching. A runbook with these queries speeds up the response.

Likely follow-ups

- Why cancel before terminate?
- How does `lock_timeout` help DDL?

Identify the blocker with the lock graph, assess impact, cancel or terminate it, and then fix the root cause with timeouts and query changes.

```
SELECT pid, pg_blocking_pids(pid), query FROM pg_stat_activity WHERE  
cardinality(pg_blocking_pids(pid))>0;
```

```
SELECT pg_cancel_backend(pid);
```

Interview tip: Cancel before terminate.

 Start here (official docs)

- [PostgreSQL documentation](https://www.postgresql.org/docs/)

 Spoken answer · ~59 s

A useful database load test needs three realism points. The data must be production-sized and similarly distributed, because planner choices and cache behaviour change with scale and skew; restoring a masked copy of production is ideal. The workload must reflect real query mix, concurrency and think time, ideally by replaying captured traffic or by modelling the main transactions with a tool like pgbench, sysbench or a custom harness. The environment should match production instance types, storage and configuration, and be isolated so nothing touches live data. I ramp load gradually to find the knee where latency rises, record throughput, latency percentiles, error rates, and resource usage like CPU, I/O, locks and replication lag, and repeat after tuning to confirm improvements. Also test failure scenarios such as failover and replica loss under load. Results feed capacity plans and go or no-go decisions.

Likely follow-ups

- Why does data skew matter in tests?
- What is the knee in a load curve?

Use production-like data volume and distribution, replay realistic workloads, ramp gradually, measure percentiles and resources, and isolate from production.

```
pgbench -c 50 -j 4 -T 300 -f workload.sql appdb
```

```
sysbench oltp_read_write --threads=64 run
```

Interview tip: Test with realistic data skew.

 Official docs

- [Google SRE Book: Testing for Reliability](#)