CSCI/ECEN 5673: Distributed Systems
Spring 2022
Homework 3
Due Date: 04/21/2021

Please submit a PDF copy of your answers via the submission link on Canvas by April 21, 2022. Write your answers in the space provided. Please DO NOT use any extra space. The space provided is sufficient for answering the questions.

Honor Code Pledge: On my honor, as a University of Colorado at Boulder student, I have neither given nor received unauthorized assistance on this work.

Name: M Sai Akhil Teja

1. What are Resilient Distributed Datasets? Explain the fault recovery method used and the kinds of operations in RDDs?

Resilient Distributed Dataset (RDD) is the fundamental data structure of Spark. They are immutable distributed collections of objects of any type. As the name suggests, it is a resilient(fault-tolerant) records of data that resides on multiple nodes. Each dataset in RDD is divided into logical partitions across the cluster and thus can be operated in parallel, on different nodes of the cluster. These RDDs can be created by deterministic operations on data on stable storage or other RDDs by either an existing Scala collection or with an external file in the HDFS (or any other supported file system).

RDDs support two types of operations: *transformations*, which create a new dataset from an existing one, and *actions*, which return a value to the driver program after running a computation on the dataset. For example, map is a transformation and reduce is an action. RDDs have transformations like map, filter, union, join, cache when defining a new RDD and Parallel operations like reduce, collect, count, save when returning a result to driver. RDDs make iterative operations and running multiple adhoc queries on same set of data efficient.

The RDDs are fault tolerant as they track data lineage information to allow for rebuilding lost data automatically on failure. To achieve fault tolerance for the generated RDD's, the archived data is replicated among various Spark executors in worker nodes in the cluster.
Ref: https://spark.apache.org/docs/latest/rdd-programming-guide.html
https://www.xenonstack.com/blog/rdd-in-spark/

2. What are the workload and technology characteristics GFS assumed in its design and what are their corresponding design choices? What design choices GFS adopts to keep the master from being a bottleneck?

Ref: https://static.googleusercontent.com/media/research.google.com/en//archive/gfs-sosp2003.pdf

Assumptions for workload and technology characteristics in GFS design:

- Frequent failures as a commodity failure:
    - Chunk server is replicated thrice.
    - Master maintains Operation logs which can be replayed to restore the file system.
    - Master keeps sending heartbeat. Which it uses to restore the master immediately even if the chunk server is down.
- Dealing with files of large size. Mostly deals with GBs of data rather than KBs and often in TBs.
    - Large chunk size of 64MB is used for storage.
    - Easy to scale Commodity hardware is used.
- Random writes are less frequent that appends.

    - Appends are faster due to a large chunk size of 64MB
    - *Relaxed consistency*: GFS guarantees the data is appended at least once, at a location of GFS' choosing.
    - Appending is atomic as one continuous sequence of bytes.
    - Multiple clients an append simultaneously.
- Most files read sequentially.
    - Large chunk size of 64MB makes sequential read faster.
- Throughput more important than Latency.
    - Any number of clients can concurrently append to same file.
    - GFS will provide Atomic append and relaxed consistency.
    - Clients send data to primary, and it sends to remaining replicas.
    - This reduces network calls.

Master is not a bottleneck:

- Client gets chunk server details from master and then contacts the chunk server directly.
- Master is not in the way of actual data flow. So, Master is not a bottleneck in the data transfer.
- Master maintains only Meta data and not actual data. It is stored in memory for fast access. Hence, Master is not a bottleneck.

3. For each of the following problems describe how you would solve it using map-reduce. You should explain how the input is mapped into (key, value) pairs by the map stage, i.e., specify what is the key and what is the associated value in each pair, and, if needed, how the key(s) and value(s) are computed. Then you should explain how the (key, value) pairs produced by the map stage are processed by the reduce stage to get the final answer(s). If the job cannot be done in a single map-reduce pass, describe how it would be structured into two or more map-reduce jobs with the output of the first job becoming input to the next one(s).

   (a) The input is a list of housing data where each input record contains information about a single house: (address, city, state, zip, value). The output should be the average house value in each zip code.

    - The map function outputs ***<zip, count, average>*** pairs for each record.
    - Our mapper will output two "columns" of data, count and average.
    - For each input record, this will simply be "1" and the value of the field.
    - The reducer will multiply the "count" field by the "average" field to add to a running sum, and add the "count" field to a running count.
    - It will then divide the running sum with the running count and output the count with the calculated average as ***<zip, count, average>.***

(b) The input contains two lists. One list gives voter information for every registered voter: (voter-id, name, age, zip). The other list gives disease information: (zip, age, disease). For each unique pair of age and zip values, the output should give a list of names and a list of diseases for people in that zip code with that age. If a particular age/zip pair appears in one input list but not the other, then that age/zip pair can appear in the output with an empty list of names or diseases, or you can omit it from the output entirely, depending on which is easier.
(Hint: the keys in a map/reduce step do not need to be single atomic values.)

- Key for the mapper will be combination of zip and age.
- Value will be combination of name and disease.
- When mapping an object of first list it will leave empty the field of disease and when mapping an object of second list it will leave empty name in value.
- Mapper will produce <<zip, age>, name, disease>.
- Reduce function will merge based on the key and produce list of names by merging name columns in the value and list of diseases by merging disease columns in the value.
- Output of reduce function would be <<zip, age>, <list<name>, list<disease>>> where the value is list of all the names and list of all diseases.

4. What Consider a DHT using ring mapping such as Chord

a) Explain how a key is mapped to a node.

- Chord is a DHT protocol.
- SHA1 hashing is used to assign nodes and keys to a n bit identifier.
- As a result, the ring can only have 2m nodes.
- Now the key is hashed to an m bit identifier, which corresponds to a ring location.
- It checks whether a node is present in the location, and if it is, it is mapped to the key.
- Otherwise, it searches for the node in a clockwise direction in the ring and assigns to the successor.
- So, the key k is assigned to 1st node whose identifier is >=k.

b) In the most basic form, nodes simply track their predecessor and successor. Very briefly state what problem this leads to as the network grows larger.

- The number of nodes in the ring increases as the network grows.

- These nodes will only contain information about their successor and predecessor.
- The queries for a given identifier are passed around in the ring until the successful identifier is discovered.
- So, if there are N nodes, it might be necessary to traverse all N nodes in order to obtain the identifier in the worst-case scenario.
- This becomes a problem when N is very large.

c) The Chord scheme overcomes this problem by having each node maintain a "finger table". Explain how this scheme works.

- When each node keeps a finger table, the entries in this table include the successors of node n.
- The ith entry in the table at node n contains the ID of the first node s on the ring that follows n by 2i-1 space.
- So now a node n can query the node j in its table closest to k for the successor to any key k.
- To obtain the identifier, O(log N) nodes must be contacted.

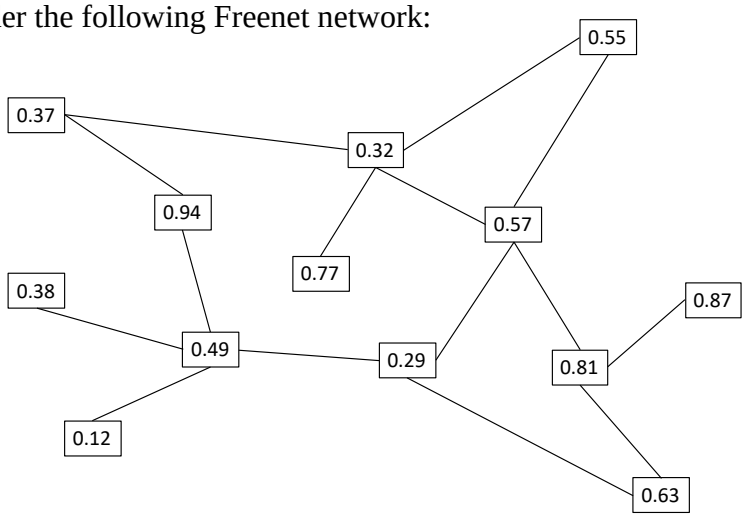
d) In a Chord structured DHT with N nodes, how many hops would a lookup operation require?

- When a Chord structured DHT with N nodes maintains "finger table" it requires log(N) hops for a lookup operation.

e) Assume a Chord network in which node q is the successor of node p. During operation, node p discovers that its successor link is no longer consistent because q has updated its predecessor link. What does this change imply must have happened in the network?

- If node q is the successor of node p, then node q's predecessor is p.
- When node p notices that node q has modified its predecessor connection, it means a new node was added to the ring between nodes p and q.
- If and only if p precedes n by at least 2i-1 and the ith finger of node p succeeds n, node n will become the ith finger of node p.
- As a result, the successor of node p has been modified to ensure consistency.

5. Consider the following Freenet network:



a) a) Suppose <0.62, 100> is stored at node 0.57. A client requests node 0.37 to retrieve the value associated with 0.62. What sequence of nodes will this get request be routed through in the network?
(0.37) -> (0.32) -> (0.57) because 0.57 is closer than 0.62 than 0.55

b) What sequence of nodes will the response be routed through?

(0.57) -> (0.32) -> (0.37)

c) Next, another client requests node 0.38 to retrieve the value associated with 0.62. What sequence of nodes will this *get* request be routed through in the network?

(0.38) -> (0.49) -> (0.94) -> (0.37)

d) A client requests node 0.55 to store <0.1, 300>. What sequence of nodes will this put request be routed through and where will <0.1, 300> be stored? Assume that <0.1, 300> is not currently stored in the network.
Assumption: TTL of 18
Request route: (0.55) -> (0.32) -> (0.37) -> (0.94) -> (0.49) -> (0.12) -> (0.49) -> (0.29) -> (0.63) -> (0.81) -> (0.87) -> (0.81) -> (0.57) -> (0.29) -> (0.57) -> (0.32) -> (0.57) -> (0.55) -> (0.57)
Final Path: (0.55) -> (0.32) -> (0.37) -> (0.94) -> (0.49) -> (0.12). coz, 0.12 has cache value.

e) Is it possible to store conflicting <key, value> pairs (pairs that have the same key but different values) in Freenet where no single node is aware of the conflict? Explain your answer.

- Yes, conflicting pairs may be stored in Freenet if no single node is aware of the dispute.
- If node 1 attempts to insert <data, val1> the key is found in two hops from node 1, and the <key val1> is saved in all nodes along this path p1.
- If another client requests to put <key, val2> to node 2 at the other end of the network, and the put request route reaches max TTL without passing through any of the nodes in path 1, the network will save the new value in the new path 2 nodes, but the old value will be saved in the nodes on path 1 (val1).