# OPTIMISATION DE REQUETES

La majorité des commandes effectuées sur les serveur sont des SELECT.

Voila pourquoi il est important d’optimiser leur écriture et de surveiller leur exécution.

# LES REGLES PRINCIPALES

## SELECT [champs] plutôt que SELECT *

Définir des champs dans l’instruction SELECT indiquera à la base de données d’interroger uniquement les données requises qui permettent de répondre aux exigences métier. Étudions un exemple dans lequel les exigences métier requièrent les adresses postales des clients.

## Eviter d’utiliser SELECT DISTINCT

**SELECT DISTINCT** permet de retirer facilement les éléments en double d’une requête. **SELECT DISTINCT** reGROUPe tous les champs dans la requête afin de créer des résultats distincts. Toutefois, une puissance de traitement considérable est nécessaire pour atteindre cet objectif. Par ailleurs, le regroupement des données peut rendre ces dernières imprécises.

Afin d’éviter l’utilisation de **SELECT DISTINCT**, il est préférable de sélectionner davantage de champs en vue de créer des résultats uniques.

## Créer des jointures avec INNER JOIN plutôt qu'avec WHERE

```
SELECT Customers.CustomerID, Customers.Name, Sales.LastSaleDate
FROM Customers, Sales
WHERE Customers.CustomerID = Sales.CustomerID
```

Ce type de jointure crée une jointure cartésienne, également appelée produit cartésien ou CROSS JOIN. Dans une jointure cartésienne, toutes les combinaisons possibles des variables sont créées.

Dans cet exemple, si nous avions 1 000 clients avec un total des ventes de 1 000, la requête générerait d'abord 1 000 000 de résultats, puis filtrerait les 1 000 enregistrements dans lesquels CustomerID est correctement joint.

## Créer des jointures avec INNER JOIN plutôt qu'avec WHERE

```
SELECT Customers.CustomerID, Customers.Name, Sales.LastSaleDate
FROM Customers, Sales
WHERE Customers.CustomerID = Sales.CustomerID
```

Il s'agit d'une utilisation inefficace des ressources de base de données, puisque cette dernière a réalisé 100 fois plus de recherches que nécessaire. Les jointures cartésiennes sont particulièrement problématiques dans les bases de données de grande échelle, car une jointure cartésienne de deux grandes tables crée des milliards ou des billions de résultats.

## Créer des jointures avec INNER JOIN plutôt qu'avec WHERE

Il est préférable d'utiliser INNER JOIN pour éviter de créer une jointure cartésienne :

```
SELECT Customers.CustomerID, Customers.Name, Sales.LastSaleDate
FROM Customers
    INNER JOIN Sales
    ON Customers.CustomerID = Sales.CustomerID
```

## Utiliser WHERE à la place de HAVING pour définir les filtres

À l'instar du concept susmentionné, l'objectif d'une requête efficace est d'extraire uniquement les enregistrements nécessaires de la base de données.

Conformément à l'ordre des opérations SQL, les instructions HAVING sont calculées après les instructions WHERE.

Une instruction WHERE est plus efficace si le but est de filtrer une requête selon des conditions.

**D’autres conseils utiles :**

- Utiliser les comparaison LIKE ET ‘%’ correctement.

- Lors de l’écriture / développement de la requête, penser à utiliser un TOP pour ne ramener qu’une partie des résultats.
- Si vos requêtes sont VRAIMENT trop gourmandes

# LES CLES

Les clés primaires et les clés étrangères sont deux types de contraintes qui peuvent être utilisées pour appliquer l'intégrité des données dans des tables SQL Server .

Ce sont des objets de base de données importants.

Règles pour la définition de la clé primaire :

- Une table ne peut contenir qu'une seule contrainte de clé primaire.
- Une clé primaire ne peut pas dépasser 16 colonnes et une longueur de clé totale de 900 octets.
- L'index généré par une contrainte de clé primaire ne peut avoir pour conséquence une augmentation du nombre d'index dans la table à plus de 999 index non cluster et un index cluster.
- Toutes les colonnes définies dans une contrainte de clé primaire doivent avoir une valeur autre que Null. Si vous ne spécifiez pas la possibilité de valeur NULL, toutes les colonnes participant à une contrainte de clé primaire ont des valeurs autres que Null.

# LES INDEX

Il existe plusieurs types d’Index :

- **Unique**

Un index unique garantit que la clé de l'index ne contient pas de valeur dupliquée et que toute ligne de la table ou de la vue est en quelque sorte unique.

L'unicité peut être une propriété à la fois des index cluster et non cluster.

Il existe plusieurs types d’Index :

- **Cluster**

Un index cluster trie et stocke les lignes de données de la table ou de la vue en fonction de la clé d'index cluster.

L'index cluster est mis en œuvre sous la forme d'une structure d'index arborescente binaire permettant la récupération rapide des lignes d'après leurs valeurs clés de l'index cluster.

Il existe plusieurs types d’Index :

- **Non-cluster**

Les index non cluster peuvent être définis dans une table ou dans une vue dotée d'un index cluster ou d'un segment de mémoire.

Chaque ligne d'un index non cluster contient la valeur clé non cluster ainsi qu'un localisateur de ligne.

Le localisateur pointe vers la ligne de données dans l'index cluster ou dans le segment doté de la valeur clé.

Les lignes de l'index sont stockées selon l'ordre des valeurs clés de l'index.

L'ordre spécifique des lignes de données n'est garanti que si un index cluster est créé sur la table.

Il existe plusieurs types d’Index :

- **columnstore**

Un index columnstore en mémoire stocke et gère des données à l'aide du stockage des données en colonnes et du traitement des requêtes en colonnes.

Les index columnstore fonctionnent bien pour les charges de travail de stockage de données qui effectuent principalement des chargements en masse et des requêtes en lecture seule.

Utilisez l’index columnstore pour atteindre des **performances des requêtes** pouvant être multipliées par 10 par rapport au stockage orienté lignes traditionnel.

Exemple