Engineer

Databasenormalisering

å ordne tabeller slik at hvert faktum lagres bare ett sted

Hva det er

Databasenormalisering er praksisen med å legge opp en relasjonsdatabase slik at hvert faktum lagres ett sted, i tabellen det beskriver, og alt annet viser til det med nøkkelen. Mange sier også normalisering av databaser, eller bare normalisering når sammenhengen gjør det klart. En kundes adresse står i én rad i en kundetabell; en ordre har bare kundens nøkkel, og en join setter de to sammen når en spørring trenger begge.

Både idéen og ordet kommer fra Edgar F. Codd, som foreslo relasjonsmodellen i Communications of the ACM i juni 1970. I den artikkelen betydde normalisering ett bestemt steg: å bli kvitt relasjoner som inneholder andre relasjoner, for eksempel en ansattpost med sin egen liste over tidligere stillinger og sin egen liste over barn. Codds fremgangsmåte kopierer forelderens primærnøkkel inn i hver nøstet gruppe og løfter gruppen ut i en egen relasjon, med forelderens nøkkel pluss det som identifiserer en rad innenfor gruppen som nøkkel. Det som står igjen, er på normalform, i dag kjent som første normalform: hver verdi i hver kolonne er én enkel verdi.

Codd definerte senere strengere normalformer, blant dem den som Ronald Fagins artikkel fra 1977 kaller Codds forbedrede tredje normalform eller Boyce–Codd-normalform. Fagins artikkel legger til en fjerde. Den bygger på en ny type avhengighet mellom kolonner, den flerverdige avhengigheten, som omfatter den velkjente funksjonelle avhengigheten som et spesialtilfelle, og Fagin viste at den fjerde formen er strengt sterkere enn Boyce–Codd-normalform.

De strengere formene nås ved å dele opp. En tabell erstattes av to eller flere av sine projeksjoner, tabeller med noen av kolonnene, valgt slik at en join av dem gir nøyaktig originalen tilbake. Fagins artikkel viser at den garantien holder for hans egen form: enhver tabell kan deles opp i tabeller på fjerde normalform uten at

informasjon går tapt, fordi en join av dem gjenskaper originalen. Denormalisering er det motsatte: å lagre en kopi av et faktum et annet sted med vilje, som regel for å spare en join.

Bruk begrepet i en beslutning

Bruk «Hva er databasenormalisering?» i en konkret beslutning. Vi kan knytte begrepet til systemet dere bygger eller kjøper. En struktur som samsvarer med hvordan virksomheten faktisk fungerer, slik at spørsmålene du trenger svar på senere, fortsatt er mulige å stille.

Se tjenestene våre Ta kontakt →

Undersøk antakelsene bak ordet

Bruk forklaringen og det tilknyttede arbeidet til å formulere spørsmål for prosjektet. Vi kan undersøke antakelsene sammen før et verktøy velges. Designet og driftet 6 ETL/ELT-pipelines — 6 ETL-pipelines på tvers av tre databasemotorer

Designet og driftet 6 ETL/ELT-pipelines Ta kontakt →

Hvorfor det er viktig

En tabell som lagrer det samme faktumet mange ganger, inviterer til tre slags problemer. Tenk deg én rad per ansatt og prosjekt, der den ansattes kontor står på hver rad. Når noen bytter kontor, må flere rader få den nye verdien, og glemmer man én, motsier databasen seg selv. Når en ansatts siste prosjekt avsluttes og raden slettes, forsvinner opplysningen om hvor vedkommende sitter. Og en nyansatt som ikke er med på noe prosjekt, har ingen plass å føre opp kontoret sitt. Dette er oppdaterings-, slette- og innsettingsanomaliene, og deles tabellen i ansatt–prosjekt og ansatt–kontor, forsvinner alle tre.

I et teams daglige arbeid dukker disse anomaliene opp som feil. En faktura viser den gamle adressen fordi én kopi ble glemt. En rapport teller en kunde to ganger fordi navnet ble skrevet på to måter. Et normalisert skjema gjør slike feil vanskelige å skrive i utgangspunktet, og når primærnøkler, fremmednøkler og CHECK-skranker er deklarert, avviser databasen dårlige data selv i stedet for å stole på at all applikasjonskoden husker reglene.

Endringer blir også billigere. Et skjema der hver tabell beskriver én ting, tar som regel imot nye typer data med mindre ombygging: det er en mindre migrering å legge til en kolonne i den ene tabellen som eier et faktum, enn å finne hver tabell som har kopiert det.

Prisen er joins. En normalisert database besvarer de fleste spørsmål ved å joine tabellene sammen igjen, og hver join er arbeid for serveren. Med indekser på kolonnene det joines på, er det arbeidet som regel lite, og derfor bør en målt treg spørring og ikke en magefølelse avgjøre når et faktum kopieres tilbake.

Slik ser det ut i praksis

En liten nettbutikk starter med én flat tabell, lastet inn fra et regnearkuttrekk:

```
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 rad er én linje i én ordre, så en kundes navn og by gjentas på hver linje i hver ordre, og et produkts navn gjentas på hver linje som selger det. En kunde som flytter, har dusinvis av rader å oppdatere. Sletter man den siste linjen som solgte et produkt, glemmer tabellen at produktet finnes. Statusen er fritekst, så `shipped`, `Shipped` og `shipd` kan alle dukke opp.

Normalisert får hvert faktum ett 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 nå lagret én gang, og en flytting er en oppdatering av én rad. Statusnavnene ligger i en liten oppslagstabell med en `smallint` som nøkkel, så hver ordre bærer et kort tall i stedet for en gjentatt tekst, og en feilstavet status kan ikke slippe inn: fremmednøkkelen avviser enhver `status_id` som `order_status` ikke har.

`unit_price` blir stående på `order_lines`, selv om `products` allerede har en pris. Det er ingen forglemmelse. Prisen kunden betalte, er et faktum om salget, ikke om produktet, og den skal ikke endre seg når katalogen gjør det. Normalisering spør hvor hvert faktum hører hjemme, og dette hører til linjen.

De to `ON DELETE`-leddene sier hva en sletting betyr. Sletter man en ordre, følger linjene med, for en linje betyr ingenting uten ordren sin. Å slette en kunde som har ordrer, feiler, fordi butikken må ta vare på de ordrene. PostgreSQLs standard, `NO ACTION`, avviser også den slettingen; `RESTRICT` gjør intensjonen tydelig, og i motsetning til `NO ACTION` kan den ikke utsettes til slutten av transaksjonen.

PostgreSQL lager en indeks for hver primærnøkkel, men ikke for den refererende siden av en fremmednøkkel. Dokumentasjonen foreslår å legge til en, fordi sletting eller oppdatering av en referert rad må finne hver rad som peker på den:

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

`order_lines.order_id` trenger ikke noe ekstra: den er den første kolonnen i tabellens primærnøkkel.

Til slutt gir en visning (et view) den flate formen tilbake til dem som vil ha 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 lese `order_report` omtrent som de leste den gamle tabellen, mens hvert faktum bak den er lagret én gang. Innføringen i PostgreSQL-dokumentasjonen argumenterer likt for visninger generelt: en visning gir en spørring et navn som kan brukes som en tabell, og lar tabellene under endre seg bak et grensesnitt som står fast.

De samme reglene gjelder i langt større skala. Den maritime datamodellen fra grunnleggerens arbeid hos NextMariner er lagt opp som normaliserte PostgreSQL-skjemaer organisert etter domene; i oktober 2026 oppga siden den til 348 tabeller fordelt på 34 skjemaer. Små, stabile mengder som statuser, typer og kategorier ble `SMALLINT`-oppslagstabeller, som holder radene kompakte og join-operasjonene billige, og ekte fremmednøkler og skranker holder relasjonene sammen, så integriteten er databasens jobb og ikke noe applikasjonen må huske.

Hvor det går galt

Lister pakket inn i én kolonne. En `tags`-kolonne med `red,large,sale`, eller en telefonkolonne med to numre, bryter første normalform. Databasen kan ikke kontrollere de enkelte verdiene, la en fremmednøkkel peke på dem eller joine på dem, og hver spørring må dele opp teksten først. Gi lista en egen tabell med én rad per verdi og forelderens nøkkel pluss verdien som nøkkel: Codds fremgangsmåte fra 1970.

Historikk som normaliseres bort. Prisen på en ordrelinje eller leveringsadressen på en ordre ser ut som en dublett av det produkt- og kundetabellene har. Det er den ikke: den registrerer hva som gjaldt da ordren ble lagt inn. Bytt den ut med en nøkkel til den gjeldende katalogen, så endrer hver gammel faktura seg neste gang en pris gjør det. Lagre fakta om en hendelse på hendelsen.

Fremmednøkler uten indekser. Den refererte siden av en fremmednøkkel har alltid en indeks, fordi den må være en primærnøkkel, en unik skranke eller en unik indeks. Den refererende siden har det ikke, så å slette en kunde eller endre et produkts SKU betyr å lete gjennom hele den refererende tabellen etter rader som peker dit. Indekser de refererende kolonnene som vist over. PostgreSQL overlater det til deg fordi det ikke alltid trengs.

Kaskader som sletter bilag. `ON DELETE CASCADE` fra kunder til fakturaer betyr at én uforsiktig sletting av en kunde i det stille fjerner data virksomheten må oppbevare. La bare slettinger gå i kaskade til rader som ikke betyr noe uten forelderen, som ordrelinjer, og bruk `RESTRICT`, eller standarden, alle andre steder.

Et normalisert design som ingenting håndhever. Tabeller som er laget for hånd eller lastet inn fra regneark, kommer ofte uten primærnøkkel og uten fremmednøkler, så ingenting stopper en duplikatrad eller en ordre til en kunde som ikke finnes. PostgreSQL krever ikke primærnøkkel, selv om dokumentasjonen påpeker at relasjonsteorien krever en i hver tabell. Deklarer nøklene, så avviser databasen den dårlige raden med en feilmelding som nevner skranken ved navn.

Lagrede summer som sklir. En `order_total`-kolonne på `orders`, holdt ved siden av linjene den summerer, er en denormalisering, og hver kodevei som endrer en linje, må oppdatere den også. En kodevei som glemmer det, etterlater en feil sum uten å utløse noen feil. Regn ut summen i en spørring eller en visning. Viser det seg å være for tregt, så behold kopien, men vedlikehold den ett sted, i den samme transaksjonen som endrer linjene.

Denormalisering før måling. Joins på indekserte nøkler er billige, og å kopiere kolonner for å unngå dem bytter en liten, kjent kostnad mot anomaliene som er beskrevet ovenfor. Les spørringsplanen først. Når lesingen virkelig er for treg, kan en materialisert visning eller en egen rapportdatabase holde den flate formen, mens kildetabellene beholder én kopi av hvert faktum. Rapportering er der denormalisering hører hjemme med vilje, som i stjerneskjemaet i et datavarehus.

For mye oppsplitting. Normalisering ber om ett hjem for hvert faktum, ikke om flest mulig tabeller. Å dele en tabell der kolonnene allerede avhenger av nøkkelen og ikke av noe annet, gir bare flere joins og flere tabellnavn som alle må

lære seg. Et skjema som trenger et dusin joins for å vise én ordre, er ikke lettere å jobbe med enn et flatt. Stopp når hvert faktum har ett hjem.

Les mer

- A Relational Model of Data for Large Shared Data Banks — Codds artikkel fra 1970 i Communications of the ACM, som innførte relasjonsmodellen, normalform og normalisering.
- Multivalued Dependencies and a New Normal Form — Fagins artikkel fra 1977, som definerer den flerverdige avhengigheten og fjerde normalform og viser at enhver tabell kan deles opp på fjerde normalform uten at informasjon går tapt.
- Constraints — PostgreSQL om primærnøkler, fremmednøkler, CHECK-skranker og `ON DELETE` -handlingene, og hvorfor refererende kolonner er verdt å indeksere.
- Views — innføringen i PostgreSQL-dokumentasjonen om visninger som et stabilt grensesnitt over tabeller med skiftende struktur.

Ordliste

Velg ut fra deres faktiske rammer

Ta med et representativt datasett, spørsmålene det skal besvare og dagens flaskehals. Vi kan avtale målinger av kvalitet, gjennomstrømning og gjenoppretting.

Se hva det koster Ta kontakt →

Gjør begrepene praktiske

Spør hvordan «Hva er databasenormalisering?» gjelder i organisasjonen deres. Et konkret eksempel fra dagens system er nok til å starte samtalen.

Se tjenestene våre

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

Ta med deg denne siden

Visittkort

Laget med kjærlighet av D.

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

Opphavsrett © 2025 – nå · Engineer ApS · Alle rettigheter forbeholdt.

Skrevet ut fra https://engineer.company/nb/glossary/normalization/