Engineer

Databasenormalisering

at ordne tabeller, så hver oplysning kun gemmes ét sted

Hvad det er

Databasenormalisering er praksissen med at indrette en relationel database, så hver oplysning gemmes ét sted, i den tabel den beskriver, og alt andet henviser til den med dens nøgle. Man taler også om normalisering af databaser, eller blot normalisering, når sammenhængen gør det klart. En kundes adresse står i én række i en kundetabel; en ordre har kun kundens nøgle, og et join samler de to, når en forespørgsel har brug for begge.

Både idéen og ordet kommer fra Edgar F. Codd, der foreslog relationsmodellen i Communications of the ACM i juni 1970. I den artikel betød normalisering ét bestemt skridt: at komme af med relationer, der rummer andre relationer, for eksempel en medarbejderpost med sin egen liste over tidligere stillinger og sin egen liste over børn. Codds fremgangsmåde kopierer forælderens primærnøgle ind i hver indlejret gruppe og løfter gruppen ud i en relation for sig, med forælderens nøgle plus det, der identificerer en række inden for gruppen, som nøgle. Tilbage står normalform, i dag kendt som første normalform: hver værdi i hver kolonne er én enkel værdi.

Codd definerede siden strengere normalformer, blandt dem den, som Ronald Fagins artikel fra 1977 kalder Codds forbedrede tredje normalform eller Boyce–Codd-normalform. Fagins artikel tilføjer en fjerde. Den bygger på en ny slags afhængighed mellem kolonner, den flerværdige afhængighed, der omfatter den velkendte funktionelle afhængighed som et specialtilfælde, og Fagin viste, at den fjerde form er strengt stærkere end Boyce–Codd-normalform.

De strengere former nås ved at dele op. En tabel erstattes af to eller flere af dens projektioner, tabeller med nogle af dens kolonner, valgt sådan, at et join af dem giver præcis originalen tilbage. Fagins artikel viser, at den garanti holder for hans egen form: enhver tabel kan deles op i tabeller på fjerde normalform, uden

at der går information tabt, fordi et join af dem genskaber originalen. Denormalisering er det omvendte: med vilje at gemme en kopi af en oplysning et andet sted, som regel for at spare et join.

Brug begrebet i en beslutning

Brug »Hvad er databasenormalisering?« i en konkret beslutning. Vi kan knytte begrebet til det system, I bygger eller køber. En struktur, der passer til, hvordan forretningen faktisk fungerer, så de svar, I får brug for senere, stadig kan hentes.

Se vores ydelser Kontakt os →

Undersøg antagelserne bag ordet

Brug forklaringen og det tilknyttede arbejde til at formulere spørgsmål til jeres projekt. Vi kan undersøge antagelserne sammen, før et værktøj vælges. Har designet og drevet 6 ETL/ELT-pipelines — 6 ETL-pipelines på tre databasemotorer

Har designet og drevet 6 ETL/ELT-pipelines Kontakt os →

Hvorfor det betyder noget

En tabel, der gemmer den samme oplysning mange gange, lægger op til tre slags problemer. Forestil dig én række pr. medarbejder og projekt, hvor medarbejderens kontor står på hver række. Når nogen skifter kontor, skal flere rækker have den nye værdi, og glemmer man én, modsiger databasen sig selv. Når en medarbejders sidste projekt slutter, og rækken slettes, forsvinder oplysningen om, hvor vedkommende sidder. Og en nyansat, der ikke er på noget projekt, har intet sted at notere sit kontor. Det er opdaterings-, sletnings- og indsættelsesanomalierne, og deles tabellen i medarbejder–projekt og medarbejder–kontor, forsvinder alle tre.

I et teams daglige arbejde viser de anomalier sig som fejl. En faktura viser den gamle adresse, fordi én kopi blev overset. En rapport tæller en kunde to gange, fordi navnet blev tastet på to måder. Et normaliseret skema gør den slags fejl svære at skrive i første omgang, og når primærnøgler, fremmednøgler og check-constraints er erklæret, afviser databasen selv dårlige data i stedet for at stole på, at al applikationskoden husker reglerne.

Ændringer bliver også billigere. Et skema, hvor hver tabel beskriver én ting, tager som regel imod nye slags data med mindre omstrukturering: det er en mindre migrering at tilføje en kolonne til den ene tabel, der ejer en oplysning, end at finde hver tabel, der har kopieret den.

Prisen er joins. En normaliseret database besvarer de fleste spørgsmål ved at joine tabellerne sammen igen, og hvert join er arbejde for serveren. Med indeks på de kolonner, der joines på, er det arbejde som regel lille, og derfor bør en målt

langsom forespørgsel og ikke en fornemmelse afgøre, hvornår en oplysning kopieres tilbage.

Sådan ser det ud i praksis

En lille webbutik starter med én flad tabel, indlæst fra et udtræk af et regneark:

```
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
);
```

Hver række er én linje i én ordre, så en kundes navn og by gentages på hver linje i hver ordre, og et produkts navn gentages på hver linje, der sælger det. En kunde, der flytter, har snesevis af rækker at opdatere. Sletter man den sidste linje, der solgte et produkt, glemmer tabellen, at produktet findes. Status er fri tekst, så `shipped`, `Shipped` og `shipd` kan alle dukke op.

Normaliseret får hver oplysning ét hjem:

```
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)
);
```

En kundes by er nu gemt én gang, og en flytning er en opdatering af én række. Statusteksterne ligger i en lille opslagstabel med en `smallint` som nøgle, så hver ordre bærer et kort tal i stedet for en gentaget tekst, og en stavefejl i en status kan ikke slippe ind: fremmednøglen afviser enhver `status_id`, som `order_status` ikke har.

`unit_price` bliver på `order_lines`, selv om `products` allerede har en pris. Det er ingen forglemmelse. Den pris, kunden betalte, er en oplysning om salget, ikke om produktet, og den må ikke ændre sig, når kataloget gør. Normalisering spørger, hvor hver oplysning hører hjemme, og denne hører til linjen.

De to `ON DELETE`-klausuler siger, hvad en sletning betyder. Sletter man en ordre, ryger dens linjer med, for en linje giver ingen mening uden sin ordre. Sletning af en kunde, der har ordrer, fejler, fordi butikken skal beholde de ordrer. PostgreSQL's standard, `NO ACTION`, afviser også den sletning; `RESTRICT` gør hensigten tydelig, og i modsætning til `NO ACTION` kan den ikke udskydes til slutningen af transaktionen.

PostgreSQL opretter et indeks for hver primærnøgle, men ikke for den refererende side af en fremmednøgle. Dokumentationen foreslår at tilføje et, fordi sletning eller opdatering af en refereret række skal finde hver række, der peger på den:

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

`order_lines.order_id` behøver ikke noget ekstra: den er den første kolonne i tabellens primærnøgle.

Til sidst giver et view den flade form tilbage til dem, der vil have den:

```
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);
```

Rapporter kan læse `order_report` stort set, som de læste den gamle tabel, mens hver oplysning bag den er gemt én gang. PostgreSQL's vejledning argumenterer på samme måde for views generelt: et view giver en forespørgsel et navn, der kan bruges som en tabel, og lader tabellerne nedenunder ændre sig bag en grænseflade, der står fast.

De samme regler gælder i langt større skala. Den maritime datamodel fra grundlæggerens arbejde hos NextMariner er bygget op som normaliserede PostgreSQL-skemaer, organiseret efter domæne; i oktober 2026 opgjorde siden den til 348 tabeller fordelt på 34 skemaer. Små, stabile mængder som statusser, typer og kategorier blev til `SMALLINT`-opslagstabeller, hvilket holder rækkerne kompakte og joins billige, og rigtige fremmednøgler og constraints holder relationerne sammen, så integriteten er databasens opgave og ikke noget, applikationen skal huske.

Hvor det går galt

Lister pakket i én kolonne. En `tags`-kolonne med `red,large,sale`, eller en telefonkolonne med to numre, bryder første normalform. Databasen kan ikke kontrollere de enkelte værdier, pege en fremmednøgle på dem eller joine på dem, og hver forespørgsel må først splitte teksten op. Giv listen sin egen tabel med én række pr. værdi og forælderens nøgle plus værdien som nøgle: Codds fremgangsmåde fra 1970.

Historik, der normaliseres væk. Prisen på en ordrelinje eller leveringsadressen på en ordre ligner en dublet af det, produkt- og kundetabellerne indeholder. Det er den ikke: den registrerer, hvad der gjaldt, da ordren blev afgivet. Udskift den med en nøgle til det aktuelle katalog, og hver gammel faktura ændrer sig, næste gang en pris gør. Gem oplysninger om en hændelse på hændelsen.

Fremmednøgler uden indeks. Den refererede side af en fremmednøgle har altid et indeks, fordi den skal være en primærnøgle, en unik constraint eller et unikt indeks. Den refererende side har ikke, så sletning af en kunde eller ændring af et produkts SKU betyder, at hele den refererende tabel skal gennemsøges for rækker, der peger på den. Indeksér de refererende kolonner som ovenfor. PostgreSQL overlader det til dig, fordi det ikke altid er nødvendigt.

Kaskader, der sletter bilag. `ON DELETE CASCADE` fra kunder til fakturaer betyder, at én uforsigtig sletning af en kunde stille fjerner data, som forretningen skal opbevare. Lad kun sletninger kaskadere til rækker, der intet betyder uden deres forælder, som ordrelinjer, og brug `RESTRICT` eller standarden alle andre steder.

Et normaliseret design, som intet håndhæver. Tabeller, der er oprettet i hånden eller indlæst fra regneark, kommer ofte uden primærnøgle og uden fremmednøgler, så intet forhindrer en dubletrække eller en ordre til en kunde, der ikke findes. PostgreSQL kræver ikke en primærnøgle, selv om dokumentationen bemærker, at relationsteorien kræver én i hver tabel. Erklær nøglerne, så afviser databasen den dårlige række med en fejl, der nævner constrainten ved navn.

Gemte totaler, der skrider. En `order_total` -kolonne på `orders` , holdt ved siden af de linjer, den summerer, er en denormalisering, og hver kodevej, der ændrer en linje, skal også opdatere den. En kodevej, der glemmer det, efterlader en forkert total uden at udløse nogen fejl. Beregn totalen i en forespørgsel eller et view. Viser det sig at være for langsomt, så behold kopien, men vedligehold den ét sted, inden for den samme transaktion, der ændrer linjerne.

Denormalisering før måling. Joins på indekserede nøgler er billige, og at kopiere kolonner for at undgå dem bytter en lille, kendt omkostning for de anomalier, der er beskrevet ovenfor. Læs forespørgselsplanen først. Når læsninger virkelig er for langsomme, kan et materialiseret view eller en separat rapporteringsdatabase rumme den flade form, mens kildetabellerne beholder én kopi af hver oplysning. Rapportering er stedet, hvor denormalisering hører hjemme med vilje, som i stjerneskemaet i et datavarehus.

For meget opsplitning. Normalisering beder om ét hjem til hver oplysning, ikke om så mange tabeller som muligt. At dele en tabel, hvis kolonner allerede afhænger af nøglen og af intet andet, giver kun flere joins og flere tabelnavne, som alle skal lære. Et skema, der kræver et dusin joins for at vise én ordre, er ikke lettere at arbejde med end et fladt. Stop, når hver oplysning har ét hjem.

Læs mere

- A Relational Model of Data for Large Shared Data Banks — Codds artikel fra 1970 i Communications of the ACM, der introducerede relationsmodellen, normalform og normalisering.
- Multivalued Dependencies and a New Normal Form — Fagins artikel fra 1977, der definerer den flerværdige afhængighed og fjerde normalform og viser, at enhver tabel kan deles op på fjerde normalform, uden at der går information tabt.
- Constraints — PostgreSQL om primærnøgler, fremmednøgler, check-constraints og `ON DELETE`-handlingerne, og hvorfor refererende kolonner er værd at indeksere.
- Views — PostgreSQL-vejledningens argument for views som en stabil grænseflade over tabeller, hvis struktur ændrer sig.

Ordliste

Vælg ud fra jeres faktiske vilkår

Medbring et repræsentativt datasæt, de spørgsmål det skal besvare, og den aktuelle flaskehals. Vi kan aftale målinger af kvalitet, gennemløb og gendannelse.

Se, hvad det koster Kontakt os →

Gør begreberne praktiske

Spørg, hvordan »Hvad er databasenormalisering?« gælder i jeres organisation. Et konkret eksempel fra jeres nuværende system er nok til at begynde samtalen.

Se vores ydelser

welcome@engineer.company +45 50 10 61 84 København, Danmark

Tag denne side med dig

Visitkort

Lavet med kærlighed af D.

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

Copyright © 2025 – nu · Engineer ApS · Alle rettigheder forbeholdes.

Udskrevet fra https://engineer.company/da/glossary/normalization/