MapReduce Online

by Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, Russell Sears

My Name

- Yuanzhi Li

The Goal

- A modified MapReduce architecture allows data to be pipelined between operators.

- Preserve the architecture of Hadoop

Pipeline?

- Mapper can push “incomplete” data to reducer.
- Reducer can generate an approximation of the final answer for incomplete data, also known as online aggregation

MapReduce

- Input: Jobs
- Map function for each job: map a job to a bunch of key-value pairs
- Reduce function: gather values for each key and process it.

Hadoop MapReduce

- Single Master Node: JobTracker: Accept jobs from client and divide them into tasks (a portion of the input file).

- Many worker nodes: TaskTracker

Map Execution

- Map phase: read the split and apply map
- Commit phase: after the map function complete a split, it registers the final output with the TaskTracker, which informs the JobTracker

Reduce Execution

- Shuffle phase: fetch the reduce task's input data produced by map (after a map has commit), using HTTP request.
- Sort phase: group records with same key
- Reduce phase: apply user defined reduce function

What is bad?

- The output of Map/Reduce must be written to disk before it can be consumed.

Pipelined MapReduce

- Pipeline between tasks
- Pipeline between jobs

Big Picture

- Map task has two phases: map, sort.

- Reduce task has three phases: shuffle, reduce, commit.

Pipeline inside a job

- Run each map function in a thread, store the output in a memory buffer (Map Phase)

- When the buffer exceed certain threshold, map function apply a combine operation of values for each key to create “spill file”.

- For each key, if the there is no reduce task for it, then write down the values to disk. If there is a reduce task, pipeline the spill file using TCP connection (Shuffle Phase).

- Reduce task can merge the “spill file” on going, once all map tasks complete, it will apply reduce function (Reduce Phase).

To make it a system

- Each reducer can only receive pipeline data from a bounded number of maps, for the rest it proceeds like traditional Hadoop — To reduce the number of TCP connections

- When reducer is too slow (the number of unsent spill file blows): map will try to merge different spill files (Sort Phase) — Adaptive load balancing.

Fault Tolerance: Map task failure

- Add bookkeeping to the reduce task to record which map task produced each pipelined spill file.

- Reduce task can merge spill files from same uncommitted map.

- Map task periodically send checkpoints to JobTracker indicating how much input file it has proceeded.

- Spills before the latest checkpoint can be merged.

Fault Tolerance: Reduce task failure

- Map tasks retain their output data on the local disk for the complete job duration.

- New reduce task just restart from beginning.

Multiple jobs setting

- User can submit a list of jobs that forms a directed acyclic graph.

- A map function consumes the output of previous reduce functions.

- Usually, this map function can not start until all its previous reduce functions complete (process all the data).

Pipeline between jobs

- Reduce function apply to the data it has so far, generate a snapshot, write it to the Hadoop file system.

- Map function of next job can consume the snapshot by pulling it out from file system.

- User can specify how often a snapshot is computed according to the progression metric (percentage of data arrived at reducer)

Multiple jobs aggregation

- Say job A uses the output of job B
- Each time the reduce of A computes a snapshot, send it to B's map and proceed it a little bit, then B's reduce compute another snapshot.
- B's snapshot must be recomputed every time it receives a new snapshot from A.

Continuous MapReduce Jobs

- Run MapReduce in real time

- Accepting Data as it becomes available and analyze it immediately.

Key Idea

- Add a optional “flush” operation to push data from map tasks to reduce tasks, when reduce task can not accept the data, the mapper will store it locally and send it later.

- User defined reduce task will periodically invoked on the output of the map available.

Fault Tolerance

- Problem here: map tasks can not remember the entire history to fast recover from reduce failure.

- They assume that reducer only depends on a suffix of the map history.

- Use ring buffer for map-side spill files.

Evaluation
where's the time consumption

- Map task two phases: map (most), sort

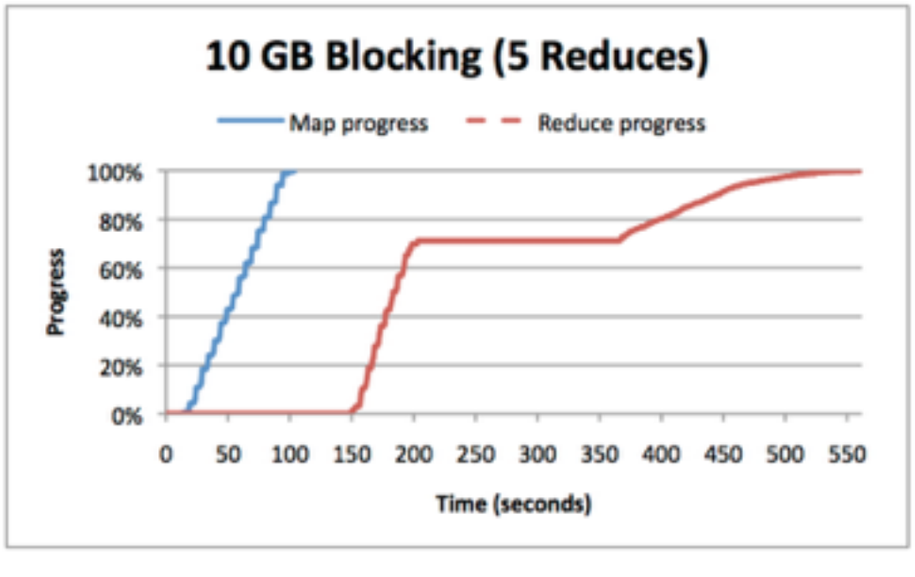
- Reduce task three phases: shuffle (75%), sort & commit (25%).

What they observe

- When reduce task is fast, map is slow, then they can improve over block MapReduce.

- When reduce task is bottleneck, they have no improvement.

When reduce task is fast



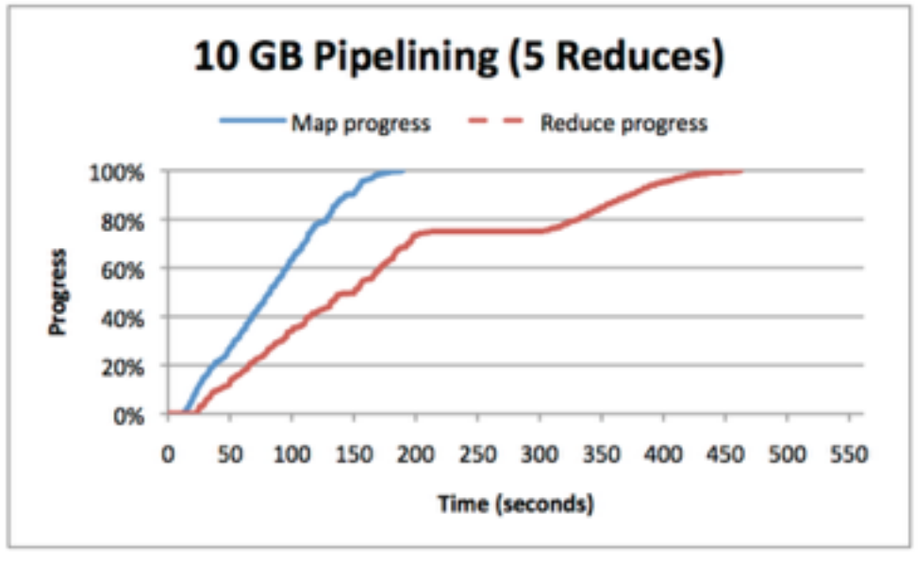
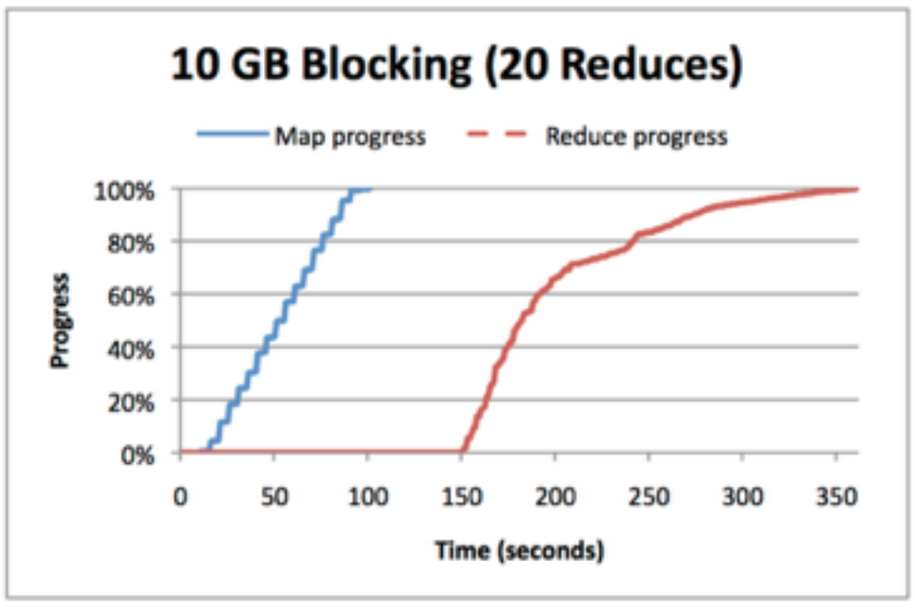


Figure 7: CDF of map and reduce task completion times for a 10GB wordcount job using 20 map tasks and 5 reduce tasks (512MB block size). The total job runtimes were 561 seconds for blocking and 462 seconds for pipelining.



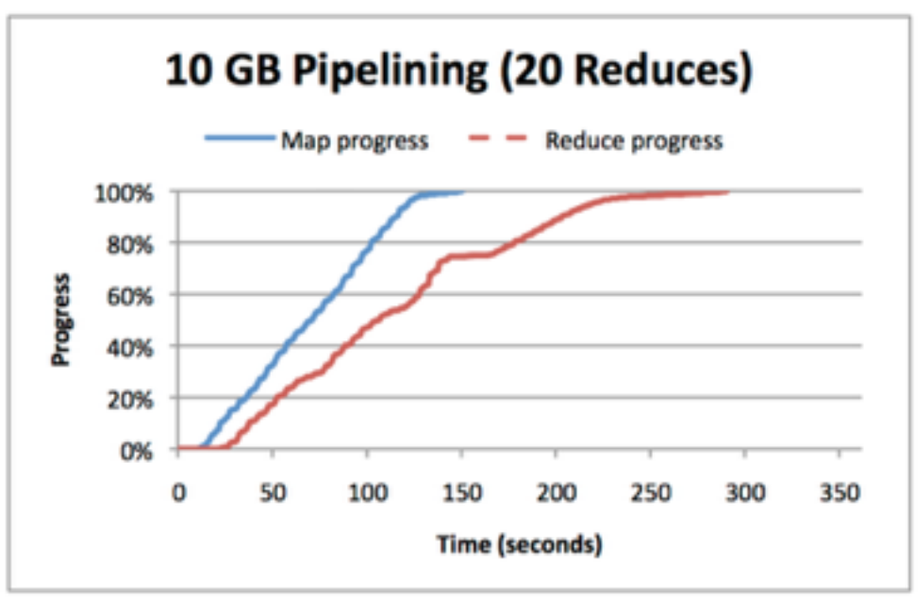
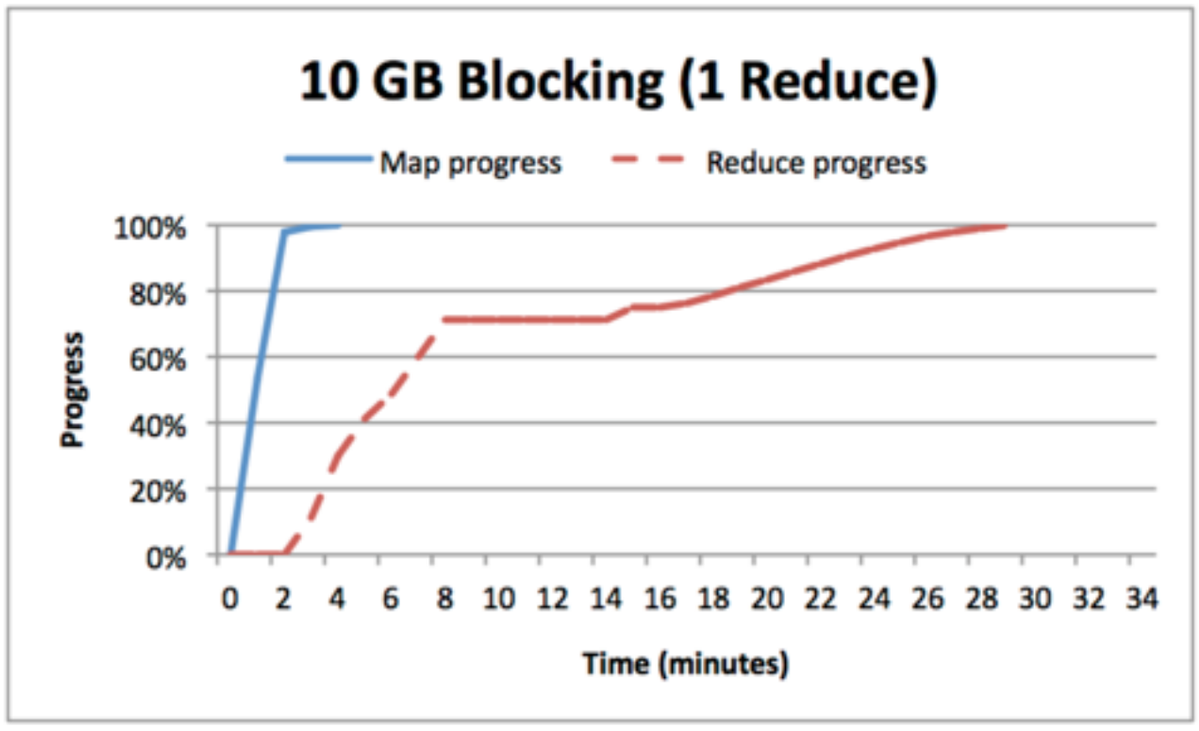


Figure 8: CDF of map and reduce task completion times for a 10GB wordcount job using 20 map tasks and 20 reduce tasks (512MB block size). The total job runtimes were 361 seconds for blocking and 290 seconds for pipelining.

When reduce task is slow



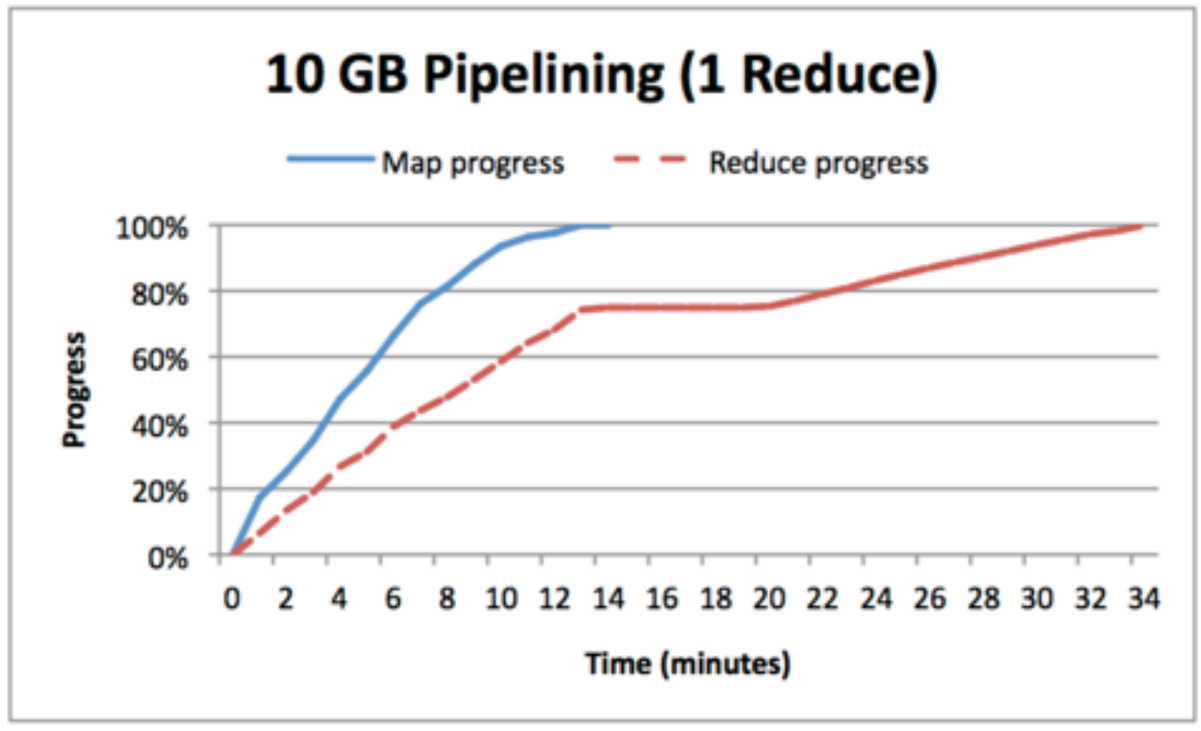


Figure 9: CDF of map and reduce task completion times for a 10GB wordcount job using 20 map tasks and 1 reduce task (512MB block size). The total job runtimes were 29 minutes for blocking and 34 minutes for pipelining.

Strength

- Preserve original MapReduce Architecture

- Allow pipeline/online aggregation

Weakness

- Snapshot accuracy is hard to evaluate
- Perform badly when reduce task is slow
- Only support fixed number of map/reduce tasks
- Failure recovery requires remembering entire history in worse case