Engineer

Datenbanknormalisierung

Tabellen ordnen: Fakten einmal unter ihrem Schlüssel ablegen

Was es ist

Datenbanknormalisierung bedeutet, eine relationale Datenbank so aufzubauen, dass jeder Fakt genau einmal in der zugehörigen Tabelle gespeichert wird. Alles andere verweist über seinen Schlüssel darauf. Im britischen Englisch heißt es database normalisation. Die Adresse eines Kunden steht in einer Zeile einer Kundentabelle. Ein Auftrag trägt nur den Kundenschlüssel, und ein Join bringt beides zusammen, wenn eine Abfrage beides benötigt.

Idee und Wort stammen von Edgar F. Codd, der das relationale Modell im Juni 1970 in Communications of the ACM vorschlug. In diesem Aufsatz bedeutete Normalisierung einen bestimmten Schritt: Relationen zu beseitigen, die andere Relationen enthalten, etwa einen Beschäftigtendatensatz mit einer eigenen Liste früherer Arbeitsstellen und einer eigenen Kinderliste. Codds Verfahren kopiert den Primärschlüssel des übergeordneten Datensatzes in jede verschachtelte Gruppe und hebt die Gruppe in eine eigene Relation heraus. Deren Schlüssel besteht aus dem übergeordneten Schlüssel und dem, was eine Zeile innerhalb der Gruppe identifiziert. Übrig bleibt die Normalform, heute erste Normalform genannt: Jeder Wert in jeder Spalte ist ein einzelner, einfacher Wert.

Codd definierte später strengere Normalformen. Dazu gehört diejenige, die Ronald Fagins Aufsatz von 1977 Codds verbesserte dritte Normalform nennt, auch als Boyce–Codd-Normalform bekannt. Fagins Aufsatz ergänzt eine vierte. Sie beruht auf einer neuen Abhängigkeit zwischen Spalten, der mehrwertigen Abhängigkeit, die die vertraute funktionale Abhängigkeit als Sonderfall einschließt. Fagin zeigte, dass die vierte Form echt stärker ist als die Boyce–Codd-Normalform.

Die strengeren Formen werden durch Zerlegung erreicht. Eine Tabelle wird durch zwei oder mehr ihrer Projektionen ersetzt: Tabellen aus einem Teil ihrer Spalten, so gewählt, dass ein Join exakt das Original zurückliefert. Fagins Aufsatz zeigt diese Garantie für seine eigene Form: Jede Tabelle lässt sich ohne Informationsverlust in Tabellen der vierten Normalform zerlegen, weil ihr Join das Original wiederherstellt. Denormalisierung ist die Umkehrung: einen Fakt absichtlich an einer zweiten Stelle zu kopieren, gewöhnlich um einen Join zu sparen.

Das Konzept für eine Entscheidung nutzen

Nutzen Sie „Was ist Datenbanknormalisierung?“ für eine konkrete Entscheidung. Wir können helfen, das Konzept mit dem System in Verbindung zu bringen, das Sie entwickeln oder einkaufen. Eine Struktur, die zur tatsächlichen Arbeitsweise Ihres Unternehmens passt, damit sich auch spätere Fragen noch beantworten lassen.

Unsere Leistungen entdecken Kontakt aufnehmen →

Die Annahmen hinter dem Begriff prüfen

Nutzen Sie die Erklärung und die verlinkten Arbeiten, um Fragen für Ihr Projekt zu formulieren. Wir können die Annahmen gemeinsam mit Ihnen prüfen, bevor ein Werkzeug ausgewählt wird. 6 ETL/ELT-Pipelines entworfen und betrieben — 6 ETL-Pipelines über drei Datenbankengines hinweg

6 ETL/ELT-Pipelines entworfen und betrieben Kontakt aufnehmen →

Warum es wichtig ist

Eine Tabelle, die denselben Fakt oft speichert, führt leicht zu drei Arten von Problemen. Stellen Sie sich eine Zeile pro Beschäftigtem und Projekt vor, mit dem Büro des Beschäftigten auf jeder Zeile. Bei einem Bürowechsel müssen mehrere Zeilen den neuen Wert erhalten. Eine vergessene Zeile lässt die Datenbank sich selbst widersprechen. Endet das letzte Projekt einer Person und wird seine Zeile gelöscht, verschwindet damit auch die Information über ihr Büro. Für einen Neuzugang ohne Projekt gibt es wiederum keinen Platz, um ein Büro zu erfassen. Das sind Änderungs-, Lösch- und Einfügeanomalien. Die Zerlegung in Beschäftigte–Projekte und Beschäftigte–Büros beseitigt alle drei.

Im Teamalltag erscheinen diese Anomalien als Fehler. Eine Rechnung zeigt die alte Adresse, weil eine Kopie vergessen wurde. Ein Bericht zählt einen Kunden doppelt, weil sein Name auf zwei Arten eingegeben wurde. Ein normalisiertes Schema macht es schwer, solche Fehler überhaupt einzubauen. Mit deklarierten Primärschlüsseln, Fremdschlüsseln und Prüfbedingungen weist die Datenbank ungültige Daten selbst zurück, statt darauf zu vertrauen, dass jeder Anwendungsteil die Regeln beachtet.

Auch Änderungen werden günstiger. Beschreibt jede Tabelle eine Sache, kann das Schema neue Datenarten gewöhnlich mit weniger Umbau aufnehmen: Eine Spalte in der einen Tabelle zu ergänzen, zu der ein Fakt gehört, ist eine kleinere Migration, als jede Tabelle mit einer Kopie zu suchen.

Der Preis sind Joins. Eine normalisierte Datenbank beantwortet die meisten Fragen, indem sie Tabellen wieder zusammenführt. Jeder Join bedeutet Arbeit für den Server. Mit Indizes auf den verbundenen Spalten bleibt diese Arbeit meist klein. Deshalb sollte eine gemessen langsame Abfrage darüber entscheiden, wann ein Fakt wieder hineinkopiert wird, nicht ein Bauchgefühl.

Wie es in der Praxis aussieht

Ein kleiner Laden beginnt mit einer flachen Tabelle aus dem Export einer Tabellenkalkulation:

```
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
);
```

Jede Zeile ist eine Position eines Auftrags. Name und Stadt eines Kunden wiederholen sich deshalb auf jeder Position jedes Auftrags, der Produktname auf jeder Position mit diesem Produkt. Zieht ein Kunde um, müssen Dutzende Zeilen geändert werden. Wird die letzte Position gelöscht, in der ein Produkt verkauft wurde, geht die Information verloren, dass dieses Produkt existiert. Der Status ist Freitext: `shipped`, `Shipped` und `shipd` können alle vorkommen.

Nach der Normalisierung erhält jeder Fakt genau einen Platz:

```
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)
);
```

Die Stadt eines Kunden ist jetzt einmal gespeichert. Bei einem Umzug muss nur eine Zeile aktualisiert werden. Die Statusbezeichnungen liegen in einer kleinen Nachschlagetabelle mit einem `smallint` als Schlüssel. Jeder Auftrag trägt damit eine kurze Zahl statt einer wiederholten Zeichenfolge. Ein falsch geschriebener Status kommt nicht hinein: Der Fremdschlüssel weist jede `status_id` zurück, die `order_status` nicht enthält.

`unit_price` bleibt auf `order_lines`, obwohl `products` schon einen Preis hat. Das ist kein Versäumnis. Der bezahlte Preis ist ein Fakt über den Verkauf, nicht über das Produkt. Er darf sich nicht ändern, wenn der Katalog sich ändert. Normalisierung fragt, wohin jeder Fakt gehört. Dieser gehört zur Position.

Die beiden `ON DELETE`-Klauseln sagen, was eine Löschung bedeutet. Wird ein Auftrag gelöscht, verschwinden seine Positionen mit ihm, weil eine Position ohne ihren Auftrag nichts bedeutet. Einen Kunden mit Aufträgen zu löschen scheitert: Diese Aufträge sind Belege, die der Laden aufbewahren muss. Auch

PostgreSQLs Standard `NO ACTION` weist die Löschung zurück. `RESTRICT` hält die Absicht fest und kann anders als `NO ACTION` nicht bis zum Transaktionsende aufgeschoben werden.

PostgreSQL erstellt für jeden Primärschlüssel einen Index, nicht aber für die verweisende Seite eines Fremdschlüssels. Die Dokumentation empfiehlt einen zusätzlichen Index dort: Beim Löschen oder Ändern einer referenzierten Zeile muss jede auf sie verweisende Zeile gefunden werden:

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

`order_lines.order_id` braucht nichts Zusätzliches: Es ist die führende Spalte des Primärschlüssels dieser Tabelle.

Schließlich stellt eine Sicht die flache Form für alle wieder her, die sie benötigen:

```
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);
```

Berichte können `order_report` fast wie die alte Tabelle lesen, während jeder Fakt dahinter einmal gespeichert bleibt. Das PostgreSQL-Tutorial begründet Sichten allgemein genauso: Eine Sicht gibt einer Abfrage einen wie eine Tabelle nutzbaren Namen. Die Tabellen darunter können sich hinter einer gleichbleibenden Schnittstelle verändern.

Dieselben Regeln gelten in weit größerem Maßstab. Im Rahmen der Arbeit der Gründerin unter NextMariner wurde das maritime Datenmodell als normalisierte, nach Domänen gegliederte PostgreSQL-Schemata entworfen. Die zugehörige Seite bezifferte es im Oktober 2026 auf 348 Tabellen in 34 Schemata. Kleine, stabile Mengen wie Statuswerte, Typen und Kategorien wurden zu `SMALLINT` -

Nachschlagetabellen. Das hält Zeilen kompakt und Joins günstig. Echte Fremdschlüssel und Constraints sichern die Beziehungen. Integrität ist damit Aufgabe der Datenbank statt etwas, das die Anwendung sich merken muss.

Wo es schiefgeht

Listen in einer Spalte. Eine `tags` -Spalte mit `red,large,sale` oder eine Telefonspalte mit zwei Nummern verletzt die erste Normalform. Die Datenbank kann die einzelnen Werte weder prüfen noch per Fremdschlüssel referenzieren oder für Joins verwenden. Jede Abfrage muss zuerst die Zeichenfolge zerlegen. Geben Sie der Liste eine eigene Tabelle mit einer Zeile pro Wert, deren Schlüssel aus dem übergeordneten Schlüssel und dem Wert besteht: Codds Verfahren von 1970.

Die Historie wird wegnormalisiert. Der Preis einer Auftragsposition oder die Lieferadresse eines Auftrags wirkt wie eine Kopie der Werte aus Produkt- und Kundentabellen. Eine solche Angabe ist keine Kopie: Sie hält fest, was bei der Auftragserteilung galt. Ersetzt man sie durch einen Schlüssel auf den aktuellen Katalog, ändert die nächste Preisänderung jede alte Rechnung. Halten Sie Fakten über ein Ereignis am Ereignis fest.

Fremdschlüssel ohne Indizes. Die referenzierte Seite eines Fremdschlüssels ist immer indiziert, weil sie ein Primärschlüssel, eine Eindeutigkeitsbedingung oder ein eindeutiger Index sein muss. Die verweisende Seite ist es nicht. Einen Kunden zu löschen oder die SKU eines Produkts zu ändern bedeutet deshalb, die ganze verweisende Tabelle nach betroffenen Zeilen zu durchsuchen. Indizieren Sie wie oben die verweisenden Spalten. PostgreSQL überlässt das Ihnen, weil es nicht immer nötig ist.

Kaskaden löschen Belege. `ON DELETE CASCADE` von Kunden zu Rechnungen bedeutet, dass eine unbedachte Kundenlöschung unbemerkt Belege entfernt, die der Betrieb aufbewahren muss. Kaskadieren Sie nur zu Zeilen, die ohne ihren übergeordneten Datensatz nichts bedeuten, etwa Auftragspositionen. Verwenden Sie überall sonst `RESTRICT` oder den Standard.

Ein normalisierter Entwurf, den nichts durchsetzt. Von Hand angelegte oder aus Tabellenkalkulationen geladene Tabellen haben oft weder Primär- noch Fremdschlüssel. Nichts verhindert dann eine doppelte Zeile oder einen Auftrag für einen nicht existierenden Kunden. PostgreSQL verlangt keinen Primärschlüssel, obwohl seine Dokumentation darauf hinweist, dass die

relationale Theorie einen in jeder Tabelle verlangt. Deklarieren Sie die Schlüssel. Die Datenbank weist die fehlerhafte Zeile dann mit einer Fehlermeldung zurück, die den Constraint nennt.

Zwischengespeicherte Summen laufen auseinander. Eine Spalte `order_total` auf `orders` neben den summierten Positionen ist eine Denormalisierung. Jeder Codepfad, der eine Position ändert, muss auch die Summe ändern. Vergisst einer das, ist die Summe ohne Fehlermeldung falsch. Berechnen Sie sie in einer Abfrage oder Sicht. Erweist sich das als zu langsam, behalten Sie die Kopie, pflegen Sie sie aber an einer Stelle innerhalb derselben Transaktion wie die Änderungen an den Positionen.

Denormalisieren vor dem Messen. Joins über indizierte Schlüssel sind günstig. Spalten zu kopieren, um sie zu vermeiden, tauscht kleine, bekannte Kosten gegen die beschriebenen Anomalien. Lesen Sie zuerst den Abfrageplan. Sind die Lesezugriffe tatsächlich zu langsam, kann eine materialisierte Sicht oder eine getrennte Berichtsdatenbank die flache Form halten, während die Quelltabellen jeden Fakt einmal speichern. Bei Berichten hat Denormalisierung absichtlich ihren Platz, etwa im Sternschema eines Data Warehouse.

Zu weit zerlegt. Normalisierung verlangt einen Platz pro Fakt, nicht möglichst viele Tabellen. Eine Tabelle zu zerlegen, deren Spalten bereits nur von ihrem Schlüssel abhängen, fügt lediglich Joins und Tabellennamen hinzu, die alle lernen müssen. Ein Schema, das für einen Auftrag ein Dutzend Joins braucht, ist nicht leichter zu benutzen als ein flaches. Hören Sie auf, sobald jeder Fakt genau einen Platz hat.

Weiterführende Lektüre

- Ein relationales Datenmodell für große gemeinsam genutzte Datenbanken — Codds Aufsatz von 1970 in Communications of the ACM, der das relationale Modell, die Normalform und Normalisierung einführte.
- Mehrwertige Abhängigkeiten und eine neue Normalform — Fagins Aufsatz von 1977: die Definition der mehrwertigen Abhängigkeit und vierten Normalform sowie der Nachweis, dass sich jede Tabelle ohne Informationsverlust in die vierte Normalform zerlegen lässt.
- Constraints — PostgreSQL über Primärschlüssel, Fremdschlüssel, Prüfbedingungen und `ON DELETE`-Aktionen sowie den Nutzen von Indizes auf verweisenden Spalten.
- Sichten — das PostgreSQL-Tutorial über Sichten als stabile Schnittstelle über Tabellen, deren Struktur sich verändert.

Glossar

Für Ihre tatsächlichen Rahmenbedingungen auswählen

Bringen Sie einen repräsentativen Datensatz, die Fragen, die er beantworten muss, und den aktuellen Engpass mit. Wir können vereinbaren, wie Qualität, Durchsatz und Wiederherstellung gemessen werden.

Preise ansehen Kontakt aufnehmen →

Die Begriffe praktisch nutzbar machen

Fragen Sie, wie „Was ist Datenbanknormalisierung?“ auf Ihre Organisation zutrifft. Ein konkretes Beispiel aus Ihrem bestehenden System reicht aus, um das Gespräch zu beginnen.

Unsere Leistungen entdecken

welcome@engineer.company +45 50 10 61 84 Kopenhagen, Dänemark

Nehmen Sie diese Seite mit

Kontaktkarte

Mit Sorgfalt und Liebe gestaltet von D.

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

Copyright © 2025 – heute · Engineer ApS · Alle Rechte vorbehalten.

Gedruckt von https://engineer.company/de/glossary/normalization/