Engineer

Databasnormalisering

att ordna tabeller så att varje uppgift bara lagras en gång

Vad det är

Databasnormalisering är praxisen att lägga upp en relationsdatabas så att varje uppgift lagras på ett ställe, i den tabell den beskriver, och allt annat hänvisar till den med dess nyckel. Man säger också normalisering av databaser, eller bara normalisering när sammanhanget gör det tydligt. En kunds adress står i en rad i en kundtabell; en order har bara kundens nyckel, och en join för samman de två när en fråga behöver båda.

Både idén och ordet kommer från Edgar F. Codd, som föreslog relationsmodellen i Communications of the ACM i juni 1970. I den artikeln betydde normalisering ett bestämt steg: att bli av med relationer som innehåller andra relationer, till exempel en post för en anställd med sin egen lista över tidigare tjänster och sin egen lista över barn. Codds tillvägagångssätt kopierar förälderns primärnyckel in i varje nästlad grupp och lyfter ut gruppen i en egen relation, med förälderns nyckel plus det som identifierar en rad inom gruppen som nyckel. Det som blir kvar är i normalform, i dag känd som första normalform: varje värde i varje kolumn är ett enda, enkelt värde.

Codd definierade sedan striktare normalformer, bland dem den som Ronald Fagins artikel från 1977 kallar Codds förbättrade tredje normalform eller Boyce–Codd-normalform. Fagins artikel lägger till en fjärde. Den bygger på en ny sorts beroende mellan kolumner, det flervärda beroendet, som omfattar det välkända funktionella beroendet som ett specialfall, och Fagin visade att den fjärde formen är strikt starkare än Boyce–Codd-normalform.

De striktare formerna nås genom uppdelning. En tabell ersätts av två eller fler av sina projektioner, tabeller med några av dess kolumner, valda så att en join av dem ger tillbaka exakt originalet. Fagins artikel visar att den garantin gäller för hans egen form: varje tabell kan delas upp i tabeller i fjärde normalform utan att

information går förlorad, eftersom en join av dem återskapar originalet. Denormalisering är motsatsen: att avsiktligt spara en kopia av en uppgift på ett andra ställe, oftast för att slippa en join.

Använd begreppet i ett beslut

Använd ”Vad är databasnormalisering?” i ett konkret beslut. Vi kan knyta begreppet till systemet ni bygger eller köper. En struktur som speglar hur verksamheten faktiskt fungerar, så att de frågor du behöver svar på senare fortfarande går att ställa.

Se våra tjänster Hör av er →

Granska antagandena bakom ordet

Använd förklaringen och det länkade arbetet för att formulera frågor till projektet. Vi kan undersöka antagandena tillsammans innan ett verktyg väljs. Utformade och drev 6 ETL/ELT-pipelines — 6 ETL-pipelines över tre databasmotorer

Utformade och drev 6 ETL/ELT-pipelines Hör av er →

Varför det spelar roll

En tabell som lagrar samma uppgift många gånger bäddar för tre sorters problem. Tänk dig en rad per anställd och projekt, där den anställdas kontor står på varje rad. När någon byter kontor behöver flera rader det nya värdet, och missar man en motsäger databasen sig själv. När en anställds sista projekt avslutas och raden tas bort försvinner uppgiften om var personen sitter. Och en nyanställd som inte är med i något projekt har ingenstans att föra in sitt kontor. Det här är uppdaterings-, borttagnings- och insättningsanomalierna, och delas tabellen i anställd–projekt och anställd–kontor försvinner alla tre.

I ett teams vardag visar sig de här anomalierna som buggar. En faktura visar den gamla adressen för att en kopia missades. En rapport räknar en kund två gånger för att namnet skrevs på två sätt. Ett normaliserat schema gör sådana buggar svåra att skriva från början, och när primärnycklar, främmande nycklar och CHECK-villkor är deklarerade avvisar databasen själv dåliga data i stället för att lita på att all applikationskod minns reglerna.

Ändringar blir också billigare. Ett schema där varje tabell beskriver en sak tar oftast emot nya slags data med mindre ombyggnad: det är en mindre migrering att lägga till en kolumn i den enda tabell som äger en uppgift än att hitta varje tabell som har kopierat den.

Priset är joinar. En normaliserad databas besvarar de flesta frågor genom att joina ihop tabellerna igen, och varje join är arbete för servern. Med index på kolumnerna som joinas är det arbetet oftast litet, och därför bör en uppmätt långsam fråga och inte en magkänsla avgöra när en uppgift kopieras tillbaka.

Så ser det ut i praktiken

En liten webbutik börjar med en enda platt tabell, inläst från en export av ett kalkylark:

```
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
);
```

Varje rad är en rad i en order, så en kunds namn och stad upprepas på varje rad i varje order, och en produkts namn upprepas på varje rad som säljer den. En kund som flyttar har dussintals rader att uppdatera. Tar man bort den sista raden som sålde en produkt glömmer tabellen att produkten finns. Statusen är fritext, så `shipped`, `Shipped` och `shipd` kan alla dyka upp.

Normaliserat får varje uppgift ett eget hem:

```
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 kunds stad lagras nu en gång, och en flytt är en uppdatering av en rad. Statustexterna ligger i en liten uppslagstabell med en `smallint` som nyckel, så varje order bär ett kort tal i stället för en upprepad sträng, och en felstavad status kan inte slinka in: den främmande nyckeln avvisar varje `status_id` som `order_status` inte har.

`unit_price` stannar på `order_lines`, fast `products` redan har ett pris. Det är inget förbiseende. Priset kunden betalade är en uppgift om försäljningen, inte om produkten, och det får inte ändras när katalogen gör det. Normalisering frågar var varje uppgift hör hemma, och den här hör till orderraden.

De två `ON DELETE`-satserna säger vad en borttagning betyder. Tar man bort en order följer dess rader med, eftersom en orderrad inte betyder något utan sin order. Att ta bort en kund som har ordrar misslyckas, eftersom butiken måste behålla de ordrarna. PostgreSQL:s standard, `NO ACTION`, avvisar också den borttagningen; `RESTRICT` gör avsikten tydlig, och till skillnad från `NO ACTION` kan den inte skjutas upp till slutet av transaktionen.

PostgreSQL skapar ett index för varje primärnyckel, men inte för den refererande sidan av en främmande nyckel. Dokumentationen föreslår att man lägger till ett, eftersom en borttagning eller uppdatering av en refererad rad måste hitta varje rad som pekar på den:

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

`order_lines.order_id` behöver inget extra: den är den första kolumnen i tabellens primärnyckel.

Till sist ger en vy den platta formen tillbaka åt dem som vill 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 läsa `order_report` ungefär som de läste den gamla tabellen, medan varje uppgift bakom den lagras en gång. Introduktionen i PostgreSQL-dokumentationen argumenterar likadant för vyer i allmänhet: en vy ger en fråga ett namn som kan användas som en tabell, och låter tabellerna därunder ändras bakom ett gränssnitt som ligger fast.

Samma regler gäller i mycket större skala. Den maritima datamodellen från grundarens arbete hos NextMariner är uppbyggd som normaliserade PostgreSQL-scheman organiserade per domän; i oktober 2026 angav sidan den till 348 tabeller fördelade på 34 scheman. Små, stabila mängder som statusar, typer och kategorier blev uppslagstabeller med `SMALLINT`, vilket håller raderna kompakta och joinarna billiga, och riktiga främmande nycklar och villkor håller ihop relationerna, så att integriteten är databasens uppgift och inte något som applikationen måste komma ihåg.

Var det går fel

Listor packade i en kolumn. En `tags`-kolumn med `red,large,sale`, eller en telefonkolumn med två nummer, bryter mot första normalform. Databasen kan inte kontrollera de enskilda värdena, låta en främmande nyckel peka på dem eller joina på dem, och varje fråga måste först dela upp strängen. Ge listan en egen tabell med en rad per värde och förälderns nyckel plus värdet som nyckel: Codds tillvägagångssätt från 1970.

Historik som normaliseras bort. Priset på en orderrad eller leveransadressen på en order ser ut som en dubblett av det produkt- och kundtabellerna innehåller. Det är det inte: det registrerar vad som gällde när ordern lades. Byt ut det mot en nyckel till den aktuella katalogen, så ändras varje gammal faktura nästa gång ett pris gör det. Spara uppgifter om en händelse på händelsen.

Främmande nycklar utan index. Den refererade sidan av en främmande nyckel har alltid ett index, eftersom den måste vara en primärnyckel, ett unikhetsvillkor eller ett unikt index. Den refererande sidan har det inte, så att ta bort en kund eller ändra en produkts SKU innebär att hela den refererande tabellen söks igenom efter rader som pekar dit. Indexera de refererande kolumnerna, som ovan. PostgreSQL lämnar det åt dig eftersom det inte alltid behövs.

Kaskader som raderar sådant som ska sparas. `ON DELETE CASCADE` från kunder till fakturor betyder att en vårdslös borttagning av en kund i tysthet tar bort data som verksamheten måste bevara. Låt bara borttagningar kaskadera till rader som inte betyder något utan sin förälder, som orderrader, och använd `RESTRICT`, eller standarden, överallt annars.

En normaliserad design som inget upprätthåller. Tabeller som skapats för hand eller lästs in från kalkylark kommer ofta utan primärnyckel och utan främmande nycklar, så inget stoppar en dubblettrad eller en order till en kund som inte finns. PostgreSQL kräver ingen primärnyckel, även om dokumentationen påpekar att relationsteorin kräver en i varje tabell. Deklarera nycklarna, så avvisar databasen den dåliga raden med ett fel som namnger villkoret.

Lagrade summor som glider isär. En `order_total`-kolumn på `orders`, som hålls bredvid raderna den summerar, är en denormalisering, och varje kodväg som ändrar en rad måste också uppdatera den. En kodväg som glömmer det lämnar en felaktig summa utan att något fel uppstår. Räkna ut summan i en fråga eller en vy. Visar det sig vara för långsamt, behåll kopian, men underhåll den på ett ställe, i samma transaktion som ändrar raderna.

Denormalisering före mätning. Joinar på indexerade nycklar är billiga, och att kopiera kolumner för att slippa dem byter en liten, känd kostnad mot de anomalier som beskrivs ovan. Läs frågeplanen först. När läsningarna verkligen är för långsamma kan en materialiserad vy eller en separat rapportdatabas hålla den platta formen medan källtabellerna behåller en kopia av varje uppgift. Rapportering är där denormalisering hör hemma med avsikt, som i stjärnschemat i ett datalager.

För mycket uppdelning. Normalisering ber om ett hem för varje uppgift, inte om så många tabeller som möjligt. Att dela en tabell vars kolumner redan beror på nyckeln och på inget annat ger bara fler joinar och fler tabellnamn som alla måste lära sig. Ett schema som behöver ett dussin joinar för att visa en order är inte lättare att arbeta med än ett platt. Sluta när varje uppgift har ett enda hem.

Läs mer

- A Relational Model of Data for Large Shared Data Banks — Codds artikel från 1970 i Communications of the ACM, som introducerade relationsmodellen, normalform och normalisering.
- Multivalued Dependencies and a New Normal Form — Fagins artikel från 1977, som definierar det flervärda beroendet och fjärde normalform och visar att varje tabell kan delas upp i fjärde normalform utan att information går förlorad.
- Constraints — PostgreSQL om primärnycklar, främmande nycklar, CHECK-villkor och `ON DELETE` -åtgärderna, och varför refererande kolumner är värda att indexera.
- Views — introduktionen i PostgreSQL-dokumentationen om vyer som ett stabilt gränssnitt över tabeller vars struktur ändras.

Ordlista

Välj utifrån era faktiska förutsättningar

Ta med ett representativt dataset, frågorna det ska besvara och dagens flaskhals. Vi kan avtala mått på kvalitet, genomströmning och återställning.

Se vad det kostar Hör av er →

Gör begreppen praktiska

Fråga hur ”Vad är databasnormalisering?” gäller i er organisation. Ett konkret exempel från ert nuvarande system räcker för att börja samtalet.

Se våra tjänster

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

Ta med dig den här sidan

Visitkort

Omsorgsfullt skapad med kärlek av D.

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

Upphovsrätt © 2025 – nu · Engineer ApS · Alla rättigheter förbehållna.

Utskrivet från https://engineer.company/sv/glossary/normalization/