Map/Reduce Goals

– Distribution
  • The data is available where needed.
  • Application does not care how many computers are being used.

– Reliability
  • Application does not care that computers or networks may have temporary or permanent failures.
Application Perspective

• Define Mapper and Reducer classes and a “launching” program.

• Mapper
  – Is given a stream of key1,value1 pairs
  – Generates a stream of key2, value2 pairs

• Reducer
  – Is given a key2 and a stream of value2’s
  – Generates a stream of key3, value3 pairs

• Launching Program
  – Creates a JobConf to define a job.
  – Submits JobConf to JobTracker and waits for completion.
Application Dataflow
Input & Output Formats

• The application also chooses input and output formats, which define how the persistent data is read and written. These are interfaces and can be defined by the application.

• InputFormat
  – Splits the input to determine the input to each map task.
  – Defines a RecordReader that reads key, value pairs that are passed to the map task

• OutputFormat
  – Given the key, value pairs and a filename, writes the reduce task output to persistent store.
Output Ordering

• The application can control the sort order and partitions of the output via OutputKeyComparator and Partitioner.

• OutputKeyComparator
  – Defines how to compare serialized keys.
  – Defaults to WritableComparable, but should be defined for any application defined key types.
    • key1.compareTo(key2)

• Partitioner
  – Given a map output key and the number of reduces, chooses a reduce.
  – Defaults to HashPartitioner
    • key.hashCode % numReduces
Combiners

- Combiners are an optimization for jobs with reducers that can merge multiple values into a single value.
- Typically, the combiner is the same as the reducer and runs on the map outputs before it is transferred to the reducer’s machine.
- For example, WordCount’s mapper generates (word, count) and the combiner and reducer generate the sum for each word.
  - Input: “hi Owen bye Owen”
  - Map output: (“hi”, 1), (“Owen”, 1), (“bye”,1), (“Owen”,1)
  - Combiner output: (“Owen”, 2), (“bye”, 1), (“hi”, 1)
Process Communication

• Use a custom RPC implementation
  – Easy to change/extend
  – Defined as Java interfaces
  – Server objects implement the interface
  – Client proxy objects automatically created

• All messages originate at the client
  – Prevents cycles and therefore deadlocks

• Errors
  – Include timeouts and communication problems.
  – Are signaled to client via IOException.
  – Are NEVER signaled to the server.
Map/Reduce Processes

- **Launching Application**
  - User application code
  - Submits a specific kind of Map/Reduce job
- **JobTracker**
  - Handles all jobs
  - Makes all scheduling decisions
- **TaskTracker**
  - Manager for all tasks on a given node
- **Task**
  - Runs an individual map or reduce fragment for a given job
  - Forks from the TaskTracker
Process Diagram
Job Control Flow

- Application launcher creates and submits job.
- JobTracker initializes job, creates FileSplits, and adds tasks to queue.
- TaskTrackers ask for a new map or reduce task every 10 seconds or when the previous task finishes.
- As tasks run, the TaskTracker reports status to the JobTracker every 10 seconds.
- When job completes, the JobTracker tells the TaskTrackers to delete temporary files.
- Application launcher notices job completion and stops waiting.
Application Launcher

• Application code to create JobConf and set the parameters.
  – Mapper, Reducer classes
  – InputFormat and OutputFormat classes
  – Combiner class, if desired

• Writes JobConf and the application jar to DFS and submits job to JobTracker.

• Can exit immediately or wait for the job to complete or fail.
JobTracker

- Takes JobConf and creates an instance of the InputFormat. Calls the getSplits method to generate map inputs.

- Creates a JobInProgress object and a bunch of TaskInProgress “TIP” and Task objects.
  - **JobInProgress** is the status of the job.
  - **TaskInProgress** is the status of a fragment of work.
  - **Task** is an attempt to do a TIP.

- As TaskTrackers request work, they are given Tasks to execute.
TaskTracker

• All Tasks
  – Create the TaskRunner
  – Copy the job.jar and job.xml from DFS.
  – Localize the JobConf for this Task.
  – Call task.prepare() (details later)
  – Catch output from Task and log it at the info level.
  – Take Task status updates and send to JobTracker every 10 seconds.
  – If job is killed, kill the task.
  – If task dies or completes, tell the JobTracker.
TaskTracker for Reduces

• For Reduces, the task.prepare() fetches all of the relevant map outputs for this reduce.
• Files are fetched using http from the map’s TaskTracker’s Jetty.
• Files are fetched in parallel threads, but only 1 to each host.
• When fetches fail, a backoff scheme is used to keep from overloading TaskTrackers.
• Fetching accounts for the first 33% of the reduce’s progress.
Map Tasks

- Use the InputFormat object to create a RecordReader from the FileSplit.
- Loop through the keys and values in the FileSplit and feed each to the mapper.
- For no combiner, a SequenceFile is written for the keys to each reduce.
- With a combiner, the frameworks buffers 100,000 keys and values, sorts, combines, and writes them to SequenceFile’s for each reduce.
Reduce Tasks: Sort

• Sort
  – 33% to 66% of reduce’s progress
  – Base
    • Read 100 (io.sort.mb) meg of keys and values into memory.
    • Sort the memory
    • Write to disk
  – Merge
    • Read 10 (io.sort.factor) files and do a merge into 1 file.
    • Repeat as many times as required (2 levels for 100 files, 3 levels for 1000 files, etc.)
Reduce Tasks: Reduce

• Reduce
  – 66% to 100% of reduce’s progress
  – Use a SequenceFile.Reader to read sorted input and pass to reducer one key at a time along with the associated values.
  – Output keys and values are written to the OutputFormat object, which usually writes a file to DFS.
  – The output from the reduce is NOT resorted, so it is in the order and fragmentation of the map output keys.