1. Describe the MapReduce paradigm in terms of Mappers and Reducers, and illustrate with an example (2 pt)

When MapReduce is in use, chances are the data is large enough that it can’t be processed by a single machine. My response hinges on this multi-node assumption. The mapper function is given to each active node, and the data on that node is sorted by a given key. The data is then redistributed (“shuffled”) among the nodes to replicated reducer functions, which process the data grouped by key, returning an output. An example here would be an incomplete deck of playing cards. To find out how many cards of each value are missing, the cards are first sorted (Mapped) by card value (Ace, 2, 3, 4…), and then are shuffled to different nodes. They are then counted (by the Reducer) and subtracted from 4, which gives us the number of missing cards for each value.

1. Describe some ways (at least 3) in which batch jobs achieve good performance while being easy to maintain (Philosophy of batch process outputs) (2 pt)

Batch jobs have an all-or-nothing output, also overwriting prior outputs. If a deployment is buggy, the job can be re-run with reverted code to rewrite the data without aftereffects in production. Kleppmann refers to this as *human fault tolerance*.

Secondly, batch jobs can be set to re-run upon failure, meaning that machine or network failures are more tolerable than in non-batch job systems. This is the case because the inputs are not altered – the outputs are only derived from them. If the inputs themselves were changed by the job, this automatic retry would not be permissible.

Thirdly, batch jobs allow for accelerated feature development for several reasons – one being related to human fault tolerance, that the code can be reverted to the last known “good” checkpoint and the job re-run, and a second being that the same input, because it is immutable, can be used for different jobs. This can act as a safety check on expected outputs, making for good performance along the workflow (good performance being defined as not necessarily *quick* in this case, but *proper*).

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

Apache Airflow is a workflow management tool that relies on directed acyclic graphs, where the edges connecting the nodes are operators. Airflow is a Python-based scheduler, though can run bash, SQL, and Email operations (among others). The scheduler allows for an interval and an execution date, with a DAGRun being a unit of tasks that are to be completed sequentially (though there is an allowance for parallel executions that may be specified).

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

A chief issue that MapReduce has with graph-like data models is that MapReduce lends itself explicitly to batch processing, where graphs do not. When traversing a graph, there is often a base case that needs to be met for the traversal to stop. This means that a new MapReduce call must be made for each level of traversal, with the Reducer function being a check on whether that condition has been met. While this workaround is requisite, it means that MapReduce is called iteratively and performance will suffer as a result.

There are other tools that may make easier work of large graphs, like Apache Giraph or Spark’s GraphX, which employ bulk synchronous parallel processing (Google’s *Pregel* model). These create a stateful environment where the only processing required from one iteration to the next are new “messages” sent along edges. A possible pitfall here is that if the graph is too large to fit onto one machine, that network latency may affect performance.

1. Using the below graph, find the optimal path from A to E using Dijkstra's Algorithm by **describing each step**. (3 pt)

Dijkstra’s algorithm, starting at A, will progress as follows:

1. Traverse A 🡪 B. Add 1 to cost, designate A as parent of B.
2. Traverse B 🡪 C. Add 2 to cost, designate B as parent of C.
3. Traverse A 🡪 D. Add 2 to new cost, designate A as parent of D.
   1. We start over with A 🡪 D because the cost to get from A to D is now less than the cost of A 🡪 B 🡪 C.
4. Traverse D 🡪 E. Add 3 to new cost, designate D as parent of E.
   1. The end node has been found, and getting from C 🡪 E is more expensive than D 🡪 E, so the algorithm terminates.

