Interrogation de données structurées en Spark

Master DAC – Bases de Données Large Echelle

Mohamed-Amine Baazizi
mohamed-amine.baazizi@lip6.fr

2021-2022

High-level data pipelines

- The RDD algebra is not well-suited
  - Unfriendly programming, unreadable code
  - Misses the opportunity of logical optimization
- Most data pipelines contain
  - Analytical queries
  - Well-defined transformations (ex. aggregation, *feature extraction*)
- Most data is structured and have an implicit schema
- Adopt a declarative approach
  - SQL queries
  - Dataframe expressions: inspired from Data Science libraries
- The RDD algebra serves as a physical layer

Spark Dataframe

- Abstraction on top of RDDs
  - RDD[Row], Row = sequence of values with a fixed schema
- Several operators
  - Relational operator
  - User Defined and built-in functions
    - Maps each element to another element
    - E.g. in IMBD, extract year from title based on a regexp
- Distinction between transformations and actions
  - Some useful actions
    - printSchema(), show(), read(), write()
- Support for different formats
  - Tabular (CSV, TSV), nested (JSON, Parquet), Text
  - Specific formats (libsvm for ML)

Data model

- Base types
  - boolean, numeric family (integer, decimal, …), String, null, timestamp
- Complex types
  - Arrays : (type of element, containsNull) → *homogenous*
  - Structure :
    - [StructField, …, StructField]
      - StructField(name, type, nullable) → name is a <u>string</u> and is <u>unique</u>
  - Maps:
    - (keyType, valueType, valueContainsNull) → key of <u>any type</u> and is <u>unique</u>
- Support for user-defined types (UDT)
  - Application-specific data
  - Object-oriented databases style
    - Data and methods Dataset
    - Dataset with Scala

containsNull, nullable and valueContainsNull indicate the presence of null, but are always set to true

Tabular data

`movies = spark.read.format("csv"). …`

`movies.show(truncate=False)`

```
+-------+-----------------------------------+-------------------------------------------+
|movieId|title                              |genres                                     |
+-------+-----------------------------------+-------------------------------------------+
|1      |Toy Story (1995)                   |Adventure|Animation|Children|Comedy|Fantasy|
|2      |Jumanji (1995)                     |Adventure|Children|Fantasy                 |
|3      |Grumpier Old Men (1995)            |Comedy|Romance                             |
|4      |Waiting to Exhale (1995)           |Comedy|Drama|Romance                       |
|5      |Father of the Bride Part II (1995)|Comedy                                      |
|6      |Heat (1995)                        |Action|Crime|Thriller                      |
|7      |Sabrina (1995)                     |Comedy|Romance                             |
+-------+-----------------------------------+-------------------------------------------+
```

`movies.printSchema()`

```
root
 |-- movieId: long (nullable = true)
 |-- title: string (nullable = true)
 |-- genres: string (nullable = true)
```

Useful Dataframe operators

- Relational algebra
  - Unary : select, where
  - Binary : join, intersect, subtract, union
- Schema
  - drop: removes a column
  - withColumn: adds a column
  - withColumnRenamed: renames a column
- Grouping
  - groupBy(col*)
    - groups on a set of columns
    - Produces a *GroupedDataset*
- Sorting
- Aggregation

GroupedDataset operators

- Partition-wise single aggregation
  - min(col), max(col), sum(col), count()
  - return type = Dataframe with <u>one</u> column
- Partition-wise multiple aggregations
  - agg(min(col), max(col), …)
  - return type = Dataframe with <u>many</u> columns
- Pivoting
  - Pivot(col) -> Dataframe(rcol_1, …, rcol_n)
  - For i in [1, #values(col)] : name(rcol_i) = value(col_i)

Built-in functions

- Array/maps manipulation
  - Arrays: containment, distinct, intersect, except, max/min, search, size, sort,..
  - Maps: concatenation, filtering, size,…
- General-purpose
  - Math: descriptive stats, trigonometry, skewness
  - Summary: *countDistinct, sumDistinct, …*
  - Elements to Collections and vice-versa
    - Nesting: *collect_list*, *collect_set*
    - Flattening: *explode, explode_outer*
  - Temporal:
    - current date, date arithmetic, day/month/year extraction, conversion
  - Data transformation
    - Json/csv parsing
    - Timestamp extraction/encoding
  - Strings: length, distance, trim, regexp extraction , splitting, case conversion, …
- Fully-documented online

Illustration:
Building and flattening lists

```
+-------+-----------------------------------+---------------------------------------------+
|movieId|title                              |genres                                       |
+-------+-----------------------------------+---------------------------------------------+
|1      |Toy Story (1995)                   |Adventure|Animation|Children|Comedy|Fantasy|
|2      |Jumanji (1995)                     |Adventure|Children|Fantasy                   |
|3      |Grumpier Old Men (1995)            |Comedy|Romance                               |
|4      |Waiting to Exhale (1995)           |Comedy|Drama|Romance                         |
|5      |Father of the Bride Part II (1995)|Comedy                                        |
|6      |Heat (1995)                        |Action|Crime|Thriller                        |
|7      |Sabrina (1995)                     |Comedy|Romance                               |
+-------+-----------------------------------+---------------------------------------------+
```

```
root
 |-- movieId: long
 |-- title: string
 |-- genres: string
```

```
from pyspark.sql.functions import col, split, explode
genres_list = movies.select(split(movies.genres,'\|').alias('genre_list'))
genres = genres_list.select(explode(col('genre_list')).alias('genre')).distinct()
genres.count()
10
```

Size → 7

```
+---------+
|    genre|
+---------+
|Adventure|
|   Comedy|
|  Romance|
| Children|
```

Illustration:
Pivoting rows to columns

movieId	title	genres
1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy
2	Jumanji (1995)	Adventure|Children|Fantasy
3	Grumpier Old Men (1995)	Comedy|Romance
4	Waiting to Exhale (1995)	Comedy|Drama|Romance
5	Father of the Bride Part II (1995)	Comedy
6	Heat (1995)	Action|Crime|Thriller
7	Sabrina (1995)	Comedy|Romance

```
root
 |-- movieId: long
 |-- title: string
 |-- genres: string
```

movies_genre = movies.select('movieId',explode(split(movies.genres,'\|')).alias('genre'))\
.**groupBy**('movieId').**pivot**('genre').count().orderBy('movieId')

movieId	Action	Adventure	Animation	Children	Comedy	Crime	Drama	Fantasy	Romance	Thriller
1	null	1	1	1	1	null	null	1	null	null
2	null	1	null	1	null	null	null	1	null	null
3	null	null	null	null	1	null	null	null	1	null

...

Illustration: applying built-in functions

```
+------+-------+------+----------+
|userId|movieId|rating| timestamp|
+------+-------+------+----------+
|     1|      1|   2.5|1260759144|
|     1|      2|   3.0|1260759179|
|     2|      1|   3.0|1260759182|
|     4|      3|   2.0|1260759185|
```

```
userId: long (nullable = true)
movieId: long (nullable = true)
rating: double (nullable = true)
timestamp: long (nullable = true)
```

```
from pyspark.sql.functions import from_unixtime, year, month, dayofmonth
```

```
ratings_date = ratings.select(from_unixtime('timestamp').alias('datetime'))\
.select('datetime', dayofweek('datetime').alias('day'))
```

```
+-------------------+---+
|           datetime|day|
+-------------------+---+
|2009-12-14 02:52:24|  2|
|2009-12-14 02:52:59|  2|
|2009-12-14 02:53:02|  2|
|2009-12-14 02:53:05|  2|
```

User-defined Functions

- Express non-relational operations
  - Ex. map rating to categories, compute user similarity
- Supported in most database systems
- Executed out of the SQL context
  - No automatic optimisation but
  - Possibility of adding a new optimisation rules (for experts)
  - If not carefully designed, performance degradation
- Only use when no available built-in function can do the work
- Two possibilities in Spark
  - Standard: invoke on each row → costly
  - Vectorized: invoke on a vector (batch of rows) → more efficient

  Benefit from memory columnar storage (Arrow format)

Row-at-time vs vectorized UDFs

col1	col2	...
v00	v01	
v10	v11	
v20	v21	

std(v00)
std(v10)
std(v20)

@udf()
def **std**(): ...
select(**std**(col1))

col1	col2	...
v00	v01	
v10	v11	
v20	v21	

Panda([v00,v10,v20])

@pandas_udf ()
def **panda**(): ...
select(**panda**(col1))

panda() must preserve the input size
and be independent of the data partitioning

Illustration: applying a standard UDF

movieId	title	genres
1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy
2	Jumanji (1995)	Adventure|Children|Fantasy
3	Grumpier Old Men (1995)	Comedy|Romance
4	Waiting to Exhale (1995)	Comedy|Drama|Romance
5	Father of the Bride Part II (1995)	Comedy
6	Heat (1995)	Action|Crime|Thriller
7	Sabrina (1995)	Comedy|Romance

```
root
 |-- movieId: long
 |-- title: string
 |-- genres: string
```

```
from pyspark.sql.functions import udf

@udf('string')
def nb_genres(l):
return str(len(l))+' genres(s)’

select('movieId', 'title', nb_genres('genre_list').alias('nb_genres'))
```

movieId	title	nb_genres
1	Toy Story (1995)	5 genres(s)
2	Jumanji (1995)	3 genres(s)
3	Grumpier Old Men (1995)	2 genres(s)

Illustration: applying a vectorized UDF

```
+-------+----------------------------------+-------------------------------------------+
|movieId|title                             |genres                                     |
+-------+----------------------------------+-------------------------------------------+
|1      |Toy Story (1995)                  |Adventure|Animation|Children|Comedy|Fantasy|
|2      |Jumanji (1995)                    |Adventure|Children|Fantasy                 |
|3      |Grumpier Old Men (1995)           |Comedy|Romance                             |
|4      |Waiting to Exhale (1995)          |Comedy|Drama|Romance                       |
|5      |Father of the Bride Part II (1995)|Comedy                                     |
|6      |Heat (1995)                       |Action|Crime|Thriller                      |
|7      |Sabrina (1995)                    |Comedy|Romance                             |
+-------+----------------------------------+-------------------------------------------+
```

```
root
 |-- movieId: long
 |-- title: string
 |-- genres: string
```

```
import pandas as pd
from pyspark.sql.functions import col, pandas_udf
from pyspark.sql.types import ArrayType

def nb_genres(s:pd.Series) -> pd.Series:
return s.str.split('\|')

panda_nb_genres = pandas_udf(nb_genres, returnType=ArrayType(StringType()))

movies.select('movieId', 'title', panda_nb_genres('genres').alias('nb_genres'))
```

```
root
 |-- movieId: long (nullable = true)
 |-- title: string (nullable = true)
 |-- nb_genres: array (nullable = true)
 |    |-- element: string (containsNull = true)
```

Data reading and writing

- Reading / Writing
  – Several sources (local files, HDFS), batch or streaming
  – Many database connectors: JDBC or system-specific (eg. Mongo, Couchbase)
  – Guided by a schema
  – Schema is either provided or inferred

- Reading/writing options
  – Fully documented (see API reference)
  – Format, encoding, separators, compression

- Side effect free by default
  – Modify a copy of the data

- Potential solution: Delta Lake
  – ACID support: insert, Update, delete operations
  – Data versioning and schema evolution
  – Metadata management

Schema management

- Option 1: passed as argument
  - Plain text
    - 'attr_name type, …, attr_nametype'
  - *Dataframe* object
    - StructType(), ArrayType(), LongType(), …
  - Basic schema enforcement
    - Type compatibility, Nullity
    - Failure raises an exception

data: RDD with [[1,1],[None,2], [3,None]]

```
f1 = StructField('f1', IntegerType(), True)
f2 = StructField('f2', IntegerType(), True)
df_sch = StructType([f1,f2])

df = spark.createDataFrame(data, df_sch)
```

Problem

```
root
 |-- f1: integer (nullable = true)
 |-- f2: integer (nullable = false)
```

Schema management

- Option 2: inferred from the data
  - Sample of the data, usually 10%
  - Useful when not shipped with the data, or when data is nested (JSON)
  - Type promotion rules
    - Triggered to solve conflicting types
    - Lossy transformation

data: RDD with [[1,1],['str',2], [3,'str']]

df = spark.createDataFrame(data)

```
root
 |-- _1: long (nullable = true)
 |-- _2: long (nullable = true)
```

```
+----+----+
|  _1|  _2|
+----+----+
|   1|   1|
|null|   2|
|   3|null|
+----+----+
```

Beyond Spark schema management

- Schema enforcement strategy
  - Only valid data or mix between non-valid and valid?
  - Discard tuples, substitute with default?
- Support integrity constraints (IC)
  - Primary key - foreign key
  - Domain constraints
  - Assertions
- Data quality issues
  - Missing values
- Potential solution: Delta Live Tables
  - Automatic support for schema evolution
  - IC specification and validation
  - Metadata collection at scale

  → Hassle-free ETL

Managing nested data

- Schema inference, data loading
  - Automatic, on a sample of the data
  - Data is shredded using the schema
    - Shredding rules
      - JSON Object → Dataframe Struct
      - JSON Arrays → Dataframe Array
      - JSON Basic types → corresponding Dataframe basic types
    - Missing fields indicated with null
    - Conflict resolution using type promotion rules

JSON and irregularity

Input: 2 documents

```
{
  "person" : {
    "firstname" : "Melena",
    "lastname" : "RYZIK",
    "role" : "reported",
    "rank" : 1,
    "organization" : ”abc"
  }
}
```

```
{
  "person" : {
    "firstname" : "other",
    "lastname" : "ABCD",
    "rank" : 1,
    "organization" : "OO"
  }
}
```

spark.read.json()

```
root
 |-- person: struct (nullable = true)
 |    |-- firstname: string (nullable = true)
 |    |-- lastname: string (nullable = true)
 |    |-- organization: string (nullable = true)
 |    |-- rank: long (nullable = true)
 |    |-- role: string (nullable = true)
```

schema

firstname	lastname	organization	rank	role
Melena	RYZIK	abc	1	reported
other	ABCD	OO	1	

Missing role

data

JSON and irregularity

Input: 3 documents

```
{
  "first" : "al",
  "coord" : [],
  "last" : "jr"
}
```

```
{
  "first" : "al",
  "coord" : null,
  "last" : "jr"
}
```

```
{
  "email" : "abc@ef",
  "first" : "li",
  "coord" : {
    "lat" : 45,
    "long" : 12
  },
  "last" : null
}
```

spark.read.json()

```
root
 |-- coord: string (nullable = true)
 |-- email: string (nullable = true)
 |-- first: string (nullable = true)
 |-- last: string (nullable = true)
```

coord	email	first	last
[ ]	null	al	jr
null	null	al	jr
{"long":12,"lat":45}	abc@ef	li	null

schema

Object stored as a string!!

data

Querying nested data

- Navigating the hierarchy of objects
  - Dot-notation
    - select (‘ level1.level2.)
    - Traversing arrays is not allowed using the dot-notation!!
- Accessing the content of arrays
  - Flatten then navigate
    - explode(), explode_outer()
  - Use existing (built-in) array functions
    - E.g. array_exist(), array_contains()
  - Performance issues:
    - Flattening allows for standard logical optimization to be applied
    - Many research effort on how to rewrite queries to take advantage of logical optimization

Navigating structures

```
{
  "person" : {
    "firstname" : "Melena",
    "lastname" : "RYZIK",
    "role" : "reported",
    "rank" : 1,
    "organization" : ""
  }
}
```

```
{
  "person" : {
    "firstname" : "other",
    "lastname" : "ABCD",
    "rank" : 1,
    "organization" : "OO"
  }
}
```

```
root
 |-- person: struct (nullable = true)
 |    |-- firstname: string (nullable = true)
 |    |-- lastname: string (nullable = true)
 |    |-- organization: string (nullable = true)
 |    |-- rank: long (nullable = true)
 |    |-- role: string (nullable = true)
```

select("person.*")

firstname	lastname	organization	rank	role
Melena	RYZIK		1	reported
other	ABCD	OO	1	null

Accessing Arrays

```
{
  "person" : [
    {
      "firstname" : "Melena",
      "lastname" : "RYZIK",
      "role" : "reported",
      "rank" : 1,
      "organization" : ""
    },
    {
      "firstname" : "derba",
      "lastname" : "OKYZ",
      "role" : "reported",
      "rank" : 1,
      "organization" : ""
    }
  ]
}
```

```
{
  "person" : [
    {
      "firstname" : "other",
      "lastname" : "ABCD",
      "rank" : 1,
      "organization" : "OO"
    }
  ]
}
```

select(col("person"),explode(col("person")))

```
root
 |-- person: struct (nullable = true)
 |    |-- firstname: string (nullable = true)
 |    |-- lastname: string (nullable = true)
 |    |-- organization: string (nullable = true)
 |    |-- rank: long (nullable = true)
 |    |-- role: string (nullable = true)
```

person	col
[[Melena, RYZIK, , 1, reported], [derba, OKYZ, , 1, reported]]	[Melena, RYZIK, , 1, reported]
[[Melena, RYZIK, , 1, reported], [derba, OKYZ, , 1, reported]]	[derba, OKYZ, , 1, reported]
[[other, ABCD, OO, 1,]]	[other, ABCD, OO, 1,]

Closing remarks

- Dataframes
  - a full-fledged API for building complex ETL and analytics
  - many room for physical optimization
    - Disk and memory columnar storage (following lectures), indexing
- Interesting topics
  - The Delta Lake approach
  - Code versioning
- 2 lab sessions
  - Querying Tabular data: gain familiarity with the Dataframe algebra, use functions
  - Querying Nested data: deal with irregularity and incompleteness, real-world data

- *Next 30’: Zennea presentation on how to manage metadata at an entreprise level*