L09b. Map-Reduce Framework

Big Data Applications:

- Big Data Applications work over large data sets, so they take a long time to process and they use vast computational resources.
- Big Data computations are Embarrassingly Parallel Computations, they are independent computations that don't require synchronization or coordination among each other, hence can be run in parallel.
- The Map-Reduce framework is a programming framework that handles the following:
  - Parallelizing big data applications to run on thousands of nodes.
  - Building a pipeline of tasks and handling data distribution.
  - Handling scheduling of tasks and monitoring them.
  - Handling failed nodes.

Map-Reduce Description:

- The following inputs are introduced to Map-Reduce by the application developer:
  - Set of records as <key, value> pairs.
  - Two functions: map and $reduce$.
- Example: Finding the number of occurrences of names in a set of documents:
  - The input to the map function would be a <key, value> pair, where the key is the file name and the value is the content of the file.
  - The map function will search the contents of each input file for a specific name and emit a <key, value> pair, where the key is that specific name and the value is the number of occurrences of this name in that input file.
  - We can spawn as many instances of the map function as the number of input documents.
  - The output of the map functions will be used as an input the $reduce$ functions.
  - Each $reduce$ function will be responsible for handling a specific key. So, the number of $reduce$ functions would be equal to the number of unique keys.
  - The $reduce$ function will aggregate the number of occurrences of each key from all the map outputs to produce the final <key, value> pairs.
  - map and $reduce$ functions can be executing on different nodes of the cluster.
- The Map-Reduce Programming Framework will automatically handle the following:
  - Instantiating the map and $reduce$ functions and handling the data pipeline between them.
  - Coordinating data movement between $map/reduce$ functions across thousands of nodes.
  - Handling scheduling of $map/reduce$.
- Why Map-Reduce? Several processing steps in GSS are expressible as $map/reduce$ (e.g. airlines search, word indexes, page ranking).

Map-Reduce Heavy Lifting:

- Spawn one Master thread (to monitor and control all worker threads) and multiple worker threads for the desired task.
- Auto-split the input files, based on an automatic or a user-specified value M, into splits of <key, value> pairs. Each split will be the input of a $Mapper$ thread.
- Each $Mapper$ thread will read the input files from the disk, parse the input, and call the user-defined map function that will do the needed.
- Then the $Mapper$ will create R (user-specified) intermediate files containing the results of the map phase. These files will be saved on the local disk of the map thread.
- The Master thread waits for all the map threads to finish, then starts R $Reducers$.
- Each $Reducers$ will perform a remote read (RPC) to fetch the intermediate files from the map local disk.
- Then the $Reducers$ will sort these data and call the user-defined $reduce$ function to aggregate the intermediate data and produce the final result.
- Once R final output files have been produced, the Map-Reduce task is considered to be completed.
- All this heavy-lifting is done transparently and automatically by the Map-Reduce runtime without any involvement from the user, who only provides the input files and the implementation of the desired $map/reduce$ functions.

Issues to be Handled by Map-Reduce:

- The Master data structure includes:
  - The location of files created by completed $Mappers$.
  - Scoreboard of $Mapper/Reducer$ assignment.
  - Fault tolerance (node down, network link down, non-homogenous machines):
    1. Start new instances on a different node if no timely response.
    2. If multiple completion messages are received from redundant stragglers, handle killing redundant threads.
  - Locality management: Make sure that the working set of computations fit in the closest level of the memory hierarchy of a process, so that the computation can make good forward progress and complete efficiently.
  - Task granularity: Come up with correct task granularity to have good load balancing and utilization of computational resources.
    → The user can override the default partitioning hash function with a user-defined one in order to organize the input data better.
    → The user can incorporate combining functions to be included in the $map/reduce$ functions.
  - Backup tasks.