

FORMATION CASSANDRA

XEBIA

Created by Matthieu Nantern / @mnantern

LE PROGRAMME !

1. Tour de table
2. Introduction
3. Architecture de Cassandra
4. Installation et configuration
5. Modèle de données
6. Le driver Java
7. Administration Cassandra

TOUR DE TABLE

- Qui suis-je ?
- Trois attentes sur la formation

INTRODUCTION

CASSANDRA ET LES BASES NOSQL

CASSANDRA ET LES BASES NOSQL

NoSQL ?

CASSANDRA ET LES BASES NOSQL

NoSQL ?

“NoSQL (Not only SQL en anglais) désigne une catégorie de systèmes de gestion de base de données qui n'est plus fondée sur l'architecture classique des bases relationnelles. Il renonce aux fonctionnalités classiques des SGBD relationnels au profit de la simplicité. Les performances restent bonnes en multipliant simplement le nombre de serveurs, solution raisonnable avec la baisse des coûts.”

LES BASES NOSQL

LES BASES NOSQL



...

LES BASES NOSQL

Les bases NoSQL se classent en quatre catégories:

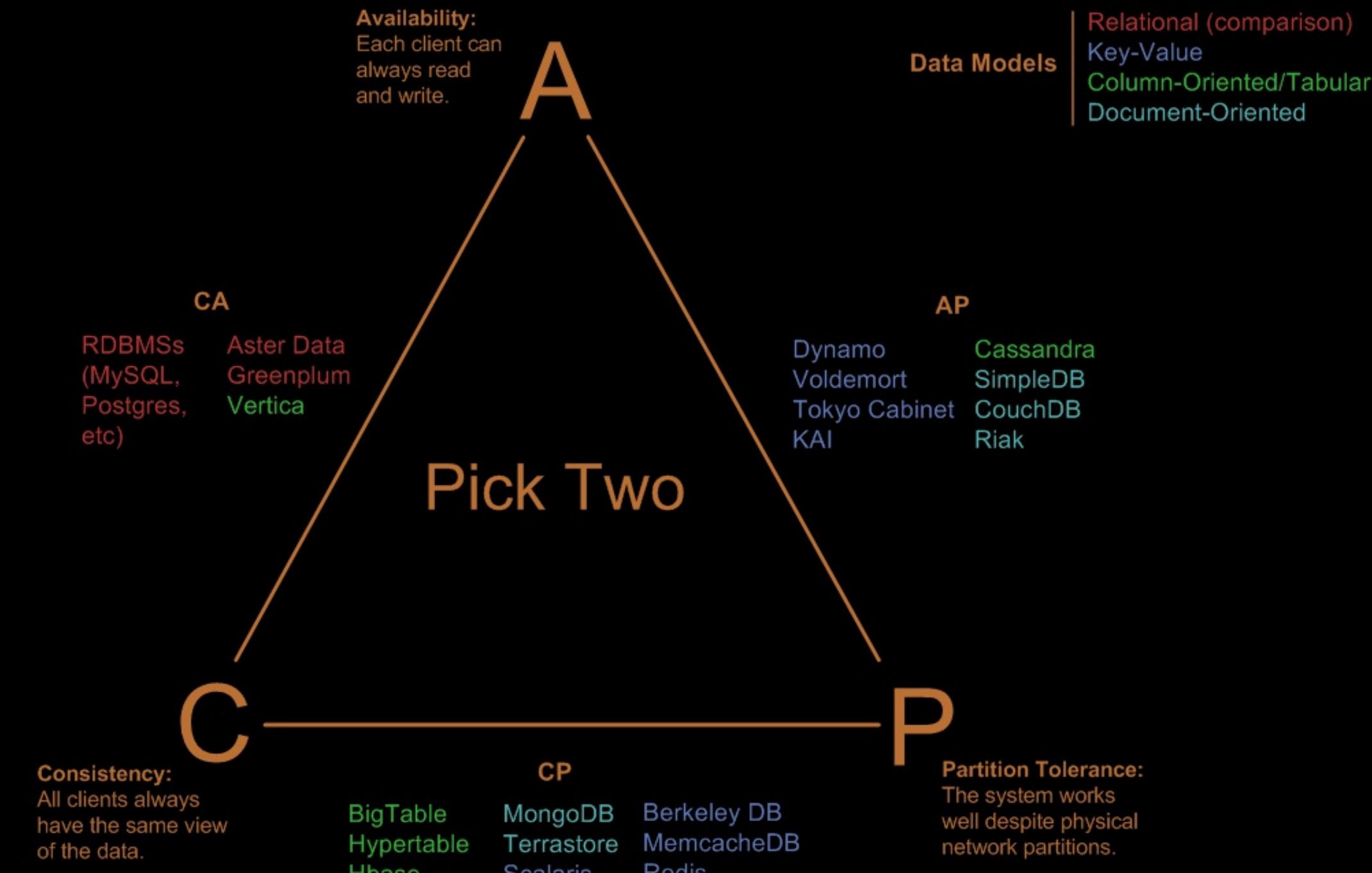
- **Colonnes:** Cassandra, HBase
- **Documents:** MongoDB, ElasticSearch
- **Clés/valeurs:** Redis, Riak, Couchbase
- **Graph:** Neo4j

UN PEU DE THÉORIE: LE THÉORÈME DE CAP

Le théorème CAP également connu sous le nom de théorème de Brewer montre qu'il est impossible pour un système distribué de satisfaire de manière simultanée les trois contraintes suivantes :

- **Cohérence (“Consistency”)** : tous les noeuds du système voient la même donnée au même moment;
- **Disponibilité (“Availability”)** : chaque requête recevra une réponse (que cela soit un succès ou un échec);
- **Résistance à la partition (“Partition tolerance”)** : le système continue de fonctionner malgré des défaillances pouvant aller jusqu'à la séparation du cluster en sous-système.

Visual Guide to NoSQL Systems



CASSANDRA

- Technologies de base:
 - Google BigTable : modèle de stockage
 - Amazon Dynamo : modèle de réPLICATION
- Historique:
 - 2008 : libération par Facebook
 - 2010 : Top Level Project Apache
 - Octobre 2011 : version 1.0
 - Septembre 2013 : version 2.0
 - Décembre 2015 : version 3.x

QUAND UTILISER CASSANDRA ?

- Pas de SPOF (Single point of failure)
- Réplication native entre serveurs et datacenters
- Scalabilité linéaire
- Beaucoup d'écriture et de lecture

QUAND NE PAS UTILISER CASSANDRA ?

- Besoin de transaction avec rollback
- ??

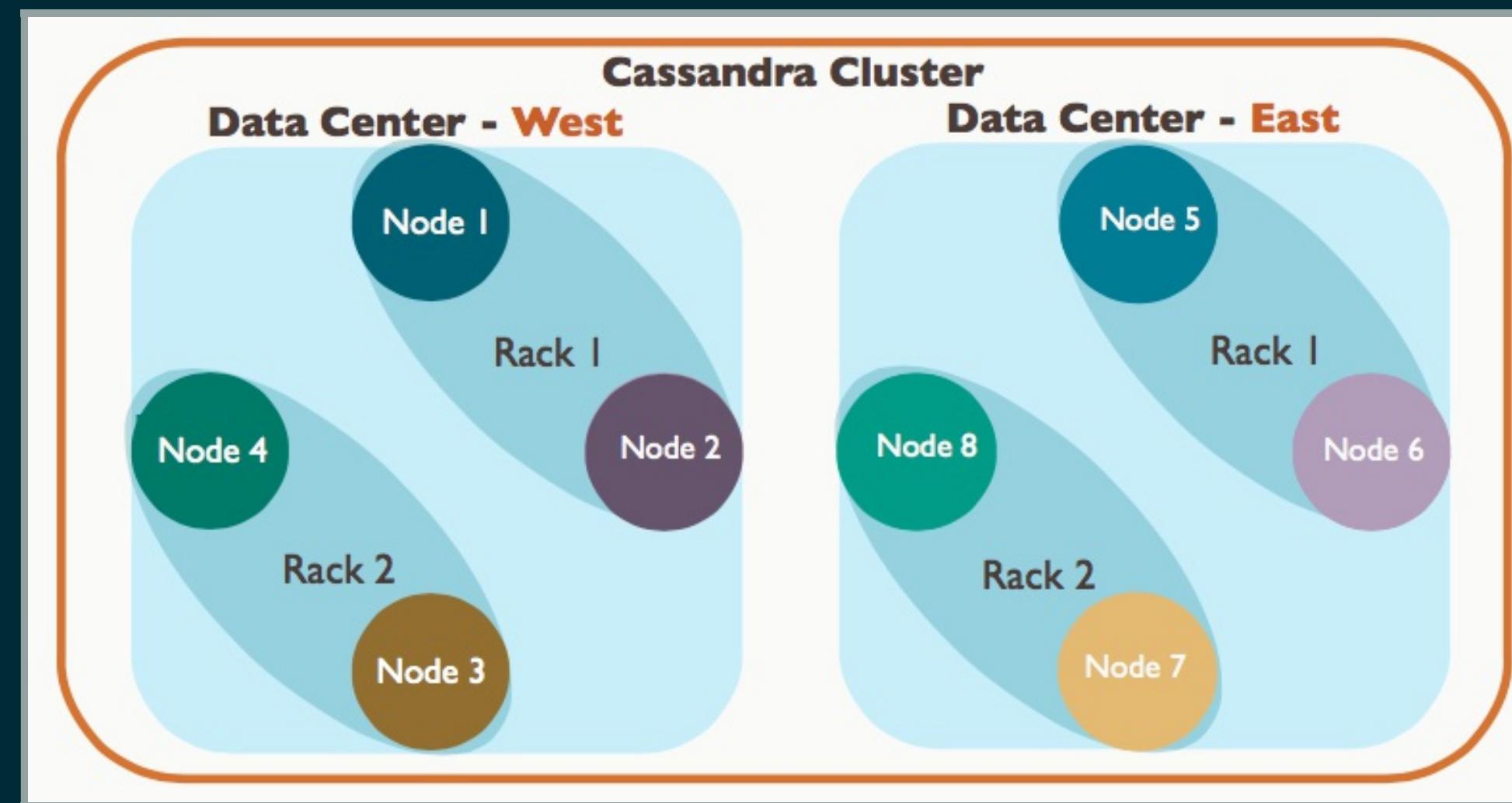
LES CAS D'UTILISATION CLASSIQUES DE CASSANDRA

- Time series / IoT
- Messaging
- Recommandation
- Catalogue produit / Playlists

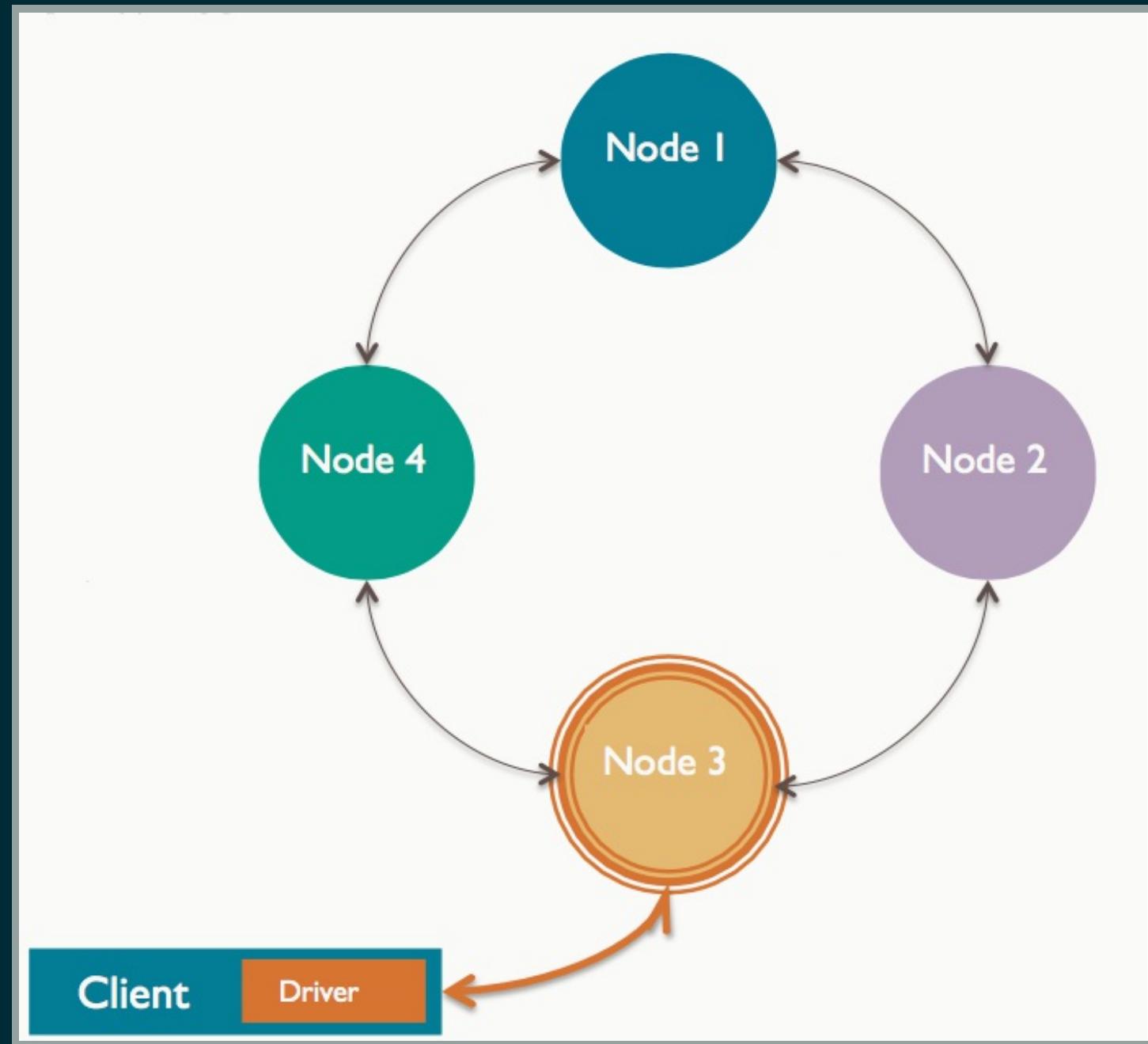
liste de cas d'utilisation C*

ARCHITECTURE DE CASSANDRA

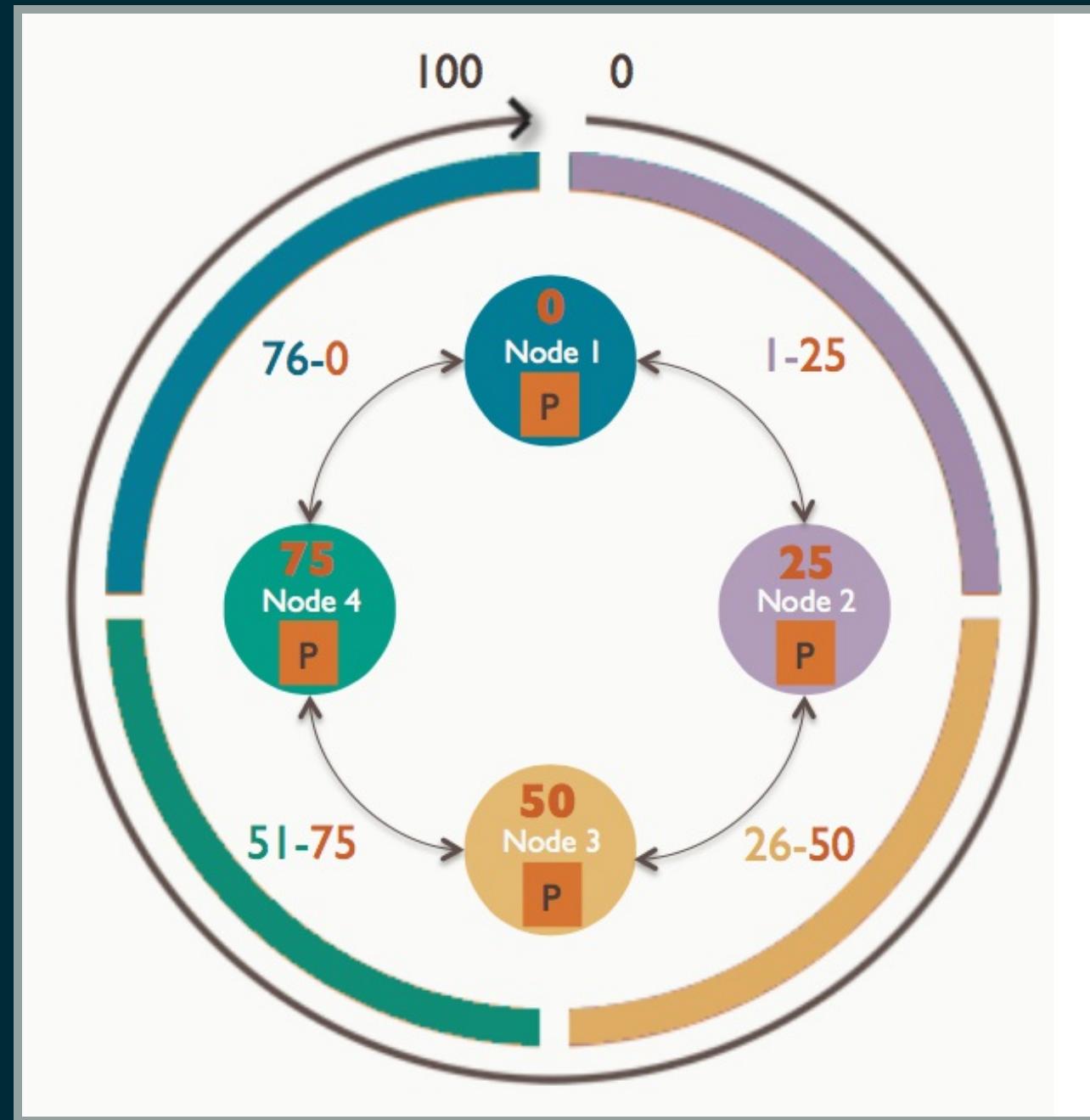
LE CLUSTER



LE COORDINATOR



LE PARTITIONNEMENT



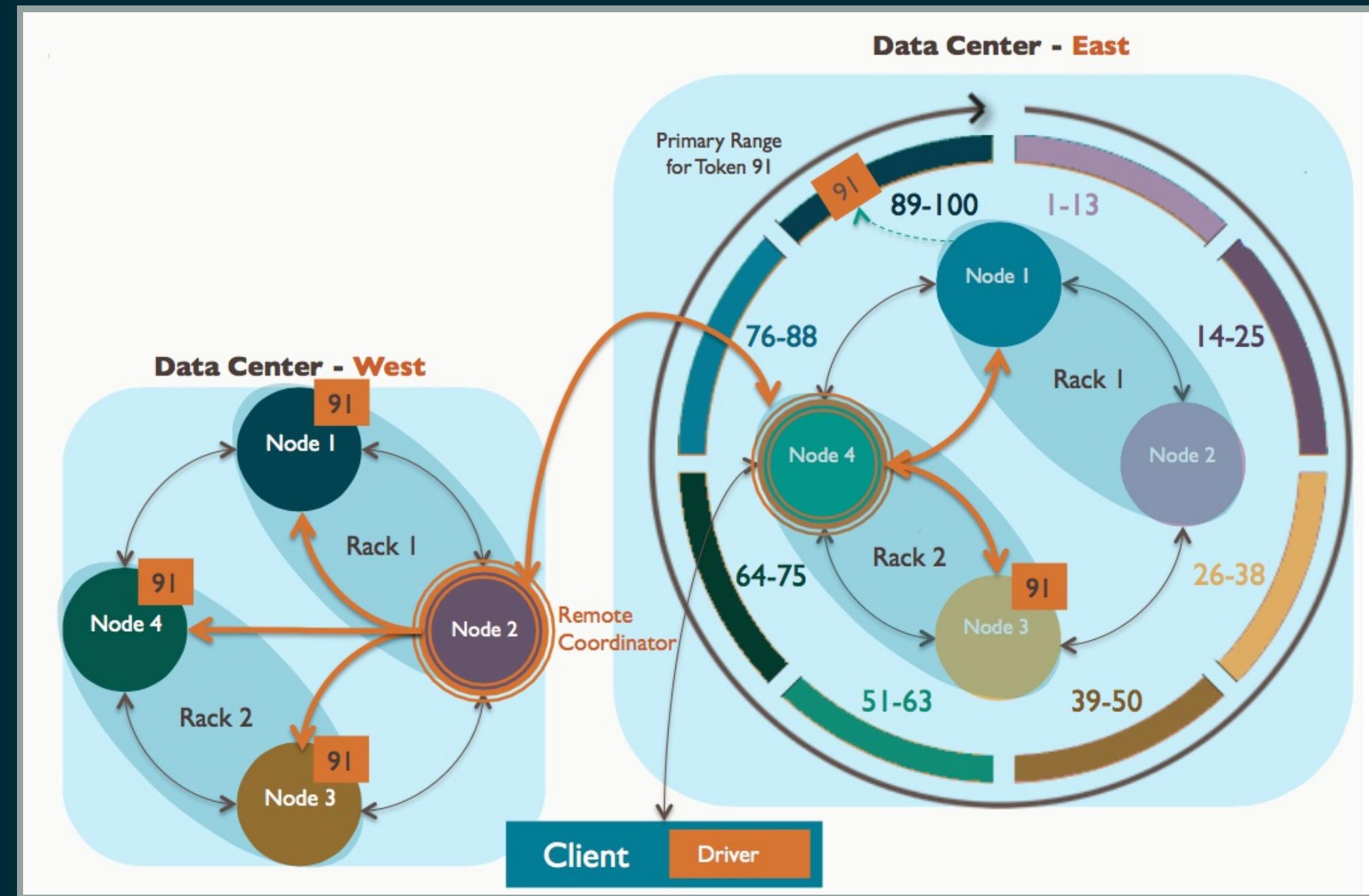
LA RÉPLICATION (1/2)

Deux éléments influent sur la réPLICATION:

1. **Le facteur de réPLICATION**: indique le nombre de fois qu'une donnée doit être répliquée sur le cluster
2. **La stratégie de réPLICATION**: indique quelle machine doit stocker la donnée

```
CREATE KEYSPACE Demo  
WITH REPLICATION = {  
    'class' : 'NetworkTopologyStrategy',  
    'dc-east' : 2,  
    'dc-west' : 3  
};
```

LA RÉPLICATION (2/2)



LA COHÉRENCE

La cohérence d'une requête définit le nombre de noeuds devant répondre à une requête avant que celle-ci ne soit considérée comme terminée par le cluster Cassandra.

Dans le cadre d'**une lecture** cela définit le nombre de noeud devant envoyer la copie la plus récente de la donnée.

Dans le cadre d'**une écriture** cela définit le nombre de noeud devant confirmer avoir écrit la donnée.

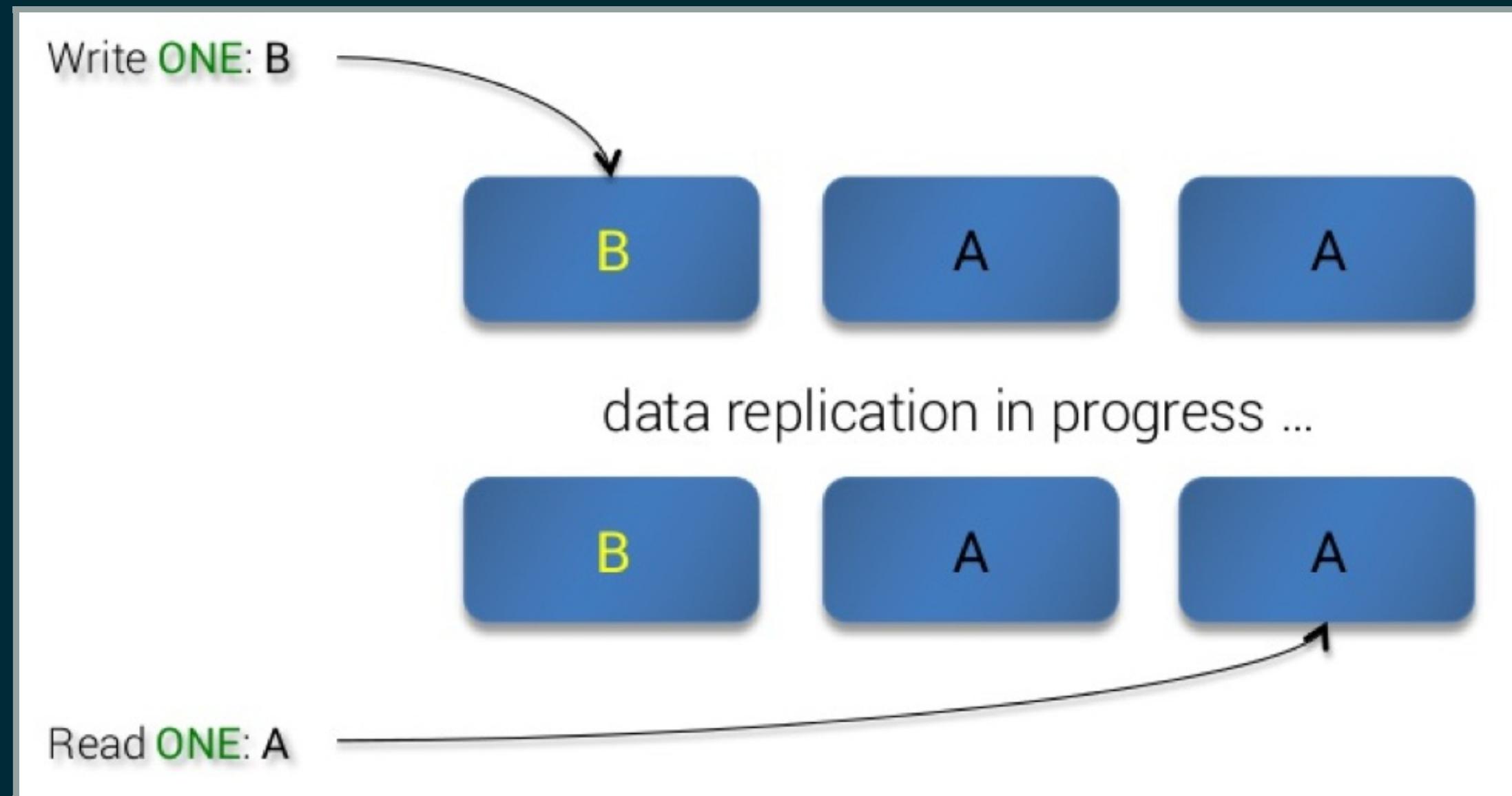
La cohérence est choisie par requête dans Cassandra.

LES DIFFÉRENTS NIVEAUX DE COHÉRENCE

Nom	Description
ONE	Attend la réponse d'un noeud avant de répondre au client.
QUORUM	Attend la réponse de RF/2+1 avant de répondre au client.
ALL	Attend la réponse de tous les noeuds avant de répondre au client.

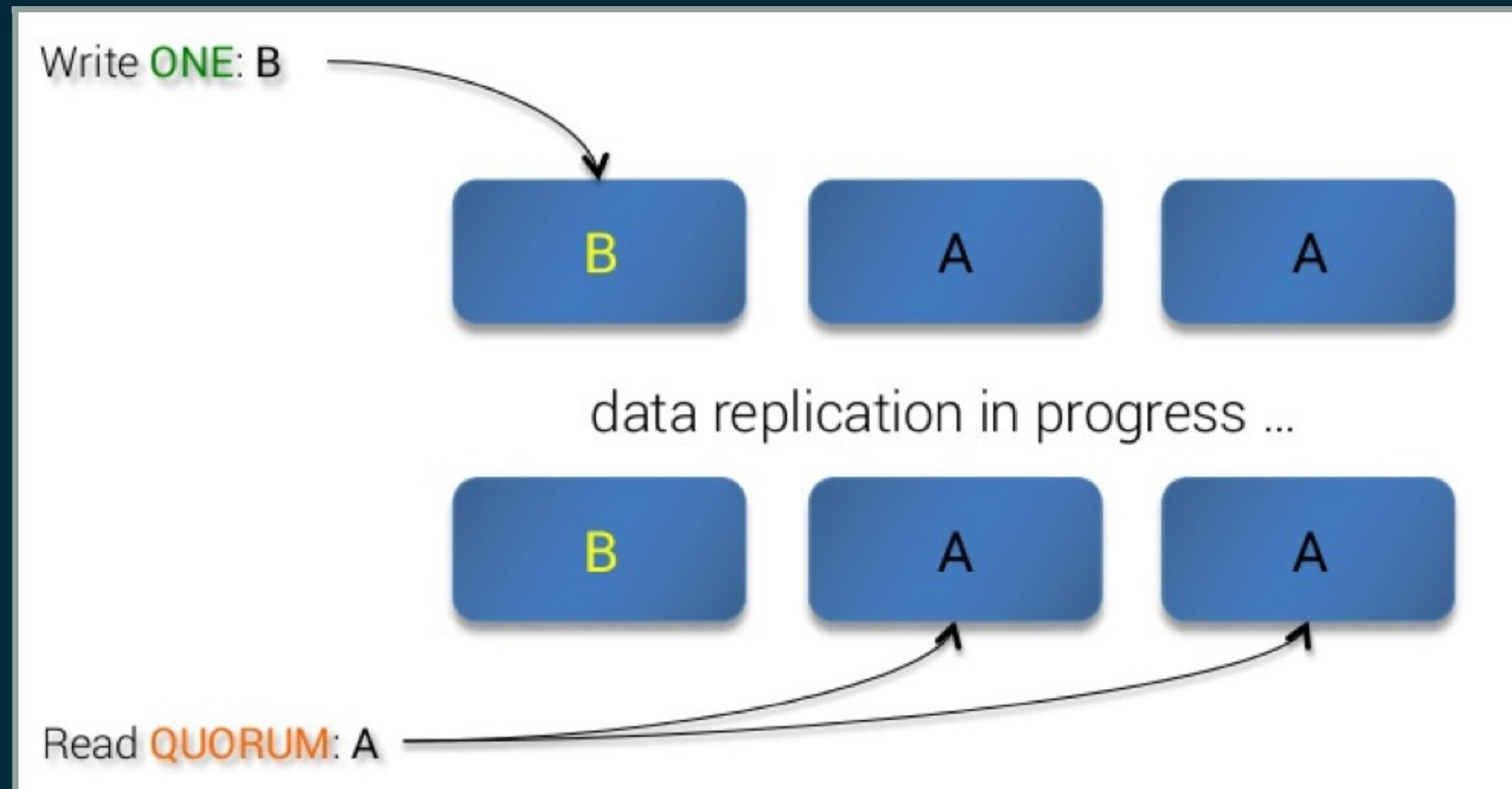
LA COHÉRENCE EN ACTION

RF = 3, Write ONE, Read ONE



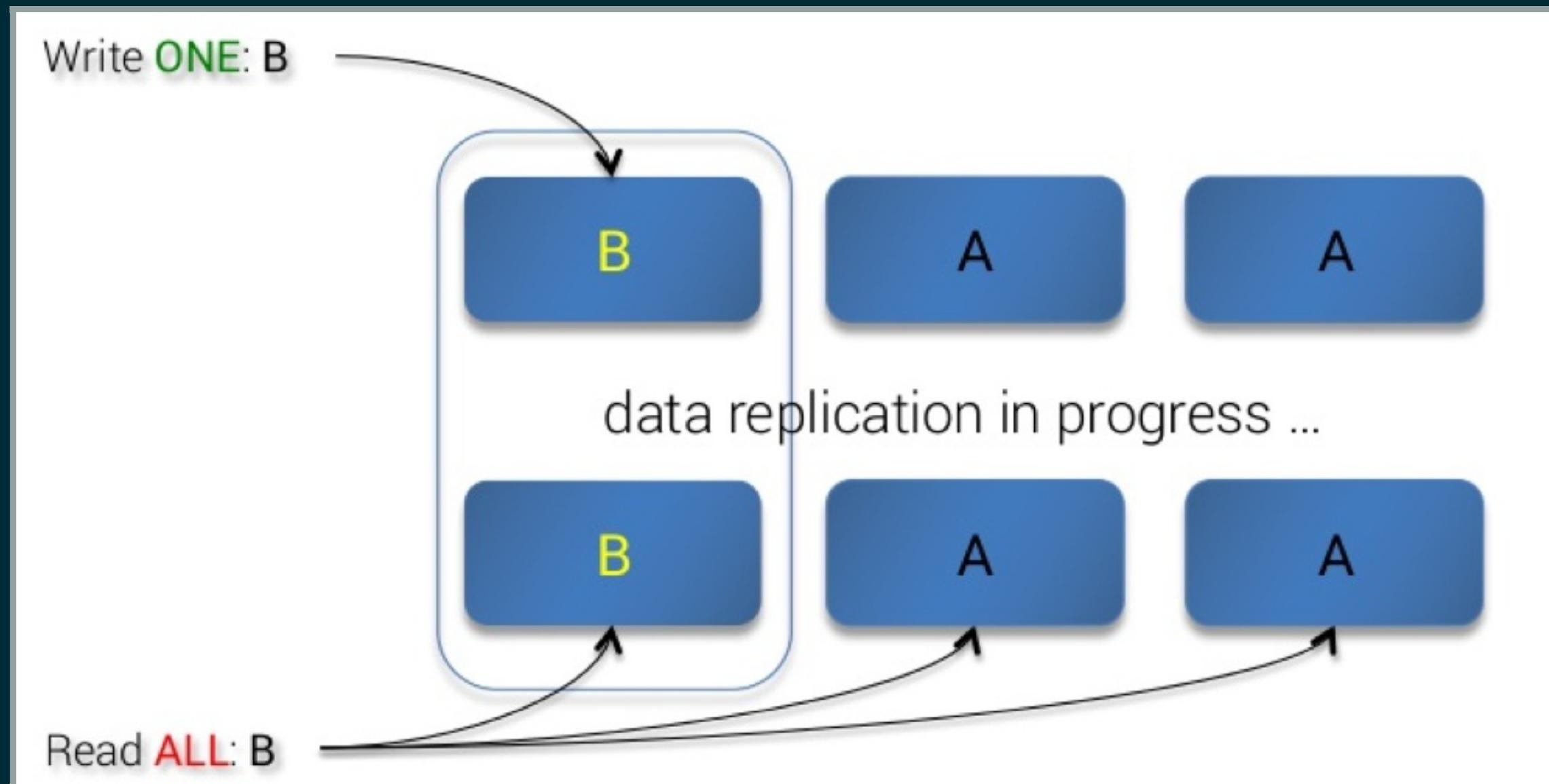
LA COHÉRENCE EN ACTION

RF = 3, Write ONE, Read QUORUM



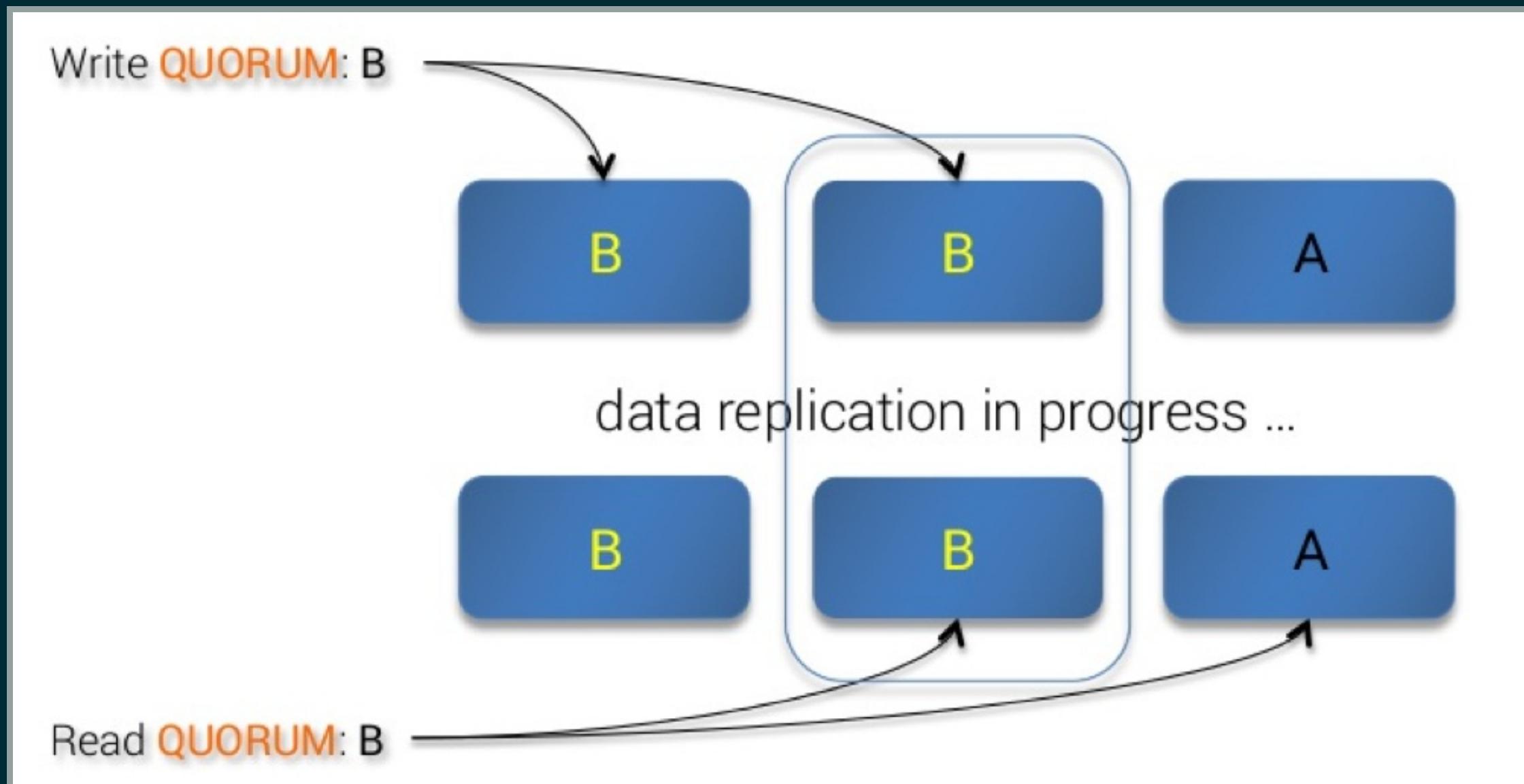
LA COHÉRENCE EN ACTION

RF = 3, Write ONE, Read ALL



LA COHÉRENCE EN ACTION

RF = 3, Write QUORUM, Read QUORUM



LES USAGES DES NIVEAUX DE COHÉRENCE (1/2)

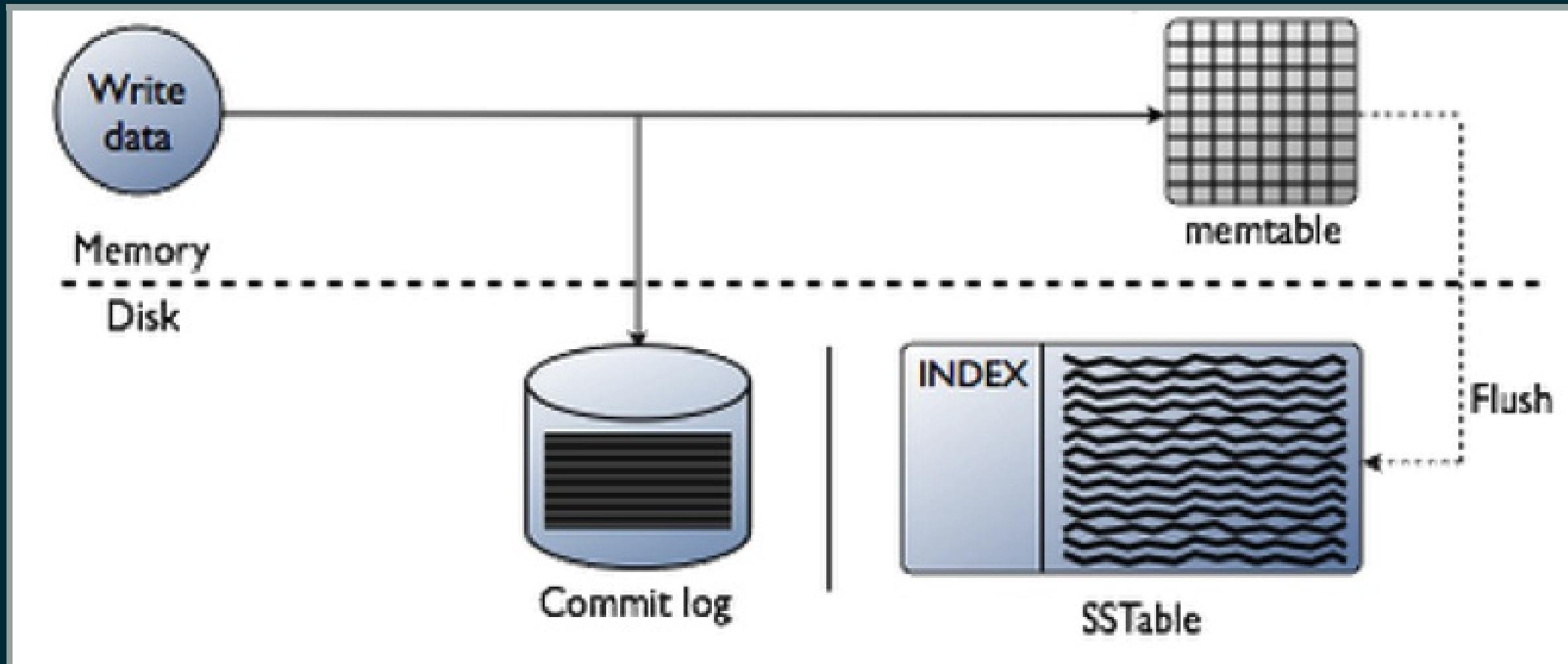
Nom	Usage
ONE	Rapide. Très bonne disponibilité. Ne lit pas forcément la toute dernière valeur.
QUORUM	Bon compromis entre la cohérence et la rapidité.
ALL	Plus haut niveau de cohérence. Très faible disponibilité.

LES USAGES DES NIVEAUX DE COHÉRENCE (2/2)

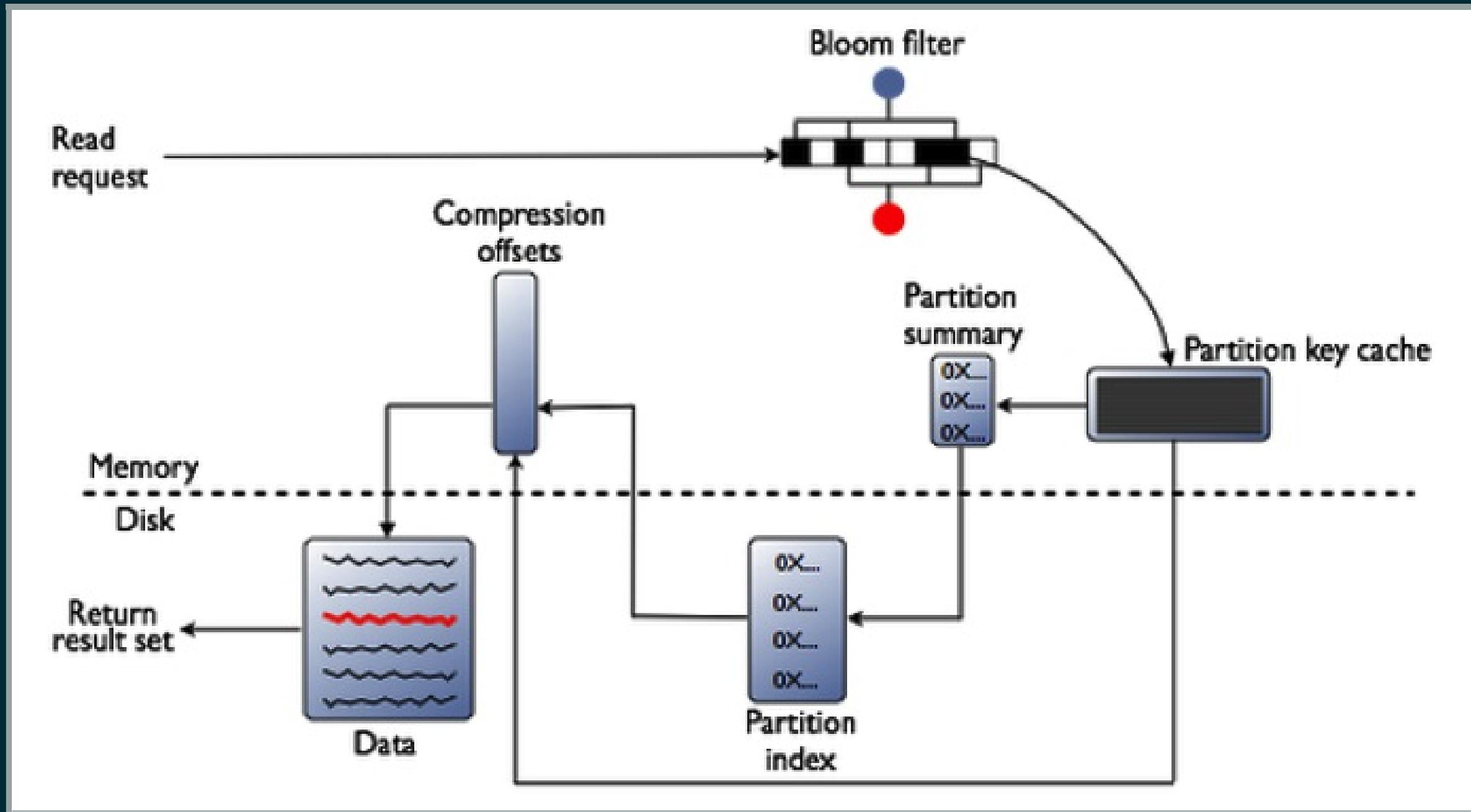
Les deux niveaux les plus couramment utilisés sont:

- **ONE read + ONE write**: rapide et fonctionne même avec RF - 1 noeuds indisponibles
- **QUORUM read + QUORUM write**: consistent. On lit toujours la dernière valeur insérée.

CASSANDRA WRITE PATH



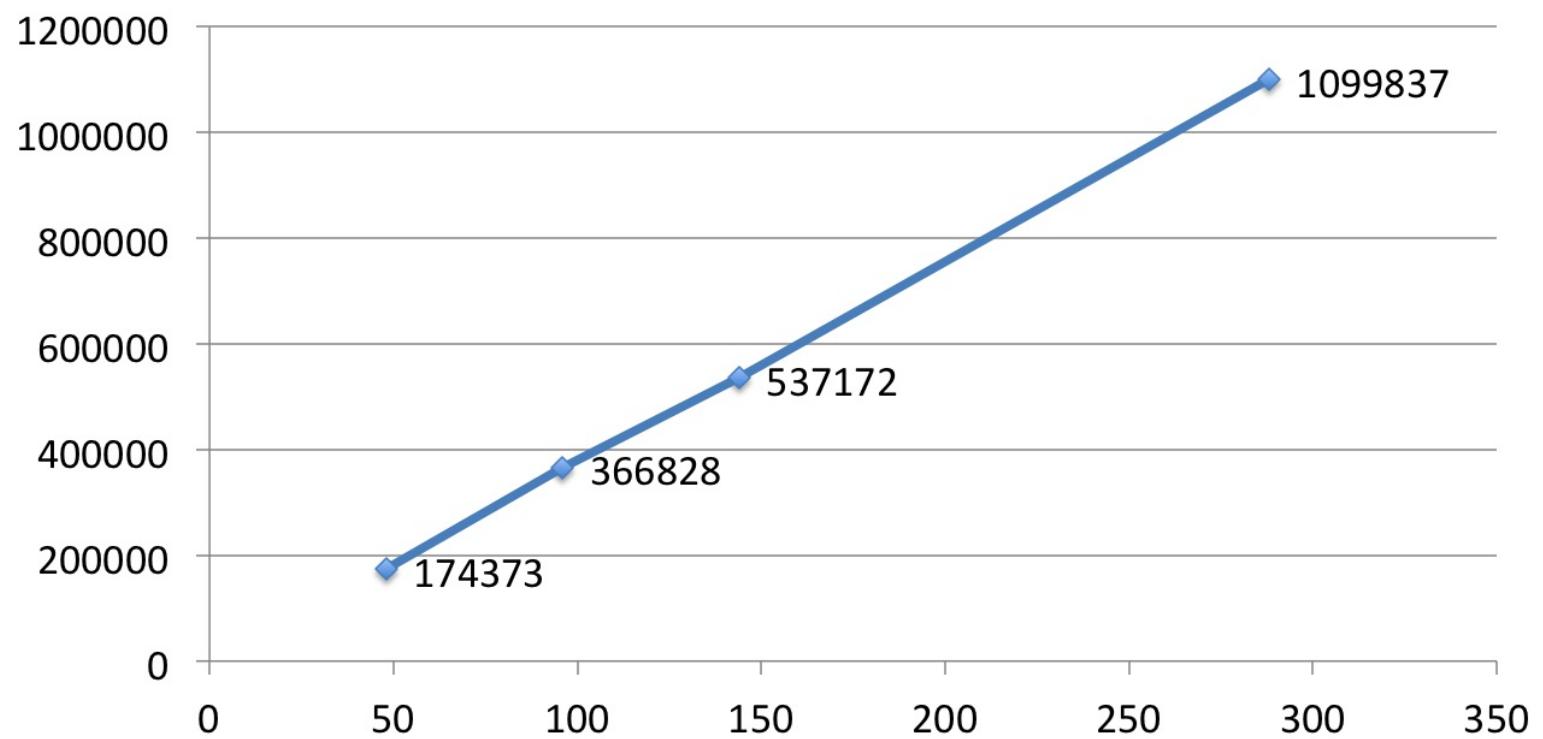
CASSANDRA READ PATH



LA SCALABILITÉ

Scale-Up Linearity

Client Writes/s by node count – Replication Factor = 3



NETFLIX

INSTALLATION ET CONFIGURATION

INSTALLATION DE C*

PRÉREQUIS

- Oracle JDK 1.8 64 bits
- Configurer JAVA_HOME
- Sélectionner sa distribution C*:
 - Apache Cassandra
 - Datastax Community Edition
 - Datastax Enterprise

INSTALLATION DE C*

CLONNER LE DÉPÔT PRÉSENT À L'ADRESSE SUIVANTE ET RÉALISER CE QUI EST INDIQUÉ DANS LE
FICHIER **EXERCICES/01_INSTALLATION.MD**

```
git clone https://github.com/mNantern/formation-cassandra.git
```

INSTALLATION DE C*

FICHIERS DE CONFIGURATION

- **cassandra.yaml**: fichier principal de configuration
- **cassandra-env.sh**: configuration de l'environnement Java (heap size,...)
- **logback.xml**: configuration des logs

LE FICHIER CASSANDRA.YAML, PRINCIPALES PROPRIÉTÉS

- **cluster_name**: toutes les machines d'un cluster doivent avoir le même nom
- **listen_address**: le port sur lequel C* écoute les autres noeuds

INSTALLATION DE C*

ON DÉMARRE !

1. Vérifier le status du service Cassandra

```
sudo service cassandra status
```

2. Visualiser les logs de C*

```
ls /var/log/cassandra/
```

3. Arrêter le service Cassandra

```
sudo service cassandra stop
```

4. Démarrer le service Cassandra

```
sudo service cassandra start
```

LES OUTILS CASSANDRA

NODETOOL

Nodetool est le couteau suisse de Cassandra. Il permet d'obtenir des informations et de gérer un cluster de machines. Les commandes les plus couramment utilisées sont:

- **status**: fournit des informations sur le cluster (état, charge, ID)
- **info**: fournit des informations sur la machine locale (mémoire, espace disque,...)
- **ring**: affiche un résumé pour chaque noeud dans le cluster. Permet de repérer les machines déséquilibrées. Privilégier plutôt status et info

NODETOOL

1. Explorer la liste des commandes de nodetool

```
nodetool help
```

2. Obtenir de l'aide sur une commande

```
nodetool help <command>
```

3. Tester les commandes status,info et ring

LES OUTILS CASSANDRA

CQLSH

CQLSH est un shell interactif permettant de tester des commandes dans le langage CQL (Cassandra Query Language). Il offre également d'autres commandes uniquement disponibles dans le shell:

- **COPY**: import ou export les données au format CSV
- **DESCRIBE**: fournit des informations sur le cluster, les keyspaces ou les tables
- **TRACING**: active ou désactive le tracing des requêtes
- **SOURCE**: exécute un fichier contenant des requêtes CQL
- Et d'autres...

ON CRÉÉ NOTRE PREMIER KEYSPACE !

1. Affichons la liste des keyspaces (Tab pour l'auto-complétion)

```
cqlsh> DESCRIBE KEYSPACES;
```

2. Création d'un keyspace:

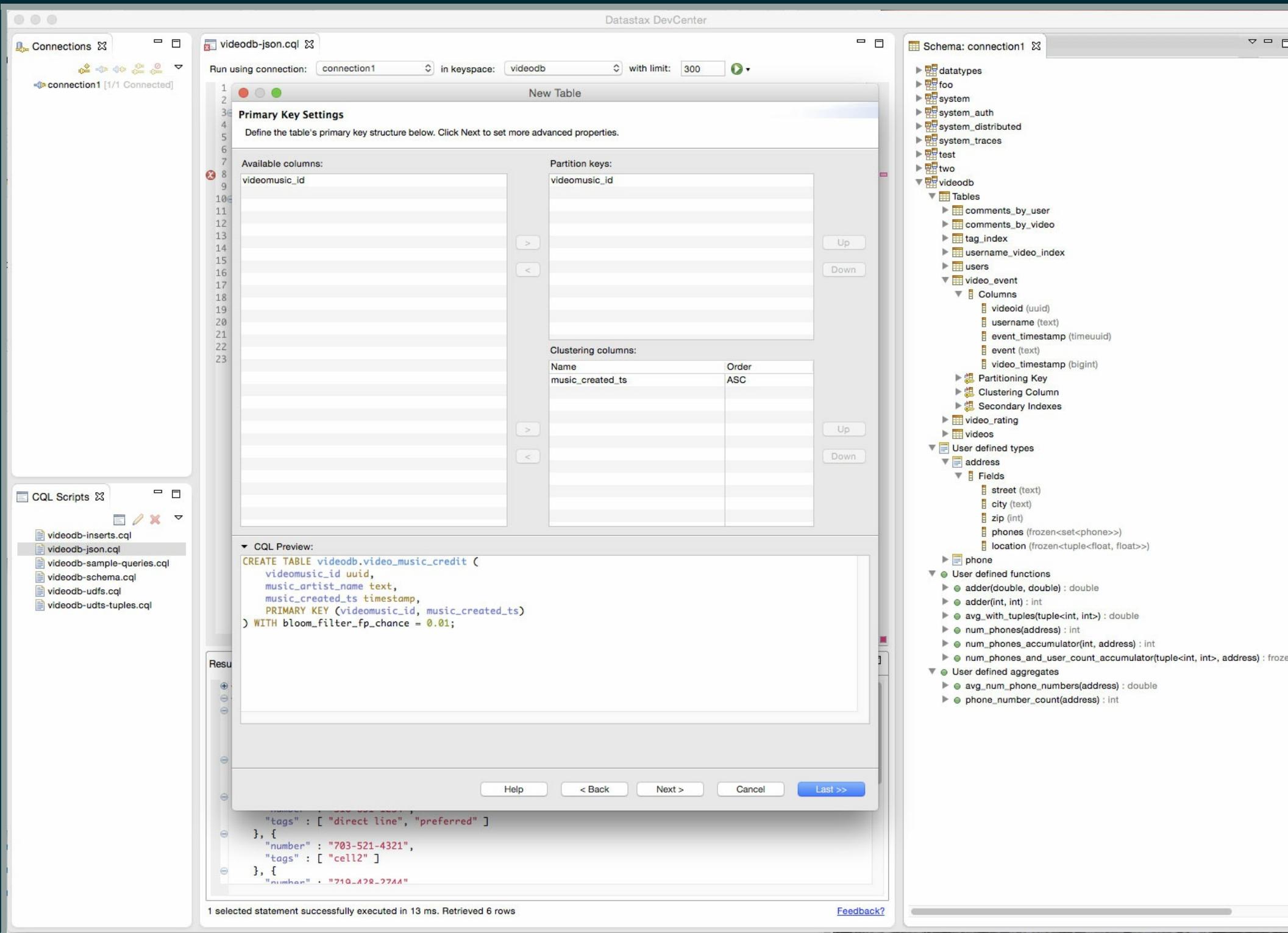
```
CREATE KEYSPACE cassandra-iot
WITH REPLICATION = {
    'class' : 'SimpleStrategy',
    'replication_factor' : 1
};
```

3. Afficher à nouveau la liste des keyspaces et le détail pour notre keyspace cassandra-iot

LES OUTILS CASSANDRA

DEVCENTER

DevCenter est un IDE (basé sur Eclipse) permettant de visualiser les tables et d'exécuter des commandes CQL avec auto-complétion, sauvegarder des requêtes et maintenir plusieurs connections simultanées



MODÈLE DE DONNÉES

PRINCIPES DE MODÉLISATIONS DES DONNÉES

Construire un schéma de base relationnelle passe bien souvent par la description des types d'objets, des attributs et des liens entre les tables. Les requêtes arrivent ensuite.

Pour Cassandra il est primordial de **commencer par les requêtes**. Les tables sont ensuite construites pour supporter ces requêtes. Cela permet d'obtenir un modèle répondant rapidement aux requêtes et scalant correctement.

L'APPLICATION CASSANDRA-IOT

Nous allons développer la partie back d'une application destinée à gérer des objets connectés.

Cette application devra être capable de gérer les données provenant de millions d'objets de manière performante et sans interruption de service.

Notre application devra être capable de déclarer des familles d'objets connectés, de créer un objet connecté, d'envoyer des données sur cet objet et de gérer des comptes utilisateurs.

KEYSPACES

- Niveau le plus élevé
- Contient des tables
- Le paramètre de réPLICATION est obligatoire:

```
CREATE KEYSPACE cassandra-iot  
WITH REPLICATION = {  
    'class' : 'SimpleStrategy',  
    'replication_factor' : 1  
};
```

- Utiliser le mot clé **USE** pour changer de keyspace:

```
USE cassandra-iot;
```

LES TABLES

Les keyspaces contiennent des tables.

Les tables contiennent les données.

```
CREATE TABLE users (
    id UUID PRIMARY KEY,
    first_name TEXT,
    last_name TEXT,
    email TEXT
);
```

LES TYPES DE DONNÉES, ÉPISODE 1

- **text**: une chaîne de caractères encodée en UTF8, équivalent à varchar
- **uuid**: un identifiant unique (UUID de type 4)
- **int**: un entier sur 32 bits signé
- **timeuuid**: un identifiant unique contenant un timestamp (UUID de type 1).
- **timestamp**: millisecondes depuis Epoch

LIRE DES DONNÉES

PREMIÈRE TABLE ET PREMIÈRES REQUÊTES

Réaliser l'exercice [exercices/02_premieres_requetes.md](#)

IMPORTANCE DE LA CLÉ DE PARTITIONNEMENT

Dans une base de données relationnelle son absence implique un "full table scan".

Avec C* cela implique un "full cluster scan".



COMMENT FAIT-ON ALORS ?

INDEX SECONDAIRE

Il existe dans Cassandra un mécanisme d'index secondaires permettant de réaliser ensuite des requêtes sur cet index sans avoir besoin de passer la clé de partitionnement.

=> Retour à l'exercice !

INDEX SECONDAIRE

QUAND NE FAUT-IL PAS LES UTILISER ?

- Sur les colonnes avec une grande cardinalité (un id ou un email par exemple)
- Sur les colonnes avec une très faible cardinalité (un boolean par exemple)
- Sur les tables contenant une grande quantité de données: les index secondaires sont locaux à chaque machine du cluster
- Sur les colonnes où les données sont régulièrement supprimées

INDEX SECONDAIRE

QUAND FAUT-IL LES UTILISER ?

- Sur des colonnes contenant peu de données et une cardinalité moyenne
- Si la requête est restreinte par la clé de partitionnement

Le plus simple est d'oublier les index secondaires.

COMMENT FAIRE ALORS ?

La bonne solution dans notre cas est d'utiliser l'email en tant que clé primaire d'une autre table. La clé primaire est globale au cluster au contraire d'un index secondaire.

Dupliquer les données dans C* ne pose aucun problème, la base est conçue pour cela. Ecrire des données est efficace, utiliser un index secondaire ne l'est pas.

=> Retour à l'exercice !

REMARQUES

La cohérence entre nos tables 'users' et 'users_by_email' doit être **assurée applicativement** lors des insertions/mises à jour et des suppressions. Il est important de ne **dénormaliser que les données qui changent peu.**

Si la duplication des données pose problème (les données sont très volumineuses) ou que les données changent très régulièrement alors il peut être plus performant de ne conserver dans la table 'users_by_email' que l'email et l'id et de faire ensuite une nouvelle requête à C* pour obtenir les autres informations de la table 'users'.

INSÉRER DES DONNÉES

LES INSERTIONS

Réaliser l'exercice [exercices/03_inserer_donnees.md](#)

UPSERT

Qu'est ce qu'il vient de se passer ?

Dans C* le dernier qui écrit a raison: LWW (Last Write Wins)

=> Retour à l'exercice !

QUELLE EST LA DIFFÉRENCE ?

QUELLE EST LA DIFFÉRENCE ?

Il n'y en a pas.

QUELLE EST LA DIFFÉRENCE ?

Il n'y en a pas.

Enfin presque: les compteurs

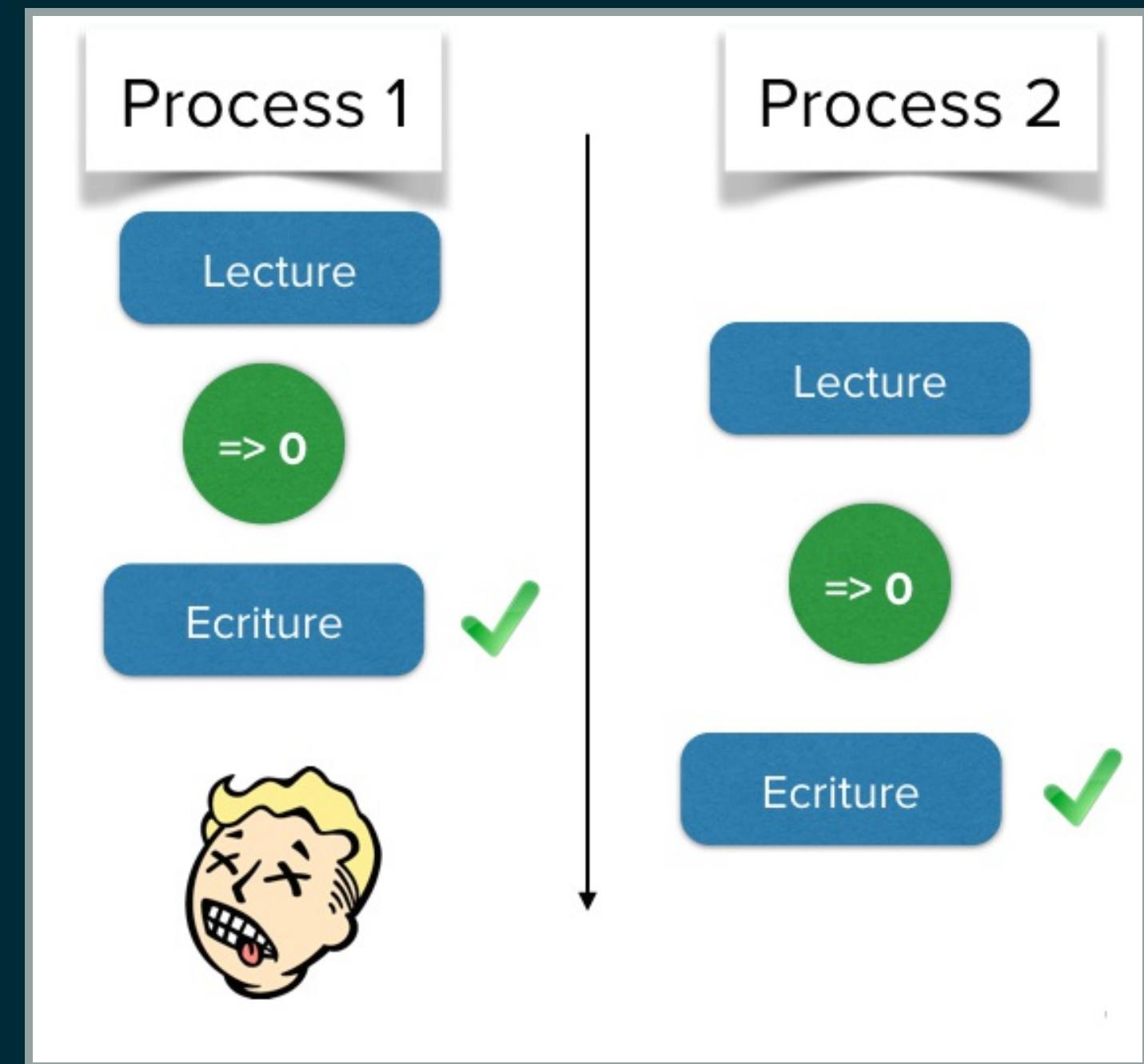
QUELLE EST LA DIFFÉRENCE ?

Il n'y en a pas.

Enfin presque: les compteurs

Mais c'est une autre histoire.

LE PROBLÈME



LA SOLUTION, LES LWT

LIGHTWEIGHT TRANSACTION

Se base sur un algorithme de consensus distribué, Paxos.

Ce mécanisme ajoute le mot clé "IF" dans CQL et permet de réaliser des requêtes de type "IF NOT EXISTS" ou "IF column1=value".

Ce processus est quatre fois plus couteux qu'une insertion classique. Il faut donc l'utiliser uniquement dans les cas où il est nécessaire.

=> Retour à l'exercice !

LA CLÉ PRIMAIRE

CLÉ PRIMAIRE

Jusqu'à présent nous avons vu des tables avec des clés primaires simples. Mais il est possible de créer des clés primaires bien plus évoluées afin de modéliser au mieux les requêtes que l'on a besoin d'exécuter.

Quelques éléments de vocabulaire:

- **clé de partitionnement:** la "première colonne" de notre clé primaire
- **colonne de clustering:** toutes les autres colonnes composant la clé primaire
- **clé de partitionnement composée:** une clé de partitionnement composée de plusieurs colonnes

CLÉ DE PARTITIONEMENT

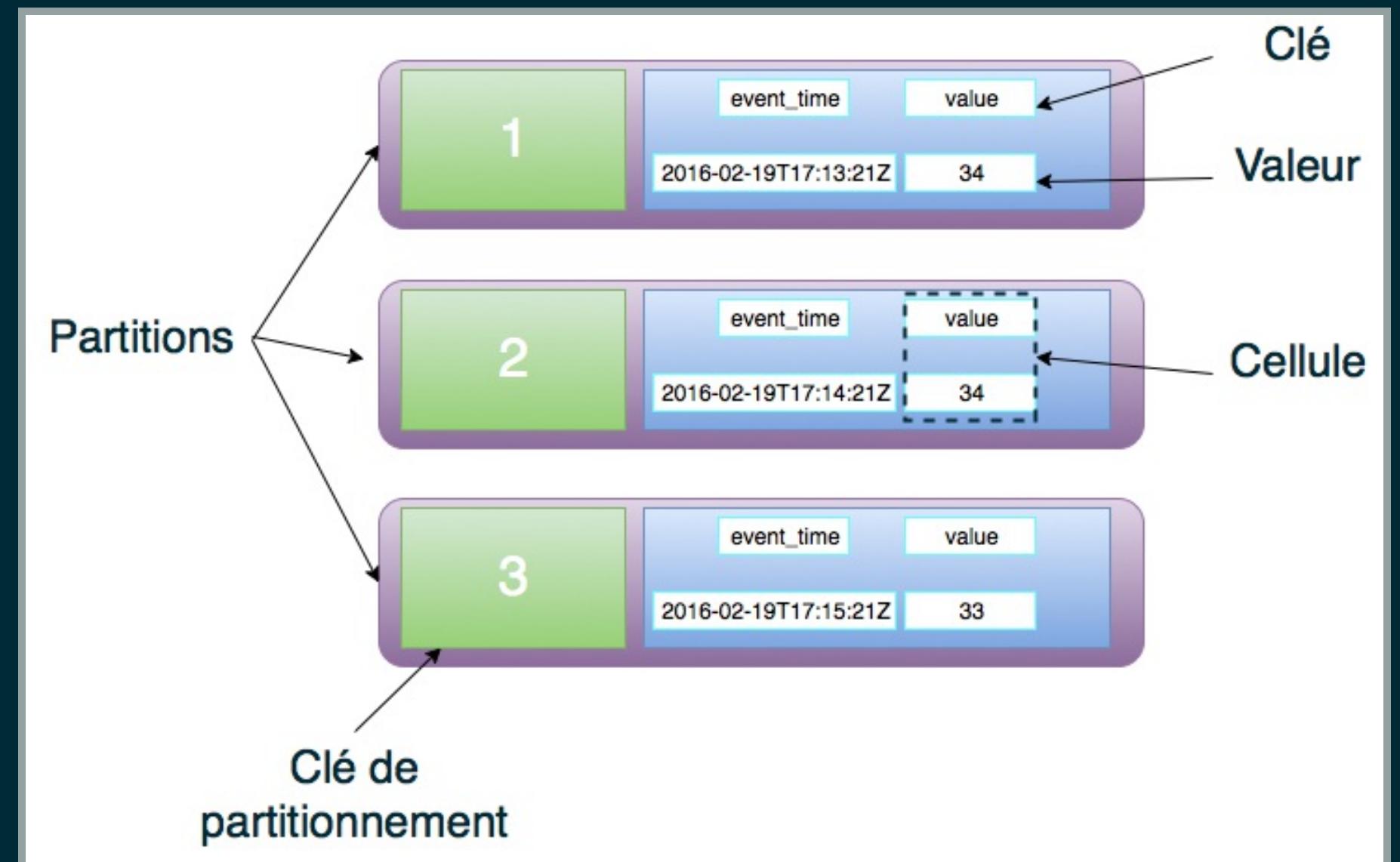
C'est un élément central dans le design de nos tables. Cette clé indique **sur quelle machine de notre cluster les données sont stockées.**

La clé de partitionnement étant obligatoire dans les requêtes réalisées sur C*, il est important de la créer afin qu'elle soit la plus proche possible des requêtes que l'on souhaite effectuer sur notre cluster.

LE DESIGN DE LA CLÉ PRIMAIRE

Réaliser l'exercice [exercices/04_cle_primaire.md](#)

STOCKAGE PHYSIQUE DES DONNÉES



STOCKAGE DES PARTITIONS

- Les partitions sont distribuées sur les différents noeuds du cluster
- Un algorithme de hash permet de déterminer rapidement quel noeud possède la partition recherchée
- Il est possible de faire un 'WHERE' sur la clé de partition mais le faire sur une autre colonne impliquerai de scanner toutes les partitions sur tous les noeuds.



CLUSTERING COLUMNS

Il est possible d'ajouter à la clé de partitionnement d'autres colonnes appelées "Colonne de clustering".

Le clustering est le processus qui trie les données au sein d'une même partition. Comme les données sont triées sur disque il est extrêmement performant de récupérer une partie de la partition (lecture séquentielle sur le disque)

CLUSTERING COLUMNS

On peut alors, grâce au clustering columns réaliser des requêtes de ce type:

```
SELECT * FROM data  
WHERE partition_key=value_partition_key  
AND clustering_column1=value1
```

Voir même des "range requests":

```
SELECT * FROM data  
WHERE partition_key=value_partition_key  
AND clustering_column1 > value1
```

=> Retour à l'exercice !

STOCKAGE PHYSIQUE DES DONNÉES 2

```
CREATE TABLE data (
    id UUID,
    year TEXT,
    value TEXT,
    event_time timestamp,
    PRIMARY KEY (year,id)
);
```

year	id	value	event_time
2016	1	7	2016-02-19T17:13:21Z
2016	2	18	2016-02-25T17:13:21Z
2015	3	2	2015-12-19T07:13:21Z

STOCKAGE PHYSIQUE DES DONNÉES 2



LIMITER LA TAILLE DES PARTITIONS

Le nombre maximal de valeurs par partition est 2 milliards en théorie. La partition doit également tenir sur le disque d'un seul noeud.

En pratique il faut éviter de dépasser quelques centaines de milliers de valeur par partition et quelques centaines de Mo.

=> Retour à l'exercice !

STOCKAGE PHYSIQUE DES DONNÉES 3

```
CREATE TABLE data (
    id UUID,
    smartphone_id UUID,
    year TEXT,
    value TEXT,
    event_time timestamp,
    PRIMARY KEY ((smartphone_id,year),id)
);
```

smartphone_id	year	id	value	event_time
SAMSUNG_1567	2016	1	7	2016-02-19T17:13:21Z
SAMSUNG_1567	2016	2	18	2016-02-25T17:13:21Z
SAMSUNG_1567	2015	3	2	2015-12-19T07:13:21Z

STOCKAGE PHYSIQUE DES DONNÉES 3



REQUÊTES

- Sur la clé de partitionnement: = et IN
- Sur les colonnes de clustering: <, <=, >, >= et IN

ORDRE

- sur la clé de partitionnement: aucun ordre
- Sur les colonnes de clustering: ordonné en fonction de l'ordre de déclaration des colonnes

MODÉLISATION DES RELATIONS

- **Relation 1-1:** une relation 1-1 peut être modélisée en utilisant une simple clé primaire
- **Relation 1-n:** une relation 1-n peut être modélisée en utilisant une clé primaire composée d'une partition key et d'au moins une clustering key
- **Relation m-n:** une relation n-n peut être modélisée en utilisant une clé primaire composée d'une clé de partition composée et d'au moins une clustering key

MODÉLISATION AVANCÉE

LES COLLECTIONS

Depuis la version 1.2 de C* il est possible de créer des colonnes de type collection (list, set et map).

Les collections ne peuvent pas être découpée. Cassandra lira la collection en entier. De ce fait il ne faut pas dépasser quelques milliers d'éléments par collection.

Il n'est pas possible d'inclure une collection dans une collection.

LES COMPTEURS

v2.1

Il est possible de stocker un compteur qui compte un nombre d'occurrence.

Attention, si une table a une colonne de type counter, toutes les colonnes non-"counter" doivent faire partie de la clé primaire.

```
CREATE TABLE devicesByUser (
    user_id UUID,
    number_devices COUNTER,
    PRIMARY KEY (user_id)
);

UPDATE devicesByUser SET number_devices = number_devices + 2
WHERE user_id = 1;
```

TTL: TIME TO LIVE

Il est possible d'ajouter une durée de vie à une donnée (ligne ou colonne individuelle) au moment de l'insertion. Cassandra se chargera alors de supprimer cette donnée une fois la durée écoulée.

```
INSERT INTO data (id, event_time, value)
VALUES ('1','2015-12-19T07:13:21Z',34) USING TTL 3600;

UPDATE data USING TTL 30 SET value=12 WHERE id='1';
```

STATIC

Une colonne statique est une colonne qui est partagée par toutes les lignes de la même partition.

```
CREATE TABLE t (
    k text,
    s text STATIC,
    i int,
    PRIMARY KEY (k, i)
);

INSERT INTO t (k, s, i) VALUES ('k', 'I''m shared', 1);
INSERT INTO t (k, s, i) VALUES ('k', 'I''m still shared', 2);
SELECT * FROM t;
```

Et le résultat:

k	s	i
k	I'm still shared	1
k	I'm still shared	2

BATCH

Une requête "batch" permet de combiner de multiples INSERT, UPDATE et DELETE au sein d'une même opération logique:

```
BEGIN BATCH  
INSERT INTO t (k, s, i) VALUES ('k', 'I''m shared', 1);  
INSERT INTO t (k, s, i) VALUES ('k', 'I''m still shared', 2);  
APPLY BATCH;
```

- Un BATCH est atomique: si une requête échoue alors toutes les requêtes du BATCH échouent.
- Un BATCH n'est pas isolé: d'autres requêtes peuvent voir les données écritent par un batch en cours d'exécution.

UDT: USER DEFINED TYPE

v2.1

Il est possible de déclarer dans C* des structures plus complexes. Par exemple si je souhaite gérer des listes d'adresses comment faire ?

Deux possibilités:

1. Déclarer une list et mettre dans le texte notre adresse (sous forme de JSON par exemple)
2. Utiliser un UDT afin de déclarer de manière explicite notre adresse

UDT: USER DEFINED TYPE

v2.1

```
CREATE TYPE address (
    street text,
    city text,
    zip int
);

CREATE TABLE user_profiles (
    login text PRIMARY KEY,
    first_name text,
    last_name text,
    email text,
    addresses map<text, frozen<address>>
);
```

UDT: USER DEFINED TYPE

v2.1

```
// Inserts a user with a home address
INSERT INTO user_profiles(login, first_name, last_name, addresses)
VALUES ('tsmith', 'Tom', 'Smith',
{ 'home': { street: '1021 West 4th St. #202',
city: 'San Fransisco',
zip: 94110 }});

// Adds a work address for our user
UPDATE user_profiles
SET addresses = addresses
+ { 'work': { street: '3975 Freedom Circle Blvd',
city: 'Santa Clara',
zip: 95050 }}
WHERE login = 'tsmith';
```

MATERIALIZED VIEW

v3.0

Nous avons vu au début de cette partie que pour faire des requêtes sur des colonnes ne faisant pas partie de la clé de partitionnement **il fallait dénormaliser et créer des tables adaptées aux requêtes** que l'on souhaitaient faire. Les index secondaires ne sont pas la solution car ils ne sont pas suffisamment performants.

Cassandra 3 fournit un nouveau mécanisme appelé Materialized View qui est en fait une **dénormalisation effectuée automatiquement côté serveur**.

MATERIALIZED VIEW

v3.0

Reprendons notre exemple initial: nous voulons pouvoir obtenir les informations de l'utilisateur en fonction de l'email de l'utilisateur et pas seulement depuis son id.

```
CREATE TABLE users (
    id UUID PRIMARY KEY,
    first_name TEXT,
    last_name TEXT,
    email TEXT
);

CREATE MATERIALIZED VIEW user_by_email
AS SELECT * //denormalize ALL columns
FROM users
WHERE email IS NOT NULL AND id IS NOT NULL
PRIMARY KEY(email, id);
```

MATERIALIZED VIEW

v3.0

Il n'y a rien de particulier à faire ensuite, on insère les données normalement dans la table users et on peut lire les données depuis la vue user_by_email:

```
INSERT INTO users(id,first_name,last_name,email) VALUES(1, 'John', 'DOE'  
INSERT INTO users(id,first_name,last_name,email) VALUES(2, 'Helen', 'SUE'  
  
SELECT * FROM user_by_email;
```

email	id	first_name	last_name
jdoe@gmail.com	1	'John'	'DOE'
hsue@yahoo.com	2	'Helen'	'SUE'

ET BIEN PLUS...

Il est possible de faire encore bien plus avec le CQL (UDF, UDA, tuples, ...).

Pour plus d'informations une documentation très complète est disponible.

CRÉER UNE APPLICATION JAVA SCALABLE

PRÉPARER CASSANDRA POUR LA PRODUCTION

STRATÉGIES DE RÉPLICATION

SimpleStrategy, NetworkTopologyStrategy

STRATÉGIES DE COMPACTION

QUERY TRACING

AUTHENTIFICATION ET AUTORISATION

MERCI !