Engineer

Partitioning

splitting one table into physical pieces

What it is

Partitioning, also called table partitioning, stores the rows of one logical table in several ordinary tables, each holding the rows whose key falls inside its own bounds, while queries and writes go on using the single name they always used. PostgreSQL, which this page follows, calls its built-in form declarative partitioning: the table is declared as partitioned, with a method and a partition key, and every partition is created as a table of its own.

The partitioned table holds no rows itself. An insert is routed to the partition whose bounds match the key, and an update that changes the key moves the row to the partition where it now belongs. The partitions share the parent's columns but can have their own indexes, storage settings and tablespace, and a partition can itself be partitioned again.

There are three methods. Range partitioning gives each partition a span of values, usually of dates; the lower bound is included and the upper one is not, so with spans 1–10 and 10–20 the value 10 belongs to the second. List partitioning names the values each partition takes, such as a set of country codes. Hash partitioning gives each partition a modulus and a remainder, and a row goes wherever the hash of its key leaves that remainder, which spreads rows evenly when there is no natural range to cut along. A default partition can catch the values that fit no other, for range and list partitioning only.

Partitioning happens inside one database server. Splitting data across separate servers, each with its own copy of the schema, is sharding, which is a much larger step. SQL Server and BigQuery have their own forms of partitioning, with their own syntax, and the ideas below carry over even where the commands differ.

Use the concept in a decision

Put “What is partitioning?” 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

Most large tables grow at one end. Orders, readings, logs and events arrive every day, stay for months or years, and most queries ask about the recent part. Without partitioning, every query works against one table and one set of indexes that keep getting larger.

With a partition key that matches the queries, the planner can prove that most partitions cannot hold a matching row and leave them out of the plan entirely. PostgreSQL calls this partition pruning. A report on last week reads last week’s partition and nothing else. The documentation adds a less obvious gain: because the partitions take over the job of the upper levels of an index tree, the parts of the indexes that are used most are more likely to fit in memory.

Removing old data changes most of all. Deleting a month of rows from a single table is a long `DELETE` that leaves dead rows behind for `VACUUM` to clean up. With a partition per month, the same retention is one `DROP TABLE` or one `DETACH PARTITION`, which the documentation calls far faster than a bulk operation and which avoids the `VACUUM` overhead entirely. A detached partition is still an ordinary table, so it can be backed up or summarised before it is dropped, and partitions that are rarely read can sit on cheaper, slower storage.

The gain has a price, and the documentation is direct about when it pays. Each partition adds work to planning, and the benefits are normally worthwhile only when the table would otherwise be very large, with a rule of thumb that the table should be larger than the database server’s physical memory. Partitioning also leaves one thing untouched: a single row that many transactions fight over is still one row in one partition, behind one lock.

What it looks like in practice

An `orders` table split by month on `created_at` is the plainest case, in a note on scaling a database before sharding it that spends a section on partitioning the tables around a hot row. The primary key is `(id, created_at)` because the key has to include the partition column. A query that names a date range reads only the partitions inside it, this month's indexes stay small enough to live in memory, and retention becomes a single `DETACH PARTITION` in place of a `DELETE` of millions of rows. The limit is just as plain: partitioning does not make the contended row faster, and a query that does not name the key reads every partition.

A PostgreSQL performance project for an application holding user accounts, tenants, geographical data and daily electricity consumption split the electricity table into time ranges by day and month, and used hash partitioning for the geographical data. That came together with GiST and composite indexes and rewritten queries, and the combined result was query response times improved by 10x and resource utilization down by 40%.

At a far larger scale, partitioning was one technique among several for at least 100 databases across PostgreSQL, SQL Server and BigQuery, much of it geographic and time-series data. Time-series partitioning was part of the schema design in PostgreSQL and SQL Server, and BigQuery used time-partitioned and clustered tables. Alongside spatial indexes, materialized views and caching, query response times for that data fell by 40–60%.

Where it goes wrong

Partitioning moves some of the work from query time to design time, and most problems come from a design decision made without the queries in view.

Inserts start failing on the first day of a new month. No partition covers the new range, and PostgreSQL refuses an insert into the parent that maps to no existing partition. Create partitions ahead of time, from a scheduled job that generates the DDL, as the documentation suggests, and alert when the newest partition is close to its end:

```
CREATE TABLE orders_2026_11
  PARTITION OF orders
  FOR VALUES FROM ('2026-11-01')
              TO ('2026-12-01');
```

A default partition stops the errors, but rows for every uncovered month then land together in that one table, and attaching a new partition later scans the default under an `ACCESS EXCLUSIVE` lock unless a `CHECK` constraint on the default already excludes the new range. Treat a non-empty default partition as an alarm.

Queries are slower after partitioning than before. The common queries do not filter on the partition key, so each one reads every partition and its indexes. A second cause is the number of partitions: the documentation says the planner copes well with up to a few thousand, provided typical queries prune all but a few, and that planning time and memory grow beyond that. Choose the key from the `WHERE` clauses the application already writes, check with `EXPLAIN` that only the expected partitions appear, and keep the count in proportion. One partition per customer is a typical trap, since the count grows with the business; the documentation suggests hash partitioning with a reasonable number of partitions instead.

A primary key or unique constraint will not create. On a partitioned table, a unique or primary key must include every partition key column, and the partition key must not contain expressions or function calls. Each partition's index can only enforce uniqueness inside that partition. Add the key column to the constraint, as the note's `orders` table does with `(id, created_at)`.

Building an index on the parent blocks writes for a long time. A plain `CREATE INDEX` on the parent locks out writes until every partition is indexed, and the `CONCURRENTLY` option works per partition rather than on the parent. Create the index with `ON ONLY` on the parent, which leaves it marked invalid, build the index on each partition with `CONCURRENTLY`, and attach each one with `ALTER INDEX ... ATTACH PARTITION`. Once every partition's index is attached, the parent index becomes valid by itself.

An existing large table cannot be switched over in place. PostgreSQL does not turn a regular table into a partitioned one. Create the new partitioned table and attach the old table as one of its partitions. Before attaching, add a `CHECK` constraint that matches the partition's bounds, so the attach does not scan the whole table while holding an `ACCESS EXCLUSIVE` lock on it.

Further reading

- Table Partitioning — the PostgreSQL chapter: the three methods, pruning, maintenance, limitations and best practices.

- CREATE TABLE — `PARTITION BY` , `PARTITION OF` , partition bounds and the default partition.
- ALTER TABLE — `ATTACH PARTITION` and `DETACH PARTITION` , with what each one locks.
- CREATE INDEX — indexes on partitioned tables, `ON ONLY` and building `CONCURRENTLY` .

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 partitioning?” 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/partitioning/