Etude du partitionnement et de la réplication des données dans Apache Cassandra

CNAM

2016

Projet NFE 204

Benoit DESILES

Enseignant : Philippe Rigaux

Table des matières

Présentation du projet

Les grandes entreprises du web des années 2000 ont dû faire face à une croissance si rapide et une telle augmentation de la fréquentation de leurs sites internet, qu'elles se sont vues obligées de concevoir de nouvelles technologies ; les systèmes existants ne répondaient plus aux exigences de performances et de volumes nécessaires aux services proposés par des sociétés comme Amazon, Google ou Facebook. C'est à ce moment qu'elles développent leurs propres solutions de base de données : le NoSQL, Not only Structured Query Language, soit en français «pas seulement un langage de requête structurée».

NoSQL est une technologie complémentaire aux systèmes de gestion de bases de données relationnels (SGBDR) classiques qui utilisent le langage SQL (Structured Query Language). Le NoSQL ne vient pas remplacer les bases de données relationnelles mais proposer une alternative ou compléter les fonctionnalités des SGBDR pour donner des solutions plus intéressantes dans certains contextes.

Nous verrons que les cas d'usage des systèmes NoSQL sont bien différents de ceux des SGBDR. Les bases NoSQL sont souvent associées au Big Data («données massives»). D'autre part, l'informatique distribuée et la virtualisation ont contribué à la prolifération des environnements NoSQL, elles-mêmes poussées par l'expansion de l'informatisation des métiers et des informations traitées.

Dans ce projet nous nous intéressons au système NoSQL Apache Cassandra. L'objectif est de répondre aux questions suivantes : comment est-il possible de retrouver rapidement une information dans une base éclatée ? Et, comment, dans ce cas, éviter la perte de données s'il survient une panne matérielle ? Pour cela, nous utiliserons des données générées aléatoirement mais représentatives d'un cas réel. Nous installerons une grappe représentative d'un petit cluster de base de données.

Présentation de Apache Cassandra

Apache Cassandra est un projet dont le code source est libre. Actuellement porté par la fondation Apache, son développement a été initié par la société américaine Facebook. Il est programmé avec le langage Java depuis 2008. Cassandra est un système de gestion de base de données de type NoSQL, et plus précisément «orienté colonnes».

Les bases NoSQL orientées colonnes sont composées de tables comme dans un SGBDR. Généralement, une base de données NoSQL orientée colonnes est architecturée comme ceci : les tables stockent des clés, triées, uniques, qui identifient des familles de colonnes, ces familles listent des colonnes triées, et les colonnes listent des valeurs.

Les familles de colonnes sont définies à l'avance, mais pas les colonnes qui les constituent. D'une ligne à l'autre dans une table, le nombre de colonne varie. Les colonnes ont un nom et un type (comme dans un SGBDR), et à chaque valeur est associé un horodatage.

La conception de Apache Cassandra est guidée par la haute disponibilité et la gestion de données massives à grande échelle. Il a la réputation d'être robuste et performant même sur un grand nombre de serveurs et sur plusieurs centres de données.

Datastax, dont nous allons reparler, est une société fortement impliquée dans le projet Apache Cassandra. D'autre part, ses développeurs mettent à disposition des utilisateurs une syntaxe appelée CQL (Cassandra Query Language) pour interagir avec la base de données.

Nous avons choisi de diriger ce projet NFE204 vers Apache Cassandra pour plusieurs raisons. La première étant qu'il est très populaire et qu'il est très bénéfique de se construire une première expérience d'un tel système alors qu'il nous est inconnu. D'autre part, à toutes proportions gardées, la mise en oeuvre de la reprise sur panne est «simple». Cassandra intègres les techniques communes du monde NoSQL. Etudier Cassandra en détail permet de connaitre les principes de base qui se retrouvent dans beaucoup d'autres systèmes NoSQL, notamment le sharding (partitionnement), la réplication des données, le consistent hashing, le système gossip et d'autres.

Par la suite, nous faisons un bref comparatif de Apache Cassandra avec les SGBDR.

Comparaison avec les SGBDR

Afin de cerner ce qu'apporte Cassandra et plus généralement les systèmes NoSQL au regard des SGBDR, nous rappelons ici quelques caractéristiques des systèmes relationnels.

Les usages des SGBDR sont nombreux. Cela va de la bureautique au développement en passant par l'exploitation. Toutes les applications traitant des informations dont les attributs sont déterminés à l'avance, pérennes et atomiques utilisent des SGBDR. Citons Oracle Database ou PostreSQL par exemple. Bien que le schéma relationnel puisse être complexe, la structure des données est simple et figée. Parmi les applications utilisant un SGBDR, on peut citer les ERP (Enterprise Resource Planning) ou les systèmes d'archivage d'imagerie médicale. D'autre part, un grand nombre d'applications utilise des moteurs embarqués ou locaux comme SQLite.

Lors de la modélisation d'une base de données pour un SGBDR, on fait en sorte que les données ne soient pas redondantes dans la base. En NoSQL et dans Cassandra, cette question ne se pose pas.

Le modèle relationnel est performant pour une utilisation transactionnelle, c'est à dire «en ligne». Ici, il faut comprendre «en ligne» par respectant les propriétés A.C.I.D. et ayant un temps de validation rapide pour chaque requête (cohérence en temps réel). C'est le modèle OLTP (Online Transaction Processing) caractérisé par des requêtes d'écriture très fréquentes. Ce type d'application est typiquement opposé au traitement OLAP (Online Analytical Processing ou traitement analytique en ligne) qui fonctionne en principe en lecture. Les bases NoSQL sont souvent associées au traitement OLAP.

Rappel des propriétés A.C.I.D.

Les propriétés A.C.I.D. s'appliquent aux transactions informatiques, c'est-à-dire à des opérations sur des données informatiques. Les bases de données (en général) sont concernées par les propriétées A.C.I.D.

Atomicité : la propriété d'atomicité assure qu'une transaction se fait complètement ou pas du tout.

Cohérence : la propriété de cohérence assure qu'une transaction se termine sur un état valide et que toutes les données du système sont à jour.

Isolation : la propriété d'isolation assure que chaque transaction s'effectue comme si elle était la seule au moment T. Les transactions sont sériées.

Durabilité : la propriété de durabilité assure qu'après une transaction validée, elle se trouve immédiatement enregistrée et demeure même à la suite d'une panne grave.

Retenons que les SGBDR font l'usage de la cohérence forte des données, avec une structure des données stricte. Si on souhaite combiner des informations de tables différentes, on utilise le mécanisme de jointure. Cela permet de construire des requêtes élaborées. Par exemple retrouver à quel client a été vendu le produit A et en quelle quantité. Dans un SGBDR les tables «client» et «produit» sont distinctes.
Dans Cassandra, chaque «ligne» dans la table reprend toutes les informations afin d'éviter les jointures. Il existe un mécanisme de jointure dans Apache Cassandra, mais il est très différent des jointures de type algèbre relationnelle.

Dans la suite de ce document, nous revenons sur les notions de cohérence propre au NoSQL et à Apache Cassandra.

Démonstration de l'installation d'un cluster Cassandra

Eléments d'informatique distribuée

Dès la conception des systèmes NoSQL, le besoin était tourné vers la distribution des traitements et l'extension en échelle horizontale (scalabilité). Ces techniques correspondent à la faculté de parallèliser (ou répartir) les traitements et le stockage sur des grappes de machines (clusters). Pour un SGBDR, cette installation est compliquée car ça n'a pas été prévu lors de leur conception. Cette mise en oeuvre est simplifiée pour les bases NoSQL et nous prenons comme exemple la solution Apache Cassandra dans la suite de ce document.

À la naissance des SGBDR, le défi a consisté à optimiser le stockage et le traitement des données sur des machines très limitées par leurs ressources matérielles. Depuis, le matériel a beaucoup évolué mais le besoin de stockage des données croit encore plus

rapidement. Les petits clusters de bases relationnelles ne peuvent plus répondre à cette montée en charge, par conséquent les grandes entreprises du web en sont venues à la construction de centres de données, constitués de plusieurs centaines machines organisées en parallèle, on parle de «scalabilité horizontale» (extension en échelle horizontale), c'est l'informatique distribuée.

Les grappes sont composées de machines à bas coût, appelées Commodity Hardware, ce qui peut se traduire péjorativement par «matériels standards». Ces serveurs et machines standards sont installés en grande quantité dans les centres de données, et les pannes sont fréquentes. Il faut savoir les remplacer simplement, rapidement et au tarif le plus faible.

Les SGBD NoSQL sont très populaires chez les entreprises ayant connu une forte montée en charge de leurs services. Beaucoup ont développé leur solution interne, certaines sont devenues des références car ayant fait leurs preuves. C'est le cas de Cassandra. On retrouve largement le NoSQL dans ces domaines :

- Big data / Open data (données massives / données ouvertes)
- Moteurs de recherches
- Commerce en ligne
- Réseaux sociaux en ligne

Description de la grappe Cassandra

Pour notre projet NFE204, considérons une grappe de quatre serveurs interconnectés sur le réseau. Ces quatre serveurs hébergent la base de données Cassandra. Une application cliente installée sur une autre machine vient stocker des données et interroger la base de la grappe Cassandra.

Pour l'environnement de notre projet, avec l'aide du site web de la société Datastax, nous installons un cluster Cassandra selon les meilleures pratiques. Nous avons choisi de monter quatre machines virtuelles sur lesquelles Linux Centos 7 est installée. La version de Cassandra utilisée est la version 2.2.

Dans Cassandra, les grappes sont organisées en Datacenter et en Rack. C'est ce que nous avons configuré dans le fichier *cassandra-rackdc.properties* dont voici le contenu :

```
# These properties are used with GossipingPropertyFileSnitch and will
# indicate the rack and dc for this node
dc=dc1
rack=rack1
```

Nous avons un Datacenter nommé «dc1» et un Rack nommé «rack1». Nos quatre machines sont dans le rack1, lui même fait partie du dc1.

Les nœuds de la grappe communiquent entre eux avec un protocole appelé gossip. Ce protocole fonctionne en mode paire-à-paire, les nœuds échangent des messages périodiquement sur leur état et l'état des nœuds qu'ils connaissent. Chacun des messages a un identifiant unique (un numéro de version), plus il est élevé plus le contenu du

message est récent. Les messages contenant un numéro de version plus ancien que celui stocké sont rejetés par les serveurs. A la fin de la convergence, tous les serveurs/nœuds ont le même numéro de version du message. Cette convergence se répète incessamment.

Parmi nos quatre nœuds, nous en avons choisi deux qui ont le rôle de Seed. Ce rôle fait référence à la capacité de reconnaitre la topologie réseau lorsque de nouveaux nœuds sont ajoutés dans la grappe.

Cassandra implémente aussi une fonction appelée Snitch (un terme familier pour désigner une personne qui dénonce). C'est la fonction Snitch qui détermine à quel Datacenter et quel Rack appartient un nœud. Nous avons fixé la fonction Snitch avec le paramètre GossipingPropertyFileSnitch. Le paramètre GossipingPropertyFileSnitch dit que ces informations sont distribuées avec le protocole gossip au reste de la grappe. C'est le paramètre recommandé pour un système en production.

Procédure d'installation de la grappe Cassandra

Après avoir installé les quatre machines virtuelles avec leur système d'exploitation, leur configuration réseau et les packages Cassandra, il nous faut adapter les fichiers de configuration.

La configuration réseau est simple, les quatre serveurs sont dans le même sous-réseau 192.168.3.0/24 avec IP statiques et leur nom enregistré dans le DNS : node1 à node4.

Les fichiers de configuration Cassandra modifiés sont les suivants :

- cassandra.yaml
- cassandra-topology.properties
- cassandra-rackdc.properties

Pour le fichier *cassandra.yaml*, les paramètres par défaut sont gardés, à l'exception de ceux-ci :

```
cluster_name: 'XYZ Cluster'

data_file_directories:
    - /var/lib/cassandra/data

commitlog_directory: /var/lib/cassandra/commitlog
saved_caches_directory: /var/lib/cassandra/saved_caches

seed_provider:
    - class_name: org.apache.cassandra.locator.SimpleSeedProvider
      parameters:
          - seeds: "192.168.3.151,192.168.3.152"

listen_interface: ens18

endpoint_snitch: GossipingPropertyFileSnitch
```

Notre grappe s'appelle «XYZ Cluster».

Nous avons créé les sous-répertoires de /var/lib/cassandra et leurs permissions UNIX à l'aide d'un script bash déployé sur les quatre serveurs.

Le rôle de Seed est attitré aux noeuds 1 et 2 (192.168.3.151 et 192.168.3.152).

L'interface de communication avec l'application cliente est ens18 (interface Ethernet de la machine) sur le port 7000 («storage_port: 7000», paramètre par défaut).

Le paramètre endpoint_snitch a été évoqué précédement.

D'autre part, on note les paramètres intéressants comme :

```
num_tokens: 256

partitioner: org.apache.cassandra.dht.Murmur3Partitioner
```

dont nous reparlerons plus loin.

Concernant le fichier cassandra-topology.properties, voici son contenu :

```
# Cassandra Node IP=Data Center:Rack
192.168.3.151=dc1:rack1
192.168.3.152=dc1:rack1
192.168.3.153=dc1:rack1
192.168.3.154=dc1:rack1

# default for unknown nodes
default=dc1:rack1
```

Le contenu du fichier *cassandra-rackdc.properties* a été détaillé précédement.

Pour finir, il reste à copier ces trois fichiers sur les trois autres noeuds de la grappe et lancer le service Cassandra sur chacun des noeuds. On peut aisément vérifier dans les fichiers journaux (notamment */var/log/cassandra/cassandra.log*) qu'il n'y a pas d'erreur au lancement.

Démonstration de l'insertion des données

Les bases de données NoSQL marquent une rupture assez brutale avec la manière de concevoir les schémas de données. Il est techniquement possible de stocker du texte dans les cellules d'une table d'un SGBDR mais ces données ne sont pas atomiques. D'autre part, d'un ensemble d'informations à l'autre, les textes sont organisés ou structurés différemment. De fait, la recherche de zones de textes dans une base de donnée relationnelle n'est pas efficace. Nous allons voir que les systèmes NoSQL, et notamment Apache Cassandra, sont plus flexibles.

Les systèmes relationnels respectent toutes les propriétés A.C.I.D., ce qui n'est pas forcément le cas de tous les systèmes NoSQL. En pratique, cela demande un effort supplémentaire au développeur pour s'assurer que les données traitées soient cohérentes.

Concernant l'accès aux données, il n'existe pas de norme, contrairement au modèle relationnel. Pour pallier à cela et utiliser une base de données NoSQL, il faut adopter et maîtriser des outils et des interfaces de programmation spécifiques permettant d'accéder aux données dans chaque SGBD NoSQL, comme CQL (Cassandra Query Language) qui se veut proche du SQL, nous y reviendrons.

Pour pouvoir étudier ce qui nous intéresse ici, le partitionnement et la réplication des données, nous devons nous procurer des données représentatives d'un cas réel de production. Elles sont décrites par la suite.

Eléments de cohérence des données

Avant d'aborder les jeux de données en question, revenons sur la notion de cohérence des données. La première nature des bases NoSQL étant de s'écarter du modèle relationnel, les propriétés A.C.I.D. font l'objet de questionnements. En effet, dans le cas des systèmes NoSQL distribués, une partie les données sont dupliquées, cela assure la reprise sur panne du système. Lorsqu'une donnée est écrite à plusieurs endroit, comment garantir que les écritures sont atomiques, cohérentes, isolées et durables ?

Parmi les propriétés A.C.I.D., la propriété la plus importante est la cohérence des données. Dans un système distribué, vulnérable aux pannes isolées, il faut considérer deux façons de travailler. Un mode synchrone ou asynchrone.

En mode synchrone, l'écriture des données est validée seulement à la fin des écritures sur toutes les machines concernées.

En mode asynchrone, les données sont écrites a posteriori sur certaines machines. Cela implique d'avoir un mécanisme qui vérifie la cohérence des données a posteriori. On parle de cohérence finale ou de cohérence à terme.

Dans un système distribué, constitué de plusieurs machines appelées nœuds, le théorème CAP nous dit qu'il serait possible d'assurer seulement deux sur trois des propriétés suivantes.

Consistency (Cohérence) : toutes les données sont à jour sur tous les nœuds au même moment.

Availability (Disponibilité) : chaque requête sur le système reçoit une réponse.

Partition-tolerent (Résistance au morcellement) : le système continue de fonctionner en dépit d'un morcellement provoqué par une panne réseau.

Bien que la disponibilité et la résistance au morcellement réseau soient les priorités d'un système NoSQL distribué, les développeurs de l'univers NoSQL se donnent pour objectif de satisfaire au mieux les trois garanties.

Description des données choisies

Dans le cadre de ce projet NFE204, nous avons choisi un générateur de données aléatoires. Il s'agit du site web http://www.json-generator.com qui génère des données au format JSON selon le schéma qui lui est confié (lui aussi au format JSON). Nous avons

gardé le schéma proposé par défaut sur le site web et lui avons demandé de générer cinquante mille jeux de données.

Chaque jeu de données représente un utilisateur/client/consommateur et ses informations associées telles que son nom, son âge, son numéro de téléphone, etc. Ici nous considérons qu'il s'agit d'une base de données stockant des informations sur cinquante mille utilisateurs fictifs d'un service quelconque.

Le format JSON d'origine des données est celui-ci :


```
  {
    "_id": "574ed0671da3d83fd54a79fb",
    "index": 0,
    "guid": "3789c961-23e8-4c2c-b098-7023298d5ec4",
    "isActive": true,
    "balance": "$2,966.30",
    "picture": "http://placehold.it/32x32",
    "age": 31,
    "eyeColor": "blue",
    "name": "Natalie Crawford",
    "gender": "female",
    "company": "UTARA",
    "email": "nataliecrawford@utara.com",
    "phone": "+1 (947) 429-2215",
    "address": "891 Fleet Street, Grantville, Pennsylvania, 7550",
    "about": "Est exercitation et Lorem quis proident consectetur sit qui dolor
sunt esse. Eiusmod cupidatat quis ullamco mollit sunt consectetur ullamco
adipisicing ut mollit fugiat ut. Cupidatat consectetur culpa sit ipsum commodo
sunt adipisicing et. Irure sint aliquip velit consectetur. Eu aute esse culpa
laboris dolor Lorem proident velit. Velit laboris nulla non officia incididunt
quis voluptate nisi Lorem aute ullamco.\r\n",
    "registered": "2015-10-22T05:42:58 -02:00",
    "latitude": 25.290866,
    "longitude": 92.767241,
    "tags": [
      "esse",
      "dolor",
      "sit",
      "proident",
      "aute",
      "fugiat",
      "amet"
    ],
    "friends": [
      {
        "id": 0,
        "name": "Edith Jordan"
      },
      {
        "id": 1,
        "name": "Hall Hooper"
      },
      {
        "id": 2,
        "name": "Fanny Middleton"
      }
    ],
    "greeting": "Hello, Natalie Crawford! You have 5 unread messages.",
    "favoriteFruit": "apple"
  }
```

On peut voir que ce schéma contient des valeurs simples, textes ou numériques, une liste simple («tags») et une liste de valeurs («friends»).

Depuis la version 2.2 de Apache Cassandra et la version 3 de CQL, il est possible de travailler avec des données au format JSON. Ce format de représentation de données étant très populaire, nous souhaitions expérimenter celui-ci avec Apache Cassandra.

Procédure d'insertion des données

Schéma des données

La première étape avant de pouvoir stocker des données dans la base est de créer un «Keyspace». Selon les meilleures pratiques données par Datastax, à chaque application cliente de la base Cassandra est dédié un Keyspace, c'est dans ce Keyspace que sont stockées les tables et les données sous forme de colonnes.

On crée le Keyspace avec la commande suivante dans le shell CQL :

```
cqlsh> CREATE KEYSPACE KeyspaceXYZ WITH REPLICATION = { 'class' :
'SimpleStrategy', 'replication_factor' : 3 };
```

Notre Keyspace s'appelle KeyspaceXYZ et nous avons fixé le facteur de réplication à 3, c'est-à-dire que les données sont répliquées sur trois serveurs parmi les quatre que nous avons (autrement dit, trois exemplaires au total pour chaque donnée). C'est un choix arbitraire de l'utilisateur. Nous considérons également le mode de réplication appelé SimpleStrategy, il est adapté à une architecture pour une grappe simple, c'est à dire pour 1 Rack dans 1 Datacenter (ce qui est bien notre cas ici).

Il nous faut maintenant créer une table dans le Keyspace et définir le nom des colonnes et leur type. Nous avons choisi d'appeler notre table «users» car son contenu est orienté vers les données utilisateurs.

Nous savons que les 50000 jeux de données (c'est-à-dire les 50000 utilisateurs) ont le même schéma. Cela signifie que pour chaque utilisateur dans la base, tous ont les mêmes attributs, il y en a 22 pour chacun, qu'ils soient renseignés ou non, de «_id» à «FavoriteFruit» en passant par «name» et «address».

Par conséquent nous prenons pour chaque paire nom/valeur le nom comme titre de colonne et la valeur comme le contenu de la colonne. La nature de la valeur détermine son type : texte, numérique (entier ou décimal), booléen ou liste dans notre cas (il en existe beaucoup d'autres autorisés dans Cassandra).

Par exemple, dans l'extrait ci-dessus, on crée notamment une colonne dont le titre est «company», son type est texte et sa valeur est «UTARA». Autre exemple, la colonne «isActive», son type est booléen et sa valeur est True. Nous choisissons la colonne «index» comme clé primaire.

Nous avons une particularité dans notre schéma JSON : une liste «friends» de paires nom/valeur, qui contient une liste de personnes avec un identifiant (un nombre entier) et leur nom. Cassandra ne sait pas reconnaitre seul le type des données dans ce cas là. Il

nous faut créer un nouveau type de données (qu'on appelle «type_friend») avec la commande CQL suivante. Cette commande doit être exécutée avant la commande de création de la table, car la table en question utilisera ce type de données.

```
cqlsh> CREATE TYPE KeyspaceXYZ.type_friend (frd_id int, frd_name text);
```

Les noms «id» et «name» de la liste «friends» ont été substitués respectivement par «frd_id» et «frd_name» par souci de clarté dans les termes et éviter les confusions maladroites. Les détails sont abordés dans le paragraphe suivant du document.

Formatage des données avant insertion

Dans Cassandra, il n'est pas autorisé de prendre les termes «_id» et «index» pour les titres de colonne, sinon des erreurs se présentent au moment de l'insertion des données.

Il nous est nécessaire d'apporter quelques substitutions à notre fichier JSON. Avec un script bash c'est très simple, les commandes sed et awk font le travail rapidement étant donné que la quantité de données est relativement faible.

Concrètement, nous avons apporté les substitutions suivantes :

Nom de colonne	Est remplacé par
_id	long_id
index	uid (clé primaire)
id (de la liste «friends»)	frd_id
name (de la liste «friends»)	frd_name

La commande CQL de création de la table suit immédiatement la commande de création du type «type_friend» précédente :

```
cqlsh> CREATE TABLE users (
... long_id text,
... uid int PRIMARY KEY,
... isActive boolean,
... balance text,
... age int,
... eyeColor text,
... name text,
... gender text,
... company text,
... email text,
... phone text,
... address text,
... about text,
... registered text,
```

```
... latitude decimal,
... longitude decimal,
... tags list<text>,
... friends list<FROZEN<type_friend>>,
... greeting text,
... favoriteFruit text
... );
```

A chaque colonne est donné un titre et un type de données pris en charge. Nous constatons que la syntaxe du langage CQL (Cassandra Query Language) est très inspirée de celle de SQL des tables relationnelles. Notons que la clé primaire est la colonne «uid» et le type «type_friend» est indiqué avec le mot clé FROZEN. Autrement, on retrouve les types text, int, boolean et list.

Pour l'insertion de données au format JSON, les titres des colonnes doivent correspondre exactement aux noms des paires nom/valeur du document JSON.

Technique d'insertion des données

Dans le cadre de notre projet NFE204, une application cliente n'est pas strictement nécessaire pour l'insertion des données. Nous utilisons un simple script bash. Il s'agit d'une boucle qui prend chaque ligne du fichier JSON et exécute la commande d'insertion avec l'utilitaire cqlsh fourni par Cassandra.

Avant cela, nous devons tester manuellement avec un jeu de données qu'il n'y a pas d'erreur de syntaxe ou de type non reconnu. La commande CQL basique est la suivante :

```
cqlsh> INSERT INTO users JSON '{
... "long_id": "56e186b0f2e62f2966315c83",
... "uid": 0,
... "isActive": true,
... "balance": "$3,547.29",
... "age": 36,
... "eyeColor": "green",
... "name": "Merle Dixon",
... "gender": "female",
... "company": "GORGANIC",
... "email": "merledixon@gorganic.com",
... "phone": "+1 (961) 426-3712",
... "address": "782 Euclid Avenue, Colton, Virgin Islands, 6172",
... "about": "Ipsum esse non amet dolor sint voluptate sit elit labore elit
irure. Nisi adipisicing nulla nisi adipisicing tempor nulla ipsum irure. Dolore
occaecat ea exercitation sit consequat proident magna.",
... "registered": "2015-06-04T11:52:17 -02:00",
... "latitude": 4.450566,
... "longitude": 107.208836,
... "tags": [
... "sit",
... "ea",
... "non",
... "esse",
... "enim",
... "nostrud",
... "nostrud"
... ],
... "friends": [
```

```
... { "frd_id": 0, "frd_name": "Price Bush" },
... { "frd_id": 1, "frd_name": "Stacie Sosa" },
... { "frd_id": 2, "frd_name": "Clark Gay" }
... ],
... "greeting": "Hello, Merle Dixon! Welcome to Gorganic !",
... "favoriteFruit": "apple"
... }';
```

Elle s'exécute sans erreur, dans ce cas nous pouvons lancer le script bash :

```
while read line;
do  cqlsh -k KeyspaceXYZ -e "INSERT INTO users JSON '$line'";
done < user-database.json
```

Pour suivre l'évolution de l'insertion des données, nous avons un outils à disposition appelé nodetool.

Dans le terminal d'un des quatre serveurs, la commande «nodetool cfstats KeyspaceXYZ» nous renvoie les informations suivantes :

```
Keyspace: KeyspaceXYZ
      Read Count: 0
      Read Latency: NaN ms.
      Write Count: 2156
      Write Latency: 0.039288497217068646 ms.
      Pending Flushes: 0
            Table: users
            SSTable count: 0
            Space used (live): 0
            Space used (total): 0
            Space used by snapshots (total): 0
            Off heap memory used (total): 0
            SSTable Compression Ratio: 0.0
            Number of keys (estimate): 2154
            Memtable cell count: 66836
            Memtable data size: 2723267
            Memtable off heap memory used: 0
            Memtable switch count: 0
            Local read count: 0
            Local read latency: NaN ms
            Local write count: 2156
            Local write latency: 0,044 ms
            Pending flushes: 0
            Bloom filter false positives: 0
            Bloom filter false ratio: 0,00000
            Bloom filter space used: 0
            Bloom filter off heap memory used: 0
            Index summary off heap memory used: 0
            Compression metadata off heap memory used: 0
            Compacted partition minimum bytes: 0
            Compacted partition maximum bytes: 0
            Compacted partition mean bytes: 0
            Average live cells per slice (last five minutes): NaN
            Maximum live cells per slice (last five minutes): 0
            Average tombstones per slice (last five minutes): NaN
            Maximum tombstones per slice (last five minutes): 0

----------------
```

L'écriture des données est en cours («Write Count: 2156» sur 50000 attendus) et on constate à ce stade que les SSTables n'ont pas déjà été écrites sur les disques, c'est la Memtable qui contient les données pour le moment. Nous revenons sur ces détails dans la partie suivante du document.

Une autre commande CQL permettant de suivre la progression des écritures dans la base (similaire à SQL encore une fois) :

```
cqlsh> use KeyspaceXYZ;
cqlsh:KeyspaceXYZ> SELECT COUNT(uid) FROM users;

 system.count(uid)
-------------------
              4104

(1 rows)
```

Etude du partitionnement et de la réplication

Apache Cassandra utilise un stockage distribué, conçu pour fonctionner sur une large grappe de machines. Nous nous intéressons aux questions suivantes : comment est-il possible de retrouver rapidement une information dans une base éclatée ? Et, comment, dans ce cas, éviter la perte de données s'il survient une panne matérielle ?

Pour répondre à ces questions, maintenant que nos données sont insérées dans la base, nous allons détailler le partitionnement réseau («sharding» en anglais) et la réplication des données proposés par Apache Cassandra

Apache Cassandra propose une architecture distribuée, multi-nœuds, asynchrone. Il n'y a pas de serveur maître ni de serveurs secondaires. Chaque serveur est interconnecté aux autres, ils se surveillent mutuellement en continu. Si un nœud disparait, les requêtes sont redirigées vers un autre serveur de la grappe.

Quelques caractéristiques sur le NoSQL

D'autre part, comme dans un SGBDR et d'autres systèmes NoSQL, Cassandra a un mécanisme de journalisation afin d'archiver chaque requête avant d'être réellement exécutée. La journalisation prend moins de temps que la requête d'écriture. Dans le cas où le système s'effondre avant la fin de l'écriture des données, après le redémarrage le journal permet de reprendre les opérations non terminées dans la base.

Les moteurs NoSQL reposent sur le principe de l'agrégat, avec un couple clé-valeur. Un agrégat est une collection d'objets (données) regroupés. Dans cet agrégat, les données sont identifiées par une racine, soit une clé ou un numéro unique. La valeur représente effectivement les données. Le point d'entrée ou de référence pour une requête NoSQL est toujours la racine ou la clé. Par analogie, on peut assimiler une bibliothèque à une base de données. Chaque livre à un identifiant, c'est la clé d'entrée (par exemple son numéro ISBN, International Standard Book Number). Les contenus des livres sont les

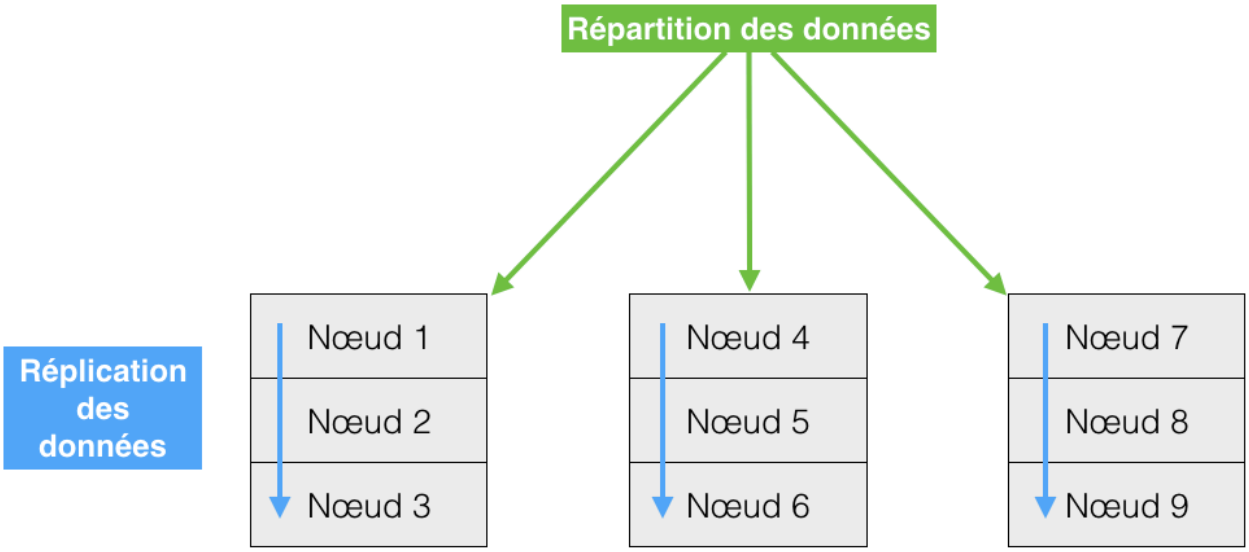
données. Pour lire ces données il faut d'abord trouver le livre avec sont identifiant, puis l'ouvrir pour le consulter.

D'autre part, Cassandra contrôle le schéma des données, ce n'est pas toujours le cas des moteurs NoSQL. Dans un SGBDR, il est nécessaire d'interroger deux tables pour faire une jointure. Du point de vue du moteur NoSQL, il n'y a pas de relation entre les tables de la base de données, donc pas de jointure. Cela en fait un critère important pour fournir simplement la distribution des données. On appelle parfois ces tables des collections.

Partitionnement des données

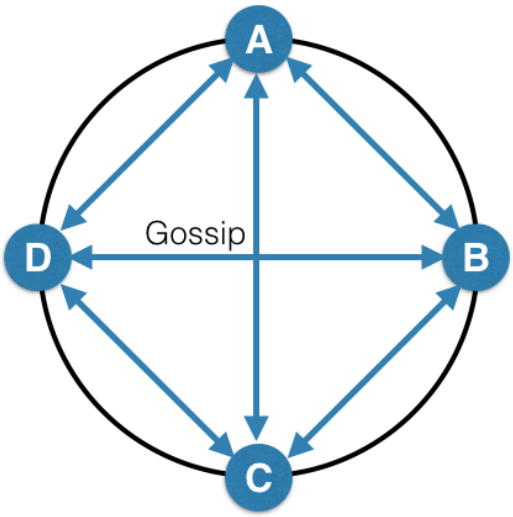
Pour assurer la reprise sur panne et une maintenance simple dans un système distribué, Cassandra utilise le partitionnement réseau et la réplication des données. Pour illustrer ce principe, faisons l'analogie avec un système RAID 10 (Redundant Array of Independent Disks), les données sont d'abord réparties entre plusieurs nœuds, par analogie au striping. Mais ces nœuds sont finalement des grappes de nœuds répliqués entre eux, par analogie au mirroring.

Le partitionnement consiste à répartir ou à distribuer les données sur différents noeuds de la grappe. Cassandra s'arrange pour répartir la charge, c'est-à-dire éviter qu'un nombre restreint de machines stocke plus de la moitié de la quantité de données. Dans le cas contraire, l'intérêt du stockage distribué est perdu.



Les requêtes de lecture/écriture peuvent être envoyées à n'importe quel nœud de la grappe. Pour la requête en question, le serveur choisi agit comme un «coordinateur», il se charge de transmettre la requête et les données aux autres serveurs de la grappe.

Dans Apache Cassandra, on représente schématiquement les serveurs de la grappe et les données sur un cercle («ring»).



Cassandra utilise le Consistent Hashing pour répartir les tranches de données dans la grappe, cette technique permet de minimiser la réorganisation des nœuds quand de nouveaux sont ajoutés ou d'autres supprimés de la grappe. Chaque nœud est responsable d'une tranche des données, sur nos quatre nœuds, les tranches sont égales (Figure 1. «Schéma d'une requête d'écriture»).

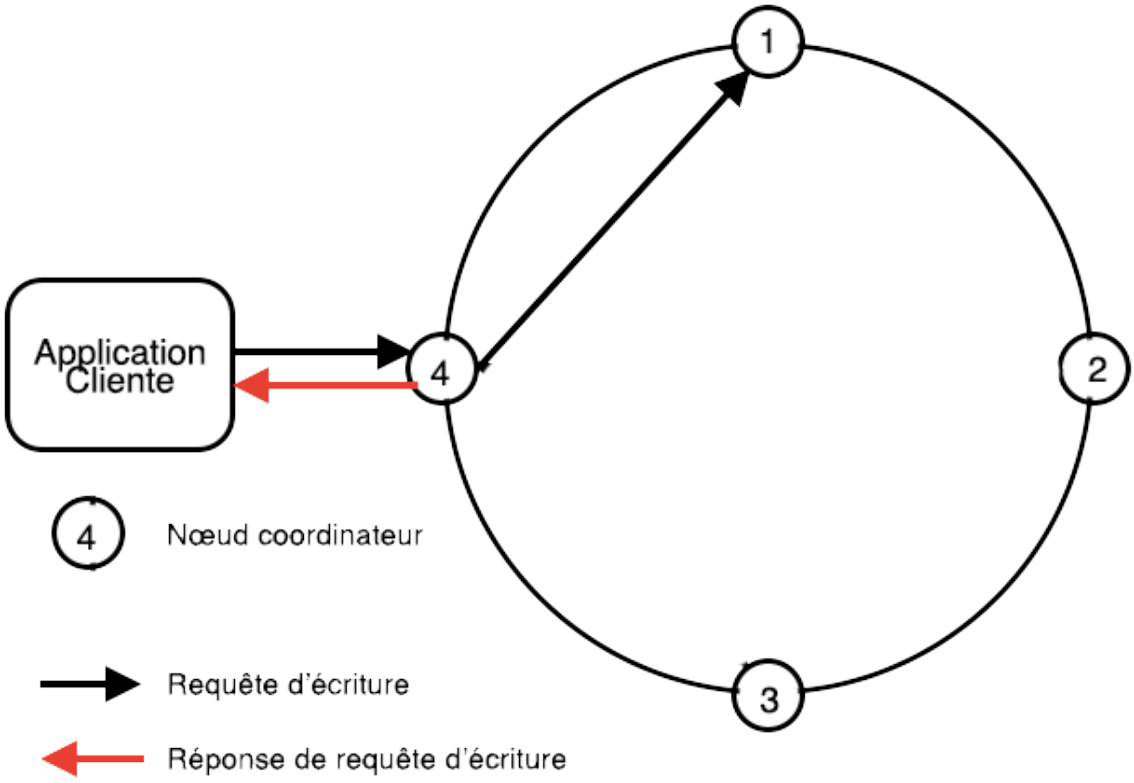


Figure 1. Schéma d'une requête d'écriture

Une fois les tranches attribuées aux serveurs, c'est le Partitioner qui détermine quelle donnée va sur quel nœud du cercle. La totalité des données (et la position des serveurs) étant représentée par ce cercle, chaque tranche est un arc de cercle. Nous avons parlé du nœud «coordinateur», mais le Partitioner est l'algorithme qui distribue les données. Le coordinateur n'est que «l'élu» pour exécuter l'opération.

Dans notre exemple de cluster avec quatre nœuds, nous choisissons le Partitioner appelé Murmur3Partitioner. C'est celui recommendé par Datastax. Il s'agit d'un algorithme de calcul («hashage») délivrant une plage de valeur allant de -2^63 à +2^63-1.

Nous répondons finalement à la question «comment est-il possible de retrouver une donnée dans une base partitionnée ?». Chaque ligne d'une table possède un «hash» aussi appelé «token» (un nombre unique calculé avec l'algorithme Murmur3Partitioner à partir de la clé primaire de la ligne). Le Partitioner lit ou calcule le «token» s'il n'existe pas pour distribuer la requête dans la grappe, c'est-à-dire dans le cercle. Cassandra conserve en mémoire une table de routage appelée Partition Summary contenant un index appelé Partition Index. Cet index contient toutes les clés primaires et leur destination dans la grappe.

Réplication des données

Pour éviter la perte de données suite à une panne matérielle grave, Cassandra s'occupe de gérer le partitionnement, comme nous l'avons vu, mais aussi la réplication des données sur les nœuds de la grappe. La réplication des données consiste à multiplier les copies d'informations sur différents serveurs. Cela permet également de répartir les requêtes de lecture sur une donnée utilisée simultanément par plusieurs applications, services ou machines.

Dans un environnement de production il nous faut configurer la réplication des données. La Figure 2 «Schéma d'une requête d'écriture avec réplication» illustre notre exemple.

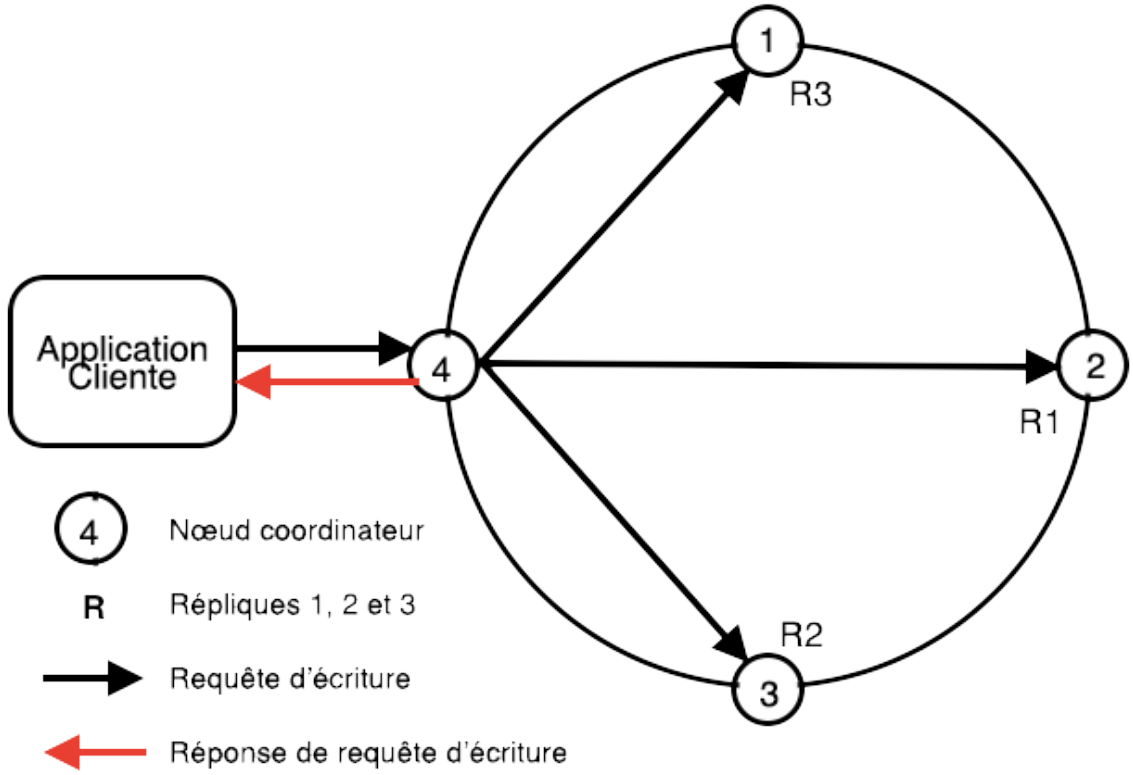


Figure 2. Schéma d'une requête d'écriture avec réplication

Pour Cassandra il n'y a pas de serveur maître, ni en partitionnement, ni en réplication. Rappelons-nous qu'un nouveau nœud coordinateur est élu pour chaque requête. Les données sont dupliquées dans le sens des aiguilles d'une montre sur le cercle sur lequel sont représentés les nœuds et la totalité des données.

Quand les requêtes sont distribuées, Cassandra utilise le mécanisme suivant : lors d'une requête d'écriture de données, Cassandra stocke les données dans une mémoire appelée Memtable, la Memtable est stockée dans la mémoire vive de la machine. En parallèle la requête est écrite dans un journal («log»). Le journal est stocké sur le disque dur.

La Memtable est déchargée périodiquement («flush») sur le disque dans une structure appelée SSTable. La SSTable est le fichier qui contient les données à terme sur le disque. Ce ne sont pas les SSTables qui sont répliquées mais les requêtes qui sont distribuées et répétées à travers la grappe. Par conséquent on retrouve les SSTables sur plusieurs nœuds (sur trois nœuds dans notre cluster).

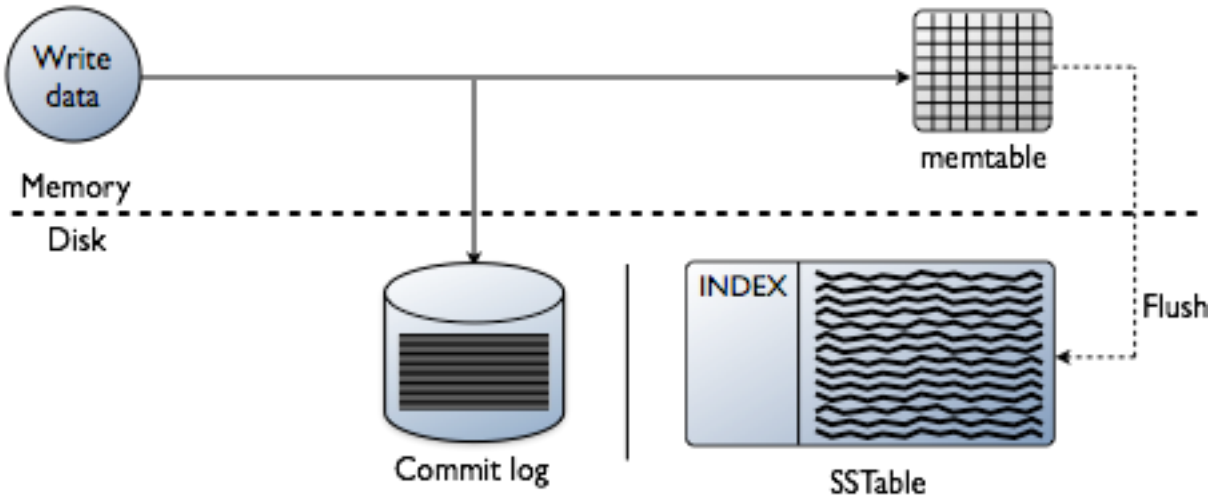


Figure 3. Mécanisme d'écriture

Ainsi, lorsque qu'un nœud/serveur s'effondre suite à une panne de courant électrique par exemple ou un défaut matériel, les données perdues sont toujours présentes sur les «replicas».

```
cqlsh> CREATE KEYSPACE KeyspaceXYZ WITH REPLICATION = { 'class' :
'SimpleStrategy', 'replication_factor' : 3 };
```

La commande qui nous a permit de créer le Keyspace précédement comporte un facteur de réplication égale à 3. Cela autorise d'avoir jusqu'à deux machines hors ligne simultanément sans rendre la base indisponible pour les applications clientes.

Dans Apache Cassandra, les concepts de Datacenter et de Rack sont utilisés pour définir la topologie et comment sont placés les répliques dans la topologie. Quel est l'intérêt pour l'administrateur d'avoir plusieurs Datacenters et Racks ? Un cluster peut être réparti sur plusieurs centres de données, construits à plusieurs kilomètres les uns des autres. Ceci pallie aux graves incendies par exemple. Les réplicas sont placés sur différents racks pour surmonter les pannes. En effet, les pannes affectent fréquemment un rack entier (panne d'électricité ou réseau), il faut éviter d'avoir des copies de données identiques dans une seule baie informatique. Apache Cassandra impose de connaitre l'emplacement physique de chaque machine : dans quel rack et dans quel centre de données. Néanmoins, pour rappel, dans notre projet NFE204 nous n'avons qu'un rack.

Outre le facteur de réplication des données, en termes d'espace de stockage, les informations se trouvent fréquemment redondées dans une base NoSQL. On peut expliquer la redondance des données par l'absence de jointure entre les tables d'une base NoSQL. Les informations communes à plusieurs entrées dans la base sont écrites à chaque fois pour chaque entrée. Par conséquent, une base NoSQL prend plus de place qu'une base relationnelle. Cependant la différence n'est pas un frein aux performances et le coût du stockage baisse.

Simulation de panne et de reprise

Dans cette partie, nous allons effectuer différents tests, afin de constater le comportement de la grappe Cassandra en situation difficile.

Panne avec deux noeuds hors ligne

Parmi nos quatre machines, nous en éteignons deux (les noeuds 2 et 3, 192.168.3.152 et 192.168.3.153), puis nous essayons des requêtes d'écriture :

```
 root@node1[~]# nodetool status KeyspaceXYZ
Datacenter: dc1
===============
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address         Load       Tokens       Owns (effective)  Host ID      Rack
DN  192.168.3.152   16,54 MB   256          77,5%             ca043656-..  rack1
DN  192.168.3.153   15,73 MB   256          72,0%             c2985ee9-..  rack1
UN  192.168.3.154   16,54 MB   256          78,8%             f66c8325-..  rack1
UN  192.168.3.151   14,47 MB   256          71,7%             0a30946c-..  rack1

 root@cass1[~]# cqlsh
Connected to HKX Cluster at 127.0.0.1:9042.
[cqlsh 5.0.1 | Cassandra 2.2.6 | CQL spec 3.3.1 | Native protocol v4]
Use HELP for help.
cqlsh> use KeyspaceXYZ;
cqlsh:KeyspaceXYZ> INSERT INTO users JSON '{ "long_id":
"56e18a8bc339979c34d197ec", "uid": 50001, "isActive": true, "balance":
"$3,011.14", "age": 24, "eyeColor": "brown", "name": "Lori Bradley", "gender":
"female", "company": "BOLAX", "email": "loribradley@bolax.com", "phone": "+1
(883) 463-3087", "address": "397 Douglass Street, Ernstville, Wisconsin, 1256",
"about": "Labore ad duis dolore consequat voluptate sit consectetur occaecat ex
veniam cillum id aliqua. Culpa exercitation anim voluptate aliqua amet mollit
fugiat mollit nostrud enim anim mollit et. Labore anim ea sint et ullamco
excepteur incididunt nostrud laborum nostrud. Culpa enim enim quis culpa
excepteur. Cupidatat sint elit cillum tempor quis consequat eiusmod. Aliqua
ipsum eiusmod culpa aute aliquip Lorem est laborum aliqua eiusmod.rn",
"registered": "2015-08-15T02:17:09 -02:00", "latitude": -62.735535,
"longitude": -88.342294, "tags": [ "adipisicing", "consequat", "magna", "amet",
"cillum", "ex", "do" ], "friends": [ { "frd_id": 0, "frd_name": "Reba Wells" },
{ "frd_id": 1, "frd_name": "Carlson Dickson" }, { "frd_id": 2, "frd_name":
"Teresa Booker" } ], "greeting": "Hello, Lori Bradley! You have 10 unread
messages.", "favoriteFruit": "apple" }';
cqlsh:KeyspaceXYZ>
```

Les mentions DN retournées par la commande nodetool nous confirme que deux noeuds sont hors ligne. Dans le shell CQL, aucun message d'erreur, la requête est validée.

Notons que la grappe fonctionne toujours car le paramètre CONSISTENCY (cohérence des données) est ONE, ce qui signifie qu'il suffit d'avoir au moins un serveur opérationnel parmi les réplicas pour que l'écriture se fasse (ici, trois réplicas pour nous pour chaque requête d'écriture en temps normal).

Reprise avec un serveur de remplacement

Nous relançons l'un des deux noeuds hors ligne (noeud 2) et nous remplaçons l'autre (noeud 3) par une nouvelle machine.

Après avoir réinstallé une nouvelle machine, copié les fichiers de configuration, ordonné le remplacement de la machine «node3», au redémarrage les opérations de Apache Cassandra sont immédiatement lancées.

Voici un extrait du fichier journal cassandra.log de la nouvelle machine installée dans la grappe (noeud 3) :

```
INFO  12:45:24 Starting Messaging Service on /192.168.3.153:7000 (ens18)
INFO  12:45:25 Node /192.168.3.153 is now part of the cluster
INFO  12:45:25 Handshaking version with /192.168.3.153
INFO  12:45:25 InetAddress /192.168.3.153 is now UP
INFO  12:45:25 Starting up server gossip
INFO  12:45:26 Updating topology for all endpoints that have changed
INFO  12:45:26 JOINING: waiting for ring information

INFO  12:45:28 Initializing KeyspaceXYZ.users

INFO  12:45:29 JOINING: schema complete, ready to bootstrap
INFO  12:45:29 JOINING: waiting for pending range calculation
INFO  12:45:29 JOINING: calculation complete, ready to bootstrap
INFO  12:45:59 JOINING: Replacing a node with token(s): [-1032798858623236886,
-1068661564665005069,

INFO  12:45:59 JOINING: Starting to bootstrap...
INFO  12:46:00 ... Executing streaming plan for Bootstrap
INFO  12:46:00 ... Starting streaming to /192.168.3.152
INFO  12:46:00 ... Starting streaming to /192.168.3.154
INFO  12:46:00 ... Beginning stream session with /192.168.3.154
INFO  12:46:00 ... Starting streaming to /192.168.3.151
INFO  12:46:00 ... Beginning stream session with /192.168.3.152
INFO  12:46:00 ... Beginning stream session with /192.168.3.151
INFO  12:46:00 ... Prepare completed. Receiving 3 files(15426565 bytes),
sending 0 files(0 bytes)
INFO  12:46:00 ... Session with /192.168.3.151 is complete
INFO  12:46:01 ... Prepare completed. Receiving 7 files(13652159 bytes),
sending 0 files(0 bytes)
INFO  12:46:17 ... Session with /192.168.3.152 is complete
INFO  12:46:19 ... Session with /192.168.3.154 is complete
INFO  12:46:19 ... All sessions completed
INFO  12:46:19 Bootstrap completed! for the tokens [-1032798858623236886,
-1068661564665005069,

INFO  12:46:19 Node /192.168.3.153 state jump to NORMAL
```

Le fichier journal nous indique les étapes de réintégration du noeud 3 dans la grappe. Dans le cadre de notre projet NFE204, il n'y a que quelques Megaoctets de données à récupérer, cependant on peut constater que la reprise du service à la normale est très rapide.

Les extraits suivants montrent l'état du noeud 3 (192.168.3.153) au lancement du service Cassandra, avant récupération des données :

```
 root@node1[~]# nodetool status KeyspaceXYZ
Datacenter: dc1
===============
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address         Load       Tokens       Owns (effective)  Host ID      Rack
UN  192.168.3.152   19,64 MB   256          77,5%             ca043656-..  rack1
DN  192.168.3.153   73,06 KB   256          72,0%             c2985ee9-..  rack1
UN  192.168.3.154   16,54 MB   256          78,8%             f66c8325-..  rack1
UN  192.168.3.151   14,48 MB   256          71,7%             0a30946c-..  rack1
```

Et après avoir récupérer les données manquantes :

```
 root@node1[~]# nodetool status KeyspaceXYZ
Datacenter: dc1
===============
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address         Load       Tokens       Owns (effective)  Host ID      Rack
UN  192.168.3.152   19,64 MB   256          77,5%             ca043656-..  rack1
UN  192.168.3.153   19,58 MB   256          72,0%             c2985ee9-..  rack1
UN  192.168.3.154   16,54 MB   256          78,8%             f66c8325-..  rack1
UN  192.168.3.151   14,48 MB   256          71,7%             0a30946c-..  rack1
```

On retrouve bien tous nos enregistrements (c'était le cas aussi en mode dégradé car il restait un réplica opérationnel) :

```
cqlsh> use KeyspaceXYZ;
cqlsh:KeyspaceXYZ> SELECT COUNT(uid) FROM users;

 system.count(uid)
-------------------
             50001

(1 rows)
```

Conclusion de l'étude

Pour compléter le partitionnement réseau, nous devions mettre en place la réplication, ces deux mécanismes sont indissociables dans un environnement de production. Ces techniques autorisent l'installation rapide de machines à bas coût comme nous l'avons évoqué. Au moment de remplacer une machine, le technicien installe les fichiers de configuration de Cassandra, les mêmes que ceux de la machine voisine dans le Rack (pour notre cluster). Cela simplifie énormément la procédure de reprise sur panne : Apache Cassandra s'occupe ensuite de la «découverte» des nouveaux nœuds avec le protocole gossip. Le nouveau nœud est intégré dans le cercle et le Partitioner lui attribue la «tranche» de données dont il à la charge. Le mécanisme de réplication va chercher les données qui avaient été perdues en les copiant à partir des autres serveurs.

Gardons à l'esprit que les systèmes NoSQL apportent une réponse à des besoins bien spécifiques. Il est nécessaire d'avoir identifié au préalable la nécessité d'utiliser cette technologie.

Un système relationnel est adapté à un stockage de données dont la structure n'est pas ou peu évolutive dans le temps et l'espace. De même pour des recherches multi-critères, dans un système relationnel les requêtes avec des jointures font le travail. Cependant la modélisation de la base de données peut s'avérer compliquée. Compte tenu de la diversité des applications à travers le monde, les SGBDR sont encore largement majoritaires, ils offrent une souplesse au niveau de la richesse des requêtes SQL. Leur modèle transactionnel est une force incontestable car les opérations sont atomiques. En dernier point, les SGBD relationnels font de la cohérence des données un élément central du stockage d'informations.

Bien que la cohérence des données ne soit pas mise de côté sur les systèmes NoSQL, il faut ici pouvoir tolérer des incohérences temporaires. Ce phénomène est lié au mode «asynchrone», il a fallu adapté le concept de cohérence aux architectures distribuées. Ainsi, le choix d'un système NoSQL non relationnel se fera sans doute par le besoin de devoir organiser ses services et ses applications dans un système distribué à très grande échelle. À grande échelle parce que la charge de traitement et la quantité d'informations est gigantesque. Bien qu'ils soient utilisables avec une quantité faible d'informations à traiter, l'extensibilité et la capacité à s'adapter à un changement d'ordre de grandeur sont au centre de la philosophie de NoSQL et de Apache Cassandra.