6

Techniques d'optimisation standard des requêtes

L'optimisation du SQL est un point très délicat car elle nécessite de pouvoir modifier l'applicatif en veillant à ne pas introduire de bogues.

6.1 Réécriture des requêtes

L'utilisation d'objets d'optimisation est un élément qui, de manière générale, améliore les performances très significativement. Cependant, la façon d'écrire votre requête peut, elle aussi, dans certains cas, avoir un impact très significatif.

L'étape de transformation de requête (voir ci-après) est de plus en plus performante sur Oracle et SQL Server. Elle donne de très bons résultats, ce qui rend inutiles certaines recommandations dans le cas général. Cependant, sur MySQL ou sur des requêtes compliquées, où on peut considérer que le SGBDR n'arriverait pas à faire certaines transformations tout seul, l'utilisation des écritures les plus performantes permettra d'améliorer les résultats.

Nous allons faire quelques comparaisons d'écritures qui intégreront parfois un hint (voir Chapitre 7, section 7.1, "Utilisation des hints sous Oracle") afin d'empêcher certaines transformations.

De façon générale, il faut privilégier les jointures. Le principe du modèle relationnel étant que, lors de l'interrogation, des jointures seront faites, on peut supposer que les éditeurs de SGBDR ont concentré leurs efforts là-dessus. Cependant, sous MySQL avec le moteur MyISAM, les sous-requêtes sont souvent plus performantes.

6.1.1 Transformation de requêtes

L'étape de transformation de requêtes (*Query Transformation*) est partie intégrante du processus de traitement des requêtes. C'est une des fonctions de l'optimiseur CBO.

Cette étape consiste à transformer la requête soumise en une requête équivalente afin de la rendre plus performante. Cela va, au-delà, de choisir le meilleur chemin d'exécution. La transformation de requêtes peut, par exemple, décider de transformer une sous-requête en jointure. Vous pouvez influer sur cette étape au moyen des hints, mais c'est rarement nécessaire.

Les principales transformations effectuées par le CBO sont :

- **Subquery unesting.** Cette transformation consiste à transformer des sous-requêtes en jointures.
- **Suppression d'éléments inutiles** (certaines jointures, des colonnes sélectionnées dans les sous-requêtes).
- **Predicate push.** Cette transformation consiste à dupliquer certains prédicats dans les sous-vues et les sous-requêtes.
- **View merging.** Intègre l'exécution des vues à la requête principale.
- **Query Rewriting.** Utilisation des vues matérialisées (voir Chapitre 5, section 5.3.5, "Les vues matérialisées").
- **Or Expansion.** Transforme des conditions OR en plusieurs requêtes fusionnées par une sorte d'"Union ALL".

Il n'y a rien de particulier à faire pour bénéficier du Query Rewriting. Connaître l'existence de ce mécanisme vous aidera à comprendre pourquoi il arrive parfois qu'un plan d'exécution ne ressemble vraiment pas à votre requête. Cette fonction est plus évoluée sous Oracle et SQL Server que sous MySQL.

INFO

Au cours de ce chapitre, nous présentons des variations de notation qui peuvent avoir un effet sur les performances. L'optimiseur étant assez performant, il transforme automatiquement les requêtes soumises en la version la plus performante. Nous recourrons à des hints pour empêcher ces transformations afin de comparer les notations. L'utilisation de hint n'est nullement une recommandation, nous le mentionnons à titre indicatif de façon que, si vous exécutez ces requêtes, vous puissiez reproduire les mêmes résultats que ceux présentés ici.

Comme cela sera expliqué au Chapitre 7, section 7.1, "Utilisation des hints sous Oracle", il faut les utiliser en dernier recours et seulement en toute connaissance de cause.

6.1.2 IN versus jointure

Selon que l'écriture d'une requête se fasse avec une sous-requête et l'opérateur `IN` ou avec une jointure, l'impact sera différent. Pour forcer le parcours des tables, et non seulement celui des index, nous utilisons dans la requête des champs non indexés (`Quantité et Editeur`).

Listing 6.1 : Requête utilisant une jointure

```
select sum(CL.quantite) from cmd_lignes CL,livres L
where CL.nolivre = L.nolivre and L.editeur='Pearson'
```

Listing 6.2 : Requête utilisant une sous-requête

```
select sum(quantite) from cmd_lignes
where nolivre in (select nolivre from livres where editeur='Pearson')
```

Nous allons tester ces requêtes avec et sans index sur la colonne Nolivre de la table CMD_LIGNES.

```
create index IS_cmd_lignes on cmd_lignes(nolivre);
```

	Sans index		*Avec index*	
MyISAM	***Jointure***	***Sous-requête***	***Jointure***	***Sous-requête***
Temps	12,89 s	13,53 s	0,01 s	13,53 s
Key_read_requests	5 242 448	5 243 442	158	5 243 442

InnoDB	***Jointure***	***Sous-requête***
Temps	3,48 s	13,51 s
Reads	3 700	2 606 480

InnoDB créant automatiquement un index dans la table fille de la *Foreign Key*, il y a forcement un index sur la colonne Nolivre de la table CMD_LIGNES. La création de l'index est donc sans effets.

	Sans index		*Avec index*	
Oracle	***Jointure***	***Sous-requête***	***Jointure***	***Sous-requête***
Temps	6,15 s	8,75 s	0,01 s	0,60 s
Consistent Gets	8 336	4 891 735	807	12 175

Pour éviter la transformation de la version utilisant une sous-requête en version jointure par le mécanisme de transformation de requête, nous devons utiliser le hint `/*+no_unnest*/` et pour forcer l'usage de l'index, le hint `/*+ index (CL IS_cmd_lignes)*/`. Si nous ne mettons aucun hint, l'optimiseur choisit systématiquement de faire une jointure sans utiliser l'index.

	Sans index	***Avec index***
MS SQL Server	***Jointure***	***Sous-requête***
Temps	0,625 s	0,046 s
Estimated Cost	35,27	60,44

Afin de comparer les résultats entre les bases, nous désactivons le parallélisme sous SQL Server au moyen du code `option (MAXDOP 1).`

Nous n'avons pas trouvé de combinaisons de hints permettant de forcer l'utilisation d'une sous-requête tant qu'il n'y a pas d'index. Une fois l'index placé, nous avons dû contraindre son utilisation avec le hint `WITH (INDEX(IS_CMD_LIGNES))`. En revanche, nous n'avons pas réussi à forcer une jointure pertinente utilisant cet index.

Conclusion : Oracle et SQL Server, grâce à leur capacité de transformation de requête évoluée, exécutent les requêtes de la même façon quelle que soit la notation employée, si aucun hint n'est utilisé.

Lorsque la transformation n'est pas effectuée, on constate que la solution à base de jointure est généralement plus performante.

Au cas où le SGBDR n'arriverait pas à faire la transformation, il est préférable de prendre l'habitude d'écrire des jointures plutôt que des sous-requêtes.

6.1.3 Sous-requêtes versus anti-jointures

Une anti-jointure est une jointure ouverte pour laquelle vous ne sélectionnez que les valeurs non jointes.

Les requêtes ci-dessous, permettent de sélectionner les livres qui n'ont jamais été commandés.

Listing 6.3 : Version Exists

```
select * from livres L
where not exists (select 1 from cmd_lignes CL where CL.NOLIVRE=L.NOLIVRE)
```

Listing 6.4 : Version anti-jointure

```
select L.* from livres L
left outer join cmd_lignes CL on CL.NOLIVRE=L.NOLIVRE
where cl.nolivre is null
```

Nous allons tester ces requêtes avec et sans index sur la colonne Nolivre de la table CMD_LIGNES.

```
create index IS_cmd_lignes on cmd_lignes(nolivre);
```

	Sans index		*Avec index*	
MyISAM	***Anti-jointure***	***Sous-requête***	***Anti-jointure***	***Sous-requête***
Temps	10,11 s	10,11 s	0,02 s	0,02 s
Key_read_requests	29 850 364	29 850 364	2 974	2 974

L'anti-jointure est transformée en sous-requête.

InnoDB	***Anti-jointure***	***Sous-requête***
Temps	24,31 s	25,34 s
Reads	2 976	5 948

InnoDB créant automatiquement un index dans la table fille de la *Foreign Key*, il y a forcément un index sur la colonne Nolivre de la table CMD_LIGNES. La création de l'index est donc sans effets.

	Sans index		*Avec index*	
Oracle	***Anti-jointure***	***Sous-requête***	***Anti-jointure***	***Sous-requête***
Temps	0,340 s	2,282 s	0,375 s	0,031 s
Consistent Gets	6 976	113 419	5 522	8 909

Nous utilisons le hint `/*+NO_QUERY_TRANSFORMATION*/` pour empêcher la transformation de la sous-requête en anti-jointure.

	Sans index	*Avec index*
MS SQL Server	***Anti-jointure***	***Anti-jointure***
Temps	2,437 s	0,079 s
Estimated Cost	47,49	0,54

Quelle que soit la notation utilisée, la requête est transformée en anti-jointure.

Les résultats sont mitigés sous Oracle. Malheureusement, l'étape de transformation de requête n'opte pas toujours pour la meilleure option. L'optimiseur choisit systématiquement l'anti-jointure qui est, en effet, généralement la meilleure option. Ici, la table LIVRE est bien plus petite que la table CMD_LIGNES, dans ce cas, si un index est présent sur la table fille de la *Nested Loop*, la solution utilisant une sous-requête est la meilleure. Il faut donc utiliser un hint pour forcer ce choix. Sur les autres SGBDR, l'étape de transformation ne laisse pas le choix, les deux écritures se valent.

6.1.4 *Exists* versus *Count*

Les requêtes effectuant un test d'existence sont équivalentes à un comptage égal à 0.

Listing 6.5 : Version *Exists*

```
select * from livres L
where not exists (select 1 from cmd_lignes CL where CL.NOLIVRE=L.NOLIVRE)
```

Listing 6.6 : Version *0=count(*)*

```
select * from livres L
where 0 = (select count(*) from cmd_lignes CL where CL.NOLIVRE=L.NOLIVRE)
```

Nous allons tester ces requêtes avec un index sur la colonne Nolivre de la table CMD_LIGNES.

```
create index IS_cmd_lignes on cmd_lignes(nolivre);
```

MyISAM	***Exists***	***Count***
Temps	0,020 s	1,110 s
Key_read_requests	11 915	166 782

InnoDB	***Exists***	***Count***
Temps	22,940 s	29,590 s
Reads	8 900	2 606 477

Oracle	***Exists***	***Count***
Temps	0,031 s	0,031 s
Consistent Gets	8 909	8 909

MS SQL Server	*Exists*	*Count*
Temps	0,062 s	0,062 s

Oracle et SQL Server considèrent ces deux écritures équivalentes, alors que MySQL les voit bien différentes. On suppose que dans la version utilisant `COUNT(*)`, il dénombre toutes les occurrences pour tester l'égalité à 0, alors que la version utilisant `EXISTS` s'arrête à la première occurrence trouvée.

6.1.5 *Exists versus IN*

Une sous-requête utilisant l'opérateur `IN` peut facilement être convertie en sous-requête utilisant l'opérateur `EXISTS`.

Nous allons tester ces requêtes avec un index sur la colonne Nolivre de la table CMD_LIGNES.

```
create index IS_cmd_lignes on cmd_lignes(nolivre);
```

Listing 6.7 : Version sous-requête *IN*

```
select sum(quantite) from cmd_lignes
where nolivre in (select nolivre from livres where editeur='Pearson')
```

Listing 6.8 : Version sous-requête *Exists*

```
select sum(quantite) from cmd_lignes cl
where exists (select nolivre from livres L
              where editeur='Pearson' and CL.nolivre=L.nolivre )
```

MyISAM	*In*	*Exists*
Temps	14,010 s	15,340 s
Key_read_requests	5 243 442	5 242 448

InnoDB	*In*	*Exists*
Temps	7,420 s	8,200 s
Reads	2 606 480	5 212 684

Oracle	*In*	*Exists*	*Forçage index*
Temps	8,594 s	8,594 s	4,234 s
Consistent Gets	4 891 735	4 891 735	2 462 067

L'optimiseur Oracle décide de ne pas utiliser l'index. Si nous forçons son usage à l'aide du hint `/*+ index(cl is_cmd_lignes)*/`, nous notons une amélioration des performances.

Nous utilisons le hint `/*+NO_QUERY_TRANSFORMATION*/` pour empêcher la transformation des sous-requêtes en anti-jointures qui ont un temps de réponse de l'ordre 0,34 seconde.

MS SQL Server	*In ou Exists*	*Forçage index*
Temps	0,625 s	0,046 s

Comme sous Oracle, nous devons forcer l'utilisation de l'index avec le hint `WITH (INDEX(IS_CMD_LIGNES))`.

Oracle et SQL Server considèrent ces deux écritures équivalentes, alors que MySQL les voit différentes, l'avantage étant à l'opérateur `IN` sur les deux moteurs de `MySQL`.

6.1.6 Clause *Exists * versus* constante

On enseigne, depuis des années, que lorsqu'on emploie des sous-requêtes corrélées utilisant l'opérateur `EXISTS`, il faut retourner une constante (numérique ou texte) dans la clause `SELECT` de la sous-requête. Est-ce seulement pour des raisons de clarté ou aussi de performances ?

Listing 6.9 : Version *Exists* constante

```
select * from livres L
where not exists (select 1 from cmd_lignes CL where CL.NOLIVRE=L.NOLIVRE);
```

Listing 6.10 : Version *Exists **

```
select * from livres L
where not exists (select * from cmd_lignes CL where CL.NOLIVRE=L.NOLIVRE);
```

Nous ne documentons pas le détail des résultats. On constate que les colonnes retournées dans la sous-requête n'ont aucun impact sur les performances, alors qu'elles en avaient sur d'anciennes versions d'Oracle.

Cette pratique de mettre une constante garde cependant tout son sens du point de vue de la compréhension. Parfois, certains développeurs mettent ici un champ particulier laissant ainsi croire qu'il y aurait une sorte de lien possible avec ce champ.

6.1.7 Expressions sous requêtes

Listing 6.11 : Requête utilisant une jointure

```
select  c.nocmd,datecommande ,sum(montant)  Total
from cmd c,cmd_lignes cl  where cl.nocmd=c.nocmd and datecommande<to_
date('2004-04-18','yyyy-mm-dd')
group by c.nocmd, datecommande
```

Listing 6.12 : Requête utilisant une expression sous-requête

```
select nocmd,datecommande
     ,(select sum(montant) from cmd_lignes where nocmd=c.nocmd) Total
from cmd c where datecommande<to_date('2004-04-18','yyyy-mm-dd')
```

Nous choisissons ici une requête qui empêche d'appliquer le prédicat de la table CMD directement à la table CMD_LIGNES et qui ne retourne que 173 lignes. En changeant la date, nous testons une autre version qui ramène beaucoup plus de lignes (132 141).

	Petite requête		***Grosse requête***	
MyISAM	***Jointure***	***Expression sous-requête***	***Jointure***	***Expression sous-requête***
Temps	5,890 s	0,160 s	6,810 s	1,950 s
Reads	4 873 397	736	4 873 397	564 513
	Petite requête		***Grosse requête***	
InnoDB	***Jointure***	***Expression sous-requête***	***Jointure***	***Expression sous-requête***
Temps	0,670 s	0,670 s	3,580 s	4,030 s
Reads	1 001 449	1 000 175	2 264 285	1 632 143
	Petite requête		***Grosse requête***	
Oracle	***Jointure***	***Expression sous-requête***	***Jointure***	***Expression sous-requête***
Temps	0,390 s	0,060 s	1,48 s	1,51 s
Consistent Gets	11 490	3 316	11 490	117 464
	Petite requête		***Grosse requête***	
MS SQL Server	***Jointure***	***Expression sous-requête***	***Jointure***	***Expression sous-requête***
Temps	0,125 s	0,109 s	0,812 s	0,953 s

Avec des requêtes ramenant peu de lignes, l'avantage est aux expressions sous-requêtes. Avec des requêtes ramenant beaucoup de lignes, l'avantage passe à la version jointure, sauf pour le moteur MyISAM qui a l'air plus à l'aise avec l'utilisation systématique d'expressions sous-requêtes. Seul l'optimiseur de SQL Server adapte automatiquement son plan d'exécution à la solution la plus efficace, indépendamment de l'écriture. Nous avons donc utilisé les hints `loop join` et `merge join` pour effectuer ces tests.

6.1.8 Agrégats : *Having versus Where*

Quand c'est possible, les conditions doivent être placées dans la clause `WHERE` plutôt que dans la clause `HAVING` qui est évaluée après les opérations d'agrégation. Cela permet de profiter d'éventuels index et réduit le volume à traiter durant l'opération d'agrégation.

Listing 6.13 : Requête utilisant *Having* (incorrecte)

```
select pays,ville,count(*) from clients t
group by pays,ville
having pays='France'
```

Listing 6.14 : Requête utilisant *Where*

```
select pays,ville,count(*) from clients t
where t.pays='France'
group by pays,ville
```

MyISAM	***Having***	***Where***
Temps	0,080 s	0,030 s
Reads	91 109	45 950
InnoDB	***Having***	***Where***
Temps	0,090 s	0,050 s
Reads	91 132	45 960
Oracle	***Having***	***Where***
Temps	0,047 s	0,015 s
Consistent Gets	624	624
MS SQL Server	***Having***	***Where***
Temps	0,046 s	0,046 s