

Rencontre R20

Performance

Bases de données et programmation Web



❖ Stratégies d'optimisation

◆ Partitions

- Méthode qui consiste à **séparer la table** en plusieurs **sous-tables** selon un critère au choix. (Ex : Chaque **catégorie de produits** est dans sa propre **partition**)
- Chaque partition peut avoir ses propres **index**.
- Chaque partition peut même être sauvegardée dans une BD ou un serveur différent.
- Les requêtes qui exploitent bien les valeurs partitionnées deviennent **plus performantes**.
 - Ex : Je fais une requête qui s'intéresse seulement aux produits de type « **Automobile** » : seule la partition pour les produits de type **automobile** sera fouillée !
- Les requêtes qui utilisent des colonnes qui n'ont pas été utilisées pour faire la **partition** seront généralement **moins performantes**.
 - Il faut fouiller dans plusieurs partitions, faire les opérations dans **chaque** partition, séparément, puis **fusionner** le résultat.



❖ Stratégies d'optimisation

◆ Cache

- Généralement, sert à stocker des données fréquemment utilisées pour limiter la quantité d'accès au disque pour améliorer la performance.
- SQL Server : Utilise deux caches
 - **Buffer cache** : contient des « pages » de données fréquemment utilisées. Lorsqu'il est plein, les données plus vieilles ou moins utilisées sont retirées pour faire de la place.
 - : lorsqu'une requête est faite, un « plan d'exécution » est préparé et puis la requête peut ensuite être **Execution plan cache** exécutée. (grâce au plan d'exécution) Les plans d'exécution sont stockés dans ce cache pour ne pas avoir à les recréer si la même requête est utilisée.
- On peut **changer la taille** du cache !
 - Cache trop **souvent nettoyé** : Il est trop petit et n'aide pas vraiment à améliorer les performances.
 - Cache trop **rarement nettoyé** : Il est trop grand et un cache plus petit est généralement plus optimisé.
- Un indice intéressant pour juger la taille du cache est la « **Page Life Expantancy** », (**PLE**) c'est-à-dire la durée du passage d'une page dans le buffer cache. Il n'y a pas de mesure parfaite, mais sous 300 secondes, il est préférable d'augmenter la taille du cache.

Requête pour obtenir la PLE

```
SELECT*  
FROM sys.dm_os_performance_counters  
WHERE [counter_name] = 'Page life expectancy'
```



cntr_value → PLE en secondes

object_name	counter_name	instance_name	cntr_value	cntr_type
MSSQL\$SQLEXPRESS:Buffer Manager	Page life expectancy		52854	65792
MSSQL\$SQLEXPRESS:Buffer Node	Page life expectancy	000	52854	65792



❖ Stratégies d'optimisation

◆ Autres

- **Allocation de mémoire (*Scaling vertical*)** : Si le serveur est uniquement consacré à opérer la BD, s'assurer qu'autant de mémoire que possible est alloué à la BD SQL Server.
- **Séparation des fichiers (*Scaling horizontal*)** : Utiliser plusieurs machines et partitionner les données permet de diviser les tâches entre plusieurs serveurs.
- **Fréquence et moment de backup** : Les backups affectent beaucoup la performance. Selon l'importance des données, en faire moins souvent (si possible) ou de les faire à des moments de faible achalandage. N'oubliez pas qu'il existe **plusieurs types** de backups. (Complet, différentiel, incrémental, etc.)
- **Utiliser une BD NoSQL ?** : Si l'application utilise beaucoup de données non structurées, (fichiers variés, photos, vidéos, etc.) une base de données **NoSQL** est généralement mieux adaptée. (Ex : Pour un réseau social avec beaucoup de contenu généré par les utilisateurs) Ne choisissez pas un **SGBD relationnel** par défaut !



- ❖ Il est important d'aller chercher uniquement les informations nécessaires.
 - ◆ Ce sera d'autant plus important lorsque vous travaillerez en entreprises. Celles-ci utilisent généralement un **cloud storage pour gérer les BD**.
 - Ce type de service facture le client selon le volume des requêtes exécutées.
 - ◆ C'est aussi très important lorsque vous développez des applications mobiles.



❖ Au niveau de la BD:

- ◆ **Utiliser les vues** au lieu des requêtes Link complexes.

Lorsque vous créez une vue, la requête sous-jacente est optimisée par le serveur SQL. Son plan d'exécution est mis en cache dans le serveur pour une performance optimale.

Plus les requêtes Link sont complexes, moins elles sont performantes par rapport à une vue.



❖ Au niveau de la BD:

◆ **Utiliser les index** pour optimiser l'exécution des requêtes.

- Si vous avez une requête pour voir les données sur les ventes les plus récentes, vous voudrez faire un index sur le champ DateVente DESC.
- Les index sur les clés étrangères peuvent améliorer l'efficacité des jointures.



❖ Au niveau du code:

◆ **Utiliser des filtres** avant d'aller chercher les données avec `.ToListAsync()`

```
List<Course> premieresCourses = await _context.Courses.Take(30).ToListAsync();
```




❖ Au niveau du code:

Vous avez une vue Razor qui vous permet d'appliquer des filtres sur les données des participations.

Sussy Kart

Tutoriel

Jouer

Statistiques

Toutes les participations

Pseudo

Course

8 Classique

Ordre

Par chrono

TypeOrdre

Décroissant

Page

1

Filtrer

Pseudo	Course	Nombre de joueurs	Position	Chrono	Date
pikachudude420	8 Classique	4	1	120	2023-03-13 23:41:36
ezpilot420	8 Classique	1	3	120	2023-02-11 13:05:56
bruhb0t3	8 Classique	3	3	120	2023-01-04 23:48:58
toyotadude4	8 Classique	2	1	120	2023-03-02 04:41:48
fastmonster3	8 Classique	3	2	120	2023-03-12 21:39:49
pr0princess7	8 Classique	2	1	120	2023-01-30 09:44:15



❖ Au niveau du code:

- ◆ Dans le contrôleur STATS, dans l'action ToutesParticipationsFiltre(FiltreParticipationVM fpvm) vous allez chercher les 25000 enregistrements de la vue des participations avec .ToListAsync().

```
// Obtenir les participations grâce à une vue SQL -- ÉTAPE 4
List<VwDetailsParticipation> participations = await _context.VwDetailsParticipations.ToListAsync();
```

- ◆ Puis vous appliquez les filtres désirés pour afficher finalement que 30 enregistrements.

CECI N'EST PAS PERFORMANT.

ALLER CHERCHER 25000 enregistrements pour n'en AFFICHER que 30!!!



❖ Au niveau du code:

- ◆ Utiliser **.AsQueryable()** au lieu de .ToListAsync() pour obtenir un objet de type **IQueryable<>** puis appliquez les filtres sur cet objet.
- ◆ Utiliser ensuite .ToListAsync() sur cet objet à la fin pour aller chercher seulement les données qui nous intéressent.

```
//Construction d'une requête sur toutes les courses
IQueryable<Course> toutesLesCourses = _context.Courses.AsQueryable();

//Filtres pour garder les 30 courses les plus récentes
toutesLesCourses = toutesLesCourses.OrderByDescending(x => x.CourseId).Take(30);

//Effectivement aller chercher les données désirées dans la BD
List<Course> dernieresCourses = await toutesLesCourses.ToListAsync();
```



❖ Autres stratégies, au niveau du code:

- ◆ Aller chercher uniquement les colonnes qui vous intéressent, avec **.Select()**
 - Surtout si certaines colonnes de la table, que vous ne voulez pas voir, sont 'lourdes' comme `nvarchar(max)` par exemple.

```
C# Copier
foreach (var blog in context.Blogs)
{
    Console.WriteLine("Blog: " + blog.Url);
}
```

ICI, on veut seulement voir le Url du blog. Mais l'entité entière du Blog est extraite et les colonnes inutiles sont transférées à partir de la base de données.

Vous pouvez optimiser cela à l'aide de `Select` pour indiquer à EF les colonnes à sélectionner :

```
C# Copier
foreach (var blogName in context.Blogs.Select(b => b.Url))
{
    Console.WriteLine("Blog: " + blogName);
}
```