Data Intensive Computing – Review Questions 3

Adriano Mundo, Riccardo Colella (Group 29)

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

   DataFrames and *DataSets* are distributed table-like collections with well-defined rows and columns. Within these tables columns must have the same number of rows of the other columns, even though there may be a null value, and each column should have type information consistent for every row in the collection. They represent immutable, lazy-plans that specific which operations apply to data in order to generate some output.

   - *DataFrames:* they try to replace RDDs with functional transformation on partitioned collection of objects while maintaining a table structure which allows declarative transformation on tuples. They are "untyped", but to say *DataFrames* are untyped is inaccurate to a certain extent since they have types, but Spark maintains them and checks whether they line up to the schema at runtime. In fact, rows are generic JVM objects and the Scala compiler cannot check SparkSQL schemas in *DataFrames*.

   - *DataSets:* they are typed collections of data and their types are checked at compile time. Since *DataSets* are type-safe there's no possibility to accidentally view objects in a *DataSets* which are part of another class. This makes them attractive for writing large applications where multiple users have to interact through well-defined interfaces. *DataSet* API unifies *DataFrame* and RDD; to Spark *DataFrames* are simply *DataSets* of Type "Row".

   To show the advantage of using *DataSets* instead of *DataFrames* we can consider the case of collecting multiple data in the master after a parallelize method. For using *DataFrames* we should cast the Rows, while using *DataSets* the schema is preserved and there's an automatic type-checking.

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 result of the execution is the following:

```
--------------------------------
| "name"   | "age" | "avg_age" |
|------------------------------|
| Michael  |  15   |    22.5   |
|------------------------------|
| Andy     |  30   |21.33333333|
|------------------------------|
| Justin   |  19   |20.33333333|
|------------------------------|
| Andy     |  12   |16.66666667|
|------------------------------|
| Jim      |  19   |14.33333333|
|------------------------------|
| Andy     |  12   |    15.5   |
--------------------------------
```

The code loads the json file in the `var people` and then it applies the windowing grouping with the command `rowsBetween(-1,1)`. This means having a reference to the row before and the row after, highlighted by the boundaries -1,1. Then, it computes the average over the window; hence the variable `avg_age` is calculated by doing the average of the values for every tuple. The tuple is selected using the row index and comprehends the rows on which the calculation is done, the one before and one after. For example, in the case of Justin the `avg_age` is calculated by doing the arithmetic average of Andy, Justin and Andy ages. Finally, the table is shown to the console through the command `show`.

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

The main difference between log-based broker systems and the others is that in typical message brokers once a message is consumed it is deleted while in log-based message brokers all the events are durably stored in a sequential log. Thus, the most important difference is about durability. What happens is that log, which is an append-only sequence of records like a list, is where producers send their messages. They are appended to the log while consumers receive them by reading sequentially the log. The log helps at decoupling producer and consumer and can partitioned on different machines for scalability reason.
An example of log-based message broker is Kafka which is a distributed, topic oriented, partitioned and replicated commit log service.

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.

Windowing is the notion of taking a data source which is unbounded or bounded and define temporal boundaries for creating finite buffer for processing. There are different windowing management policies. Windowing is essentially in both processing and event time domain.

- *Windowing by Processing Time:* the system essentially buffers up incoming data into windows until some amount of processing time has passed. For example, for five-minute fixed windows the system would buffer data up for five minutes. It has some properties: it is simple to implement since you just buffer things as they arrive; judging window completeness is straightforward since the system has perfect knowledge of all inputs; it's perfect for inferring information about the source which is observed.

- *Windowing by Event Time*: it can be used for observing a data source which is divided in finite chunks that reflect the times at which those events actually happened. Event time correctness and the possibility to create dynamically sized windows (sessions) are two nice things to mention. On the contrary, buffering and lateness are two notable drawbacks.

To deal with lateness it is possible to use *watermarking*, which is a notion of input completeness with respect to event times. A watermark is an amount of time following a given event or set of events after which we do not expect to see any more data from that time.
Watermarks flows as part of the data stream and carry a timestamp t. It defines a threshold to specify how long the system should wait for late evets (they are used to measure progress in event time). The method keeps track of event that enter the stream at event t and guarantees that there are no elements in the stream with a timestamp t'<= t. Some elements will violate the condition and if an arriving event lies within the watermark it is used to update a query.

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

In stream processing systems there are two kinds of delivery guarantees:

- *At-least-once:* this delivery guarantees that a message will be always delivered. It might appear many times but there will never be a message lost. This the case of Kafka, in fact most of the times a message is delivered exactly once to each consumer group. Also, Kafka guarantees that messages from a single partition are delivered to a consumer in order; however, there's no guarantee on the ordering of messages coming from different partitions.

- *Exactly-once:* this delivery guarantees that all messages will always be delivered exactly once; thus, it is consumed just once. This does not many there will be not failures or retries but the retries will succeed. This needs a two-phase commit protocol.