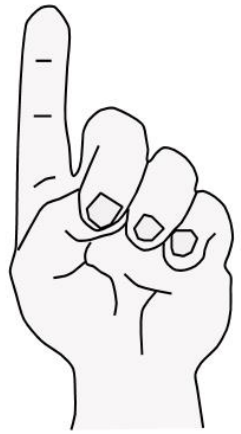
Optimisation des requêtes

Les index et les plan d’exécution



Sommaire

- Généralité
- Optimisation logique
  - Les sélections et projection
  - Les jointures
  - Les conditions dans les jointures
  - Le WHERE
- Optimisation physique
- Plan d’exécution
- Pour conclure

Généralités

Problématique :

On souhaite développer une application web grand publique qui fait appelle à des requêtes SQL.

Pour peut que la base de données soit un temps soit peut **volumineuse** avec des **requêtes** allant au-delà du simple SELECT (jointure, requêtes imbriqués, clause where, …) , on peut atteindre des temps de réponse de quelques minutes.

Comment optimiser le temps d'exécution des requêtes SQL ?

Généralité

Principe de base :

Réduire le plus rapidement possible le nombre d'enregistrements sur le quel on travaille.

Plus la restriction la plus forte arrive **tôt**, plus la requêtes est **performante**.

Exemple :

La requêtes pour rechercher les coureurs de plus de 70 ans et de sexe masculin est :

```
select CO_NOM, round(datediff(now(), CO_NAISSANCE)/365, 0) as AGE, CO_SEXE
from COUREUR
where CO_SEXE = 1
and datediff(now(), CO_NAISSANCE) > 70;
```

Généralité

Autre syntaxe pour cette requête :

```
select CO_NOM, round(datediff(now(), CO_NAISSANCE)/365, 0) as AGE, CO_SEXE
from COUREUR
where datediff(now(), CO_NAISSANCE) > 70
and CO_SEXE = 1;
```

Le résultat de cette requête est le même que le précèdent, mais le temps d'exécution est moindre car la recherche des homme se fait uniquement sur les coureurs de plus de 70 ans.

(On par du principe qu'il y a peut de sportifs de plus de 70 ans et que ½ est un homme).

Optimisation logique

D’une manière générale, il convient de porter une attention particulière à chacune des parties d’une requête, à savoir :

1) Les sélections est projections :

Il est plus performant d’identifier et de nommer dans le select les champs dont on à besoin plutôt que de faire un SELECT *.

-> réduction des données en mémoire.

Optimisation logique

2) Les jointures :

Elles doivent être organiser pour réduire au plus vite le volume des données traitées.

Exemple : Pour afficher tout les coureurs ayant participé au marathon de Bordeaux

```
select CO_NOM, CO_PRENOM
from COUREUR
inner join INSCRIPTION on CO_ID = IN_COUREUR_FK
inner join EPREUVE on IN_EPREUVE_FK = EP_ID
inner join MANIFESTATION on EP_MANIF_FR = MA_ID
where MA_NOM = 'Marathon de Bordeaux';
```

Si on analyse la démarche de cette requête, on commence par travailler sur tout les coureurs pour finir par poser la restriction sur la manifestation « MARATHON BORDEAUX ».

Optimisation logique

Pour optimiser cette requête on va poser immédiatement la restriction sur la manifestation.

```
select CO_NOM, CO_PRENOM
from MANIFESTATION
inner join EPREUVE on EP_ID = IN_EPREUVE_FK
inner join INSCRIPTION on IN_COUREUR_FK = EP_ID
inner join COUREUR on CO_ID = In_COUREUR_FK
where MA_NOM = 'Marathon de Bordeaux';
```

Cette requête affiche le même résultat mais en recherchant, dés le départ, les participants de la manifestation.

Optimisation logique

On peut encore optimiser cette requête en posant la condition au moment de la jointure et non dans le WHERE.

Cela va permettre de réduire rapidement le nombre d'observation traités.

```
select CO_NOM, CO_PRENOM
from MANIFESTATION
inner join EPREUVE on EP_ID = IN_EPREUVE_FK
and MA_NOM = 'Marathon de Bordeaux'
inner join INSCRIPTION on IN_COUREUR_FK = EP_ID
inner join COUREUR on CO_ID = In_COUREUR_FK;
```

Optimisation logique

3) Le WHERE

Comme vu plus haut, si l'ordre des condition du WHERE ne change le résultat il à un effet sur les performances. Il est préférable de :

- Poser les conditions qui filtre le plus d'enregistrement en premier.
- S'assurer que les LIKE ne sont pas remplacé par des égales.
- S'assurer que certaines conditions ne pas factorisables.

Optimisation logique

Pour chercher tout les coureur de sexe masculin ayant plus de 80 ans ou ayant déjà adhéré à un club bordelais la syntaxe SQL est :

```
Select distinct CO_NOM, CO_PRENOM,
    round(datediff(now(), CO_NAISSANCE)/365, 0) as AGE,
    CL_VILLE
    from COUREUR
    inner join ADHESION on CO_ID = AD_COUREUR_FK
    inner join CLUB on AD_CLUB_FK = CL_ID
    where (CO_SEXE = 1 and round(datediff(now(), CO_NAISSANCE)/365, 0) > 80)
        or (CO_SEXE = 1 and CL_VILLE = 'Bordeaux');
```

Optimisation logique

En factorisant les condition la requête devient :

```
Select distinct CO_NOM, CO_PRENOM,
    round(datediff(now(), CO_NAISSANCE)/365, 0) as AGE,
    CL_VILLE
    from COUREUR
    inner join ADHESION on CO_ID = AD_COUREUR_FK
    inner join CLUB on AD_CLUB_FK = CL_ID
    where CO_SEXE = 1 and
        (round(datediff(now(), CO_NAISSANCE)/365, 0) > 80
            or CL_VILLE = 'Bordeaux');
```

Optimisation physique

L'optimisation physique des requêtes passe par l'étude des conditions de sélection les plus souvent rencontrées, afin de mettre en place des index sur les champs concernés pour accélérer les recherches sur ces derniers.

Il est conseillé d'ajouter des index sur les champs faisant l'objet de restrictions :

- WHERE
- GROUP BY
- ORDER BY

Indication : les clés primaires et les clés étrangères sont des index par défaut.

Optimisation physique

La syntaxe :

```
CREATE INDEX index_nom ON table1 (colonne1);
```

Pour un index sur plusieurs colonnes :

```
CREATE INDEX index_nom ON table1 (colonne1, colonne2);
```

Un index unique permet de spécifier qu'une ou plusieurs colonnes doivent contenir des valeurs uniques à chaque enregistrement.

```
CREATE UNIQUE INDEX index_nom ON table1 (colonne1);
```

Optimisation physique

L’index est, tout comme dans le secteur **bibliographique**, un moyen plus rapide qu’une lecture séquentielle pour accéder à une information.

Placé sur une ou plusieurs colonnes, un index permet au moteur d’une base de données d’accéder plus rapidement aux données recherchées car ce dernier **débute toujours par les index.**

Optimisation physique

Les index on en revanche deux inconvénients :

- Ils prennent de l'espace de stockage supplémentaire dans la base de données
- Ils ralentissent l'exécution des requêtes actions INSERT, UPDATE, ou, DELETE car en plus de la modification des données il faut également mettre à jour l'index.

Les index seront une aide précieuse pour l'optimisation du temps dans une base ayant beaucoup d'enregistrements.

Plan d’exécution

On peut apprécier les gain de temps apporté par l’optimisation d’une requête grâce au **plan d’exécution**.

Un plan d’exécution est un schéma représentant, étape par étape, les accès aux données pour chacune des instruction d’une requête.

Pour ce faire on utilise la commande EXPLAIN.

```
explain select title, category.name
    from film
    inner join film_category
    on film.film_id = film_category.film_id
    inner join category
    on film_category.category_id = category.category_id
    where category.name like 'Action';
```

Plan d’exécution

Représentation des trois étapes par lesquels passe la requête pour s’exécuter.

	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
▶	1	SIMPLE	category	NULL	ALL	PRIMARY	NULL	NULL	NULL	16	11.11	Using where
	1	SIMPLE	film_category	NULL	ref	PRIMARY,fk_film_category_category	fk_film_category_category	1	sakila.category.category_id	62	100.00	Using index
	1	SIMPLE	film	NULL	eq_ref	PRIMARY	PRIMARY	2	sakila.film_category.film_id	1	100.00	NULL

Plan d’exécution

- **id :** identifiant de SELECT
- **select_type :** type de cause SELECT (exemple : SIMPLE, PRIMARY, UNION, DEPENDENT UNION, SUBQUERY, DEPENDENT SUBSELECT ou DERIVED)
- **table :** table à laquelle la ligne fait référence
- **type :** le type de jointure utilisé (exemple : system, const, eq_ref, ref, ref_or_null, index_merge, unique_subquery, index_subquery, range, index ou ALL)
- **possible_keys :** liste des index que MySQL pourrait utiliser pour accélérer l’exécution de la requête. Dans notre exemple, aucun index n’est disponible pour accélérer l’exécution de la requête SQL
- **key :** cette colonne présente les index que MySQL a décidé d’utiliser pour l’exécution de la requête
- **key_len :** indique la taille de la clé qui sera utilisée. S’il n’y a pas de clé, cette colonne renvois NULL
- **ref :** indique quel colonne (ou constante) sont utilisés avec les lignes de la table
- **rows :** estimation du nombre de ligne que MySQL va devoir analyser examiner pour exécuter la requête
- **Extra :** information additionnelle sur la façon dont MySQL va résoudre la requête. Si cette colonne retourne des résultats, c’est qu’il y a potentiellement des index à utiliser pour optimiser les performances de la requête SQL. Le message “using temporary” permet de savoir que MySQL va devoir créer une table temporaire pour exécuter la requête. Le message “using filesort” indique quant à lui que MySQL va devoir faire un autre passage pour retourner les lignes dans le bon ordre

Pour conclure

- L’optimisation des requête n’est pas primordial dans une phase d’apprentissage, mais elle devient obligatoire dés que l’on passe en exploitation.
- Elle est l’un des gage du maintient optimum de la base de données en conditions opérationnelles.