**Overview**

* Built-in means for distributing processing over the cluster
* Divide data in partitions
* Mapping
  + Transforming data
  + Stream of data input is transformed
* Reducing
  + Associate data with a key value
  + Aggregates your data
* Resilient to failure
  + Application master monitors mappers and reducers on each partition

**Example**

* Movie data
* Mapper converts raw data into key/value pairs
  + Take
  + Extracting rate count from ratings data
  + Extract and organize target information
  + “Shuffle and Sort”
    - Aggregates values associated with each key (merges rated movies)
    - Sorts result by key
      * Is this done by using a heap?
      * Alternative could be aggregate, then use counting sort
        + Counting sort might become less efficient if range is big
* Reducer process each key’s values
  + Could be run in parallel
  + For each key reducer is called once
  + (EX) counts movie count for each key
    - turns the merged list of rated movie\_id into a count value
* Mapper – Reducers are flexible in setup
  + (EX) the mapper could have just return a single count (1) for each rating entry
    - The reducer would have just been able to sum of these counts
      * Probably more efficient in the movie context

**How Mapreduce Scales**

* Partition input data
  + Assign each node to process a partition (or partitions)
* Shuffle and Sort goes through a process which is essentially a Merge Sort
  + External mergesort?
  + Divide and Concur
* Reducer jobs can also be partitioned
  + Each reduce is tasked to aggregate a set of keys

**Flow**

* New Request
  1. Client Node -> YARN
     + YARN starts taking care of logicstics to spawn a MapReduce task
     + YARN tries to make sure the chunk of data needed for a certain node to run the Mapper/Reducer job is closest (physically), if not on the same machine, to the node
       - Optimize by reducing the need to send data over the network
  2. Client Node -> HDFS
     + Copies over data to the HDFS for nodes to access
  3. Yarn spawns a MapReduce Application Master
     + Runs under a NodeManager which keeps track of the node’s state
  4. MapReduce Application Master partitions and delegates jobs across a # of nodes
     + Multiple nodes could be run under a single NodeManager
     + Multiple NodeMangers run under the Application Master

**Mappers and Reducers**

* Natively written in Java
* STREAMING allows interfacing to other languages
  + Uses stdin and stdout to communicate between the Map/Reduce Task and a process written in a different language

**Hadling Failure**

* Application monitors tasks for errors or hanging
  + Restarts as needed (preferable on a different node)
* Application Master failure
  + YARN tries to restart it
* If an entire node fails
  + YARN tries to restart the processes tasked to those nodes (preferably on a different node)
* YARN failiure
  + Can setup “high availability” (HA) using Zookeeper to keep a hot standby

**Additional Optimization**

* Counters
  + Can keep shared counts across the cluster
    - How is this done?
    - Increment these shared counts instead of distributing mapping then reducing?
    - Wouldn’t this depend on network latency (most likely would have to send data across nodes)?
* Combiners
  + Reduction on mapper jobs