ID2221 - Data Intensive Computing - Review Questions 3

Deepak Shankar   Reethika Ambatipudi
deepak | reethika @kth.se

September 22, 2023

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

DataFrames and DataSets are both high-level abstractions in Spark SQL that provide structured data processing capabilities. Here's a brief comparison between the two:

DataFrames are distributed collections of data organized into named columns.	DataSets are a superset of DataFrames, combining the benefits of DataFrames and the strong typing of RDDs (Resilient Distributed Datasets).
They are similar to tables in a relational database and provide a schema for the data and can be manipulated using SQL queries and DataFrame-specific operations.	They provide the benefits of static typing, compile-time type checking, and object-oriented programming.
DataFrames are represented as DataFrame objects in Spark.	DataSets are represented as Dataset[T], where T is the type of the data.
They offer optimizations in terms of query planning and execution.	They are suitable for use cases where you want to leverage the benefits of both DataFrames and RDDs while maintaining strong type safety.

To provide a clear example of when it's beneficial to use a DataSet instead of a DataFrame, let's consider a scenario where we work with semi-structured data with varying schemas for different records. This is a situation where the strong typing and flexibility of DataSets can be advantageous.

Suppose we have a dataset that contains information about various types of events. Each event record can have different attributes based on its type. We want to filter and process events of a specific type while ensuring type safety. Here's how we can do it using a DataSet:

```
import org.apache.spark.sql.{Dataset, SparkSession}

// Define a case class to represent event data with strong typing
case class Event(timestamp: String, eventType: String, details: Map[String, String])

object DataSetExample {
  def main(args: Array[String]): Unit = {
    val spark = SparkSession.builder()
      .appName("DataSetExample")
      .master("local[*]")
      .getOrCreate()

    import spark.implicits._

    // Create a DataSet of Event objects
    val eventsDS: Dataset[Event] = Seq(
      Event("2023-09-15T08:00:00", "Login", Map("user" -> "Alice")),
      Event("2023-09-15T09:30:00", "Purchase", Map("product" -> "Phone", "price" -> "500")),
      Event("2023-09-15T10:15:00", "Logout", Map("user" -> "Bob"))
    ).toDS()

    // Filter and process events of a specific type (e.g., "Purchase")
    val filteredEventsDS: Dataset[Event] = eventsDS.filter(_.eventType == "Purchase")

    // Show the filtered events
    filteredEventsDS.show()

    spark.stop()
  }
}
```

In this example, we define a case class Event to represent event data. Each event has a timestamp, an event type, and details represented as a Map[String, String]. We create a DataSet of Event objects and filter events of a specific type ("Purchase").

The strong typing of DataSets allows us to define the schema of the event data with confidence, even though the details of each event can vary. This ensures type safety while working with semi-structured or dynamically structured data.

Using DataFrames in this scenario would be less beneficial because DataFrames are designed for structured data with a fixed schema. DataSets provide the flexibility needed to work with varying attributes within records, making them a better choice for such semi-structured data scenarios.

Q-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}
```

This Result shows the "name" and "age" columns from the original data, along with the calculated "avgAge" values for each person within the specified window of rows.

name	age	avg_age
Michael	15	22.5
Andy	30	21.33
Justin	19—	20.33
Andy	12	26.67
Jim	19	14.43
Andy	12	15.5

Calculates the average age of each person in a window of rows that includes the current row and the two rows before and after it. The window specification is defined using Window.rowsBetween(-1, 1).

1. Row 1 ("Michael", 15, ...). Window includes ages 15 and 30. Average Age = 22.5
2. Row 2 ("Andy", 30, ...). Window includes ages 15, 30, and 19. Average Age = 21.33

3. Row 3 (”Justin”, 19, ...). Window includes ages 30, 19, and 12. Average Age = 20.33
4. Row 4 (”Andy”, 12, ...). Window includes ages 19, 12, and 19. Average Age = 16.67
5. Row 5 (”Jim”, 19, ...). Window includes ages 12, 19, and 12. Average Age = 14.43
6. Row 6 (”Andy”, 12, ...). Window includes ages 19, 12. Average Age = 15.5

The ”show” command displays these average age values, along with the corresponding names and ages, for each row in the ”people” DataFrame.

Q-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 like Apache Kafka and other broker systems lies in their fundamental architecture and data handling approach. Here’s the key distinction:

Log-Based Broker Systems (e.g., Kafka):

- **Log-Centric:** Log-based systems are primarily centered around the concept of immutable logs. In Kafka, data is stored in distributed, partitioned logs, where each log represents a sequence of records.
- **Ordered and Append-Only:** Logs are ordered, meaning data is appended to them in a strictly ordered fashion. Older records are never modified or deleted; they are retained for a specified period.
- **Distributed Publish-Subscribe:** Kafka follows a publish-subscribe model, where producers publish records to topics, and consumers subscribe to topics to receive records.
- **High Throughput and Low Latency:** Log-based systems are optimized for high throughput and low-latency data ingestion and processing. They excel in scenarios where massive volumes of data need to be processed in real-time or near-real-time.

Other Broker Systems (e.g., RabbitMQ, ActiveMQ):

- **Message-Centric:** Traditional broker systems are message-centric, where messages are units of data sent between producers and consumers. Messages can be consumed and acknowledged, and the broker manages their storage and routing.
- **Message Queues:** These systems often use message queues, which are temporary storage areas for messages. Messages are typically removed from the queue once they are consumed.

- **Acknowledgment and Guaranteed Delivery:** In message-centric systems, there is usually acknowledgment of message receipt, and the broker ensures guaranteed delivery by retaining messages until they are successfully consumed.
- **More Features and Complexity:** Traditional brokers offer features like message routing, filtering, and complex routing patterns, making them suitable for various messaging patterns, including request-response, point-to-point, and publish-subscribe.

In summary, the main difference is that log-based broker systems like Kafka are optimized for handling large volumes of immutable data logs with a focus on high throughput and low-latency data processing, making them well-suited for use cases like real-time event streaming and log aggregation. On the other hand, other broker systems are more message-centric, offering features for managing message queues, guaranteed delivery, and complex routing patterns, making them versatile for a wider range of messaging scenarios. The choice between them depends on the specific requirements of your application.

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

Windowing by Processing Time:

- Windowing by processing time defines windows based on the system clock's time. It segments the data stream into fixed time intervals, such as 1-minute or 5-minute windows.
- Events are assigned to windows based on when they arrive at the processing system.
- It's suitable for scenarios where event timing is not critical, and you want to analyze data as it arrives, without considering the event timestamps.

Windowing by Event Time:

- Windowing by event time segments data based on the timestamps attached to each event. Events are grouped into windows based on their actual occurrence time.
- This approach is essential when event order or time sequence matters, especially in scenarios like log analysis, financial data processing, or IoT applications where event timing is critical.
- It can handle out-of-order events, where events with earlier timestamps arrive later due to network delays or other factors.It can handle out-of-order events, where events with earlier timestamps arrive later due to network delays or other factors.

Watermarks in Streaming Processing:

Watermarks are a critical concept in event time-based streaming systems, such as Apache Kafka. They are used to deal with the challenges of out-of-order events and late arrivals. Here's how watermarks work:

- **Event Timestamps:** Each event in a streaming system is tagged with a timestamp indicating when it actually occurred.
- **Watermarks:** Watermarks are progress indicators generated by the streaming system. They represent a threshold or a point in event time up to which the system believes it has received all relevant events. Watermarks are generated based on observed event timestamps and may lag behind real-time.
- **Handling Late Events:**
  - When a watermark passes a certain time T, the system considers all events with timestamps earlier than T as complete for processing.
  - Any late-arriving events with timestamps older than T are considered "out-of-order" and are processed accordingly.
  - Late events can be dropped, sent to a separate late data stream, or included in a special late data processing window.
- **Advantages of Watermarks:**
  - Watermarks allow streaming systems to provide correctness guarantees even in the presence of out-of-order events.
  - They enable accurate event time-based windowing and aggregation without waiting indefinitely for late events, which can lead to resource exhaustion.

In summary, windowing by processing time is based on the system clock and is suitable for real-time processing where event timestamps are not crucial. Windowing by event time, on the other hand, considers the actual event timestamps, making it suitable for scenarios where event order and timing are important. Watermarks help streaming systems maintain correctness in event time-based processing by handling out-of-order and late-arriving events effectively.