Project Guideline

DSCI 551 – Spring 2023

The theme of this semester's project is **emulation** as mentioned in class. **The goal is to develop a prototype system that emulates the interface and working of a big data system.** Here big data system refers to a system that manages a large amount of data. This includes big data software and NoSQL databases we have seen or will see in class: Hadoop, Spark, Firebase, MongoDB.

Note: there is no hard requirement on the size of data set. However, we expect you to show the emulated system is capable of handling KBs and MBs of data.

Both two-person and three-person teams should meet the stated requirements (i.e., getting the job done), with three-person teams expected to have a higher quality implementation and UI/app if required (i.e., doing a better job).

Recommended Projects:

1. Emulating Firebase, e.g., by using Flask, WebSockets, and MongoDB.
   Requirements on your prototype system (database server):
   - It should have RESTful API which supports functions in Firebase RESTful API, which include: PUT, GET, POST, PATCH, DELETE, and filtering functions: orderBy="$key"/"$value"/"name", limitToFirst/Last, equalTo, startAt/endAt.
   - You should store JSON data in another database, e.g., MongoDB.
   - It should have proper index created in database (e.g., MongoDB) to support orderBy. For example, for orderBy="name" on users.json, it should create an index on name.
   - It should have a command-line interface that allows users to query/update content of database using curl command (similar to that in Firebase), for example:
     - curl -X GET 'http://localhost:5000/users.json?orderBy="name"&limitToFirst=5'
     - curl -X PUT 'http://localhost:5000/users/200.json' -d '{"name": "john", "age": 25}'
     - …
     - Note: the command should return data/response in JSON format (again similar to that in Firebase).
     - Note also that this means that you are required to implement a Restful server so that it can accept and process request sent from curl commands. Refer to sample code in Resources on how to use Flask to build a server that listens at a given port and intercepts all PUT and GET responses. Obviously, this is just a sample code and you will need to also implement support of other HTTP methods: POST, PATCH, and DELETE.
   - **(Multi-person team)** Develop a Web app that demonstrate the real-time update and syncing of the data with the server. Refer to in-class demo for an example on real-time syncing. An example is a chat app where a user John may send a message to another user David or broadcast it to all users (as in a group chat session). Your firebase realtime database in this case can store the new message sent by John and notifies David or all

users in the group of the new message. Note: there is no need for users to enter curl commands in web browser. The CRUD functions should be embedded into the app as chat app example above shows.

Useful resources:

- A Flask-based server example that can be used as a starting point to implement RESTful server.
- A Websocket server example based on Python Stream library.
- A Web browser client that shows communication with server via WebSocket.

2. Emulating HDFS (EDFS)

Requirements on your prototype system (EDFS):

- It should have a metadata server (similar to NameNode) and a number of data servers (similar to DataNode).
- EDFS should split the file into blocks if the file exceeds a predefined size, and store blocks with different data servers.
- It should implement a DFS client that can accepts shell commands as in HDFS, including: ls, rm, put, get, mkdir, rmdir, cat.
  For example,
  - edfs -ls /user/john
  - edfs -mkdir /user/john
  - …
- **(Multi-person team)** Implement a Web browser based UI that allows users to upload and download files from EDFS, explore the directory structure of EDFS, and the content of each partition/block of a file on EDFS.

Useful resources:

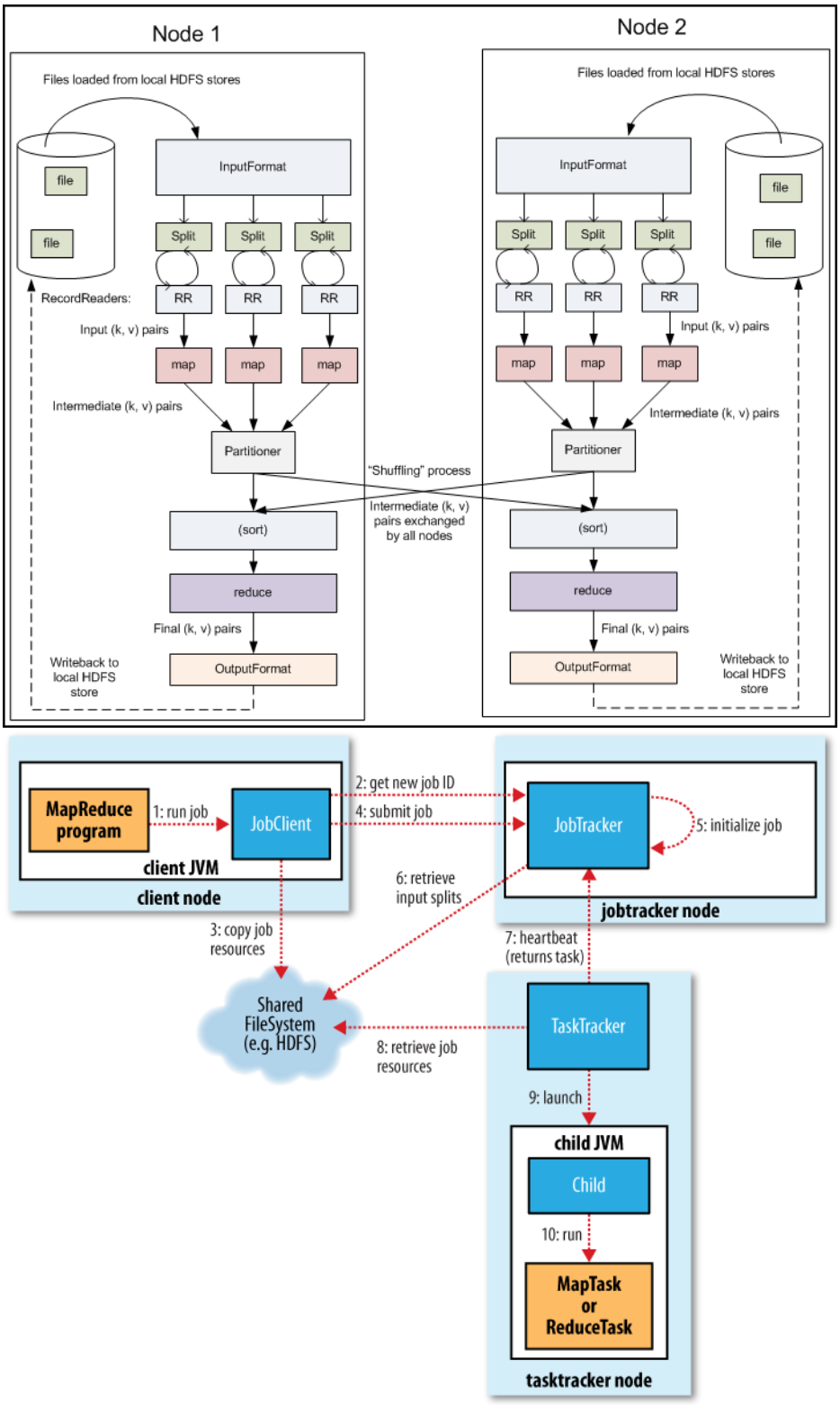
- A socket server using Python' stream library. The server emulates namenode (it will return size of file1; otherwise, report file not found).
- A socket client that sends shell commands to the server

3. Emulating Hadoop MapReduce

Requirements on your prototype framework:

- Implement basic mapper and reducer classes and allow users to extend the classes to provide their own map and reduce function (similar to that in WordCount.java).
- Map function should output key-value pairs and reduce function should take a key and a list of values as input like in Hadoop.
- Implement partitioner, shuffler, and data grouper which take intermediate key-values produced by map tasks and produces input for reduce tasks.
- **(Multi-person team)** Create a number of data servers as in EDFS and distribute map and reduce tasks among the server (note data servers perform computation too). Implement JobTracker-TaskTracker like function for task scheduling.
- **Notes:**

- Data partitioner partitions the output from map tasks so that key-values for different reduce tasks are stored in different files. For example, assume that there are two reduce tasks: r1 and r2, a map task m1 may produce output files: m1_r1.out, m1_r2.out.
- Data shuffler will send the right files to the right reduce tasks. For example, assume that there are two map tasks. Then the reduce task r1 will need to fetch files m1_r1.out (produced by map task m1) and m2_r1.out (produced by map task m2). The files for a reduce task from different map tasks may be merged into a single file, e.g., r1.out (by merging m1_r1.out and m2_r1.out).
- Data grouper will sort the merged file, e.g., r1.out to find groups and values in the same group. You need to use external sorting for this. In other words, you need to assume that the data to be sorted might be larger than the memory available. For each group, it should call user-supplied reduce function (e.g., see reduce function in the attached sample code: WordCount.java).
- You may also follow the description Chapter 6 of "Hadoop: The Definitive Guide" (see link in resource below) by sorting the files produced by map tasks and using a single file to store data for different reduce tasks with proper partition index (see Figure 6-4 of the above book).
- It is important to show that users can build map reduce jobs using the emulated framework and run the job to get the results.
- Need to show multiple map tasks may be generated, one for each input split (partition of the input file), and multiple reduce tasks may be requested.
- Also show how reduce tasks obtain its input key and list of values from the output of mask tasks (through the partitioner, shuffler, and grouper provided by the framework).
- Multi-person teams are expected to show the tasks are indeed computed in multiple servers (although these servers may all run on the same machine).

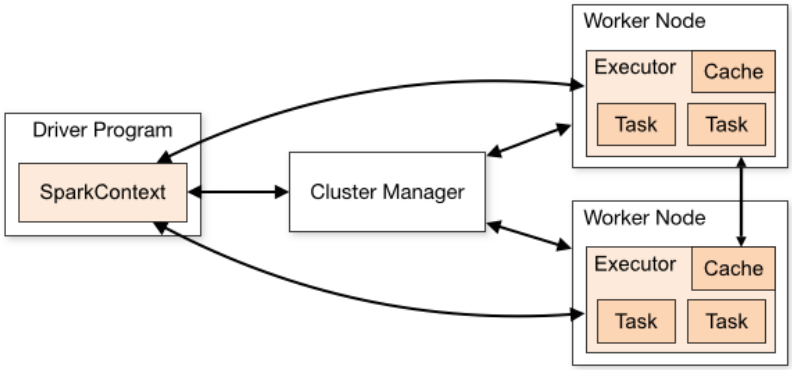


Resources:

- WordCount.java
- https://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html

- Also see the book “Hadoop: The Definitive Guide” in the syllabus. USC library link (free). Note: this book is based on old version of Hadoop.
    - https://uosc.primo.exlibrisgroup.com/discovery/fulldisplay?docid=alma991043008979103731&context=PC&vid=01USC_INST:01USC&lang=en&search_scope=MyInst_and_CI&adaptor=Primo%20Central&tab=Everything&mode=Basic
- Refer to socket stream library as in EDFS project.

4. Emulating Spark RDD API, e.g., in Python:
    - Develop a class SC that emulates SparkContext (see diagram below). The class should allow users to create an RDD from a file or a Python list, specify the number of partitions, partition the dataset, and support major operations such as map, filter, and reduce on RDD.
    - Develop a class RDD which should keep track of the partitions in the RDD including its locations (e.g., in which worker nodes). It should also implement an iterator-like interface for the RDD to allow pipelining of data from one RDD to another.
    - Implement a scheduler for SC which will be responsible for sending tasks to worker nodes, coordinating with executors, and fetching task results. Data and codes for the tasks may be sent over the network to worker node by using serialization library, e.g., pickle or dill.
    - **(Multi-person team)** Implement a Web app that monitors the job submission and progress. Similar to that in Spark: https://spark.apache.org/docs/latest/web-ui.html



Resources:

- Python socket stream library as in EDFS project for implementing servers that run driver program and executor.

5. Emulating other big data systems of your choice.
You can also choose another big data system to emulate. However, we might not be able to provide as much support as the recommended projects. Also you will need to make sure you are emulating a big data system. Acceptable examples are: Cassandra, Hive, AWS TimeStream, AWS DynanoDB, AWS Neptune. Redis is acceptable too, but you will need to deploy your prototype system in multiple machines (e.g., multiple EC2 instances) to make a case for it to handle big data since Redis stores data in RAM.

Resource:

- Sample code for data and code serialization.
- Refer to socket stream library as in EDFS project, to build servers for worker node, etc.

Note for multi-person teams, you will be required to implement a Web UI as in other projects.

Note there are links in the provided sample code where you can read more on documentation of libraries used.

Team formation: You can form a team of up to 3 people. If there are multiple persons in your team, you are also required to implement the components above that are marked with **(Multi-person team).** All team members should come from the sections that meet at the same time. In other words, you are not allowed to team up with someone in Tuesday section if you are in MW section.

Deliverables, points, and deadlines (all deadlines are 11:59pm on the due date):

1. Proposal (5 points, due on **2/10, Friday**)
   Team member, project topic, planned implementation, timeline (when to finish which items), tasks for each member.
2. Midterm progress report (5 points, due on **3/27, Monday**)
   Report progress made and any update on your plan.
3. Final report (10 points, due on **4/28, Friday**)
   Title, topic, implementation, and learning experiences.
   As part of your final report, we will be asking you to fill out a self-assessment on your project learning outcomes. Details will be announced near the end of semester.
4. Implementation (70 points, provide a link to your source codes, e.g., on Google drive, in the final report). Due as the same time as your final report.
   There will be points assigned to each aspect of the project as detailed in the requirements stated above. Missing or incomplete work will be subject to deduction of points.
5. Demo (5 points, in class in the week of 4/24). All members of project team should be present and presenting part of the project. 5 points will be deducted from a member if he/she is absent.
6. Video (5 points, about 20 minutes, due on **4/28, Friday**): in-depth presentation of technical aspects of your project. A spreadsheet will be provided for you to fill out the details including link to your video. All members of team should be present, with **camera on** during the presentation.

Important notes: **No LATE submissions will be accepted**. Note that projects are semester-long assignments. All assignments including projects should NOT be done in the last minute! The last homework of 551 may be due in the same week and some students might have final exams in the last lecture week. These will not be considered as conflicts. You should only spend the last week of semester to finish up your project deliverables, e.g., summarizing your learning experiences. All implementations and write up should be done before the last week when you demo your projects.

Also note that the project is meant to be a teamwork and all members of the team will receive the same grade. Choose your project partners wisely, e.g., having students with CS backgrounds to work with non-CS students would be a good idea.