ID2221 Data Intensive Computing - Review Questions 4 - Group 19

by Artem Sliusarenko, Mihailo Cvetkovic and Eugen Lucchiari Hartz

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

A DStream (Discretized Stream) is a sequence of RDDs that represents a stream of data. It's designed for processing real-time data streams by splitting the continuous data stream into small batches, which are then processed by Spark's core engine. This design makes it possible to handle stream processing in a scalable and fault-tolerant way.

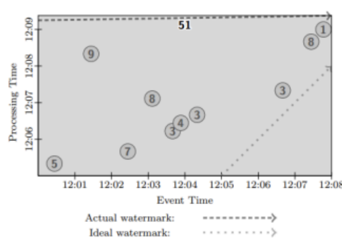
Operators that do not need to maintain any information across different batches of data are referred to as stateless operators. These operators process each batch of data independently, without considering what happened in previous or future batches. One example of a stateless operator is the map function.

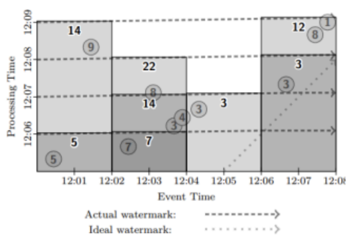
When applied to a DStream, the map operation works by applying a user-defined function to each element in the current batch of data. This transforms the elements and creates a new DStream, where each batch contains the transformed elements from the corresponding batch in the original DStream. Since each batch is handled separately, there's no dependency on data from other batches.

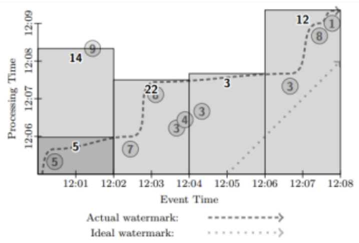
2.) Explain briefly how mapWithState works.

mapWithState is a stateful operation in Spark Streaming used to manage and update state information over time. It allows you to track and update specific keys in a stream as new data arrives in each micro-batch. The operation works by taking a StateSpec, which defines the rules and functions needed to manage the state associated with each key in the stream. For each key, an update function is applied that maintains and modifies the state based on the incoming data. This enables precise control over state management, making mapWithState useful for keeping track of evolving values for individual keys over time in a streaming application.

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







Batch processing:
The first picture represents Batch processing. In batch processing of bounded data (finite data), processing begins only after all the data is collected. For example, consider figure 1 above, if we have 10 events, we wait to receive all 10 before starting the processing. The processing time, shown by the dotted line at the top of the graph, starts once the last event arrives. In this case, the processing task is to calculate the sum of the values from all 10 events.

Mini-batch processing:
The second picture represents mini-batch processing. Mini-batch processing involves defining the window of size x, that can be time or data based, on both processing and event time/data. For example, on the middle figure above, we have a processing time window of size **1 min** and the event time window of size **2 min**. That means we need to group events that have been received for processing in **1 min** windows and additionally base that grouping on the **2 min** window of the event time.

Observe how at the **1 min** processing window of **12:06 to 12:07** we are grouping events based on the **2 min** event time window. Which groups the 2 events with values **3** and **4** in a **12:02 to 12:04** event time window and a single event with valu **3** into an event window of **12:04 to 12:06**.

Streaming processing:
The third picture represents streaming processing. The concept of watermarking helps the processing unit to deal with lateness. It provides the processing unit with the current time of the event streaming unit so that the processing unit can decide which events are late.

The last image defines the fixed window based on the event time, however, the processing is triggered when the watermark is received.

According to the graph the processing unit received the watermark **12:02** at the processing time **12:06**. It now will process the events buffered since the last watermark was received. Next watermark, **12:04** was received at approximately **12:07:30** so the processing unit grouped all buffered events since last watermark (**12:02**) and processed them. Processing unit time does not affect grouping!

4.) Explain briefly how the command pregel works in GraphX?

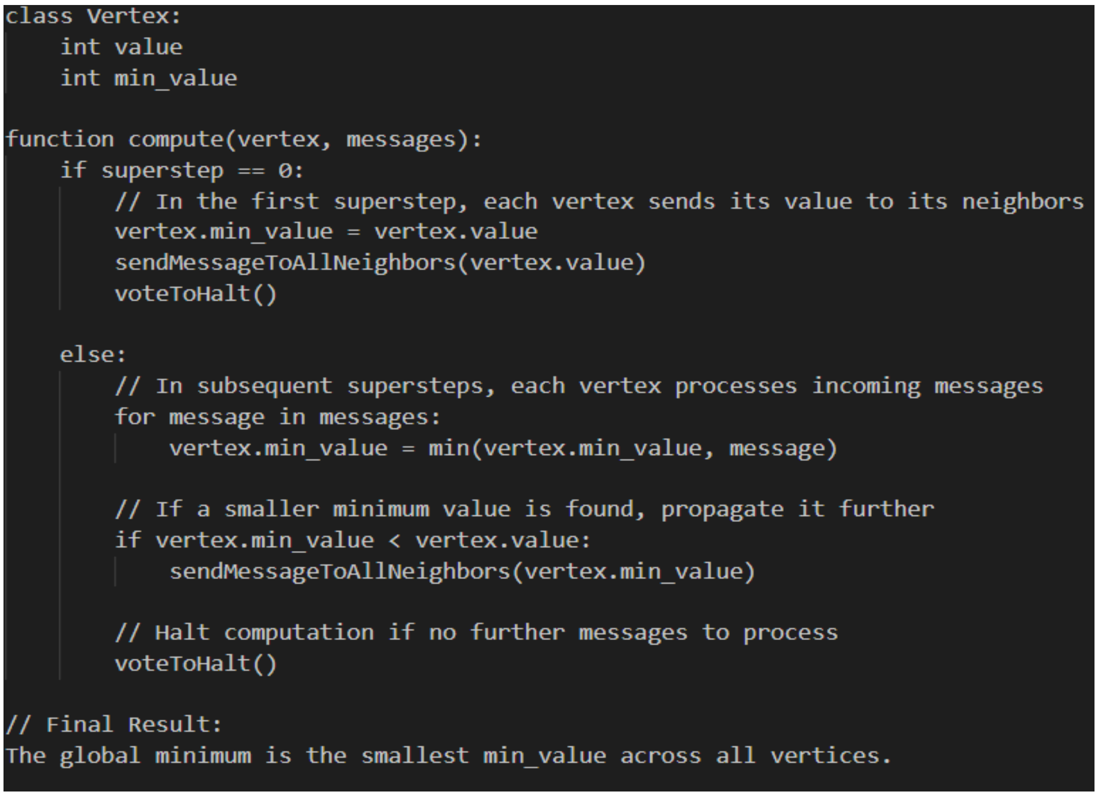
The **pregel** operation in GraphX is a message-passing algorithm that proceeds in synchronized iterations (supersteps). Each superstep has three key phases:
Vertex Program: Each vertex updates its value based on its current state and the messages received in the previous superstep.
Message Sending: Each vertex sends messages to its neighbors (or all other vertices) based on its updated value
Message Aggregation: Messages are aggregated (e.g., summed, minimized) for each vertex and passed to the next superstep.

The **pregel** function has the following key inputs:
Initial message: The message each vertex receives before any supersteps (e.g., infinity for a shortest-path algorithm)
Vertex Program (vprog): Function that takes a vertex ID, the current value, and an incoming message, and computes the new vertex value.
Send Message (sendMsg): Function to define how messages are sent from one vertex to its neighbors.
Message Combiner (mergeMsg): Function to aggregate multiple incoming messages for a vertex (e.g., selecting the minimum or sum of messages)

The Pregel command in GraphX allows iterative, synchronized, distributed computations on graphs by enabling vertices to exchange messages and update their values based on the results of those messages.

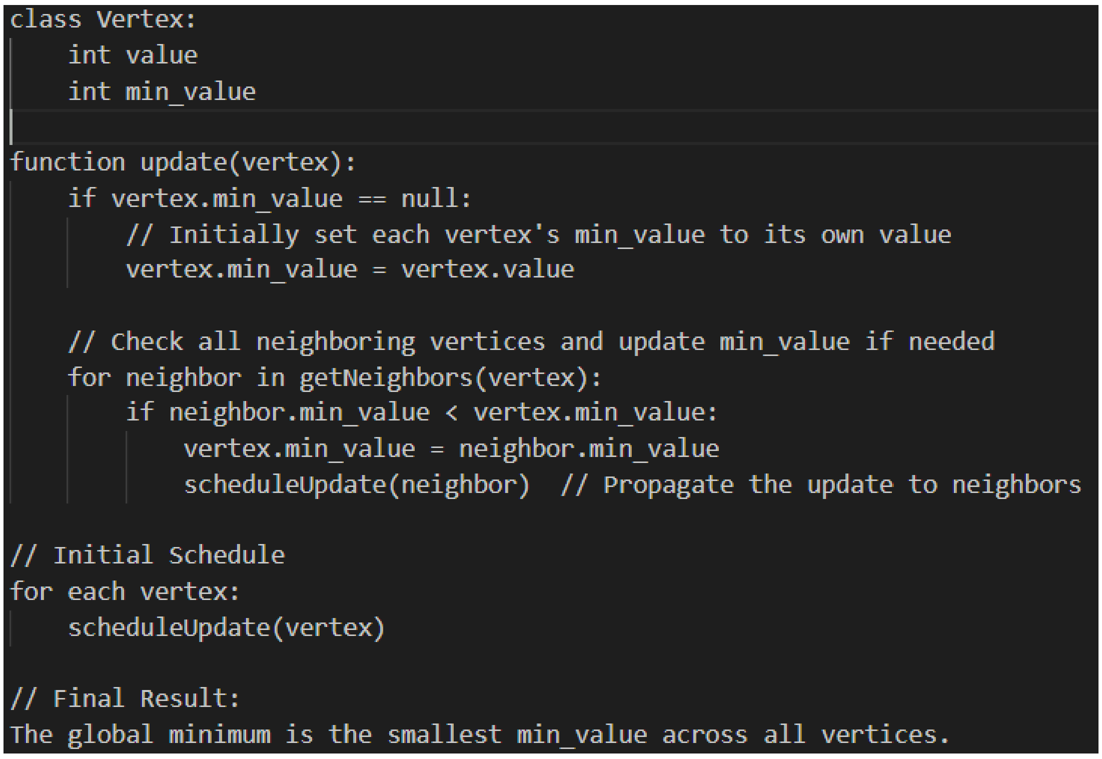
5.) Assume we have a graph and each vertex in the graph stores an integer value. Write three pseudo-codes, in Pregel, GraphLab, and PowerGraph to find the minimum value in the graph.

Pregel pseudo-code



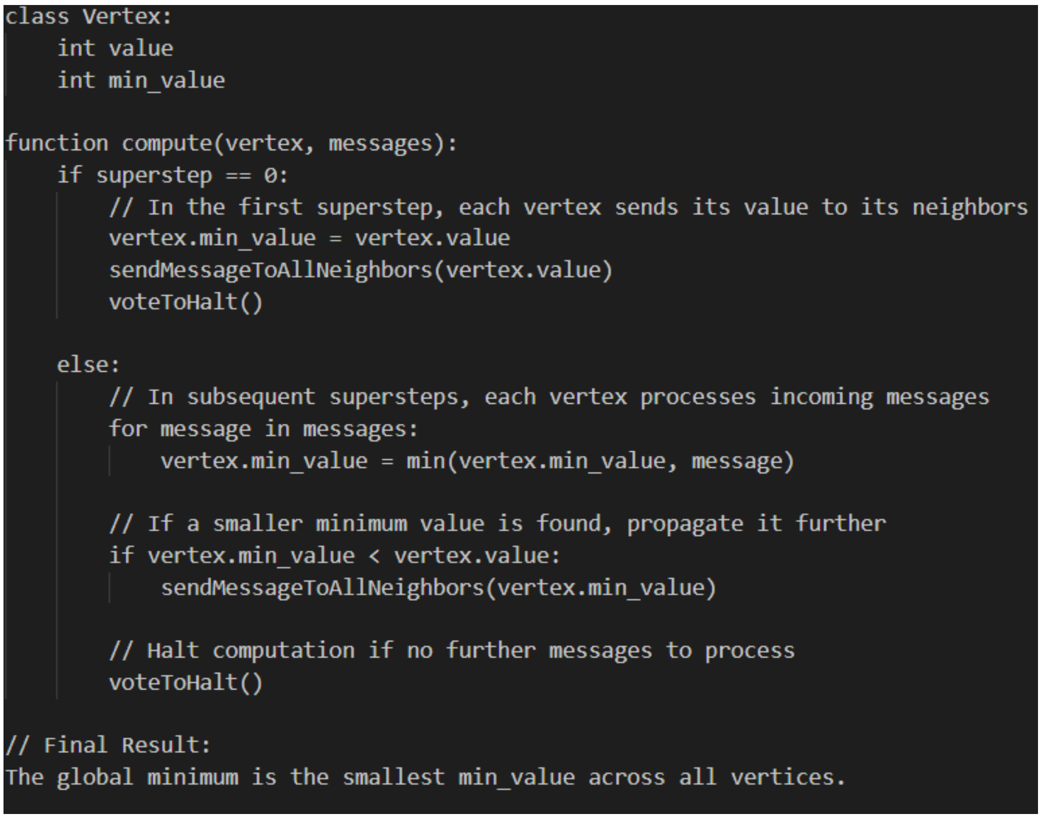
```
class Vertex:
    int value
    int min_value

function compute(vertex, messages):
    if superstep == 0:
        // In the first superstep, each vertex sends its value to its neighbors
        vertex.min_value = vertex.value
        sendMessageToAllNeighbors(vertex.value)
        voteToHalt()

    else:
        // In subsequent supersteps, each vertex processes incoming messages
        for message in messages:
            vertex.min_value = min(vertex.min_value, message)

        // If a smaller minimum value is found, propagate it further
        if vertex.min_value < vertex.value:
            sendMessageToAllNeighbors(vertex.min_value)

        // Halt computation if no further messages to process
        voteToHalt()

// Final Result:
The global minimum is the smallest min_value across all vertices.
```

GraphLab Pseudo code



```
class Vertex:
    int value
    int min_value

function update(vertex):
    if vertex.min_value == null:
        // Initially set each vertex's min_value to its own value
        vertex.min_value = vertex.value

    // Check all neighboring vertices and update min_value if needed
    for neighbor in getNeighbors(vertex):
        if neighbor.min_value < vertex.min_value:
            vertex.min_value = neighbor.min_value
            scheduleUpdate(neighbor)  // Propagate the update to neighbors

// Initial Schedule
for each vertex:
    scheduleUpdate(vertex)

// Final Result:
The global minimum is the smallest min_value across all vertices.
```

Power-Graph Pseudo



```
class Vertex:
    int value
    int min_value

function compute(vertex, messages):
    if superstep == 0:
        // In the first superstep, each vertex sends its value to its neighbors
        vertex.min_value = vertex.value
        sendMessageToAllNeighbors(vertex.value)
        voteToHalt()

    else:
        // In subsequent supersteps, each vertex processes incoming messages
        for message in messages:
            vertex.min_value = min(vertex.min_value, message)

        // If a smaller minimum value is found, propagate it further
        if vertex.min_value < vertex.value:
            sendMessageToAllNeighbors(vertex.min_value)

        // Halt computation if no further messages to process
        voteToHalt()

// Final Result:
The global minimum is the smallest min_value across all vertices.
```