

# NoSQL et Bases de données documentaires et réparties

- Draft -

Khaled Jouini

j.khaled@gmail.com

Institut Supérieur d'Informatique et des Technologies de Communication

2017-2018

## Chapitre 1 - Syllabus

- Sujet du cours

- Objectifs
- Pré-requis
- Pour aller plus loin
- Plan du cours

## Le sujet, en bref

### Documents et bases de documents

Comprendre ce qu'est une base de données constituée de **documents** et savoir évaluer les **méthodes**, **outils** et **systèmes** pour les gérer.

Problématiques associées :

- notion de **document**, structuré, semi-structuré, non structuré ;
- **représentation et interrogation**, et aussi mise à jour et transactions.
- **distribution, élasticité** pour gérer de très grands volumes de données.

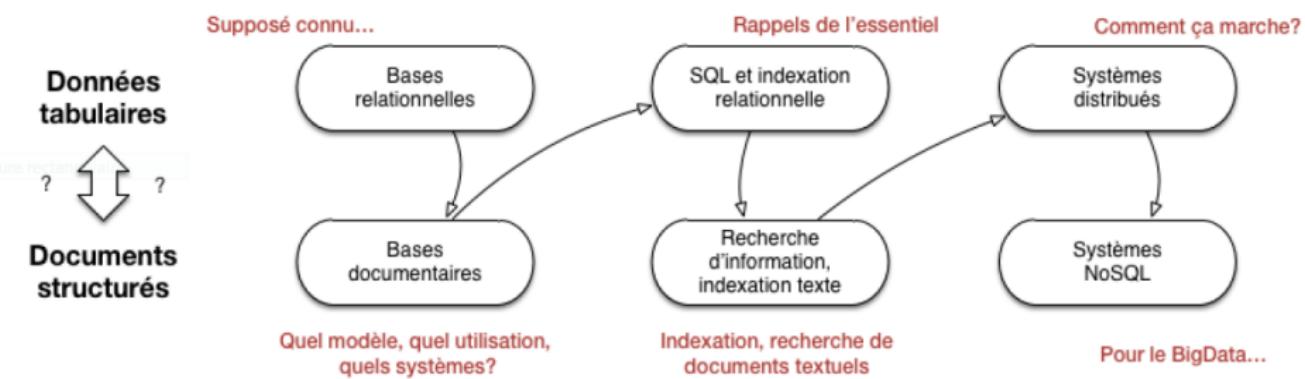
**Question transversale** : base relationnelle ou base NoSQL ? Quand, pourquoi, comment ?

Des mots-clés à comprendre (et à critiquer) : données structurées/non structurées, bases NoSQL, Cloud, BigData, moteurs de recherche, etc.

## Un panorama du sujet

### Mise en perspective base relationnelle / base documentaire

#### Représentation → Recherche → Passage à l'échelle



## Bases relationnelles

Les **bases relationnelles** sont utilisées dans toutes les applications gérant des informations **structurées** et **régulières** : applications de “gestion”, applications Web, applications mobiles.

Des fonctionnalités extrêmement fortes

- Une modélisation (quasi-) normalisée.
- Un langage (SQL) très bien défini, normalisé lui aussi.
- De très bonnes performances, obtenues **automatiquement**
- Une gestion robuste de la concurrence d'accès

**Attention à bien apprécier ce qu'on gagne / perd en passant au "NoSQL"**

## Problématique 1 : Notion de "document" et de base documentaire

Document = unité d'information autonome ou quasi-autonome.

- Peu ou pas de référence à d'autres documents.
- Peu ou pas de structure ; ou une structure très flexible.
- Un contenu souvent à orientation multimédia.

**Examples (1)** : documents textuels, types documents Web.

**Examples (2)** : images, documents audios, vidéos ; pas de structure explicite, production de descripteurs synthétiques pour tenter de les indexer.

**Examples (3)** : jeux en ligne : artifacts graphiques, objets 3D, actions utilisateur.

**Examples (4)** : tous les fichiers de votre ordinateur...

### Défi

Impose de repenser la notion de schéma et de représentation.

## Problématique 1 : Notion de "document" et de base documentaire

Soit un ensemble (important) de **documents**, comment les gérer ?

**Dans un système de fichier ?** Pourquoi pas, mais

- Manipulations laborieuses ; aucun contrôle de contenu ; peu sécurisé ; fonction de recherche primitive.
- Vraiment pas pratique pour construire des applications.

**Dans une base relationnelle ?** Pourquoi pas, mais

- Sous-utilisation du système (stockage par BLOB, SQL inutilisable, transactions élémentaires).
- Peu de gains.

**Dans un système orienté-document ?**

- Un système de Gestion Electronique de Documents (GED), type Alfresco.
- Une **base documentaire** = des fonctionnalités BD, spécialisées documents.

## Problématique 2 : La recherche

Dans les bases documentaires, peu ou pas de structure fixe,

- SQL inadapté.
- Recherche « exacte » souvent insatisfaisante.

La recherche s'effectue souvent **par similarité**

- on fournit un document "requête"
- le système recherche les documents **proches** du document-requête.
- implique une notion de distance, et donc un **classement** du résultat.

Par exemple, quand on recherche sur le Web :

- on fournit un ensemble de mots-clés : c'est le document requête
- le moteur de recherche trouve les documents les plus proches (on verra comment)
- le classement (et sa pertinence) sont des éléments essentiels.

## Problématique 3 : données à très grande échelle

On atteint facilement des **volumes** de données extrêmement importants.

- les moteurs de recherche qui collectent des **documents** disponibles sur le Web.
- les applications utilisées à l'échelle du Web ; commerce électronique (Amazon) ; réseaux sociaux (Facebook).
- données gérées par les jeux en ligne.

Les collections occupent typiquement des centaines de Gigaoctets, voire des Téraoctets.

### Solution

Nouveaux systèmes, dits “NoSQL” pour gérer de vastes collections de documents de manière scalable

- pour les accès temps réel ;
- pour les traitements analytiques (MapReduce et au-delà).

## Chapitre 1 - Syllabus

- Sujet du cours
- **Objectifs**
- Pré-requis
- Pour aller plus loin
- Plan du cours

# Objectifs

Le cours vise à vous transmettre, **dans un contexte pratique**, deux types de connaissances.

## Connaissances fondamentales

- **Représentation de documents textuels** : les formats XML et JSON ; les langages de manipulation ;
- **Recherche dans les bases documentaires** : principes, techniques, moteurs de recherche, index, algorithmes.
- **Stockage, gestion, et scalabilité par distribution.** L'essentiel sur les systèmes distribués ; le cas des systèmes NOSQL.

## Connaissances pratiques

- Des systèmes "NoSQL" orientés « documents » ; pour JSON (MongoDB, CouchDB) pour XML (BaseX).

## Chapitre 1 - Syllabus

- Sujet du cours
- Objectifs
- **Pré-requis**
- Pour aller plus loin
- Plan du cours

## Pré-requis

Les connaissances suivantes sont requises pour suivre ce cours.

- **Compréhension des bases relationnelles**, soit au moins la conception d'un schéma, SQL, ce qu'est un index et des notions de base sur les transactions.  
⇒ si nécessaire, des rappels seront effectués en cours.
- **Une aisance minimale dans un environnement de développement**. Editer un fichier, lancer une commande, ne pas paniquer devant un nouvel outil, savoir résoudre un problème avec un minimum de tenacité, etc.  
⇒ Vous devez reproduire les exemples donnés.
- **La connaissance de langage de programmation comme Java ou PHP vous aidera**.  
⇒ La réalisation de quelques fonctionnalités vous permettra de consolider vos nouvelles connaissances.

## Chapitre 1 - Syllabus

- Sujet du cours
- Objectifs
- Pré-requis
- **Pour aller plus loin**
- Plan du cours

## Pour aller plus loin...

### NoSQL à la maison

- MongoDB. <https://www.mongodb.org/downloads#production>
- Alternativement, Hadoop/HBase, ElasticSearch, BaseX, CouchDB et une centaine d'autres systèmes NoSQL (<http://nosql-database.org/>)

### Références utiles

- Document de référence : *Bases de données documentaires et distribuées*, Ph. Rigaux, Notes de cours, CNAM, 2015.
- Documentation MongoDB : [https://docs.mongodb.org/manual/?\\_ga=1.24474105.237649131.1443698254](https://docs.mongodb.org/manual/?_ga=1.24474105.237649131.1443698254). (Documents de référence.)
- *Dynamo: Amazon's Highly Available Key-value Store*, Giuseppe DeCandia & al., SOSP'07, 2007.
- *Bigtable: A Distributed Storage System for Structured Data*, Fay Chang & al., OSDI'06, 2006.
- *Tutorial: MapReduce Theory and Practice of Data-intensive Applications*, Pietro Michiardi, Notes de cours, Eurecom, 2010.
- *Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web*. D. Karger & al., ACM Symposium on Theory of Computing. pp. 654(663), 1997.

## Pour aller plus loin...

### Jeux de test (JSON)

- <https://github.com/10gen-labs/ipsum> : générateur de documents JSON conçu par 10-gen pour fournir des jeux de test à MongoDB.
- <http://dblp.org/search/index.php> : export des résultats de recherche sous format JSON
- <http://generatedata.com/> : générateur de documents avec différents formats. Ne permet pas la génération de documents imbriqués.
- Collection movies utilisée dans les exemples du cours [https://drive.google.com/file/d/0BxrjRB\\_M10MVOHJHQ3RDDW5KM2c/view?usp=sharing](https://drive.google.com/file/d/0BxrjRB_M10MVOHJHQ3RDDW5KM2c/view?usp=sharing)

## Chapitre 1 - Syllabus

- Sujet du cours
- Objectifs
- Pré-requis
- Pour aller plus loin
- Plan du cours

1 Syllabus

2 Documents structurés (données semi-structurées)

3 MongoDB, une base JSON

4 Cloud et données massives

5 Systèmes NoSQL : la réPLICATION

6 Systèmes NoSQL : le partitionnement (sharding)

7 Systèmes NoSQL : calcul distribué avec MapReduce

8 Systèmes NoSQL et transactions

## Chapitre 2 - Documents structurés (données semi-structurées)

- Introduction
- JSON
- Modèle relationnel vs. documents structurés
- Bases de données documentaires

# Introduction

- Documents structurées (XML et plus récemment JSON) :
  - Permettent la représentation des données semi-structurées (*i.e.* structure et typage souples)
  - Facilitent l'échange de données entre applications hétérogènes
  - Fournissent des règles de représentation de l'information "neutres" (*i.e.* non propriétaires) par rapport aux applications susceptibles de traiter l'information
- Notions essentielles :
  - Les données sont auto-décrivantes** Le contenu vient avec sa propre description.
  - Structures riches** Le contenu se décrit avec des listes, des enregistrements imbriqués, des ensembles.
  - Typage flexible** Les données peuvent être typées ("c'est un entier"), et/ou structurées ("cette partie du graphe doit avoir telle forme"), et/ou ni l'un ni l'autre. Dans ce dernier cas c'est à l'application d'effectuer le contrôle.
  - Sérialisation** Structure et contenu doivent pouvoir être transformés ensemble en une chaîne de caractères autonome.

XML est supposé connu, nous présentons brièvement JSON.

## Chapitre 2 - Documents structurés (données semi-structurées)

- Introduction
- **JSON**
- Modèle relationnel vs. documents structurés
- Bases de données documentaires

## Format général

JSON (JavaScript Object Notation) :

- Format d'échange de données, léger, text-based, neutre, interprétable facilement par les machines et les humains.
- Dérivé de la notation des objets du langage JavaScript
- Initialement utilisé comme langage de transport de données par AJAX et les services Web  
(var ObjJS = JSON.parse(json\_data);)

Deux structures de données : Objets et tableaux.

**Objet** : liste associative de paires (**clé : valeur**).

```
{"nom": "Jouini", "tel": 66557788, "email": "j.khaled@gmail.com"}
```

**Tableau** (array) : séquence ordonnée de 0 ou plusieurs valeurs.

```
{nom: "Khaled", "tels": [66557788, 11223344]}
```

**Valeur** : peut être une chaîne de caractères, un nombre, un booléen (True/False), Null, ou bien un objet ou un tableau (imbrication de structures)

```
{"nom": {"prenom": "Khaled", "famille": "Jouini"},  
"tel": [{"fixe": 66557788}, {"portable": 11223344}],  
"email": "j.khaled@gmail.com"}
```

# Format général

## Remarques

Les exemples précédents montrent la version sérialisée<sup>1</sup>, soit un codage du graphe sous forme d'une chaîne de caractères stockable sur disque et échangeable par réseau.

MongoDB utilise pour le stockage et la transmission de données le format BSON (Binary JSON). BSON, entre autres, préfixe les éléments volumineux d'un document pour optimiser le scan.

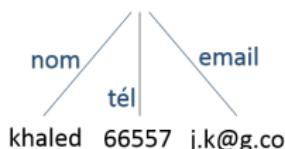
---

<sup>1</sup>la sérialisation désigne la capacité à coder un document sous la forme d'une séquence d'octets qui peut "voyager" sans dégradation sur le réseau

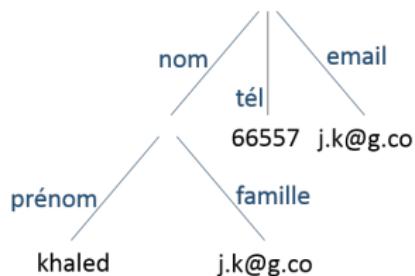
## Vue conceptuelle : modélisation sous forme arborescente

Dans un document JSON on distingue :

- **Une structure** : graphe ou **arbre**. Les étiquettes sont sur les arêtes, les valeurs sur les feuilles.
- **Un contenu** : Le texte dans les feuilles.



Arbre représentant un agrégat



Arbre représentant une composition d'agrégats

Toutes **les opérations** (de recherche, de navigation, ...) s'appuient sur la structure.

## Chapitre 2 - Documents structurés (données semi-structurées)

- Introduction
- JSON
- Modèle relationnel vs. documents structurés
  - Passage relationnel/semi-structuré
  - Jointure
  - Discussion
- Bases de données documentaires

## Plus puissant que la représentation relationnelle?

Oui, car facile de représenter des données régulières. Exemple d'une table :

<b>id</b>	<b>nom</b>	<b>prénom</b>
37	Tarantino	Quentin
167	De Niro	Robert
168	Grier	Pam

Facile à représenter sous forme d'un document (très régulier).

```
[  
    artiste: {"id": 37, "nom": "Tarantino", "prenom": "Quentin"},  
    artiste: {"id": 167, "nom": "De Niro", "prenom": "Robert"},  
    artiste: {"id": 168, "nom": "Grier", "prenom": "Pam"}  
]
```

Table relationnelle = arbre de hauteur constante. Trois niveaux : ligne, attribut, valeur.

## Plus puissant que la représentation relationnelle?

On peut donc aussi représenter des données régulières

Probablement pas du tout efficace. **Pourquoi ?**

⇒ peut être utile pour échanger des informations ; pas pour les stocker.

### Premier constat

- Dans une base relationnelle, les données sont très contraintes / structurées : **on peut stocker séparément la structure (le schéma) et le contenu (la base).**
- Une représentation arborescente XML / JSON est plus appropriée pour des données de structure **complexe** et / ou **flexible**.

**Exercice** : trouvez une autre représentation permettant de séparer la description du contenu et de compacter la représentation JSON.

## Qu'en est-il de la jointure relationnelle?

Grâce à **L'imbrication des structures**, il est possible de représenter des données avec une complexité arbitraire.

Par exemple on peut représenter ensemble un film **et** son metteur en scène.

```
{  
    "title": "Pulp fiction",  
    "year": "1994",  
    "genre": "Action",  
    "country": "USA",  
    "director": {  
        "last_name": "Tarantino",  
        "first_name": "Quentin",  
        "birth_date": "1963"  
    }  
}
```

## Questions de fond

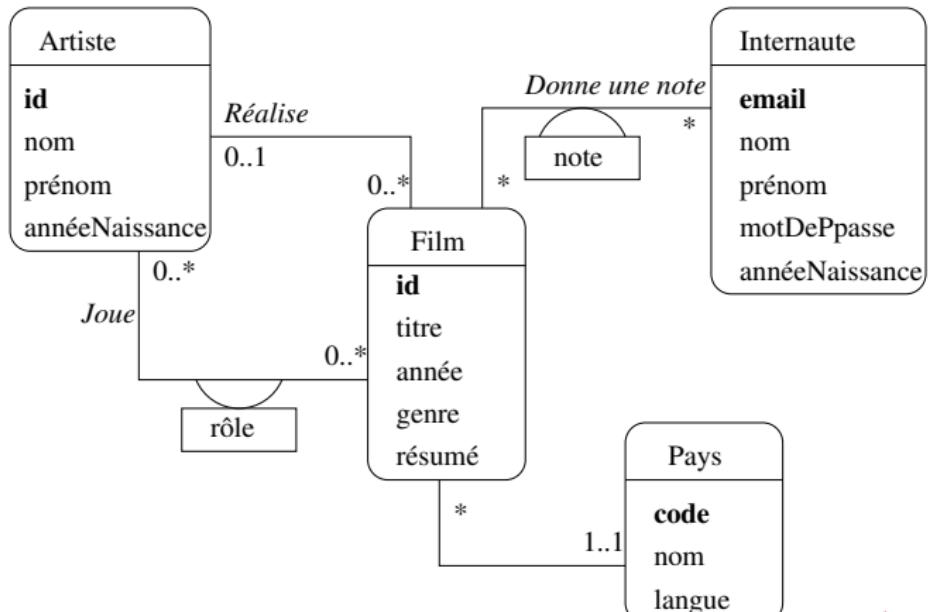
Est-ce **plus** puissant qu'en relationnel ? Avantages ? Inconvénients ?

## Qu'en est-il de la jointure relationnelle?



```
{ "customer_id": 1,  
  "first_name": "Mark",  
  "last_name": "Smith",  
  "city": "San Francisco",  
  "phones": [ {  
      "type": "work",  
      "number": "1-800-555-1212"  
    },  
    { "type": "home",  
      "number": "1-800-555-1313",  
      "DNC": true  
    },  
    { "type": "home",  
      "number": "1-800-555-1414",  
      "DNC": true  
    }  
  ]  
}
```

## Exemple de la base des films



## En relationnel

Exemple avec associations 1 à plusieurs, et plusieurs à plusieurs.

- Film (**id**, titre, année, *idReal*)
- Artiste (**id**, nom, prénom, année)
- Role (*idFilm*, *idArtiste*, role)

Caractéristique du relationnel :

- on **normalise** en représentant tout "à plat" (lignes)
- il faut faire des **jointures** pour reconstituer l'information.
- il faut effectuer **plusieurs écritures** pour une même entité (transactions)

Dans les bases documentaires : on essaie de créer des unités d'information **autonomes** pour éviter d'avoir à faire des jointures.

### Intuition

Les systèmes NoSQL sont conçus pour passer à l'échelle par distribution. C'est en grande partie incompatible avec les jointures et les transactions.

## Représentation relationnelle du film

Les films :

	id	titre	année	idReal
	17	Pulp Fiction	1994	37
	57	Jackie Brown	1997	37

Les artistes :

	id	nom	prénom	naissance
	37	Tarantino	Quentin	1963
	11	Travolta	John	1954
	27	Willis	Bruce	1955

Les rôles

	idFilm	idArtiste	rôle
	17	11	Vincent Vega
	17	27	Butch Coolidge
	17	37	Jimmy Dimmick

### Essentiel : effet de la normalisation

- il faut effectuer **plusieurs écritures** pour une même entité (transactions)
- il faut effectuer des **jointures** pour reconstituer une entité.

## Modèle agrégé de données (Aggregate Data Model)

```
{  
    "_id": "movie:17",  
    "title": "Pulp Fiction",  
    "year": "1994",  
    "director": {  
        "last_name": "Tarantino", "first_name": "Quentin",  
        "birth_date": "1963"  
    },  
    "actors": [  
        {  
            "first_name": "John", "last_name": "Travolta",  
            "birth_date": "1954", "role": "Vinent Vega"  
        },  
        {  
            "first_name": "Bruce", "last_name": "Willis",  
            "birth_date": "1955", "role": "Butch Coolidge"  
        },  
        {  
            "first_name": "Quentin", "last_name": "Tarantino",  
            "birth_date": "1963", "role": "Jimmy Dimmick"  
        }  
    ]  
}
```

## Discussion

La représentation par document a deux inconvénients forts.

- **Chemin d'accès privilégié** : les films apparaissent près de la racine des documents, les artistes sont enfouis dans les profondeurs ;  
L'accès aux films est donc privilégié
- **Redondance** : la même information doit être représentée plusieurs fois, ce qui est tout à fait fâcheux (Quentin Tarantino est représenté deux fois).

La redondance mène à des incohérences.

Privilégier un chemin d'accès est bon pour certaines applications, mauvais pour d'autres.

### Bien comprendre

Le seul avantage d'une modélisation par document structuré, **pour des données régulières**, et d'éviter les jointures. Pour tous les autres aspects c'est une **régression** qui mène à de nombreux problèmes. Vous êtes prévenus !

## Avec références vers d'autres documents

Solution : une **collection** de films

```
{  
    "title": "Pulp fiction",  
    "year": "1994",  
    "director": "artiste:37",  
    "actors": [{"artist:11", "role": "Vincent Vega"},  
              {"artist:27", "role": "Butch Coolidge"},  
              {"artist:37", "role": "Jimmy Dimmick"}]  
}
```

Qui référence une **collection** d'artistes.

```
[  
  {"_id": "11", "first_name": "John", "last_name": "Travolta"},  
  {"_id": "27", "first_name": "Bruce", "last_name": "Willis"},  
  {"_id": "37", "first_name": "Quentin", "last_name": "Tarantino"}]
```

### Second constat

Dès qu'on manipule des notions **d'entités**, **d'identité** et de **référence**, la « bonne » représentation tend à devenir celle du modèle relationnel.

## Pour aller plus loin...

A faire chez vous :

- Suivre la conférence en ligne de Tugdual Grall sur la modélisation de données documentaires  
<https://www.youtube.com/watch?v=csKBT8zkRf0>
- Préparer un mini rapport (ou une petite présentation pour ceux qui le souhaitent) résumant :
  - la problématique;
  - les différences entre le modèle documentaire et le modèle relationnel;
  - et surtout les patterns communs utilisés pour transposer les associations 1-1, 1-N et M-N dans le modèle documentaire.

Lecture intéressante : Patterns de modélisation pour les bases documentaires

<https://docs.mongodb.org/manual/data-modeling/>

## Chapitre 2 - Documents structurés (données semi-structurées)

- Introduction
- JSON
- Modèle relationnel vs. documents structurés
- Bases de données documentaires

## Rôle d'une BD documentaire

Un **système de gestion de bases documentaires** fournit les services suivants.

- Stockage, préservation, sécurité des accès.
- Accès partagé et distant.
- Interrogation, recherche par contenu.
- Outils de traitement, de transformation.
- Passage à l'échelle par distribution.

## Petit panorama

C'est la galaxie des systèmes NoSQL...

### Format XML

- BaseX,
- eXist,
- Oracle XML DB

### Format JSON

- MongoDB
- CouchDB,
- autres....

Pas de format du tout (clé, valeur)

- Riak, MemCache
- HBase, Cassandra
- ....

## Discussion

Quand utiliser (ou pas) une base documentaire ?

- quand les documents contiennent peu ou pas de références ;
- **ou** quand on peut se permettre la redondance (peu de MaJ), **totale ou partielle** ;
- **on veut traiter de très gros volumes de manière “scalable”.**

Conditions non remplies : un système relationnel est **toujours** une option à considérer.

### Conclusion

C'est du cas par cas en fonction de l'application. Décision basée sur une réflexion préalable approfondie.

## Ce qu'il faut retenir

Les documents semi-structurels se caractérisent par les aspects suivants.

- Une structure **flexible** (présence/absence, variations), **complexe** (imbrication), et **auto-décrise**.
- Une double représentation : une vue conceptuelle (arbre) et son codage (forme sérialisée).
- **Valable** pour les documents "multimédia" au sens large : rapports, images, vidéos. **Dangereux** pour des données fortement structurées.
- La notion d'autonomie (pas d'association entre documents) est essentielle.
  - Autonomie ? On peut mettre en œuvre un **passage à l'échelle par distribution**. Typique des systèmes NoSQL.
  - Pas d'autonomie : un jour ou l'autre il faudra faire des jointures ; les systèmes NoSQL ne sont pas bons pour ça.
- Ne pas oublier : les schémas, le langage de requêtes, l'optimisation, la concurrence d'accès, ça compte beaucoup !

## Chapitre 3 - MongoDB, une base JSON

- Qu'est ce que MongoDB?
- Installation et opérations basiques
- Requêtes Mongo
- Map Reduce : premiers pas

## Qu'est ce que MongoDB?

MongoDB, un système "NoSQL", l'un des plus populaires

- fait partie des NoSQL dits "documentaires" (avec CouchDB)
- s'appuie sur un modèle de données semi-structuré (encodage JSON) ;
- pas de schéma (complète flexibilité) ;
- un **langage d'interrogation** original (et spécifique) ;
- pas (ou très peu) de **support transactionnel**.

Construit dès l'origine comme un système **scalable** et **distribué**.

- distribution par partitionnement (*sharding*) ;
- technique adoptée : découpage par intervalles (type BigTable, Google) ;
- tolérance aux pannes par réPLICATION.

## Qu'est ce que MongoDB?

MongoDB, un système "NoSQL", l'un des plus populaires

- fait partie des NoSQL dits "documentaires" (avec CouchDB)
- s'appuie sur un modèle de données semi-structuré (encodage JSON) ;
- pas de schéma (complète flexibilité) ;
- un **langage d'interrogation** original (et spécifique) ;
- pas (ou très peu) de **support transactionnel**.

Construit dès l'origine comme un système **scalable** et **distribué**.

- distribution par partitionnement (*sharding*) ;
- technique adoptée : découpage par intervalles (type BigTable, Google) ;
- tolérance aux pannes par réPLICATION.

# Qu'est ce que MongoDB?

Vue d'ensemble sur les principaux concepts (communs à plusieurs systèmes NoSQL).

SQL Terms/Concepts	NoSQL Terms/Concepts
database	database
table	collection
row	document or BSON document
column	field
index	index
table joins	embedded documents and linking
primary key Specify any unique column or column combination as primary key.	primary key In MongoDB, the primary key is automatically set to the <code>_id</code> field.

Suite de la comparaison **SQL to MongoDB Mapping Chart**

<http://docs.mongodb.org/manual/reference/sql-comparison/>

## Chapitre 3 - MongoDB, une base JSON

- Qu'est ce que MongoDB?
- **Installation et opérations basiques**
- Requêtes Mongo
- Map Reduce : premiers pas

## Éléments d'installation

- MongoDB est un système libre de droits (pour sa version de base), téléchargeable à <http://www.mongodb.org/downloads>.
- Après le téléchargement et la décompression vous obtenez un répertoire mongo, agrémenté du numéro de version et autres indicateurs. Nous le désignerons par mongodir.
- Ajoutez mongodir/bin dans la variable d'environnement PATH
- Pour lancer le serveur il suffit de taper la commande mongod (se placer sous mongodir/bin si ce chemin n'a pas été ajouté à PATH)
- Si tout se passe bien vous le serveur se met à l'écoute sur le port 27017 et vous obtenez le message suivant :

```
[initandlisten] waiting for connections on port 27017
```

## Opérations basiques

- Il vous faut maintenant un client pour envoyer les requêtes au serveur mongodb.
- Vous pouvez utiliser soit l'interpréteur de commandes de mongo, soit un client graphique
- Il existe plusieurs clients graphiques, les plus répandus étant Rockmongo et RoboMongo
- Pour lancer l'interpréteur de commandes (toujours sous ) : `mongo`
- Interpréteur de commandes mongo : un interpréteur Javascript (ce qui est cohérent avec la représentation JSON). On peut donc lui soumettre des instructions en Javascript, ainsi que des commandes propres à MongoDB.
- La base par défaut s'appelle `test`
- Pour se placer dans une base : `use <nombase>`
- Une base est constituée d'un ensemble de **collections**
- Une collection est l'équivalent d'une table en relationnel. Pour créer une collection :  
`db.createCollection("movies")`

## Opérations CRUD

```
db.users.insertOne( ← collection
{
    name: "sue", ← field: value
    age: 26, ← field: value
    status: "pending" ← field: value
}
) ← document

db.users.find( ← collection
    { age: { $gt: 18 } }, ← query criteria
    { name: 1, address: 1 } ← projection
).limit(5) ← cursor modifier

db.users.updateMany( ← collection
    { age: { $lt: 18 } }, ← update filter
    { $set: { status: "reject" } } ← update action
)
) ←

db.users.deleteMany( ← collection
    { status: "reject" } ← delete filter
)
)
```

## Opérations basiques

- La liste des collections est obtenue par :

```
show collections
```

- Pour insérer un document JSON dans une collection (ici, movies) :

```
db.movies.insert ( {"nom": "mrid" })
```

- Il existe donc un objet (javascript) implicite, db, auquel on soumet des demandes d'exécution de certaines méthodes.

- Pour afficher le contenu d'une collection :

```
db.movies.find()
```

- On obtient des objets (javascript, encodés en JSON)

```
{ "_id" : ObjectId("5422d9095ae45806a0e66474") , "nom" : "mrid" }
```

- MongoDB associe un identifiant unique à chaque document, de nom conventionnel \_id, et lui attribue une valeur si elle n'est pas indiquée explicitement.

## Opérations basiques

- Pour insérer un autre document :

```
db.movies.insert ({ "produit": "Grulband", prix: 230, enStock: true})
```

- Vous remarquerez qu'il n'y a pas de schéma (et donc pas de contraintes), ce qui revient à reporter les problèmes (contrôles, contraintes, tests sur la structure) vers l'application.

- On peut affecter un identifiant explicitement :

```
db.movies.insert ({_id: "1", "produit": "Kramolk", prix: 10, enStock: true})
```

- On peut compter le nombre de documents dans la collection :

```
db.movies.count()
```

- Import d'un fichier JSON (à partir de l'invite de commandes et non de l'interpréteur)

```
mongoimport -d mrid -c movies --file movies.json --jsonArray
```

L'argument jsonArray indique à l'utilitaire d'import qu'il s'agit d'un tableau d'objets à créer individuellement, et pas d'un unique document JSON.

- Suppression d'une collection :

```
db.movies.drop()
```

## Chapitre 3 - MongoDB, une base JSON

- Qu'est ce que MongoDB?
- Installation et opérations basiques
- **Requêtes Mongo**
  - Aperçu du langage de requêtes
  - Jointure
- Map Reduce : premiers pas

## Premier exemple

```
db.inventory.insertMany([
  { item: "journal", qty: 25, size: { h: 14, w: 21, uom: "cm" }, status: "A" },
  { item: "notebook", qty: 50, size: { h: 8.5, w: 11, uom: "in" }, status: "A" },
  { item: "paper", qty: 100, size: { h: 8.5, w: 11, uom: "in" }, status: "D" },
  { item: "planner", qty: 75, size: { h: 22.85, w: 30, uom: "cm" }, status: "D" },
  { item: "postcard", qty: 45, size: { h: 10, w: 15.25, uom: "cm" }, status: "A" }
]);
```

```
db.inventory.find( {} )
```

```
db.inventory.find( { status: "D" } )
```

```
db.inventory.find( { status: { $in: [ "A", "D" ] } } )
```

```
db.inventory.find( { status: "A", qty: { $lt: 30 } } )
```

```
db.inventory.find( { "size.h": { $lt: 15 }, "size.uom": "in", status: "D" } )
```

```
db.inventory.find( { $or: [ { status: "A" }, { qty: { $lt: 30 } } ] } )
```

```
db.inventory.find( {
  status: "A",
  $or: [ { qty: { $lt: 30 } }, { item: /^p/ } ]
} )
```

## Document exemple

```
{  
    "_id": "movie:1",  
    "actors": [  
        {  
            "_id": "artist:15",  
            "birth_date": "1908",  
            "first_name": "James",  
            "last_name": "Stewart",  
            "role": "John Ferguson"  
        },  
        {  
            "_id": "artist:16",  
            "birth_date": "1925",  
            "first_name": "Kim",  
            "last_name": "Novak",  
            "role": "Madeleine Elster"  
        }  
    ],  
    "country": "USA",  
    "director": {  
        "_id": "artist:3",  
        "birth_date": "1899",  
        "first_name": "Alfred",  
        "last_name": "Hitchcock"  
    },  
    "genre": "drama",  
    "summary": "Scottie Ferguson, ancien inspecteur de police, est  
    ayant des tendances suicidaires. Amoureux de la jeune femme Scot  
    "title": "Vertigo",  
    "year": 1958  
}
```

## Aperçu du langage de requêtes

On peut exprimer l'équivalent du select-from-where, **sur une seule collection** (pas de jointure).

```
db.movies.find ({"_id": "movie:2"})
```

Avec des **expressions de chemin** (simples !)

```
db.test.find ({"director.last_name": "Scott"} )
```

Fonctionne **aussi** pour les acteurs (même si c'est un **tableau**).

```
db.test.find ({"actors.last_name": "Weaver"})
```

Mise à jour :

```
db.test.update ({"_id": "movie:2"},  
                {$set: {sous_titre: "Le passager"} } )
```

```
db.test.update ({"_id": "movie:2"}, { $unset: {country: 1} } )
```

Et on supprime tout :

```
db.test.remove ()
```

## Aperçu du langage de requêtes

Recherche par clé (très rapide).

```
db.artists.find({_id: "artist:99"})
```

Remarquez l'utilisation de `findOne`. Avec un chemin.

```
db.movies.find({"actors._id": "artist:99"})
```

Avec pagination

```
db.movies.find({"actors._id": "artist:99"}).skip(0).limit(2)
```

Avec une **projection**

```
db.movies.find({"actors._id": "artist:99"}, {"title": 1})
```

Avec une expression régulière (pas de guillemets !) :

```
db.movies.find({"title": /^Re/}, {"title": 1})
```

Par intervalle :

```
db.movies.find({"year": {$gte: 2000, $lte: 2005}, {"title": 1}})
```

## Aperçu du langage de requêtes

### Quelques clauses ensemblistes

```
db.artists.find({_id: {$in: ["artist:34", "artist:98", "artist:1"]}}, {
```

Autres possibilités : \$nin, \$all, \$exist

```
db.movies.find({'summary': {$exists: true}}), {"title": 1})
```

## Opérateurs booléens

- Par défaut, quand on exprime plusieurs critères, c'est une conjonction (`and`) qui est appliquée.
  - Exemple (les films tournés avec Leonardo DiCaprio en 1997) :

```
db.movies.find({$and : [{"year": "1997"}, {"actors.last_name": "DiCaprio"}] })
```
- L'opérateur `and` s'applique à un tableau de conditions.
- Il existe un opérateur `or` avec la même syntaxe.
- Exemple (Les films parus en 1997 ou avec Leonardo DiCaprio) :

```
db.movies.find({$or : [{"year": "1997"}, {"actors.last_name": "DiCaprio"}] })
```
- Grossso modo, on obtient la même expressivité que pour SQL pour les recherches dans une même collection. Que faire quand on doit croiser des informations présentes dans plusieurs collections?

## Jointure

- La **jointure** proprement dite, **n'existe pas** dans MongoDB.
- C'est cohérent avec l'approche NoSQL et le Aggregate Data Model, mais on peut imaginer plein de situations où les jointures sont nécessaires.
- Le serveur ne sachant pas faire de jointures, on est réduit à les faire **côté client** (ou alors avec un Map/Reduce, nous y reviendrons)
- Exemple

### Collection moviesRef

```
{  
    "_id": "movie:1"  
    "title": "Pulp fiction",  
    "year": "1994",  
    "genre": "dark comedy",  
    "director": "artist:37",  
    "actors": [ {"_id": "artist:11", "role": "Vincent Vega" },  
               {"_id": "artist:27", "role": "Butch Coolidge"},  
               {"_id": "artist:37", "role": "Jimmy Dimmick"} ]  
}
```

...

### Collection artists

```
{"_id": "artist:11", "f_name": "John", "l_name": "Travolta"}  
{"_id": "artist:27", "f_name": "Bruce", "l_name": "Willis"}  
{"_id": "artist:37", "f_name": "Quentin", "l_name": "Tarantino"}
```

...

## Jointure

- Le client ici est le Shell Mongo (qui est un interpréteur JavaScript).

- Exemple 1 : "Trouver les films de Quentin Tarantino"

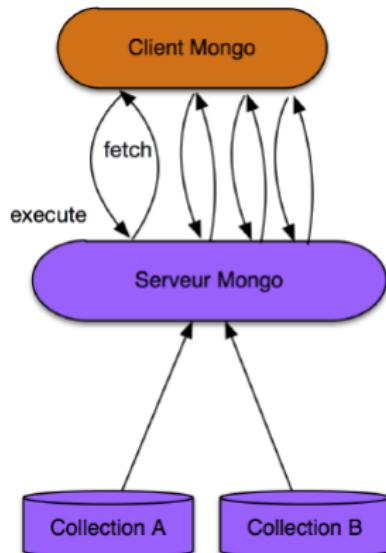
- var tarantino = db.artists.findOne({ "first\_name": "Quentin", "last\_name": "Tarantino" })
  - db.moviesRef.find({ "director": tarantino.\_id }, { "title": 1 })

- Exemple 2 : "Retourner pour chaque film son titre et les informations sur son réalisateur"

- var lesFilms = db.moviesRef.find()  
while(lesFilms.hasNext()) {  
 var film = lesFilms.next();  
 var mes = db.artists.findOne("\_id": film.director);  
 printjson(film.title);  
 printjson(mes);  
}

## Jointure

- On a donc une boucle et une requête imbriquée!
- C'est exactement la **méthode qui serait utilisée par le serveur** si ce dernier implantait les jointures.
- L'exécuter du côté client induit un surcoût en programmation, et en **échanges réseau** entre le client et le serveur.



## Chapitre 3 - MongoDB, une base JSON

- Qu'est ce que MongoDB?
- Installation et opérations basiques
- Requêtes Mongo
- **Map Reduce : premiers pas**
  - Préliminaires
  - MongoDB et MapReduce
  - MapReduce et Jointure
  - Clés composées

# MapReduce qu'est ce que c'est?

## Quelques chiffres (2010)

- Volume de données traitées quotidiennement par Google : 20 PB
- Volume de données traitées quotidiennement par FaceBook : 15 TB

## Constats

- ➊ *Data-Intensive Workloads* (ou encore **Big Data**) : le traitement des données est rapide comparativement à l'accès aux données sur le disque et leur transfert sur le réseau.
- ➋ Grâce au partitionnement/parallélisation, un ensemble de machines serveurs à bas coût (*Commodity Servers*), permettent en général une meilleure **scalabilité** et de meilleures performances que les super-calculateurs

## Pistes pour le traitement de données massives

- Partitionner les données ou les traitements sur plusieurs "travailleurs"<sup>2</sup>
  - Traiter les partitions en parallèle là où elles se trouvent (pas de déplacements de données)
  - Agréger les résultats intermédiaires issus des différents travailleurs
- => c'est le principe sur lequel s'appuie le **framework MapReduce**

<sup>2</sup>Un "travailleur" peut être : des threads d'un processeur, des coeurs d'un processeur multi-coeurs, des processeurs d'une même machine, des machines (virtuelles ou réelles) d'une même grappe de serveurs (*cluster*)

# Principe

## Fonction Map

- Appliquée à chaque **item** : prend toujours en entrée un document en ce qui nous concerne;
- Pour chaque document elle **peut retourner 0, 1 ou plusieurs paires (key, value)**

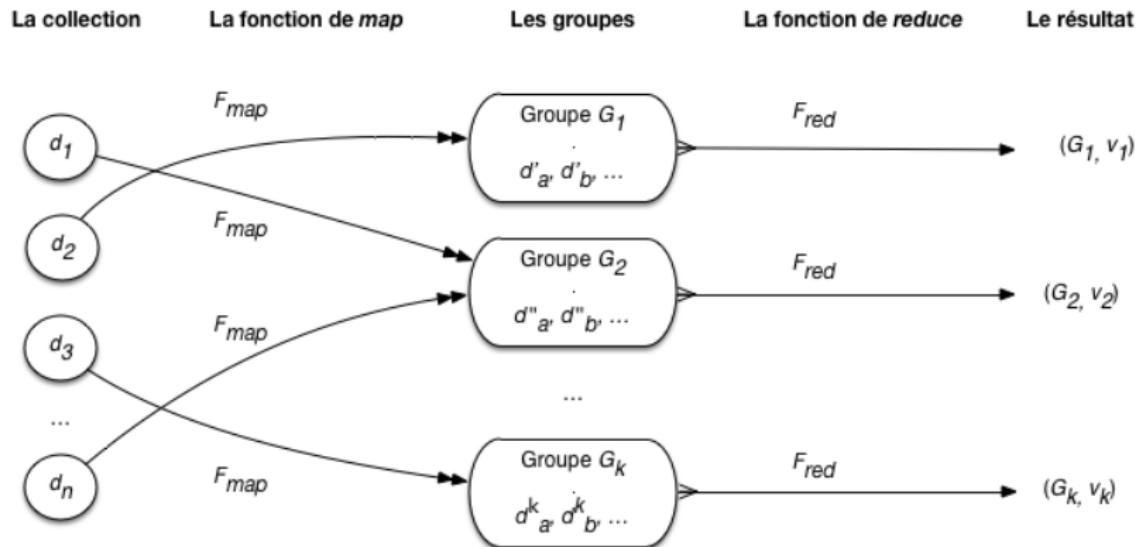
## Shuffle and sort

- Les **valeurs** émises par les différents mappers sont **groupées par clé**  $\Rightarrow$  **il existe un groupe par clé**
- Un groupe est formé par une clé associée à une liste de valeurs **(k,list(v))**
- Shuffle : chaque **(k,list(v))** est transmise à un reducer  $\Rightarrow$  Reduce est exécutée autant de fois qu'il n'y a de clé (par des noeuds différents)
- Phase transparente à l'utilisateur (rappelez-vous, c'est un framework)

## Fonction Reduce

- Appliquée à chaque paire **(k,list(v))** et retourne une paire **(k,v)**
- Important : logiquement ne démarre que lorsque TOUS les mappers ont terminé
- Dans la pratique : les reducers se lancent sans attendre que tous les mappers terminent. Si un groupe reçoit une nouvelle valeur, le reducer correspondant est re-exécuté. Le output de la dernière exécution du reducer est mis en input de l'exécution suivante.

# Principe



# MapReduce et MongoDB

- Il existe plusieurs implémentations de MapReduce :Hadoop, Mongo, etc.
- Nous nous concentrerons sur le cas centralisé sous Mongo, avant de passer plus tard au cas réparti
- Document exemple

```
{  
    "_id": "movie:1",  
    "actors": [  
        {  
            "_id": "artist:15",  
            "birth_date": "1908",  
            "first_name": "James",  
            "last_name": "Stewart",  
            "role": "John Ferguson"  
        },  
        {  
            "_id": "artist:16",  
            "birth_date": "1925",  
            "first_name": "Kim",  
            "last_name": "Novak",  
            "role": "Madeleine Elster"  
        }  
    ],  
    "country": "USA",  
    "director": {  
        "_id": "artist:3",  
        "birth_date": "1899",  
        "first_name": "Alfred",  
        "last_name": "Hitchcock"  
    },  
    "genre": "drama",  
    "summary": "Scottie Ferguson, ancien inspecteur de police, est ayant des tendances suicidaires. Amoureux de la jeune femme Scottie, il tente de l'empêcher de se jeter du Golden Gate Bridge."}  
}
```

## Exemple : regroupement de films

(SELECT title FROM movies GROUP BY directorId)

On veut regrouper les films par réalisateur

La fonction  $F_{map}$ .

```
var mapRealisateur = function() {
    emit(this.director._id, this.title);
};
```

Combien aura-t-on d'accumulateurs ?

La fonction  $F_{red}$ .

```
var reduceRealisateur = function(directorId, titres) {
    var res = new Object();
    res.director = directorId;
    res.films = titres;
    return res;
};
```

Soumission au framework

```
db.movies.mapReduce(mapRealisateur, reduceRealisateur, {out: {"inline": 1}} )
```

## Exemple : regroupement de films

Chaque exécution de  $F_{red}$  produit un document de la forme.

```
{  
  "_id" : "artist:3",  
  "value" : {  
    "director" : "artist:3",  
    "films" : [  
      "Vertigo",  
      "Psychose",  
      "Les oiseaux",  
      "Pas de printemps pour Marnie",  
      "La mort aux trousses"  
    ]  
  }  
}
```

## Exemple : regroupement de films

- MongoDB n'applique la fonction `reduce` que lorsqu'une clé possède plusieurs valeurs qui lui sont associées.
- Si une clé possède une seule valeur, la paire (clé, valeur) est retournée telle quelle.
- **Exemple :** Grouper les films par réalisateur

En supposant que les documents soient bien formés (chaque document de la collection d'entrée est un film ayant un et un seul réalisateur), interprétez les métriques ci-après, retournées après l'exécution du précédent MapReduce (combien existe t-il de films, de réalisateurs, de réalisateurs qui ont réalisé un seul film, etc.?)

```
{  
    "results" : [...],  
    "timeMillis" : 172,  
    "counts" : {  
        "input" : 65,  
        "emit" : 65,  
        "reduce" : 13,  
        "output" : 40  
    }  
}
```

- Important : la fonction `reduce` de MongoDB ne peut pas retourner un tableau de valeurs (dans un tel cas mettre le tableau comme attribut d'un objet et retourner l'objet)»

## Exercices

Écrire en pseudo-code les fonctions Map/Reduce permettant de :

- ① Compter les films par genre (SELECT count(\_id) FROM..GROUP BY genre)

```
var mapGenre = function() { emit(this.genre, 1); };
var reduceGenre = function(genre, values) { return values.length; }
```

- ② Compter le nombre de films par acteur (SELECT count(\_id) FROM..GROUP BY actor.\_id)

```
var mapFilmsActeur = function() {
    for (var i = 0; i < this.actors.length; i++) {
        emit(this.actors[i]._id, 1);
    }
}
var reduceFilmsActeur = function(acteur_id, values) {return values.length; }
```

- ③ Compter le nombre d'acteurs dirigés par un réalisateur (SELECT count(actors.\_id) FROM..GROUP BY director.\_id)

```
var mapActeursRealisateur = function(){
    emit(this.director._id, this.actors.length);
}
var reduceActeursRealisateur = function(realisateur, acteurs) {
    var res=0;
    for (var i=0; i<acteurs.length; i++) {res = res + acteurs[i];}
    return res;
}
```

## Exercices

Écrire en pseudo-code les fonctions Map/Reduce permettant de :

- ① Compter les films par genre (SELECT count(\_id) FROM..GROUP BY genre)

```
var mapGenre = function() { emit(this.genre, 1); };
var reduceGenre = function(genre, values) { return values.length; }
```

- ② Compter le nombre de films par acteur (SELECT count(\_id) FROM..GROUP BY actor.\_id)

```
var mapFilmsActeur = function() {
    for (var i = 0; i < this.actors.length; i++) {
        emit(this.actors[i]._id, 1);
    }
}
var reduceFilmsActeur = function(acteur_id, values) {return values.length; }
```

- ③ Compter le nombre d'acteurs dirigés par un réalisateur (SELECT count(actors.\_id) FROM..GROUP BY director.\_id)

```
var mapActeursRealisateur = function(){
    emit(this.director._id, this.actors.length);
}
var reduceActeursRealisateur = function(realisateur, acteurs) {
    var res=0;
    for (var i=0; i<acteurs.length; i++) {res = res + acteurs[i];}
    return res;
}
```

## Exercices

Écrire en pseudo-code les fonctions Map/Reduce permettant de :

- ① Compter les films par genre (SELECT count(\_id) FROM..GROUP BY genre)

```
var mapGenre = function() { emit(this.genre, 1); };
var reduceGenre = function(genre, values) { return values.length; }
```

- ② Compter le nombre de films par acteur (SELECT count(\_id) FROM..GROUP BY actor.\_id)

```
var mapFilmsActeur = function() {
    for (var i = 0; i < this.actors.length; i++) {
        emit(this.actors[i]._id, 1);
    }
}
var reduceFilmsActeur = function(acteur_id, values) {return values.length; }
```

- ③ Compter le nombre d'acteurs dirigés par un réalisateur (SELECT count(actors.\_id)  
FROM..GROUP BY director.\_id)

```
var mapActeursRealisateur = function(){
    emit(this.director._id, this.actors.length);
}
var reduceActeursRealisateur = function(realisateur, acteurs) {
    var res=0;
    for (var i=0; i<acteurs.length; i++) {res = res + acteurs[i];}
    return res;
}
```

## Exercices

Écrire en pseudo-code les fonctions Map/Reduce permettant de :

- ① Compter les films par genre (SELECT count(\_id) FROM..GROUP BY genre)

```
var mapGenre = function() { emit(this.genre, 1); };
var reduceGenre = function(genre, values) { return values.length; }
```

- ② Compter le nombre de films par acteur (SELECT count(\_id) FROM..GROUP BY actor.\_id)

```
var mapFilmsActeur = function() {
    for (var i = 0; i < this.actors.length; i++) {
        emit(this.actors[i]._id, 1);
    }
}
var reduceFilmsActeur = function(acteur_id, values) {return values.length; }
```

- ③ Compter le nombre d'acteurs dirigés par un réalisateur (SELECT count(actors.\_id)  
FROM..GROUP BY director.\_id)

```
var mapActeursRealisateur = function(){
    emit(this.director._id, this.actors.length);
}
var reduceActeursRealisateur = function(realisateur, acteurs) {
    var res=0;
    for (var i=0; i<acteurs.length; i++) {res = res + acteurs[i];}
    return res;
}
```

## Exercices

Écrire en pseudo-code les fonctions Map/Reduce permettant de :

- 1 Afficher les titres des films du genre "drama" par année

```
var mapDramaYear = function(){
    if (this.genre == "drama") {emit (this.year, this.title);}
}

var reduceDramaYear = function(year, Films) {return films;}
```

- 2 Afficher pour chaque acteur la liste des acteurs avec lesquels il a tourné des films

```
var mapAA = function(){
    for (var i=0; i<this.actors.length; i++){
        for (var j=0; j<this.actors.length; j++) {
            if(this.actors[i] != this.actors[j]){
                emit(this.actors[i]._id, this.actors[j]._id);
            }
        }
    }
}

var reduceAA = function(actorId, actors) {
    var res = new Object();
    res.actId=actorId; res.actors=[];
    for(var i=0; i < actors.length; i++)
    {
        var j=0;
        while(j<res.actors.length && actors[i] != res.actors[j]) {j++;}
        if(j==res.actors.length)res.actors.push(actors[i]);
    }
    return res;
}
```

## Exercices

Écrire en pseudo-code les fonctions Map/Reduce permettant de :

- 1 Afficher les titres des films du genre "drama" par année

```
var mapDramaYear = function(){
    if (this.genre = "drama") {emit (this.year, this.title);}
}

var reduceDramaYear = function(year, Films) {return films;}
```

- 2 Afficher pour chaque acteur la liste des acteurs avec lesquels il a tourné des films

```
var mapAA = function(){
    for (var i=0; i<this.actors.length; i++){
        for (var j=0; j<this.actors.length; j++) {
            if(this.actors[i]!==this.actors[j]){
                emit(this.actors[i]._id, this.actors[j]._id);
            }
        }
    }
}

var reduceAA = function(actorId, actors) {
    var res = new Object();
    res.actId=actorId; res.actors=[];
    for(var i=0; i < actors.length; i++)
    {
        var j=0;
        while(j<res.actors.length && actors[i]!==res.actors[j]) {j++;}
        if(j==res.actors.length)res.actors.push(actors[i]);
    }
    return res;
}
```

## Exercices

Écrire en pseudo-code les fonctions Map/Reduce permettant de :

- 1 Afficher les titres des films du genre "drama" par année

```
var mapDramaYear = function(){
    if (this.genre = "drama") {emit (this.year, this.title);}
}

var reduceDramaYear = function(year, Films) {return films;}
```

- 2 Afficher pour chaque acteur la liste des acteurs avec lesquels il a tourné des films

```
var mapAA = function(){
    for (var i=0; i<this.actors.length; i++){
        for (var j=0; j<this.actors.length; j++) {
            if(this.actors[i] != this.actors[j]){
                emit(this.actors[i]._id, this.actors[j]._id);
            }
        }
    }
}

var reduceAA = function(actorId, actors) {
    var res = new Object();
    res.actId=actorId; res.actors=[];
    for(var i=0; i < actors.length; i++)
    {
        var j=0;
        while(j<res.actors.length && actors[i] != res.actors[j]) {j++;}
        if(j==res.actors.length)res.actors.push(actors[i]);
    }
    return res;
}
```

## Options MongoDB : output du map/reduce

MongoDB propose plusieurs options pour l'exécution d'un traitement MapReduce.

La plus utile (présente dans tous les systèmes) consiste à prendre en entrée le résultat d'une requête.

- L'objet query est passé en troisième position de l'invocation du framework. Exemple :  
`db.movies.mapReduce(mapRealisateur, reduceRealisateur, {out: {"inline": 1}, query: {"country": "USA"}})`
- Un autre exemple : <http://docs.mongodb.org/manual/core/map-reduce/>

Une autre possibilité intéressante est de mettre le résultat d'un Map/Reduce dans une nouvelle collection. Exemple :

```
db.movies.mapReduce(mapRealisateur, reduceRealisateur,  
{out : "mrResults"})
```

## MapReduce et Jointure

- MapReduce est un mécanisme de base qui peut être utilisé pour implanter des opérateurs de plus haut niveau.
- Implantation de la jointure avec MapReduce : pas très élégante, mais permet la parallélisation du calcul dans des systèmes conçus pour gérer des données massives.
- Elle est aussi représentative de la transposition en MapReduce de traitements plus sophistiqués.
- Exemple : notre base contient les films avec références à des documents d'artistes.
- Le but est de partir des films avec une référence vers l'artiste / réalisateur, et d'obtenir pour chaque film le document contenant toutes les données de son réalisateur

## MapReduce et Jointure

### Collection moviesRef

```
{  
    "_id": "movie:1"  
    "title": "Pulp fiction",  
    "year": "1994",  
    "genre": "dark comedy",  
    "director": "artist:37",  
    "actors": [{"_id": "artist:11", "role": "Vincent Vega"},  
               {"_id": "artist:27", "role": "Butch Coolidge"},  
               {"_id": "artist:37", "role": "Jimmy Dimmick"}]  
}
```

...

### Collection artists

```
{"_id": "artist:11", "f_name": "John", "l_name": "Travolta"}
```

```
{"_id": "artist:27", "f_name": "Bruce", "l_name": "Willis"}
```

```
{"_id": "artist:37", "f_name": "Quentin", "l_name": "Tarantino"}
```

...

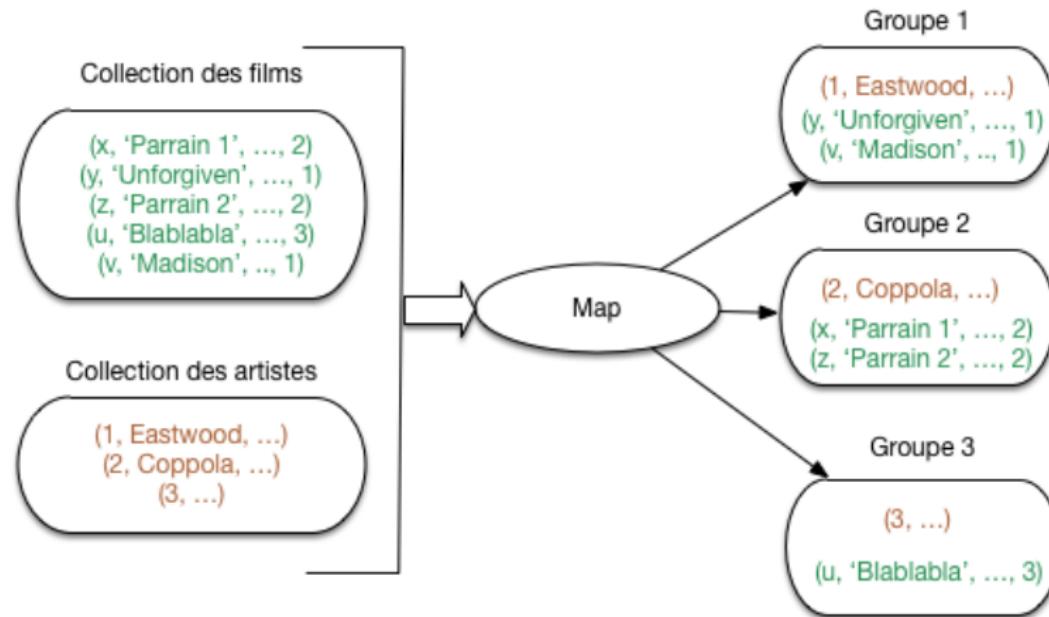
## MapReduce et Jointure

- Le MapReduce de MongoDB ne prend qu'une seule collection en entrée (raison : ça viole le paradigme de la localité des données?).
- Première étape : commencer par copier les données dans une collection commune, nommée jointure :

```
mongoimport -d moviesref -c jointure --file movies-refs.json --jsonArray  
mongoimport -d moviesref -c jointure --file artists.json --jsonArray
```

- Principe (général) : on exploite le mécanisme de regroupement pour associer, dans un même groupe les informations à combiner.
- On va créer autant de groupes que d'artistes. Dans chaque groupe on place : l'artiste dont l'identifiant correspond à la clé du groupe + les films dont l'identifiant du metteur en scène correspond à la clé du groupe.
- Méthode très représentative de l'application de Map/Reduce à des algorithmes complexes.

## MapReduce et Jointure



## MapReduce et Jointure

```
var mapJoin = function() {
    // Est-ce que l'id du document contient le mot "artist"?
    if (this._id.indexOf("artist") != -1) {
        // Oui ! C'est un artiste. Ajoutons-lui son type.
        this.type="artist";
        // On produit une paire avec pour id celle de l'artiste
        emit(this._id, this);
    }
    else {
        // Non: c'est un film. Ajoutons-lui son type.
        this.type="film";
        // Simplifions un peu le document pour l'affichage
        delete this.summary;
        delete this.actors;
        // On produit une paire avec pour id celle du metteur en sc.
        emit(this.director._id, this);
    }
};
```

## MapReduce et Jointure

```
var reduceJoin = function(id, items) {  
  
    var director = null, films={result: []}  
  
    // On cherche l'artiste dans cette liste  
    for (var idx = 0; idx < items.length; idx++) {  
        if (items[idx].type=="artist") {  
            director = items[idx];  
        }  
    }  
  
    // Maintenant, 'director' contient l'artiste : on l'affecte aux films  
    for (var idx = 0; idx < items.length; idx++) {  
        if (items[idx].type=="film" && director != null) {  
            items[idx].director = director;  
            films.result.push (items[idx]);  
        }  
    }  
    return films;  
};
```

## Clés composées

- Il est possible que la clé du map soit composée
- Exemple 1 : "Les films par genre et par année"

```
• var mapGenreAnnee = function() {  
    emit({genre: this.genre, annee : this.year}, 1);  
}  
• var reduceGenreAnnee = function(id, values) {  
    var sum = 0; values.forEach(function(value){sum += value;}); return  
    {count: sum};  
}
```

## Clés composées

- Exemple 2 : "Le graphes des co-artistes"

- ```
db.movies.updateMany({}, {$push: {actors: {$each: [], $sort: {"_id": 1}}}})
```
- ```
var mapCoArt = function() {if (this.actors!=undefined) {for(var i =0; i<this.actors.length-1; i++) {for(var j=i+1; j<this.actors.length; j++) {if(this.actors[i]._id!=this.actors[j]._id) {emit({art1 : this.actors[i]._id, art2 : this.actors[j]._id}, {count : 1});}}}}}}
```
- ```
var reduceCoArt = function(id, values) {var sum = 0; values.forEach(function(value) {sum += value.count;}); return {count: sum};}
```
- Remarque : Attention, tout n'est pas Map/Reduce. Exemple : "Trouver le nombre de films du genre drama produits en 2017". Généralement Map/Reduce si la requête équivalente en SQL s'exprime en GROUP BY (mais pas que)

## Cles composées

- Exemple 2 : "Le graphes des co-artistes"

- ```
db.movies.updateMany({}, {$push:{actors:{$each:[],$sort:{"_id":1}}}})
```
- ```
var mapCoArt = function(){if (this.actors!=undefined){for(var i =0; i<this.actors.length-1; i++){for(var j=i+1; j<this.actors.length; j++){if(this.actors[i]._id!=this.actors[j]._id){emit({art1 : this.actors[i]._id, art2 : this.actors[j]._id},{count : 1});}}}}}
```
- ```
var reduceCoArt = function(id, values){var sum = 0; values.forEach(function(value){sum += value.count;}); return {count: sum};}
```
- Remarque : Attention, tout n'est pas Map/Reduce. Exemple : "Trouver le nombre de films du genre drama produits en 2017". Généralement Map/Reduce si la requête équivalente en SQL s'exprime en GROUP BY (mais pas que)

## Chapitre 4 - Cloud et données massives

- Introduction
- Scalabilité
- Infrastructure Cloud
- Performances Cloud
- Systèmes répartis (distribués)
- Le NoSQL

# Données massives et infrastructures Cloud

## Données massives (*BigData*) : Qu'est-ce que c'est?

- Les données sont massives quand elles dépassent les capacités d'une seule machine.
- En mémoire RAM : quelques dizaines de GOs ;
- En mémoire persistante : quelques TOs.

Qui dit **données massives** dit donc **système distribué**. Point commun (le seul?) des systèmes dits NoSQL.

**Système distribué** : de fait, l'infrastructure est constituée de **grappes de serveurs**, avec

- des machines à bas coût (*commodity servers*),
  - la possibilité d'en ajouter/retirer en quelques minutes (**élasticité**),
  - des environnements logiciels spécialisés (hyperviseur de virtualisation, NoSQL, etc.).
- ◆ Nous appellerons l'ensemble un *Cloud* (très réducteur).

Cloud : envisagé dans la suite comme une nouvelle machine de calcul à part entière (très réducteur), élastique et scalable, apte à prendre en charge des données massives.

# Données massives, infrastructures cloud et NoSQL

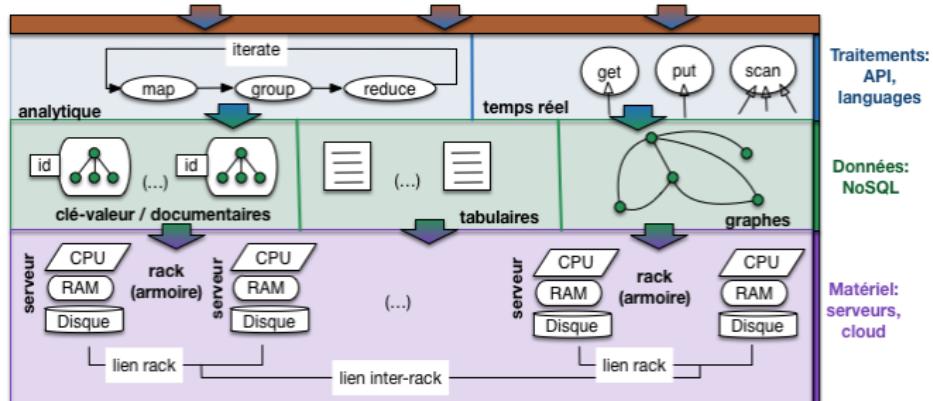
## Vision globale : 3 couches

- les applications clientes utilisent des frameworks de traitement distribué et parallèle (Ex. MapReduce)
- les données sont stockées selon des modèles adaptés au partitionnement/parallelisation (ex. documentaire, clé/valeur, etc.). Le modèle relationnel ne l'est pas, pourquoi?
- le stockage et le traitement se font sur une infrastructure matérielle **élastique** et **scalable**

Architecture en couches = abstraction.

⇒ Chaque couche prend en charge ses propres problèmes techniques

*Application données massives: analytiques, jeux vidéos, gestion documentaire, applications web, ...*



## Chapitre 4 - Cloud et données massives

- Introduction
- **Scalabilité**
- Infrastructure Cloud
- Performances Cloud
- Systèmes répartis (distribués)
- Le NoSQL

## Scalabilité horizontale et verticale

### Scalabilité (Scalability)

Capacité d'un système à **s'adapter aux montées en charge** ou au passage à l'échelle (*i.e.* changement d'ordre de grandeur des demandes de traitement).

"S'adapter" : **maintenir** ses fonctionnalités et **ses performances**.

La **scalabilité** peut être **horizontale** ou **verticale**

**Scalabilité horizontale** : augmentation de la puissance du système par ajout de machines à bas coût (typique dans les environnements cloud).

**Scalabilité verticale** : augmentation de la puissance du système par l'augmentation des capacités d'une machine.

## "Performance" : débit et latence

### Système scalable

Un système est scalable si **performances** sont **proportionnelles aux ressources** qui lui sont allouées

Exemple : si le parcours d'une collection sur 1 serveur prend  $n$  secondes, on devrait pouvoir parcourir les partitions réparties sur 2 serveurs de cette collection en  $\frac{n}{2}$  secondes

Dans la pratique c'est quasiment impossible, mais c'est à ça qu'on devrait tendre.

Performance : terme "fourre-tout". On peut néanmoins identifier 2 métriques clés.

### Débit

- Désigne le nombre d'unités d'information (ex. documents) qu'il est possible de traiter par unité de temps (ex. temps de parcours d'une collection).
- On mesure souvent le nombre d'octets par seconde (par exemple, 10 MO/s) et non par exemple le nombre de documents/s.

### Latence

- Temps mis pour accéder à une unité d'information (ex. document dans notre cas) en lecture et/ou en écriture.

## Quelques conditions nécessaires

Quelques conditions favorables (voire indispensables) pour atteindre la **scalabilité** dans un **environnement distribué**

- Données équitablement réparties.
- **Data Locality** : traitement essentiellement local (pourquoi?).
- Et un temps de calcul global qui reste raisonnable pour les ressources qu'on peut/veut lui allouer.

## Chapitre 4 - Cloud et données massives

- Introduction
- Scalabilité
- **Infrastructure Cloud**
  - Baie de serveurs
  - Baie de stockage
  - DataCenter
- Performances Cloud
- Systèmes répartis (distribués)
- Le NoSQL

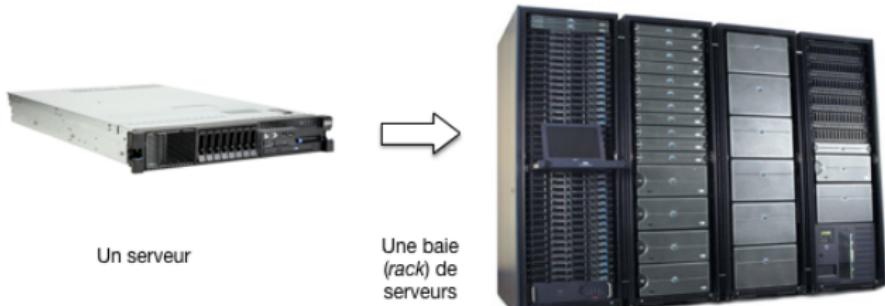
## Infrastructures Cloud : Baie de serveurs

Dans un centre de données, les serveurs sont empilés dans des baies (armoire ou Rack) qui leur fournissent l'alimentation, la connexion réseau, la ventilation, etc..<sup>3</sup>

Ordres de grandeur : une baie contient environ 40 serveurs, un centre de données quelques centaines de baies.

Les entreprises comme Google et Amazon ont depuis quelques années dépassé le cap du million de serveurs

Les serveurs sont généralement de **qualité moyenne**, voire médiocre, ce qui permet d'en ajouter facilement à la demande (**élasticité**), mais implique également **des pannes tout le temps!**



<sup>3</sup>On rencontre soit des serveurs Rack soit des serveurs Blade (serveur lame ou carte serveur) qui sont d'un encombrement moindre

## Infrastructures Cloud : Baie de stockage

Baie de stockage (*disk array*) : serveur spécialisé fournissant des espaces sécurisés de stockage aux serveurs d'applications

Peut contenir des dizaines, voire des centaines, de disques et mémoires SSD. Il existe des baies 100% SSD (EMC XtremIO)

Utilise généralement le RAID et une suite logicielle permettant la protection des données (réPLICATION LOCALE ET DISTANTE, archivage, snapshot, etc.)

Utilise généralement un OS embarqué

Intérêt : élasticité (ajout de disques à la demande), plus de robustesse (alimentation redondante, chemin redondant, etc.), de disponibilité, d'économie, etc.



# Systèmes de stockage : DAS, NAS et SAN

## DAS (*Direct Attached Storage*)

- Disques (ou baie) non partagés et attachés directement au système (sans passer par une connexion réseau). Le disque interne est la forme de DAS la plus simple.
- Avantage : peu onéreux et extrêmement performant. Inconvénient : scalabilité (verticale) limitée, pas de mutualisation de l'espace disque non utilisé, administration difficile, etc.

## NAS (*Network Attached Storage*)

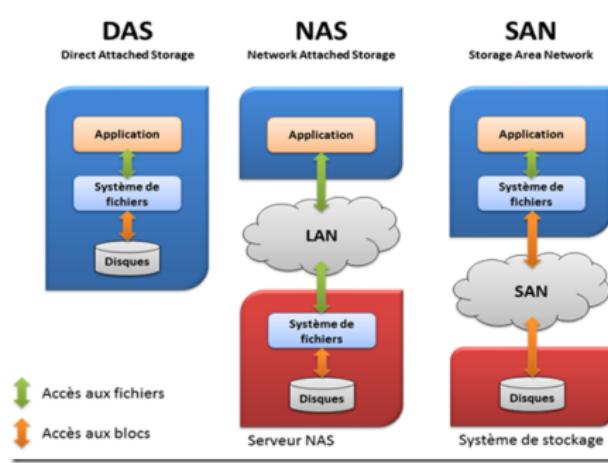
- les baies de stockage apparaissent comme des volumes partagés sur le réseau IP. Les volumes sont accessibles au niveau fichier (pas d'accès bloc), presque comme un serveur de fichiers.
- Avantage : mutualisation, administration simplifiée (RAID et sauvegarde centralisés), simple à déployer et peu onéreux. Inconvénient : ne peut envoyer que des fichiers (et non des blocs de données), ce qui peut engendrer des congestions du réseau. Peu performant.

## SAN (*Storage Area Network*)

- Les baies de stockage sont partagées en réseau, mais cette fois ce réseau leur est dédié (en parallèle du LAN).
- Utilise généralement le protocole Fibre Channel permettant d'atteindre des débits de l'ordre de quelques GO/s
- Les volumes sont directement accessibles en mode bloc depuis les systèmes de fichiers des serveurs. Un serveur "voit" donc l'espace de stockage comme son propre disque dur.
- Avantage : meilleures performances que le NAS. Inconvénient : déploiement complexe, onéreux.



## Données massives et systèmes de stockage



Dans un cloud orienté données massives, le DAS est généralement préféré aux baies de stockage séparés (NAS et SAN). Pourquoi?

Un point de vue intéressant :

Big Data Design Considerations, Hadoop and Storage Choices

<https://blogs.vmware.com/vsphere/2014/08/big-data-design-considerations-hadoop-storage-choices.html>

## Infrastructures Cloud : Data Center

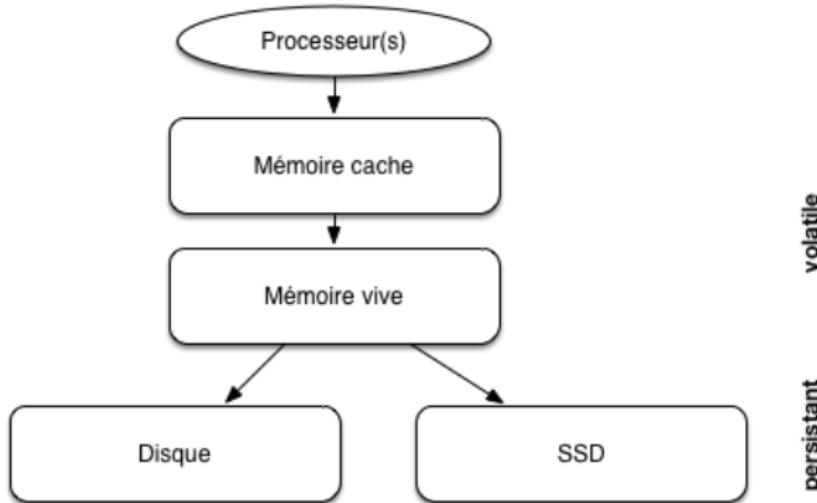
- Les baies sont alignés dans de grands hangars (*i.e. data centers*)
- Les baies sont connectées les unes aux autres grâce à des routeurs
- La grappe de serveurs est elle-même connectée à l'Internet par un troisième niveau de connexion (après ceux intra-baie, et inter-baies).
- On obtient une **connectique hiérarchique** qui joue un rôle dans la gestion des données massives.



## Chapitre 4 - Cloud et données massives

- Introduction
- Scalabilité
- Infrastructure Cloud
- **Performances Cloud**
  - Virtualisation
- Systèmes répartis (distribués)
- Le NoSQL

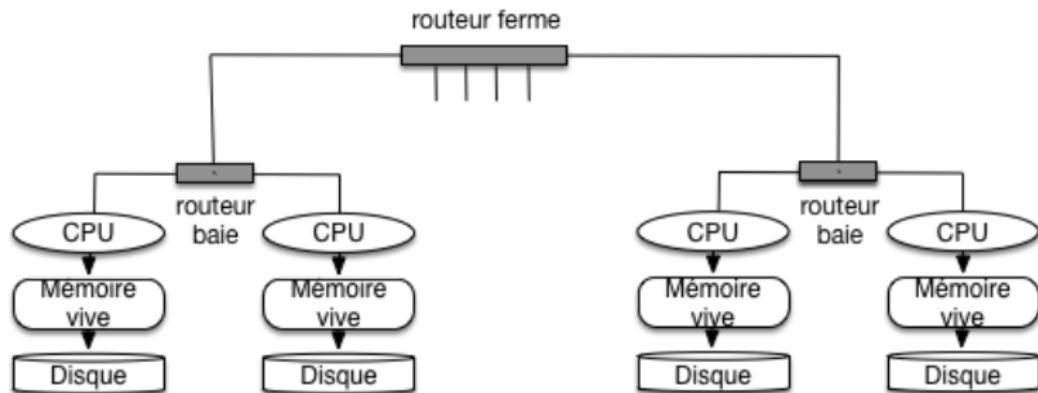
## Hiérarchie des mémoires en centralisé



Classique : un SGBD place en mémoire les données les plus utilisées

## Hierarchie des mémoires dans le cloud

Différence essentielle : le réseau (hiérarchique)



## Performances : RAM vs. Disque Dur Magnétique vs. SSD

Estimations (ce sont des ordres de grandeur).

Type	Taille	Latence	Débit
RAM	O(10 GOs)	$\approx 10^{-8}$ s (10 nanosec.) ;	$\approx 1 GO/s$
SSD	O(100 GOs)	$\approx 10^{-4}$ s (0,1 millisec.) ;	Qq GO/s
Disque	O(1 TOs)	$\approx 10^{-3}$ s (10 millisec.) ;	100 MO/s

Pour le réseau :

- compter 1 Gbits/s (intra)-baie ;
- diviser par 5 ou 10 pour le débit inter-baies.

### Retenir

- Accès (aléatoire) au disque : très lent par rapport à un accès en RAM.
- Débit du disque : faible (100 MO/s au mieux)
- SSD : 100 fois plus rapide qu'un disque (latence), 10 fois plus en débit

Autres estimations importantes, pour 1GO : disque  $\approx 0.04\$/s$ , SSD  $\approx 0.4\$/s$ , RAM  $\approx 7\$/s$  (début 2015)

## Performances dans le Cloud : importance du réseau

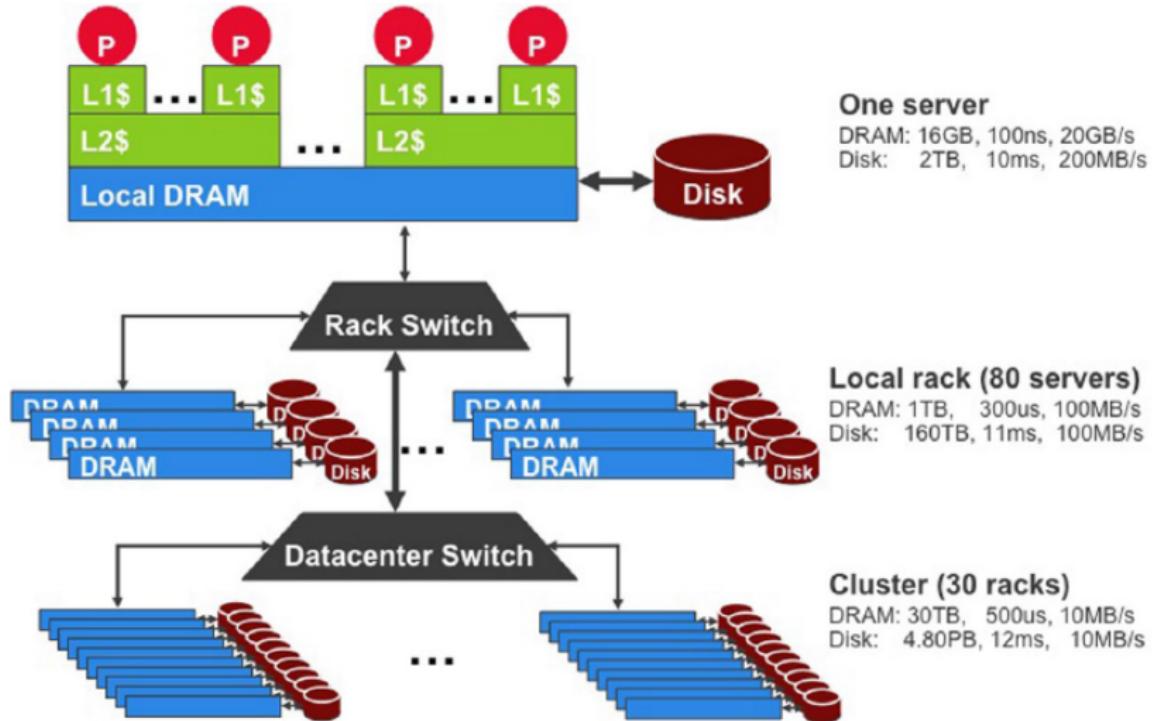


Figure: Storage hierarchy ("The Datacenter as a Computer", L.A. Barroso, U. Holzle, Google Inc, Morgan & Claypool Publishers, 2009)

## Performances dans le Cloud : importance du réseau

### Localité des données (*data locality*)

Toujours tenter de traiter les données **localement**, par le CPU du serveur stockant les données.

- Pas toujours possible, mais *best effort* pour des systèmes comme Hadoop.
- **Implique un déplacement des traitements vers les données** (en analytique)

### Essentiel

- Systèmes temps réel : **placer les données en RAM** (latence reste faible).
- Systèmes analytiques : **placer les données séquentiellement sur le disque** (évite la latence du disque).

### Pour aller plus loin...

MongoDB utilise les fichiers mappés en mémoire centrale.

Voir [http://en.wikipedia.org/wiki/Memory-mapped\\_file](http://en.wikipedia.org/wiki/Memory-mapped_file) et  
<http://docs.mongodb.org/manual/faq/storage/>.

## Infrastructures Cloud : Virtualisation

### Constats

- ➊ Hormis quelques pics de charges sporadiques, la plupart des serveurs fonctionnent en deçà de ce que leurs capacités matérielles ne le permettent
- ➋ Un serveur fonctionnant à pleine charge consomme quasiment autant d'énergie qu'un serveur fonctionnant avec une charge moindre

Idée de base : faire cohabiter plusieurs systèmes (machines virtuelles) sur une même machine physique (de préférence, si leurs pics de charges ne coïncident pas)

### Intérêts

- Economie sur le matériel par mutualisation (consommation électrique, entretien physique, surveillance, support, compatibilité matérielle, etc.)
- Allocation dynamique des ressources matérielles en fonction des besoins de chaque machine virtuelle à un instant donné,
- Disponibilité accrue : si une machine physique plante, ses machines virtuelles sont déplacées vers d'autres machines
- Dimensionnement plus simple, Load Balancing, etc.

## Infrastructures Cloud : Virtualisation

### Hyperviseur

- logiciel gérant les machines virtuelles : allocation de ressources, création d'environnements clos et indépendants, déplacement des machines virtuelles d'une machine physique à une autre (pour le load balancing ou la reprise sur panne instantanée), etc.
- Exemples : Microsoft Hyper-V, VMWare vSphere et vCloud, Citrix Xen, etc.

### Domaines d'application (en somme, les services Cloud!)

- Bureau virtuel (Virtual Desktop Infrastructure ou Desktop as a Service) : dématérialisation de l'environnement de travail (le "bureau") du terminal sur lequel il va s'afficher.
- Software as a Service : mise à disposition de logiciels. Exemple : le compilateur L<sup>A</sup>T<sub>E</sub>X Overleaf (<https://www.overleaf.com/>).
- Infrastructure as a Service : mise à disposition d'un environnement d'exécution, en laissant au client la maîtrise des applications qu'il peut installer, configurer et utiliser.

Exemple : Amazon Elastic Compute Cloud (EC2) fournit une interface web par laquelle un client peut créer des machines virtuelles sur lesquelles il peut charger n'importe quel logiciel et paye en fonction du temps d'usage des serveurs (Pay Per Use)

⇒ La virtualisation est un élément clé des infrastructures Cloud

## Chapitre 4 - Cloud et données massives

- Introduction
- Scalabilité
- Infrastructure Cloud
- Performances Cloud
- Systèmes répartis (distribués)
  - Maître-Esclave
  - Master-Master
  - Failover (reprise sur panne)
- Le NoSQL

## Systèmes répartis (distribués) : définition

Système distribué = ensemble de composants logiciels, ou **nœuds**, répartis sur une **grappe de serveurs**, coopérant pour une tâche précise.

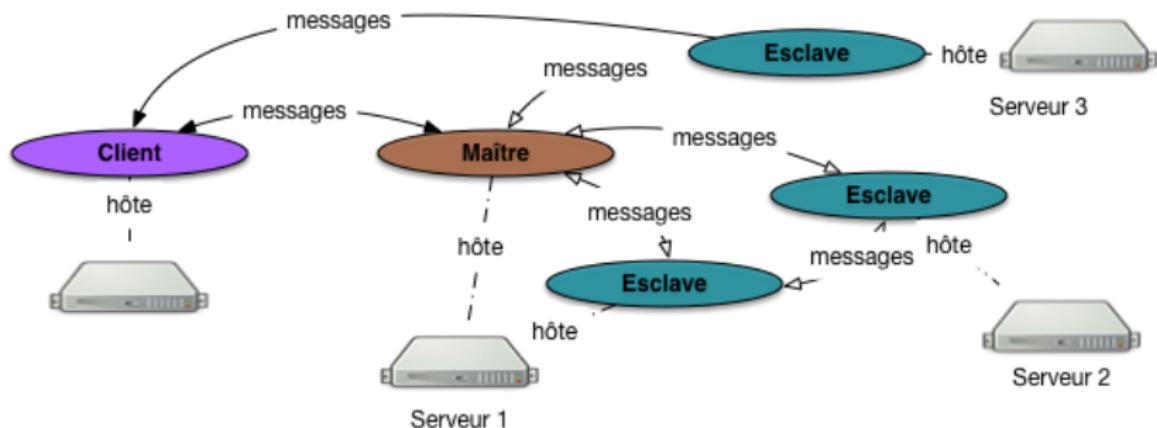
Ces nœuds communiquent par **envoi de messages** (aucun partage de mémoire).

Chaque nœud est en écoute sur un port réseau (ex., le 80 pour le Web, 27017 pour Mongodb).

On peut avoir un système distribué (mais pas scalable) sur une seule machine !  
En général, un nœud par machine.

## Maître-Esclave

Deux organisations principales. La première (plus courante) est dite **maître-esclave**.



Rôle essentiel du maître (mais aussi *single point of failure*).

## Maître-Esclave

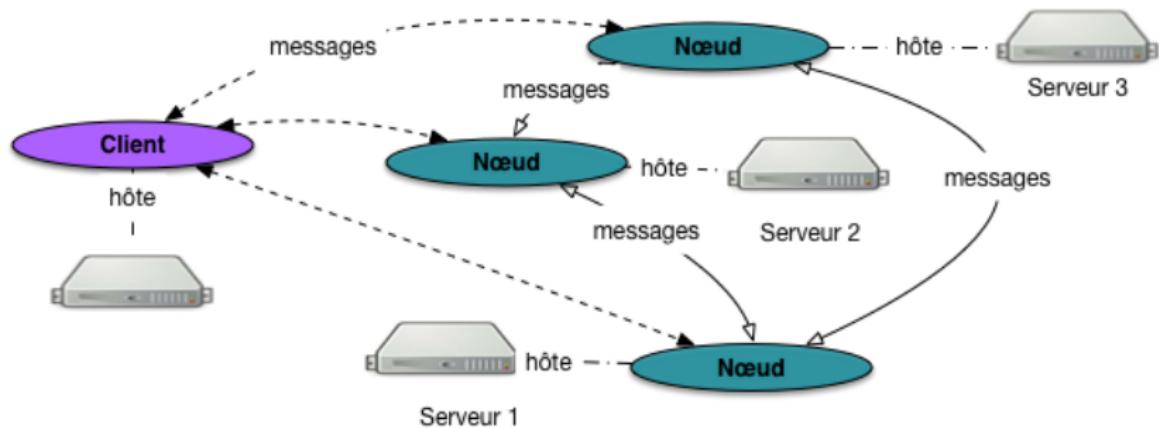
Le maître est notamment chargé des tâches administratives du système

- ajouter un noeud, en supprimer un autre,
- surveiller la cohérence et la disponibilité du système,
- appliquer une méthode de reprise sur panne le cas échéant.

Souvent chargé aussi de communiquer avec l'application cliente (qui constitue un troisième type de noeud)

## Master-Master

C'est la démocratie ! Il n'y a que des maîtres (*master-master*, ou aussi multi-nœuds).



Plus robuste en théorie, mais soulève des problèmes compliqués (donc, moins courant)

## Failover (reprise sur panne)

Un système distribué peut s'appuyer des centaines, des milliers de serveurs : **il y a des pannes tout le temps.**

Caractéristique d'un bon système : tolérants aux pannes, assure une disponibilité 24/7.

- surveillance automatisée de tous les nœuds participants (*heartbeat*, messages toutes les  $n$  secondes).
- **méthode de reprise sur panne (failover)** basée sur la redondance/réPLICATION.

Un cas particulièrement épineux : le **partitionnement réseau**.

## Chapitre 4 - Cloud et données massives

- Introduction
- Scalabilité
- Infrastructure Cloud
- Performances Cloud
- Systèmes répartis (distribués)
- **Le NoSQL**

# Le NoSQL, qu'est ce que c'est?

Tout et n'importe quoi, mais quelques points communs!

## Systèmes NoSQL

Un système NoSQL est un **système distribué** dont la tâche est la gestion de **données persistantes** dans un **environnement cloud**. Il fournit les fonctionnalités suivantes :

- Recherche et mise à jour de données
- Adaptation automatique aux ressources matérielles : équilibrage, élasticité.
- Performances scalables par distribution/parallelisation.
- Reprise sur panne automatisée par réPLICATION.

Beaucoup de modèles différents ("documents", ou simplement (clé, valeur), graphes ...). Deux types principaux :

- Systèmes temps réel : accès en millisecondes aux unités d'information (généralement, en mémoire vive).
- Systèmes analytiques : traitements appliqués à tout ou partie d'une collection.

## Petit historique : NoSQL analytique

Quelques publications de Google : *Google File System* (2003), MapReduce (2004), BigTable (2006).

Clonés par des systèmes Open Source dans l'environnement Hadoop : HDFS, MapReduce, HBase.

Autres successeurs : Cassandra, maintenant Spark, Flink, etc.

Applications : construction d'index, analyses de données massives.

## Petit historique : NoSQL temps réel

Publication majeure d'Amazon : le système Dynamo (2007).

Application : gestion robuste, distribuée, scalable de centaines de millions de comptes client, produits, transactions.

Cloné par le système Voldemort ; principes repris par Mongodb, CouchDB, Riak, beaucoup d'autres

NB : précédé par beaucoup de recherche et de systèmes distribués bien sûr.

## Ce qu'il faut retenir

Environnement dit *cloud* = grappes de serveurs à bas coût, stockés dans des centres de données.

Très "élastique" car facile d'ajouter/supprimer une ressource = **scalabilité horizontale**.

Très sujet aux **pannes**, de disque, de réseau, de logiciels.

Système distribué (NoSQL) : système exploitant au mieux les ressources du *cloud*, tolérant aux pannes.

## Chapitre 5 - Systèmes NoSQL : la réPLICATION

- La réPLICATION pourquo?
- La réPLICATION, comment?
- Cohérence des données
- Reprise sur panne (failover)
- RéPLICATION et failover sous MongoDB

## Répliquer, à quoi ça sert?

Outil indispensable, universel pour la robustesse des systèmes distribués.

- **Tolérance aux pannes**

Vous cherchez un document sur  $S_1$ , qui est en panne ? On le trouvera sur  $S_2$

- **Distribution des lectures**

Répartissons les lectures sur  $S_1, S_2, \dots, S_n$  pour satisfaire les millions de requêtes de nos clients.

- **Distribution des écritures ?**

Oui, mais attention, il faut **réconcilier** les données ensuite.

- et autres avantages, comme la construction d'un index sur un des serveurs sans affecter les autres,

Le bon niveau de réPLICATION ? **Trois copies pour une sécurité totale ; deux au minimum.**

## Chapitre 5 - Systèmes NoSQL : la réPLICATION

- La réPLICATION pourquoi?
- **La réPLICATION, comment?**
- Cohérence des données
- Reprise sur panne (failover)
- RéPLICATION et failover sous MongoDB

## Méthode générale de réPLICATION

Une **application** (le client) demande au **système** (le serveur)  $S_1$  **l'écriture** d'un document (unité d'information).

- le serveur écrit le document sur le disque ;
- $S_1$  **transmet la demande d'écriture à un ou plusieurs autres serveurs**  $S_2, \dots, S_n$ , créant des **replicas** ou **copies** ;
- $S_1$  "rend la main" au client.

Essentiel, réPLICATION synchrone/asynchrone ?

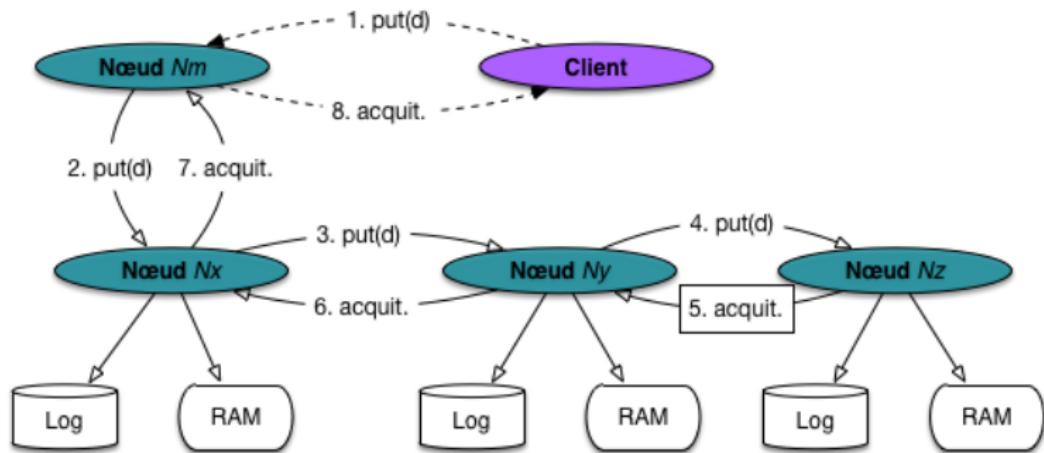
- **synchrone** :  $S_1$  **attend** confirmation de  $S_2, \dots, S_n$  avant de rendre la main ;
- **asynchrone** :  $S_1$  rend la main **sans attendre**.

### Conséquences

Une réPLICATION **asynchrone** est beaucoup plus rapide, mais elle favorise les **incohérences**, au moins temporaires.

## RéPLICATION avec écritures synchrones

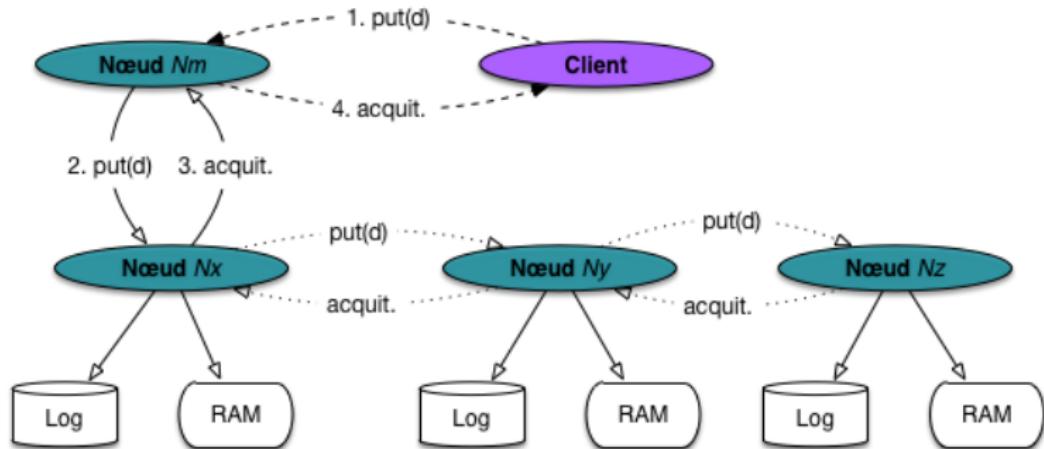
Le client est acquitté quand **tous** les serveurs ont effectué l'écriture.



Sécurité totale ; cohérence forte ; **très lent**.

## RéPLICATION avec écritures asynchrones

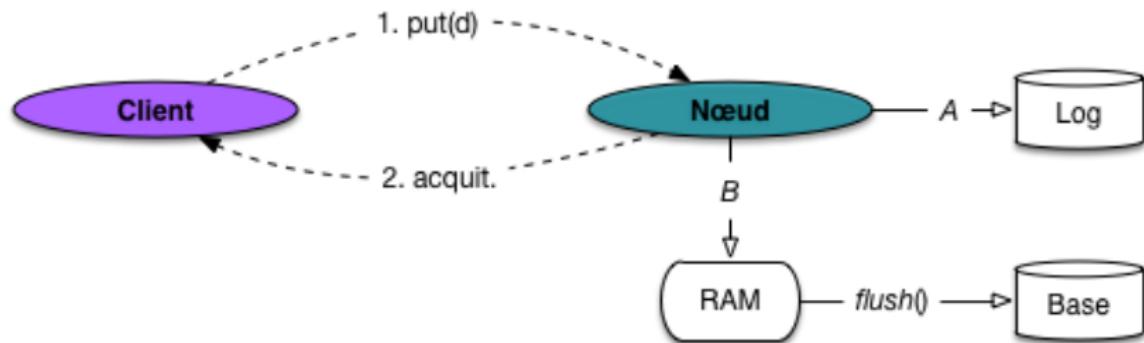
Le client est acquitté quand **un** des serveurs a effectué l'écriture. Les autres écritures se font indépendamment.



Sécurité partielle : cohérence faible : **Efficace.**

## Pour éviter la latence disque : le fichier journal

Technique universellement utilisée en centralisé



Permet de se ramener à des entrées/sorties **séquentielles**.

## Chapitre 5 - Systèmes NoSQL : la réPLICATION

- La réPLICATION pourquoi?
- La réPLICATION, comment?
- **Cohérence des données**
- Reprise sur panne (failover)
- RéPLICATION et failover sous MongoDB

## Parlons cohérence

### Cohérence forte (au sens large)

A tout instant, il ne peut pas y avoir deux valeurs ou **valeurs différentes d'une même donnée**

Pour garantir la cohérence, un système doit refléter fidèlement les opérations faites sur les données

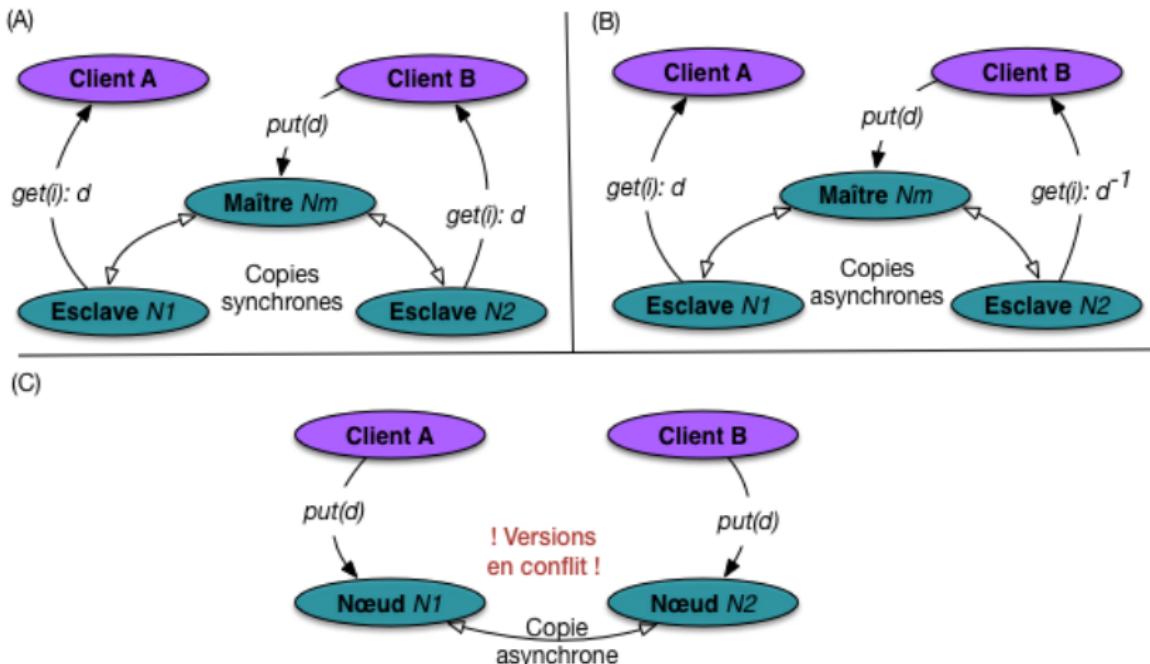
⇒ **Toute opération** (validée) est **permanente** et **immédiatement visible**.

**Cohérence forte** = une des fonctionnalités marquantes des **systèmes relationnels**.

### Dans les systèmes NoSQL

La cohérence forte est sacrifiée au profit des performances

## Architectures avec réPLICATION : topologie et (a)synchronicité



## Soyons cohérents (ou pas...)

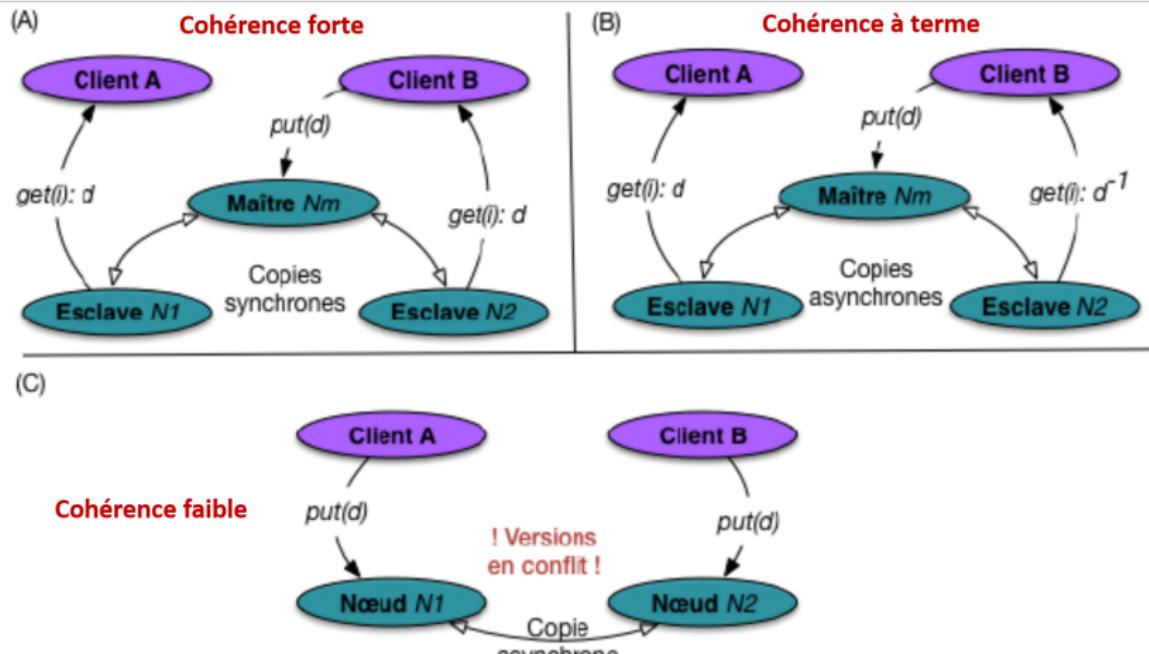
Plusieurs niveaux de cohérence dans les systèmes distribués / NoSQL.

- **Cohérence forte** (ACID) – nécessite une réPLICATION synchrone, et potentiellement des verrouillages complexes (*two phases commit*).
- **Cohérence faible** – on accepte le risque de lectures ne reflétant pas les mises à jour.
- **Cohérence à terme** (*Eventual consistency*) – le système garantit que les incohérences ne sont que transitoires.

### Remarque

Les systèmes NoSQL dans l'ensemble abandonnent la cohérence forte pour favoriser la réPLICATION asynchrone, et donc le débit en écriture/lecture.

## Soyons cohérents (ou pas...)

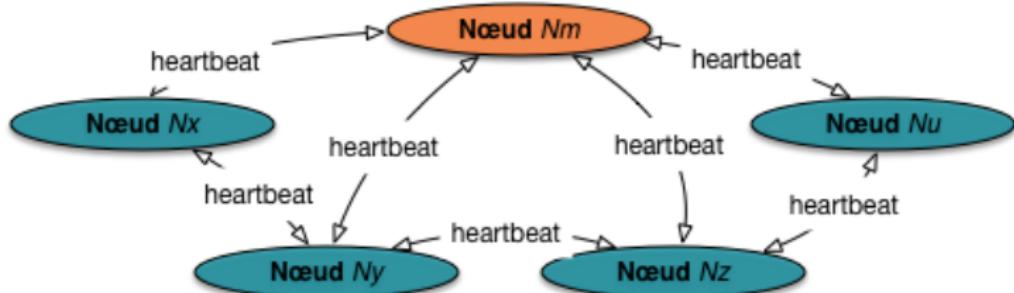


## Chapitre 5 - Systèmes NoSQL : la réPLICATION

- La réPLICATION pourquoi?
- La réPLICATION, comment?
- Cohérence des données
- **Reprise sur panne (failover)**
- RéPLICATION et failover sous MongoDB

## RéPLICATION et reprise sur panne (failover)

Principe général : tout le monde est interconnecté et échange des messages (*heartbeat*).



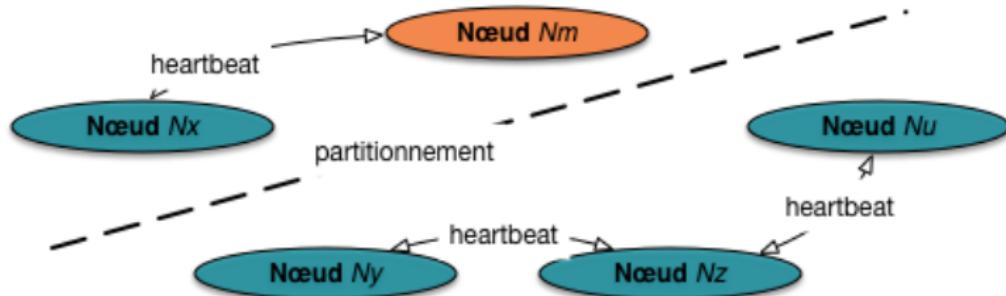
Si un **esclave** tombe en panne, le système continue à fonctionner, **tant qu'il est en contact avec une majorité d'esclaves**

Les lectures sont dirigées vers les autres noeuds et un nouvelle réPLICATION vers un autre serveur esclave est initiée.

Si le maître tombe en panne, les slaves doivent **élire** un nouveau maître

## Cas du partitionnement réseau

Un **partitionnement** divise la grappe en deux groupes.



### Principe de majorité

Seul le groupe représentant la **majorité** des participants initiaux peut élire un maître.

Election : algorithme de négociation, voir par exemple Paxos.

## Culture, culture : le théorème CAP

Un "théorème" qui dit qu'un système distribué ne peut satisfaire à la fois

- ① La cohérence (**C**)
- ② La disponibilité (**Availability**)
- ③ La tolérance au **Partitionnement**

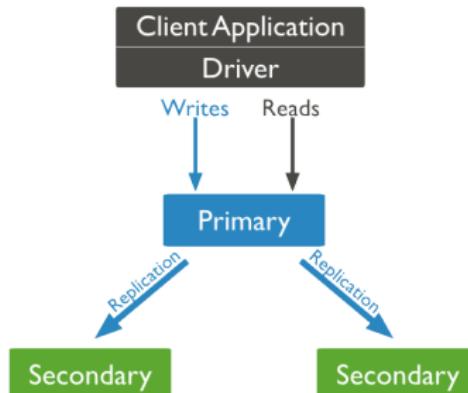
Exprime une limitation intrinsèque aux systèmes distribués, applicable à tout système NoSQL, et expliquant qu'il faut toujours faire un compromis.

## Chapitre 5 - Systèmes NoSQL : la réPLICATION

- La réPLICATION pourquoi?
- La réPLICATION, comment?
- Cohérence des données
- Reprise sur panne (failover)
- RéPLICATION et failover sous MongoDB
  - MongoDB : un système réparti maître-esclaves
  - Replica Set, Primary, Secondaries
  - Cohérence forte vs. cohérence à terme
  - Reprise sur panne (failover) dans MongoDB
  - A l'action!

## MongoDB, système distribué maître-esclave

- MongoDB : Système réparti (distribué) maître-esclaves, utilisant une **réPLICATION ASYNCHRONE**
- L'ensemble de serveurs mongoDB<sup>4</sup> (grappe de serveurs) partageant des replicas d'un même ensemble de documents, est appelé **Replica Set**
- Typiquement un RS contient un maître, *i.e.* **Primary** et deux esclaves, *i.e.* **Secondaries** et éventuellement un serveur arbitre.
- Dans une grappe MongoDB avec des documents répartis, on peut trouver plusieurs replica sets, chacun contenant un sous-ensemble d'une très grande collection (nous y reviendrons)



<sup>4</sup>Instances mongod

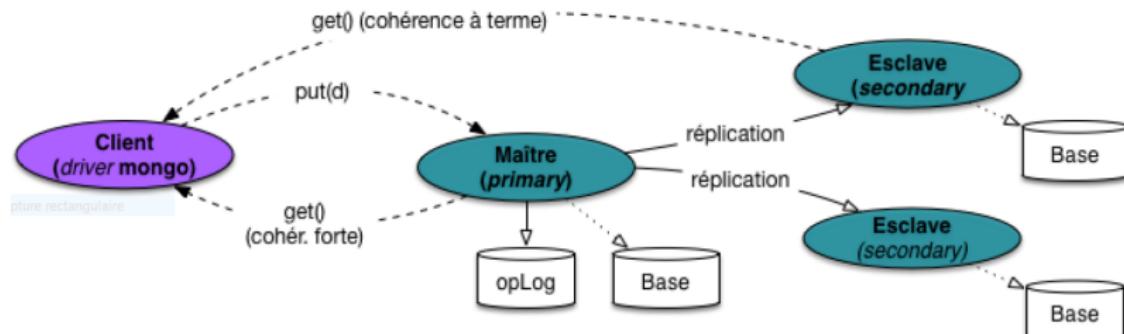
# Replica Set, Primary, Secondaries

## Primary (maître)

- Les écritures ont toujours lieu sur le Primary (il en existe un seul dans un replica set)
- Consigne toutes les opérations de modification des données dans un fichier journal (log), appelé opLog (operations log)

## Secondary (esclave)

- Récupère l'opLog (du primary ou de n'importe quel autre secondary) et le stocke localement dans la collection local.oplog.rs
- Met à jour la copie locale à partir de l'opLog récupéré
- La mise-à-jour des copies est asynchrone, i.e. on n'attend pas la confirmation de l'écriture des copies.



## Cohérence forte vs. cohérence à terme

2 niveaux de cohérence dans MongoDB : **Cohérence forte** et **cohérence à terme**.

### Cohérence forte

- Obtenu en imposant au client d'**effectuer toujours les lectures via le maître**.
- Les esclaves ne servent pas à répartir la charge
- Les esclaves jouent le rôle restreint d'une sauvegarde/réPLICATION continue, avec remplacement automatique du maître si celui-ci subit une panne

### Cohérence à terme (*Eventual Consistency*)

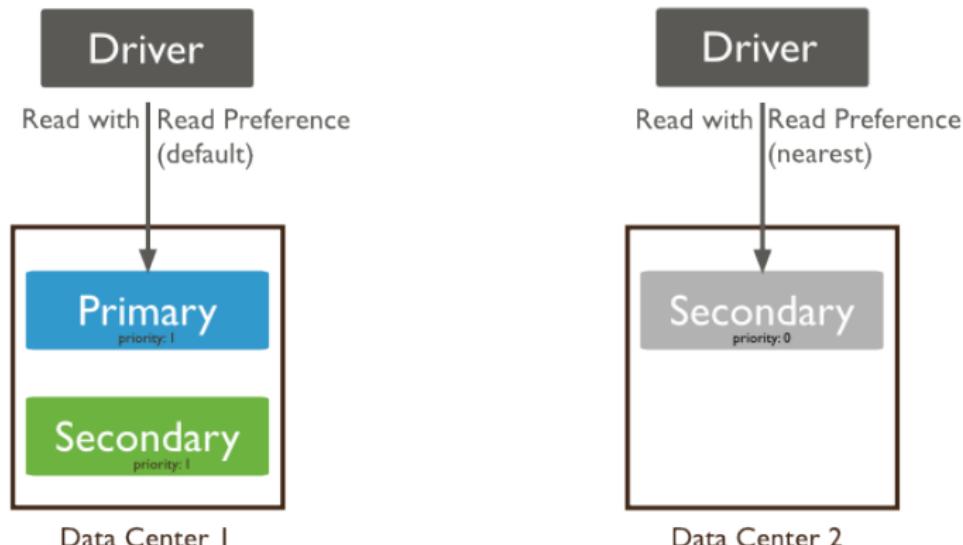
- **Le client peut choisir de lire sur un esclave**
- Plusieurs options : lire à partir du noeud ayant la plus faible latence réseau (*nearest*), sur le noeud secondaire préféré (*SecondaryPreferred*), etc.

### Remarque

La réPLICATION n'est pas, dans MongoDB, le moyen privilégié de passer à l'échelle. Le partitionnement est un mécanisme plus puissant (On y reviendra!)

# Cohérence forte vs. cohérence à terme

## Read Preference

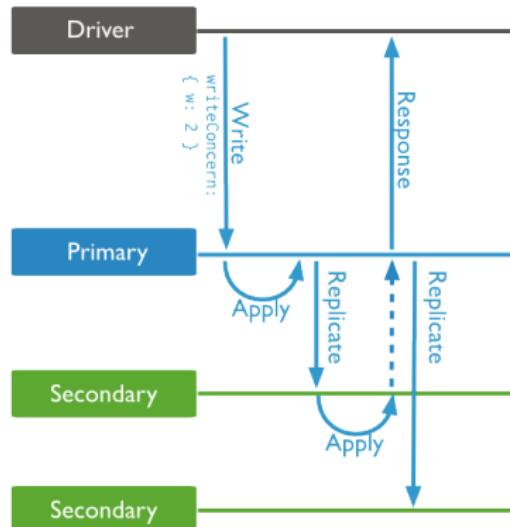


```
db.coln.find().readPref('nearest')
```

# Cohérence forte vs. cohérence à terme

## Write Concern

Une autre option : l'application cliente peut attendre que **n serveurs** ou bien une **majorité** (*majority*) de serveurs aient effectué l'écriture avant de reprendre la main



```
db.col.insert({ "_id":1, "nom": "abc"},  
             {writeConcern: {w:2, wtimeout:5000}})
```

## Election d'un maître dans MongoDB

Reprise sur panne : identique à celle décrite précédemment

- Les serveurs se surveillent par *heartbeat* (ping)
- Perte d'un esclave : redirection des éventuelles lectures et initiation d'une nouvelle réPLICATION
- Perte du maître : élection d'un nouveau maître. Tous les clients (drivers) connectés au RS sont réinitialisés pour dialoguer avec le nouveau maître.

Élection d'un nouveau maître : déclenchée

- Lors de l'initialisation d'un nouveau RS
- Si le maître perd son statut car il ne voit plus qu'une minorité d'esclaves.

Beaucoup d'options pour paramétrer le vote

- Une priorité donnée à chaque serveur
- Un Secondary ne peut devenir Primary que s'il est le plus "à jour" parmi tous les Secondaries connectés
- Un Secondary ne peut devenir Primary que s'il "voit" la majorité des secondaries connectés
- etc.

## Election d'un maître dans MongoDB

### Serveur arbitre (arbiter)

- Pour garantir qu'une élection aboutisse, il faut que le nombre de participants soit impair
- On peut ajouter un serveur arbitre pour rendre impair le nombre de participants
- Arbitre : serveur ne contenant pas de replica, mais participant au vote.

Number of Members.	Majority Required to Elect a New Primary.	Fault Tolerance.
3	2	1
4	3	1
5	3	2
6	4	2

Pour aller plus loin : Replication Election and Consensus Algorithm Refinements for MongoDB  
<https://www.youtube.com/watch?v=3n9MTE-QSg4>

## A l'action! Test de la réPLICATION

On crée deux instances du serveur sur la même machine, avec une base chacun.

```
cd /data; mkdir noeud1; mkdir noeud2
```

Lançons maintenant les 2 serveurs, chacun sur un port. Ils forment le RS **test**.

```
mongod --port 27017 --replSet test --dbpath /data/noeud1&
mongod --port 27018 --replSet test --dbpath /data/noeud2&
```

Connectons un client au premier serveur.

```
mongo --port 27017
```

Et initialisons le RS

```
rs.initiate()
```

On peut alors ajouter le second nœud (remplacer localhost par le nom de la machine).

```
rs.add("localhost:27018")
```

## Continuons

`db.isMaster()` : indique s'il s'agit du monogod primary

`rs.status()` : informations complètes sur le replica set

### Test d'insertion

```
use dbtp;
db.col.insert ({"_id": 1, "nom": "abc"})
```

On devrait trouver le document sur le esclave ?

```
mongo --port 27018
use dbtp;
db.col.find ()
```

Que se passe-t-il? Et après `rs.slaveOk()`? Pourquoi donc?

Les lectures ne sont autorisées sur le noeud secondaire qu'après l'exécution de `rs.slaveOk()`

## Continuons

`db.isMaster()` : indique s'il s'agit du monogod primary

`rs.status()` : informations complètes sur le replica set

### Test d'insertion

```
use dbtp;
db.col.insert ({"_id": 1, "nom": "abc"})
```

On devrait trouver le document sur le esclave ?

```
mongo --port 27018
use dbtp;
db.col.find ()
```

Que se passe-t-il? Et après `rs.slaveOk()`? Pourquoi donc?

Les lectures ne sont autorisées sur le noeud secondaire qu'après l'exécution de  
`rs.slaveOk()`

## Et la reprise sur panne?

Tuons (gentiment) notre *maître*.

```
db.shutdownServer()
```

Question : le second va-t-il s'élire *maître* ?

Relancer le premier serveur. Que se passe-t-il ?

Répéter l'expérience en ajoutant un troisième nœud.

```
mkdir /db/noeud3;  
mongod --port 27019 --replSet test --dbpath /data/noeud3\&
```

### Remarque

Un bon *driver* devrait enregistrer la liste des serveurs du RS, et se reconnecter automatiquement au *maître* quand celui-ci change.

## Quizz

- 1 Qu'est-ce que la technique de journalisation ? Quel est son but ?
- 2 Rappeler la définition des réPLICATIONS synchrones et asynchrones, et indiquez brièvement les avantages/ inconvénients de chacune.
- 3 Dans quelle architecture distribuée peut-on aboutir à des écritures conflictuelles ? Donnez un exemple.
- 4 La majorité des serveurs dans une grappe maître-esclaves perd le contact avec le noeud maître. Comment s'effectue le *failover* ?
- 5 Comment obtient-on une cohérence forte dans un système maître-esclave avec des réPLICATIONS asynchrones ?
- 6 L'ajout d'un serveur à un replica set, n'implique pas toujours une meilleure tolérance aux pannes, pourquoi ? Donnez un exemple.
- 7 La réPLICATION est un outil essentiel pour garantir : la disponibilité des données(High Availability), la protection des données (Data Protection) ou la répartition de la charge (Load Balancing) ?

## Chapitre 6 - Systèmes NoSQL : le partitionnement (sharding)

- Principes généraux

- Partitionner, pourquoi?
  - Partitionner, comment?

- Tolérance aux pannes et évolution du système
- Choix de la clé de partitionnement
- Le sharding dans la pratique
- Partitionnement distribué par intervalle
- Partitionnement distribué par hachage

## Partitionner, pourquoi?

**Big Data** : nécessite de **très hautes capacités de stockage** (disques) **et de traitement** (CPU et RAM)

Classiquement on a 2 approches pour l'augmentation des capacités : la scalabilité verticale et la scalabilité horizontale.

Scalabilité verticale : augmentation de la puissance de calcul sur une seule machine.

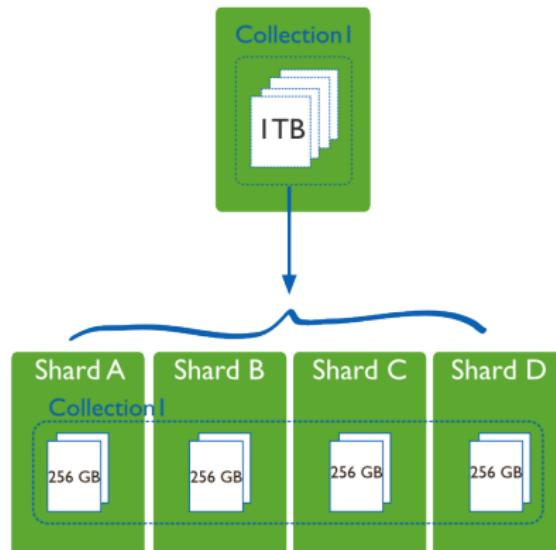
### Limites de la **scalabilité verticale**

- **Scalabilité limitée** : les ressources d'une seule machine ne peuvent pas croître indéfiniment
- A performances égales, une machine à très hautes performances est **disproportionnellement** plus **chères** qu'un cluster de machines à bas coût (*commodity servers*).
- Les fournisseurs d'infrastructures cloud ne permettent que l'approvisionnement de "petites" instances (machines virtuelles).

## Partitionner, pourquoi?

**Scalabilité horizontale** : augmentation de la puissance de calcul par l'allocation de nouvelles machines

- Les données et les calculs sont répartis sur une grappe de serveurs
- **Augmentation du débit et réduction de la charge** (nombre d'opérations) de chaque serveur



## Partitionner, comment?

Déroulement du **sharding** (i.e. partitionnement)

1. Découper une (grande) collection en **fragments (chunks)** en fonction d'un champ.

le champ selon lequel les données sont partitionnées est appelé **clé de partitionnement** (i.e. **shard key**).

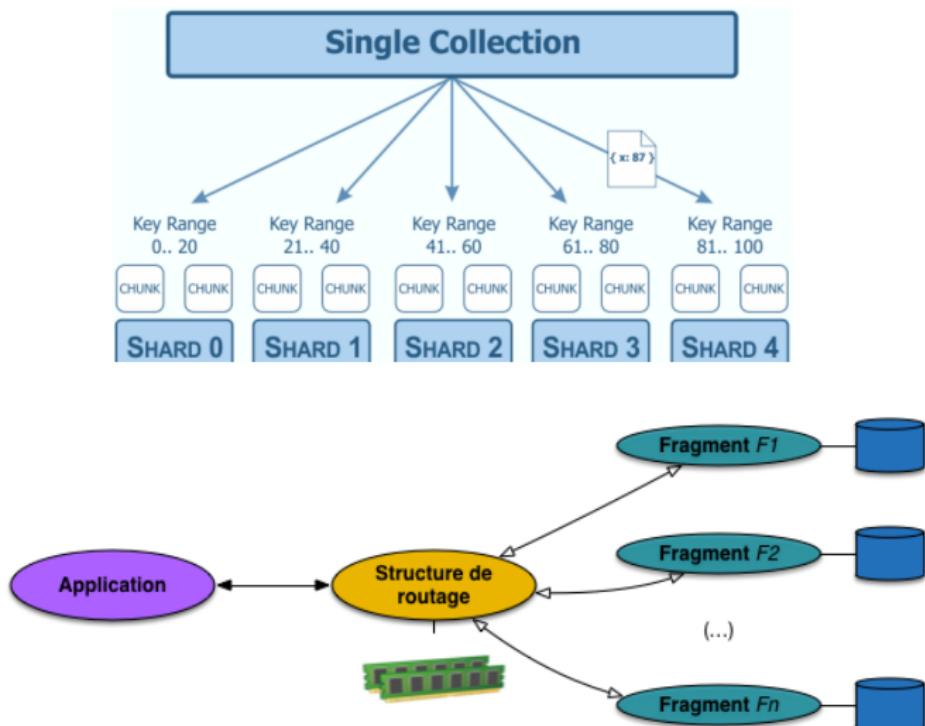
**Chunk** : a généralement une **taille maximale fixe prédefinie** (de 64 MO à qq GO). Appelé également *tablet, region, bucket*, etc.

2. Distribuer les chunks sur les serveurs (i.e. **shards**) ;
3. Maintenir un **répertoire** indiquant que tel chunk se trouve dans tel shard
4. Un serveur (le **Routeur**), ou ensemble de serveurs, consulte le répertoire et oriente les recherches vers le bon shard<sup>5</sup>.

---

<sup>5</sup> Autres solutions : (i) le répertoire est répliqué sur chaque noeud (acceptable quand on n'ajoute/retire pas de serveurs tout le temps), (ii) le routeur est intégré au client.

## Partitionner, comment?



## Chapitre 6 - Systèmes NoSQL : le partitionnement (sharding)

- Principes généraux
- Tolérance aux pannes et évolution du système
- Choix de la clé de partitionnement
- Le sharding dans la pratique
- Partitionnement distribué par intervalle
- Partitionnement distribué par hachage

# Équilibrage de la charge et élasticité

Le sharding doit assurer l'**élasticité** et l'**équilibrage de charge**

## Load balancing

- La charge doit être équitablement répartie sur les serveurs de la grappe, sinon le serveur le plus lent ralentit tout le traitement
- e.g.* Rappelez-vous, les reducers ne peuvent démarrer que si TOUS les mappers ont terminé leur travail!

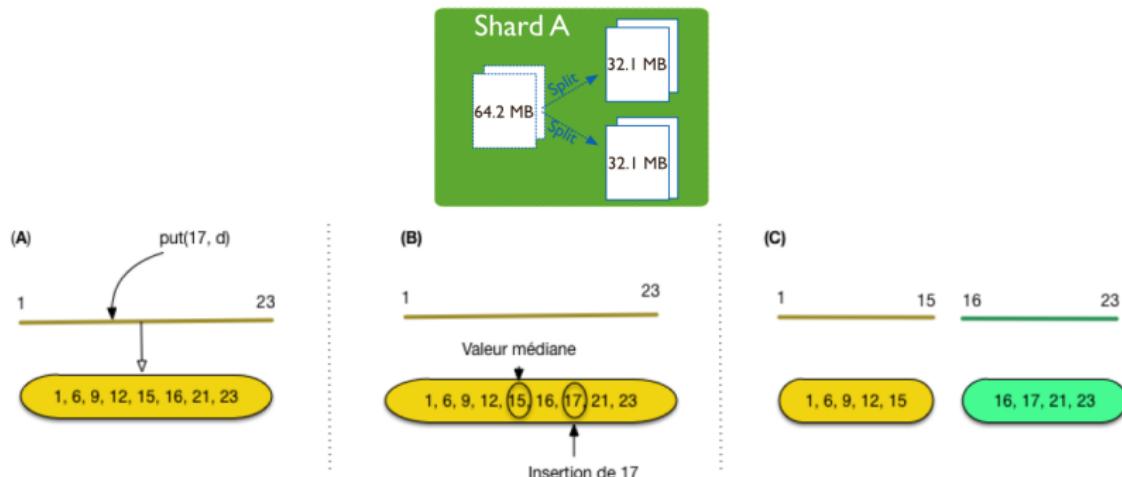
**Répartition dynamique (élasticité)** : adaptation automatisée

- à l'**ajout/suppression de données**
- à l'**ajout/disparition de serveurs**

# Équilibrage de la charge et élasticité

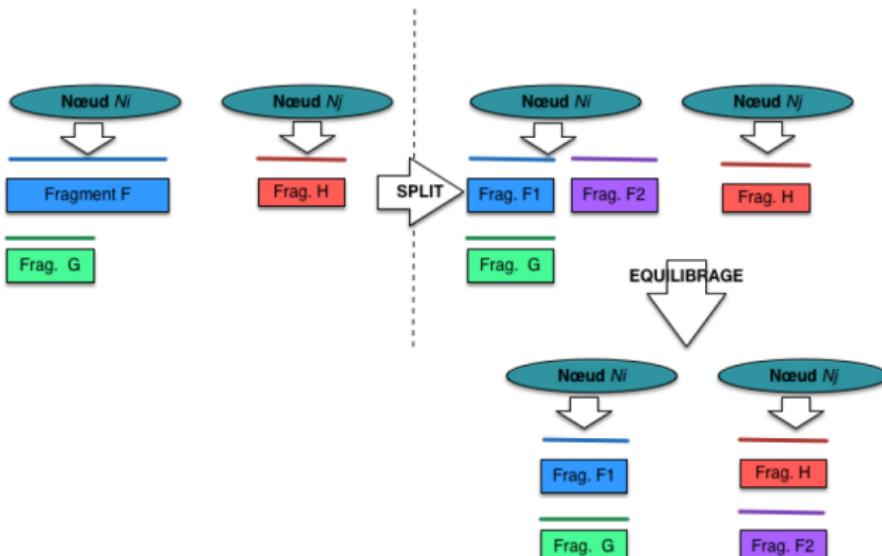
L'équilibrage de charge et l'élasticité se basent sur 2 opérations : le **split** et la **migration** de chunks

- **Split** : Diviser un chunk qui a dépassé la taille maximale pré définie



# Équilibrage de la charge et élasticité

- Migration et équilibrage** : déplacement d'un chunk d'un shard à un autre pour une répartition équitable



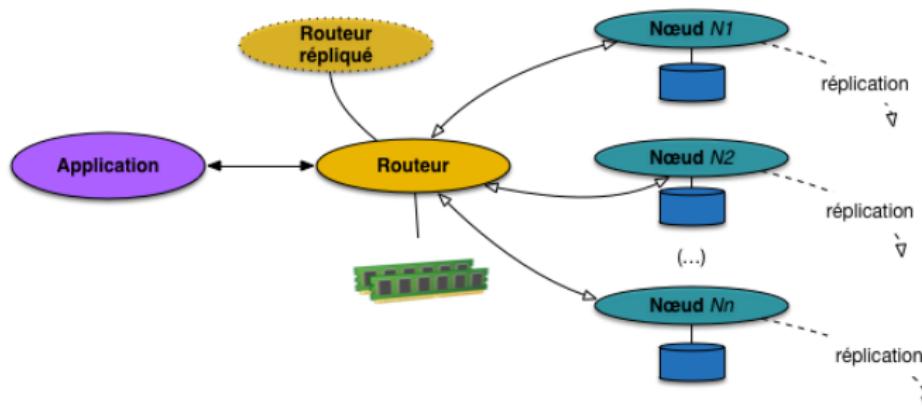
- Le "split" n'est pas coûteux, la migration l'est. Pourquoi?

⇒ généralement le système effectue une seule migration à la fois (à un instant donné)

## Tolérance aux pannes

Un système NoSQL doit savoir gérer la reprise sur pannes **failover**

- Le serveur stockant le répertoire est un SPOF ⇒ **Le répertoire est répliquée** : 3 copies au minimum dans un environnement de production.
- Les **fragments sont répliqués**



## Chapitre 6 - Systèmes NoSQL : le partitionnement (sharding)

- Principes généraux
- Tolérance aux pannes et évolution du système
- **Choix de la clé de partitionnement**
- Le sharding dans la pratique
- Partitionnement distribué par intervalle
- Partitionnement distribué par hachage

## Choix de la clé de partitionnement (shard key)

**Champ** ou ensemble de champs (à éviter) **en fonction duquel les documents sont répartis**

Devrait être présent dans chacun des documents (pourquoi?)

Si un grand nombre de documents partagent la **même clé**, le système **perd en souplesse de distribution**. Pourquoi?

La valeur de la clé pour un document **ne devrait pas être modifiable**. Pourquoi?

L'identifiant séquentiel d'un document est-il un bon choix? (nous y reviendrons)

**Le choix de la clé a une très grande incidence sur les performances des requêtes**

## Opérations sur la clé

*get(k)* : chercher l'intervalle qui contient  $k$ , transmettre au serveur correspondant.

*put(k, d)* : identifier le serveur avec  $k$ , transmettre l'insertion

*delete(k)* : idem

*range(k<sub>1</sub>, k<sub>2</sub>)* : transmettre à tous les serveurs gérant un intervalle chevauchant [k<sub>1</sub>, k<sub>2</sub>]

Toute autre opération : transmettre à tous les serveurs.

Intervalle	Serveur
$] -\infty, a]$	A
$[a+1, b]$	B
$[b+1, c]$	C
$[c+1, d]$	B
$[d+1, \infty[$	A

## Taille des fragments et efficacité du routage

La structure de routage est constituée de paires  $(I, a)$  où  $I$  est la description d'un intervalle et  $a$  l'adresse du fragment correspondant.

- si la taille d'un fragment est de 4 KO (choix typique d'un SGBD relationnel), le routage décrit 250 millions de fragments, soit une taille de 5 GO ;
- si la taille d'un fragment est de 1 MO, il faudra 1 million de fragments, et seulement 20 MO pour la structure de routage.

Dans tous les cas, le routage tient en mémoire RAM

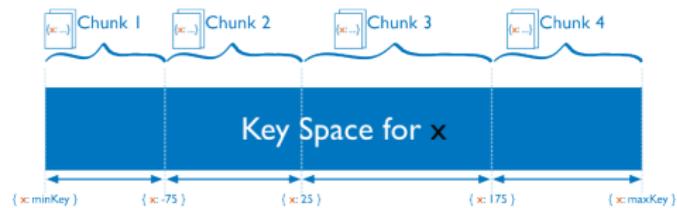
Intervalle	Serveur
$] - \infty, a]$	A
$[a + 1, b]$	B
$[b + 1, c]$	C
$[c + 1, d]$	B
$[d + 1, \infty[$	A

# Techniques de partitionnement

Deux techniques de partitionnement

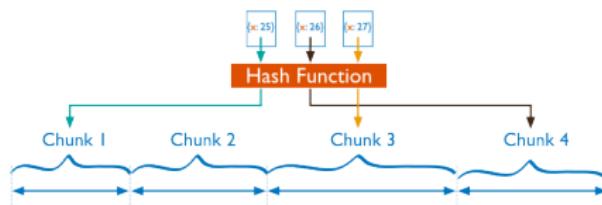
- **par intervalle** : données triées sur la clé, puis groupées par intervalles.

Exemples représentatifs : MongoDB, HDFS (Hadoop), GFS (Google).



- **par hachage** : par hachage sur la clé, avec répertoire de hachage.

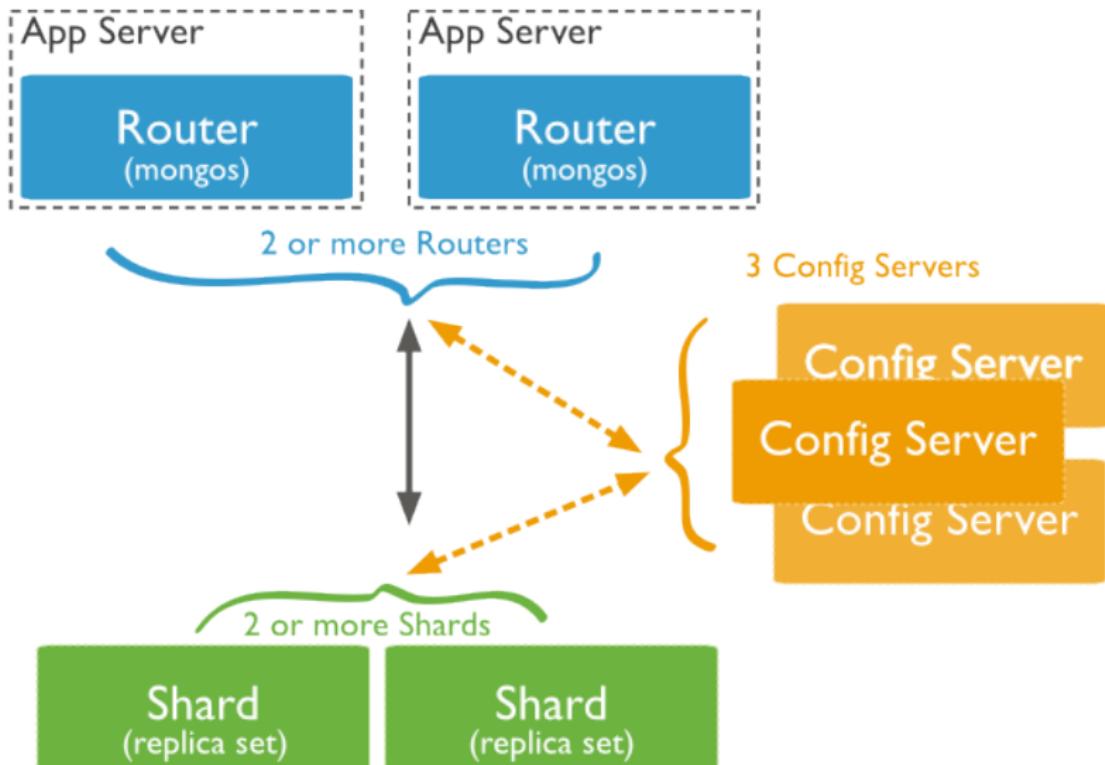
Exemples représentatifs : Chord, Dynamo, Cassandra, beaucoup d'autres.



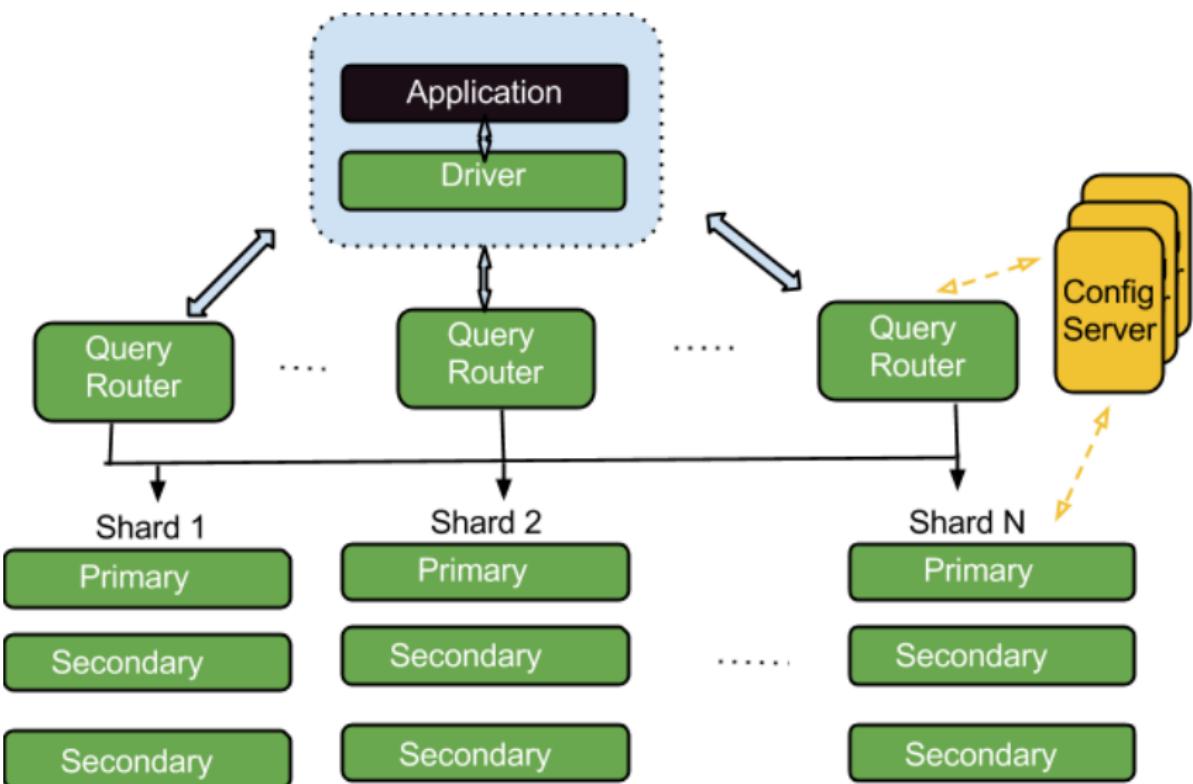
## Chapitre 6 - Systèmes NoSQL : le partitionnement (sharding)

- Principes généraux
- Tolérance aux pannes et évolution du système
- Choix de la clé de partitionnement
- **Le sharding dans la pratique**
  - Le sharding de MongoDB
  - Le sharding de Hadoop HDFS
  - Le sharding de Google File System (GFS)
- Partitionnement distribué par intervalle
- Partitionnement distribué par hachage

## Le sharding de MongoDB



## Le sharding de MongoDB



# Le sharding de MongoDB

## Config Server

- Stocke le répertoire (obligatoirement répliqué) : "**Serveurs de métadonnées**"
- Processus mongod

## Instances mongos (Query Routers)

- prend en charge les requêtes des clients.
- consulte le répertoire (Config Server) et oriente les requêtes vers le shard approprié
- agrège les résultats et envoie la réponse au client.
- stateless** (ne stocke pas les données)
- Il peut y avoir plusieurs routeurs (mais un client envoie une requête à un seul)

## Chaque shard est un replica set complet

- Shard : "**serveur de données**"
- Il s'agit d'un processus mongod maître avec un ou plusieurs autres processus mongod secondaires et éventuellement un mongod arbitre
- On peut **ajouter ou supprimer un shard dynamiquement** (le processus d'arrière plan Balancer se charge de répartition de la charge).

Pour en savoir plus : <https://www.youtube.com/watch?v=j2mYoEW9ehk>

## Allons-y pour un petit test

On lance un config server

```
mkdir /data/configdb
```

```
mongod --configsvr --dbpath /data/configdb --port 27019
```

(Si le configserver est répliqué)

```
mongod --configsvr --replicaSet nomRS --dbpath <path>
```

On lance un routeur en lui indiquant où se trouve(nt) le(s) config server (s)

```
mongos --configdb localhost:27019
```

(Si le configserver est répliqué)

```
mongos --configdb
```

```
nomRS/confServer1:port1,confServer2:port2,confServer3:port3
```

Et deux serveurs (il faudrait des replica sets complets en production)

```
mongod --dbpath /data/node1 --port=30001
```

```
mongod --dbpath /data/node2 --port=30002
```

À partir de mongos, déclarons les deux serveurs comme shards.

```
mongo --port 27017
```

```
sh.addShard("localhost:30001")
```

```
sh.addShard("localhost:30002")
```

```
sh.enableSharding("dbtp")
```

## Choisir la clé de partitionnement

On indique la clé de partitionnement avec la commande suivante :

```
sh.shardCollection ("dbtp.movies", {"title": 1})
```

### Attention au choix de la clé

- Les clés-séquences : on envoie toujours les insertions au même serveur; bon ou pas?
- Les clés à faible cardinalité (ex : pays) : pas très bon non plus, car il pourra être difficile de faire un split

Bonne option : appliquer un hachage pour bien distribuer les insertions (mais attention, requêtes par intervalle lentes)

```
sh.shardCollection ("dbtp.movies", {"title": "hashed"})
```

## Inspecter la configuration

### Liste des *shards*

```
mongos> db.runCommand({listshards: 1})
```

### Statut général du cluster

```
mongos> sh.status()  
mongos> db.stats()  
mongos> db.printShardingStatus()
```

### Les fragments sont dans une collection *chuncks*.

```
use config  
db.chuncks.count()  
db.chuncks.findOne()
```

# Le sharding de Hadoop HDFS (Hadoop Distributed File System)

**Hadoop** : **Framework** open source pour le **stockage** et le **traitement** (MapReduce) **distribués**.

Écrit en Java/C<sup>6</sup>, inspiré de Google FS et son MapReduce

Principaux supports/contributeurs : Yahoo, FaceBook (0.5 PB/jour en 2015), IBM, Hortonworks, etc.

Composantes clés

- **HDFS pour le stockage distribué**
- **MapReduce pour le traitement distribué**

Écosystème

- HBASE : BD Distribuées
- ZooKeeper : Configuration/synchronisation
- Pig/Hive : Langages de haut niveau
- YARN (Hadoop 2.0) : Traitements autres que MapReduce
- Etc.

<sup>6</sup>Mais utilisable avec d'autres langages

## Le sharding de Hadoop HDFS (Hadoop Distributed File System)

**HDFS** : repose sur 2 types de noeuds, les NameNodes et les DataNodes

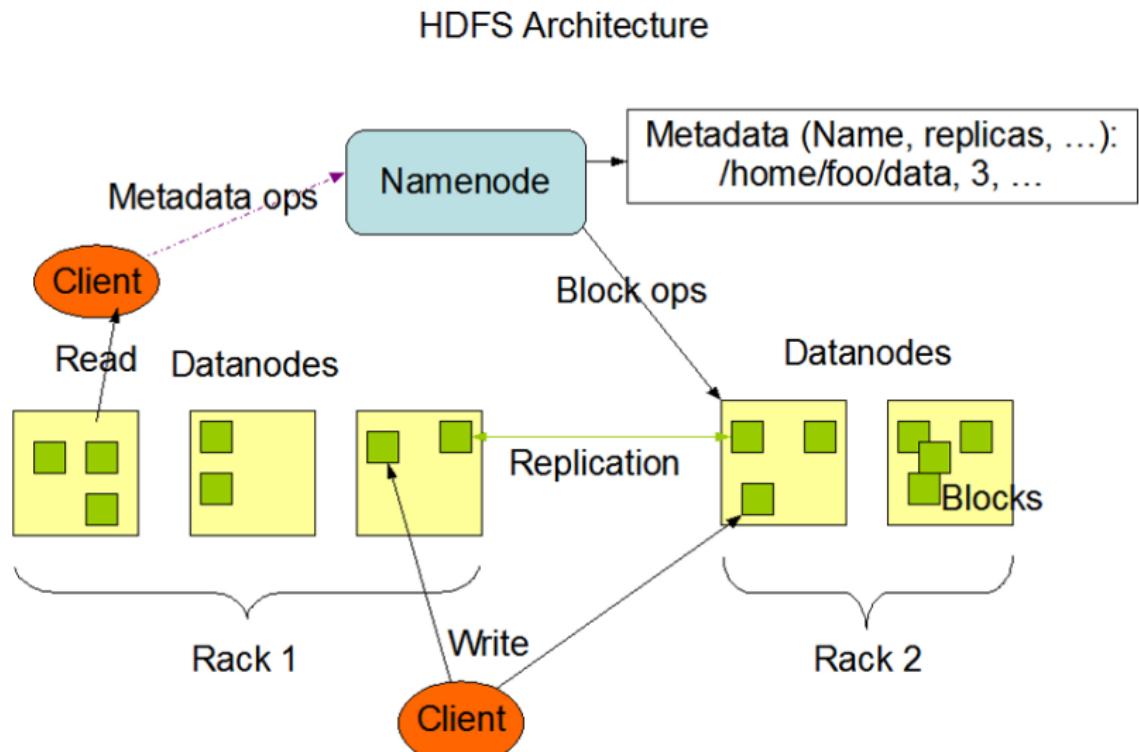
**DataNode** (serveur de données) : **stocke** et **restitue** les **blocs de données** (les chunks)

**NameNode** (serveur de métadonnées)

- **Stocke le répertoire**, l'espace des noms, l'arborescence et les métadonnées des fichiers.
- **Centralise la localisation des blocs** dans le cluster HDFS
- Lors de la lecture des blocs d'un fichier, le NameNode est interrogé. Il renvoie pour chaque bloc, l'adresse du DataNode le plus accessible (qui a la plus grande bande passante)
- Les DataNodes envoient périodiquement au NameNode la liste des blocs que chacun héberge.
- Si le NameNode constate qu'un bloc n'est pas suffisamment répliqué, il **initie une réPLICATION sur d'autres DataNodes**
- Le NameNode capture périodiquement des **snapshots (fsImage)** des métadonnées et enregistre les modifications intermédiaires dans un fichier edit Log (e.g. renommage d'un fichier, migration d'un chunk, etc.).
- Il est unique mais dispose d'un noeud secondaire
- Hadoop v1 : le noeud secondaire actualise le fsImage à partir de l'edit log et ne sert qu'à une reprise plus rapide du NameNode (pas de *High Availability* et pas de *failover automatique*). 

## Le sharding de Hadoop HDFS (Hadoop Distributed File System)

Architecture un peu différente, mais les grands principes sont là!



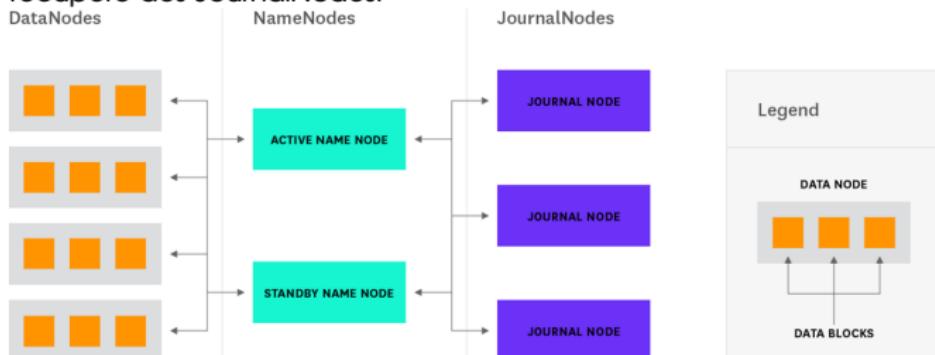
## Le sharding de Hadoop HDFS (Hadoop Distributed File System)

Hadoop v2 : entre autres, amélioration de la haute disponibilité

**StandBy NameNode** : permet une reprise sur panne automatisée et une haute disponibilité.

Pour pouvoir récupérer l'editlog même en cas d'indisponibilité du NameNode, celui-ci est stocké sur des noeuds tiers (JournalNodes)

**JournalNodes** : le NameNode stocke l'editlog dans les JournalNodes et le standby le récupère des JournalNodes.

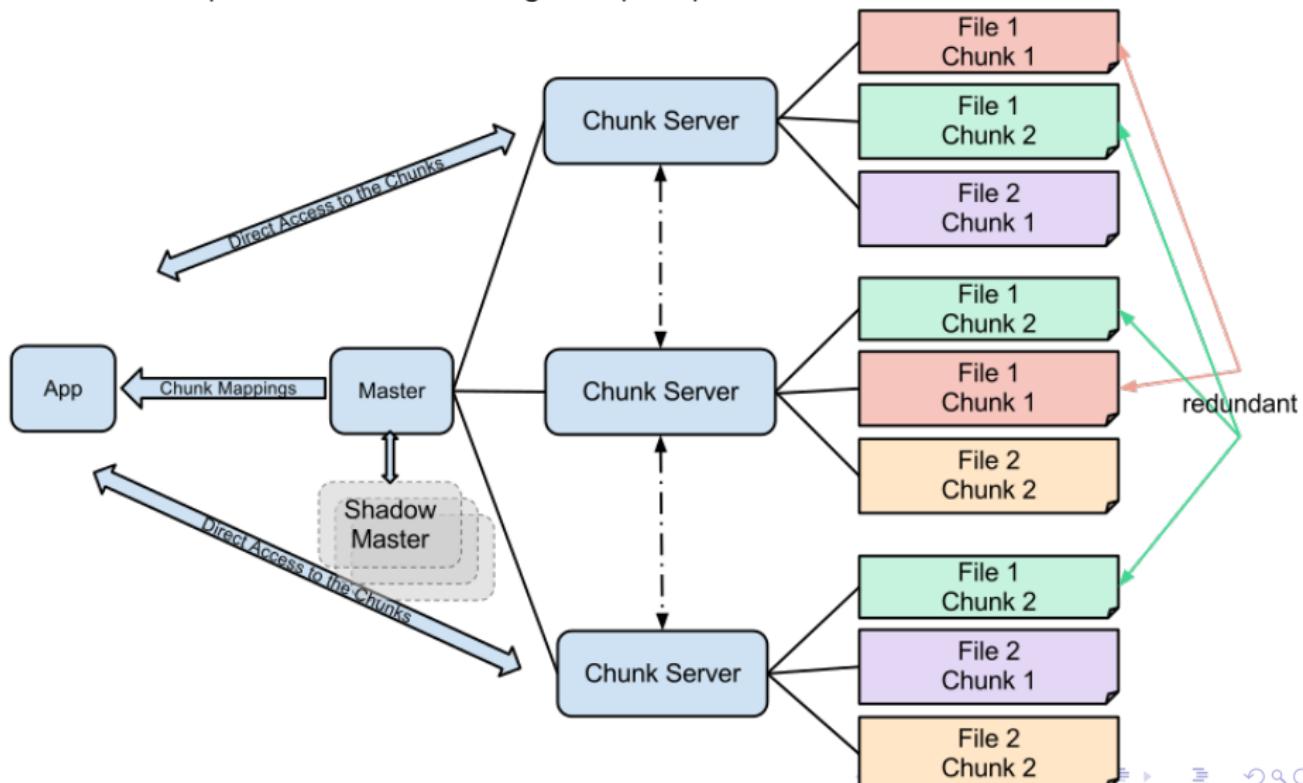


Les JournalNodes ne requièrent pas beaucoup de ressources et peuvent tourner sur des machines existantes du cluster.

Hadoop peut tolérer la perte de  $(N+1)/2$  JournalNodes

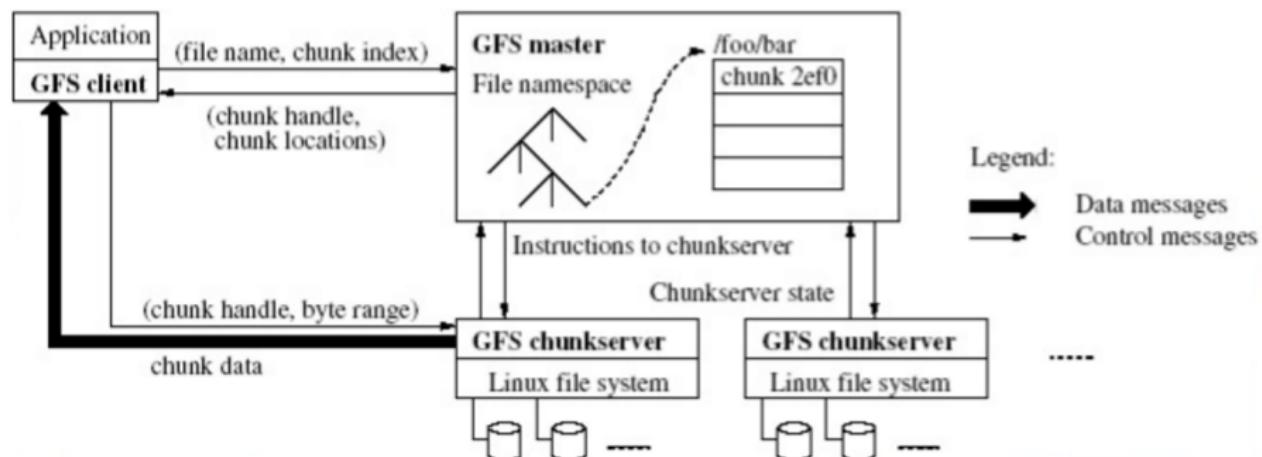
## Le sharding de Google File System (GFS)

Architecture un peu différente, mais les grands principes sont là!



## Le sharding de Google File System (GFS)

Architecture un peu différente, mais les grands principes sont là!



## Chapitre 6 - Systèmes NoSQL : le partitionnement (sharding)

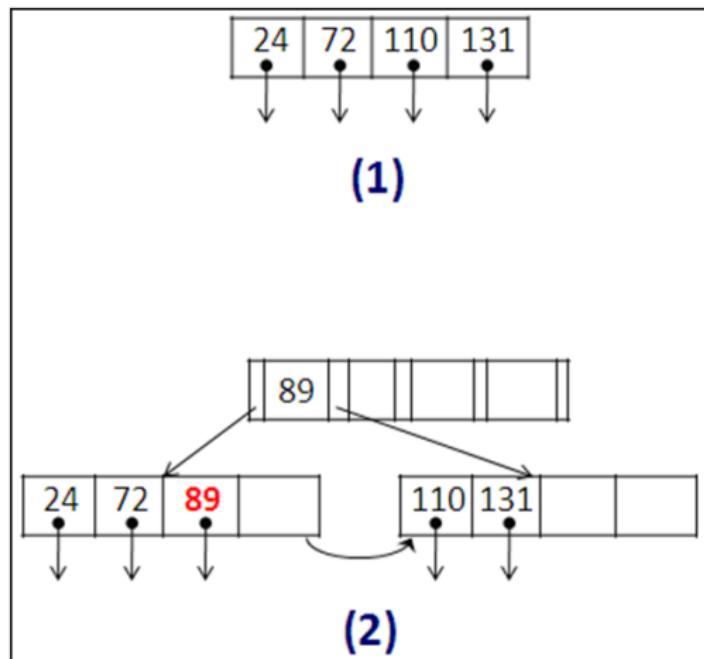
- Principes généraux
- Tolérance aux pannes et évolution du système
- Choix de la clé de partitionnement
- Le sharding dans la pratique
- Partitionnement distribué par intervalle
  - Rappel de l'arbre B+
- Partitionnement distribué par hachage

## Définition

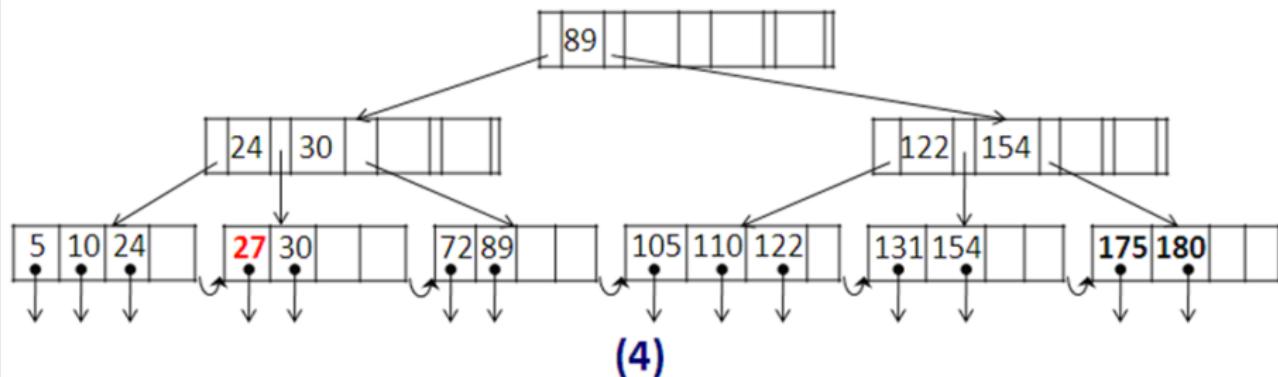
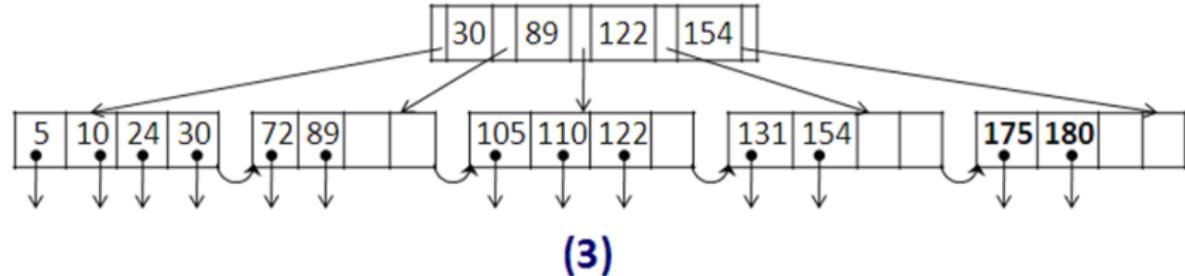
Un arbre B+ d'**ordre  $m$**  est un arbre équilibré tel que :

- ① Chaque noeud "père" (non terminal) possède au maximum  $m$  fils
- ② Chaque noeud non terminal, excepté la racine, possède au minimum  $\lceil \frac{m}{2} \rceil$  fils
- ③ La racine a 0 ou au moins deux fils
- ④ Un noeud non terminal contenant  $q$  clés possède  $q + 1$  fils
- ⑤ Les clés de chaque noeud sont triées
- ⑥ Pour optimiser les recherches par intervalle
  - ① Les noeuds feuilles sont chaînés
  - ② Toutes les clés apparaissent au niveau de feuilles

## Exemple



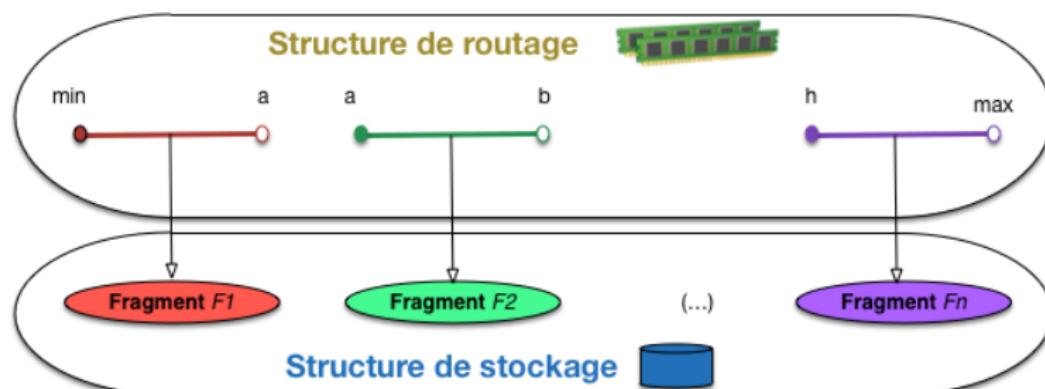
## Exemple



## Principe du partitionnement par intervalle

La collection est **triée** sur la clé et on la découpe en fragments d'une taille maximale pré-déterminée (de 64MO à qq GO)

Chaque fragment couvre donc un intervalle  $[min_f, max_f]$ .



## Chapitre 6 - Systèmes NoSQL : le partitionnement (sharding)

- Principes généraux
- Tolérance aux pannes et évolution du système
- Choix de la clé de partitionnement
- Le sharding dans la pratique
- Partitionnement distribué par intervalle
- **Partitionnement distribué par hachage**
  - Rappel du hachage
  - Principe du partitionnement par hachage

# Hachage statique

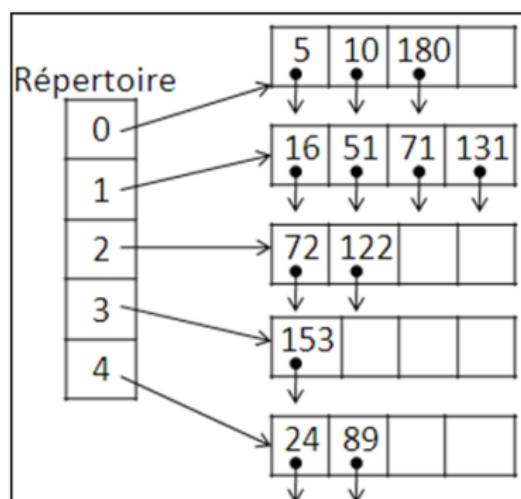
## Principe

- Nous disposons de  $s$  shards ( $s \ll n$ , avec  $n$  le nombre total d'entrées ( $k, s$ ))
  - Une **fonction de hachage  $h$**  associe chaque valeur de clé à un des  $s$  shards. Exemple :  
$$h(k) = k \bmod s$$
  - Recherche d'une clé  $k \Rightarrow$  calcul de  $h(k)$  pour trouver l'adresse du serveur contenant  $k$
  - $h$  est choisie de manière à répartir uniformément les clés dans les shards
- ⇒ Simple et efficace !

## Hachage statique

**Exemple :**  $n = 12, s = 5$

- $h(k) = k \bmod 5$
- Un **répertoire** de 5 entrées de 0 à 4 pointe vers les shards



## Hachage statique

### Limites

- Pas de recherche par intervalle
- En cas d'ajout d'un nouveau serveur, nous devons modifier la fonction de hachage (*i.e.*  $h(k) = k \bmod 6$ ), ce qui implique de réaffecter tous les documents, régénérer le répertoire, etc.

**Pas du tout efficace, ni envisageable!**

⇒ Nous devons trouver un moyen pour permettre l'élasticité sans affecter la répartition déjà existante

**Pas du tout simple!**

## Hachage cohérent (*consistent hashing*) - Cas pratique Hash Ring Cassandra

En 1997, un article a proposé une solution maintenant très largement utilisé : le **hachage cohérent** ou *Consistent hashing*.

- D'abord conçu et utilisé pour des **caches distribués** (*memcached*)
- Popularisé pour la gestion de données par le système Dynamo (Amazon, 2007)
- Maintenant intégré à de très nombreux systèmes : Voldemort, Riak, Chord, ...

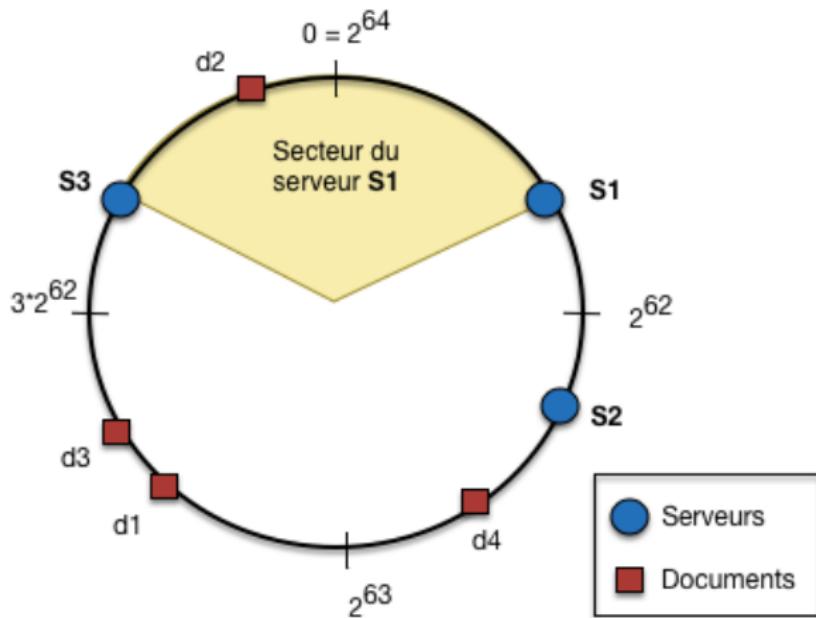
Avec le **Consistent hashing**, ajouter ou retirer un serveur n'affecte la collection que de façon **marginale**.

- on prend une fonction **permanente**,  $h$ , appliquée aux clés **et** aux serveurs vers l'espace d'adressage  $A = [0, 2^{64} - 1]$  ;
- $A$  est organisé comme un anneau parcouru dans le sens des aiguilles d'une montre ;
- si  $S$  et  $S'$  sont deux serveurs adjacents sur l'anneau, toutes les clés de l'intervalle  $[h(S), h(S')]$  appartiennent à  $S'$ .

<sup>7</sup> valeur maximale pour un entier stocké 8 octets

## Illustration - Cas pratique Hash Ring Cassandra

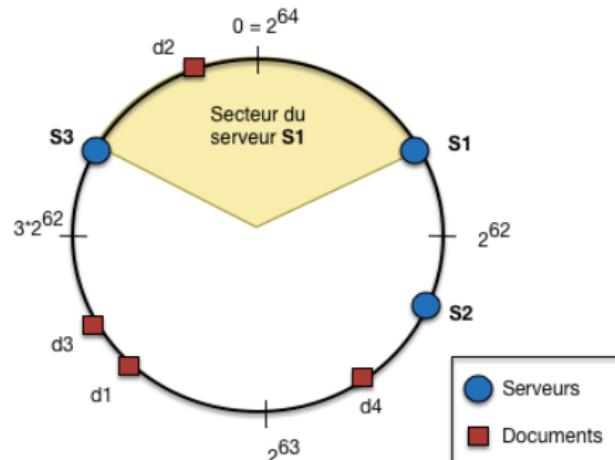
Serveurs **et** documents sont placés sur l'anneau.



## La table de hachage - Cas pratique Hash Ring Cassandra

Elle établit une correspondance entre le découpage de l'anneau en arcs de cercle, et l'association de chaque arc à un serveur.

Intervalle	Serveur
$]c, a]$	S1
$]a, b]$	S2
$]b, c]$	S3



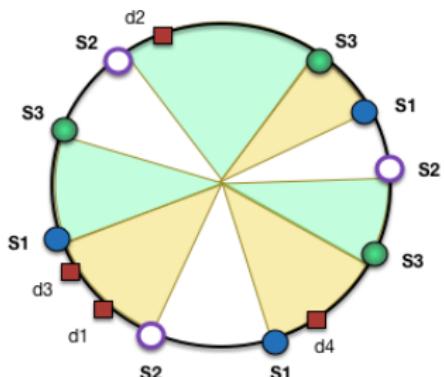
## Quelques détails vraiment utiles - Cas pratique Hash Ring Cassandra

Comment fonctionne la tolérance aux pannes ? Comment fonctionne l'équilibrage ?

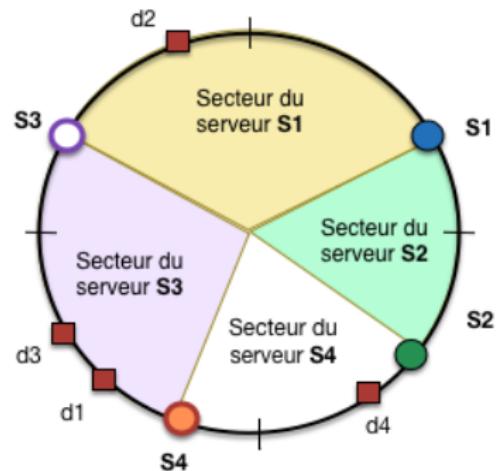
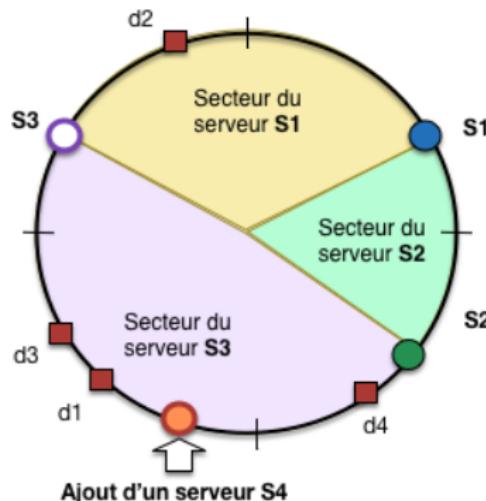
**Panne** ⇒ géré par la réPLICATION (surprise !) ; par exemple on copie sur la machine suivante dans l'anneau, etc.

**Équilibrage** ⇒ un même serveur (physique) est distribué en plusieurs points (virtuels) sur l'anneau.

- plus il y a de points, plus le serveur recevra de données ;
- en cas de panne, réorganisation répartie plus justement.



## Ajout et suppression de serveurs - Cas pratique Hash Ring Cassandra



Une organisation **locale** est suffisante ( $S_3$  transmet à  $S_4$ )

## Quizz

- Qu'est-ce qu'un bon shard key?
- Je décide de partitionner ma collection de films sur le genre. Discuter des avantages et inconvénients de ce choix.
- Expliquez brièvement le rôle du routeur?
- Dans une structure de partitionnement, quelles opérations ne sollicitent qu'un seul serveur ?
- Et si on envoyait la structure de routage aux clients au moment où ils se connectent ?  
Discuter des avantages et inconvénients.
- Une structure de routage indexe les fragments. Pourquoi ne pas indexer les documents eux-mêmes ?
- En quoi la taille des fragments influence-t-elle la table de routage ?
- Peut-on avoir des fragments trop gros ? Des fragments trop petits ? Justifiez, discutez.
- Quelle est le type de panne le plus problématique dans une structure de partitionnement ?
- La clé de ma collection de vidéos est le titre. Est-il possible de faire une recherche par année dans un système distribué ?
- Un même serveur peut-il gérer plusieurs fragments ?
- Un même serveur peut-il héberger un noeud de routage et un noeud de stockage ?
- Mes documents sont identifiés par un numéro engendré automatiquement et incrémentalement au moment de l'insertion. Que se passe-t-il avec un partitionnement par intervalle ? Quel est l'inconvénient / l'avantage ?
- Même question avec un partitionnement par hachage.

## Exercice

Un système documentaire réparti utilise le hachage cohérent (*Consistent Hashing*) pour la répartition d'une collection de documents sur une grappe de serveurs.

La grappe est composée de 2 serveurs  $S_1$  et  $S_2$ , identifiés respectivement par 12007 et 12011. La collection contient 4 documents  $d_1$ ,  $d_2$ ,  $d_3$  et  $d_4$ , identifiés respectivement par 10, 14, 15 et 17. Les identifiants des serveurs et des documents sont récapitulés dans les tableaux ci-après. Pour simplifier nous supposons que le système en question utilise la fonction de hachage :  $h(k) = k \bmod 6$  (le reste de la division par 6).

Serveur	Identifiant
$S_1$	12007
$S_2$	12011

Document	Identifiant
$d_1$	10
$d_2$	14
$d_3$	15
$d_4$	17

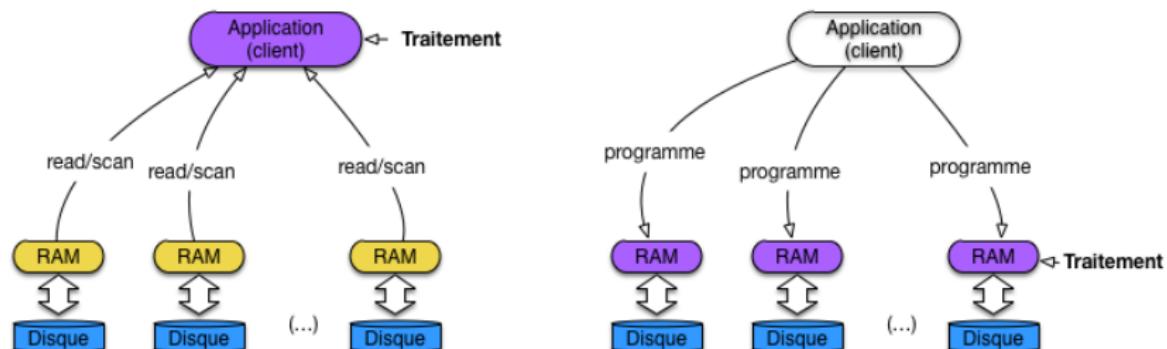
1. Représentez les serveurs et les documents sur l'anneau qui correspond à la table de hachage. En déduire le répertoire (la répartition des documents sur les différents serveurs).
2. Dites ce qui se passe lors de l'ajout d'un serveur  $S_3$  identifié par 12015 à la grappe.
3. Expliquez (en une phrase) ce qu'il faudrait faire pour obtenir un meilleur équilibrage de la charge (*Load Balancing*) entre les serveurs de la grappe.
4. Expliquez (en une phrase) ce qu'il faudrait faire pour gérer les pannes qui peuvent survenir sur un des serveurs.

## Chapitre 7 - Systèmes NoSQL : calcul distribué avec MapReduce

- **MapReduce**
- Le Map/Reduce de Hadoop v1
- Gestion des pannes
- Pig et Hive, des langages de haut niveau

## Les programmes vers les données, pas l'inverse!

Client serveur pour traiter des TOs de données ? **Ne marche pas.**



Il faut placer les traitements au plus près des données.

## Pourquoi un Framework

Un framework **pilote** un traitement, conduit selon un processus générique (notion d'inversion de contrôle).

L'application "injecte" des fonctions dans le framework.

Exemples : Applications MVC, XSLT, Couches ORM, **MapReduce**.

Le framework applique les fonctions pendant l'exécution du processus.

- Une fonction *map()* à appliquer pendant la phase de Map
- Une fonction *reduce()* à appliquer pendant la phase de Reduce.

### Rôle d'un framework MapReduce

Prendre en charge la distribution, et gérer les reprises sur panne.

## Comptons les mots : la fonction de Map

Phase de Map : on extrait les termes, on les compte localement, on transmet au framework.

```
function mapTF($id, $contenu)
{
    // $id: identifiant du document
    // $contenu: contenu textuel du document

    // On boucle sur tous les termes du contenu
    foreach ($t in $contenu) {
        // Comptons le nb d'occurrences de $t dans $contenu
        $count = nbOcc ($t, $contenu);
        // Emission du terme et de son nombre d'occurrences
        emit ($t, $count);
    }
}
```

NB : rien n'indique le contexte de distribution.

## Comptons les mots : la fonction de Reduce

Phase de Reduce : on reçoit, pour chaque terme, tous les compteurs, et on les additionne.

```
function reduceTF($t, $compteurs)
{
    // $t: un terme
    // $compteurs: les nombres d'occurrences, un pour chaque doc.
    $total = 0;

    // Boucles sur les compteurs et calcul du total
    foreach ($c in $compteurs) {
        $total = $total + $c;
    }

    // Et on produit le total
    return $total;
}
```

Même remarque : rien n'indique le contexte de distribution.

## L'exécution par le Framework

URL	Document
$u_1$	the jaguar is a new world mammal of the felidae family.
$u_2$	for jaguar, atari was keen to use a 68k family device.
$u_3$	mac os x jaguar is available at a price of us \$199 for apple's new "family pack".
$u_4$	one such ruling family to incorporate the jaguar into their name is jaguar paw.
$u_5$	it is a big cat.

## Déroulons le processus

term	count
jaguar	1
mammal	1
family	1
jaguar	1
available	1
jaguar	1
family	1
family	1
jaguar	2
...	

Sortie du *map*  
Entrée du *shuffle*

## Déroulons le processus

term	count
jaguar	1
mammal	1
family	1
jaguar	1
available	1
jaguar	1
family	1
family	1
jaguar	2
...	

Sortie du *map*

Entrée du *shuffle*

term	count
jaguar	1,1,1,2
mammal	1
family	1,1,1
available	1
...	

Sortie du *shuffle*

Entrée du *reduce*

## Déroulons le processus

term	count
jaguar	1
mammal	1
family	1
jaguar	1
available	1
jaguar	1
family	1
family	1
jaguar	2
...	

Sortie du *map*

Entrée du *shuffle*

term	count
jaguar	1,1,1,2
mammal	1
family	1,1,1
available	1
...	

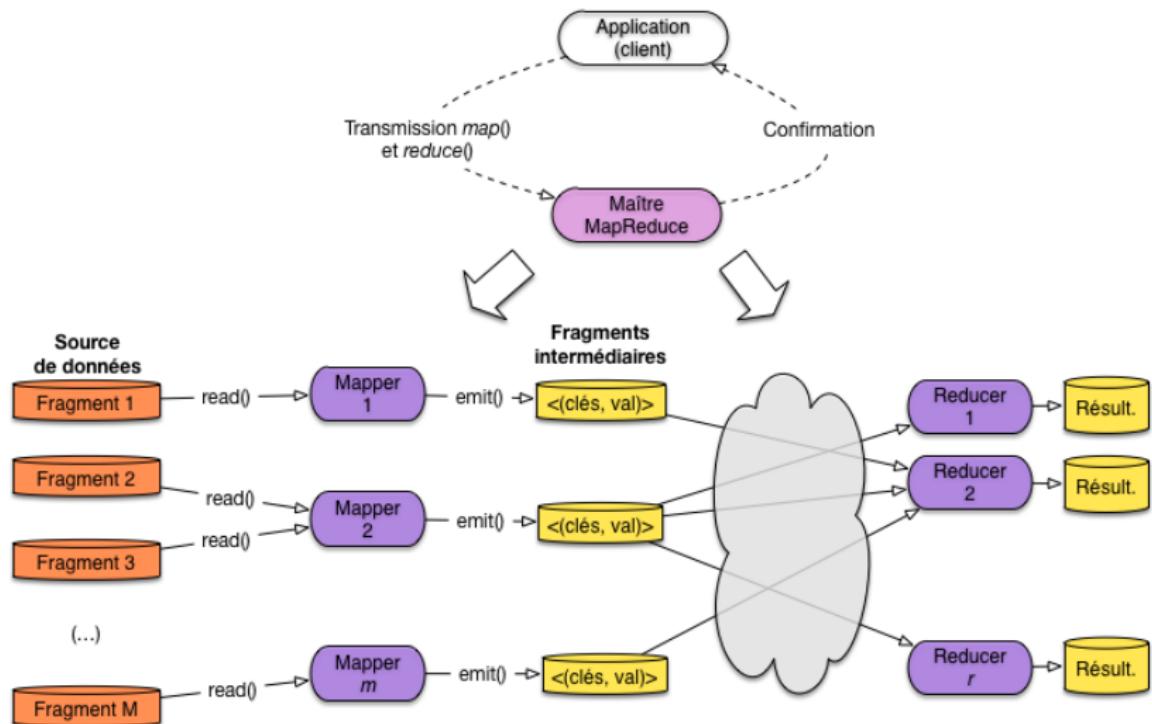
Sortie du *shuffle*

Entrée du *reduce*

term	count
jaguar	5
mammal	1
family	3
available	1
...	

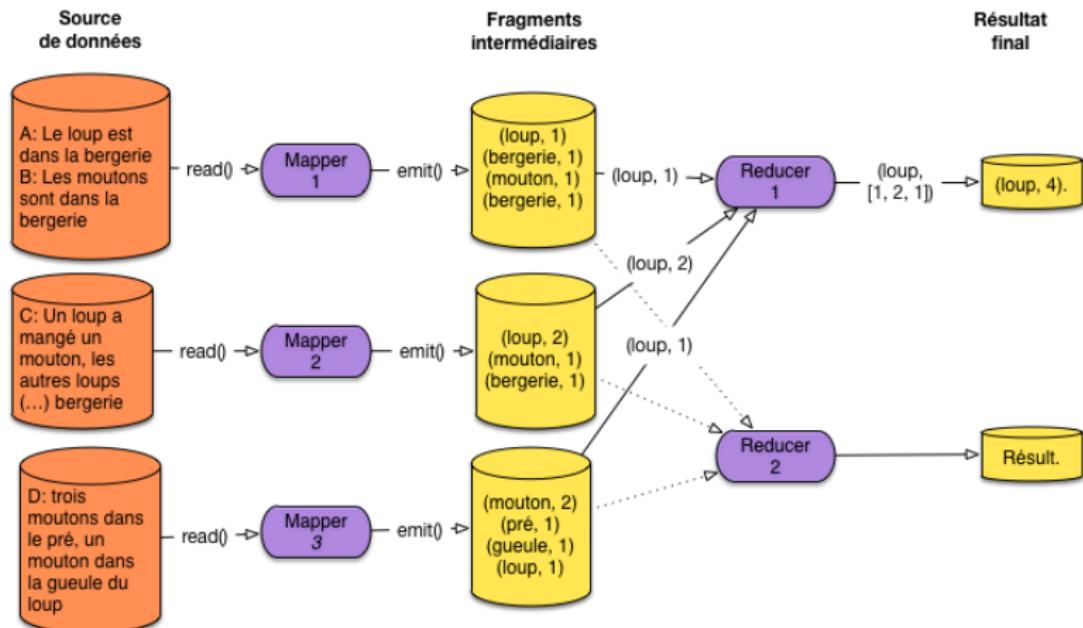
Résultat final

# La grande vision



## Un tout petit exemple

Quatre documents qui parlent de loups, de moutons, de bergerie.

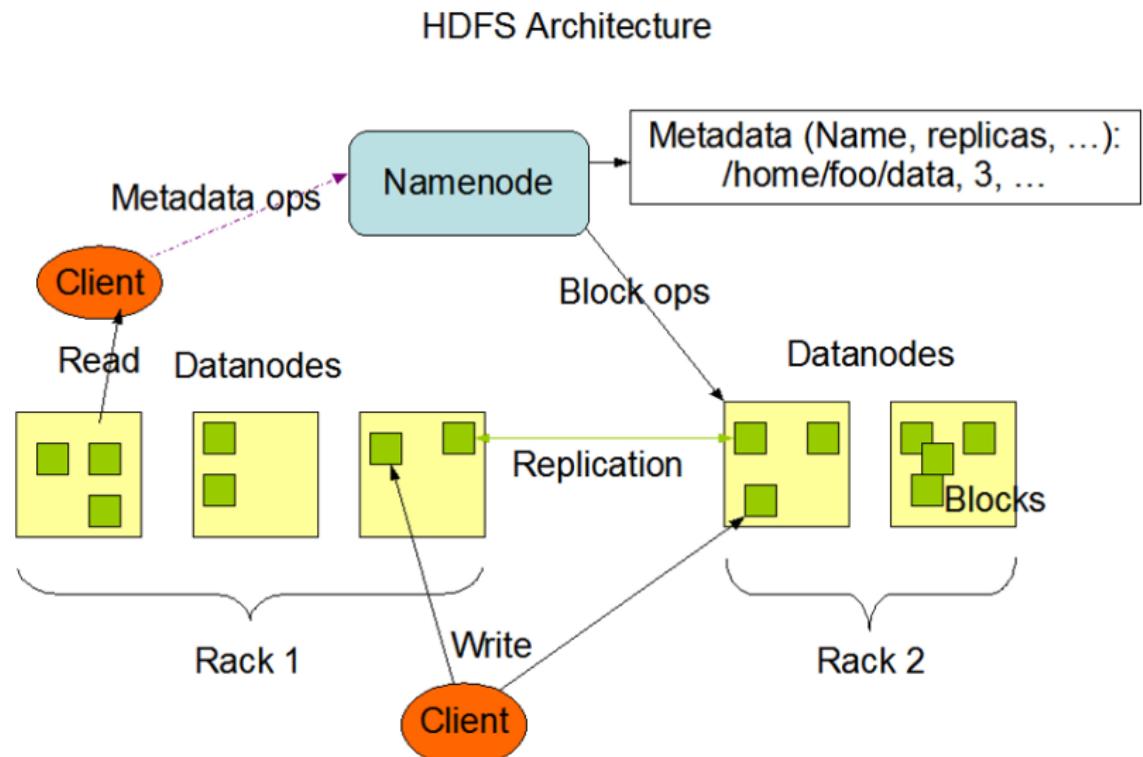


## Chapitre 7 - Systèmes NoSQL : calcul distribué avec MapReduce

- MapReduce
- **Le Map/Reduce de Hadoop v1**
- Gestion des pannes
- Pig et Hive, des langages de haut niveau

# Hadoop v1 Job Tracker et Task Tracker

Rappel



## Hadoop v1 JobTracker et TaskTracker

Hadoop Map/Reduce utilise 2 types de processus :

- **JobTracker** : pour l'**orchestration** (surveillance/reprise sur panne) et l'**injection** des Map() / Reduce() aux noeuds du cluster
- **TaskTracker** : pour l'**exécution d'une tâche** (Map ou Reduce) **sur un noeud**

Déroulement typique

- 1 Les applications clientes soumettent les traitements Map/Reduce au JobTracker
- 2 Le **JobTracker consulte le NameNode** et décide à quels noeuds **injecter les Map()**.
- 3 Idéalement, les Map() sont affectés aux noeuds DataNodes stockant les données interrogées. Si un noeud DataNode est occupé, le Map() est affecté à un noeud se trouvant dans la même rack
- 4 Chaque **TaskTracker exécute localement la fonction Map()** qui lui a été soumise
- 5 Le TaskTracker **notify périodiquement** (par heartbeat) le JobTracker de sa présence
- 6 Si un TaskTracker échoue (n'envoie pas de notification), la tâche est affectée à un autre noeud
- 7 Une fois la phase Map est terminé, le JobTracker **injecte les Reduce()** aux noeuds disponibles. Chaque noeud a un nombre de slots fixe dans lesquels sont mises les tâches en attente.

**Le JobTracker est un SPOF** du Map/Reduce de Hadoop v1. Cette limitation a été résolue dans Hadoop v2.0 (YARN)

## Hadoop v2 YARN

JobTracker et TaskTrakers sont remplacés par : (JobTracker) ResourceManager, ApplicationMaster et (TaskManagers) NodeManagers

### **ResourceManager** : le noeud maître de YARN (un par cluster)

- Responsable de la collecte des informations sur les ressources (RAM et CPU) disponibles. Exécute plusieurs services dont le plus important est le scheduler
- **Scheduler** : le scheduler YARN est responsable de l'allocation des ressources aux applications tournant sur Hadoop. Plusieurs stratégies (FIFO, Fair, Capacity, etc.)

### **ApplicationMaster** (un par application)

- Chaque application a un AM qui lui est propre, tournant sur dans un conteneur indépendant.
- l'AM envoie périodiquement des heartbeats au ResourceManager et lui réclame des ressources supplémentaires au besoin.

### **NodeManagers** (un par noeud)

- Gère les ressources au niveau d'un noeud (il en existe par noeud).
- Crée les containers au niveau d'un noeud et notifie périodiquement le RM d'informations sur l'utilisation des ressources au niveau du noeud.
- Remplace les TaskTrackers. Alors qu'un tasktracker gère un nombre fixe de slots, un NM gère des containers dynamiquement créés. Contrairement aux slots, les containers peuvent être utilisés pour des tâches map, des tâches reduce et des tâches d'autres frameworks que Map/Reduce.

## Hadoop v2 YARN : déroulement typique de l'exécution d'une application

- ➊ Le client soumet une application MapReduce au ResourceManager
- ➋ Le ResourceManager négocie un container pour l'ApplicationMaster et lance l'ApplicationMaster.
- ➌ L'ApplicationMaster démarre et s'enregistre auprès du ResourceManager.
- ➍ L'ApplicationMaster négocie des ressources (*resource containers*) pour l'application cliente.
- ➎ Le NodeManager lance des containers pour l'application.
- ➏ Lors de l'exécution, le client informe l'ApplicationMaster de la progression de l'exécution. Une fois l'exécution terminée, l'ApplicationMaster s'arrête et libère les containers associés à l'application.

## Chapitre 7 - Systèmes NoSQL : calcul distribué avec MapReduce

- MapReduce
- Le Map/Reduce de Hadoop v1
- **Gestion des pannes**
- Pig et Hive, des langages de haut niveau

## Gestion des pannes

Un traitement MapReduce peut durer des jours, sur des centaines de machines : la probabilité de panne est très grande.

Ré-exécuter le traitement complètement en cas de panne ⇒ on est à peu près sûr de ne jamais terminer.

Dans Hadoop, le Maître (*JobTracker*) surveille l'ensemble du processus (tâches de Map, tâches de Reduce).

- ➊ Panne d'un Reducer : on peut reprendre à partir du résultat des Mappers, stocké sur disque.
- ➋ Panne d'un Mapper : on recommence la tâche sur le même fragment (ou sur un réplica)
- ➌ Si le Maître tombe en panne : on recommence tout, mais la probabilité est très faible.

### Essentiel

Tout repose sur une **sérialisation** (écriture sur disque) aux différentes étapes.  
**Robuste mais très lent.**

## Chapitre 7 - Systèmes NoSQL : calcul distribué avec MapReduce

- MapReduce
- Le Map/Reduce de Hadoop v1
- Gestion des pannes
- Pig et Hive, des langages de haut niveau

## Pig et Hive, des langages de haut niveau

## Chapitre 8 - Systèmes NoSQL et transactions

- Rappel
  - Notion de transaction
  - Le protocole 2 Phase Commit
- NoSQL et transactions
- Atomicité et transactions dans MongoDB

## Transactions

**Problème** : Nécessité d'annuler des actions sur les données, suite à une panne ou autre

**Exemple** : virement bancaire de 100DT d'un compte A à un compte B

- 2 actions : débiter A de 100 et créditer B de 100
- A est débité, mais une panne empêche B d'être crédité
  - ⇒ la mise-à-jour de A doit être annulée!!
  - ⇒ il est nécessaire de savoir avec quelles autres actions la mise-à-jour de A forme une séquence d'opérations indivisible (=**transaction**)

# Transactions

## Transaction

- Séquence d'actions d'un utilisateur ou d'une application qui doivent être exécutées, soit toutes, soit aucune
- Commence par un **BEGIN TRANSACTION** et se termine par un **COMMIT** (validation) ou un **ROLLBACK** (annulation)
- Enchaînement type
  - ① Transaction lancée (**BEGIN TRANSACTION**)
  - ② Les nouvelles images des données modifiées sont stockées dans le segment de table et les anciennes dans le segment d'annulation
  - ③ En cas de **ROLLBACK**, l'ancienne image remplace la nouvelle dans le segment de table

## Transactions ACID

### **Atomicité** : Tout ou rien

- Une transaction effectue toutes ses actions ou aucune. En cas d'annulation, les modifications engagées doivent être défaillantes.

### **Cohérence** : Intégrité des données

### **Isolation** : Pas d'interférence entre transactions

- Les résultats des modifications des transactions ne sont pas visibles par les autres transactions qu'après sa validation

### **Durabilité** : L'enregistrement des données modifiées doit être garanti même en cas de pannes

- Journalisation des mises-à-jour (Fichiers Log ou Redo Log)
- Les modifications validées sont rejouées à partir des logs en cas de perte.

## 2 PC

**Transaction répartie** : les opérations de la transaction ont lieu sur des **sites distants**

Exemple : virement d'une banque A à une banque B

Problème : **comment garantir l'atomicité** (si l'une des 2 opérations échoue, l'autre doit être annulée)?

Protocoles les plus utilisés : validation à 2 phases (atomicité) et MVCC (isolation).

Hypothèses du protocole 2 PC

- Il existe un site coordinateur (*Coordinator*), les autres sites sont des participants (*Cohorts*)
- Chaque participant journalise ses opérations et génère ses propres données d'annulation.
- La panne sur un site est temporaire

# 2 PC

Principe : 2 phases, une de vote et une de validation

## Phase de vote

- 1 Le coordinateur envoie les opérations et une demande de validation à chacun des participants et attend sa réponse.
- 2 Chaque participant exécute ses opérations. Il génère les données d'annulation et les log et les stocke localement.
- 3 Lorsqu'un participant termine ses opérations avec succès, il envoie un acquittement au coordinateur (le participant vote par oui à la validation globale). Sinon il envoie un message de demande d'annulation (vote négatif)



## 2 PC

### Phase de validation en cas de succès (tous les participants ont voté Oui)

- ① Le coordinateur envoie un message commit à tous les participants
- ② Chaque participant valide ses opérations, libère ses verrous et envoie un acquittement au coordinateur
- ③ Le coordinateur termine la transaction lorsqu'il reçoit les acquittements de tous les participants

### Phase de validation en cas d'échec (au moins un des participants a voté Non)

- ① Le coordinateur envoie un message Rollback à tous les participants
- ② Chaque participant annule ses opérations et envoie un acquittement au coordinateur
- ③ Le coordinateur termine la transaction lorsqu'il reçoit les acquittements de tous les participants

## Chapitre 8 - Systèmes NoSQL et transactions

- Rappel
- **NoSQL et transactions**
- Atomicité et transactions dans MongoDB

## NoSQL et transactions

### Transactions et jointures : très lentes dans un contexte réparti

Systèmes NoSQL : **sacrifient la jointure (par imbrication redondante) et les transactions au profit des performances.**

Les systèmes NoSQL ne sont pas bons pour gérer les transactions et les jointures<sup>8</sup>  
(Rappelez-vous de l'implantation de la jointure avec MapReduce!)

Il est toujours possible de gérer des **transactions** dans un système NoSQL, mais c'est généralement **à la charge de l'application cliente** et non du SGBD NoSQL

Il n'existe pas de moyens communs à tous les systèmes NoSQL pour la gestion des transactions : certains ne le permettent pas, certains le permettent<sup>9</sup>, etc.

Etude du cas MongoDB : pas de gestion des transactions, mais quelques pattern pour les simuler

<sup>8</sup>Si elles sont fréquentes, il vaut mieux envisager un système relationnel

<sup>9</sup>essentiellement en utilisant le protocole MVCC

## Chapitre 8 - Systèmes NoSQL et transactions

- Rappel
- NoSQL et transactions
- **Atomicité et transactions dans MongoDB**
  - Atomicité
  - Pattern pour la simulation du 2PC
  - Concurrence d'accès, versionnement et pattern Update if Current

## Atomicité des écritures dans MongoDB

Écriture : insert, update, upsert, remove, update

Une écriture dans MongoDB est

- **atomique sur un document**
- **non atomique sur un ensemble de documents non imbriqués** (*multi-documents transactions*)

Exemple

- Un document film imbrique son metteur en scène et ses artistes (documents imbriqués ou *embedded*)
- Atomicité au niveau document : une mise-à-jour d'un document film modifiant le film et son metteur en scène ne réussit que si le film et le metteur en scène sont tous les 2 effectivement modifiés.
- Pas d'atomicité sur un ensemble de documents : si l'update d'un film échoue, les modifications des films affectés par le même update ne sont pas annulées (pas de *rollback*)

L'atomicité au niveau document est suffisante dans la plupart des cas (compte tenu de l'imbrication)

Dans les cas où elle n'est pas suffisante, l'application cliente se doit d'implanter l'atomicité.

## Pattern pour la simulation du 2PC : organisation des données

Exemple : nous voulons effectuer un virement bancaire d'un compte *source* à un compte *destination*

Nous disposons de 2 collections

- accounts : stocke les informations sur les comptes bancaires

```
db.accounts.insert(  
  [  
    { _id: "A", balance: 1000, pendingTransactions: [] },  
    { _id: "B", balance: 1000, pendingTransactions: [] }  
  ]  
)
```

- transactions : stocke les informations sur les mouvements de fond

```
db.transactions.insert(  
  { _id: 1, source: "A", destination: "B", value: 100, state: "initial", lastModif:  
  })
```

state : reflète l'état courant de la transaction. Nous supposons qu'il peut prendre comme valeurs : "initial", "pending", "applied", "done", "canceling", et "canceled".

## Pattern pour la simulation du 2PC : organisation des données

### Étape 1 : chercher les transactions à démarrer

```
var t = db.transactions.findOne( { state: "initial" } )
```

### Étape 2 : modifier l'état de la transaction à "pending"

```
db.transactions.update(
  { _id: t._id, state: "initial" },
  {
    $set: { state: "pending" },
    $currentDate: { lastModified: true }
  }
)
```

### Étape 3 : mettre-à-jour les 2 comptes (et indiquer qu'il y a une transaction en-cours)

```
db.accounts.update(
  { _id: t.source, pendingTransactions: { $ne: t._id } },
  { $inc: { balance: -t.value }, $push: { pendingTransactions: t._id } }
)

db.accounts.update(
  { _id: t.destination, pendingTransactions: { $ne: t._id } },
  { $inc: { balance: t.value }, $push: { pendingTransactions: t._id } }
)
```

- pendingTransactions: { \$ne: t.\_id }: condition d'application de la mise-à-jour. S'assurer que la transaction n'est pas ré-appliquer si cette étape est refaite.

## Pattern pour la simulation du 2PC : organisation des données

**Étape 4 : modifier l'état de la transaction à "applied"** (si la mise-à-jour a bien faite sur les 2 comptes)

```
db.transactions.update(  
    { _id: t._id, state: "pending" },  
    {  
        $set: { state: "applied" },  
        $currentDate: { lastModified: true }  
    }  
)
```

**Étape 5 : retirer de la transaction de la liste des transactions en attente de chacun des comptes**

```
db.accounts.update(  
    { _id: t.source, pendingTransactions: t._id },  
    { $pull: { pendingTransactions: t._id } }  
)  
db.accounts.update(  
    { _id: t.destination, pendingTransactions: t._id },  
    { $pull: { pendingTransactions: t._id } }  
)
```

**Étape 6 : Modifier l'état de la transaction à "done"**

```
db.transactions.update(  
    { _id: t._id, state: "applied" },  
    {  
        $set: { state: "done" },  
        $currentDate: { lastModified: true }  
    }  
)
```

## Pattern pour la simulation du 2PC : annulation

L'annulation doit se faire lorsque

- la mise-à-jour n'a pas aboutit sur un des comptes (e.g. un des comptes "disparaît" durant la mise-à-jour (pas d'isolation), timeout atteint (test avec lastModified, etc.))
- L'application cliente la demande explicitement.

Deux cas, selon que la transaction soit à l'état "applied"/"done" ou "pending"

### La transaction est à l'état "applied" ou "done"

- Pas d'annulation proprement dite
- Création d'une nouvelle transaction où les valeurs des champs "source" et "destination" sont inversées

## Pattern pour la simulation du 2PC : annulation

**La transaction est à l'état "pending"** (e.g. la modification a eu lieu sur un compte, mais pas sur l'autre ou c'est l'application cliente qui le demande)

- **Étape 1 : modifier l'état de la transaction à "canceled"**

```
db.transactions.update(  
  { _id: t._id, state: "pending" },  
  {  
    $set: { state: "canceling" },  
    $currentDate: { lastModified: true }  
  }  
)
```

- **Étape 2 : annuler la mise-à-jour sur les 2 comptes** (la condition

pendingTransaction=t.\_id permet de savoir si la modification a été ou non appliquée)

```
db.accounts.update(  
  { _id: t.destination, pendingTransactions: t._id },  
  {  
    $inc: { balance: -t.value },  
    $pull: { pendingTransactions: t._id }  
  }  
)
```

```
db.accounts.update(  
  { _id: t.source, pendingTransactions: t._id },  
  {  
    $inc: { balance: t.value},  
    $pull: { pendingTransactions: t._id }  
  })
```

## Pattern pour la simulation du 2PC : annulation

- **Étape 3 : modifier l'état de la transaction à "canceled"**

```
db.transactions.update(  
  { _id: t._id, state: "canceling" },  
  {  
    $set: { state: "cancelled" },  
    $currentDate: { lastModified: true }  
  }  
)
```

Exercice : trouvez un moyen simple pour annuler les mises-à-jour sur les comptes (sans avoir à faire l'opération inverse sur les soldes des comptes)

## Concurrence d'accès

**Concurrence d'accès** : deux applications **accèdent en écriture** en même temps **au même document**

Si elle n'est pas gérée, cette situation peut engendrer des incohérences

Exemple : l'application qui enregistre le document en dernier l'emporte (perte de mise-à-jour)

Dans la plupart des systèmes **NoSQL la gestion de la concurrence d'accès est à la charge de l'application cliente**

Deux méthodes pour la gestion de la concurrence : **versionnement** et pattern ***update if current***

## Versionnement

### Adjonction d'un numéro de version à chaque document (**Version Stamp**)

Chaque fois qu'un document est modifié, son numéro de version est incrémenté

Lorsqu'une application lit un document, elle sauvegarde son numéro de version dans une variable.

**Avant d'enregistrer une modification** l'application vérifie que le document n'a pas changé (i.e. : n'a pas reçu un nouveau numéro de version) entre le moment de la lecture et celui de l'écriture

Si tel n'est pas le cas, il revient à l'application de résoudre le conflit en écriture

Pour la création d'un attribut de versionnement "auto-increment" voir  
<http://docs.mongodb.org/manual/tutorial/create-an-auto-incrementing-field/>

Versionnement : appelé également verrouillage optimiste

## Pattern *Update if Current*

Même principe que le versionnement, mais appliqué avec une **granularité plus fine (le champ)**

L'application récupère le document dans une variable (ex. *myDocument*)

Si lors de l'enregistrement du document, la valeur actuelle d'un champ est différente de celle de *myDocument*, le champ n'est pas mis-à-jour.

De cette manière, **seuls les valeurs des champs non modifiées par une application concurrente sont mis-à-jour**

Que faire pour les champs en cours de modification par une application concurrente?  
**c'est à l'application cliente de décider**

Implantation : voir

<http://docs.mongodb.org/manual/tutorial/update-if-current/>

## Pattern Update if Current

```
var myDocument = db.products.findOne( { sku: "abc123" } );

if ( myDocument ) {
    var oldQuantity = myDocument.quantity;
    var oldReordered = myDocument.reordered;

    var results = db.products.update(
        {
            _id: myDocument._id,
            quantity: oldQuantity,
            reordered: oldReordered
        },
        {
            $inc: { quantity: 50 },
            $set: { reordered: true }
        }
    )

    if ( results.hasWriteError() ) {
        print( "unexpected error updating document: " + toJson(results) );
    }
    else if ( results.nMatched === 0 ) {
        print( "No matching document for " +
            "{ _id: " + myDocument._id.toString() + +
            ", quantity: " + oldQuantity +
            ", reordered: " + oldReordered +
            + " } " );
    };
}
}
```

## Pour aller plus loin...

- "CAP Twelve Years Later: How the 'Rules' Have Changed", E. Brewer, Computer, pp. 23-29, Fev 2012. <http://www.cs.berkeley.edu/~rxin/db-papers/CAP.pdf>
- "NoSQL systems: sharding, replication and consistency", Riccardo Torlone Notes de cours (Università Roma), 2015  
<http://www.dia.uniroma3.it/~torlone/bigdata/L8-NoSQL.pdf>
- "De-mystifying \eventual consistency" in distributed systems", Oracle,  
<http://www.oracle.com/technetwork/products/nosqldb/documentation/consistency-explained-1659908.pdf>
- et beaucoup d'autres lectures intéressantes..