Group: Group Therapy
Casper Kristiansson
Nicole Wijkman

RQ 4: Data-Intensive Computing

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

DStream is short for Discretized Stream and is a sequence of RDDs which represent a stream of data. It is a framework designed for processing real-time data streams. A continuous stream of data is divided into small batches which are processed by Spark's core engine. This makes the stream processing both scalable and fault-tolerant.

Operations that do not rely on maintaining state across batches of data are called stateless operators. A stateless operator will process each batch separately, not considering any previous data from previous batches. Map is an example of a stateless operator.

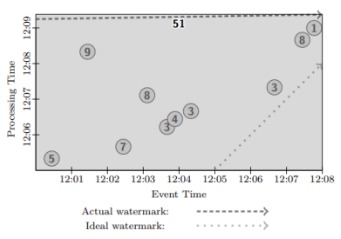
When it comes to DStream, the map operation applies some kind of user-defined function to each element within a batch. This produces transformed elements, which are collected into a new DStream. Each batch in the new DStream contains the results of the corresponding batch in the original DStream. Each batch is processed independently, without any relation on data from future or past batches.

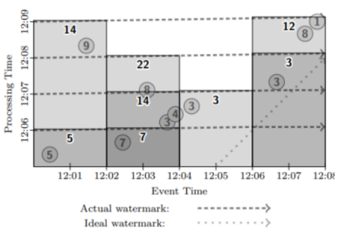
2. Explain briefly how mapWithState works, and how it differs from updateStateByKey.

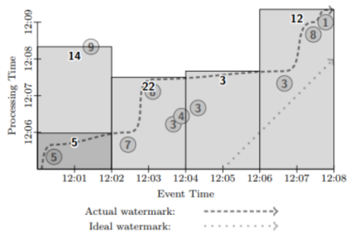
MapWithState and updateStateByKey are both stateful stream operations used in streaming data processing to keep track of and update information as new data arrives. They are important tools that maintain and update states across batches of data.

MapWithState operates selectively, focusing only on a set of keys available in the last micro-batch of data. It takes a StateSpec as an argument, which is a specification providing rules and functions necessary to maintain and modify state information associated with specific keys in a DStream. When using mapWithState, an update function is applied to each key individually, enabling precise state management for each key. By contrast, updateStateByKey operates on all keys available within each micro-batch, making it more suitable for more global state management. One could say that mapWithState is good for focusing on specific things, while updateStateByKey is better for keeping track of everything.

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







The first graph represents Batch Processing. Batch processing involves handling data in discrete chunks, often referred to as “batches.” In Google Cloud Dataflow, batch processing is made possible by defining processing windows, as seen in the picture. These windows allow the system to trigger processing once it has received all the data tokens within a given window. For example, in this scenario, the processing function combines all tokens into a single result, which is computed as 51. This approach allows users to efficiently manage and process data in structured, batch-like intervals.

The second graph represents mini-batch processing. In mini-batch processing, the buffering is done both at the source and the destination. The data is processed in fixed-sized mini-batches at regular intervals, as is indicated by the partitioned windows in the image. Each window corresponds to a mini-batch of data processed within a specific time window (event time). Mini-batch processing in Google Cloud Dataflow allows users to process data more frequently than with traditional batch processing while still working with structured, manageable chunks of data.

The third graph represents streaming processing. This graph shows continuous, real-time data processing where the data processing is triggered by the watermark. In streaming processing, data is processed as it arrives, with no distinct batch intervals. Google Cloud Dataflow supports streaming processing by enabling users to define streaming pipelines that process data as it becomes available, making it suitable for real-time analytics, monitoring, and alerting.

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

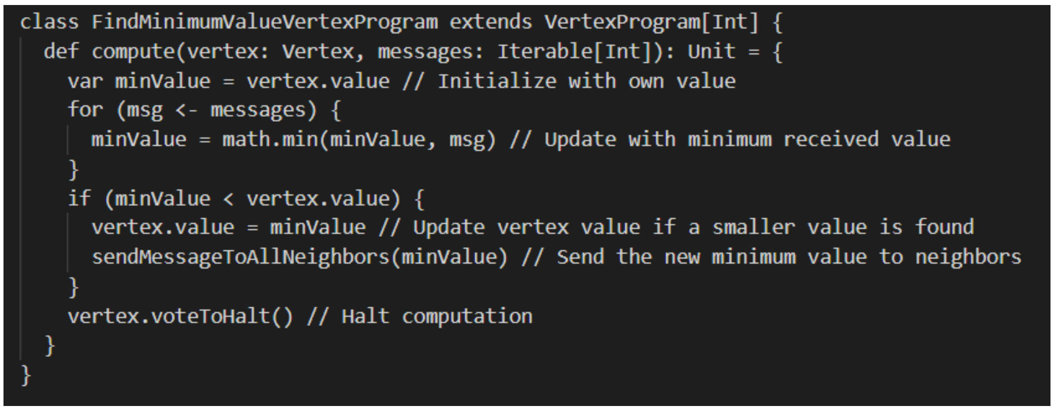
GraphX is a graph processing library built on top of Apache Spark, and it provides a Pregel-like API for graph computations. The Pregel operator in GraphX is a tool for expressing and executing iterative graph algorithms efficiently and it operates in a series of steps “super-steps.” With every super-step, vertices in the graph will receive and process messages with information needed for the computation from their neighbouring vertices. The message computation is done in parallel, enhancing the performance of iterative graph algorithms.

The pregel operator is notably very flexible, since users can define their own custom messaging functions to dictate how the messages are constructed and sent between the vertices. It is thus suitable for a very wide range of graph computations. The operator will keep iterating until there are no more messages to process or the maximum number of iterations have been reached.

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:

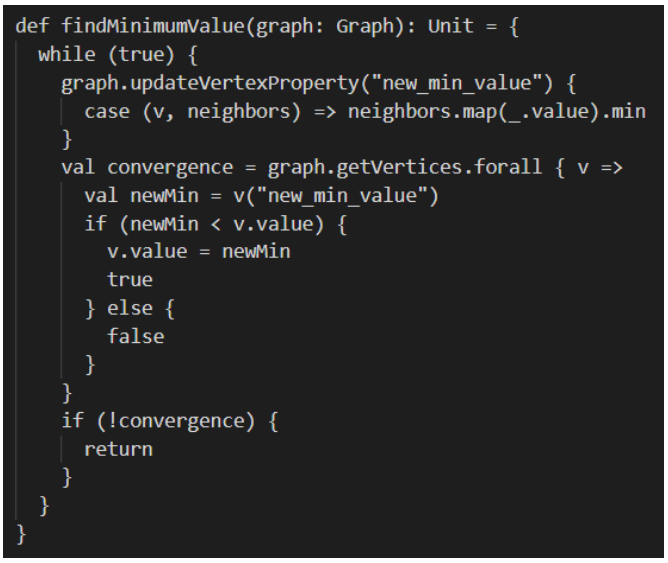
The Pregel framework is a vertex-centric model, so in the Pregel pseudo-code each vertex initializes with its own value as the minimum. It receives messages from neighboring vertices, updating its minimum value. If the minimum value changes, it updates its value and sends the new minimum to neighbors. The computation halts when there are no more changes.'



```
class FindMinimumValueVertexProgram extends VertexProgram[Int] {
  def compute(vertex: Vertex, messages: Iterable[Int]): Unit = {
    var minValue = vertex.value // Initialize with own value
    for (msg <- messages) {
      minValue = math.min(minValue, msg) // Update with minimum received value
    }
    if (minValue < vertex.value) {
      vertex.value = minValue // Update vertex value if a smaller value is found
      sendMessageToAllNeighbors(minValue) // Send the new minimum value to neighbors
    }
    vertex.voteToHalt() // Halt computation
  }
}
```

GraphLab Pseudo-Code:

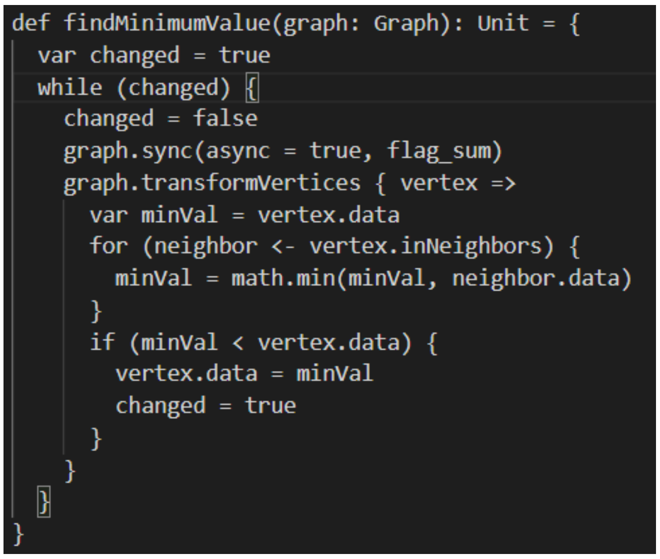
The GraphLab pseudo-code iteratively updates the "new_min_value" property for each vertex by aggregating minimum values from neighbors. It checks if any vertices have new minimum values, updates them, and continues until convergence. Convergence is reached when no vertex's value changes.



```
def findMinimumValue(graph: Graph): Unit = {
  while (true) {
    graph.updateVertexProperty("new_min_value") {
      case (v, neighbors) => neighbors.map(_.value).min
    }
    val convergence = graph.getVertices.forall { v =>
      val newMin = v("new_min_value")
      if (newMin < v.value) {
        v.value = newMin
        true
      } else {
        false
      }
    }
    if (!convergence) {
      return
    }
  }
}
```

PowerGraph Pseudo-Code:

The PowerGraph pseudo-code utilizes an iterative approach with a "changed" flag. It synchronizes data among vertices using a "sync" operation. Each vertex checks the minimum value of its neighbors and updates itself if a smaller value is found. The iteration continues until no more changes occur.



```
def findMinimumValue(graph: Graph): Unit = {
  var changed = true
  while (changed) {
    changed = false
    graph.sync(async = true, flag_sum)
    graph.transformVertices { vertex =>
      var minVal = vertex.data
      for (neighbor <- vertex.inNeighbors) {
        minVal = math.min(minVal, neighbor.data)
      }
      if (minVal < vertex.data) {
        vertex.data = minVal
        changed = true
      }
    }
  }
}
```