Data Intensive Computing - Review Questions 3

Deadline: Sep. 20, 2025

Group AKA
Ahmad Al Khateeb
Kusumastuti Cahyaningrum
Aleksandra Burdakova

1. **Briefly compare the `DataFrame` and `DataSet` in SparkSQL and via one example show when it is beneficial to use `DataSet` instead of `DataFrame`.**
   DataFrame: represents data as a collection of Row objects. It does not check column types at Compile Time, so type-related errors are caught only at Runtime.
   DataSet: use classes to represent rows as strongly-typed objects. This allows the compiler to catch errors at Compile Time, making code safer and less error-prone (usually used in production).

   Example:
   Using the same data as No. 2, suppose we accidentally try to access a non-existing field due to a typo (column ge instead of age):
   DataFrame → people.filter("ge < 20") → shows error in Runtime
   DataSet → people.filter(x => x.ge < 20) → shows error in Compile Time

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 code is using a window function where the aggregation avg(col("age")) is computed over the previous row, current row and next row, and it will create a new column called avg_age.

   The final table will have 3 columns: name, age, avg_age. The row order is arbitrary.

name	age	avg_age
Michael	15	22.5

Andy	30	21.2
Justin	19	20.3
Andy	12	16.7
Jim	19	14.3
Andy	12	15.5

3. **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 dividing a continuous stream of data into discrete chunks (windows). It can be either based on processing time or on event time. When processing time, the system buffers up incoming data into windows until some amount of processing time has passed (e.g., if we decide the processing time to be 1 sec, then data is buffered into the window until 1 sec is passed). While in event time, the system reflects the times at which events actually happened. This opens the door for out-of-order events (because of latency). Watermarking helps a stream processing system to deal with lateness by continuously, and on a regular basis, sending watermarks as part of the data stream that carry a timestamp t. A received function W(t) declares then that event time has reached time t in that stream, and the current window is complete and a new one is started.

4. **Spark Streaming is based on micro-batches, while Structured Streaming introduces continuous processing. Compare these approaches in terms of throughput, latency, and fault tolerance.**
   In Spark Streaming (micro-batch model), incoming data is divided into small, fixed-time intervals (sets of bounded data), and each batch is then processed using Spark's batch engine. It has high throughput because each batch can be processed in parallel using the full Spark engine, which is optimized for handling large volumes of data. The latency is directly dependent on the batch interval, making it not suitable for sub-second latency requirements. From the aspect of fault tolerance, Spark uses RDD lineage and checkpointing to recompute lost data, which guarantees that all processes will run when failure.
   Structure Streaming (continuous processing) is introduced as part of Spark SQL. Each node in the system continually listens to messages from other nodes and outputs new updates to its child nodes. It can achieve lower latency but often with lower throughput, since it processes one record at a time rather than batches. When it comes to latency, it's better for real-time applications. And finally, the continuous processing is more complex in terms of failure recovery than micro-batch.

5. **What happens when Spark drops or fills in missing values during processing? Who or what might disappear in this process? What alternative designs could help keep missingness visible rather than hiding it?**

   If missing values are dropped or filled, it will result in bias or distort the distribution of the data. People and experiences can disappear from the dataset. Filling in with defaults (0, “unknown”) can create a false sense of completeness.

   Alternative design: actually report the missingness instead of assuming certain values, trace why it’s missing, represent uncertainty instead of erasing it.

6. **Schemas in Spark SQL are strict and enforce fixed column names and types. What are the risks of forcing diverse, real-world data into these rigid structures? Imagine an alternative system that could better represent multiple identities, ambiguity, or community-defined categories.**

   Complex identities (e.g., gender, race, culture) get flattened into rigid boxes. Values that don’t fit the schema may be deleted/misclassified. Systems reflect the worldview of the schema designer, not the data subjects.
   Alternative: allows flexible schema (NoSQL, graph database), self-identification, multiple categories, context-aware structures