

# Optimisations des SGBDR

Étude de cas : MySQL

# Introduction

Pourquoi optimiser son application ?

# Introduction

Pourquoi optimiser son application ?

1. Gestion de gros volumes de données
2. Application critique (finance, ...)
3. Gagner des clients (Amazon, ... )

# Introduction

Où faut il optimiser son application ?

# Introduction

Où faut il optimiser son application ?

Identifier et optimiser les *bottlenecks* !

1. CPU
2. I / O (disque, réseau, ... )
3. Allocation mémoire

# Introduction

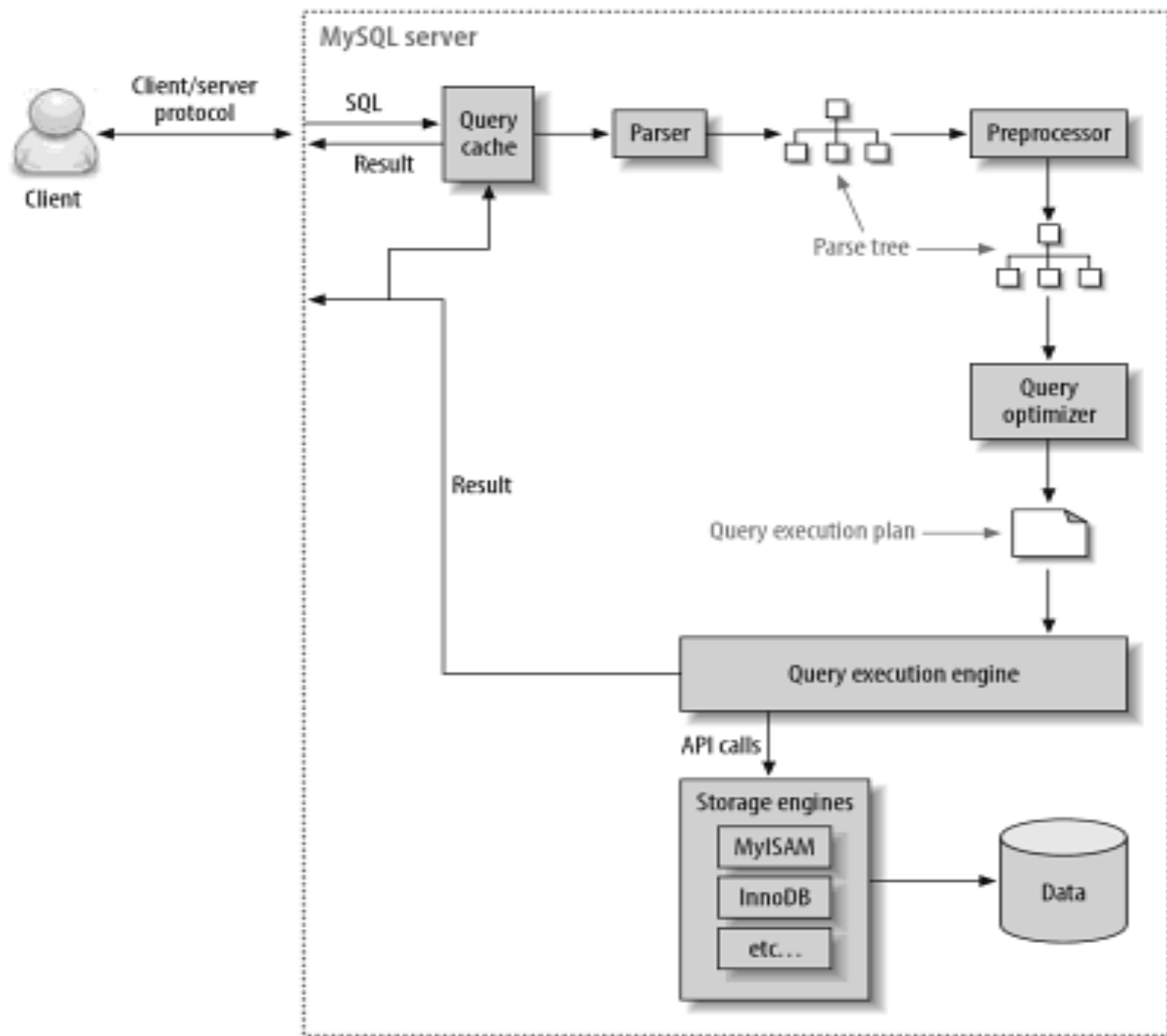
Un SGBDR utilise toutes ces ressources de manière intensive.

Il devient donc le *bottleneck* !

De manière générale, il faut améliorer sa base avant d'améliorer son application.

# Optimisations

1. Architecture
2. Moteurs de stockage
3. Index
4. Types de données
5. Performances et requêtes





# Moteurs de stockage

MyISAM, InnoDB, Archive

# Moteurs de stockage

Un moteur de stockage est un ensemble d'algorithmes permettant de gérer une table d'une base de données.

Chaque moteur a une vocation, un objectif, est une réponse à une problématique.

Choisir le bon moteur de stockage change de manière drastique les performances.

# Moteurs de stockage

Les plus populaires sont :

- MyISAM
- InnoDB
- Archive
- Memory

# MyISAM

Moteur par défaut.

Fonctionnalités : Non transactionnel, réparation manuelle et automatique, écriture de clés différées, compression, ...

Mais se veut en partie transactionnel...

# InnoDB

Transactionnel : MVCC, niveaux d'isolation, row-level locking, ...

Predictable read, performances énormes pour les requêtes sur clés primaires, ...

Chaque modification de table provoque sa reconstruction complète !

Réputé pour être plus lent que MyISAM

# Archive

Ne supporte que les insertions / sélections.

Les lignes sont compressées et les écritures placées dans un buffer.

Row-level locking (pourquoi ?)

En bref : optimisé pour les insertions et l'espace disque.

# Indexation

B-Tree et hash

# Indexation

Quel est l'intérêt d'indexer ses données ?



# Indexation

Quel est l'intérêt d'indexer ses données ?

Les performances !

Un index permet de retrouver très rapidement un ensemble de données.

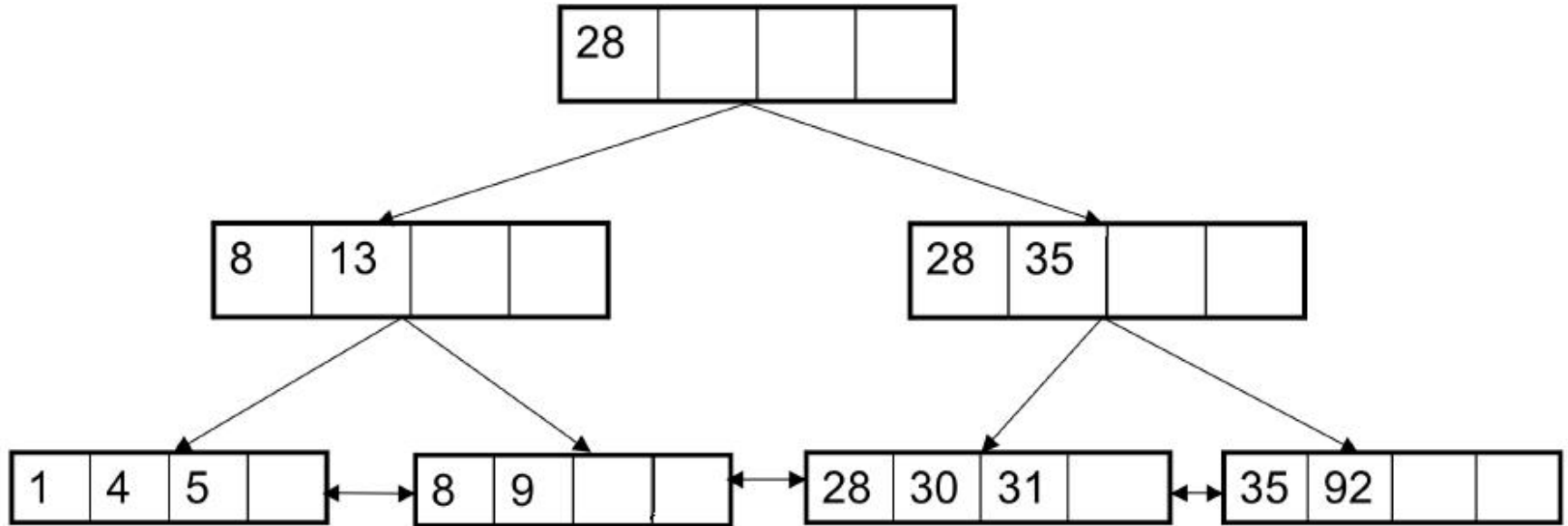
# B-Tree

Arbre équilibré

Avantages : Lecture séquentielle rapide -  
Accès très rapides

Inconvénients : Potentielle mauvaise localité -  
Taille de l'index pouvant être importante

# B-Tree



# Hachage

Création d'une clé associée à une valeur ou un groupe.

Avantages : Très rapide pour les insertions / suppressions / sélections avec égalité.

Inconvénients : Sélections ordonnées

# Indexation

Choisir son index :

```
CREATE INDEX id_index ON lookup (id) USING BTREE;
```

```
CREATE INDEX id_index ON lookup (id) USING HASH;
```

Par défaut, le B-Tree est utilisé.

# Types de données

# Types de données

Qu'apporte le bon choix du type de données ?

# Types de données

Qu'apporte le bon choix du type de données ?

Moins d'espace disque, de mémoire, de cycles CPU et d'espace dans le cache du CPU.



# Les entiers

TINYINT, SMALLINT, MEDIUMINT, INT, BIGINT codé respectivement sur 8, 16, 24, 32, 64 bits.

Peuvent être non signés (mot-clé UNSIGNED)

Quelles sont les différences entre INT(1) et INT(24) ?

# Le texte

Les implémentations des CHAR et VARCHAR varient avec le moteur de stockage.

CHAR : Taille fixe

VARCHAR : Taille variable

# VARCHAR

Seul le nécessaire est persisté.

Utilise entre 1 et 2 bytes pour stocker la taille.

L'action associée au dépassement varie avec le moteur de stockage.

# CHAR

Les espaces en fin de chaîne sont supprimés.

Des espaces sont ajoutés à la fin pour la comparaison.

Efficace lorsque la taille est prédéfinie ou très faible.

# NULL

Eviter NULL dès que possible !

Ajoute 1 byte par entrée.

Rend les index et les comparaisons plus complexes, donc l'optimisation des requêtes.

# Performances et requêtes

Requêtes SQL et cache

# Performances et requêtes

Récupérer plus de données que nécessaire.

MySQL ne renvoie pas progressivement les lignes, mais un ResultSet complet !

Comment réduire le nombre de lignes récupérées sans WHERE ?

# Performances et requêtes

Récupérer toutes les colonnes lors d'une jointure multiple.

Abuser du `SELECT *` est un crime !

Pourquoi ne pas simplement récupérer les colonnes qui nous intéressent ?



# Performances et requêtes

Utiliser des requêtes préparées.

La requête n'est parsée et optimisée qu'une seule fois. Utilisation d'un protocole unique pour l'envoi des paramètres.

Propre à une connexion, inutile si appelée une seule fois, attention au leak !

# Performances et requêtes

Compresser les données à travers le réseau.

Via le shell : `--compress`

Via JDBC : `<db_name>?useCompression=true`

# Performances et requêtes

Plein de requêtes VS requêtes complexes.

MySQL a été pensé pour gérer rapidement les petites requêtes.

Penser au cache ...

# Performances et requêtes

MySQL met en cache les requêtes de sélection.

Attention aux requêtes ne pouvant être mises en cache ! (NOW, CURRENT\_DATE).

Cache entièrement paramétrable

# Performances et requêtes

Lors de la réception d'une requête, si celle-ci commence par "SEL", on regarde le cache.

Comment s'effectue la recherche dans le cache ?

Est-ce toujours pertinent d'utiliser le cache ?

# Performances et requêtes

`SQL_CACHE / SQL_NO_CACHE` : Force la requête à être mise en cache, ou non.

`EXPLAIN` : Renvoie les informations sur le plan d'exécution

`DESCRIBE` : Décrit une table d'une base de données

# Conclusion

L'optimisation d'une application commence par son SGBD !

MySQL est un produit complet et souple grâce à ses moteurs de stockage.

Utilisé par les grands du Web (Twitter, Facebook, ...)