Engineer

Normalisation de base de données

Tables où chaque fait n’est stocké qu’une fois, sous sa clé

Ce que c’est

La normalisation de base de données consiste à organiser une base relationnelle pour stocker chaque fait une seule fois, dans la table qu’il décrit, tout le reste y faisant référence par sa clé. En anglais britannique, database normalization s’écrit database normalisation. L’adresse d’un client tient dans une ligne de la table des clients ; une commande ne porte que la clé du client et une jointure les réunit si une requête a besoin des deux.

L’idée comme le mot viennent d’Edgar F. Codd, qui a proposé le modèle relationnel dans Communications of the ACM en juin 1970. Dans cet article, la normalisation était une étape précise : supprimer l’imbrication de relations à l’intérieur d’autres relations, comme une fiche de salarié portant ses listes d’emplois antérieurs et d’enfants. La procédure de Codd copie la clé primaire du parent dans chaque groupe imbriqué et extrait celui-ci dans sa propre relation, identifiée par la clé du parent et ce qui distingue une ligne du groupe. Ce qui reste est en forme normale, aujourd’hui première forme normale : chaque colonne ne contient, dans chaque ligne, qu’une seule valeur simple.

Codd a ensuite défini des formes plus strictes, dont celle que l’article de Ronald Fagin de 1977 appelle la troisième forme normale améliorée de Codd, aussi connue comme forme normale de Boyce–Codd. Fagin en ajoute une quatrième. Elle repose sur une nouvelle dépendance entre colonnes, la dépendance multivaluée, qui comprend la dépendance fonctionnelle habituelle comme cas particulier ; Fagin a montré que sa quatrième forme est strictement plus forte que celle de Boyce–Codd.

Les formes plus strictes s’atteignent par décomposition. Une table est remplacée par au moins deux de ses projections, tables composées de certaines de ses colonnes, choisies pour que leur jointure restitue exactement l’originale. L’article

de Fagin établit cette garantie pour sa forme : toute table se décompose en tables de quatrième forme normale sans perte d'information, car les joindre reconstitue l'originale. La dénormalisation est l'inverse : conserver volontairement une copie d'un fait ailleurs, généralement pour éviter une jointure.

Utilisez le concept pour prendre une décision

Utilisez « Que désigne la normalisation de base de données ? » pour éclairer une décision réelle. Nous pouvons vous aider à relier ce concept au système que vous construisez ou achetez. Une structure fidèle au fonctionnement de l'entreprise, qui permet encore de poser demain les questions dont vous aurez besoin.

Découvrir nos prestations Contactez-nous →

Vérifiez les hypothèses derrière le terme

Appuyez-vous sur l'explication et les réalisations liées pour formuler les questions de votre projet. Nous pouvons examiner les hypothèses avec vous avant le choix d'un outil. 6 pipelines ETL/ELT conçus et exploités — 6 pipelines ETL sur trois moteurs de bases de données

6 pipelines ETL/ELT conçus et exploités Contactez-nous →

Pourquoi c'est important

Stocker le même fait plusieurs fois expose à trois types de problèmes. Imaginons une ligne par salarié et projet, avec le bureau du salarié répété sur chaque ligne. Un changement de bureau demande de modifier plusieurs lignes : en manquer une rend la base contradictoire. Quand le dernier projet d'un salarié se termine et que sa ligne est supprimée, l'information sur son bureau disparaît aussi. Et une nouvelle recrue sans projet n'a aucune ligne où enregistrer son bureau. Ce sont les anomalies de mise à jour, suppression et insertion ; décomposer la table en deux tables, salarié–projet et salarié–bureau, élimine les trois.

Au quotidien, ces anomalies deviennent des bugs. Une facture affiche l'ancienne adresse car une copie a été oubliée. Un rapport compte deux fois un client dont le nom a été saisi de deux façons. Un schéma normalisé rend difficile l'introduction de tels bugs ; lorsque les clés primaires, les clés étrangères et les contraintes de vérification sont déclarées, la base refuse elle-même les mauvaises données au lieu de faire confiance à chaque morceau de code pour se souvenir des règles.

Le changement coûte aussi moins cher. Un schéma dont chaque table décrit une chose accueille généralement de nouvelles données avec moins de restructuration : ajouter une colonne à la table propriétaire d'un fait est une migration plus petite que retrouver toutes les tables qui le copient.

Le prix se paie en jointures. Une base normalisée répond à la plupart des questions en réunissant les tables, et chaque jointure demande du travail au serveur. Avec des index sur les colonnes jointes, ce travail est généralement limité : c'est donc une requête dont la lenteur a été mesurée, pas une intuition, qui devrait décider de recopier un fait.

À quoi cela ressemble en pratique

Une petite boutique commence par une table plate, chargée depuis un export de tableur :

```
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
);
```

Chaque ligne représente une ligne de commande : le nom et la ville du client se répètent donc sur chaque ligne de chaque commande, et le nom du produit sur chaque ligne qui le vend. Un déménagement impose de modifier des dizaines de lignes. Supprimer la dernière ligne vendant un produit fait oublier son existence. Le statut est du texte libre : `shipped`, `Shipped` et `shipd` peuvent tous apparaître.

Après normalisation, chaque fait n'a qu'un emplacement :

```
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)
);
```

La ville du client est maintenant stockée une seule fois et un déménagement ne modifie qu'une ligne. Les libellés de statut sont dans une petite table de référence identifiée par un `smallint` : chaque commande porte un petit entier plutôt qu'une chaîne répétée, et un statut mal orthographié ne peut être enregistré. La clé étrangère refuse tout `status_id` absent de `order_status` .

`unit_price` reste sur `order_lines` , même si `products` porte déjà un prix. Ce n'est pas un oubli. Le prix payé est un fait de la vente, pas du produit, et ne doit pas changer avec le catalogue. La normalisation demande où placer chaque fait ; celui-ci appartient à la ligne.

Les deux clauses `ON DELETE` définissent le sens d'une suppression. Supprimer une commande emporte ses lignes, qui n'ont pas de sens sans elle. Supprimer un client ayant des commandes échoue : la boutique doit conserver ces enregistrements. Le choix par défaut de PostgreSQL, `NO ACTION` , refuse aussi cette suppression ; `RESTRICT` exprime l'intention et, contrairement à `NO ACTION` , ne peut être différé à la fin de la transaction.

PostgreSQL crée un index pour chaque clé primaire, pas pour le côté référençant d'une clé étrangère. La documentation suggère d'en ajouter un : supprimer ou modifier une ligne référencée demande de retrouver toutes les lignes qui

pointent vers elle :

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

`order_lines.order_id` ne demande rien de plus : c'est la première colonne de la clé primaire de sa table.

Enfin, une vue restitue la forme plate à qui la souhaite :

```
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);
```

Les rapports peuvent lire `order_report` presque comme l'ancienne table, alors que chaque fait sous-jacent n'est stocké qu'une fois. Le tutoriel PostgreSQL défend de la même façon les vues en général : nommer une requête permet de l'utiliser comme une table et laisse les tables changer derrière une interface stable.

Les mêmes règles s'appliquent à bien plus grande échelle. Dans le travail de la fondatrice chez NextMariner, le modèle de données maritime a été organisé en schémas PostgreSQL normalisés et regroupés par domaine. La page qui le présente indiquait 348 tables dans 34 schémas en octobre 2026. Les petits ensembles stables, comme statuts, types et catégories, sont devenus des tables de référence `SMALLINT`, gardant les lignes compactes et les jointures peu coûteuses. De vraies clés étrangères et contraintes maintiennent les relations : l'intégrité relève ainsi de la base plutôt que de règles dont l'application doit se souvenir.

Où les problèmes surviennent

Des listes sont tassées dans une colonne. Une colonne `tags` contenant `red,large,sale`, ou une colonne de téléphone contenant deux numéros, viole la première forme normale. La base ne peut vérifier les valeurs séparément, les viser par clé étrangère ni les joindre ; chaque requête doit d'abord découper la chaîne. Donnez à la liste sa propre table, une ligne par valeur, identifiée par la clé du parent et la valeur : la procédure de Codd de 1970.

L'historique disparaît avec la normalisation. Le prix d'une ligne de commande ou l'adresse de livraison d'une commande ressemble à un doublon des données des tables produits et clients. Ce n'est pas un doublon : cette donnée conserve ce qui était vrai au moment de la commande. Remplacez cette donnée par une clé du catalogue actuel et toutes les anciennes factures changeront à la prochaine modification de prix. Conservez les faits d'un événement sur cet événement.

Des clés étrangères n'ont pas d'index. Le côté référencé est toujours indexé, car il doit être une clé primaire, une contrainte unique ou un index unique. Le côté référençant ne l'est pas : supprimer un client ou modifier le SKU d'un produit impose donc de parcourir toute la table référençante pour trouver les lignes concernées. Indexez ces colonnes comme ci-dessus ; PostgreSQL vous laisse cette responsabilité, car cet index n'est pas toujours nécessaire.

Des cascades effacent les enregistrements à conserver. `ON DELETE CASCADE` des clients aux factures signifie qu'une suppression imprudente de client retire silencieusement les pièces que l'entreprise doit conserver. N'utilisez la suppression en cascade que pour les lignes qui n'ont pas de sens sans leur parent, comme les lignes de commande ; ailleurs, utilisez `RESTRICT` ou le choix par défaut.

Une conception normalisée n'est imposée par rien. Les tables créées à la main ou chargées depuis des tableurs arrivent souvent sans clé primaire ni clés étrangères : rien n'arrête un doublon ou une commande pour un client inexistant. PostgreSQL n'exige pas de clé primaire, même si sa documentation note que la théorie relationnelle en exige une par table. Déclarez les clés et la base refuse la mauvaise ligne avec une erreur nommant la contrainte.

Des totaux en cache dérivent. Une colonne `order_total` sur `orders`, gardée près des lignes qu'elle additionne, est une dénormalisation : chaque chemin de code modifiant une ligne doit aussi la mettre à jour. Un oubli laisse un total faux

sans erreur. Calculez-le dans une requête ou une vue. Si c'est trop lent, conservez la copie, mais entretenez-la à un seul endroit, dans la transaction qui change les lignes.

La dénormalisation précède la mesure. Les jointures sur clés indexées coûtent peu ; copier des colonnes pour les éviter échange un petit coût connu contre les anomalies décrites. Lisez d'abord le plan de requête. Si les lectures sont réellement trop lentes, une vue matérialisée ou une base de rapports distincte peut conserver la forme aplatie, tandis que les tables sources gardent chaque fait une seule fois. La production de rapports se prête à une dénormalisation délibérée, comme le schéma en étoile d'un entrepôt de données.

La décomposition va trop loin. La normalisation demande un emplacement par fait, pas autant de tables que possible. Séparer une table dont les colonnes ne dépendent déjà que de la clé ajoute seulement des jointures et des noms à apprendre. Un schéma exigeant une douzaine de jointures pour une commande n'est pas plus facile à utiliser qu'un schéma plat. Arrêtez-vous quand chaque fait n'a qu'un emplacement.

Pour aller plus loin

- Un modèle relationnel de données pour de grandes banques de données partagées — article de Codd de 1970 dans Communications of the ACM : modèle relationnel, forme normale et normalisation.
- Dépendances multivaluées et nouvelle forme normale — article de Fagin de 1977 définissant dépendance multivaluée et quatrième forme normale, et montrant la décomposition sans perte de toute table vers cette forme.
- Contraintes — clés primaires, étrangères, contraintes de vérification et actions `ON DELETE`, et intérêt d'indexer les colonnes référençantes.
- Vues — tutoriel PostgreSQL sur les vues comme interface stable de tables dont la structure change.

Glossaire

Choisissez selon vos contraintes réelles

Présentez un jeu de données représentatif, les questions auxquelles il doit répondre et le goulot d'étranglement actuel. Nous pouvons convenir de la manière de mesurer la qualité, le débit et la reprise.

Consulter les tarifs Contactez-nous →

Rendez la terminologie concrète

Demandez comment « Que désigne la normalisation de base de données ? » s'applique à votre organisation. Un exemple concret tiré de votre système actuel suffit pour engager la discussion.

Découvrir nos prestations

welcome@engineer.company +45 50 10 61 84 Copenhague, Danemark

Emportez cette page avec vous

Fiche de contact

Créé avec soin et amour par D.

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

Droit d'auteur © 2025 – aujourd'hui · Engineer ApS · Tous droits réservés.

Imprimé depuis https://engineer.company/fr/glossary/normalization/