Engineer

Database index

a structure that finds table rows without a full scan

What it is

A database index is a separate structure, stored alongside a table, that lets the database go straight to the rows matching a condition instead of reading the table from the first row to the last. In everyday talk, and in most of the PostgreSQL documentation, it is simply an index. The documentation compares it to the index at the back of a book: a reader who wants one topic looks it up there and turns to the pages listed, rather than leafing through the whole volume.

Take a table of orders and the condition `WHERE customer_id = 42`. With no index on `customer_id`, PostgreSQL has one way to answer: a sequential scan, which reads every row and keeps the ones that match. That costs the same whether the customer has two orders or two thousand, and it grows with the table. With an index on the column, PostgreSQL walks a few levels of a search tree to the entries for 42 and fetches only those rows. Creating an index is a single statement, `DROP INDEX` removes it again, and either can happen at any time.

Once an index exists, PostgreSQL keeps it in step with the table on every write, and the query planner decides, query by query, whether using it is cheaper than a scan. Nothing in the SQL names the index, so one index can speed up a `SELECT`, an `UPDATE` or `DELETE` with a search condition, and a join.

PostgreSQL ships several kinds, each built for a different shape of question:

- **B-tree** is the default, and what `CREATE INDEX` builds unless told otherwise. It answers equality and range comparisons (`<`, `<=`, `=`, `>=`, `>`), `BETWEEN`, `IN`, `IS NULL`, and a `LIKE` whose pattern is anchored at the start. It keeps its entries in order, so it can also return rows already sorted.
- **Hash** stores a 32-bit hash code of each value and answers equality only.

- **GiST** and **SP-GiST** are frameworks for data such as geometry, where the question is “overlaps”, “contains” or “nearest to” rather than “equals”.
- **GIN** is an inverted index for values made of many parts, such as arrays, and finds the rows whose value contains a given element.
- **BRIN** keeps only a summary for each range of table blocks. It suits a column whose values follow the physical order of the rows, as an insert timestamp often does.

On top of its type, an index can be shaped to the question. A unique index refuses duplicate values. A multicolumn index covers several columns, up to 32. A partial index has a `WHERE` clause and holds only the rows that satisfy it. An expression index stores a computed value such as `lower(email)`. And an `INCLUDE` list carries extra columns in the index without searching on them, so that a query can be answered from the index alone.

Use the concept in a decision

Put “What is a database index?” to work in a real decision. We can help relate the concept to the system you are building or buying. Finding the queries and the missing indexes behind a system that has got heavier every year, and fixing the ones that matter.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. 30 days of electricity data in 180 ms — 180 ms to aggregate a month of meter readings

30 days of electricity data in 180 ms Get in touch →

Why it matters

The difference an index makes grows with the table. On a small table a scan is cheap, and the planner will often, rightly, prefer one: reading a handful of pages in order costs less than walking a tree to reach them. On tens of millions of rows, the gap between a scan and a lookup is the gap between a page that loads and one that times out. A query that was quick at launch can turn slow a year later without a line of code changing, simply because the table it reads has grown.

Indexes are not free, and the bill arrives on writes. Every insert adds an entry to every index on the table. An update can avoid that only as a heap-only tuple (HOT) update, where the new row version stays on the same page and the index entries already there still lead to it. PostgreSQL allows that only when the update leaves alone every column the table’s indexes use, BRIN indexes aside, since they

hold only a summary. Each index therefore narrows the cheap path, which is why the documentation advises dropping indexes that are seldom or never used. A table with a dozen indexes pays a dozen times on every insert.

Building an index costs something too. A plain `CREATE INDEX` lets other sessions read the table but blocks its writes until the build finishes, and the documentation warns that on a very large table that can take many hours.

Finally, an index is only as good as the planner's decision to use it, and the planner works from statistics gathered by `ANALYZE`. The question to ask is not "is there an index?" but "do the queries that matter use it?", and `EXPLAIN` is how to find out.

What it looks like in practice

An online shop's account page lists a customer's twenty most recent orders:

```
SELECT id, created_at, total
FROM   orders
WHERE  customer_id = 42
ORDER  BY created_at DESC
LIMIT  20;
```

With only a primary key on `id`, PostgreSQL has to read the whole of `orders`, keep the customer's rows, sort them and throw away all but twenty. `EXPLAIN` shows it: a `Seq Scan` on `orders` with the condition as a `Filter`, and a sort above it. `EXPLAIN ANALYZE` also runs the query and prints the real row counts and timings beside the estimates.

The index that fits this question puts the equality column first and the sort column second:

```
CREATE INDEX CONCURRENTLY orders_customer_created_idx
ON orders (customer_id, created_at);
```

Run `EXPLAIN` again and the scan becomes an index scan using `orders_customer_created_idx`, with `customer_id = 42` now shown as an `Index Cond` rather than a filter. The sort goes too. A B-tree keeps the entries for customer 42 ordered by `created_at`, PostgreSQL can read them backwards to get the newest first, and with a `LIMIT` it stops after twenty entries instead of touching the rest.

The order of the columns matters. The same index serves a query on `customer_id` alone, its leading column, but a report on every order placed yesterday, filtering on `created_at` alone, gets far less from it: in a multicolumn B-tree, equality on the leading columns plus a range on the next one are what narrow the part of the index that is read. PostgreSQL 18 can skip-scan, repeating the search once per distinct leading value, but the planner usually picks that only when the leading column has few distinct values, and customer IDs have many. If the report matters, it gets its own index on `created_at`. Turned round, as `(created_at, customer_id)`, the index would scatter customer 42's orders across the whole of it.

The note on optimising the database before you shard it puts indexes first among the fixes it ranks by payoff, once `EXPLAIN (ANALYZE, BUFFERS)` has shown which scans a hot query runs. Its own example is a partial index, `units_unsold`, holding only the units still for sale, so it shrinks as a sale goes on instead of growing. The note also counts the price. On its table, 2,000 claims that set `reserved`, `reserved_at` and `order_id` were all HOT updates, while of 2,000 updates that set `sold`, a column the partial index reads, none were. Its advice is to drop the indexes whose `idx_scan` in `pg_stat_user_indexes` stays at zero.

The case study on a better choice of key shows why the values going into a B-tree matter as much as the columns. Random UUIDs scatter inserts across the index, which can cause fragmentation. The platform made time-ordered UUID v7 the default for entity primary keys, with a timestamp in the leading bits so new identifiers carry time ordering. Reduced fragmentation and faster queries were design aims; the recorded result is the configured key convention, with no before-and-after workload benchmark established by the case.

Where it goes wrong

Adding an index stops the table's writes. A plain `CREATE INDEX` on a busy table blocks inserts, updates and deletes for the whole build, and a routine deployment turns into an outage. Use `CREATE INDEX CONCURRENTLY`: it scans the table twice and waits for transactions that could use the index, so it takes

significantly longer, but writes carry on. It cannot run inside a transaction block, so a migration tool that wraps every step in one has to be told to run this statement on its own. Only one concurrent build can run on a table at a time.

A concurrent build fails and leaves an invalid index behind. A deadlock, or a duplicate value while building a unique index, makes `CREATE INDEX CONCURRENTLY` fail, but the index it started stays in the catalogue, marked invalid. Queries ignore it, yet every write still pays to maintain it, and an invalid unique index can go on enforcing uniqueness. psql's `\d` lists it with `INVALID` after its definition. Drop it and run the concurrent build again, or rebuild it with `REINDEX INDEX CONCURRENTLY` .

The query hides the column inside a function. An index on `email` does nothing for `WHERE lower(email) = 'ana@example.com'` , because the index holds `email` , not `lower(email)` . Index the expression the query uses:

```
CREATE INDEX users_lower_email_idx ON users (lower(email));
```

The expression is computed on every insert and every update that is not HOT, which is why the documentation recommends expression indexes where retrieval speed matters more than write speed.

Indexes pile up that nobody uses. Each one added "just in case" costs every write and can rule out HOT updates. `pg_stat_user_indexes` counts the scans started on each index in `idx_scan` and records the time of the last one in `last_idx_scan` :

```
SELECT relname, indexrelname, idx_scan, last_idx_scan
FROM   pg_stat_user_indexes
WHERE  idx_scan = 0
ORDER  BY relname;
```

Look before dropping. An index behind a primary key or unique constraint does its job whether or not a query ever reads it, and a count of zero means little on a server that has only just started taking traffic.

The planner ignores a good index. After a bulk load the statistics lag behind the data, the estimates are wrong, and a scan wins on paper. Run `ANALYZE`; the documentation calls judging index use without it “a lost cause”. Test on realistic volumes too: on a hundred rows that fit in one disk page, no index beats reading that page. To learn whether an index can be used at all, set `enable_seqscan = off` in a test session and run `EXPLAIN` again. If the plan still scans, the condition does not match the index.

Further reading

- Indexes: Introduction — what an index is for, how it is created and dropped, and the overhead it adds to every write.
- Index Types — B-tree, Hash, GiST, SP-GiST, GIN and BRIN, and the operators each one can answer.
- Multicolumn Indexes — how column order decides what a B-tree can narrow, and when skip scan applies.
- CREATE INDEX — the full syntax, including `CONCURRENTLY`, `INCLUDE`, partial indexes and recovering from an invalid build.
- Using EXPLAIN — reading a plan: scan types, estimated costs and the real figures from `EXPLAIN ANALYZE`.
- Examining Index Usage — why `ANALYZE` and real data come first, and how to test whether an index can be used.



Glossary

Choose for your actual constraints

Bring a representative dataset, the questions it must answer and the current bottleneck. We can agree how to measure quality, throughput and recovery.

See what it costs Get in touch →

Make the terminology practical

Ask how “What is a database index?” applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Carefully Crafted With Love by D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/database-index/