

pour
l'optimisation

Les VUES de données

Plan du document

01 *Principe*

02 *Avantage*

03 *Coût*

04 *Accès*

PRINCIPE

1

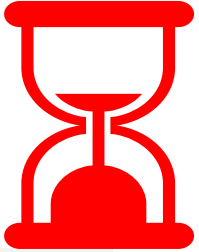


Aperçu
précalculé

C'est un aperçu

Une présélection sur toutes les données disponibles

Principe de la VUE



Une VUE c'est comme le résultat d'un SELECT particulier auquel on aurait donné un NOM l'identifiant et qui serait préparé d'avance pour les prochaines fois.

La VUE peut restreindre les données en utilisant :

- des colonnes nommées pour ne choisir que quelques colonnes
- des conditions pour restreindre les rangées incluses

La VUE peut enrichir les données et comprendre :

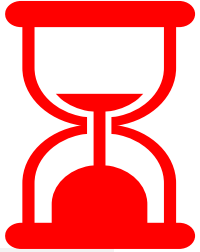
- des agrégations
- une jointure
- une union

La vue comporte son propre INDEX et propose une vue partielle des données, que ce soit les colonnes et les rangées.

C'est donc une requête précalculée, toujours maintenue par le système, pour optimiser cette requête particulière.

Elle permet de précalculer des choses (nombre de moutons par troupeau) sans ajouter de redondance réelle. Cette redondance est simulée. Cependant il y a un coût d'insertion et de modification aux données car la VUE doit aussi être ajustée à chaque changement.

Syntaxe et exemples



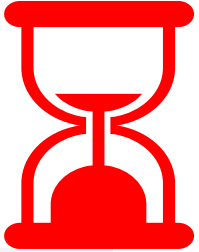
CREATE VIEW AS …. (ici on place le code du SELECT normal)

Exemples de vues présentant du filtrage de rangées

```
CREATE VIEW film_fantaisie AS
SELECT film.* FROM film
WHERE film.categorie = 'fantaisie';
```

```
CREATE VIEW membres_non_admin AS
SELECT membre.* FROM membre
WHERE membre.type != 'admin';
```

Syntaxe et exemples



CREATE VIEW AS …. (ici on place le code du SELECT normal)

Exemples de vues présentant des aggrégations et des jointures

```
CREATE VIEW troupeau_avec_total
SELECT troupeau.*, COUNT(mouton.*) as moutons FROM troupeau
LEFT JOIN mouton ON mouton.id_troupeau = troupeau.id
GROUP BY troupeau.id
```

```
CREATE VIEW statistiques_bateau
SELECT bateau.nom as nom_bateau, SUM(voyage.*) as total_voyage,
SUM(reparation.*) as total_reparation FROM bateau
LEFT JOIN voyage ON voyage.id_bateau = bateau.id
LEFT JOIN reparation ON reparation.id_bateau = bateau.id
```

AVANTAGE

2



Optimisation
avec les vues

C'est optimal



Les vues sont des sélections calculées d'avance

Un outil d'optimisation



Les VUES de données présentent le résultat d'un SELECT qui peut être beaucoup plus coûteux à exécuter que le simple SELECT de la vue.

Par exemple : on a 4 tables reliées entre elles, par exemple les étudiants qui sont inscrits à des cours, avec une table de relation appelée inscription. À chaque inscription est associée une série d'entrée dans une autre table qui sont les documents remis dans le cours par l'étudiant.

Des statistiques sur ces données pour un cours donné impliquent au moins 2 jointures, donc des opérations de multiplication (combinaison de données de NxMxD.

C'est beaucoup plus long à exécuter que le simple SELECT sur la vue de taille N ou M.

Quand les utiliser



Les VUES de données devraient être utilisées pour optimiser les requêtes exécutées fréquemment contre des données mises à jour occasionnellement.

Par exemple : on a un panneau de statistiques comportant plusieurs jointures qui est sollicité souvent par les administrateurs ou par un script Ajax, ces requêtes ont été identifiées comme étant celles qui écrasent la base de données.

Il est possible d'optimiser ce fonctionnement en préparant une VIEW et en spécifiant qu'elle ne sera mise à jour qu'à toutes les 15 minutes. Ensuite les requêtes qui s'opèrent sur la vue n'ont que le coût d'un simple SELECT.

Cette optimisation peut se combiner avec d'autres stratégies comme le Caching sur serveur, intermédiaire ou client.

Insérer dans une VUE

Les VUES peuvent aussi servir d'interface d'accès en ÉCRITURE (c'est moins utilisé).

Il existe 2 options que l'on place à la fin de la déclaration pour configurer cet INSERT.

- WITH LOCAL CHECK OPTION
  Va valider l'INSERT avec seulement les conditions de la VUE elle-même.

- WITH RECURSIVE CHECK OPTION
  Va valider l'INSERT avec les conditions de toutes les tables impliquées.

Ceci fonctionne si la vue est considérée par le système comme UPDATABLE
Il existe 4 conditions :

- La vue n'a qu'une table (ou vue) listée dans le FROM de la requête
- La vue n'a pas de WITH, DISTINCT, GROUP BY, HAVING, LIMIT, ou OFFSET.
- La vue n'a pas d'opération de SET comme UNION, INTERSECT ou EXCEPT
- Il n'y a pas de fonctions d'agrégation

COÛT

3



Compromis

sélection versus insertion

Sélection versus insertion



La VUE favorise la **SÉLECTION**

En ayant préparé des sélections spéciales en avance, le système se trouve à stocker le résultat de cette requête dans un INDEX qui est un arbre binaire des clés d'accès. Retourner lire la VUE prendra beaucoup moins de temps que refaire la requête, surtout si celle-ci comporte des jointures.

La VUE augmente le coût de l'**INSERTION**

À chaque AJOUT de données, il faut réexécuter la requête de VUE pour mettre à jour les données.

Sinon bris de normalisation



La seule alternative aux vues pour stocker des informations précalculées sont les optimisations par bris de normalisation. Grâce aux vues, ceci n'est presque jamais nécessaire.

Par exemple, on pourrait vouloir stocker le nombre de moutons par troupeaux dans la table troupeaux et ajouter un champs qui viole la normalisation 'totalMouton' ou 'moutons' dans lequel on update le nombre de mouton à chaque INSERT ou DELETE de la table mouton. C'est une solution acceptable pour une situation dans laquelle le nombre de mouton est calculé souvent et que l'on souhaite optimiser cette partie de l'application. Même si elle est acceptable en de rares circonstances, on peut encore l'éviter grâce aux vues.

On peut simplement laisser la table originale de mouton inchangée, et faire une vue presque identique à la table mouton qui elle comporte le champs supplémentaire par exemple :

```
CREATE VIEW troupeau_avec_total
SELECT troupeau.*, COUNT(mouton.*) as moutons FROM mouton
LEFT JOIN mouton ON mouton.id_troupeau = troupeau.id
```

Pour forcer la mise à jour



Dans PostGreSQL, pour forcer le système à calculer les VUES en avance, on peut utiliser la syntaxe des vues MATERIALIZED.

https://www.postgresql.org/docs/9.5/sql-creatematerializedview.html

Attention, c'est propriétaire à PostGreSQL

```
CREATE MATERIALIZED VIEW troupeau_avec_total
SELECT troupeau.*, COUNT(mouton.*) as moutons FROM mouton
LEFT JOIN mouton ON mouton.id_troupeau = troupeau.id
```

=> donc si on porte la table vers une autre base de données, il faut effacer les 'MATERIALIZED'

On peut ensuite forcer le recalcul de cette vue avec la commande REFRESH

https://www.postgresql.org/docs/9.3/sql-refreshmaterializedview.html

```
REFRESH MATERIALIZED VIEW troupeau_avec_total;
```

Cette commande peut s'automatiser avec un script linux sur CRONTAB ou être liée à une procédure stockée, lors des INSERT ou DELETE de la table mouton.

ACCÈS

4



Accès
filtré

Idée

On peut exploiter la limitation d'accès aux VUES pour forger une interface, comme avec les services de données.

Pour filtrer l'accès



Si on n'a pas de services de données et qu'on souhaite présenter seulement une fraction de notre base de données ou même de notre table à un service pour des raisons de sécurité, on peut aussi utiliser une vue.

Par exemple, une APP de statistiques n'a pas besoin d'accéder aux détails des champs alors on peut inclure dans cette APP un accès de LECTURE seulement sur une VIEW qui précalcule des STATISTIQUES.

On peut faire des GRANT sur les VIEW tout comme si c'était des TABLES.

https://www.postgresql.org/docs/9.0/sql-grant.html

GRANT SELECT ON mavue TO PUBLIC;

GRANT SELECT, UPDATE, INSERT ON mavue TO admin;

GRANT SELECT (col1), UPDATE (col1) ON mavue TO joe;

Syntaxe et exemples



GRANT (SELECT|INSERT|UPDATE|DELETE) ON …. TO ….

Exemples de vues présentant du filtrage de rangées

```
CREATE VIEW film_fantaisie
SELECT film.* FROM film
WHERE film.categorie = 'fantaisie';
```

```
GRANT SELECT ON film to moderateur_fantaisie;
```

```
GRANT SELECT, INSERT, UPDATE, DELETE ON film_fantaisie to moderateur_fantaisie;
```

Syntaxe et exemples



GRANT (SELECT|INSERT|UPDATE|DELETE) ON …. TO ….

Exemples de vues présentant du filtrage de rangées

```
CREATE VIEW membres_non_admin
SELECT membre.* FROM membre
WHERE membre.type != 'admin';
```

```
GRANT SELECT, INSERT, UPDATE, DELETE ON membre to super_admin;
```

```
GRANT SELECT, INSERT, UPDATE, DELETE ON membre_non_admin to coordonnateur_membres;
```

Syntaxe et exemples



GRANT (SELECT|INSERT|UPDATE|DELETE) ON …. TO ….

Exemples de vues présentant de l'agrégation

```
CREATE VIEW statistiques_bateau
SELECT bateau.nom as nom_bateau, SUM(voyage.*) as total_voyage,
SUM(reparation.*) as total_reparation FROM bateau
LEFT JOIN voyage ON voyage.id_bateau = bateau.id
LEFT JOIN reparation ON reparation.id_bateau = bateau.id
```

Aucun GRANT à l'APP dans la table bateau, voyage ou reparation.

```
GRANT SELECT ON statistiques_bateau TO app_bateau;
```

```
GRANT SELECT, INSERT, UPDATE, DELETE ON bateau TO panneau_admin;
GRANT SELECT, INSERT, UPDATE, DELETE ON voyage TO panneau_admin;
```

À vos

VUES

