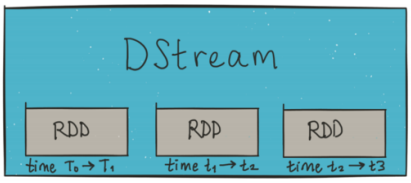
Data Intensive Computing – Review Questions 4

Adriano Mundo, Riccardo Colella (Group 29)

1. **What is `DStream` data structure, and explain how a stateless operator, such as `map`, works on `DStream`?**

   `DStreams` stand for discretized streams and the main idea behind them is to structure a streaming computation as a series of short, stateless, deterministic batch computations on small time intervals. The easiest way to define a `DStream` is to say that it is a collection of RDDs. Precisely, it is a continuous sequence of immutable, partitioned datasets (RDDs) of the same type representing a continuous stream of data. During the execution, Spark Streaming receives the data stream, divides it into batches and stores them in Spark's memory as RDDs.

   `DStreams` support many of the stateless (applied separately on the RDDs in each time interval) transformations available in typical batch frameworks such as Spark RDDs, including `map`. The result of applying `map(func)` is to return a new `DStream` by passing each element of the source `DStream` through a function `func`.

2. **Explain briefly how `mapWithState` works, and how it differs from `updateStateByKey`.**

   `mapWithState` and `updateStateByKey` are stateful operations that allow to build stateful stream processing pipelines. The motivation for stateful operations is that by design streaming operators are stateless and don't know anything about previous records. If there's the need to react to new records appropriately given the previous ones it is mandatory to provide a checkpointing directory for persistent storage.

   The `updateStateByKey` is executed on the whole range of keys in `DStream`. As a result, performance of this operation is proportional to the size of the state. The operation allows you to maintain arbitrary state while continuously updating it with new information. To use this, you will have to do two steps: define the state and define the state update function. It returns a new "state" `DStream` where the state for each key is updated by applying the given function on the previous state of the key and the new values for the key. Differently, the `mapWithState` is executed only on a set of keys that are available in the last micro batch. As a result, performance is proportional to the size of the batch. With respect to the other operations this approach increases performance of processing state. This is the main difference between the two operations.

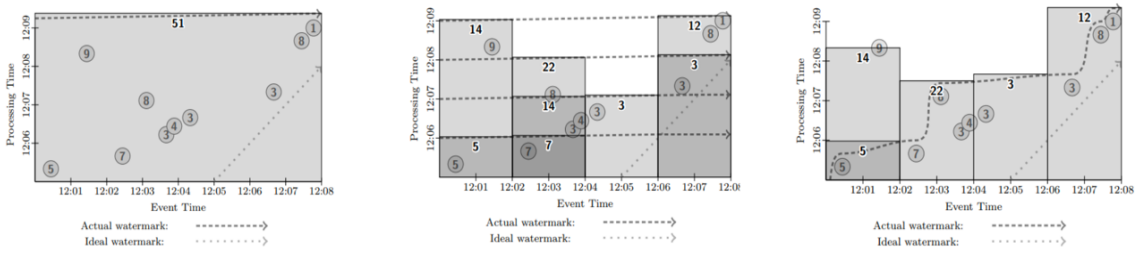
3. Explain how Spark streaming and MillWheel provide fault tolerance?

The fault tolerance feature is important to keep the system working as long as possible, since losing a portion of the data stream can lead to inaccurate decisions.

Spark Streaming provides two strategies for fault tolerance: replication and upstream backup. It periodically writes metadata information into HDFS directories. Hence, when a node fails, another can restart and continue processing from the last checkpoint. On the other hand, input data should be replicated in memory between nodes. Thus, if a node fails, the other node can still process the input data. It does an excellent job of efficient checkpointing but limits the space of operators that are available to the user code.

MillWheel was designed with fault-tolerance and scalability in mind. It provides fault-tolerance at the framework level, where any node or any edge in the topology can fail at any time without affecting the correctness of the result. As part of this fault tolerance, every emitted record is checkpointed before delivery in the system. Thus, they are guaranteed to be delivered to its consumers.

4. Through the following pictures, explain how Google Cloud Dataflow supports batch, mini-batch and streaming processing.



These pictures are *Dataflow model* examples which describe the execution of a pipeline over 10 input data. Each one plots the input and the output across two dimensions: on the X axis the event time (the time at which the event occurred) and on the Y axis the processing time (the time at which an event is observed).

Batch processing: in a classic batch system the system waits for all the data to arrive and when it has seen all of the inputs group them together into one bundle producing its single output of 51. There are no specific transformations, hence it calculates a sum over all of the even time. Since batch processing is time-agnostic the result is contained within a single global window covering all of the event time and since outputs are calculated only when all the inputs are received the result covers all of processing time for the execution. There's only a watermark at the end of the time.

Mini-batch processing: in this case the picture shows the execution of a micro-batch engine over the data source with one-minute micro-batches. The system gathers the input data for one minute, processes them and then repeats. The watermarks for the current batch start at the beginning of the time and advances to the end of the time. At the end, there's a watermark for every micro-batch and the outputs are different from the one considered in the batch processing case.

Streaming processing: the last case shows a pipeline executed on a streaming engine and most of the windows are emitted when the watermark passes them. The input data 9 is late with respect to the watermark and the system does not realize that datum has not been injected and allows the watermark to proceed. When the late datum arrives, it causes the first window range to retrigger with an updated sum. The overall latency is worse with respect to the micro-batch system, but it is nice to have one output per window.