Engineer

Vue matérialisée

Vue stockant le résultat de sa requête comme une table

Ce que c’est

Une vue matérialisée est une requête dont la base conserve le résultat : sa lecture renvoie des lignes calculées auparavant au lieu de relancer la requête. En anglais britannique, elle s’écrit materialised view. Elle se situe entre deux objets plus anciens. Une vue ordinaire ne stocke que sa requête, exécutée sur les tables sous-jacentes à chaque lecture. Une table créée avec `CREATE TABLE AS` conserve un résultat mais oublie la requête d’origine. Une vue matérialisée garde les deux : lignes servant aux lectures et requête réexécutable pour les remplacer.

PostgreSQL a ajouté les vues matérialisées dans la version 9.3 du 9 septembre 2013 ; ses notes attribuent ce travail à Kevin Grittner. La documentation les décrit comme utilisant le système de règles des vues ordinaires, tout en conservant le résultat sous une forme semblable à une table. Pour l’analyseur de requêtes, c’est une relation comme n’importe quelle table ou vue : un `SELECT` lit donc directement les lignes stockées. La requête conservée n’intervient qu’au remplissage. La vue ne peut pas être mise à jour directement. Pour remplacer son contenu par un résultat de requête actualisé, il faut réexécuter la requête stockée.

Deux commandes font ce travail. `CREATE MATERIALIZED VIEW` définit la vue et exécute immédiatement la requête pour la remplir, sauf avec `WITH NO DATA`. `REFRESH MATERIALIZED VIEW` réexécute la requête stockée et remplace tout le contenu par le nouveau résultat. La version 9.4 du 18 décembre 2014 a ajouté `CONCURRENTLY`, permettant l’actualisation sans bloquer les sessions de lecture. Ces deux commandes sont des extensions PostgreSQL hors de la norme SQL ; d’autres bases proposent le principe avec leurs syntaxes et règles.

Utilisez le concept pour prendre une décision

Utilisez « Qu'est-ce qu'une vue matérialisée ? » pour éclairer une décision réelle. Nous pouvons vous aider à relier ce concept au système que vous construisez ou achetez. Identifier les requêtes et les index manquants dans un système alourdi au fil des ans, puis corriger ce qui compte.

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. 30 jours de données électriques traités en 180 ms — 180 ms pour agréger un mois de relevés de compteurs

30 jours de données électriques traités en 180 ms Contactez-nous →

Pourquoi c'est important

Certaines requêtes coûtent cher et reviennent souvent : tableau de bord sommant un an de commandes par jour, rapport joignant six tables, classement des produits par ventes. Sur les tables sources, chaque demande repaie le parcours, les jointures et l'agrégation complets, alors que la réponse change à peine. Une vue matérialisée paie une fois par actualisation ; chaque lecture parcourt ensuite une petite table calculée d'avance. La documentation PostgreSQL dit que lire les données stockées est « souvent bien plus rapide que d'accéder directement aux tables sous-jacentes ou par une vue ».

Le prix est la fraîcheur. Entre actualisations, la vue présente l'état de la dernière ; dans PostgreSQL, rien ne l'actualise sans demande humaine ou tâche planifiée. Beaucoup d'usages l'acceptent. Les ventes d'hier ne changent pas aujourd'hui, et un rapport mensuel n'a pas besoin des commandes de cette minute. La vue impose une décision explicite : l'âge acceptable de la réponse et la personne chargée de l'empêcher de vieillir davantage.

Elle offre aussi une place aux index : le résultat stocké peut avoir ses propres index, adaptés aux requêtes qui le lisent. La documentation PostgreSQL montre un deuxième usage : des données importées d'un système par un adaptateur de données externes, qui peut ne pas gérer les index du tout, peuvent être copiées dans une vue matérialisée qui les accepte.

Pour une équipe gardant un schéma normalisé, c'est souvent le bon endroit pour une forme aplatie. La normalisation stocke chaque fait une fois et répond par des jointures à la plupart des questions. Une vue matérialisée peut garder le résultat joint et agrégé des rapports, tandis que les tables sources conservent leur exemplaire unique et que chaque écriture vise toujours un seul endroit.

À quoi cela ressemble en pratique

Prenez les tables de commandes d'une boutique : `orders` , une ligne par commande avec la date de passation, et `order_lines` , avec SKU, quantité et prix unitaire de chaque ligne. Le tableau de bord veut les unités et le chiffre d'affaires par produit et par jour. La requête est courte, mais lit chaque ligne sur plusieurs années :

```
CREATE MATERIALIZED VIEW daily_sales AS
SELECT o.ordered_on                  AS day,
       l.sku,
       sum(l.quantity)               AS units,
       sum(l.quantity * l.unit_price) AS revenue
FROM   orders o
JOIN   order_lines l USING (order_id)
WHERE  o.ordered_on < CURRENT_DATE
GROUP  BY o.ordered_on, l.sku;
```

La clause `WHERE` exclut aujourd'hui, dont les chiffres bougent encore : le tableau n'affiche que les jours complets, comme l'exemple de la documentation PostgreSQL pour une journée de ventes inachevée. La commande exécute une fois la requête et stocke le résultat. Vient ensuite un index :

```
CREATE UNIQUE INDEX daily_sales_sku_day ON daily_sales (sku, day);
```

L'index sert deux objectifs : trouver les jours d'un produit sans lire toute la vue et permettre ensuite l'actualisation concurrente. La requête groupe par jour et SKU : chaque paire apparaît une fois, l'index peut donc être unique. Le tableau de bord lit désormais la vue comme une table :

```
SELECT day, units, revenue
FROM   daily_sales
WHERE  sku = 'TEA-250'
ORDER  BY day;
```

Chaque nuit, peu après minuit, une tâche planifiée la met à jour :

```
REFRESH MATERIALIZED VIEW CONCURRENTLY daily_sales;
```

Sans `CONCURRENTLY`, actualiser beaucoup de lignes tend à consommer moins de ressources et finir plus tôt, mais peut bloquer les lectures jusqu'à la fin. Avec cette option, elles continuent, et la documentation note que cela peut même être plus rapide si peu de lignes changent, comme ici où chaque nuit ajoute un jour. L'option a deux conditions : la vue doit déjà contenir des données et posséder au moins un index unique sur des noms de colonnes simples, couvrant toutes les lignes, donc sans expression ni clause `WHERE`. L'index ci-dessus satisfait la seconde ; la création avec données a satisfait la première.

Les autres bases placent la limite ailleurs. Dans BigQuery, les vues matérialisées sont actualisées en arrière-plan : les changements des tables de base sont ajoutés incrémentalement sans demande ; une requête sur la vue intègre les changements récents pour rester à jour ; l'optimisation intelligente peut rediriger vers la vue une requête écrite pour les tables. Cela vaut pour les vues incrémentales, limitées à un sous-ensemble SQL. Une vue avec `allow_non_incremental_definition = true` accepte la plupart des requêtes, mais chaque actualisation réexécute la requête entière et les résultats ne sont pas toujours à jour.

SQL Server obtient un résultat similaire par les vues indexées. Ajouter à une vue un index cluster imposant l'unicité stocke son résultat selon la même organisation de stockage qu'une table indexée ; la vue doit être créée avec `WITH SCHEMABINDING` et une définition déterministe. L'optimiseur peut l'utiliser même si la requête ne la nomme pas. SQL Server l'actualise avec chaque insertion, modification et suppression dans les tables dont elle dépend : aucune actualisation à planifier. Le coût passe aux écritures, dont les performances peuvent fortement se dégrader, avertit Microsoft.

Le même compromis apparaît en production. Dans un poste antérieur, la fondatrice a construit des pipelines ETL et ELT PL/pgSQL utilisant des tables partitionnées et des vues matérialisées pour accélérer la lecture de grands jeux de données, avec contraintes de clés primaires et étrangères pour l'intégrité référentielle et index uniques et composites pour les jointures.

Où les problèmes surviennent

Des chiffres anciens semblent actuels. Une tâche arrêtée depuis une semaine laisse la vue répondre à chaque requête, rapidement mais incorrectement, sans que le résultat indique son âge. Faites consigner l'heure de chaque actualisation réussie dans un endroit affichable par le tableau de bord, et déclenchez une alerte si la dernière actualisation est plus ancienne que ce que les lecteurs ont accepté.

L'actualisation bloque les lecteurs. Un simple `REFRESH MATERIALIZED VIEW` sur une grande vue peut bloquer toutes les sessions de lecture jusqu'à la fin. Ajoutez un index unique et utilisez `CONCURRENTLY`, ou planifiez sans cette option lorsque personne ne lit. Si la plupart des lignes changent à chaque exécution, l'actualisation simple peut rester moins coûteuse.

`CONCURRENTLY` **est refusé.** Il faut un index unique sur des colonnes simples couvrant toutes les lignes. Un index unique sur `lower(email)` ou limité par `WHERE` ne convient pas. La vue doit aussi déjà contenir des données : l'option échoue sur une vue créée `WITH NO DATA`, et les deux options sont incompatibles. Ajoutez un index admissible et remplissez la vue une fois par une actualisation simple.

La vue est impossible à parcourir. `WITH NO DATA` crée la vue sans exécuter la requête et la marque non parcourable : toute requête échoue jusqu'à la première actualisation. Cela peut convenir à une migration ne devant pas attendre une longue requête, mais il faut alors planifier ce premier calcul, pas l'oublier.

La modification de la définition n'est pas appliquée. Dans une migration, `CREATE MATERIALIZED VIEW IF NOT EXISTS` ne fait rien sauf émettre une notice si le nom existe. La documentation ne garantit aucune ressemblance entre la vue existante et la nouvelle définition. Changer ainsi la requête laisse l'ancienne. Supprimez la vue avec `DROP MATERIALIZED VIEW` et recréez-la dans la même migration.

Les actualisations se chevauchent. Une vue matérialisée ne peut en exécuter qu'une à la fois, même avec `CONCURRENTLY` : planifier les actualisations à des intervalles plus courts que leur durée ne fait qu'accroître le retard. Mesurez la durée et planifiez en conséquence.

L'ordre des lignes est supposé acquis. Un `ORDER BY` dans la définition ordonne le premier contenu, mais une actualisation ne promet pas de le conserver. Placez `ORDER BY` dans les requêtes de lecture.

Les fonctions perdent leurs tables. Pendant la requête de définition, à la création ou à l'actualisation, PostgreSQL fixe temporairement `search_path` à `pg_catalog, pg_temp` et restreint les actions autorisées : une fonction créant des tables temporaires échoue. Une fonction trouvant ses tables par nom non qualifié peut réussir tous les tests manuels puis échouer seulement pendant l'actualisation. Qualifiez les noms de tables par leur schéma dans toute fonction appelée par la vue.

Le mauvais rôle exécute la tâche. Actualiser demande le privilège `MAINTAIN` sur la vue matérialisée. Une tâche exécutée sous un rôle de rapports autorisé à lire la vue, mais pas à la maintenir, échoue chaque nuit. Donnez ce privilège au rôle exécutant l'actualisation.

Une copie stockée là où un index suffirait. Si une requête est lente faute d'index, la vue matérialisée masque le problème et ajoute un problème de fraîcheur. Lisez d'abord le plan. Lorsqu'il s'agit seulement de donner un nom à une requête, une vue ordinaire suffit et reste toujours à jour ; réservez les vues matérialisées aux résultats coûteux à calculer dont on peut lire sans risque une version légèrement ancienne.

Pour aller plus loin

- Vues matérialisées — leur rapport aux vues et tables dans PostgreSQL, avec résumé des ventes actualisé la nuit et exemple d'adaptateur de données externes.
- CREATE MATERIALIZED VIEW — référence de la commande, dont `WITH NO DATA`, `IF NOT EXISTS` et le `search_path` restreint pendant la requête.
- REFRESH MATERIALIZED VIEW — actualisation, conditions de `CONCURRENTLY` et privilège `MAINTAIN` requis.
- Notes de PostgreSQL 9.3 — la version de septembre 2013 ajoutant les vues matérialisées.
- Introduction aux vues matérialisées — vues BigQuery incrémentales ou non, actualisation automatique et optimisation intelligente.
- Créer des vues indexées — vues SQL Server, exigences et coût des écritures.

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 « Qu'est-ce qu'une vue matérialisée ? » 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/materialized-view/