PostgreSQL Indexing and EXPLAIN

Version PostgreSQL 18 · verified 2026-11-14

Which index to build and how to prove it works — B-tree rules, column order, covering, partial and GIN indexes, and reading EXPLAIN.

Contents

Mental model 2

B-tree basics 2

Composite indexes and column order 3

Covering, partial and expression indexes 4

GIN and other index types 5

Foreign keys and the cost of indexes 6

Building indexes in production 7

Running EXPLAIN 8

Scan nodes 9

Join and sort nodes 11

Further reading 12

Mental model

An index is a second, sorted copy of some columns that points back to table rows. It never changes what a query returns — only whether the planner chooses to use it, which it decides by estimating the cost of every plan from sampled statistics. So an index helps only when it matches the query's shape and the planner believes it is cheaper, and every index you add slows down every write. `EXPLAIN` is how you see the planner's decision instead of guessing it.

B-tree basics

Serves	Example
Equality and ranges	`=`, `<`, `>=`, `BETWEEN`, `IN`
Null tests	`IS NULL`, `IS NOT NULL`
Sorting	`ORDER BY created_at` with no `Sort` node
Prefix patterns	`LIKE 'abc%'`, `~ '^abc'`
Not served	`LIKE '%abc'`, `lower(col) = ...`

`CREATE INDEX` builds a B-tree unless you say otherwise, and it is the right choice for nearly every scalar column. Because entries are sorted, the same index answers range filters and returns rows already in order, so `ORDER BY ... LIMIT` reads only the first few entries.

```
CREATE INDEX ON orders (created_at);


SELECT id FROM orders
```

```
WHERE created_at >= now() - interval '1 day'
ORDER BY created_at DESC
LIMIT 50;
```

> **Gotcha:** Prefix `LIKE` uses a plain B-tree only in the `C` locale. Under a normal locale such as `en_US.UTF-8`, the index is silently ignored; build it with `text_pattern_ops` — `CREATE INDEX ON users (name text_pattern_ops)`.

Composite indexes and column order

Query	Index `(status, created_at)`
`WHERE status = 'open'`	Used
`... ORDER BY created_at`	Used, rows already sorted
`WHERE status = 'open' AND created_at > $1`	Used for both
`WHERE created_at > $1`	Poorly: leading column unconstrained

Order columns as **equality first, then the range or sort column**. The index narrows the scan with equality on leading columns plus one range on the first column that has no equality; conditions on columns after that are checked inside the index but do not shrink the part that is read. One `(a, b)` index also covers queries on `a` alone, so a separate index on `a` is usually redundant.

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


-- One index scan, no Sort node:
SELECT id, total FROM orders
WHERE status = 'open'
ORDER BY created_at
LIMIT 20;
```

> **Note:** From PostgreSQL 18, B-tree **skip scan** lets `(status, created_at)` serve `WHERE created_at > $1` by probing each distinct `status`. It pays off only when the leading column has few distinct values; do not design column order around it.

Covering, partial and expression indexes

Kind	Syntax	Use when
Covering	`INCLUDE (total)`	Avoid heap reads
Partial	`WHERE status = 'open'`	Queries hit a small subset
Expression	`(lower(email))`	Filter on a computed value

A covering index stores extra non-key columns so the planner can use an **Index Only Scan** and skip the table — but only for pages the visibility map marks all-visible, which `VACUUM` maintains. A partial index holds only rows matching its `WHERE`, so it stays small when you query a

minority of rows. An expression index stores a computed value and is used only when the query repeats the expression exactly.

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

CREATE INDEX ON orders (customer_id)
  WHERE status = 'open';

CREATE INDEX ON customers (lower(email));
SELECT id FROM customers
WHERE lower(email) = 'user7@ex.com';
```

> **Gotcha:** The planner matches a partial index's predicate when it plans, not when it runs. `WHERE status = $1` in a prepared statement with a generic plan cannot prove `$1` is `'open'`, so the partial index is skipped. Write the constant in the query text.

GIN and other index types

Type	Fits
GIN	`jsonb`, arrays, `tsvector` full text
GIN + `pg_trgm`	`LIKE '%abc%'`, fuzzy match
GiST	Ranges, geometry, nearest neighbour
BRIN	Huge append-only tables ordered on disk
HNSW (pgvector)	Vector similarity

GIN indexes many keys per row — every key of a document, every element of an array, every lexeme of a text — so it answers “contains” and “overlaps” questions a B-tree cannot. It is slower to update than a B-tree. BRIN stores a summary per block range and is tiny, but only works when the column correlates with physical order, such as an insert timestamp.

```
CREATE INDEX ON orders
  USING gin (meta jsonb_path_ops);

SELECT count(*) FROM orders
WHERE meta @> '{"tags": ["gift"]}';
```

> **Tip:** `jsonb_path_ops` supports only `@>` (and JSON path operators), but is smaller and faster than the default `jsonb_ops`. Use the default only if you need key-exists operators like `?`.

Foreign keys and the cost of indexes

Index	Cost
Every index	Extra write per `INSERT` and most `UPDATE`s
Unindexed FK column	Seq scan per parent `DELETE`
Unused index	Write cost, disk, no benefit

PostgreSQL indexes the referenced side of a foreign key — it must be a primary key or unique constraint — but **not** the referencing column.

Without that index, deleting a parent row or updating its key scans the whole child table, and joins from parent to children cannot use an index. Check `pg_stat_user_indexes` for indexes with zero scans before adding more.

```
CREATE INDEX ON orders (customer_id);


SELECT indexrelid::regclass AS idx,
  idx_scan
FROM pg_stat_user_indexes
WHERE idx_scan = 0;
```

> **Gotcha:** `idx_scan` counts since the statistics were last reset and is per server. An index that looks unused on the primary may be what a read replica's reports depend on — check every node first.

Building indexes in production

Statement	Blocks writes	In a transaction
`CREATE INDEX`	Yes, for the build	Allowed
`CREATE INDEX CONCURRENTLY`	No	Not allowed
`DROP INDEX CONCURRENTLY`	No	Not allowed
`REINDEX INDEX CONCURRENTLY`	No	Not allowed

`CONCURRENTLY` builds without locking out writes, at the cost of two table scans and waiting for every transaction that predates the build. Only

one concurrent build runs per table at a time. If it fails — a deadlock, or a duplicate for a unique index — it leaves an `INVALID` index that queries ignore but every write still maintains.

```
CREATE INDEX CONCURRENTLY orders_cust_idx
  ON orders (customer_id);

-- Find leftovers from failed builds:
SELECT indexrelid::regclass
FROM pg_index
WHERE NOT indisvalid;
```

> **Warning:** A failed concurrent build is not rolled back. Drop the invalid index (`DROP INDEX CONCURRENTLY`) and retry, or `REINDEX INDEX CONCURRENTLY` it. A retry with `IF NOT EXISTS` succeeds while doing nothing, leaving the broken index in place.

Running EXPLAIN

Option	Adds
`EXPLAIN`	Estimated plan; runs nothing
`ANALYZE`	Executes it; actual rows and time
`BUFFERS`	Cache hits vs reads (default with `ANALYZE` from 18)
`rows=` estimate vs actual	Whether statistics can be trusted
`loops=N`	Per-loop figures — multiply by N

Read a plan from the innermost, most indented node outward: children produce rows for their parent. `cost` is in arbitrary planner units — compare costs to each other, never to milliseconds — and with `ANALYZE` on, trust `actual time` instead. A large gap between estimated and actual `rows` usually means stale statistics: run `ANALYZE orders;` and look again. In `Buffers`, `shared hit` is a read from cache and `shared read` came from the OS or disk.

```
Limit (actual rows=20 loops=1)
  Buffers: shared hit=5 read=3
  -> Index Scan using
       orders_status_created_at_idx
     (rows=24533) (actual rows=20)
       Index Cond: (status = 'open')
Execution Time: 0.062 ms
```

> **Warning:** `ANALYZE` really executes the statement, so an `UPDATE` or `DELETE` changes rows. Run it between `BEGIN;` and `ROLLBACK;`.

Scan nodes

Node	Means	Usual action
Seq Scan	Reads the whole table	Fine if small or most rows match

Node	Means	Usual action
Index Scan	Index lookup, then table	Good for selective filters
Index Only Scan	Index alone	Watch `Heap Fetches`
Bitmap Heap Scan	Index collects pages, then reads them	Medium selectivity

A sequential scan is not a fault in itself: reading most of a small table front to back beats random index lookups. It is a problem when a selective `WHERE` on a large table still reports a `Seq Scan` with `Rows Removed by Filter` in the hundreds of thousands. For an Index Only Scan, `Heap Fetches` counts rows that had to be checked in the table anyway because their page was not all-visible.

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

> **Tip:** A high `Heap Fetches` on an Index Only Scan means the visibility map is behind. `VACUUM orders;` updates it, and autovacuum tuning keeps it current.

Join and sort nodes

Node	Means	Watch for
Nested Loop	Inner side runs per outer row	Large `loops` on an unindexed inner side
Hash Join	Hash table built on one side	`Batches` > 1 spills to disk
Merge Join	Joins two sorted inputs	Extra `Sort` nodes feeding it
Sort	Sorts rows	`Sort Method: external merge  Disk:`

A nested loop is ideal when the outer side is small and the inner side is an index lookup; it is disastrous when a bad estimate makes the planner expect 1 outer row and it gets 100,000. Sorts and hashes that do not fit in `work_mem` spill to temporary files. Either raise `work_mem` for that session or add an index that returns rows already in order.

```
SET work_mem = '64MB';  -- this session only
EXPLAIN (ANALYZE, BUFFERS)
SELECT customer_id, sum(total)
FROM orders
GROUP BY customer_id
ORDER BY 2 DESC;
```

> **Gotcha:** `work_mem` is a limit per sort or hash operation, not per query or per connection. One complex query can use several multiples of it at once; raising it server-wide on a busy system can exhaust memory.

Further reading

- PostgreSQL — Indexes
- PostgreSQL — Multicolumn indexes
- PostgreSQL — Index-only scans and covering indexes
- PostgreSQL — Partial indexes
- PostgreSQL — CREATE INDEX, building concurrently
- PostgreSQL — Using EXPLAIN
- PostgreSQL — Statistics used by the planner