Data Intensive Computing - Review Questions 5

Daniele Montesi, Francesco Staccone

1. What are the differences between vertex-centric and edge-centric graph processing models?

   Vertex-centric and edge-centric are both possible implementations of the scatter-gather programming model for graph computation. The computation is structured as a loop and the difference among the two approaches lies in how the iteration happens:

   - in the **vertex-centric** implementation of the scatter-gather programming model both the scatter and the gather phase iterate over all vertices. To scale-up processing, edges are sorted by originating vertex and an index over the sorted edge list is built. The execution then involves random access through the index to locate edges connected to a vertex. This approach represents a trade-off between sequential and random access, favoring a small number of random access through the edge index over streaming a large number of potentially unrelated edges;
   - in the **edge-centric** implementation of the scatter-gather programming model both the scatter and the gather phase iterate over edges. This approach allows to stream the set of edges from storage and to avoid random accesses into it, that is often advantageous since a common property for graph is to have the edge set much larger than the vertex set and therefore the access to edges and updates dominates the processing cost. Doing so comes, however, at the cost of random access into the set of vertices, that can nevertheless be mitigated using *streaming partitions* over it, so that each partition fits in high-speed memory. This is what X-Stream does, partitioning also the set of edges such that they appear in the same partition as their source vertex and there is no need to sort the edge list.

2. Explain briefly when we should use pure graph-processing platforms (such as GraphLab), and when we need to use platforms such as GraphX?

   The comparison between general-purpose distributed dataflow frameworks (Map-Reduce, Spark, etc.) and specialized graph processing systems (GraphLab etc.) has been bypassed by the introduction of GraphX, a graph processing framework built on top of Spark that unifies the advantages of both the approaches, enabling a single system to address the entire analytics pipeline.
   Pure graph-processing platforms can effeciently execute iterative graph algorithms leveraging on the exposition of specialized abstractions backed by graph-specific optimizations, and outperform general-purpose distributed dataflow frameworks thanks to that. As a consequence, they should be used for PageRank and community detection algorithms on graphs with billions of vertices and edges. But they have some limitations: graphs are only part of the larger analytics process which often combines graphs with unstructured and tabular data, that leads to unnecessary data movement and duplication; they often abandon fault tolerance in favor of snapshot recovery; they do not generally enjoy the broad support of distributed dataflow frameworks.
   Here comes GraphX, that represents a great step forward concerning the above-mentioned limitations: unlike existing graph processing systems, it enables the composition of graphs with unstructured and tabular data and permits the same physical data to be viewed both as a graph and as collections without data movement or duplication. Moreover, it introduces a range of optimizations both in how graphs are encoded as collections as well as the execution of the common dataflow operators, and also achieves low-cost fault tolerance by leveraging logical partitioning and lineage.
   In conclusion, GraphX should be used on Natural graphs (those derived from natural phenomena), since it can achieve performance parity with specialized graph processing systems while preserving the advantages of a general-purpose dataflow framework.

3. 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 maximum value in the graph.

   1- Pregel
   The idea of Pregel is in dividing the task in 2 steps:
   1- wait until receive all messages from neighbors and compute;
   2- if there is an update, send the new value to all neighbors, else, vote for halt;
   If everyone votes for halt, the computation terminates;

```
Pregel_maxValue(i):

    //waiting messages & compute
    for each message in messages_received:
        if message > i.value:
            i.value = message
        else
            do_nothing()

    // send new value or vote for halt
    if has_updated(i.value):
        send_message(i.value)
    else:
        vote_for_halt()
```

2- GraphLab
The idea of Graphlab is to use a shared memory between 2 vertex in order to avoid the waiting of updated for each vertex.

```
GraphLab_maxValue(i):

    foreach(j in neighbor_list(i)):
        value_neigh = R[j]
        if value_neigh > i.value:
            i.value = value_neigh
            update = True
        else
            do_nothing

    // trigger neighbors in case of updates
    if update:
        for j in neighbor_list(i):
            signal vertex-program to j
```

3- GraphX The idea of GraphX is to speed up the computation switching easily from table to graph and viceversa. This is achieved thanks to the **triplet**. Three phases are followed:
1- **Gather**: messages are gathered and merged applying a same function on them all;
2- **Apply**: apply a specific function to the merged message;
3- **Scatter**: send the message to the vertex neighbors, if satisfied a predetermined condition;
Note that each phase is made to work sequentially: *gather* gets message from step S-1, *apply* computes in step S, *scatter* sends away messages that will be received on step S+1 by the neighbors.

```
GraphX_maxValue(i):

    // merge messages from neighbors
    gather(j -> i):
        for each j1, j2 in neighbor_list:
        math.max(j2,j1)

    // update performed on local value
    apply(i, max):
        i.val = max(i.val, received_val)

    // send my value value, if different from the neighbor's
    scatter(i -> j)
        if (i.value != j.value):
            send_message(j, i.value)
```