

SparkSQL

Structured APIs

DataFrames

Résumé des RDDs

- No mutable
- Pour des données non structurées
- Utilisation à l'aide d'une approche prog. fonctionnelle
- Stockage object
 - 1 zone mémoire pour chaque élément
- Pas de compression automatique
 - Sérialisation Java ou Kryo à la demande

Autres abstractions Spark

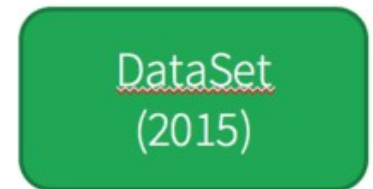
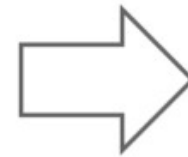
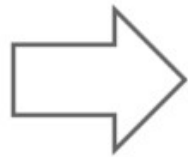
- DataFrame (\rightarrow 1.3)
 - No mutable
 - Collection distribuée avec nommage de colonnes
 - Chaque enregistrement est du type Row \rightarrow Dataset[Row]
 - Utilisation du modèle relationnel: DSL ou SQL
 - Optimisations
 - Stockage de tuples (plusieurs éléments dans un même bloc mémoire)
 - Compression automatique avec stockage en colonne
 - Chaque DataFrame représente un plan logique.

Autres abstractions Spark

- DataSet (\rightarrow 1.6)
 - Une extension de DataFrame API
 - Prends le meilleur des 2 mondes RDD et DataFrames
 - Compressé
 - Fortement typé (en Scala et Java)
 - Optimisé
 - Approches relationnelle et fonctionnelles

Abstractions Spark

History of Spark APIs



Distribute collection
of JVM objects

Functional Operators (map,
filter, etc.)

Distribute collection
of Row objects

Expression-based operations
and UDFs

Logical plans and optimizer


Fast/efficient internal
representations

Internally rows, externally
JVM objects

Almost the “Best of both
worlds”: type safe + fast

But slower than DF
Not as good for interactive
analysis, especially Python

Abstractions Spark - Résumé

	RDD	DataFrame	Dataset
Immutability	✓	✓	✓
Schéma	✗	✓	✓
Apache Spark 1	✓	✓	✓ (since 1.6 as experimental, but not in all the languages)
Apache Spark 2	✓	✓ (it does not exist in Java anymore)	✓ (not in the untyped languages such as Python)
Performance optimization	✗	✓	✓ 
Level	Low	High (built upon RDD)	High (DataFrame extension)
Typed	✓	✗	✓
Syntax Error	Compile time	Compile time	Compile time
Analysis Error	Compile time	Runtime	Compile time

DataFrame

- Influencé par le concept de data frame de R
- Mais évalue les opérations paresseusement pour effectuer certaines optimisations
- Collections d'enregistrements structurés pouvant être manipulés avec l'API procédurale de Spark ou les API relationnelles pour des optimisations plus riches
- stocke les données dans un format en colonnes nettement plus compact que les objets Java/Python

DataFrame

- Chaque DataFrame peut être considéré comme un RDD d'objets Row.
 - Un tableau de lignes
- Les opérations relationnelles peuvent être effectuées à l'aide d'un DSL similaire aux trames de données R et aux Python Pandas :
 - select, where, join, groupBy.

SparkSession

- Comme SparkContext est le point d'entrée pour les RDD, SparkSession est l'entrée pour l'API structurée
- Créé automatiquement dans le spark shell
- Doit être créé dans l'IDE
- Créé automatiquement dans le bloc-notes Databricks Community Edition

DataFrame - création

1	Action
2	Adventure
3	Animation
4	Children's
5	Comedy
6	Crime
7	Documentary
8	Drama
9	Fantasy
10	Film-Noir
11	Horror
12	Musical
13	Mystery
14	Romance
15	Sci-Fi
16	Thriller
17	War
18	Western

```
val genres = sc.textFile("genres.txt")  
  .map(_.split(" "))  
  .map(x=>(x(0), x(1)))  
  .toDF("id", "genre")
```

```
genres: org.apache.spark.sql.DataFrame =  
[id: string, genre: string]
```

DataFrame - création

id	name
1	Action
2	Adventure
3	Animation
4	Children's
5	Comedy
6	Crime
7	Documentary
8	Drama
9	Fantasy
10	Film-Noir
11	Horror
12	Musical
13	Mystery
14	Romance
15	Sci-Fi
16	Thriller
17	War
18	Western

```
val genres =  
  spark.read.format("csv").option("header",  
    "true").option("inferSchema", "true").load  
    ("genres.csv")  
  
genres: org.apache.spark.sql.DataFrame =  
[id: int, name: string]
```

Sources de données

- Un ensemble de méthodes pour la lecture et l'écriture de données est disponible
- `DataFrameReader` est la principale Classe supportant la lecture de données.
- Un motif standard est `SparkSession.read` qui retourne un objet `DataFrameReader` depuis lequel on peut appeler diverses méthodes:

```
DataFrameReader  
.format(args)  
.option("key", "value")  
.schema(args)  
.load(file)
```

Sources de données

- Format (par défaut: parquet)
 - csv, txt, json, jdbc, parquet, orc, avro, ..
- Option
 - Pour csv et JSON:
 - “header”, {“true”/”false”}
 - “inferSchema”/{“true”/”false”}
 - “mode”/{PERMISSIVE”, ”FAILFAST”, ”DROPMALFORMED”}
- Schema
 - ‘id INT, name STRING’
 - StructType(StructField ..)

Sources de données

- DataFrameWriter pour l'écriture de données
- L'option Format est similaire à la lecture
- Option
 - “mode”,
 {“append”, “overwrite”, “ignore”, “error”}
 - “bucketBy()”, (numBuckets, col, col,..)

Schéma

- Il est plus efficace de fournir un schéma que de laisser Spark le déduire.
- En Scala:

```
val schema =  
  StructType(Array(StructField("num", IntegerType, false),  
    StructField("label", StringType, false)))  
  
schema: org.apache.spark.sql.types.StructType =  
StructType(StructField(num,IntegerType,false),StructField(label,StringType,false))  
  
val g =  
  spark.read.option("header", "true").schema(schema).csv("genres.csv")  
g.printSchema  
root  
  |-- num: integer (nullable = true)  
  |-- label: string (nullable = true)
```

Création de schéma (DDL)

```
Scala> val schema2 = "ide INT, lab STRING"
schema2: String = ide INT, lab STRING
```

```
scala> val g2 =
spark.read.option("header", "true").schema(schema2).csv("genres.csv")
g2: org.apache.spark.sql.DataFrame = [ide: int, lab: string]
```

```
scala> g2.printSchema
root
|-- ide: integer (nullable = true)
|-- lab: string (nullable = true)
```


Affichage de résultats

id,name

1,Action

2,Adventure

3,Animation

4,Children's

5,Comedy

6,Crime

7,Documentary

8,Drama

9,Fantasy

10,Film-Noir

11,Horror

12,Musical

13,Mystery

14,Romance

15,Sci-Fi

16,Thriller

17,War

18,Western

`genres.show()`

Ou `genres.take(10)`

```
+---+-----+
| id|  genre|
+---+-----+
| 1|  Action|
| 2| Adventure|
| 3| Animation|
| 4| Children's|
| 5|   Comedy|
| 6|   Crime|
| 7| Documentary|
| 8|   Drama|
| 9|   Fantasy|
|10| Film-Noir|
|11|   Horror|
|12|   Musical|
|13|   Mystery|
|14|   Romance|
|15|   Sci-Fi|
|16|  Thriller|
|17|     War|
|18|  Western|
+---+-----+
```

Affichage de résultats

id,name

1,Action

2,Adventure

3,Animation

4,Children's

5,Comedy

6,Crime

7,Documentary

8,Drama

9,Fantasy

10,Film-Noir

11,Horror

12,Musical

13,Mystery

14,Romance

15,Sci-Fi

16,Thriller

17,War

18,Western

genres.collect

```
Array[org.apache.spark.sql.Row] =
```

```
Array([1,Action], [2,Adventure],
```

```
[3,Animation], [4,Children's],
```

```
[5,Comedy], [6,Crime], [7,Documentary],
```

```
[8,Drama], [9,Fantasy], [10,Film-Noir],
```

```
[11,Horror], [12,Musical], [13,Mystery],
```

```
[14,Romance], [15,Sci-Fi], [16,Thriller],
```

```
[17,War], [18,Western])
```

Ecriture/lecture

```
genres.write.format("json").save("genres.json")
```

- Cela enregistre un répertoire nommé genres.json. Enregistré dans le style HDFS
- Lecture sans aucune transformation préalable

```
val genresjson =  
spark.read.format("json").option("inferSchema", "true").load("genres.json")
```

```
genres.show(5)
```

```
+---+-----+  
| id|      name|  
+---+-----+  
|  1|    Action|  
|  2| Adventure|  
|  3| Animation|  
|  4|Children's|  
|  5|    Comedy|  
+---+-----+
```

DataFrame Opération

DataFrame supporte:

- Tous les opérateurs relationnels courants : select, where, join, groupBy, etc.
- Opérateurs de comparaison et arithmétiques :
- Tous les opérateurs mentionnés ci-dessus créent un arbre de syntaxe abstraite (AST) de l'expression, qui est transmis à Catalyst (l'optimiseur de requêtes de Spark).
- Les DataFrame enregistrés dans le catalogue correspondent à des vues non matérialisées, donc des optimisations sont possibles

DataFrame vs Relational Query Languages

Une API analyse l'exactitude d'un plan logique :

- Nom de colonnes utilisées dans l'expression
- Types de données



Spark signale les erreurs dès que les utilisateurs tapent du code invalide au lieu d'attendre l'exécution (compile-time vs run-time)

Manipulation DSL

```
genres.select("name").show(10)
```

```
+-----+  
|      name      |  
+-----+  
|      Action    |  
|  Adventure     |  
|  Animation     |  
| Children's     |  
|      Comedy    |  
|      Crime     |  
| Documentary    |  
|      Drama     |  
|      Fantasy   |  
|  Film-Noir    |  
+-----+
```

only showing top 10 rows

Manipulation DSL

```
genres.where("id=15").select("name").show
```

```
+-----+  
|  name  |  
+-----+  
| Sci-Fi |  
+-----+
```

```
genres.where($"name".like("A%")).select("name").show
```

```
+-----+  
|  name  |  
+-----+  
|  Action  |  
| Adventure |  
| Animation |  
+-----+
```

Manipulation DSL

```
df.groupBy("col1").agg(avg("age"))
```

```
df.select(expr("col1")).show(2) is similar to  
df.select(col("col1")).show(2) which is similar to  
df.select("col1").show(2)
```

```
df.sort(col("id").desc) similar to  
df.sort($"id".desc)
```

A frequent pattern is:

```
df. ...  
  .groupBy("col1")  
  .count()  
  .orderBy(desc("count"))  
  .show(10)
```


Manipulation SQL

```
genres.createOrReplaceTempView("Genre")
```

```
val res = spark.sql("SELECT * FROM Genre")
```

```
res: org.apache.spark.sql.DataFrame = [id: int, name: string]
```

```
scala> res.count
```

```
res28: Long = 18
```

```
scala> res.show(8)
```

```
+---+-----+
| id|      name|
+---+-----+
|  1|    Action|
|  2|  Adventure|
|  3| Animation|
|  4|Children's|
|  5|    Comedy|
|  6|    Crime|
|  7|Documentary|
|  8|    Drama|
+---+-----+
```

Supports ANSI SQL:2003