PostgreSQL

Version 18 · verified 2026-08-11

Types, indexes, plans, MVCC and vacuum, isolation, locks, JSONB and operations — the parts that are Postgres, not SQL.

Contents

Mental model

Three facts explain most of PostgreSQL's behaviour. Every connection gets its own operating-system process; every `UPDATE` writes a new row version and leaves the old one behind for vacuum to reclaim; and every query is shaped by a cost-based planner working from sampled statistics. Idle connections eating memory, a table that grows when you delete from it, and a plan that changed overnight are the same three facts seen from the outside.

Choosing types

Prefer	Over
`text`	`varchar(n)`, `character(n)`
`timestamptz`	`timestamp`
`numeric` for money	`float8`
`jsonb`	`json`
`uuid` from `uuidv7()`	Random v4 as a primary key

There is “no performance difference” between `text`, `varchar` and `varchar(n)`, and `character(n)` “is usually the slowest of the three” — a length cap is a business constraint, not an optimization. A `timestamptz` is stored as UTC and converted to the session’s `TimeZone` on output; the input’s zone “is not retained”, so the type records an instant, not a wall clock. Time-ordered UUIDs cluster far better in a B-tree than random ones; `uuidv7()` is built in from 18, and before that you generate them in the application.

```
CREATE TABLE orders (
  id          uuid PRIMARY KEY DEFAULT uuidv7(),
  customer_id bigint NOT NULL,
  total       numeric(12,2) NOT NULL,
  status      text NOT NULL DEFAULT 'open',
  placed_at   timestamptz NOT NULL DEFAULT now(),
  meta        jsonb NOT NULL DEFAULT '{}'
);
```

> **Gotcha:** `timestamp` without a time zone does not reject an offset — "any time zone indication in the input is silently ignored". A value arriving as `2026-01-01T00:00:00+09:00` is stored as midnight, and the nine hours are gone with no error.

Writing rows

Clause	Use for
`ON CONFLICT DO UPDATE`	Upsert against a unique key
`ON CONFLICT DO NOTHING`	Inserts that must be idempotent
`MERGE`	Several branches in one statement
`RETURNING`	Generated values, no second query

`DO UPDATE` requires a conflict target — the unique index or constraint to arbitrate on — while `DO NOTHING` may

omit it and consider all of them. Inside the update, the table name refers to the existing row and the special `excluded` table holds “the row proposed for insertion”, so both are available at once. `RETURNING` emits “only rows that were successfully inserted or updated”, and from 18 its expressions can be qualified with `old.` and `new.`.

```
INSERT INTO stock (sku, qty)
VALUES ('A-1', 5)
ON CONFLICT (sku) DO UPDATE
  SET qty = stock.qty + excluded.qty
RETURNING sku, qty;
```

> **Gotcha:** One statement “will not be allowed to affect any single existing row more than once”. A batch insert whose `VALUES` list repeats a key raises a cardinality violation, so deduplicate or pre-aggregate the batch client-side.

Indexes

Type	Fits
B-tree	Equality, ranges, sorting — the default
GIN	Many values per row: `jsonb`, arrays, FTS
GiST	Geometry, ranges, nearest-neighbour
BRIN	Huge tables already ordered on disk
Hash	Equality only; rarely the right answer

A B-tree serves `<`, `<=`, `=`, `>=`, `>`, `BETWEEN`, `IN`, `IS NULL` and anchored patterns like `LIKE 'foo%'`, and can also return rows already sorted. In a multicolumn B-tree the leading column drives the lookup, though 18's skip scan lets the index work when early columns are unconstrained. Three modifiers do most of the tuning work: a `WHERE` clause makes the index partial, an expression indexes a computed value, and `INCLUDE` carries extra columns so an index-only scan never touches the heap.

```
-- Partial: index only the rows you query.
CREATE INDEX ON orders (placed_at)
  WHERE status = 'open';

-- Covering: answer from the index alone.
CREATE INDEX ON orders (customer_id)
  INCLUDE (total);
```

> **Gotcha:** An expression index is used only when the query repeats the expression exactly. `WHERE lower(email) = $1` needs an index on `lower(email)`; one on `email` will not be considered, and nothing warns you.

Reading query plans

Look at	It tells you
`rows` estimated vs actual	Whether statistics are trustworthy

Look at	It tells you
`Seq Scan` on a large table	Index missing or not usable
`Buffers: shared read`	Real I/O, not a cache hit
`loops=N`	Per-loop figures — multiply them

`EXPLAIN` prints the plan with estimates; `EXPLAIN ANALYZE` runs the statement and prints actuals beside them, with `BUFFERS` included by default from 18. Costs are "arbitrary units" anchored to `seq_page_cost = 1.0`, so compare them to each other and never read them as milliseconds. A wide gap between estimated and actual rows is the signal to run `ANALYZE`, since a bad row estimate is what makes the planner choose a bad join. A sequential scan is not automatically a fault — reading most of a small table that way is the correct plan.

```
EXPLAIN (ANALYZE, BUFFERS)
SELECT * FROM orders
 WHERE customer_id = 42;
```

> **Warning:** `EXPLAIN ANALYZE` executes the statement, including `INSERT`, `UPDATE`, `DELETE` and `MERGE` — the rows really change. Wrap it in `BEGIN` and `ROLLBACK` when the statement writes.

MVCC and vacuum

Symptom	Cause
Table grows while you delete	Dead row versions not reclaimed
Index-only scan not chosen	Visibility map behind
`VACUUM FULL` froze the app	It holds `ACCESS EXCLUSIVE`
“must be vacuumed within…”	Wraparound approaching

An `UPDATE` does not overwrite: it writes a new version and leaves the old one until no snapshot can still see it. Vacuum then does four jobs — reclaim that space, refresh planner statistics, update the visibility map that makes index-only scans possible, and freeze rows so the 32-bit transaction counter cannot wrap, which requires vacuuming every table “at least once every two billion transactions”. Plain `VACUUM` runs beside normal traffic and marks space reusable; `VACUUM FULL` rewrites the table under a lock that blocks everything, which is why “administrators should strive to use standard `VACUUM` and avoid `VACUUM FULL`”.

```
SELECT relname, n_dead_tup, last_autovacuum
  FROM pg_stat_user_tables
 ORDER BY n_dead_tup DESC
 LIMIT 10;
```

> **Gotcha:** One long-lived transaction holds back vacuum everywhere, because rows it might still see cannot be removed from any table. An idle-in-transaction session left open by a nightly job bloats tables it never touched.

Transactions and isolation

Level	Still allows
Read Committed (default)	Nonrepeatable reads, phantoms
Repeatable Read	Serialization anomalies
Serializable	Nothing — it aborts instead

Read Committed takes a fresh snapshot for every statement, and on a concurrent update it re-evaluates the `WHERE` clause against the new version — so an `UPDATE ... WHERE hits = 10` can match nothing after a neighbour incremented the row. Repeatable Read holds

one snapshot for the whole transaction. Serializable adds predicate locking (SSI, visible as `SIReadLock` in `pg_locks`) so the outcome always matches some serial order; predicate locks never block, they only cause aborts. Read Uncommitted is accepted and behaves as Read Committed.

```
BEGIN ISOLATION LEVEL SERIALIZABLE;
SELECT sum(amount) FROM entries
 WHERE acct = 1;
INSERT INTO entries (acct, amount)
VALUES (1, -50);
COMMIT;  -- may abort with SQLSTATE 40001
```

> **Warning:** Above Read Committed, correctness depends on the application retrying the **whole** transaction on `40001`. Retrying only the failed statement cannot work — the transaction is already aborted, and its snapshot is the thing that was doomed.

Locks and safe migrations

Operation	Lock
`SELECT`	`ACCESS SHARE`
`INSERT`, `UPDATE`, `DELETE`	`ROW EXCLUSIVE`
`ALTER TABLE`, `TRUNCATE`	`ACCESS EXCLUSIVE`
`CREATE INDEX`	Blocks writes, not reads
`CREATE INDEX CONCURRENTLY`	Blocks neither

`ACCESS EXCLUSIVE` "conflicts with locks of all modes", and lock requests are granted in order — which is what makes migrations dangerous. Set `lock_timeout` before DDL so a blocked statement gives up rather than

dragging the queue behind it.

`CREATE INDEX CONCURRENTLY` avoids blocking writes but "cannot" run inside a transaction block, needs two table scans, waits for open transactions, and on failure leaves "an 'invalid' index" to drop or rebuild. Deadlocks are detected automatically and resolved by aborting one transaction, so lock objects in a consistent order and retry.

```
SELECT id FROM jobs
 WHERE state = 'ready'
 ORDER BY id
 LIMIT 10
   FOR UPDATE SKIP LOCKED;
```

> **Gotcha:** A migration waiting for a lock blocks every statement that arrives after it, including plain `SELECT`s. `ALTER TABLE` stuck behind one slow query takes the table down for everyone — a one-second change becomes an outage.

JSONB

Operator	Yields
`->`	Field or element, as `jsonb`
`->>`	Field or element, as `text`
`@>`	Does the left contain the right?
`?`	Does this top-level key exist?

“Most applications should prefer to store JSON data as `jsonb`”: it is stored decomposed rather than as text, so it is slower to write, much faster to process, and — decisively — indexable. The trade is that it “does not preserve whitespace or key order” and keeps only the last of duplicate keys. The default GIN operator class

supports containment and the key-existence operators; `jsonb_path_ops` indexes only values, so it handles `@>` alone but produces indexes that are “usually smaller and faster”.

```
CREATE INDEX ON api
  USING gin (doc jsonb_path_ops);

SELECT doc ->> 'name' FROM api
 WHERE doc @> '{"tags": ["db"]}';
```

> **Gotcha:** Containment is not a search through nested levels — `'{"foo": {"bar": "baz"}}' @> '{"bar": "baz"}'` is false, because the key is not at the top level. Match the shape you stored, or index the sub-document with an expression index.

Operating it

Concern	First move
Hundreds of app connections	A pooler in front
Slow queries, unknown source	`pg_stat_statements`
Something is blocking	`pg_stat_activity`, `pg_locks`
Recovery to a point in time	WAL archiving, not dumps

Because every connection is a process with its own memory, connection count is a capacity decision, not a config number to raise — put a pooler in front before touching `max_connections`. Backups come in three shapes: SQL dump, file-system-level copy, and continuous WAL archiving; only the third can recover to a moment between backups. Minor upgrades are deliberately boring — stop, replace binaries, start, no

dump and reload — so run the current minor of your major, and plan majors around `pg_upgrade`.

```
SELECT pid, state, wait_event_type, query
  FROM pg_stat_activity
 WHERE state <> 'idle';
```

> **Note:** `pg_stat_statements` must be loaded through `shared_preload_libraries`, which needs a restart. Enable it while the system is healthy — you cannot add it in the middle of the incident it would have explained.

Further reading

- PostgreSQL 18 documentation
- Index types
- Using EXPLAIN
- Routine vacuuming
- Transaction isolation

- Explicit locking
- JSON types and operators
- PostgreSQL 18 release notes