# MapReduce: Simplified Data Processing on Large Clusters

Presenter: Scott Wolchok

Paper authors: Jeffrey Dean and Sanjay Ghemawat

# The Problem

- We're writing lots of special-purpose code as part of our search engine.
- The parts of this code that handle distributing and parallelizing it are obnoxious.
- Let's build a fault-tolerant automatic parallelization system to hide this!

# Basic Programming Model

- map: (k1,v1) → [(k2,v2)]
- Manager does GROUP BY k2, passes to reduce
- Reduce: k2, [v2] → [v3]
- Typically the output of reduce is smaller than the input.
- System implicitly sorts on k2.

# Example query

- Count names beginning with each letter
- SELECT SUBSTR(name, 1, 2), count(*) FROM tbl

  GROUP BYSUBSTR(name, 1, 2)

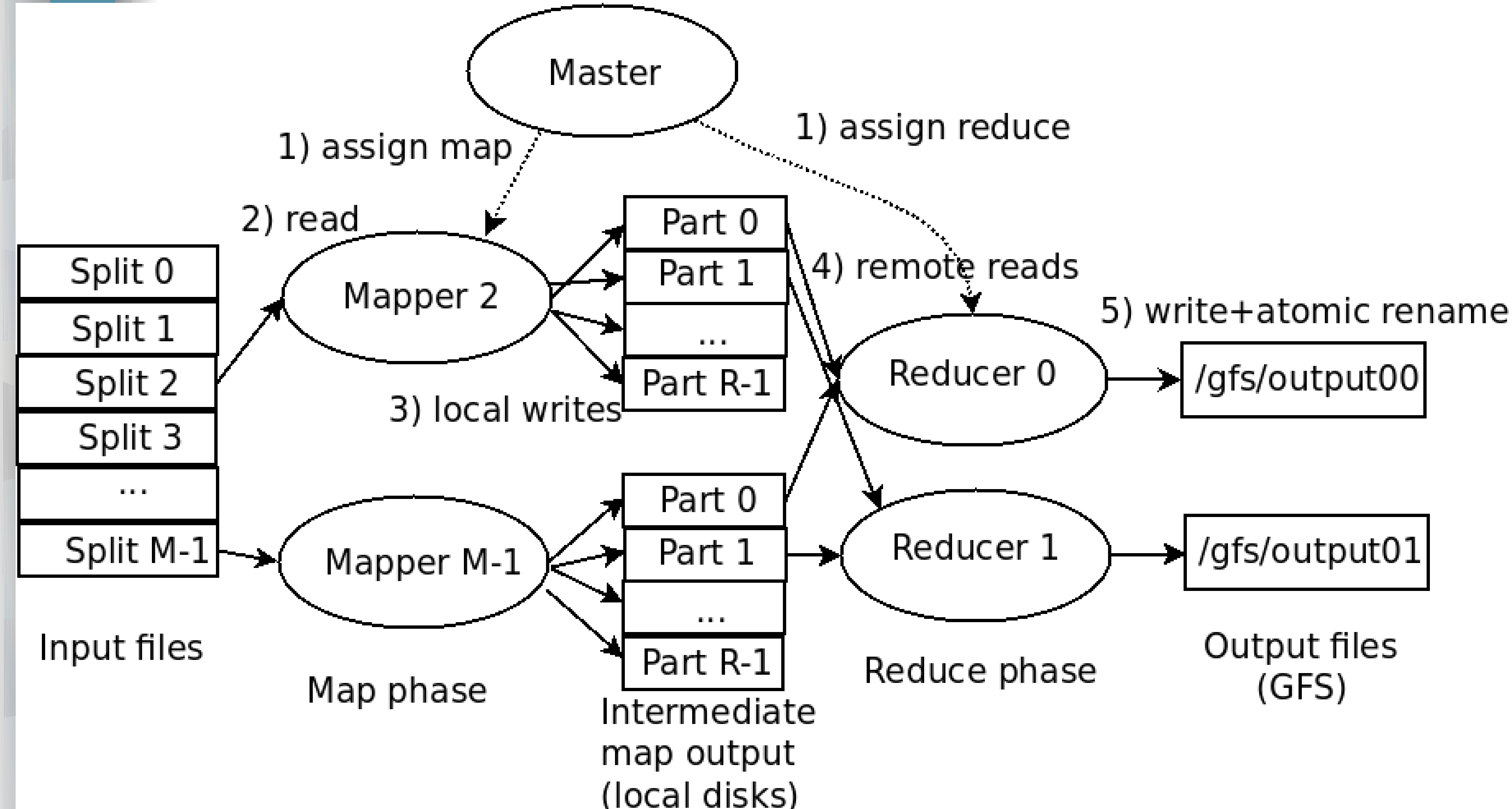
# Example query (cont.)

- map(String key, String value) {
  ```
  // Group by key is the first character of the key
  EmitIntermediate(key[0], 1);
  ```
  }
- reduce(String key, Iterator values) {
  ```
  int result = 0;
  foreach v in values:
    result += ParseInt(v);
  Emit(AsString(result));
  ```
  }

# Implementation Environment

- Large cluster of commodity PCs
- Machine failures are common
- Shared-nothing; local disks
- Central scheduler maps tasks to nodes

# Execution


Master
1) assign map
1) assign reduce
2) read
Split 0
Split 1
Split 2
Split 3
...
Split M-1
Mapper 2
Mapper M-1
Part 0
Part 1
...
Part R-1
Part 0
Part 1
...
Part R-1
3) local writes
4) remote reads
5) write+atomic rename
Reducer 0
Reducer 1
/gfs/output00
/gfs/output01
Input files
Map phase
Intermediate map output (local disks)
Reduce phase
Output files (GFS)

# Execution

- M partitions of input (any method)
- R reduce tasks, typically hash partition
- Map workers read 1 of M partitions, execute map function, write to R output partitions on local disk
- Reduce workers read partitions from map workers, sort by key, execute reduce function, atomic write to GFS

# Fault tolerance

- Master pings workers; on timeout, re-execute all maps and pending reduces.
- If master fails, give up. Client can retry.
- Mappers tell master about output files (atomic commit)
- Reducers atomically rename temporary to output files (rely on GFS for commit)
- Atomic commits => serializability

# Non-determinism

- Not globally serializable for non-deterministic map/reduce functions
- Instead, each reduce task's output is equivalent to some serial execution; no single global execution
- Why? R1 may read different map execution from R2 if a mapper failed

# Locality

- Input stored on local disks (GFS, 64 MB block size)
- Master schedules map tasks on or close to the input replicas

# Granularity

- Fine granularity (many tasks) => good load balancing, fast recovery
- Scheduling cost O(M+R), state size O(MR) (but it's MR bytes)
- M chosen for input data = 16-64 MB
- Typically, M=200k, R=5k, 2k machines

# Stragglers

- Small percentage of slow machines (e.g., bad disk, disabled L1/L2 cache)
- Solution: redundantly execute the last few tasks, tune threshold for low cost

# Extra features

- Reduce partitioning fxn is user-defined
- Reduce partitions are sorted by key
- Map workers can run reduce function early as a "combiner" (e.g., to reduce k ('the', 1) output tuples to ('the', k))

# Input and Output Types

- The built-in types:
  - Text: key = line number, value = line
  - key/value pairs sorted by key
- Support user-defined Readers as well, such as from DB or from RAM
- Similar support for output types
- [Can Readers be shared to enforce a partial schema? Revisit later]

# Skipping Bad Records

- User code might crash on some records
- Worker sends a UDP packet with record number to the master upon segfault
- If master gets more than one segfault for a record, it tells the next worker to skip that record.
- [Partial method of tolerating garbage input]

# Performance Evaluation

- They show that it works
- 1800 machines with dual Xeons, 4 GB of RAM, 160 GB disk, gigabit Ethernet
- Distributed grep: search 10 billion 100-byte records for a ~0.001% likely value
- Distributed sort:1 TB, same record size

# Grep performance

Actual time: 150 seconds (~1 minute of startup overhead)

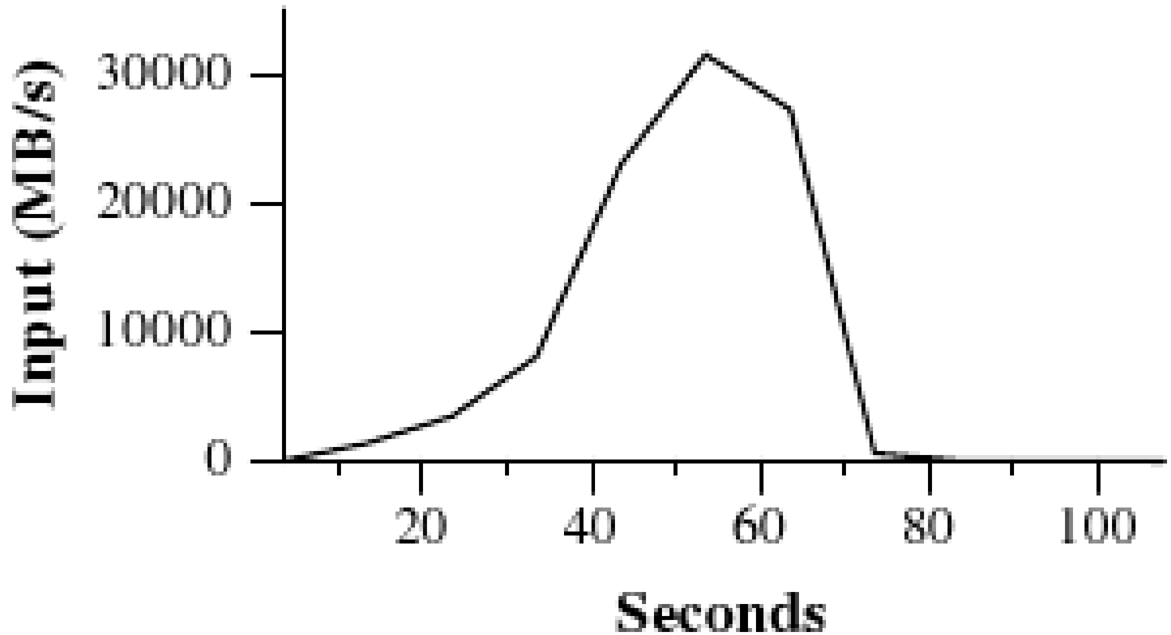
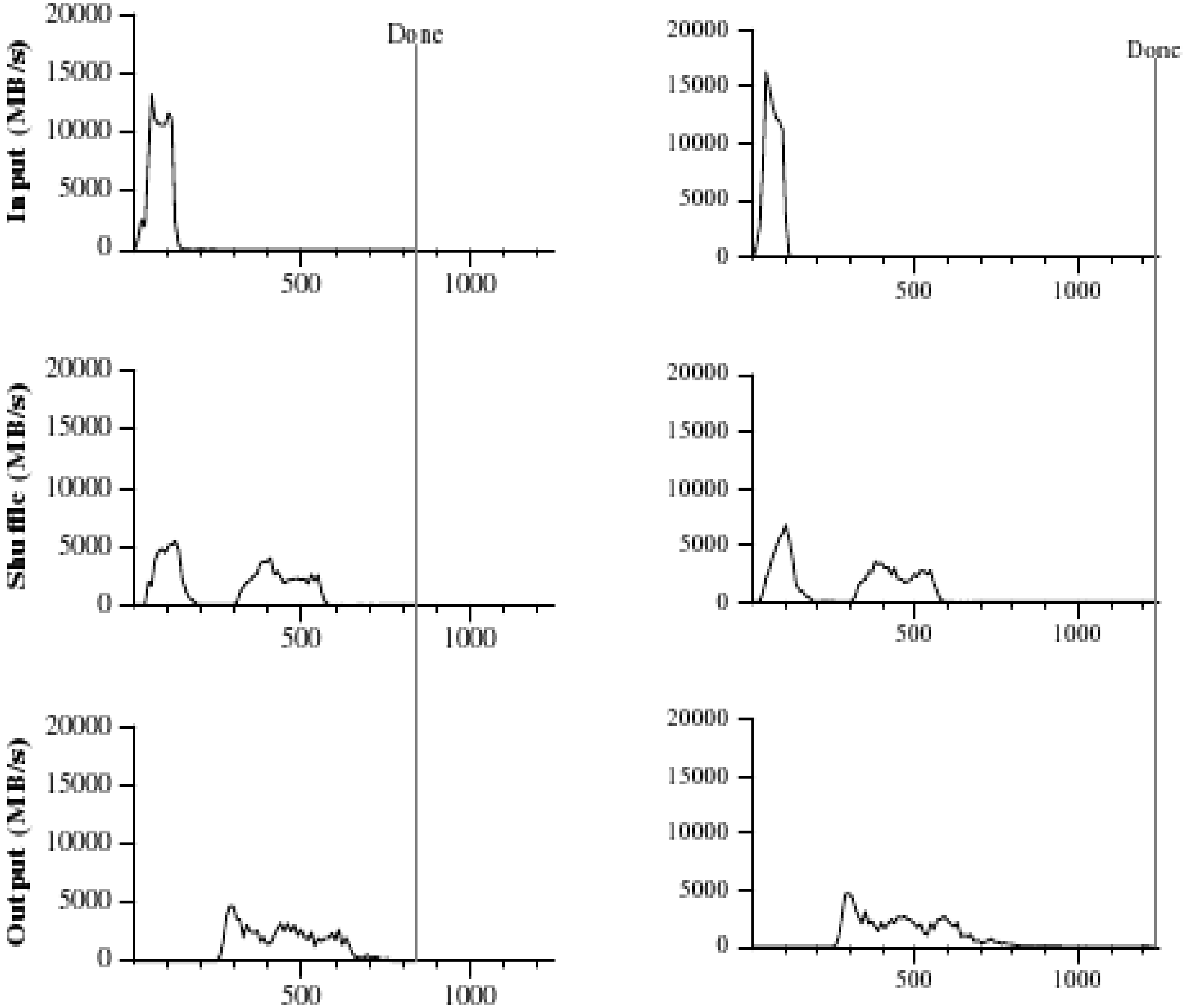



Figure 2: Data transfer rate over time

# Sort & Stragglers



# Conclusion

- People can understand this model
- Google uses MapReduce for processing our Web crawl results
- We need more network bandwidth
- Replicated execution is good

# MapReduce Controversy

- “MapReduce: A Major Step Backwards” by DeWitt and Stonebraker, 2008
- Lots of hype around MapReduce; Berkeley teaching to freshmen
- Five criticisms outlined in detail with respect to traditional database workloads

# Step Backwards in Paradigm

- MapReduce has no whole-table schema
- Cannot check data-set consistency
- Must find source code to infer schema

# Poor Implementation

- A file scan is the only access method; there are no indexes (poor SELECT)
- Skew in the map phase delays the reduce phase, delaying finish times
- Simultaneous reads of materialized files cause disk seeks [really?]
- We've had distributed DBMSes that use schemas and indexing for 20 years

# Not Novel

- We already had:
  - – Data partitioning
  - – Parallel hash joins on shared-nothing clusters
  - – Parallel aggregates with or without GROUP BY
  - – User-defined functions and aggregates

# Missing Features

- MapReduce doesn't have many DBMS features, including:
  - Bulk load
  - Indexes
  - Updates
  - Transactions
  - Integrity constraints
  - Views [unclear if MR Section 4.4 will do]

# Incompatible with DBMS Tools

- Because MapReduce isn't a DBMS, you can't use DBMS tools with it

# Conclusion

- Please go read the last 25 years of // DBMS research
- Fault tolerance is a nice contribution
- Database usability **is** a problem

See also “A Comparison of Approaches to Large-Scale Data Analysis” by Pavlo et al. from SIGMOD 2009.