Utiliser Cassandra avec Spark

Jonathan Lejeune

Ce document vous permet de vous familiariser avec l'API du driver Spark qui permet de faire l'interface avec Cassandra. Vous devez donc avoir préalablement installé le driver Spark-Cassandra (exercice 2 TP préparatoire Cassandra).

Dans ce document on se basera sur une table `etudiant` dans un keyspace `test` qui nous servira d'exemple tout au long de ce document. On suppose que cette table est créé avec l'instruction CQL suivante :

```
CREATE TABLE test.etudiant (numero INT PRIMARY KEY, nom VARCHAR, prenom VARCHAR);
```

Utiliser l'API de driver

Pour pouvoir utiliser l'API du driver dans un programme Spark, il faut que le code importe l'ensemble des types du package `com.datastax.spark.connector`.

Ainsi il faudra ajouter :

```
import com.datastax.spark.connector._;
```

Se connecter à Cassandra avec Spark

Pour indiquer à Spark comment se connecter au cluster Cassandra il faut paramétrer la `SparkConf` du `SparkContext` en précisant un pair (ou éventuellement plusieurs pairs du cluster séparés par une virgule) dans la property `spark.cassandra.connection.host`. Si besoin il est possible de préciser le login et le mot de passe de l'utilisateur Cassandra avec les properties `spark.cassandra.auth.username` et `spark.cassandra.auth.password` (inutile normalement dans notre cadre, car nous utiliserons l'utilisateur anonyme qui ne nécessite pas de login). Ainsi dans notre cas il faut faire :

```
val conf = new SparkConf().setAppName("Spark on Cassandra")
                .setMaster("url du spark master")
                .set("spark.cassandra.connection.host", "pair1,pair2,pair3,...")
val sc = new SparkContext(conf)
```

Charger une table Cassandra en tant que RDD

Première méthode

Pour charger une table de Cassandra dans un RDD il faut utiliser la méthode `cassandraTable` du sparkcontext. Cette méthode prend deux arguments explicites de type String qui sont respectivement le keyspace et le nom de la table à charger. Le résultat sera de type `RDD[CassandraRow]`. Un objet de type `CassandraRow` contient le contenu d'une ligne d'une table Cassandra. Pour accéder aux différents champs de la ligne, il faut utiliser des méthodes `get<TypeDeDonnée>(..)` qui peuvent être paramétrer soit par le nom de la colonne dans la base, soit par l'indice de colonne.

Dans notre exemple on ferait :

```
val rdd_table_etudiant:RDD[CassandraRow] = sc.cassandraTable("test","etudiant")
val rdd_tuple:RDD[(Int,String,String)] = rdd_table_etudiant.map(
                        cr => (cr.getInt("numero"),
                        cr.getString("nom"),
                        cr.getString("prenom")))
```

Seconde méthode

Il existe une autre manière moins fastidieuse (c.-à-d. sans faire appel aux différents getters) de récupérer les données d'une ligne. Il suffit de déclarer une case classe dont les noms, les types et l'ordre des attributs correspondent parfaitement aux colonnes de la table. Lors de l'appel à la méthode `cassandraTable`, il faut préciser le type de cette case classe entre crocheta avant les arguments. Le RDD renvoyé contiendra ainsi directement des éléments ayant pour type la case classe. Chaque élément correspondra donc à une ligne de la table.

Dans notre exemple on ferait :

```
case class Etudiant(numero:Int, nom:String, prenom:String)

val rdd_etudiant:RDD[Etudiant] = sc.cassandraTable[Etudiant]("test","etudiant")
```

NB : Dans l'absolu, il n'est pas obligatoire que `Etudiant` soit une case classe mais une classe qui implante `Serializable` et un constructeur dont les paramètres correspondent aux type des colonnes.

Sauvegarder un RDD dans une table Cassandra

L'import des types de `com.datastax.spark.connector` vous permet d'appeler sur n'importe quel RDD les méthodes `saveToCassandra` et `saveAsCassandraTable` qui permettent respectivement de remplir/modifier une table déjà existante et de sauvegarder le RDD dans une nouvelle table.

Ces méthodes 4 arguments dont deux arguments ayant une valeur par défaut :

- `keyspaceName:String` ⇒ le nom du key space
- `tableName:String` ⇒ le nom de la table où l'on souhaite envoyer les données. Dans le cas de `saveAsCassandraTable`, la table ne doit pas exister.

- `columns: ColumnSelector` (valeur par défaut : `AllColumns`) ⇒ permet de décrire la correspondance entre les données du RDD et les colonnes de la table (cf. paragraphe dédié ci-dessous).
- `writeConf: WriteConf` (valeur par défaut issue du SparkConf) ⇒ permet d'ajouter des paramètres de configuration lié à la requête Cassandra (ex : niveau de cohérence attendu)

Le type `ColumnSelector`

Le type `ColumnSelector` permet de décrire la correspondance qu'il y a entre des objets contenus dans un RDD et le schéma d'une table Cassandra. Un objet d'un RDD représente une ligne de la table que l'on souhaite modifier ou ajouter dans la table cible.

Le sélecteur `AllColumns`

Le sélecteur par défaut est l'`object` (= singleton) `AllColumns` où chaque colonne de la table doit être référencée par les attributs des objets du RDD. Le mapping se fait automatiquement :

- soit par un tuple où l'ordre des types des éléments doit correspondre l'ordre des types des colonnes
- soit le type des objets du RDD est une classe où le nom des attributs correspond aux noms des colonnes. Le type d'un attribut peut être différent du type de sa colonne s'il est possible de le convertir (ex : `Int` vers `DOUBLE`, `String` vers `INTEGER` si la valeur de la chaine est un nombre entier, `Int` vers `VARCHAR`).

Dans notre exemple, on pourrait faire ceci :

```
val sc = new SparkContext(...)
val etudiants = Seq(Etudiant(2,"Durand","Julien"),Etudiant(5,"Legrand","Thomas"))
val rdd = sc.parallelize(etudiants)
rdd.saveToCassandra("test","etudiant")
```

Ou bien ceci

```
val sc = new SparkContext(...)
val etudiants = Seq((2,"Durand","Julien"),(5,"Legrand","Thomas"))
val rdd = sc.parallelize(etudiants)
rdd.saveToCassandra("test","etudiant")
```

Le sélecteur `SomeColumns`

Le sélecteur `SomeColumns` est une case classe qui prend en argument de son constructeur une liste de références de colonne. Ceci permet de désigner un sous-ensemble de colonnes de la table et ainsi avoir la possibilité de ne pas renseigner les valeurs de toutes les colonnes. **Il est cependant obligatoire de renseigner les colonnes de clé primaire** afin de savoir si l'écriture sera l'ajout d'une nouvelle ligne ou bien une modification d'une ligne existante. Les colonnes non renseignées et qui ne sont pas des clés primaires auront une valeur nulle.

Dans notre cadre nous référencerons une colonne par son nom (columnKey) dans la table. Il est possible de définir manuellement la correspondance entre les attributs des objets du RDD et des colonnes si celle-ci n'est pas évidente en utilisant des alias. La syntaxe à utiliser est la suivante :

```
SomeColumns("nom_colonne" as "nom_attribut1" , ...)
```

Dans notre exemple si l'on souhaite juste changer le prénom de l'étudiant d'identifiant 42, nous ferions ceci :

```
val sc = new SparkContext(...)
val etudiant = Seq((42,"Bernard"))
val rdd = sc.parallelize(etudiant)
rdd.saveToCassandra("test","etudiant",
      SomeColumns("numero" as "_1", "prenom" as "_2" ))
```

Si la colonne de la table est une collection (`LIST`, `MAP` ou `SET`) et si c'est une modification de ligne, il est possible de préciser l'opération que l'on souhaite faire en ajoutant dans la définition de la colonne l'un des modificateurs suivant :

- `overwrite` (par défaut) : efface et remplace la collection
- `append` ou `add` : ajoute les éléments dans la collection
- `prepend` (seulement pour les `LIST`) : ajoute les éléments au début de la liste
- `remove` (seulement pour les `LIST` et `SET`) : supprime les éléments de la collection