There are 4 questions. Answer **all​** parts of **all**​ 4 questions. Max score = **35 points**

1. **HDFS + MapReduce: 2 + 4 + 4 points**
2. (1) What is the role of the NameNode in HDFS? (2) Block locations are not part of the HDFS namenode checkpoint. Given this, how is the location map recovered upon NameNode failure?
3. Describe what functionalities would the Map and Reduce tasks perform for (1) a job that takes as input list of (book name, author name) k-v pairs and returns the pairs sorted by the author name. (2) Consider an input list of documents as (unique document id, list of words in document). Write a map-reduce program (pseudo-code) that counts the occurrence of each unique word across all documents. The expected output is (word, total number of occurrences summed across all documents).

1. Briefly describe how does the MapReduce framework handle the following failure scenarios (Be sure to mention your assumptions, if you make any) -
   1. Completed Map task failure
   2. In-progress Map task failure
   3. In-progress Reduce task failure
   4. Completed Reduce task failure
2. **Spark: 3 + 2 + 4 points**

Iterative algorithms were one of the motivations behind Spark. For example, an iterative PageRank algorithm can be implemented atop Spark using the following code snippet -

val links = spark.textFile(...).map(...).persist()

var ranks = // RDD of (URL, rank) pairs

for (i <- 1 to 5)

{

val contribs = links.join(ranks).flatMap {

(url, (links, rank)) => links.map(dest => (dest, rank/links.size)) }

ranks = contribs.reduceByKey((x,y) => x+y).mapValues(sum => a/N + (1-a)\*sum)

}

1. Spark introduces the notion of a RDD. For the application above, how many RDDs are generated? Why? Give any one benefit of RDDs.
2. Describe the default fault recovery mechanism adopted by Spark. In the context of the application above, what additional measures (and on which RDDs) need to be taken to reduce the fault recovery time? Give a potential drawback of the additional measures you outlined.

1. Briefly outline a way in which you can improve the overall execution speed of the above application (hint: think co-partitioning tables). Explain how it helps relative to a naive alternative.
2. **DRF and YARN: 3 + 2 + 2 points**
   1. Consider a system with 10 CPUs, 30 GB RAM, and three users, where user A runs tasks with demand vector <2 CPU, 3 GB>, user B runs tasks with demand vector <1 CPU, 6 GB> and user C runs tasks with demand vector <2 CPU, 1 GB>. Here tasks are scheduled using DRF. How many tasks will be allocated by DRF for each user? How many resources will be unused in the system? (You do not have to write the iterative steps involved in the DRF algorithm. You just need to write the output i.e. no. of tasks)

* 1. Would DRF be an ideal strategy for applications with inter-task dependencies? (e.g. a reduce task can be scheduled only after its map task). Support your answer using an example.
  2. Name one advantage and one disadvantage of the YARN architecture compared to using a per job independent scheduler.

1. **Spark Streaming and Storm: 3 + 1 + 1 + 4 points**

Streaming systems broadly adhere to two models (a) mini-batched model and (b) record-at-a-time (i.e. continuous operator) model.

1. Name two performance bottlenecks in Storm. Explain why Storm cannot support exactly-once processing.
2. Define a discretized-stream (D-stream).
3. One way Spark streaming can handle delayed records (or out-of-order records) is to use a “wait time”. Outline one disadvantage of this approach.
4. Which model from (a) or (b) above do the systems -- Spark Streaming and Storm -- adhere to? Which system is naturally better in the following aspects and why:
   1. Latency (i.e. Execution Latency - time from ingesting a record to getting a result)
   2. Recovery from Failures / Straggler Mitigation
   3. Network Usage

**ANSWERS**

1. HDFS + MadReduce

A)

1) Metadata, namespace tree, mapping of file blocks to DataNodes

2) Gets block reports from DataNodes

B)

1) Map to (author\_name, book\_name), reducer sorts

2) Mapper: emit (word,1) for each word in the list. Reducer: reduce to (word,sum)

C)

1) Re-executed because output is stored on local disk of failed machine

2) Reset to idle and eligible to reschedule

3) Reset to idle and eligible to reschedule

4) Don’t need to be re-executed since output is stored in global file system

**2. Spark**

12 RDDs (1 links + 6 ranks + 5 contribs). Due to the immutable nature of RDDs. Enables easier straggler mitigation by running backup tasks (would be harder to achieve if mutable).

Uses lineage approach. Checkpointing of the ranks RDD is necessary since the lineage can grow. Checkpointing incurs storage overhead.



Use custom partitioning and partition links and ranks in the same way. Leads to narrow dependency and thus reduces shuffle.

**3. DRF and YARN**

A)

Ans: User A - 2, User B - 2, User C - 2. Unused CPU = 0, Unused Memory = 10

B) Ans: No, it won’t be ideal. (Something similar to the motivation of graphene)

C)

Advantages: Is optimal on global metrics like avg. JCT, fairness, makespan.

Disadvantages: Does not scale with the number of jobs.

**4. Spark Streaming and Storm**

VI. Spark Streaming:

A) Performance Bottlenecks:

* One slow tasks slows down other tasks on executor since slows down queue.

Shared use of queue by multiple topologies. Any queue build up would cause all topologies to run at speed of slowest topology in cluster. (This is the main point)

* Overheads to support atleast once semantics (Sec 3.3.2 of the paper)
* Overheads due to externalising heartbeat state to an external storage daemon

Cannot support exactly once:

Metadata node tracks if input tuple has been processed by all relevant nodes. But if processed by some and then restarted, it may get processed multiple times.

Another reasoning is as follows - for exactly once semantics -- we essentially want to guarantee that the state updates due to an incoming message happen only once. This is tricky to do in storm as in storm we replay on failure (or when an ACK is not received). When an ACK is not received, it is non-trivial to know whether the ACK is lost or the message. Trident solves it by introducing transactions, Spark Streaming solves it by introducing mini-batches.

B) Realizing a stream processing job as a series of time-ordered batch jobs / structuring computations as a set of short, stateless, deterministic tasks instead of continuous, stateful operators

C) It does not provide provide a guarantee of processing out-of-order record. Reason being, a record can always show up after the configured “wait time”.

D) Spark Streaming is mini-batched, Storm is record-at-time.

* 1. Storm has better latency - because the latency in Storm is of the order of time to ingest and process a single record whereas in Spark Streaming it is of the order of ingesting and processing a mini-batch of records.
  2. Spark Streaming has better recovery from failure and better Straggler Mitigation - because in Spark Streaming recovery of failed node(s) state is naturally parallelizable in both the different partitions of the operator and time
  3. Spark Streaming has lesser network usage - because aggregation of output can happen naturally in a mini-batched model