KEVIN JOHNSON

MODULE 7 READING COMPREHENSION

1. **Discuss the MapReduce paradigm in terms of Mappers and Reducers, and illustrate with an example. (2 Pts)**

   MapReduce is a programming framework with which you can write code to process large datasets in a distributed filesystem. The pattern of data processing in MapReduce can be explained in four steps:

   1. *Break files into records:* Read a set of input files and it up into records. This is handled by the input format parser.
   2. *Map:* Call the mapper function to extract a key and value from each input record.
   3. *Sort:* Sort all of the key-value pairs by key.
   4. *Reduce:* Call the reducer function to iterate over the sorted key-value pairs. If there are multiple occurrences of the same key, the sorting has made them adjacent in the list, so it is easy to combine those values without having to keep a lot of state in memory.

   The mapper is called once for every input record, and its job is to extract the key and value from the input record. For each input, it may generate any number of key-value pairs. It does not keep any state from one input record to the next, so each record is handled independently.

   The MapReduce framework takes the key-value pairs produced by the mappers, collects all the values belonging to the same key, and calls the reducer with an iterator over that collection of values. The reducer can produce output records.

2. **Describe some ways (at least 3) in which batch jobs achieve good performance while being easy to maintain. (2 Pts)**

   The handling of output from MapReduce jobs follows the same philosophy as the Unix philosophy. Experimentation is encouraged by being very explicit about dataflow: a program read its input and writes its output. In the process, the input is left unchanged, any previous output is completely replaced with the new output, and there are no other side effects. In MapReduce, by treating inputs as immutable and avoiding side effects, batch jobs not only achieve good performance but also become much easier to maintain:

   - If you introduce a bug into the code and the output is wrong or corrupted, you can roll back to a previous version of the code and rerun the job, and the output will be correct again. Even simpler, you could also keep the old output in a different directory and switch back to it.
   - Due to the ease of rolling back, feature development can proceed more quickly than in an environment where mistakes could mean irreversible damage. The principle of minimizing irreversibility is beneficial for Agile software development.

- If a map or reduce task fails, the MapReduce framework automatically reschedules it and runs it again on the same input. If the failure, is due to a bug in the code, it will keep crashing and eventually cause the job to fail after a few attempts. But if the failure is due to a transient issue, the fault is tolerated. This automatic retry is only safe because inputs are immutable and outputs from failed tasks are discarded by the MapReduce framework.
- The same set of files can be used as input for various different jobs, including monitoring jobs that calculate metrics and evaluate whether a job's output has the expected characteristics.
- MapReduce jobs separate logic from wiring, which provides a separation of concerns and enables potential reuse of code: one team can focus on implementing a job that does one thing well, while other teams can decide where and when to run the job.

3. **Choose one workflow scheduler that is mentioned in the reading and provide a description of the tool. (1 pt)**

Several execution engines for distributed batch computations have been developed. They have been designed to handle an entire workflow as one job, rather than breaking it up into independent subjobs. Since they explicitly model the flow of data into several processing stages, these systems are known as dataflow engines. Like MapReduce, they work by repeatedly calling a user-defined function to process one record at a time on a single thread. They parallelize work by partitioning units and they copy the output of one function over the network to become the input to another function.

4. **What are some challenges MapReduce has with Graph-Like Data Models and what are some solutions? (2 Pts)**

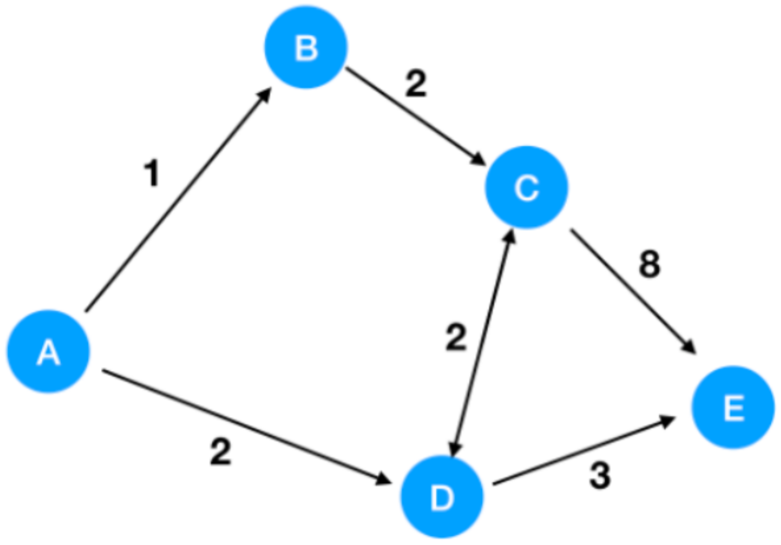
Many graph algorithms are expressed by traversing one edge at a time, joining one vertex with an adjacent vertex in order to propagate some information, and repeating until some condition is met. It is possible to a store a graph in a distributed filesystem, but the idea of "repeating until done" cannot be expressed in plain MapReduce, since it only performs a single pass over the data. This algorithm is often implemented in an iterative style. However, implementing it with MapReduce is often inefficient, because MapReduce does not account for the iterative nature of the algorithm: it will always read the entire input dataset and produce a completely new output dataset, even if only a small part of the graph has changed compared to the last iteration.

Below are some solutions for batch processing graphs:

- *The Pregel Processing Model:* One vertex can "send a message" to another vertex, and typically those messages are sent along the edges in a graph. In each iteration, a function is called for each vertex, passing it all the messages that were sent to it. In the Pregel model, a vertex remembers its state in memory from one iteration to the next, so the function only needs to process new incoming messages.

- *Fault Tolerance:* Achieved by periodically checkpointing the state of all vertices at the end of an iteration. If a node fails and its in-memory state is lost, the simplest solution is to roll back the entire graph computation to the last checkpoint and restart the computation.

5. Using the blow graph, find the optimal path from A to E using Dijkstra's Algorithm by describing each step. (3 Pt)



Dijkstra's algorithm is done in four steps:

1. Find the "cheapest" node. This is the node you can get to in the least amount of time.
2. Update the costs of the neighbors of this node.
3. Repeat until you have done this for every node in the graph.
4. Calculate the final path.

To get from A to E, we first have to find the shortest path from A to the next node. A to B takes 1 while A to D takes 2. However, if we look further down the graph, going from A to B will result in a path of length 11 or length 8. However, A to D will take length 5 or 12. Therefore, the first path will be from A to D. At point D, D to C takes only length 2. However, there will be another step of length 8 if we go that way. D to E, meanwhile, is just one stop of length E, so we choose D to E. The optimal path from A to E is A->D->E and takes length 5.