Cassandra : partez sur une bonne base !

Cassandra est une base NoSQL orientée colonne et créée à l'origine par Facebook en s'appuyant sur deux papiers de recherche : BigTable de Google, DynamoDB d'Amazon.
Le projet a été libéré en 2008 et inclus à la fondation Apache dès 2009. Il est maintenant un top level project utilisé par un grand nombre d'entreprises. Nous pouvons citer par exemple NetFlix, Apple ou encore le CERN.

Matthieu Nantern,
Consultant chez Xebia

Concepts de base

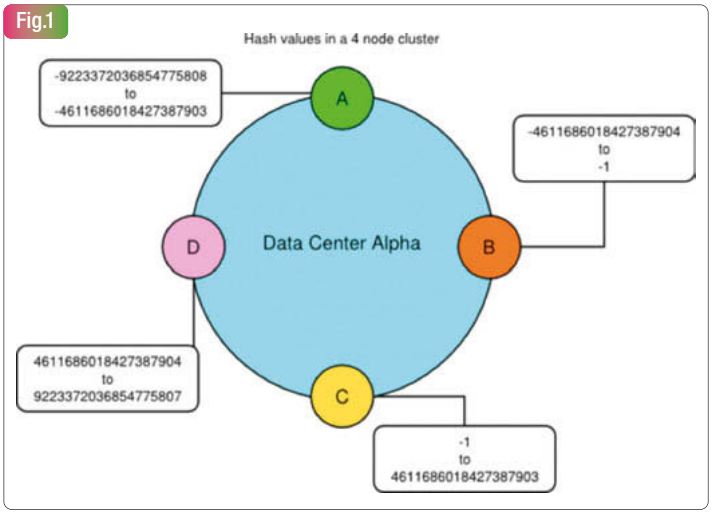
Dans ce chapitre, nous allons voir les principes de base de Cassandra, ceux qui en font une base NoSQL de plus en plus utilisée. La première caractéristique est sa scalabilite linéaire. Si vous avez besoin de servir plus de requêtes, il vous suffit d'ajouter plus de serveurs, pas besoin de mettre en place des mécanismes de réplication (compliqués) ou de re-écrire du code. C'est aussi valable pour la quantité de données : si vous voulez stocker plus de données il suffit d'ajouter des machines, et les données seront réparties sur votre cluster automatiquement.
Cette fonctionnalité est très intéressante dans le cas où vous ne connaissez pas la charge que vous aurez dans 1 ou 5 ans. Il existe à l'heure actuelle en production des clusters Cassandra allant de 3 machines à plus de 1000 (Netflix ou Apple) et gérant des péta-octets de données. La montée en charge est facilitée par une deuxième caractéristique de Cassandra : chaque serveur a le même rôle (architecture "masterless"). Il n'y a pas de serveur principal ou de serveur gérant la coordination. Un serveur assure l'ensemble des fonctionnalités de Cassandra. Ainsi, monter en charge revient toujours à ajouter le même type de serveur. De plus, Cassandra étant une base facile à installer (un jar et deux fichiers de configuration), elle est très bien acceptée par les équipes opérationnelles. Cassandra est également une base haute disponibilité qui peut tolérer (suivant votre configuration) la perte de plusieurs nœuds sans difficulté. Pour cela, comme de nombreuses autres bases NoSQL, Cassandra s'appuie sur la réplication des données sur de multiples nœuds. La réplication (asynchrone) entre plusieurs datacenters est même prévue nativement dans Cassandra : ainsi même la perte d'un datacenter n'entraînera pas d'interruption de votre service ! Ces avantages entraînent quelques contraintes au niveau des requêtes pouvant être effectuées sur Cassandra, ceci étant dû à sa nature distribuée. En effet, il n'est pas possible de réaliser des transactions ou des jointures sur notre cluster. Néanmoins, une bonne modélisation et une dénormalisation adaptée des données permettent le plus souvent de pallier à ces limitations.

Distribution des données

Chaque noeud du cluster Cassandra est en charge de gérer une partie des données. Cette répartition est effectuée via un hash de la clé de partition. Une clé primaire peut être composée d'une ou plusieurs colonnes (dans ce cas il s'agit d'une clé composée).
Le premier élément de la clé primaire, appelé clé de partition, détermine sur quel noeud du cluster la donnée sera stockée.
Par exemple, prenons les clés de partitions et les hash suivants :

Clé de partitionnement	Hash
jim	-2245462676723223822
suzy	1168604627387940318

Fig.1



Ici, la donnée ayant comme clé de partitionnement "jim" sera placée sur le noeud A (qui gère l'intervalle dans lequel est compris le hash de la clé) et la donnée "suzy" sur le noeud C Fig.1.
La réplication a ensuite lieu afin de copier les données sur d'autres noeuds afin d'assurer la haute disponibilité.
Les autres éléments d'une clé primaire composée, appelés "clustering columns" servent ensuite à ordonner les données sur un même noeud.
Voici un exemple :

```
CREATE TABLE grades (
 grade_id uuid,
 student_id  uuid,
 obtained_at  timestamp,
 grade   text,
 subject  text,
 PRIMARY KEY (student_id, obtained_at, grade_id)
);
```

Dans ce cas, la clé de partitionnement est "student_id" et les clustering columns sont "obtained_at" et "grade_id". Il est possible d'avoir plusieurs "clustering columns" pour une même table.
Les données appartenant à cette table sont placées sur un noeud en fonction de la colonne student_id puis triées sur le disque en fonction des colonnes "obtained_at" puis "grade_id".
Ainsi pour une table représentant les notes d'un élève, toutes celles-ci seront gérées par une même machine et classées en fonction de leur date d'obtention (de la plus ancienne à la plus récente) puis de leur identifiant.
Un dernier point important à souligner : il est possible de spécifier pour chaque requête le niveau de cohérence requis.
Comme tout système distribué, Cassandra est régi par le théorème CAP (voir encadré) et se place dans les domaines AP (Availability et Partition tolerance).

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.

Les bases relationnelles sont de type CA (Cohérence et Disponibilité), les bases NoSQL comme MongoDB ou HBase sont de type CP tandis que Cassandra ou Riak sont de type AP.

Il existe plusieurs types de cohérence utilisables pour chaque requête. Les plus communément utilisés sont :

- ALL : le cluster attend la réponse de tous les noeuds devant écrire ou lire la donnée avant de répondre au client,
- ONE : le cluster attend seulement la réponse d'un des noeuds avant de répondre au client,
- QUORUM : le quorum est défini comme la majorité absolue des réplicas. Si la donnée est répliquée 3 fois, alors le cluster attendra la réponse de 2 noeuds avant de répondre au client. Si la donnée est répliquée 4 fois alors le quorum est constitué de 3 machines.

Le choix de la cohérence est important et doit être déterminé en fonction des cas métiers et de la performance attendue. Le niveau ONE est le plus performant mais se fait au détriment de la fraîcheur des données, tandis que le niveau ALL offre la plus forte cohérence mais au détriment de la haute disponibilité. Pour que cela soit plus parlant, prenons un exemple : nous répliquons les données 3 fois au sein de notre cluster pour notre table "grades" définie ci-dessus.

Cas 1 : On écrit et on lit avec un niveau de cohérence "ONE": Fig.2.
Si un client écrit la valeur B sur l'un des noeuds et qu'un autre client lit la valeur alors que la réplication est toujours en cours il lira une ancienne valeur (A en l'occurrence). L'écriture tout comme la lecture sera extrêmement performante (on n'attend la réponse que d'un seul serveur) mais avec le risque de ne pas toujours avoir la donnée la plus récente.

Cas 2: On écrit avec le niveau "ONE" et on lit avec le niveau "ALL" Fig.3.
Ici, on conserve la performance en écriture (un seul noeud doit persister la donnée avant de répondre au client) et on obtient toujours la donnée la plus fraîche en lecture. Cependant, ceci a un coût non négligeable, le cluster doit attendre d'avoir la réponse des trois noeuds avant de répondre au client; en plus, si l'on perd un serveur, il ne pourra plus honorer la lecture (car Cassandra attend la réponse de 3 serveurs avant de répondre au client).

Cas 3: On écrit et on lit avec le niveau "QUORUM" Fig.4.
Ce cas représente un bon compromis entre la performance et la disponibilité des données. Les écritures sont un peu moins performantes que précédemment (le cluster attend deux réponses au lieu d'une dans les cas précédents), mais les lectures ont l'assurance d'obtenir la donnée la plus à jour et il est possible de perdre machine sans que cela affecte la disponibilité des données.

Installation

Passons maintenant à un peu plus de pratique ! Nous avons vu dans le chapitre précédent que la facilité d'installation de Cassandra était un de ses points forts, réalisons donc cette installation sur une Ubuntu 14.04. La façon la plus rapide est d'utiliser le dépôt de paquets mis à disposition par Datastax. Les seuls pré-requis sont un JRE en version 7 (celui d'Oracle est recommandé) et JNA pour les installations de production. On ajoute ensuite le dépôt community, la clé publique, puis on installe Cassandra après avoir mis à jour la liste de nos dépôts :

```
$ echo "deb http://debian.datastax.com/community stable main" | sudo tee -a /etc/apt/sources.list.d/cassandra.sources.list
$ curl -L http://debian.datastax.com/debian/repo_key | sudo apt-key add -
$ sudo apt-get update
$ sudo apt-get install cassandra=2.0.12
```

Et voilà, Cassandra est installé et démarré ! On va maintenant s'y connecter et créer notre keyspace :

```
$ cqlsh
Connected to trace_consistency at 1.2.3.4:9160.
[cqlsh 4.1.1 | Cassandra 2.0.9 | CQL spec 3.1.1 | Thrift protocol 19.39.0]
Use HELP for help.
cqlsh> CREATE KEYSPACE school WITH REPLICATION = { 'class' : 'SimpleStrategy', 'replication_factor' : 1 };
```

Comme nous n'avons qu'une machine dans notre cluster nous demandons à Cassandra de ne pas répliquer la donnée (paramètre 'replication_factor'). Le langage utilisé est proche du SQL et se nomme CQL ("Cassandra Query Language"). Nous avons maintenant une base, il ne nous reste plus qu'à l'utiliser dans notre code.

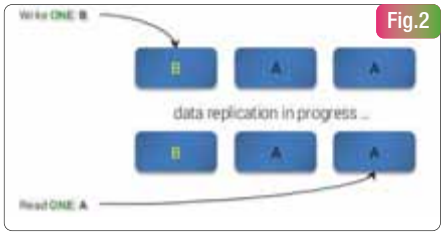
Premiers tests

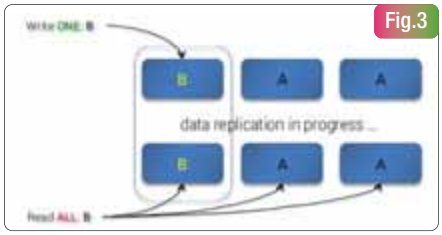
De nombreux drivers existent pour s'interfacer avec Cassandra, C#, Java, NodeJs, Ruby, Python, etc.
Pour la suite de cet article nous allons utiliser le driver Java développé par Datastax qui offre de nombreuses fonctionnalités :

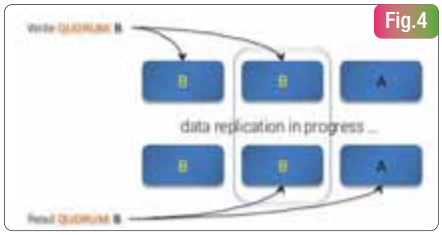
- Découverte de la topologie du cluster : le driver en se connectant à un cluster identifie l'ensemble des machines, et peut ainsi s'adresser directement au noeud contenant la donnée requêtée,
- Failover transparent : en cas de défaillance d'un nœud, le driver essaye automatiquement d'autres machines, puis tente de joindre à nouveau la machine défaillante en tâche de fond,
- Asynchrone : il est possible avec le driver de profiter des fonctionnalités asynchrones offertes par Cassandra,
- Compatible avec toutes les fonctionnalités de Cassandra.

Notre cas d'utilisation sera simple : nous allons nous connecter à notre machine Cassandra, créer une table, y insérer des données puis les lire avec le driver Java. La première étape est d'ajouter le driver à notre projet. Cela se fait simplement via une dépendance Maven :

```
<dependency>
 <groupId>com.datastax.cassandra</groupId>
```







```
<artifactId>cassandra-driver-core</artifactId>
<version>2.1.4</version>
</dependency>
```

Pour nous connecter à notre machine Cassandra, définissons deux méthodes :

- "connect" : permettant d'ouvrir la connexion vers un cluster Cassandra. Cette méthode utilise un Builder auquel on ajoute la liste des noeuds de notre cluster. Il n'est pas nécessaire de tous les indiquer, le driver prenant à sa charge la découverte complète du cluster.

```
public void connect(String node) {
   cluster = Cluster.builder()
       .addContactPoint(node)
       .build();
}
```

- "close" : fermant la connexion.

```
public void close() {
   cluster.close();
}
```

Ensuite, intégrez la classe complète avec un "main" permettant de l'exécuter :

```
package fr.xebia.cassandra;

import com.datastax.driver.core.Cluster;
import com.datastax.driver.core.Host;
import com.datastax.driver.core.Metadata;

public class SimpleClient {
 private Cluster cluster;

 public void connect(String node) {
   cluster = Cluster.builder()
       .addContactPoint(node)
       .build();
 }

 public void close() {
   cluster.close();
 }

 public static void main(String[] args) {
   SimpleClient client = new SimpleClient();
   client.connect("127.0.0.1");
   client.close();
 }
}
```

Cette classe va se connecter à notre machine puis fermer la connexion. Pas très utile... Ajoutons donc la création d'une table au keyspace "school" déclaré plus haut. Pour exécuter des requêtes, il nous faut obtenir une session du cluster déclaré ci-dessus. Cela se fait par le biais de l'appel de la méthode "connect" sur notre objet Cluster :

```
public void connect(String node) {
   cluster = Cluster.builder()
       .addContactPoint(node)
       .build();
   session = cluster.connect();
}
```

Une fois notre session obtenue, nous pouvons créer une table destinée à stocker les notes de nos élèves :

```
public void createSchema() {
   session.execute(
       "CREATE TABLE IF NOT EXISTS school.grades (" +
           "grade_id uuid," +
           "student_id uuid," +
           "obtained_at timestamp," +
           "grade text," +
           "subject text," +
           "PRIMARY KEY (student_id, obtained_at, grade_id)" +
           ");");
}
```

Puis insérez une donnée :

```
public void loadData() {
   session.execute(
      "INSERT INTO school.grades (grade_id, student_id, obtained_at, grade, subject) " +
       "VALUES (" +
          "756716f7-2e54-4715-9f00-91dcbea6cf50," +
          "5dad4ece-b8bb-46b0-ab5d-22cce12527dd," +
          "'2015-01-12T16:02:51Z'," +
          "'B+'," +
          "'chinese')" +
          ";");
}
```

Enfin, intégrez la classe complète :

```
package fr.xebia.cassandra;

import com.datastax.driver.core.Cluster;
import com.datastax.driver.core.Host;
import com.datastax.driver.core.ResultSet;
import com.datastax.driver.core.Row;
import com.datastax.driver.core.Session;

public class SimpleClient {
 private Cluster cluster;
 private Session session;

 public Session getSession() {
   return this.session;
 }

 public void connect(String node) {
   cluster = Cluster.builder()
       .addContactPoint(node)
       .build();
   session = cluster.connect();
 }

public void createSchema() {
   session.execute(
       "CREATE TABLE IF NOT EXISTS school.grades (" +
           "grade_id uuid," +
           "student_id uuid," +
           "obtained_at time," +
           "grade text," +
```

```
            "subject text," +
            "PRIMARY KEY (student_id, obtained_at, grade_id)" +
            ");");
    }

    public void loadData() {
        session.execute(
            "INSERT INTO school.grades (grade_id, student_id, obtained_at, grade, subject) " +
            "VALUES (" +
                "756716f7-2e54-4715-9f00-91dcbea6cf50," +
                "5dad4ece-b8bb-46b0-ab5d-22cce12527dd," +
                "'2015-01-12T16:02:51Z'," +
                "'B+'," +
                "'chinese')" +
                ";");
    }

    public void close() {
        session.close();
        cluster.close();
    }

    public static void main(String[] args) {
        SimpleClient client = new SimpleClient();
        client.connect("127.0.0.1");
        client.createSchema();
        client.loadData();
        client.close();
    }
}
```

Nous allons maintenant lire la donnée que l'on vient d'insérer. Cette fois-ci nous utiliserons plutôt une requête préparée qui n'a besoin d'être parsée qu'une seule fois par le cluster, ce qui améliore les performances.
Trois étapes sont nécessaires :
1 - Déclarer la requête préparée,
2 - Associer des valeurs au paramètre,
3 - Exécuter la requête préparée.
Ces trois étapes sont réalisées via la méthode ci-dessous :

```
public void queryData() {
    // Etape 1
    PreparedStatement statement = getSession().prepare(
        "SELECT * from school.grades WHERE student_id = ?;");

    // Etape 2
    BoundStatement boundStatement = new BoundStatement(statement);
    boundStatement.bind(UUID.fromString("5dad4ece-b8bb-46b0-ab5d-22cce12527dd"));

    // Etape 3
    ResultSet rs = getSession().execute(boundStatement);
    for (Row row : rs) {
        System.out.println(row.getString("subject")+" - "+ row.getString("grade"));
    }
}
```

Un cas pratique

Maintenant que nous avons installé et vu les bases de la programmation avec Cassandra, passons à un cas pratique. Nous souhaitons modéliser des élèves qui disposeront de quelques champs :

- Un nom,
- Un prénom,
- Un email unique servant d'identifiant et permettant d'accéder à notre site,
- Une liste de cours.

Ces élèves auront des notes, et l'on souhaite pouvoir obtenir l'ensemble des notes d'un élève pour une plage de temps (un trimestre ou une année par exemple). La première étape et la plus importante est de modéliser correctement les tables sous Cassandra.

Modélisation des tables

Un point crucial avec Cassandra, trop souvent oublié, est la nécessité d'adapter la structure des tables de notre base aux requêtes que l'on souhaite faire. Une bonne modélisation est cruciale afin d'obtenir des requêtes performantes et de profiter des points forts de Cassandra. Ainsi les écritures sont peu coûteuses et de ce fait, il vaut mieux privilégier la duplication de données immutables afin d'éviter de faire une lecture supplémentaire.

Gestion des notes

Nous souhaitons maintenant gérer dans notre système les notes des élèves. Chaque objet Note est composé d'un identifiant technique, d'un élève à qui elle est liée, d'une valeur (la note), d'une matière et de la date d'obtention de la note. Nous souhaitons pouvoir consulter toutes les notes d'un élève, éventuellement entre deux dates définies.
Dans une modélisation pour un SGBDR classique, nous aurions probablement une table note ayant :

- Comme clé primaire l'identifiant technique,
- Comme clé étrangère l'identifiant de l'élève lié,
- Et ensuite une colonne pour les champs note, matière et date.

Dans Cassandra, une telle modélisation est possible mais sera au mieux lente. Si une telle modélisation était utilisée dans notre cas, nous aurions alors une ligne par note et obtenir toutes les notes d'un élève obligerait à utiliser un index secondaire :

```
CREATE TABLE grades (
 grade_id uuid,
 student_id  uuid,
 obtained_at  timestamp,
 grade   text,
 subject  text,
 PRIMARY KEY (grade_id)
);
CREATE INDEX student_id ON grades (student_id);
```

Voici la requête correspondante :

```
SELECT * FROM grades WHERE student_id = "<my_id>"
```

Cela fonctionne, mais ce ne sera ni performant, ni scalable car l'index secondaire oblige à interroger tous les noeuds de notre cluster Cassandra afin de vérifier qu'il dispose de données. Il faut vraiment limiter l'utilisation des index secondaires à des cas exceptionnels et pour des tables ne comprenant que peu d'éléments.
Une bien meilleure modélisation est de profiter de la performance de Cassandra sur des lignes ayant de nombreuses colonnes et donc de ranger chaque note d'un élève dans une colonne, puis d'utiliser l'identifiant de l'élève comme clé de partitionnement comme dans notre table ci-dessous :

```
CREATE TABLE grades (
 grade_id uuid,
 student_id  uuid,
```

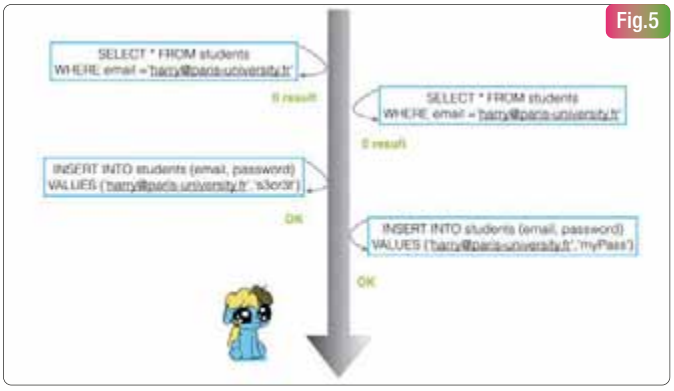


Fig.5

```
 obtained_at  timestamp,
 grade   text,
 subject   text,
 PRIMARY KEY (student_id, obtained_at, grade_id)
);
```

On ajoute également en tant que clustering column “obtained_at” afin que les notes soient triées sur disque en fonction de la date d’obtention de la note. Cela permet de s'appuyer sur la manière dont Cassandra range les données afin d'offrir de bonnes performances pour notre requête. Enfin, la dernière clustering column “grade_id” est un identifiant technique permettant de s’assurer que notre clé primaire est bien unique. Obtenir toutes les notes d’un élève est tout aussi facile qu’avec un index secondaire mais n’interroge plus qu’un noeud du cluster (celui hébergeant les données pour l’étudiant) :

```
SELECT * FROM grades WHERE student_id =“<my_id>”
```

Obtenir toutes les notes d'un élève pour les deux premiers mois de l’année est tout aussi aisé :

```
SELECT * FROM grades WHERE student_id =“<my_id>”
AND obtained_at >=“2015-01-01 00:00:00”
AND obtained_at <“2015-03-01 00:00:00”
```

Cette requête est performante car la clé de partitionnement permet à Cassandra de déterminer le serveur du cluster hébergeant la donnée. Ensuite, comme les données sont triées sur disque dans l’ordre des “clustering columns”, réaliser des requêtes sur la date d’obtention (notre première clustering column) revient à obtenir une partie contiguë d'un fichier de données. La lecture séquentielle est bien plus performante sur un disque dur que des lectures aléatoires, ce qui permet d’obtenir une bonne latence sur cette interrogation. Ce type de requête est appelée une "slice query" et doit être privilégiée le plus possible.

Gestion des étudiants

Nous avons maintenant une structure robuste et scalable afin de gérer les requêtes nous ayant été demandées sur les notes d’un élève. Afin de remplir complètement notre cas pratique, il nous reste à gérer notre table étudiant avec deux contraintes à respecter :

- Il ne peut pas y avoir deux étudiants avec le même email dans notre base Cassandra,
- On veut pouvoir ajouter des cours à un étudiant.

Pour gérer notre première contrainte une implémentation naïve serait l’algorithme suivant:

- On réalise un SELECT afin de vérifier que l’email de l’étudiant n’est pas encore utilisé,
- Si le SELECT ne renvoie pas de valeur on peut alors réaliser l’INSERT afin de créer le nouvel étudiant.

Cela fonctionne bien tant que deux étudiants n’essayent pas de s’inscrire en même temps. Sinon, nous nous retrouvons dans la situation suivante : Fig.5. Une réponse positive a été envoyée au premier étudiant car l’insertion a été réalisée avec succès. Cependant, il va ensuite tenter de se loguer à son compte et cela ne fonctionnera pas.
Dans Cassandra, le dernier processus ayant écrit a raison.
Afin d’éviter ce cas problématique il existe depuis le début de la version 2 de Cassandra, les lightweight transactions. Ce mécanisme, se basant sur Paxos, permet de s’assurer que l’insertion n’a lieu que si l’étudiant n’existe pas. Il permet l’écriture d'une requête de ce type :

```
INSERT into students VALUES (...) IF NOT EXISTS;
```

Dans le cas où l’étudiant existe déjà, une réponse négative sera envoyée au client au lieu d’écraser les valeurs précédemment insérées.
Il est important de ne pas abuser de cette fonctionnalité, qui implique quatre allers-retours réseaux au lieu d’un seul au sein du cluster Cassandra. La dernière tâche à effectuer avant de considérer notre cas pratique comme résolu est de permettre l’ajout d’un cours à un étudiant. Nous allons tenter l’approche naïve :

- On lit dans Cassandra les données de notre étudiant puis l’on ajoute le nouveau cours à notre utilisateur,
- On insère les nouvelles données dans Cassandra.

Lire avant d’écrire est considéré comme un anti-pattern dans Cassandra et cela pour deux raisons :

- D’un point de vue performance, cela oblige à effectuer deux requêtes sur le cluster,
- D’un point de vue cohérence des données, comme vu dans le paragraphe précédent, on peut perdre certaines mises à jour. La bonne façon de traiter ce cas est d’utiliser les données de type collection dans Cassandra. Par exemple un Set (une liste unique d’éléments) sera particulièrement adapté pour la gestion des cours de notre élève :

```
CREATE TABLE students (
 email text PRIMARY KEY,
 first_name text,
 last_name text,
 courses set<text>
);
```

Avec une telle structure, ajouter un cours peut se faire sans lecture préalable :

```
UPDATE students
 SET courses = courses + {'chinese'} WHERE email = 'harry@paris-university.fr';
```

De la même manière la suppression peut se faire facilement :

```
UPDATE students
 SET courses = courses - {'chinese'} WHERE email = 'harry@paris-university.fr';
```

Et voilà! Plus de lecture avant les écritures !

Aller plus loin

Au cours de cet article nous avons vu les bases nécessaires afin de bien débuter avec Cassandra : les concepts de cette base NoSQL, comment les données sont stockées au sein du cluster, comment installer et utiliser notre base avec un client Java, et, enfin un cas de modélisation des données avec les erreurs classiques à éviter. Cependant, il est possible de faire bien plus avec Cassandra :

- Déployer et faire évoluer son cluster en production afin de gérer toujours plus de trafic,
- Mettre en place du Spark avec Cassandra afin de faire du machine learning sur toutes nos données,
- Faire de l’élection de leader avec Cassandra.

Nous vous laissons découvrir ceci.