ID2221 - Data Intensive Computing - Review Questions 4

Deepak Shankar Reethika Ambatipudi
deepak | reethika @kth.se

September 29, 2023

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

A DStream (Discretized Stream) in Apache Spark Streaming is used for processing real-time data streams. It represents a sequence of data arriving continuously over time, and it is analogous to Resilient Distributed Datasets (RDDs) in Spark. However, DStreams are designed for processing streaming data, and they provide a high-level API for performing stateful and stateless operations on the data.

Stateless operators like map work on DStreams by applying a function independently to each element as it arrives in the stream. These operators do not rely on or maintain state across elements, making them suitable for simple, element-wise transformations in real-time. Each element is processed in isolation within a batch, and the results form a new DStream. This allows for straightforward, stateless data processing on streaming data.

Q-2 Explain briefly how mapWithState works, and how it differs from updataStateByKey.

mapWithState and updateStateByKey are two stateful operations in Apache Spark Streaming, used for maintaining and updating state while processing streaming data. Here's a brief explanation of each and how they differ:

Feature	updateStateByKey	mapWithState
State Maintenance	Maintains and updates arbitrary state across multiple batches of streaming data.	Maintains and updates state but in a more functional and declarative manner.
Data Granularity	Key-based: Operates on key-value pairs, maintaining separate states for different keys.	Element-based: Operates on individual elements of the DStream, regardless of keys.
Complex State Handling	Suitable for complex state management scenarios where state updates may depend on both current data and previous state.	Simplified state management, ideal for scenarios where state transitions are defined for each element.
Data Independence	Each batch's state update can depend on both the current batch's data and the previous state.	The state update for an element can depend on the current element's data and the previous state.

In summary, while both mapWithState and updateStateByKey are used for stateful operations, they differ in their data granularity and approach. updateStateByKey is key-based and suitable for complex state management across batches, while mapWithState is element-based and provides a more functional, simplified approach for handling state at the granularity of individual elements in a DStream. The choice between them depends on the specific streaming use case and state management requirements.

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

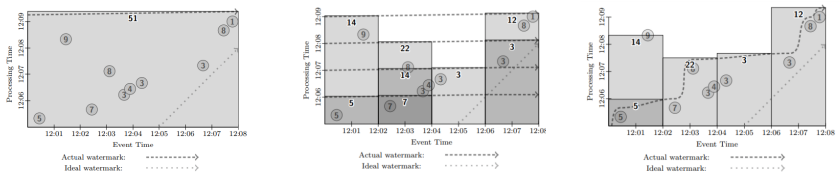

Figure 1: lineage graph - version 2

- **Batch Processing:** In batch processing, Dataflow processes data in finite, bounded datasets. We can run batch pipelines to process historical data or recurring large-scale data loads. As seen in the first image, as the sum of all processes nodes reach 51, the entire batch is executed. It is a Trigger at count (data-driven triggers).
- **Mini-Batch Processing:** Mini-batch processing involves processing data in small, fixed-size batches. While Dataflow primarily supports a unified model that seamlessly transitions between batch and streaming, we can implement mini-batch processing by configuring the processing window size. As seen in the second image, several windows are created based on time frame, and all the processes that fall into that particular window are executed. The sum of processes varies in each window as seen in the image. It is based on Trigger at period (time-based triggers).
- **Streaming Processing:** Dataflow provides native support for stream processing, allowing us to process unbounded data continuously. Streaming pipelines are designed to handle real-time data and can process events as they arrive. As seen in the third image, It is based on Fixed window, trigger at watermark (streaming). There are fixed interval windows of two minutes, but as the watermark approaches, the execution is triggered. The dotted line shows the flow of watermark.

Q-4 Explain briefly how the command pregel works in GraphX?

In GraphX, the pregel method is used to execute Pregel-style iterative graph computations on a graph. It stands for "Pregel" which is inspired by the Pregel model for distributed graph processing. Here's a brief explanation of how the pregel command works in GraphX:

- **1. Initialization:** We start by defining an initial graph with vertices and edges. Each vertex typically contains some data relevant to our problem, and each edge represents a relationship between vertices.
- **2. Vertex Program:** We specify a function called the "Vertex Program" that defines how each vertex processes its incoming messages and updates its state during each iteration. This function takes the current vertex's state, incoming messages from neighboring vertices, and produces a new state for the vertex. This function is defined by the user to implement the specific graph algorithm you're interested in.
- **3. Message Sending:** In each iteration, each vertex can send messages to its neighboring vertices. These messages are typically used to communicate information about the vertex's state or to request information from neighbors.
- **4. Message Aggregation:** Messages sent by vertices are aggregated for each target vertex, typically using a user-defined aggregation function. This can be useful when multiple messages are sent to the same vertex, and you want to combine them into a single message.
- **5. Iteration:** The process of message sending, aggregation, and vertex state updates is repeated for a fixed number of iterations or until a termination condition is met. Each iteration represents a step in the algorithm.
- **6. Termination Condition:** We can specify a termination condition for the algorithm, such as a maximum number of iterations or a convergence criterion based on the changes in vertex states.
- **7. Result**: After the iterations are complete, we have the final state of each vertex, which represents the result of the algorithm you implemented.

In summary, The pregel method in GraphX allows us to express and execute iterative graph algorithms in a distributed manner by defining how vertices process messages and perform iterative graph computations. It works by initializing a graph, executing custom vertex programs, and passing messages between vertices in each iteration until a termination condition is met. The result is an updated graph with the final computation results. This approach is well-suited for various graph-based computations, including graph traversal, connected components, and more.

Q-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 Pseudo-Code:

```
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)
```

GraphLab Pseudo-Code:

```
GraphLab_MinimumValue(i)
    // Initialize variables
    min_val = val;

    // Iterate over incoming neighbors
    foreach(j in in_neighbors(i))
        // Compute minimum value among neighbors
        if (R[j] < min_val)
            min_val = R[j];

    // Update the minimum value
    R[i] = min_val;

    // Trigger neighbors to run again
    foreach(j in out_neighbors(i))
        signal vertex-program on j;
```

PowerGraph Pseudo-Code:

```
PowerGraph_MinimumValue(i)
    // Initialize variables
    min_val = val;

    // Gather phase: Compute minimum value from incoming neighbors
    Gather(j -> i)
        return (R[j] < min_val) ? R[j] : min_val; //val is calculated here.

    // Apply phase: Update the minimum value
    Apply(i)
        if (val < min_val)
        {
            min_val = val;
            R[i] = min_val;
        }
```

```
// Check for convergence and vote to halt if necessary
if (val == min_val)
    vote_to_halt;
else
    // Signal neighbors to re-run their computations
    Scatter(i -> j);
```