KTH Royal Institute of Technology
Stockholm

School of Electrical Engineering and
Computer Science

Data-Intensive Computing - ID2221

Review Questions 3

Author
Emil Ståhl

Author
Selam Fsha

September 26, 2021

Review Questions 3 - ID2221

Emil Ståhl and Selemawit Fsha Nguse

September 26, 2021

1 Briefly compare the DataFrame and DataSet in SparkSQL and via one example show when it is beneficial to use DataSet instead of DataFrame.

Spark has two notions of structured collections, which are DataFrames and Datasets. They are distributed table-like collections with well-defined rows and columns and represent immutable lazily evaluated plans. When an action is performed on them, Spark performs the actual transformations and return the result.

- DataFrame
  - Has rows and columns equivalent to a RDBS
  - Organized into named columns
  - When used together with SQL, you get declarative transformations on partitioned collections of tuples
  - Created from RDD or raw data sources
  - Immutable distributed collection of data
  - DataFrames elements are Rows, which are generic untyped JVM objects.
  - Scala compiler cannot type check Spark SQL schemas in DataFrames
  - Analysis errors during Runtime
  - Imposes a structure onto a distributed collection of data
  - Allowing higher-level abstraction
  - Uses a catalyst optimizer for optimization.
  - Faster than both RDD and DataSet

- DataSet
  - Extension of DataFrame API
  - Provides a type-safe, object-oriented programming interface
  - DataSet allows for both Spark SQL optimizations and typesafety
  - DataSet is a typed distributed collection of data
  - Brings together RDD and DataFrame API
  - DataSet[Row] is equivalent to a DataFrame
  - DataFrame and RDD are subsets of a DataSet
  - Analysis errors during compile time
  - Transformations on Datasets are the same as those that we had on DataFrames
  - Datasets allow for more complex and strongly typed transformations
  - Uses Spark's Catalyst optimizer by exposing expressions and data fields to a query planner.

If you want higher degree of type-safety at compile time and typed JVM objects, take advantage of Catalyst optimization, and benefit from Tungsten's efficient code generation, use Dataset instead of DataFrame. DataSet also offers compile-time type-safety which makes Datasets most productive for developers. The compiler catches most errors. However, non-existing column names in DataFrames detect on runtime. A rich set of semantics and high-level functions make the Dataset API simple to use.[1]

[1]Spark SQL: Relational Data Processing in Spark Michael Armbrust, Databricks, MIT, UC Berkeley

2 What will be the result of running the following code on the table people.json, shown below? Explain how each value in the final table is calculated.

val people = spark.read.format(”json”).load(”people.json”)
val windowSpec = Window.rowsBetween(-1, 1)
val avgAge = avg(col(”age”)).over(windowSpec)
people.select(col(”name”), col(”age”),
avgAge.alias(”avg_age”)).show

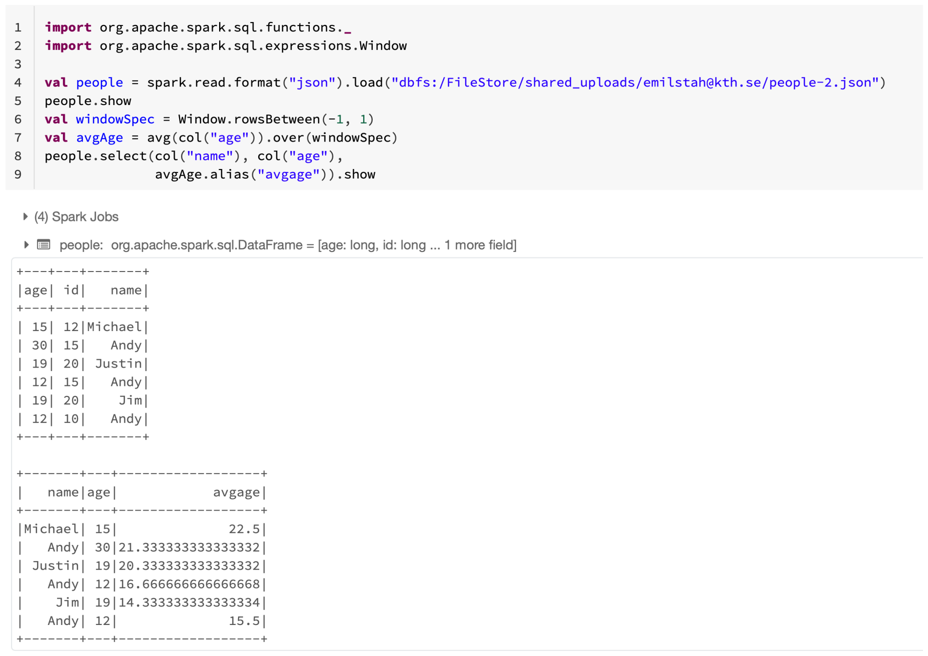
people.json
”name”:”Michael”, ”age”:15, ”id”:12
”name”:”Andy”, ”age”:30, ”id”:15
”name”:”Justin”, ”age”:19, ”id”:20
”name”:”Andy”, ”age”:12, ”id”:15
”name”:”Jim”, ”age”:19, ”id”:20
”name”:”Andy”, ”age”:12, ”id”:10

The people.json file is loaded into a DataFrame. We now have the following schema;

```
StructType(StructField(age,LongType,true),
StructField(id,LongType,true),
StructField(name,StringType,true))
```

Next, we create a window that determines which rows will be passed in to our functions. A window is defined by using a reference to the current data. A group of rows is called a frame. With rowsBetween(-1, 1) we select all the rows from our people Dataframe and store them in windowSpec. Now it's time to calculate the average age of the people in the table. By using the **avg()** we can automatically calculate the average value of some selected data. In this case, we use the **col()** functions that returns a reference to the column named **age** holding the age of the people in the table. We iterate over this column by calling the **.over()** and passing our frame of rows into it. That way, we will iterate on the complete column and all values of it. The average is saved to **aveAge**. Next we select the name and age columns by using Spark select() that is a transformation function that is used to select the columns from DataFrame by taking column type. Lastly, we use the alias keyword to Returns a new DataFrame with an alias set, by using .show() we can print the values.

The average age is calculated iteratively throughout the table starting with the first two values obtaining (30+15)/2 = 22.5. The complete table looks like this:



```scala
import org.apache.spark.sql.functions._
import org.apache.spark.sql.expressions.Window

val people = spark.read.format("json").load("dbfs:/FileStore/shared_uploads/emilstah@kth.se/people-2.json")
people.show
val windowSpec = Window.rowsBetween(-1, 1)
val avgAge = avg(col("age")).over(windowSpec)
people.select(col("name"), col("age"),
              avgAge.alias("avgage")).show
```

▸ (4) Spark Jobs

▸ people: org.apache.spark.sql.DataFrame = [age: long, id: long ... 1 more field]

```
+---+---+-------+
|age| id|   name|
+---+---+-------+
| 15| 12|Michael|
| 30| 15|   Andy|
| 19| 20| Justin|
| 12| 15|   Andy|
| 19| 20|    Jim|
| 12| 10|   Andy|
+---+---+-------+

+-------+---+------------------+
|   name|age|            avgage|
+-------+---+------------------+
|Michael| 15|              22.5|
|   Andy| 30|21.333333333333332|
| Justin| 19|20.333333333333332|
|   Andy| 12|16.666666666666668|
|    Jim| 19|14.333333333333334|
|   Andy| 12|              15.5|
+-------+---+------------------+
```

3 What is the main difference between the log-based broker systems (such as Kafka), and the other broker systems.

A message broker decouples the producer-consumer interaction. It runs as a server, with producers and asynchronous consumers connecting to it as clients. Producers write messages to the broker, and consumers receive them by reading them from the broker. In typical message brokers, once a message is consumed, it is deleted from the system. However, in log-based broker systems such as Kafka, all events are durably stored in a sequential log which is an append-only sequence of records on disk. A producer sends a message by appending it to the end of the log. A consumer receives messages by reading the log sequentially.

For scaling log-based systems, the logs can be partitioned on different machines. Each partition can be read and written independently of others. A topic is a group of partitions that all have messages of the same type. Within each partition, the broker assigns a monotonically increasing sequence number (offset) to every message. However, there are no ordering order guarantee across partitions.[2]

[2]Kafka: a Distributed Messaging System for Log Processing, Jay Kreps, LinkedIn

4 Compare the windowing by processing time and the windowing by event time, and explain how watermarks help streaming processing systems to deal with late events.

A Window is a buffer associated with an input port to retain previously received tuples. Four different windowing management policies are used:

- Count-based policy: the maximum number of tuples a window buffer can hold
- Delta-based policy: a delta threshold in a tuple attribute
- Punctuation-based policy: a punctuation is received
- Time-based policy: based on processing or event time period

For time-based policy, we have windowing by processing time and event time:

- **Windowing by Processing Time**
  - The system buffers up incoming data into windows until some amount of processing time has passed
  - Simple and requires no coordination between streams and machines
  - Events may arrive out-of-order and asynchronously
  - Incorporates all events that arrive at the operator during the time window, for example 2 hours between 09:00 - 11:00 AM.
  - High performance and low latency
- **Windowing by Event Time**
  - Specifies the time when each individual event is generated at the producing source
  - Store time in metadata signifying at what time a event is produced at its source
  - Stateful streaming
  - When applications need processing or computations based on the time that an event is generated
  - Can handle events out of order

In streaming processing, events can arrive late. To help with this systems make use of **watermarks**. Watermarks flow as part of the data stream and carry a timestamp *t*. A watermark is a threshold to specify how long the system waits for late events. Streaming systems uses watermarks to measure progress in event time. In general, a watermark is a declaration that a specific point in the stream, all events up to a certain timestamp should have arrived. Once a watermark reaches an operator, the operator can advance its internal event time clock to the value of the watermark. To give an example, in a case where we have an hourly-window operation, watermarks will allow us to understand when the specific hourly window goes beyond the hour so that the operator can close the existing window in operation. A Watermark (t) declares that event time has reached time t in that stream, meaning that there should be no more elements from the stream with a timestamp t' ¡= t. If an arriving event lies within the watermark, it gets used to update a query. This is the equivalent of Clock in the Lamport algorithm.[3]

5 Compare different "delivery guarantees" in stream processing systems.

Kafka guarantees that messages from a single partition are delivered to a consumer in order. There is no guarantee on the ordering of messages coming from different partitions. Kafka only guarantees at-least-once delivery which means a message may might appear many times. Besides at-least-once delivery, there is exactly-once and at-most-once delivery. Below is a comparison of the different guarantees:

at-most-once delivery

Means that for each message handed to the mechanism, that message is delivered zero or one times; in more casual terms it means that messages may be lost. At-most-once delivery is the cheapest—highest performance, least implementation overhead—because it can be done in a fire-and-forget fashion without keeping state at the sending end or in the transport mechanism.

at-least-once delivery

Means that for each message handed to the mechanism potentially multiple attempts are made at delivering it, such that at least one succeeds; again, in more casual terms this means that messages may be duplicated but not lost. This guarantee requires retries to counter transport losses, which means keeping state at the sending end and having an acknowledgement mechanism at the receiving end.

exactly-once delivery

The holy grail of messaging. All messages will be delivered exactly one time since for each message handed to the mechanism exactly one delivery is made to the recipient, the message can neither be lost nor duplicated. Exactly-once

[3]Kafka: a Distributed Messaging System for Log Processing, Jay Kreps, LinkedIn

guarantee is most expensive and has the worst performance—because in addition to the second it requires state to be kept at the receiving end in order to filter out duplicate deliveries. It requires less handling on the client side bu increases the complexity and cost for the producer or provider. Exact-once is not fault tolerant in large distributed systems, because it is impossible for all systems to agree on each message if some of systems may fail. Exact-once is not considered better because it comes with high cost, whereas at-least-once is good enough in most circumstances.[4]

Both RabbitMQ and Kafka offer durable messaging guarantees. Both offer at-most-once and at-least-once guarantees but also Kafka offers exactly-once guarantees in a very limited scenario.[5]

[4]Kafka: a Distributed Messaging System for Log Processing, Jay Kreps, LinkedIn
[5]2020 - A Survey on the Evolution of Stream Processing Systems