Engineer

Database normalization

arranging tables so each fact is stored once, under its key

What it is

Database normalization is the practice of laying out a relational database so that each fact is stored once, in the table it describes, and everything else refers to it by its key. In British spelling it is database normalisation. A customer's address sits in one row of a customers table; an order carries only the customer's key, and a join brings the two together when a query needs both.

The idea and the word both come from Edgar F. Codd, who proposed the relational model in Communications of the ACM in June 1970. In that paper normalization meant one particular step: getting rid of relations that hold other relations inside them, such as an employee record that carries its own list of past jobs and its own list of children. Codd's procedure copies the parent's primary key into each nested group and lifts the group out into a relation of its own, keyed by the parent's key plus whatever identifies a row within the group. What is left is in normal form, now known as first normal form: every value in every column is a single, simple value.

Codd went on to define stricter normal forms, among them the one that Ronald Fagin's 1977 paper calls Codd's improved third normal form, also known as Boyce–Codd normal form. Fagin's paper adds a fourth. It rests on a new kind of dependency between columns, the multivalued dependency, which includes the familiar functional dependency as a special case, and Fagin showed the fourth form to be strictly stronger than Boyce–Codd normal form.

The stricter forms are reached by splitting. A table is replaced by two or more of its projections, tables made of some of its columns, chosen so that joining them gives back exactly the original. Fagin's paper shows that guarantee holds for its own form: any table can be split into tables in fourth normal form without losing

information, because joining them recovers the original. Denormalization is the reverse: keeping a copy of a fact in a second place on purpose, usually to save a join.

Use the concept in a decision

Put “What is database normalization?” to work in a real decision. We can help relate the concept to the system you are building or buying. A structure that matches how the business actually works, so the questions you need answered later are still possible to ask.

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. Designed and ran 6 ETL/ELT pipelines — 6 ETL pipelines across three database engines

Designed and ran 6 ETL/ELT pipelines Get in touch →

Why it matters

A table that stores the same fact many times invites three kinds of trouble. Picture one row per employee and project, with the employee’s office repeated on every row. When someone changes office, several rows need the new value, and missing one leaves the database contradicting itself. When an employee’s last project ends and its row is deleted, the record of where they sit goes with it. And a new hire who is not on any project has nowhere to record an office. These are the update, deletion and insertion anomalies, and splitting the table into employee–project and employee–office removes all three.

In a team’s daily work these anomalies arrive as bugs. An invoice shows the old address because one copy was missed. A report counts a customer twice because the name was typed two ways. A normalized schema makes such bugs hard to write in the first place, and with primary keys, foreign keys and check constraints declared, the database refuses bad data itself instead of trusting every piece of application code to remember the rules.

Change gets cheaper too. A schema whose tables each describe one thing tends to absorb new kinds of data with less restructuring: adding a column to the one table that owns a fact is a smaller migration than finding every table that copied it.

The price is joins. A normalized database answers most questions by joining tables back together, and each join is work for the server. With indexes on the joined columns that work is usually small, which is why a measured slow query, and not a hunch, should decide when a fact gets copied back in.

What it looks like in practice

A small shop starts with one flat table, loaded from a spreadsheet export:

```
CREATE TABLE order_flat (
    order_no       integer,
    ordered_on     date,
    customer_email text,
    customer_name  text,
    customer_city  text,
    sku            text,
    product_name   text,
    unit_price     numeric,
    quantity       integer,
    status         text
);
```

Each row is one line of one order, so a customer's name and city repeat on every line of every order, and a product's name repeats on every line that sells it. A customer who moves has dozens of rows to update. Deleting the last line that sold a product forgets that the product exists. The status is free text, so `shipped`, `Shipped` and `shipd` can all turn up.

Normalized, each fact gets one home:

```
CREATE TABLE customers (
    customer_id integer PRIMARY KEY,
    email       text NOT NULL UNIQUE,
    name        text NOT NULL,
    city        text NOT NULL
);

CREATE TABLE products (
    sku        text PRIMARY KEY,
    name       text NOT NULL,
    list_price numeric NOT NULL CHECK (list_price >= 0)
);

CREATE TABLE order_status (
    status_id smallint PRIMARY KEY,
    label     text NOT NULL UNIQUE
);

CREATE TABLE orders (
    order_id    integer PRIMARY KEY,
```

```
    customer_id integer NOT NULL REFERENCES customers ON DELETE RESTRICT,
    status_id   smallint NOT NULL REFERENCES order_status,
    ordered_on  date NOT NULL
);

CREATE TABLE order_lines (
    order_id   integer REFERENCES orders ON DELETE CASCADE,
    sku        text NOT NULL REFERENCES products,
    quantity   integer NOT NULL CHECK (quantity > 0),
    unit_price numeric NOT NULL CHECK (unit_price >= 0),
    PRIMARY KEY (order_id, sku)
);
```

A customer's city is now stored once, and a move is a one-row update. The status labels sit in a small lookup table keyed by a `smallint`, so each order carries a short number instead of a repeated string, and a misspelt status cannot get in: the foreign key refuses any `status_id` that `order_status` does not hold.

`unit_price` stays on `order_lines`, although `products` already has a price. That is no lapse. The price a customer paid is a fact about the sale, not about the product, and it must not change when the catalogue does. Normalization asks where each fact belongs, and this one belongs to the line.

The two `ON DELETE` clauses say what a deletion means. Deleting an order takes its lines with it, since a line means nothing without its order. Deleting a customer who has orders fails, because those orders are records the shop has to keep. PostgreSQL's default, `NO ACTION`, refuses that deletion too; `RESTRICT` states the intent, and unlike `NO ACTION` it cannot be deferred to the end of the transaction.

PostgreSQL builds an index for every primary key, but not for the referencing side of a foreign key. Its documentation suggests adding one, because deleting or updating a referenced row has to find every row that points at it:

```
CREATE INDEX ON orders (customer_id);
CREATE INDEX ON order_lines (sku);
```

`order_lines.order_id` needs nothing extra: it is the leading column of that table's primary key.

Finally, a view puts the flat shape back for whoever wants it:

```
CREATE VIEW order_report AS
SELECT o.order_id, o.ordered_on, c.email, c.name AS customer_name, c.city,
       l.sku, p.name AS product_name, l.unit_price, l.quantity, s.label
FROM   orders o
JOIN   customers c    USING (customer_id)
JOIN   order_status s USING (status_id)
JOIN   order_lines l  USING (order_id)
JOIN   products p     USING (sku);
```

Reports can read `order_report` much as they read the old table, while every fact behind it is stored once. The PostgreSQL tutorial makes the same case for views in general: a view gives a query a name that can be used like a table, and lets the tables underneath change behind an interface that stays put.

The same rules hold at a far larger scale. In the founder's work under NextMariner, the maritime data model was laid out as normalized PostgreSQL schemas organised by domain, which its page put at 348 tables across 34 schemas in October 2026. Small, stable sets such as statuses, types and categories became `SMALLINT` lookup tables, which keeps rows compact and joins cheap, and real foreign keys and constraints hold the relationships together, so that integrity is the database's job rather than something the application has to remember.

Where it goes wrong

Lists packed into one column. A `tags` column holding `red,large,sale`, or a phone column holding two numbers, breaks first normal form. The database cannot check the separate values, point a foreign key at them or join on them, and every query has to split the string first. Give the list a table of its own, one row per value, keyed by the parent's key plus the value: Codd's procedure from 1970.

Normalizing away history. The price on an order line, or the delivery address on an order, looks like a duplicate of what the products and customers tables hold. It is not one: it records what was true when the order was placed. Swap it for a key into the current catalogue, and every old invoice changes the next time a price does. Keep facts about an event on the event.

Foreign keys without indexes. The referenced side of a foreign key is always indexed, since it has to be a primary key or a unique constraint or index. The referencing side is not, so deleting a customer, or changing a product's SKU, means scanning the whole referencing table for rows that point at it. Index the referencing columns, as above. PostgreSQL leaves that to you because it is not always needed.

Cascades that delete records. `ON DELETE CASCADE` from customers to invoices means one careless delete of a customer quietly removes records the business needs to keep. Cascade only to rows that mean nothing without their parent, such as order lines, and use `RESTRICT`, or the default, everywhere else.

A normalized design that nothing enforces. Tables created by hand or loaded from spreadsheets often arrive with no primary key and no foreign keys, so nothing stops a duplicate row or an order for a customer who does not exist. PostgreSQL does not insist on a primary key, though its documentation notes that relational theory requires one in every table. Declare the keys, and the database rejects the bad row with an error that names the constraint.

Cached totals that drift. An `order_total` column on `orders`, kept beside the lines it sums, is a denormalization, and every code path that changes a line has to update it as well. One that forgets leaves the total wrong without raising any error. Compute the total in a query or a view. If that proves too slow, keep the copy, but maintain it in one place, inside the same transaction that changes the lines.

Denormalizing before measuring. Joins on indexed keys are cheap, and copying columns to avoid them trades a small, known cost for the anomalies described above. Read the query plan first. When reads really are too slow, a materialized view or a separate reporting database can hold the flattened shape while the source tables keep a single copy of each fact. Reporting is where denormalization is at home on purpose, as in the star schema of a data warehouse.

Splitting too far. Normalization asks for one home per fact, not for as many tables as possible. Splitting a table whose columns already depend on its key and on nothing else only adds joins, and table names for everyone to learn. A schema that needs a dozen joins to show one order is no easier to work with than a flat one. Stop once each fact has a single home.

Further reading

- A Relational Model of Data for Large Shared Data Banks — Codd's 1970 paper in Communications of the ACM, which introduced the relational model, normal form and normalization.
- Multivalued Dependencies and a New Normal Form — Fagin's 1977 paper defining the multivalued dependency and fourth normal form, and showing that any table can be split into fourth normal form without losing information.
- Constraints — PostgreSQL on primary keys, foreign keys, check constraints and the `ON DELETE` actions, and why referencing columns are worth indexing.
- Views — the PostgreSQL tutorial's case for views as a stable interface over tables whose structure changes.

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 database normalization?" 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/normalization/