Engineer

Scalability

whether more load needs more than more machines

What it is

Scalability is how a system's capacity grows as resources are added to it. A system scales well when twice the machines carry roughly twice the load at the same response times, and badly when the extra hardware mostly waits on something it cannot help with. The word describes a relationship between load and cost, not a feature a system either has or lacks.

There are two directions to grow in. Scaling up, or vertical scaling, means a bigger machine: more cores, more memory, faster disks. It needs no change to the software and stops at the largest machine worth buying. Scaling out, or horizontal scaling, means more machines sharing the work. It has no such ceiling, but only work that can be divided gains from it.

HTTP was shaped with scaling out in mind. RFC 9110 notes that many implementations rely on HTTP being stateless to reuse proxied connections or to load balance requests dynamically across multiple servers. A web tier that keeps nothing between requests can grow by adding copies of itself behind a load balancer, and any copy can answer any request.

Load has more than one dimension, and a system can scale along one and stall along another: requests per second, the volume of stored data, the number of tenants, and the number of writers contending for the same record. The last is the hardest, because a single row can be changed by one transaction at a time, however many servers stand around it.

The arithmetic behind that is Amdahl's law. When part of the work has to run in sequence, parallel hardware speeds up only the rest, so the sequential part sets the limit: with 5% of the work serial, the whole job can never run more than 20

times faster, even on a thousand machines. In a database the serial part is often a lock. In an application it is often one shared counter, queue or session store.

Use the concept in a decision

Put “What is scalability?” to work in a real decision. We can help relate the concept to the system you are building or buying. Monitoring and alerts that speak up while a fault is still small, and the figures that show how often it stays that way.

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. Built the company's first observability dashboard — the company's first view of its own systems

Built the company's first observability dashboard Get in touch →

Why it matters

Scalability decides what growth costs. When a system scales well, a doubling of traffic is a line on an invoice. When it scales badly, the bill grows faster than the load, or more hardware stops helping altogether, and the fix becomes a redesign carried out while the traffic is arriving.

Hardware can relieve bottlenecks in processors, memory or disks, but inefficient work and serialization can remain. A query that needlessly scans a whole table can still waste work on a larger server. A bigger machine does not remove the serialization of updates to one shared row, and more application servers can add writers to the queue. Most systems that look as if they need more machines need reading first.

In PostgreSQL the place to start reading is `pg_stat_statements`, an extension that records each normalised statement with its call count and its execution time. Sorted by total time rather than by mean, it shows where the server actually spends itself: a query that takes 2 ms and runs a million times an hour costs 2,000 seconds of that hour, while a two-second report run twice costs four.

```
SELECT calls, round(total_exec_time) AS total_ms,
       left(query, 60) AS query
  FROM pg_stat_statements
 ORDER BY total_exec_time DESC
 LIMIT 10;
```

Each step towards more machines brings running costs of its own, which is why the order matters. A tighter statement or a better index can improve performance without adding another server. A read replica is configuration. Partitioning changes one table. Sharding, splitting the data across separate database servers, changes the surrounding operations: migrations, backups, restores and reports have to account for separate shards, joins across shards may move into application code, and a query that does not know its shard key may have to ask all of them.

What it looks like in practice

A PostgreSQL database behind a mid-sized application held user accounts, tenants, geographical data and daily electricity consumption. It ran under low traffic, yet slowed as the data volumes grew. The brief was to make it ten times faster without overhauling the architecture, with room for future data growth as part of the goal. The work was GiST indexes for the spatial data, composite indexes on tenant and date range, the electricity table partitioned by time, hash partitioning for the geographical data, rewritten queries checked with `EXPLAIN ANALYZE`, and connection pooling. Response times improved tenfold, the database's resource use fell by 40%, and the system took growing data volumes without slowing down, with less need for hardware upgrades.

At a mid-sized company running ten PostgreSQL and SQL Server database servers, the infrastructure was scaled to carry a 40% rise in user traffic without service degradation. Alongside it, query response times improved by 30% through configuration tuning and index optimisation.

The order the work follows is set out in a note on optimising before sharding: look, tighten, take the reads off the primary, partition the tables around the hot spot, and shard only when one machine is full of good work. It names three cases where sharding is reasonable: the primary is saturated by statements that are already lean, the data has outgrown its maintenance, or isolation is itself the requirement.

Where it goes wrong

More web servers in front of one database. Each application server opens its own connections, and when many times more sessions are active than the database has cores, the server spends its time switching between them. A

connection pool fixes that before any hardware does, and another web server makes it worse.

One hot row. When every request writes to the same record, such as a stock counter or a balance, throughput is capped at roughly the writes one lock can carry divided by how long the lock is held, on any number of machines. Anything slow inside that lock lowers the ceiling. Keep the transaction short, keep external calls outside it, and when one item outgrows its machine, split its stock into smaller pieces that each have exactly one owner.

State that lives on one server. Sessions held in one process's memory, uploads written to its local disk, or a scheduled job that assumes it is the only copy all stop a second instance from being an equal one. Keep that state in a shared store, so that any instance can answer any request.

Replicas treated as copies of the primary. A streaming replica runs a little behind, so a user who has just written something and then reads from the replica may not see it; read recent writes from the primary. A long report on a standby can also be cancelled when it holds up the changes being replayed. PostgreSQL documents these as query conflicts, together with the settings that trade replica lag against them.

Partitions the queries never name. Partitioning splits one table by a key, such as a month or a tenant, so a query that names the key reads only the partitions it needs. A query that leaves the key out reads every partition and gains nothing from the split. Partition by what the queries already filter on, and expect it to pay off only for large tables: the PostgreSQL manual's rule of thumb is a table bigger than the database server's physical memory.

Further reading

- RFC 9110: HTTP Semantics, section 3.3 — connections, clients and servers, and the stateless design that lets implementations spread requests across multiple servers.
- PostgreSQL: Table Partitioning — what partitioning is, when it pays, and how partition pruning skips the partitions a query cannot need.
- PostgreSQL: pg_stat_statements — the extension that keeps call counts and timings for every normalised statement.
- PostgreSQL: Hot Standby — read-only queries on a standby and the conflicts that can cancel them.

- RFC 9111: HTTP Caching — how HTTP caches store and reuse responses, so repeat requests need not reach the origin server.

Glossary

Choose for your actual constraints

Bring the deployment path, the operational risks and the people on call. We can scope a change with checks, a recovery route and clear ownership.

See what it costs Get in touch →

Make the terminology practical

Ask how “What is scalability?” 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/scalability/