# Not Only SQL

# NoSQL

Qu'est-ce que le NoSQL ?

# NoSQL

Qu'est-ce que le NoSQL ?

Catégorie de SGBD s'affranchissant du modèle relationnel des SGBDR. Mouvance apparue par le biais des "grands du Web", popularisée en 2010.

# NoSQL

Pourquoi NoSQL ?

# NoSQL

Pourquoi NoSQL ?

- Licence des SGBDR très chère (Oracle, ...).

- Le SQL a un schéma fermé.

- Performances faibles, sur de gros volumes de données, comparées au NoSQL.

# NoSQL

Le NoSQL vise :

1. Gestion d'énormes quantités de données
2. Structuration faible du modèle
3. Montée en charge

# NoSQL

Il existe quatre types de SGBD NoSQL :

1. Orienté document (MongoDB, ...)
2. Clé / valeur (Redis, ...)
3. Orienté colonne (Cassandra, ...)
4. Orienté graphe (Neo4J, ...)

# Présentation de MongoDB

# Documents

MongoDB est orienté document. Qu’est-ce qu’un document ?

# Documents

MongoDB est orienté document. Qu’est-ce qu’un document ?

Un document est la représentation d'une donnée en BSON.

BSON = *Binary JSON*. Extension du JSON (support officiel du type Date, … ).

# Documents

Exemple :

```
{
   "name" : "MongoDB",
   "type" : "database",
   "count" : 1,
   "info" : {
                x : 203,
                y : 102
              }
}
```

# Organisation

Un serveur MongoDB est composé de bases de données.

Une base de données contient des collections.

Les collections possèdent les documents.

Chaque document possède un identifiant unique généré par MongoDB, le champ _id.

# Démarrage

MongoDB vient avec un shell : bin/mongo

Démarrage avec : bin/mongod

Quelques arguments :

- --dbpath <path> : Chemin de stockage des données.
- --port <port> : Port du serveur
- --replSet <nom> : Introduire le serveur dans un cluster de réplicas.

# Développement en Java avec Morphia

# Driver MongoDB / Morphia

MongoDB vient avec son propre driver Java

Morphia est une bibliothèque haut niveau pour MongoDB

Développement proche des ORM

# Initialisation

Créer une connexion au serveur :

```
MongoClient m = new MongoClient(“127.0.0.1”, 27017);
```

Créer et/ou récupérer une base de données :

```
new Morphia(Employee.class).createDatastore(m, “db”);
```

Le Datastore est l’interface pour communiquer avec MongoDB.

# DBObject

L'interface DBObject représente un document.

Très utilisée dans le driver MongoDB

Heureusement Morphia nous évite de l’utiliser

# Entity Morphia

```
@Entity("employees")
@Indexes(
    @Index(value = "salary", fields = @Field("salary"))
)
public class Employee {
    @Id
    private ObjectId id;
    private String name;
    @Reference
    private Employee manager;
    @Reference
    private List<Employee> directReports;
    @Property("wage")
    private Double salary;
}
```

# Insertion / Mise à jour

Méthode save de la classe Datastore

```
Employee employee = new Employee("Alain");
datastore.save(employe);
```

Cette méthode est surchargée et possède plusieurs variantes pour insérer. Le champ @Id est enregistré automatiquement

# Compter le nombre de documents

Méthode getCount() de la classe Datastore.

```
System.out.println(datastore.getCount(Employee.class));
```

Retourne le nombre de documents de la collection.

# Récupérer tous les documents

Méthode createQuery() de la classe Datastore.

```
return datastore.createQuery(Employee.class).asList();
```

Retourne l'ensemble des documents de la collection.

# Effectuer des requêtes

Méthode createQuery() de la classe Datastore

```
return datastore.createQuery(Employee.class)
.filter(“age”, 25).asList();
```

Retourne l'ensemble des documents de la collection dont le champ "age" égal 25.

# Suppression

Méthode delete de la classe Datastore.

```
Query<Employee> q = datastore
.createQuery(Employee.class).filter(“i”, 7);
datastore.delete(q);

Employee employee = new Employee("Alain");
datastore.delete(employee);
```

# Libération

Comme toutes les ressources persistantes, il faut toujours les libérer pour éviter les fuites.

On peut récupérer la connexion via le Datastore.

```
MongoClient m = new MongoClient(“127.0.0.1”, 27017);
m.close();
```

# Java

Pour une utilisation simple de MongoDB :

Créer vos entités avec un @Id de type ObjectId

Lisez la documentation :
http://mongodb.github.io/morphia/1.3/

# Indexation

# Introduction

Très similaire aux SGBDR, l'indexation dans MongoDB se fait sur un ou plusieurs champs.

Permet d'améliorer les performances de recherche.

Cela améliore t'il toujours les performances ?

# Présentation

Les indexes sont stockés au niveau des collections.

Apporte une surcharge pour les opérations d'écriture.

Le fonctionnement interne est très proche de ce que l’on trouve dans les SGBD actuels.

# Présentation

Quel est le type d'index dans MongoDB ?

# Présentation

Quel est le type d'index dans MongoDB ?

- B-Tree

- Hash

# Créer un index

Créer un index se résume à annoter votre entité avec @Index et la paramétrer.

```
Datastore datastore = new Morphia(Employee.class)
.createDatastore(m, “db”);

datastore.ensureIndexes();
```

# Conclusion

Penser à utiliser les indexes de manière efficace.

Un champ peu requêté n'a aucun intérêt à être indexé

Bien que l'on parle de NoSQL, le fonctionnement des indexes est similaire au monde SQL.

# Mongo Shell

# Shell

Le meilleur moyen d'interroger MongoDB est d'utiliser le shell.

Les commandes s'effectuent en JavaScript et les données sont en BSON.

Le shell possède l'autocomplétion.

# Shell

Afficher la base de données courante : db

Afficher la liste des bases de données : show dbs

Sélectionner une base de données : use <name>

Afficher les collections : show collections

# Shell

Les commandes ont la syntaxe suivante :

db.<collection>.<methode>

Exemple :

```
db.inventory.find( { qty: { $gt: 20 } } )
db.students.insert({"name": "Olivier", "etude" : "Master"})
```

# Shell

Toute l'administration de MongoDB se fait grâce au shell.

La documentation et les exemples donnés par le site sont en JavaScript, autrement dit pour le shell.

Le shell MongoDB est très simple à utiliser.

# Réplication

# Introduction

Qu'apporte la réplication ?

# Introduction

Qu'apporte la réplication ?

- Redondance

- Simplification de tâches (backups, ... )

- Augmentation de la capacité de lecture

# Introduction

Un *replica set* est un cluster d'instances MongoDB.

Stratégie maître / esclaves

Il doit TOUJOURS y avoir un unique maître.

Les clients effectuent les écritures sur l'instance ... ?

# Type de réplication

La réplication du maître vers les esclaves est asynchrone.

Quels sont les avantages et inconvénients ?

# Type de réplication

La réplication du maître vers les esclaves est asynchrone.
Quels sont les avantages et inconvénients ?

Synchrone : Bloquant / Coûteux / Forte cohérence

Asynchrone : Non bloquant / Rafraîchissement des données obligatoires.

# Tolérance aux pannes

Un *replica set* est tolérant aux pannes.

Si le noeud primaire tombe, les noeuds secondaires peuvent élire un nouveau noeud primaire.

Comment rendre l'élection automatique ?

# Tolérance aux pannes

Comment rendre l'élection automatique ?

- Détection de la mort du noeud primaire (ping / heartbeat)

- Lancement d'une élection

- Le noeud ayant reçu une majorité de vote devient le noeud primaire, grâce à une **priorité**.

# Consistance

Que se passe t'il si un noeud primaire accepte une écriture et tombe en panne avant la réplication de l'écriture ?

On perd la donnée, et le *replica set* devient inconsistent.

Comment corriger ce problème ?

# Consistance

Idée inspirée des SGBDR : Le rollback

Le noeud primaire écrit en local les opérations demandées lorsqu'il accepte une écriture.

Lors de son retour, soit il relance les opérations, soit il les annule (les rollback).

# Préoccupations

Lors de la mise en place d'un *replica set*, deux paramètres sont à prendre en compte :

- Write Concern : Message envoyé pour vérifier la validité d'une opération.

- Read Preferences : Favoriser les lectures sur les noeuds secondaires.

# Write Concern

Qualité de chaque opération d'écriture et décrit le montant de préoccupation d'une application pour l'écriture.

Plus la préoccupation augmente, plus les performances augmentent, plus la cohérence diminue.

# Type de Write Concern

Erreurs ignorés : Pas de notification d'erreurs (réseau, ...)

Sans acquittement : Opérations non acquittées.

Acquittement : Opérations acquittées. Ne résiste pas au failover.

Journalisé : Opérations valides si acquittées et écrites dans le journal.

Acquittement du réplica : Tous les noeuds secondaires acquittent les opérations.

# Read Preferences

Par défaut, les opérations de lecture sont envoyées au noeud primaire.

Les lectures sur le noeud primaire garantissent d'obtenir toujours les données les plus fraîches.

Les lectures sur les noeuds secondaires améliorent le débit de lecture en distribuant les lectures.

# Read Preferences

Penser à modifier cela lorsque :

- Opérations n'affectant pas le front-end (backup, reporting, ...).

- Application distribuée géographiquement. On envoie le client sur le noeud secondaire le plus proche.

# Types de Read Preferences

Les différents type de read preferences sont :

**primary** : Toujours utiliser le noeud primaire. Exception si pas de noeud primaire.

**primaryPreferred** : Toujours utiliser le noeud primaire. On utilise les noeuds secondaires si pas de noeud primaire.

**secondary** : Toujours les noeuds secondaires. Exception si pas de noeuds secondaires.

**secondaryPreferred** : Toujours les noeuds secondaires. On utilise le noeud primaire si pas de noeuds secondaires.

**nearest** : On prend le noeud le plus proche, selon le choix fait par l'utilisateur.

# Conclusion

La réplication est l'un des fondements du NoSQL.

De fait, il est important d'en connaître le fonctionnement interne et les implications : élection, tolérance aux pannes, cohérence, ...

Notions supplémentaires : Arbitres, ...

# MapReduce

# Introduction

Qu'est-ce que le MapReduce ?

# Introduction

Qu'est-ce que le MapReduce ?

MapReduce est un patron d'architecture de développement informatique, popularisé (et non inventé) par Google, dans lequel sont effectués des calculs parallèles, et souvent distribués, de données potentiellement très volumineuses.

Utilisé dans tous les systèmes à forte volumétrie (NoSQL, BigData, ... ).

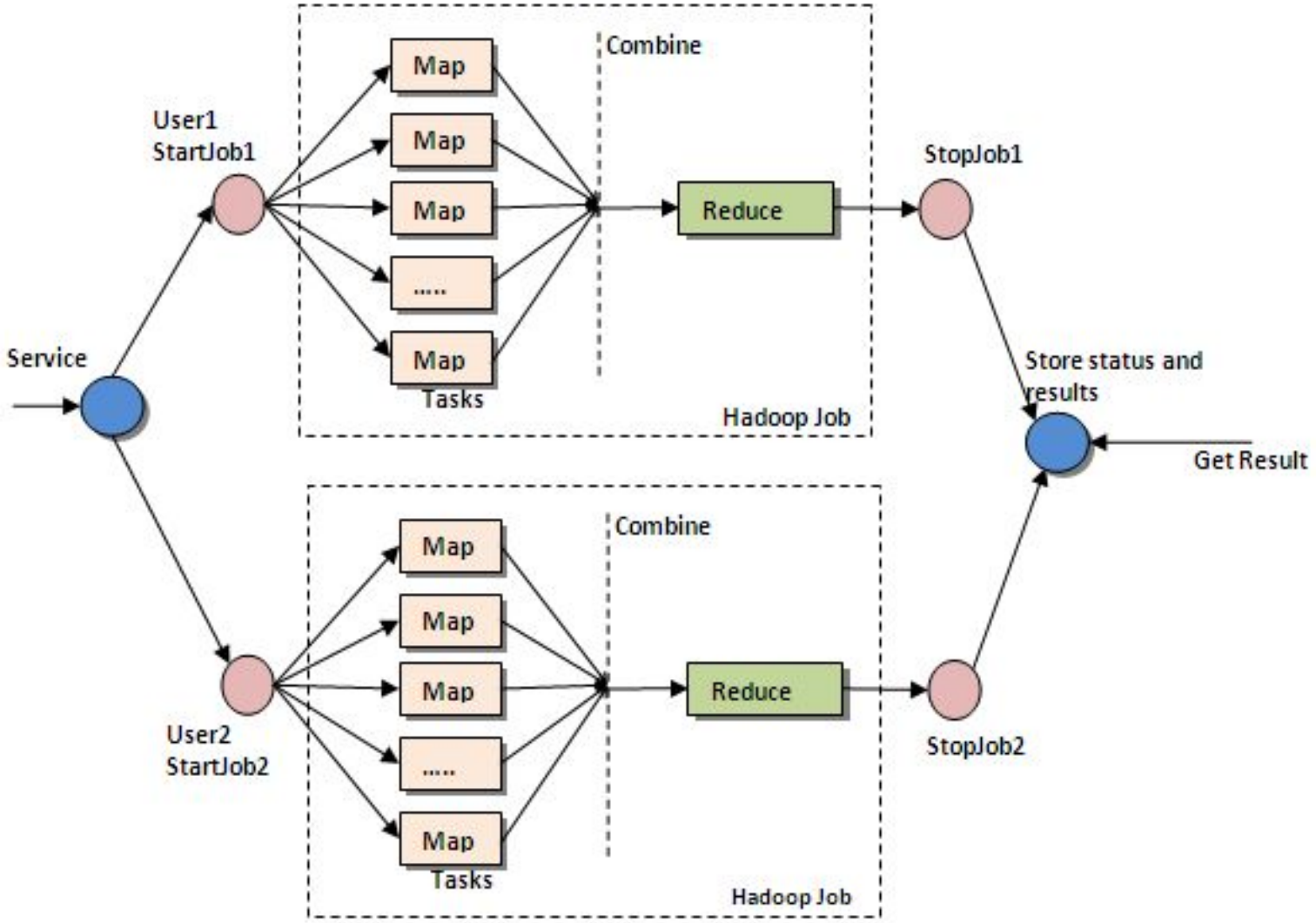
# Présentation

Une tâche MapReduce s'effectue en deux temps :

- Map : Analyse d'un problème, découpé en sous-problèmes (peut être récursif).

- Reduce : Remontée des résultats au noeud parent l'ayant sollicité.

# Exemple : Hadoop



# Dans MongoDB

Une tâche MapReduce dans MongoDB réalise :

- Lecture depuis la collection donnée en entrée
- Map
- Reduce
- Écriture dans la collection de sortie

On utilise donc une collection temporaire pour faire les opérations.

# Dans MongoDB

Consistance dans une opération MapReduce :

- La phase de lecture consomme un verrou partagé. Libéré tous les 100 documents.

- L'insertion dans la collection temporaire consomme un verrou exclusif pour chaque écriture.

- Si la collection de sortie n'existe pas, la création consomme un verrou exclusif.

- Si la collection de sortie existe, les actions de sorties consomme un verrou exclusif.

# Exemple

## Création de l'opération Map

```
var mapFunction1 = function() {emit(this.cust_id, this.price);};
```

## Création de l'opération Reduce

```
var reduceFunction1 = function(keyCustId, valuesPrices) {
                return Array.sum(valuesPrices);
 };
```

## Lancement de l'opération MapReduce

```
db.orders.mapReduce(mapFunction1,reduceFunction1,{ out:
"map_reduce_example" })
```

# Conclusion

L'objectif du MapReduce est de gérer de gros volumes de données. C'est inutile dès lors que vous en avez peu.

Pour cela, vous pouvez utiliser *Aggregation Framework*.

Avec l'avènement du BigData, le MapReduce a le vent en poupe. Il est donc primordial de le connaître.

# Conclusion

# Conclusion

MongoDB est l'un des plus importants SGBD NoSQL actuel.

Cette technologie est jeune et contient d'importants pièges !
Ne vous fiez pas à 100% à cette tendance.

Il est néanmoins sûr qu'elle sera présente dans les prochaines années à venir.