Big Data Analytics

Résumé - Séances 2 et 3

Apache Spark APIs et Spark SQL

Pr EL GHALI Btihal

18 janvier 2026

Table des matières

1 Introduction aux APIs Spark

1.1 Spark Shell

Le shell de Spark fournit un moyen simple d'apprendre l'API et un outil puissant pour analyser les données de manière interactive.

1.2 SparkSession

— Dans les versions antérieures : SparkContext était le point d'entrée

— Nécessitait différents contextes : StreamingContext, SQLContext, HiveContext

— **SparkSession** : combinaison unifiée de tous ces contextes

— Point d'entrée unique pour toutes les fonctionnalités Spark

1.3 Hiérarchie des APIs

L'écosystème des APIs Spark comporte 3 niveaux :

1. **APIs bas-niveau** : RDDs (Resilient Distributed Dataset)
2. **APIs haut-niveau** : Datasets, DataFrames et SQL
3. **Bibliothèques avancées** : Structured Streaming, Advanced Analytics, MLlib, GraphX

1.4 Data Sources API

— Fournit un mécanisme modulaire pour accéder aux données structurées

— Permet de lire et stocker des données structurées et semi-structurées

— Plus que de simples canaux de conversion de données

2 RDD - Resilient Distributed Dataset

2.1 Définition

RDD signifie :

— **Resilient** : Tolérant aux pannes, capable de reconstruire les données

— **Distributed** : Données distribuées entre les nœuds du cluster

— **Dataset** : Collection de données partitionnées avec des valeurs

Structure fondamentale de Spark :

— Collection immuable d'objets distribués

— Divisé en partitions logiques calculées sur différents nœuds

— Calculé à partir d'une source de données et placé en mémoire RAM

— Peut contenir n'importe quel type d'objets (Python, Java, Scala)

2.2 Création des RDDs

Deux méthodes principales :

1. **Parallélisation d'une collection existante**

```
val somedata = Array(1, 2, 3, 4, 5)
val distributedData = sc.parallelize(somedata)
```

2. **Référence à un système de stockage externe**

```
val distributedFile = sc.textFile("test.txt")
```

2.3 Caractéristiques des RDDs

— Données immuables
— Calcul en mémoire
— Tolérance aux pannes
— Évaluation paresseuse (lazy evaluation)
— Données distribuées et structurées
— Peut être manipulé par plusieurs opérations
— Peut être persisté
— Support de multiples sources de données

2.4 DAG - Graphe Acyclique Dirigé

— DAG = ensemble d'étapes représentant les opérations sur le RDD
— Créé lorsqu'une action est appelée sur le RDD
— Permet l'optimisation de l'exécution

3 Opérations sur les RDDs

3.1 Transformations

Principe fondamental :

— Les collections sont immuables
— Les transformations créent de nouveaux RDDs
— Chaîne de collections constituant les étapes du traitement
— Évaluation paresseuse : exécutées seulement lors d'une action

3.1.1 Transformations Étroites (Narrow)

— Tous les éléments requis résident dans une partition unique du RDD parent
— Pas de shuffle de données entre partitions
— **Exemples :** map, flatMap, filter

```
val file = sc.textFile("test.txt")
val words = file.flatMap(l => l.split("␣"))
```

3.1.2 Transformations Larges (Wide)

— Éléments requis peuvent résider dans plusieurs partitions du RDD parent
— Nécessite un shuffle de données
— **Exemples :** groupByKey(), reduceByKey()

```
val words = file.flatMap(l => l.split("␣"))
                .map(word => (word, 1))
words.reduceByKey(_ + _).collect
```

3.2 Actions

Caractéristiques :

— Produisent des valeurs (contrairement aux transformations)

— Déclenchent l'exécution du graphe de transformations

— Permettent l'évaluation paresseuse et l'optimisation

Actions principales :

— `collect()` : Retourne tous les éléments

— `count()` : Compte le nombre d'éléments

— `first()` : Retourne le premier élément

— `take(n)` : Retourne les n premiers éléments

— `reduce()` : Agrège les éléments

— `foreach()` : Applique une fonction à chaque élément

Exemple - WordCount :

```
val f = sc.textFile("test.txt")
val words = f.flatMap(l => l.split("␣"))
             .map(word => (word, 1))
words.reduceByKey(_ + _).collect.foreach(println)
// Resultat: (this,2) (is,2) (test,2) (a,2)
```

Exemple - Calcul de longueur :

```
val lines = sc.textFile("test.txt")
val lineLengths = lines.map(s => s.length)
val totalLength = lineLengths.reduce(_ + _)
```

4 Persistance des RDDs

4.1 Principe

— Par défaut, les RDDs sont transitoires (non conservés en mémoire)

— Possibilité de marquer un RDD comme persistant

— Stockage en RAM pour réutilisation ultérieure

— Évite de recalculer le RDD à chaque utilisation

4.2 Niveaux de stockage

Chaque RDD peut être stocké avec différents niveaux via `persist()` :

— `MEMORY_ONLY` : En mémoire uniquement

— `MEMORY_AND_DISK` : Mémoire + disque si nécessaire

— `DISK_ONLY` : Sur disque uniquement

— `MEMORY_ONLY_SER` : Sérialisé en mémoire

Exemple :

```
val w = f.flatMap(l => l.split("␣"))
         .map(word => (word, 1)).cache()
w.reduceByKey(_ + _).collect
```

4.3 Tolérance aux pannes

— Si panne sur un fragment persistant : relance à partir de ce fragment

— Si panne sur un RDD non persistant : réapplication de la chaîne complète

— Spark conserve l'historique des opérations pour reconstitution

5 Limitations des RDDs

1. **Pas de moteur d'optimisation**
   — Ne peut pas utiliser Catalyst Optimizer
   — Ne bénéficie pas de Tungsten execution engine
2. **Dégradation des performances**
   — Performance réduite quand mémoire insuffisante
   — Surcharge de Garbage Collection
3. **Traitement des données structurées**
   — RDD = "boîte non typée" sans préjugé de structure
   — Pas de manipulation fine des constituants
   — Programmeur doit fournir toutes les fonctions de filtrage

6 Spark SQL

6.1 Définition et objectifs

Apache Spark SQL :

— Module Spark pour simplifier l'utilisation de données structurées

— Utilise les abstractions DataFrame et Dataset

— Disponible en Python, Java, Scala et R

— Collections distribuées organisées en colonnes nommées

— Optimisation via **Catalyst Optimizer**

6.2 Évolution et performance

— Intègre traitement relationnel + programmation fonctionnelle

— Remplace Apache Hive avec meilleures performances

— Plus rapide que Hive en termes de vitesse de traitement

— API universelle pour chargement et stockage de données structurées

6.3 Architecture de Spark SQL

6.3.1 Composants principaux

1. **API de langages**
   - Support : Python, Scala, Java, R
   - Interrogation depuis programmes Spark ou outils externes
   - Connecteurs JDBC/ODBC standard
2. **Console SQL**
   - Accessible via commande `SPARK-SQL`
   - Exécution directe de requêtes SQL
3. **Compatibilité Hive**
   - Réutilise Hive MetaStore
   - Compatible avec données, requêtes et UDF Hive existants
4. **Connectivité JDBC/ODBC**
   - Mode serveur avec connectivité standard
   - Connexion des outils BI (Tableau, Pentaho)

6.3.2 Catalyst Optimizer

Caractéristiques :

- Basé sur programmation fonctionnelle en Scala
- Composant le plus avancé de Spark SQL
- Framework général pour transformation d'arbres
- Optimisation basée sur les coûts (temps + ressources)
- Optimisation basée sur les règles
- Bibliothèque modulaire avec règles spécialisées
- Rend les requêtes beaucoup plus rapides que RDD

Fonctions :

- Analyse et évaluation
- Optimisation des requêtes
- Planification d'exécution
- Génération du plan d'exécution

6.3.3 API de sources de données

Support de multiples formats :

- Fichiers Parquet
- Fichiers CSV
- Documents JSON
- Tables Hive
- Base de données Cassandra
- Bases de données externes

6.4 Fonctionnalités principales

— **Intégration avec Spark** : Utilisation dans programmes Spark

— **Accès uniforme aux données** : API unique pour diverses sources

— **Connectivité standard** : JDBC/ODBC

— **Compatibilité Hive** : Réutilisation des ressources Hive

— **Performances et évolutivité** : Optimisations avancées

— **UDF** : Fonctions définies par l'utilisateur

6.5 Exemple - UDF (User Defined Function)

```
// Creation du dataset
val dataset = Seq((0, "hello"),(1, "world"))
              .toDF("id","text")

// Definition de la fonction
val upper: String => String = _.toUpperCase

// Import et creation de l'UDF
import org.apache.spark.sql.functions.udf
val upperUDF = udf(upper)

// Application et affichage
dataset.withColumn("upper", upperUDF('text)).show
```

7 Évolution des APIs Spark

7.1 Timeline historique

Année	Version	API
2011	Spark 1.0	RDD
2013	Spark 1.3	DataFrame
2015	Spark 1.6	DataSet
2016	Spark 2.0	Fusion DataFrame + DataSet
2020	Spark 3.0	Améliorations multiples

8 DataFrames

8.1 Définition

— Collection de données distribuée et immuable (comme RDD)

— Données organisées en **colonnes nommées**

— Similaire à une table de base de données relationnelle

— API disponible en Scala, Java, Python et R

8.2 Sources de construction

— Tables Hive

— Fichiers de données structurées

— Bases de données externes
— RDDs existants

8.3 Relation avec Dataset

Spark 2.0 - Fusion des APIs :

— En Scala : `DataFrame = Dataset[Row]`
— En Java : `DataFrame = Dataset<Row>`
— Unification des capacités de traitement

8.4 Avantages des DataFrames

8.4.1 1. Gestion mémoire personnalisée (Project Tungsten)

— Stockage en mémoire hors pile (off-heap)
— Hors de la JVM (Java Virtual Machine)
— Pas de surcharge de Garbage Collection
— Optimisation de l'utilisation mémoire

8.4.2 2. Plans d'exécution optimisés (Catalyst Optimizer)

— DataFrames converties en RDDs optimisés
— Requêtes optimisées automatiquement
— Amélioration considérable des performances

8.4.3 3. Équivalence relationnelle

— Équivalent de tables relationnelles
— Bonnes techniques d'optimisation
— Manipulation SQL-like

8.5 Limitations des DataFrames

1. **Pas de sécurité de type au moment de la compilation**
   — Erreurs détectées uniquement à l'exécution
   — Impossible de manipuler si structure inconnue
2. **Exemple d'erreur runtime :**

```
case class Student(id: Int, name: String)
val df = sqlContext.read.json("student.json")
df.filter("age␣>␣24").show
// Exception: cannot resolve 'age' given input id, name
```

9 Datasets

9.1 Définition

— Extension de DataFrames
— Collection **type-safe** et fortement typée

— Collection immuable d'objets
— Mappés à un schéma relationnel
— API disponible uniquement en Scala et Java

9.2 Architecture Spark 2.0

Dataset adopte deux caractéristiques d'API :

9.2.1 1. API fortement typée (Dataset)

— Collection d'objets JVM fortement typés
— Dictée par une case class (Scala/Java)
— Type connu à la compilation
— **Compile-time type safety**

```
case class T(id: Int, name: String)
val dataset: Dataset[T] = ...
```

9.2.2 2. API non typée (DataFrame)

— Alias pour `Dataset[Row]`
— Row = objet JVM générique non typé
— Structure inconnue à la compilation
— **No compile-time type safety**

9.3 Encodeurs

Rôle de l'encodeur :

— Conversion entre objets JVM et représentation tabulaire
— Stockage en format binaire Tungsten
— Opérations sur données sérialisées (ex : filter)
— Amélioration de l'utilisation mémoire
— Pas besoin de désérialiser pour certaines opérations

9.4 Avantages des Datasets

1. **Sécurité de type à la compilation**
   — Détection des erreurs avant l'exécution
   — Typage fort des objets
2. **Performance optimisée**
   — Format binaire Tungsten
   — Opérations sur données sérialisées
   — Meilleure utilisation mémoire
3. **Optimisation Catalyst**
   — Bénéficie de l'optimiseur Catalyst
   — Plans d'exécution optimisés

10 Comparaison RDD vs DataFrame vs Dataset

10.1 Disponibilité par langage

Langage	RDD	DataFrame	Dataset
Scala			
Java			
Python			×
R			×

10.2 Caractéristiques comparées

Critère	RDD	DataFrame	Dataset
Type safety	Non	Runtime	Compile-time
Optimisation	Aucune	Catalyst + Tungsten	Catalyst + Tungsten
Données	Non structurées	Structurées	Structurées + typées
Performance	Baseline	Élevée	Élevée
Schéma	Pas de schéma	Schéma requis	Schéma + types
Immutabilité	Oui	Oui	Oui
Lazy eval	Oui	Oui	Oui

10.3 Quand utiliser quoi ?

- **RDD** :
  - Contrôle bas-niveau nécessaire
  - Données non structurées
  - Optimisation manuelle souhaitée
- **DataFrame** :
  - Données structurées/semi-structurées
  - Requêtes SQL-like
  - Utilisation en Python/R
  - Performance optimale automatique
- **Dataset** :
  - Besoin de type safety
  - Utilisation en Scala/Java
  - Données structurées avec schéma connu
  - Avantages de l'optimisation + sécurité

11 Recommandations pratiques

11.1 Choix de l'API

- **Préférer DataFrame/Dataset aux RDDs** en raison des optimisations
- En Scala/Java : Utiliser Dataset[T] pour type safety
- En Python : Utiliser DataFrame uniquement (pas de Dataset)
- Conversion facile entre les différentes structures
- La plupart des exemples RDD s'adaptent facilement

11.2 Passage entre APIs

```
// RDD vers DataFrame
val rdd = sc.parallelize(data)
val df = rdd.toDF()

// DataFrame vers Dataset
case class Person(name: String, age: Int)
val ds = df.as[Person]

// Dataset vers RDD
val rdd = ds.rdd
```

12 Points clés à retenir

1. **RDD** : Structure fondamentale, bas-niveau, contrôle total
2. **DataFrame** : Optimisé, données structurées, disponible dans tous langages
3. **Dataset** : Type-safe, optimisé, Scala/Java uniquement
4. **Spark SQL** : Catalyst Optimizer + Tungsten = performances élevées
5. **Évaluation paresseuse** : Optimisation du plan d'exécution
6. **Transformations** : Narrow (pas de shuffle) vs Wide (avec shuffle)
7. **Actions** : Déclenchent l'exécution du DAG
8. **Persistance** : Évite les recalculs coûteux
9. **Evolution** : Spark 2.0+ unifie DataFrame et Dataset
10. **Best practice** : Préférer APIs haut-niveau pour performances