KTH Royal Institute of Technology
Stockholm

School of Electrical Engineering and
Computer Science

Data-Intensive Computing - ID2221

Review Questions 4

Author
Emil Ståhl

Author
Selemawit Fsha

October 3, 2021

Review Questions 4 - ID2221

Emil Ståhl and Selemawit Fsha Nguse

October 3, 2021

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

A DStream (Discretized Stream) is a continuous sequence of RDDs (Resilient Distributed Dataset) representing a stream of data. The term "continuous" means that we never start or stop receiving data as part of the stream. Any operation applied on a DStream translates to operations on the underlying RDDs. A DStream can be created from a simple network socket, file system, or more exotic sources such as Kafka and Flume. DStreams support most of the transformations that are available on normal Spark RDDs which allows data inside Input DStreams to be modified. One such operator is **map** which maps each element of the source DStream through a given function. For example, when reading from a stream of data, we can use the stateless **flatMap** operator to split the records text to words. The property of stateless refers to **not** being able to access data from a previous point in time in the current point in time. In Spark Streaming, state is built into the pipeline and mostly abstracted away from the user in order to be able to expose aggregation operators, such as agg.

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

mapWithState is a stateful stream operation and is executed only on set of keys that are available in the last micro batch of the stream. A micro batch is defined as the procedure in which the incoming stream of messages is processed by dividing them into group of small batches. This helps to achieve the performance benefits of batch processing. Spark's stateful operations work with key-value entries. **mapWithState** is capable of mapping data with respect to previous states. For example, it can easily recognize changes in data.

Apart from **mapWithStat**, we have **UpdateStateByKey** that counts the state of the global key and returns the state of the key before each batch interval, regardless of data entry. **UpdateStateByKey** updates the status of existing keys and performs the same update function for each new key. If none is returned after the state is updated by the update function, the state corresponding to the key is deleted at this time. It can return all previous historical data, including new, changed, and unchanged data, within a specified batch interval. Because updateStateByKey must use checkpoints it can take up a large amount of data when the amount of data is too large, affecting performance and being inefficient.

mapWithState, on the other hand, only returns the value of the changed key. The advantage of this is that we can only care about the key that has changed, and for no data input, we will not return the data for the key that has not changed. In this way, even with a large amount of data, the checkpoint will not consume as much storage and be more efficient.

mapWithState was added to Spark to combat the weaknesses of **updateStateByKey** which are:

- at every batch arrival, **updateStateByKey** iterates over all entries in state store
- not flexible return value
- doesn't provide timeout mechanism
- **updateStateByKey** is focused on handling new data

mapWithState solves this by adding the following functionality:

- stays focused only on entries concerned by state change
- possible to return different state than the parameter
- handles timeouts automatically through appropriated API method
- adds partitioner

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

Google DataFlow is a managed service for unified batch and stream data processing that lets you express your data processing pipeline using FlumeJava. When run in batch mode, it gets executed on the MapReduce framework. But when run in streaming mode it gets executed on the MillWheel framework. Dataflow uses concepts such as Windowing and Triggering.

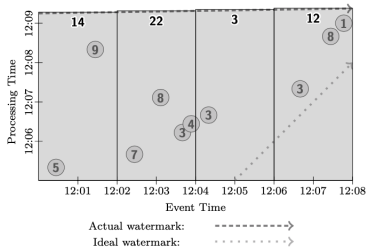
- Window Windowing determines where in event time data are grouped together for processing.
  - Fixed time windows (tumbling windows)
  - Sliding time windows
  - Session windows
- Triggering determines when in processing time the results of groupings are emitted as panes.
  - Time-based triggers
  - Data-driven triggers
  - Composit triggers

In Google Dataflow, users can make use of three processing models:

- Batch processing (time- and data based triggers)
- Mini-batch processing with fixed window (time based trigger)
- Streaming processing with fixed window (watermark based trigger)

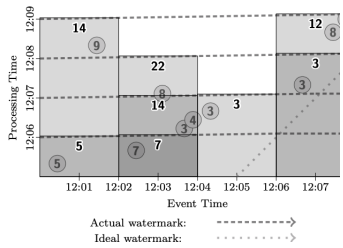
Shown below is three images characterizing the three processing models.

3.1 Batch processing



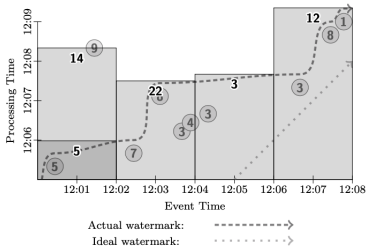
This image shows how a pipeline of data is executed on a classic batch processing engine where we wait for all data to arrive and pile up before processing it in an event-time order (i.e. when the event occurred on the producing device). Batch processing should be used when data freshness is not a mission-critical issue or you are working with large data-sets and are running a complex algorithm that requires access to the entire batch. Batch process if you get access to the data in batches rather than in streams, for example when you are joining tables in relational databases.

3.2 Mini-batch processing



This image shows how a pipeline of data is executed on a micro- or mini-batch engine where each batch corresponds to a window of one minute. Here, the system gathers data during one minute and is then processing the gathered data. This repeats over and over again. Each micro-batch gets a new watermark. Micro-batch processing is very similar to traditional batch processing in that data are usually processed as a group. The primary difference is that the batches are smaller and processed more often. In practice, there is little difference between micro-batching and stream processing, and the terms would often be used interchangeably in data architecture descriptions and software platform descriptions. Micro-batch processing is useful when we need very fresh data, but not necessarily real-time.

3.3 Streaming processing



Finally, the pipeline gets executed on a streaming processing model as shown above. In stream processing, we process data as soon as it arrives in the storage layer which can be very close to when the data is generated. Here the windows

are emitted once the watermark passes them. If an arriving event lies within the watermark, it gets used to update a query. Otherwise, if it's older than the watermark, it will be dropped. Streaming processing is what is used in actual streaming of, for example, live sports where a late package don't get processed. Stream processing is especially suitable for applications that exhibit three application characteristics:

- Compute Intensity
- Data Parallelism
- Data Parallelism

The reason for using streaming processing is that some data naturally comes as a never-ending stream of events. To do batch processing, you need to store it, stop data collection at some time and processes the data. Then you have to do the next batch and then worry about aggregating across multiple batches. In contrast, streaming handles never ending data streams gracefully and naturally. Also, sometimes data is huge and it is not even possible to store it. Stream processing is the right approach when data is being generated in a continuous stream and arriving at high velocity or sub-second latency is crucial.

4 Explain briefly how the command pregel works in GraphX?

Pregel is a large-scale graph-parallel processing platform developed at Google. Inspired by the bulk synchronous parallel (BSP) model. Pregels execution model consists of sequences of iterations, called supersteps where each vertex (node) can read messages that were sent to it in the previous superstep. Each vertex sends messages to other vertices which they receive in the next superstep. All communication between vertices is done by sending messages. The whole algorithm terminates when all vertices are simultaneously inactive and there are no messages in transit. However, Pregels has limitations making it inefficient if different regions of the graph converge at different speed and the runtime of each phase is determined by the slowest machine in the network.

Pregel can be used together with GraphX which is a library to perform graph-parallel processing in Spark. Spark represent graph structured data as a property graph. GraphX provides Pregel message-passing and other operators on top of RDD. One important characteristics of distributed graph processing which makes it different from the classic MapReduce approach is the iterative nature of many the algorithms. Pregel is one of the computation models that supports such kind of processing very well, GraphX comes with its own Pregel implementation allowing it to iterate on graphs. GraphX implementation of Pregel is a bit different from the original API. In GraphX, Pregel lets the exchanged messages access the attributes of both source- and destination vertices. In addition to that, GraphX uses a reduce approach that doesn't need to wait

for all messages to be received by each vertex to begin the computation of its new value. Instead it computes them partially on each partition and merges into the final value at the end.

Pregel works as follows; The Pregel method take 2 argument lists

```
pregel(< argument list 1 >)(< argument list 2 >)
```

argument list 1 is a list of value arguments, argument list 1 is to set a initial value. **argument list 2** is a list of functional arguments, which is an example of Scala higher order function, meaning each function argument is free to have its own execution algorithm, like its own logic.

5 Assume we have a graph and each vertex in the graph stores an integer value. Write three pseudo-codes, in Pregel, GraphLab, and PowerGraph to find the minimum value in the graph.

Pregel:

```
i_val = val

for each message m
    if m < val then val := m
if i_val == val then
    vote_to_halt
else
    for each neighbor v
        send_message(v, val)
```

Initially do some computation and send the result to the neighbours and go to the next iteration

- First define initial value 'val' and keep it locally in 'i-val' and execute that code inside each vertex
- If it receive any value 'm' and initially it check if it has any new value that comes from the neighbors or not. If it has a new value from the neighbors which is less than current value than update the current value val:=m
- Then it checks if the updated value is similar to the current value it doesn't do anything else so it changes its state to inactive
- But if it is a new value greater than the previous value then it goes to the neighbour and send the current value to the neighbour

- Each node sends it's value to its neighbour
- Finally terminate when all get the minimum value

GraphLab:

PowerGraph: