CSE 487/587
Data Intensive Computing

Lecture 5: GFS, HDFS, Mapreduce

Vipin Chaudhary

vipin@buffalo.edu

716.645.4740
305 Davis Hall

Overview of Today's Lecture

- Google File System (contd)
- Hadoop Distributed File System – HDFS
- MapReduce

- Program submission
  - /gpfs/courses/cse487/Spring2015
  - UBLearns

Distributed File System

- Don't move data to workers… move workers to the data!
  - Store data on the local disks of nodes in the cluster
  - Start up the workers on the node that has the data local
- Why?
  - Not enough RAM to hold all the data in memory
  - Disk access is slow, but disk throughput is reasonable
- A distributed file system is the answer
  - GFS (Google File System) for Google's MapReduce
  - HDFS (Hadoop Distributed File System) for Hadoop

GFS: Assumptions

- Commodity hardware over "exotic" hardware
  - Scale "out", not "up"
- High component failure rates
  - Inexpensive commodity components fail all the time
  - Fault-tolerance and auto-recovery essential
- "Modest" number of huge files
  - Multi-gigabyte files are common, if not encouraged
- Files are write-once, mostly appended to
  - Perhaps concurrently
- Large streaming reads over random access
  - High sustained throughput over low latency
- Concurrent appends by multiple clients (e.g., producer-consumer queues)
  - Want atomicity for appends without synchronization overhead among clients

GFS: Architecture

- One master server (state replicated on backups)
- Many chunk servers (100s – 1000s)
  - Spread across racks; intra-rack b/w greater than inter-rack
- Each Chunk: 64 MB portion of file, identified by 64-bit, globally unique ID
- Many clients accessing same and different files stored on same cluster

GFS: Architecture (Contd. 2)

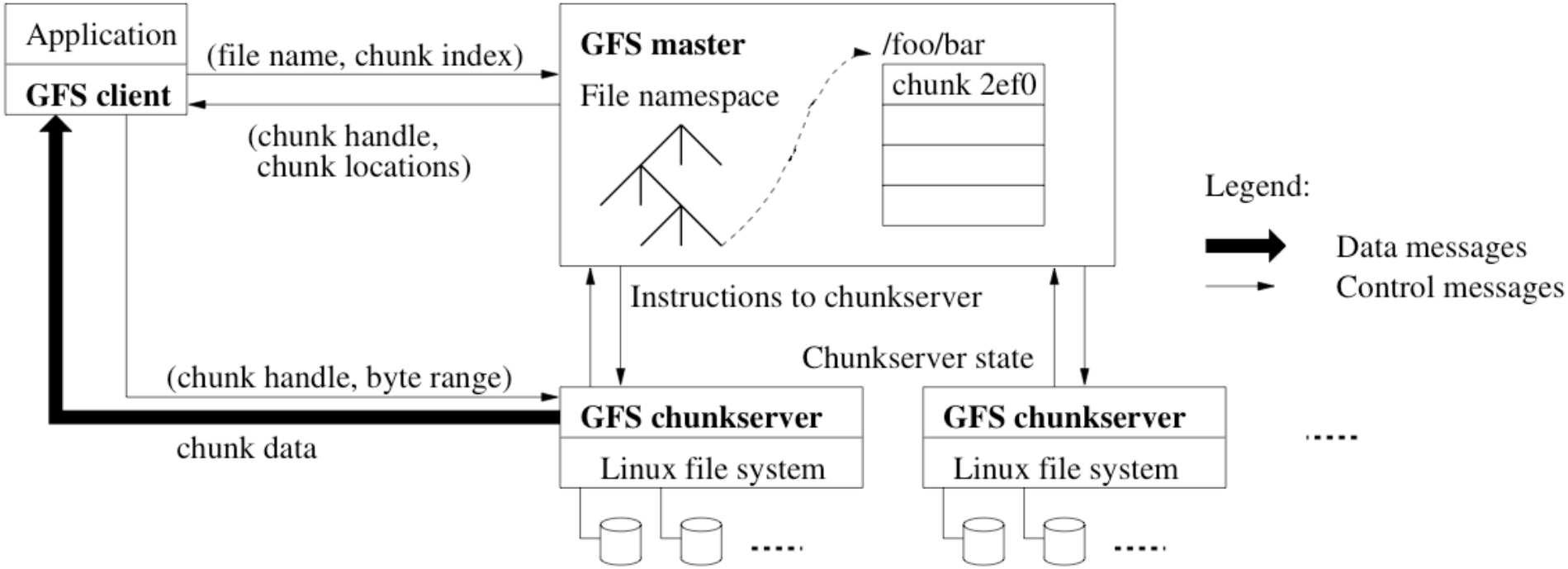


Figure 1: GFS Architecture

Master Server

- Holds all metadata:
  - Namespace (directory hierarchy)
  - Holds all metadata in RAM; very fast operations on file system metadata
  - Current locations of chunks (chunkservers)
- Delegates consistency management
- Garbage collects orphaned chunks
- Migrates chunks between chunkservers

Chunkserver

- Stores 64 MB file chunks on local disk using standard Linux filesystem, each with version number and checksum
- Read/write requests specify chunk handle and byte range
- Chunks replicated on configurable number of chunkservers (default: 3)
- No caching of file data (beyond standard Linux buffer cache)

Client

- Issues control (metadata) requests to master server
- Issues data requests directly to chunkservers
- Caches metadata
- Does no caching of data
  - No consistency difficulties among clients
  - Streaming reads (read once) and append writes (write once) don't benefit much from caching at client

GFS: Architecture (Contd. 2)

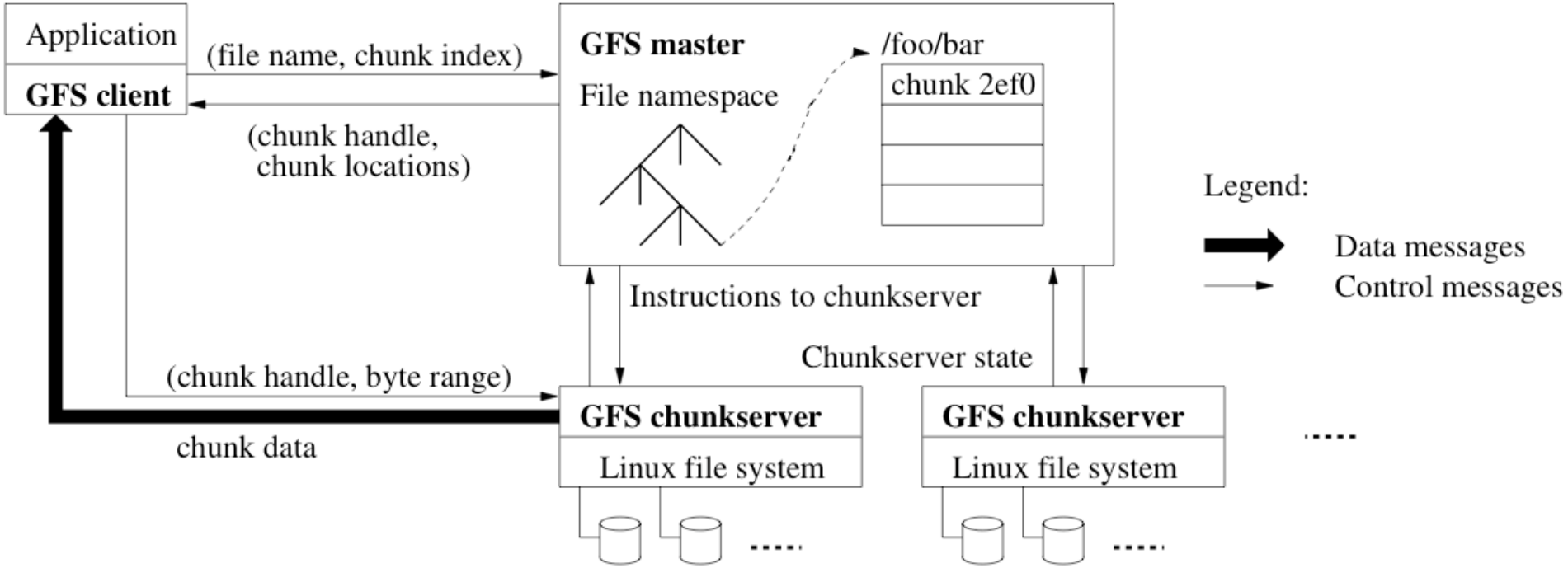


Figure 1: GFS Architecture

Client API

- Not a filesystem in traditional sense
  - Not POSIX compliant
  - Does not use kernel VFS interface
  - Library that apps can link in for storage access
- API:
  - open, delete, read, write (as expected)
  - snapshot: quickly create copy of file
  - append: **at least once, possibly with gaps and/or inconsistencies among clients**

Client Read

- Client sends master:
  - read(file name, chunk index)
- Master's reply:
  - chunk ID, chunk version number, locations of replicas
- Client sends "closest" chunkserver w/replica:
  - read(chunk ID, byte range)
  - "Closest" determined by IP address on simple rack-based network topology
- Chunkserver replies with data

Client Write

- Some chunkserver is primary for each chunk
  - Master grants lease to primary (~ 1 min)
  - Lease renewed using periodic heartbeat messages between master and chunkservers
- Client asks master for primary and secondary replicas for each chunk
- Client sends data to replicas in daisy chain
  - Pipelined: each replica forwards as it receives
  - Switched network with full duplex links

Client Write (2)

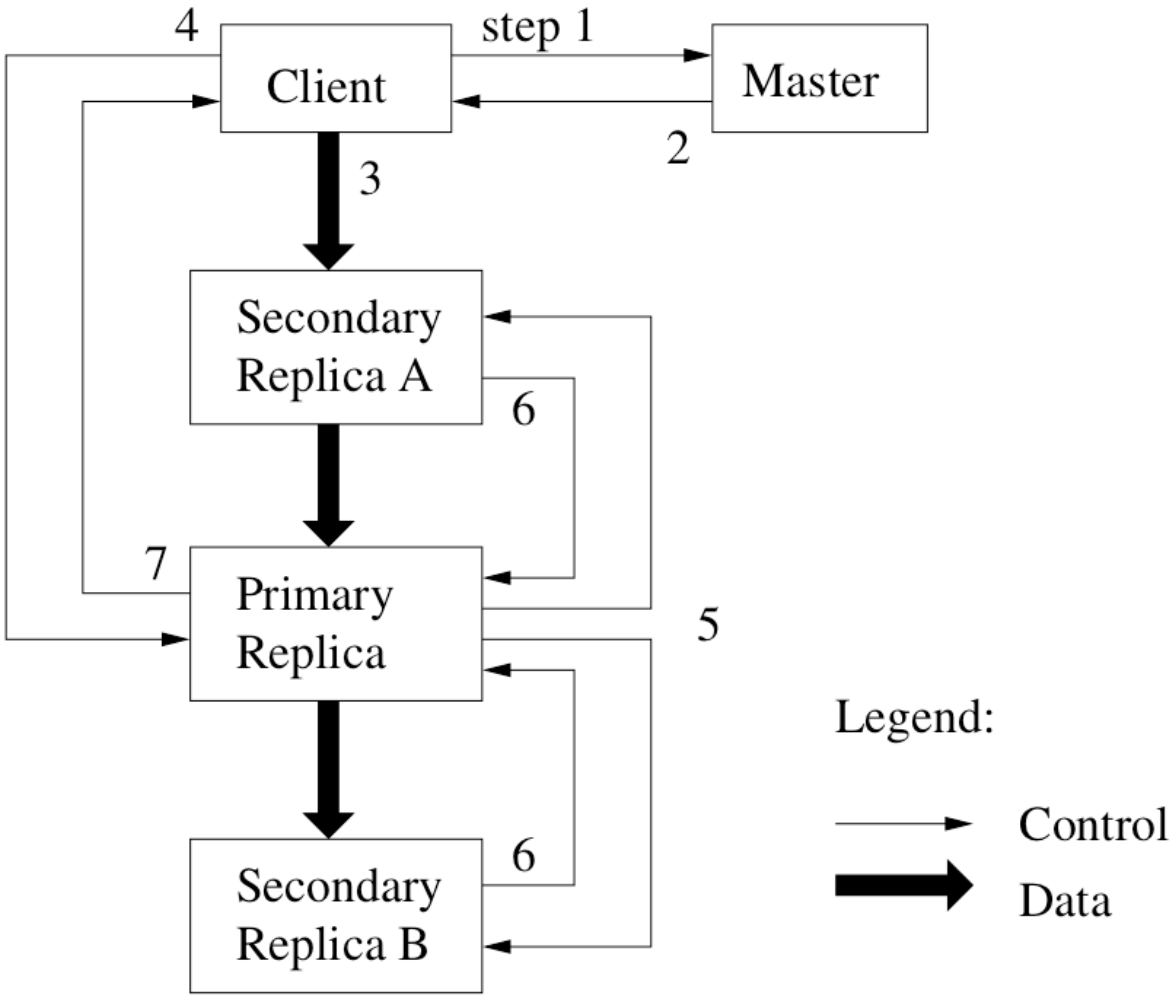


Figure 2: Write Control and Data Flow

Client Write (3)

- All replicas acknowledge data write to client
- Client sends write request to primary
- Primary assigns serial number to write request, providing ordering
- Primary forwards write request with same serial number to secondaries
- Secondaries all reply to primary after completing write
- Primary replies to client

Client Write (2)

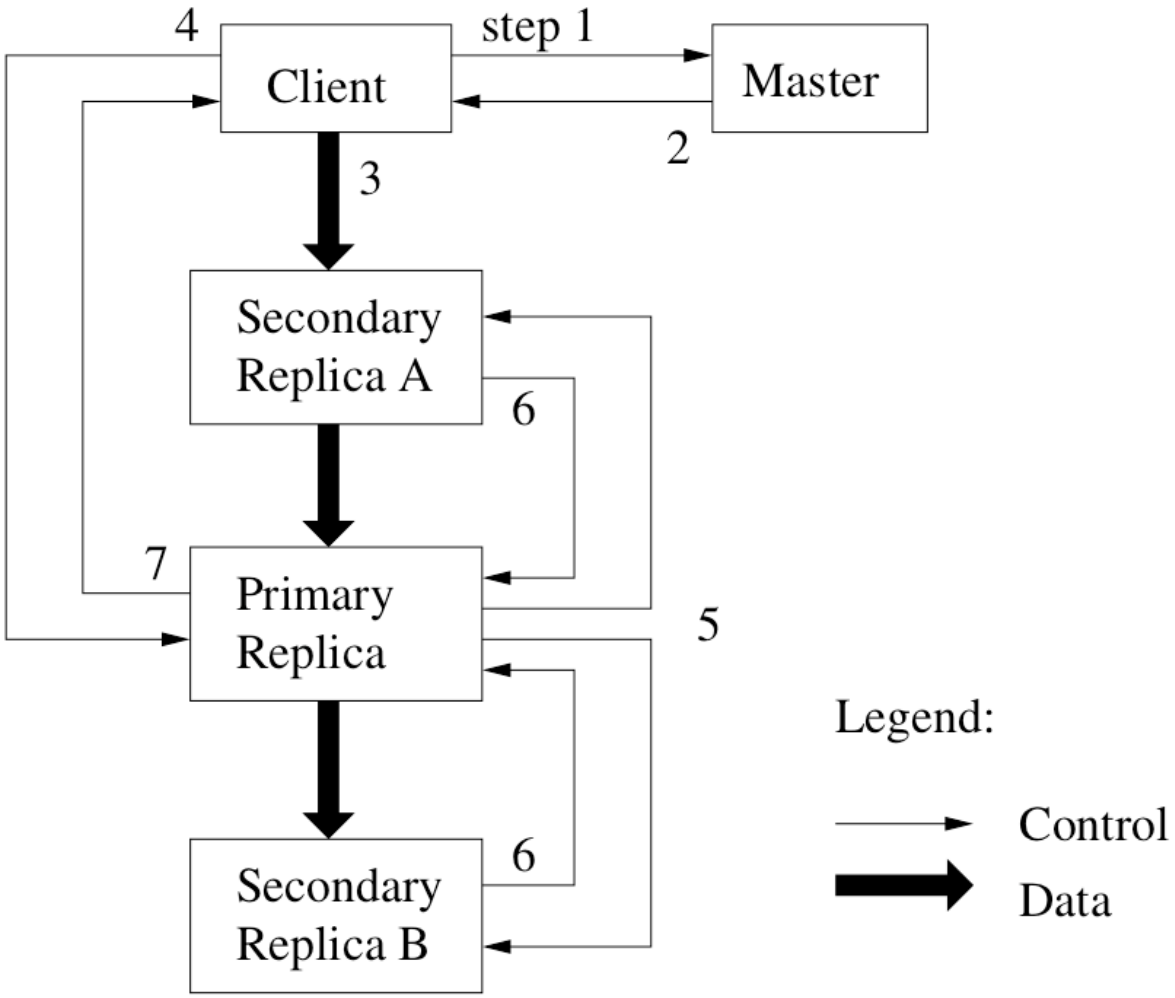


Figure 2: Write Control and Data Flow

Client (Atomic) Record Append

- Google uses large files as queues between multiple producers and consumers
- Same control flow as for writes, except…
- Client pushes data to replicas of last chunk of file
- Client sends request to primary
- Common case: request fits in current last chunk:
  - Primary appends data to own replica
  - Primary tells secondaries to do same at same byte offset in theirs
  - Primary replies with success to client

Client (Atomic) Record Append (2)

- When data won't fit in last chunk:
  - Primary fills current chunk with padding
  - Primary instructs other replicas to do same
  - Primary replies to client, "retry on next chunk"
- If record append fails at any replica, client retries operation
  - So replicas of same chunk may contain different data—even duplicates of all or part of record data
- **What guarantee does GFS provide on success?**
  - Data written at least once in atomic unit

GFS: Consistency Model

- Changes to namespace (i.e., metadata) are atomic
  - Done by single master server!
  - Master uses log to define global total order of namespace-changing operations

GFS: Consistency Model (2)

- Changes to data are ordered as chosen by a primary
  - All replicas will be consistent
  - But multiple writes from the same client may be interleaved or overwritten by concurrent operations from other clients
- Record append completes at least once, at offset of GFS's choosing
  - Applications must cope with possible duplicates

GFS: Data Mutation Consistency

	Write	Record Append
serial success	defined	defined interspersed with inconsistent
concurrent success	consistent but undefined	
failure	inconsistent	

Applications and Record Append Semantics

- Applications should use self-describing records and checksums when using Record Append
  - Reader can identify padding / record fragments
- If application cannot tolerate duplicated records, should include unique ID in record
  - Reader can use unique IDs to filter duplicates

Logging at Master

- Master has all metadata information
  - Lose it, and you've lost the filesystem!
- Master logs all client requests to disk sequentially
- Replicates log entries to remote backup servers
- Only replies to client after log entries safe on disk on self and backups!

Chunk Leases and Version Numbers

- If no outstanding lease when client requests write, master grants new one
- Chunks have version numbers
  - Stored on disk at master and chunkservers
  - Each time master grants new lease, increments version, informs all replicas
- Master can revoke leases
  - e.g., when client requests rename or snapshot of file

What If the Master Reboots?

- Replays log from disk
  - Recovers namespace (directory) information
  - Recovers file-to-chunk-ID mapping
- Asks chunkservers which chunks they hold
  - Recovers chunk-ID-to-chunkserver mapping
- If chunk server has older chunk, it's stale
  - Chunk server down at lease renewal
- If chunk server has newer chunk, adopt its version number
  - Master may have failed while granting lease

What if Chunkserver Fails?

- Master notices missing heartbeats
- Master decrements count of replicas for all chunks on dead chunkserver
- Master re-replicates chunks missing replicas in background
  - Highest priority for chunks missing greatest number of replicas

File Deletion

- When client deletes file:
  - Master records deletion in its log
  - File renamed to hidden name including deletion timestamp
- Master scans file namespace in background:
  - Removes files with such names if deleted for longer than 3 days (configurable)
  - In-memory metadata erased
- Master scans chunk namespace in background:
  - Removes unreferenced chunks from chunkservers

Limitations

- Security?
  - Trusted environment, trusted users
  - But that doesn't stop users from interfering with each other…
- Does not mask all forms of data corruption
  - Requires application-level checksum

Cluster	A	B
Chunkservers	342	227
Available disk space	72 TB	180 TB
Used disk space	55 TB	155 TB
Number of Files	735 k	737 k
Number of Dead files	22 k	232 k
Number of Chunks	992 k	1550 k
Metadata at chunkservers	13 GB	21 GB
Metadata at master	48 MB	60 MB

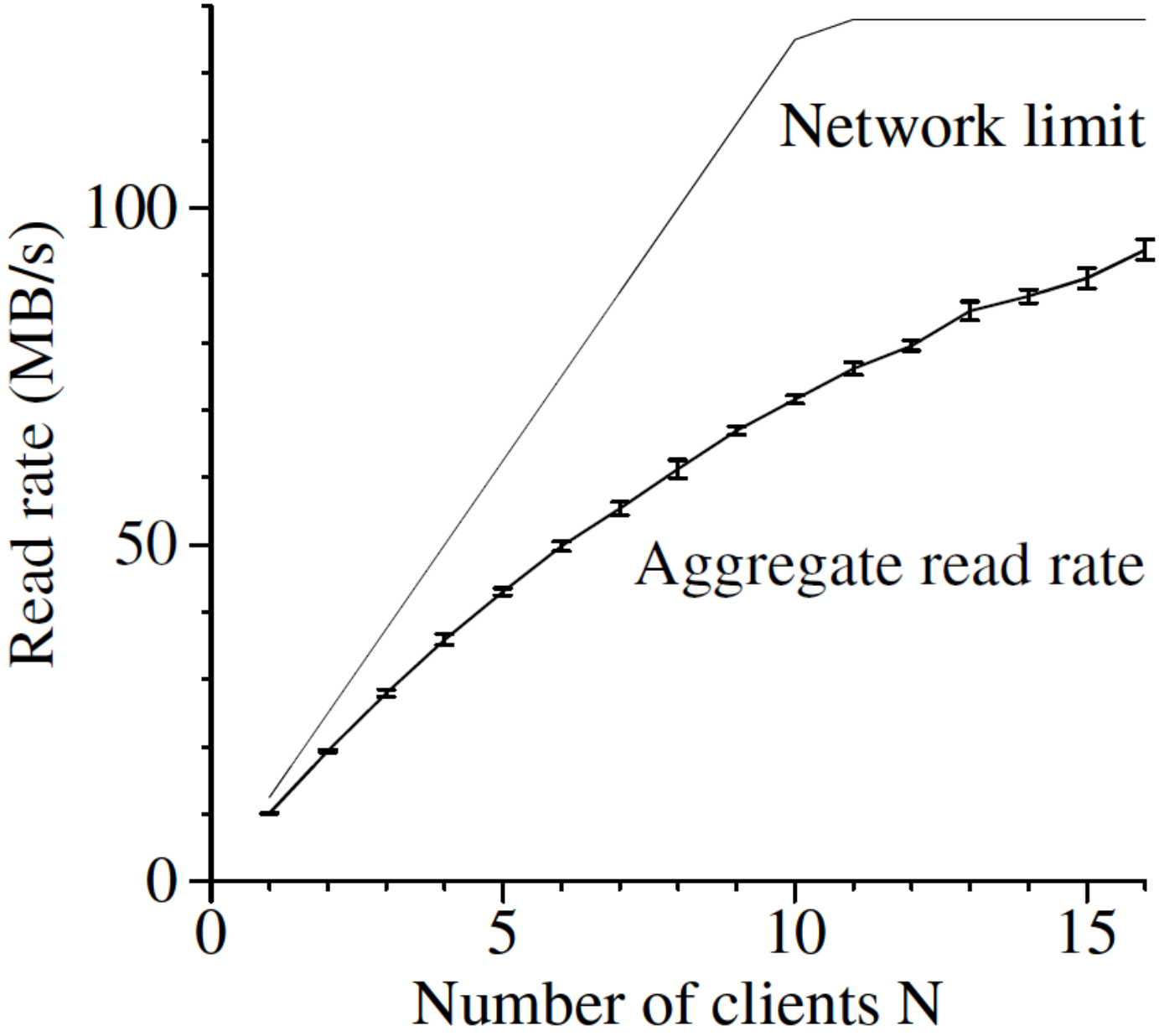
Table 2: Characteristics of two GFS clusters

Cluster	A	B
Read rate (last minute)	583 MB/s	380 MB/s
Read rate (last hour)	562 MB/s	384 MB/s
Read rate (since restart)	589 MB/s	49 MB/s
Write rate (last minute)	1 MB/s	101 MB/s
Write rate (last hour)	2 MB/s	117 MB/s
Write rate (since restart)	25 MB/s	13 MB/s
Master ops (last minute)	325 Ops/s	533 Ops/s
Master ops (last hour)	381 Ops/s	518 Ops/s
Master ops (since restart)	202 Ops/s	347 Ops/s

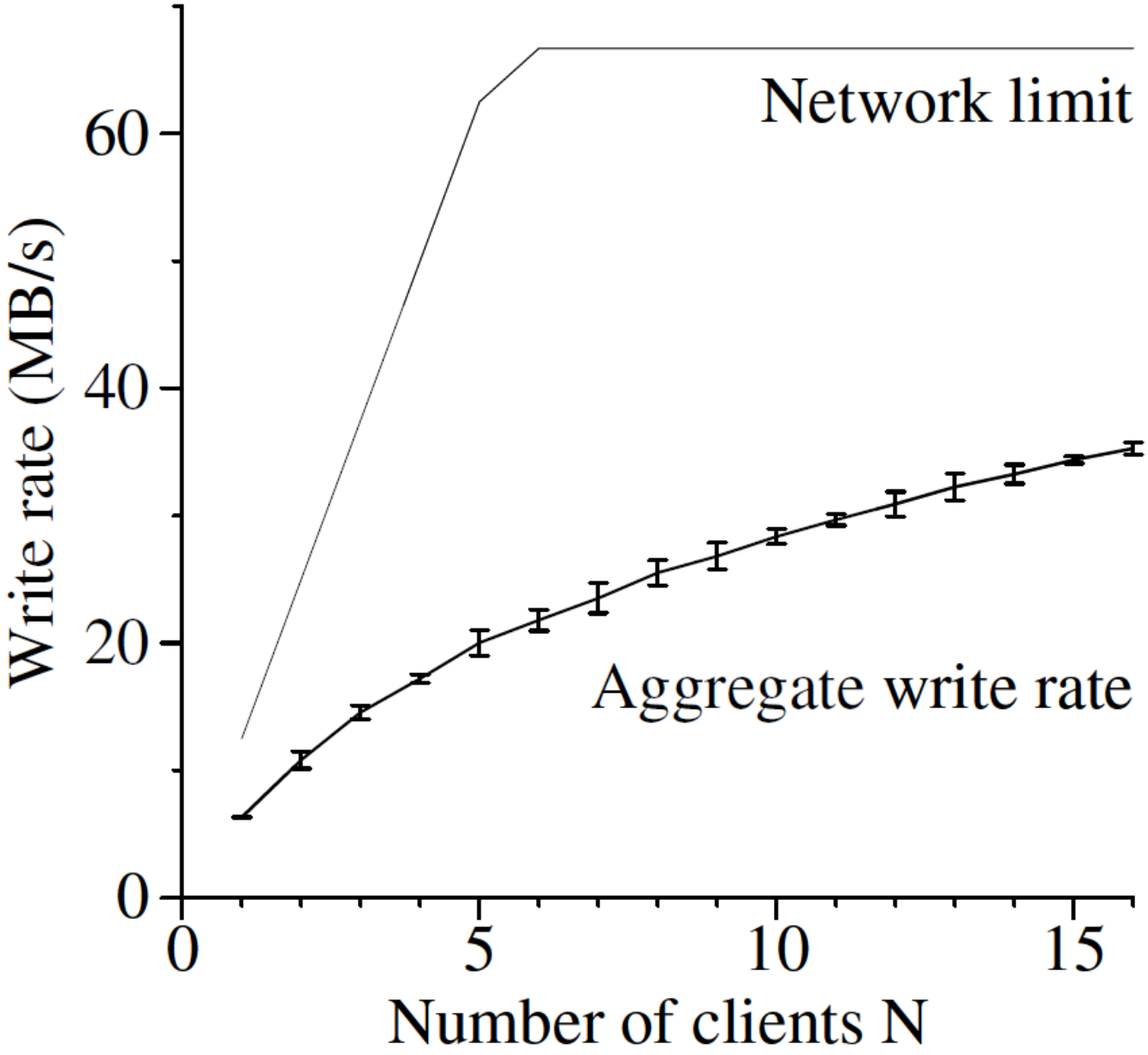
Table 3: Performance Metrics for Two GFS Clusters

Cluster	X	Y
Open	26.1	16.3
Delete	0.7	1.5
FindLocation	64.3	65.8
FindLeaseHolder	7.8	13.4
FindMatchingFiles	0.6	2.2
All other combined	0.5	0.8

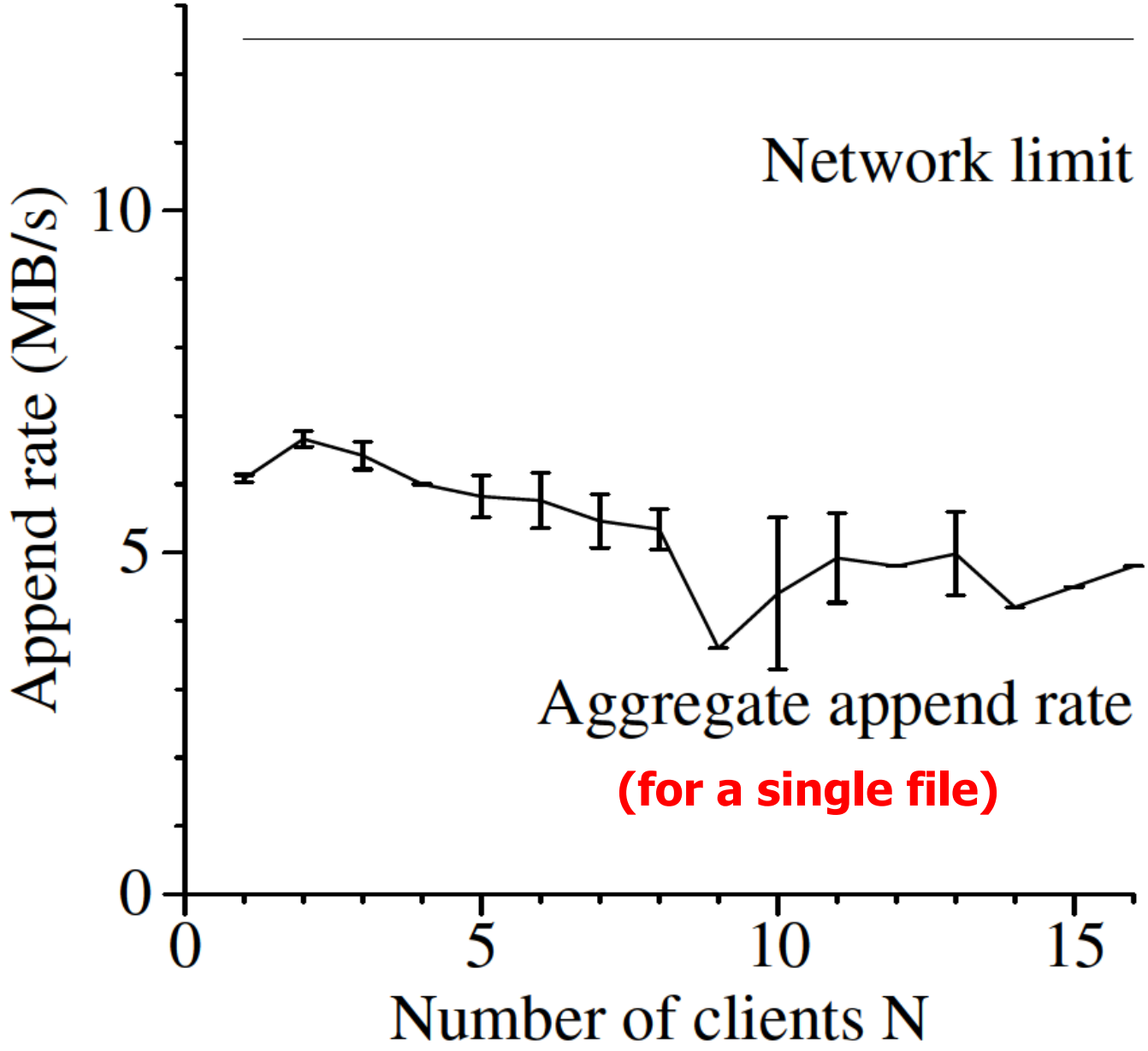
Table 6: Master Requests Breakdown by Type (%)



(a) Reads



(b) Writes



(c) Record appends

Recovery Time

- Experiment: killed 1 chunkserver
  - Clonings limited to 40% of the chunkservers and 50 Mbps each to limit impact on applications
  - All chunks restored in 23.2 min
- Experiment: killed 2 chunkservers
  - 266 of 16,000 chunks reduced to single replica
  - Higher priority re-replication for these chunks
  - Achieved 2x replication within 2 min

Operation	Read		Write		Record Append	
Cluster	X	Y	X	Y	X	Y
0K	0.4	2.6	0	0	0	0
1B..1K	0.1	4.1	6.6	4.9	0.2	9.2
1K..8K	65.2	38.5	0.4	1.0	18.9	15.2
8K..64K	29.9	45.1	17.8	43.0	78.0	2.8
64K..128K	0.1	0.7	2.3	1.9	< .1	4.3
128K..256K	0.2	0.3	31.6	0.4	< .1	10.6
256K..512K	0.1	0.1	4.2	7.7	< .1	31.2
512K..1M	3.9	6.9	35.5	28.7	2.2	25.5
1M..inf	0.1	1.8	1.5	12.3	0.7	2.2

Table 4: Operations Breakdown by Size (%). For reads, the size is the amount of data actually read and transferred, rather than the amount requested.

Operation	Read		Write		Record Append	
Cluster	X	Y	X	Y	X	Y
1B..1K	< .1	< .1	< .1	< .1	< .1	< .1
1K..8K	13.8	3.9	< .1	< .1	< .1	0.1
8K..64K	11.4	9.3	2.4	5.9	2.3	0.3
64K..128K	0.3	0.7	0.3	0.3	22.7	1.2
128K..256K	0.8	0.6	16.5	0.2	< .1	5.8
256K..512K	1.4	0.3	3.4	7.7	< .1	38.4
512K..1M	65.9	55.1	74.1	58.0	.1	46.8
1M..inf	6.4	30.1	3.3	28.0	53.9	7.4

Table 5: Bytes Transferred Breakdown by Operation Size (%). For reads, the size is the amount of data actually read and transferred, rather than the amount requested. The two may differ if the read attempts to read beyond end of file, which by design is not uncommon in our workloads.

GFS: Summary

- Semantics not transparent to apps
  - Must verify file contents to avoid inconsistent regions, repeated appends (at-least-once semantics)
- Performance not good for all apps
  - Assumes read-once, write-once workload (no client caching!)

GFS: Design Decisions

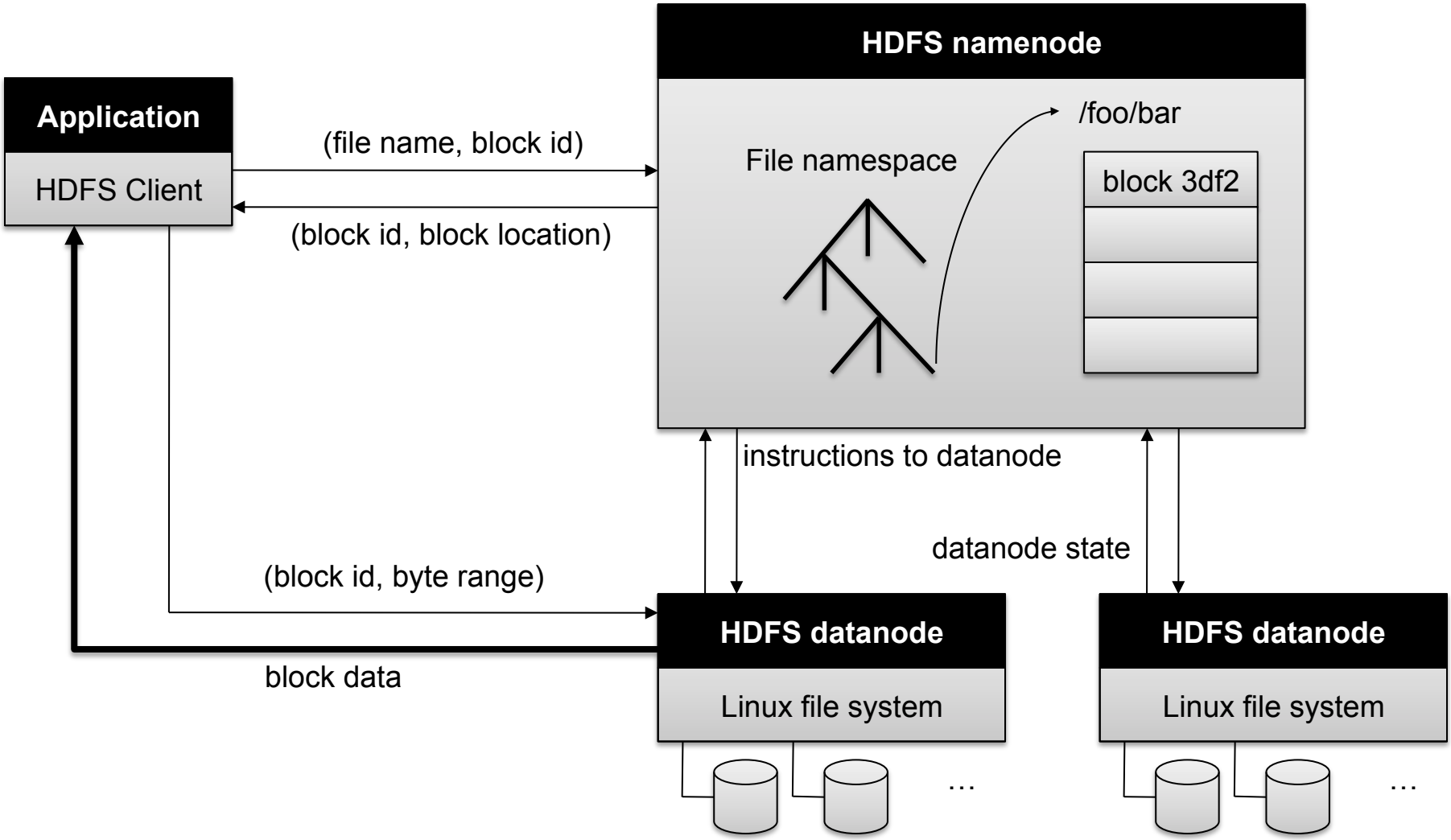
- Files stored as chunks
  - Fixed size (64MB)
- Reliability through replication
  - Each chunk replicated across 3+ chunkservers
- Single master to coordinate access, keep metadata
  - Simple centralized management
- No data caching
  - Little benefit due to large datasets, streaming reads
- Simplify the API
  - Push some of the issues onto the client (e.g., data layout)

HDFS = GFS clone (same basic ideas)

From GFS to HDFS

- Terminology differences:
  - GFS master = Hadoop namenode
  - GFS chunkservers = Hadoop datanodes
- Differences:
  - Different consistency model for file appends
  - Implementation
  - Performance

HDFS Architecture

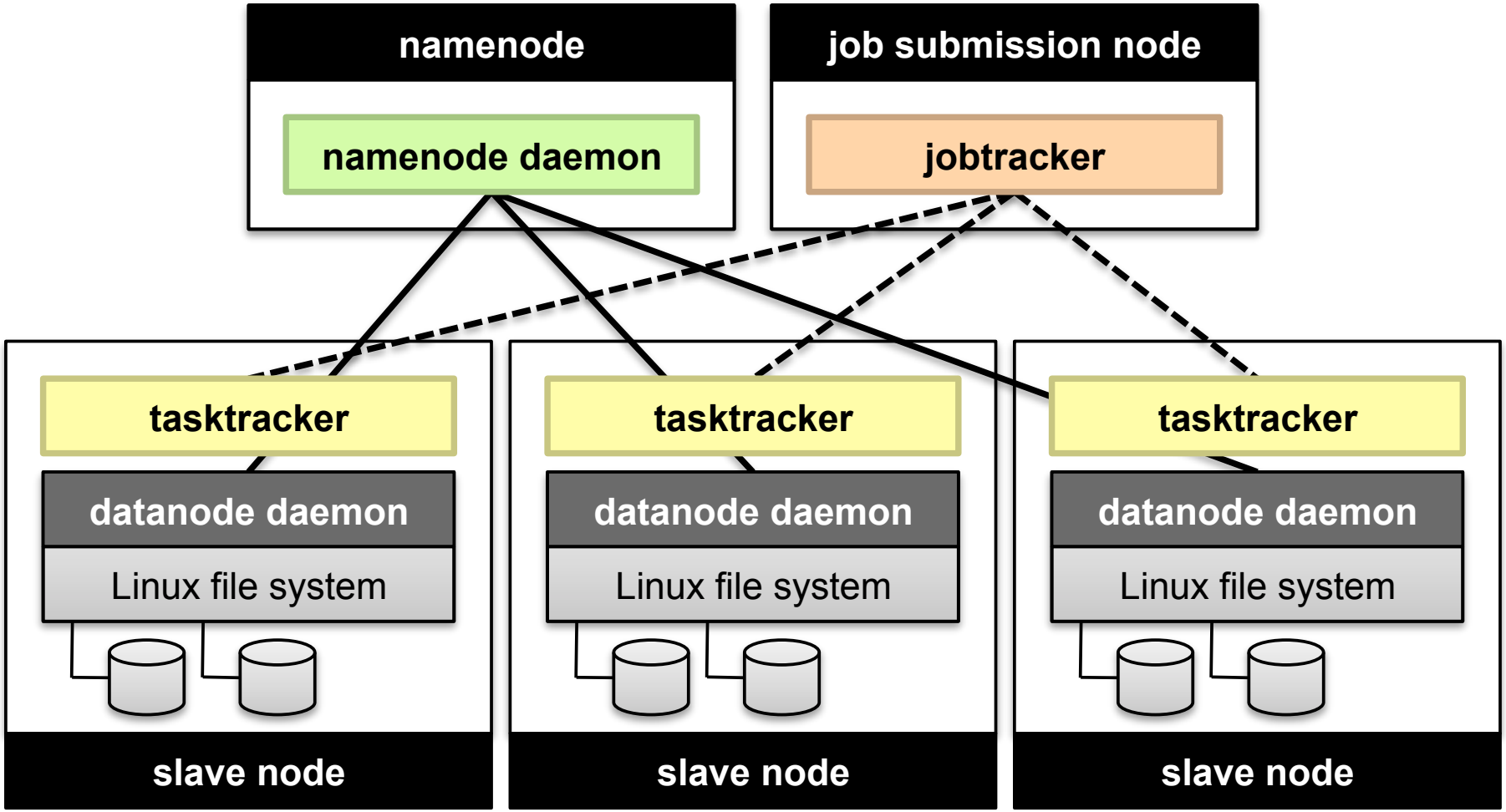


Adapted from (Ghemawat et al., SOSP 2003)

Namenode Responsibilities

- Managing the file system namespace:
  - Holds file/directory structure, metadata, file-to-block mapping, access permissions, etc.
- Coordinating file operations:
  - Directs clients to datanodes for reads and writes
  - No data is moved through the namenode
- Maintaining overall health:
  - Periodic communication with the datanodes
  - Block re-replication and rebalancing
  - Garbage collection

Putting everything together…



(Not Quite… We'll come back to YARN later)

More on HDFS

- Go through the HDFS paper
- And associated presentation
- Posted on UBLearns

MapReduce: Simplified Data Processing on Large Clusters

OSDI 2004

Original Paper, other tutorials from Google

Motivation

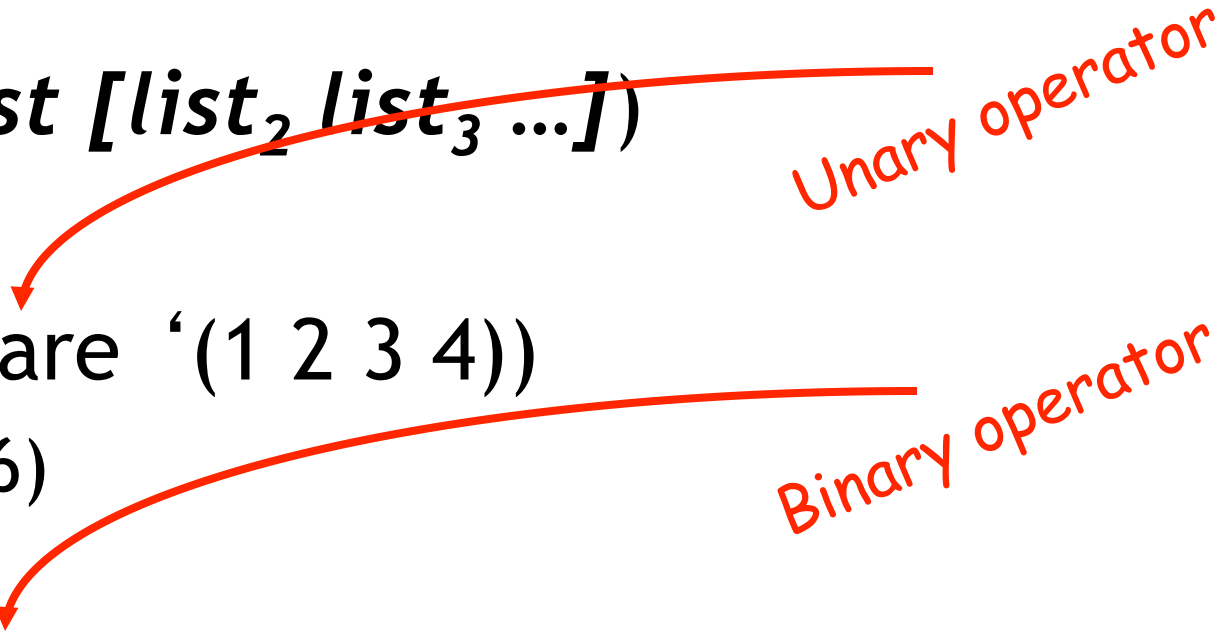
- Large-Scale Data Processing
  - Want to use 1000s of CPUs
    - But want it to be easy
- MapReduce provides
  - Automatic parallelization & distribution
  - Fault tolerance
  - I/O scheduling
  - Monitoring & status updates

Map/Reduce

- Map/Reduce
  - Programming model from Lisp
  - (and other functional languages)
- Many problems can be phrased this way
- Easy to distribute across nodes
- Nice retry/failure semantics

Map in Lisp (Scheme)

- (map ***f list*** ***[list$_2$ list$_3$ …]***)
- (map square ‘(1 2 3 4))
  - (1 4 9 16)
- (reduce + ‘(1 4 9 16))
  - 30
- (reduce + (map square (map – l_1 l_2))))



Programming Model

- Input and Output: each a set of key/value pairs
- Programmer specifies two functions:

Map (in_key, in_value) -> list (out_key, intermediate_value)

- Processes input key/value pair
- Produces set of intermediate pairs

Reduce (out_key, list (intermediate_value)) -> list (out_value)

- Combines all intermediate values for a particular key
- Produces a set of merged output values (usually just one)

count words in docs

- Input consists of (url, contents) pairs
- map(key=url, val=contents):
  - For each word *w* in contents, emit (w, "1")
- reduce(key=word, values=uniq_counts):
  - Sum all "1"s in values list
  - Emit result "(word, sum)"

Count Illustrated

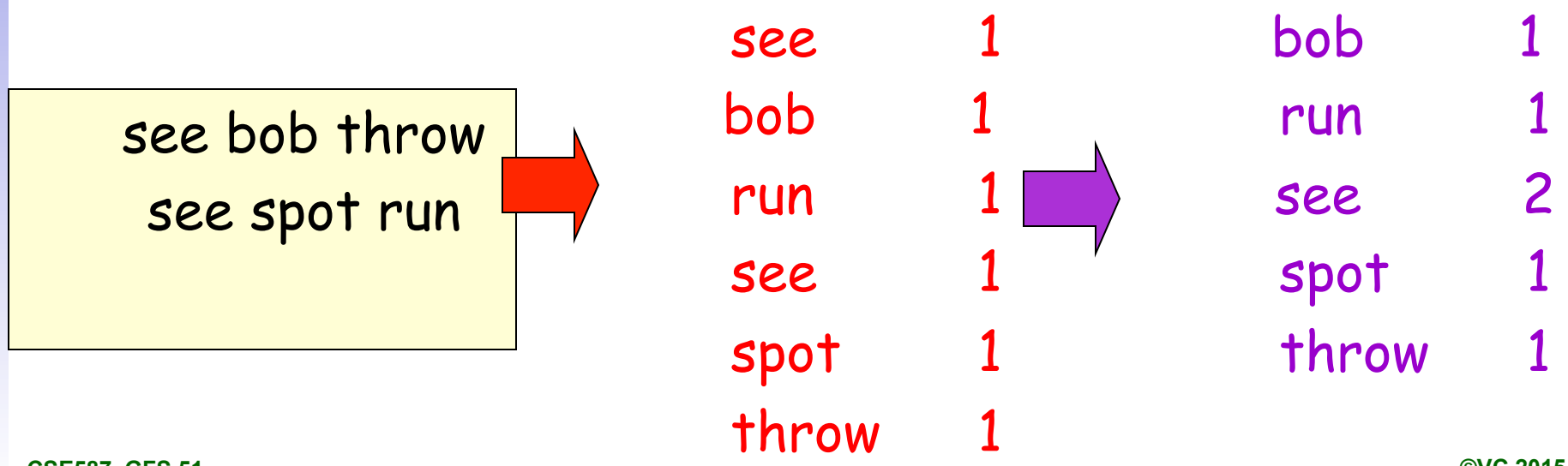
map(key=url, val=contents):

For each word *w* in contents, emit (w, "1")

reduce(key=word, values=uniq_counts):

Sum all "1"s in values list

Emit result "(word, sum)"



Grep

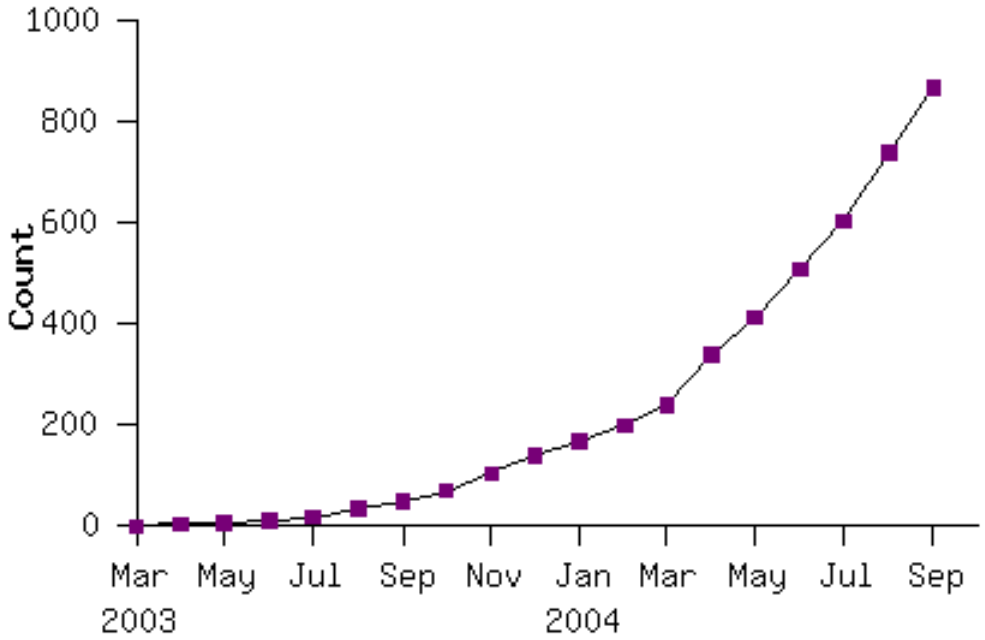
- Input consists of (url+offset, single line)
- map(key=url+offset, val=line):
  - If contents matches regexp, emit (line, “1”)
- reduce(key=line, values=uniq_counts):
  - Don’t do anything; just emit line

Reverse Web-Link Graph

- Map
  - For each URL linking to target, …
  - Output <target, source> pairs
- Reduce
  - Concatenate list of all source URLs
  - Outputs: <target, ***list*** (source)> pairs

Model is Widely Applicable

MapReduce Programs In Google Source Tree



Example uses:

distributed grep	distributed sort	web link-graph reversal
term-vector / host	web access log stats	inverted index construction
document clustering	machine learning	statistical machine translation
...	...	...

Implementation Overview

Typical cluster:

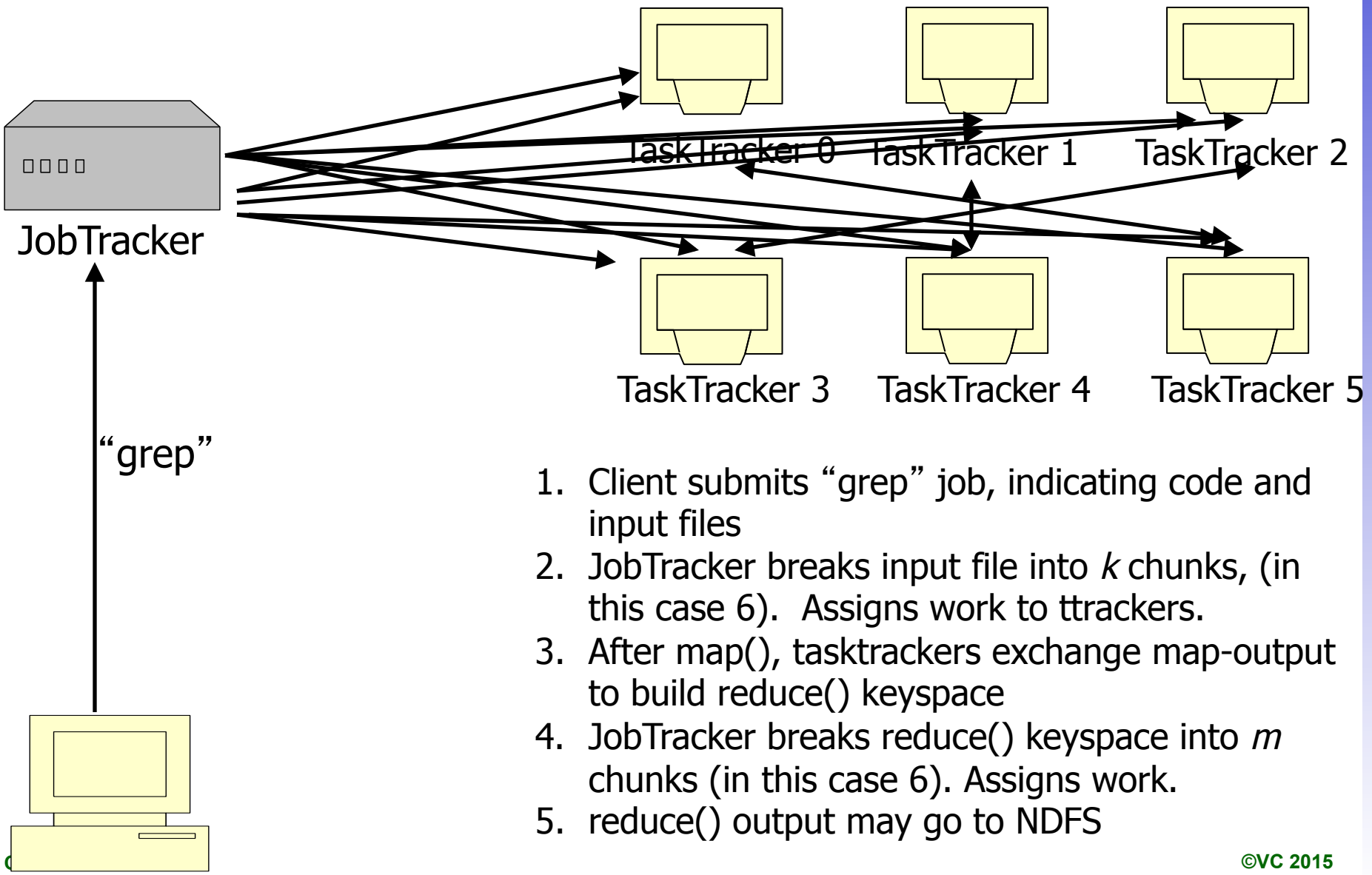
- 100s/1000s of 2-CPU x86 machines, 2-4 GB of memory
- Limited bisection bandwidth
- Storage is on local IDE disks
- GFS: distributed file system manages data (SOSP'03)
- Job scheduling system: jobs made up of tasks, scheduler assigns tasks to machines

Implementation is a C++ library linked into user programs

Execution

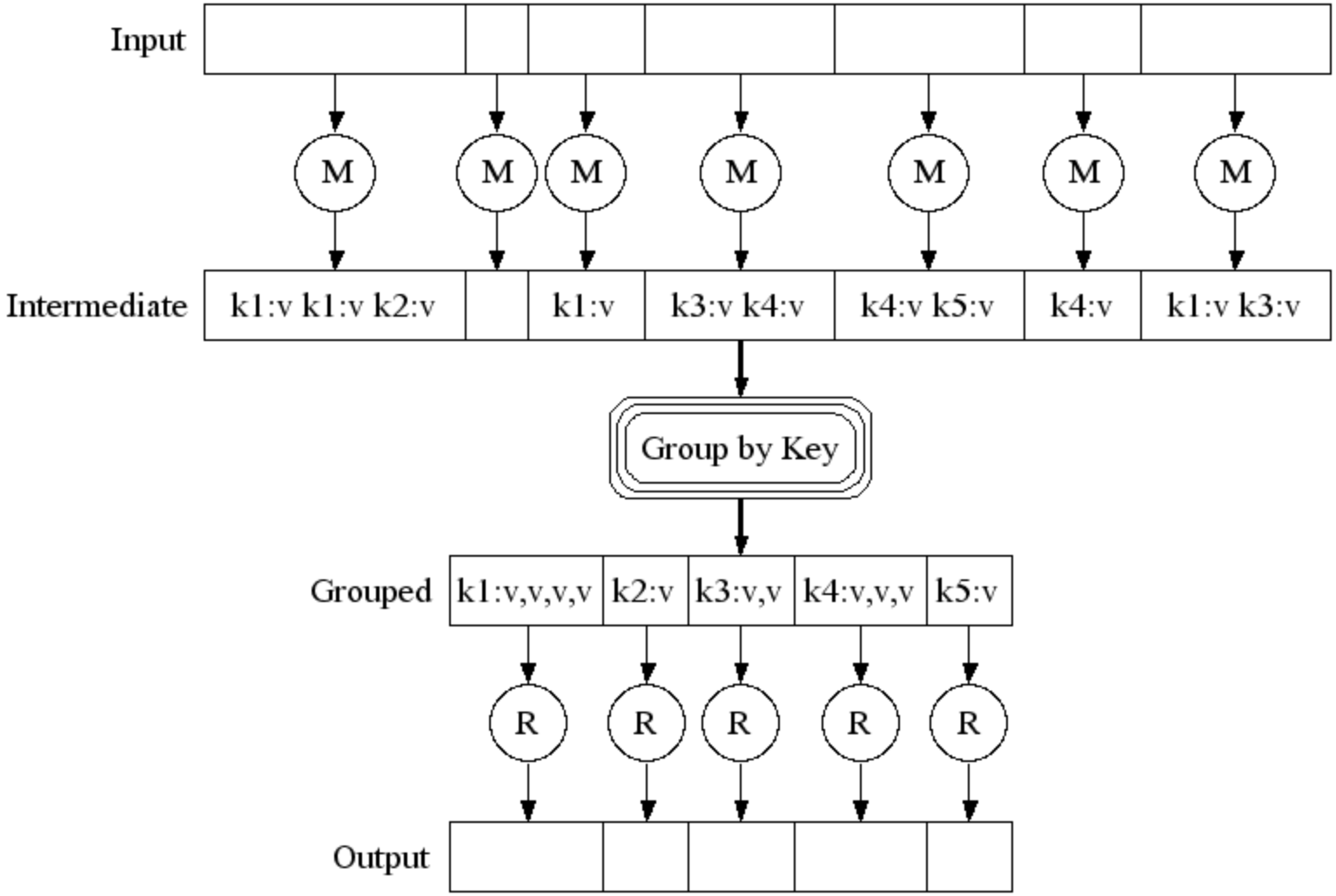
- How is this distributed?
  1. Partition input key/value pairs into chunks, run map() tasks in parallel
  2. After all map()s are complete, consolidate all emitted values for each unique emitted key
  3. Now partition space of output map keys, and run reduce() in parallel
- If map() or reduce() fails, reexecute!

Job Processing

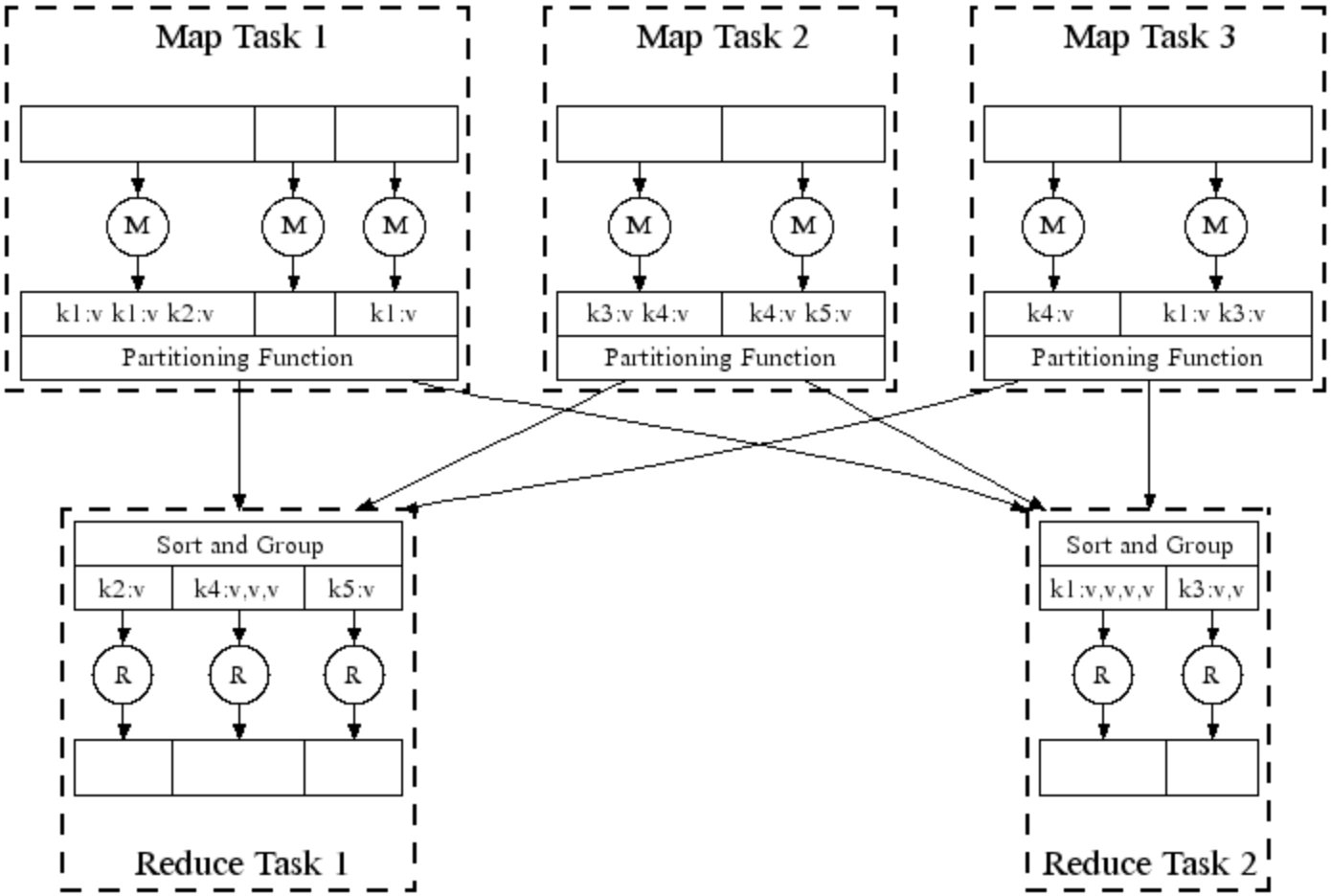


1. Client submits "grep" job, indicating code and input files
2. JobTracker breaks input file into *k* chunks, (in this case 6). Assigns work to ttrackers.
3. After map(), tasktrackers exchange map-output to build reduce() keyspace
4. JobTracker breaks reduce() keyspace into *m* chunks (in this case 6). Assigns work.
5. reduce() output may go to NDFS

Execution

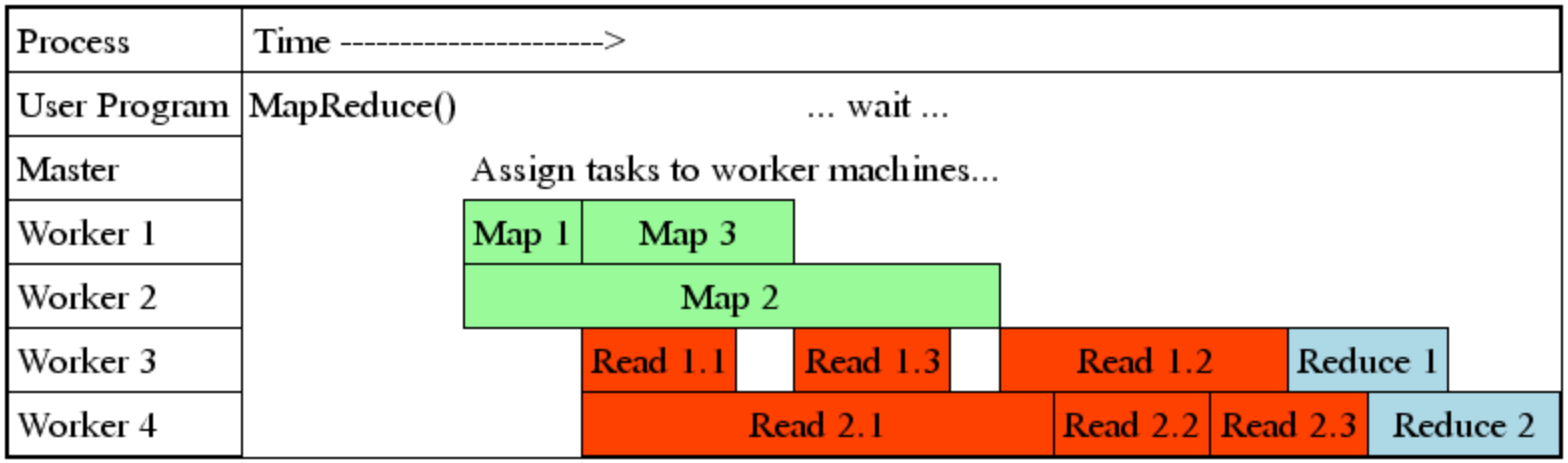


Parallel Execution



Task Granularity & Pipelining

- Fine granularity tasks: map tasks >> machines
  - Minimizes time for fault recovery
  - Can pipeline shuffling with map execution
  - Better dynamic load balancing
- Often use 200,000 map & 5000 reduce tasks
- Running on 2000 machines

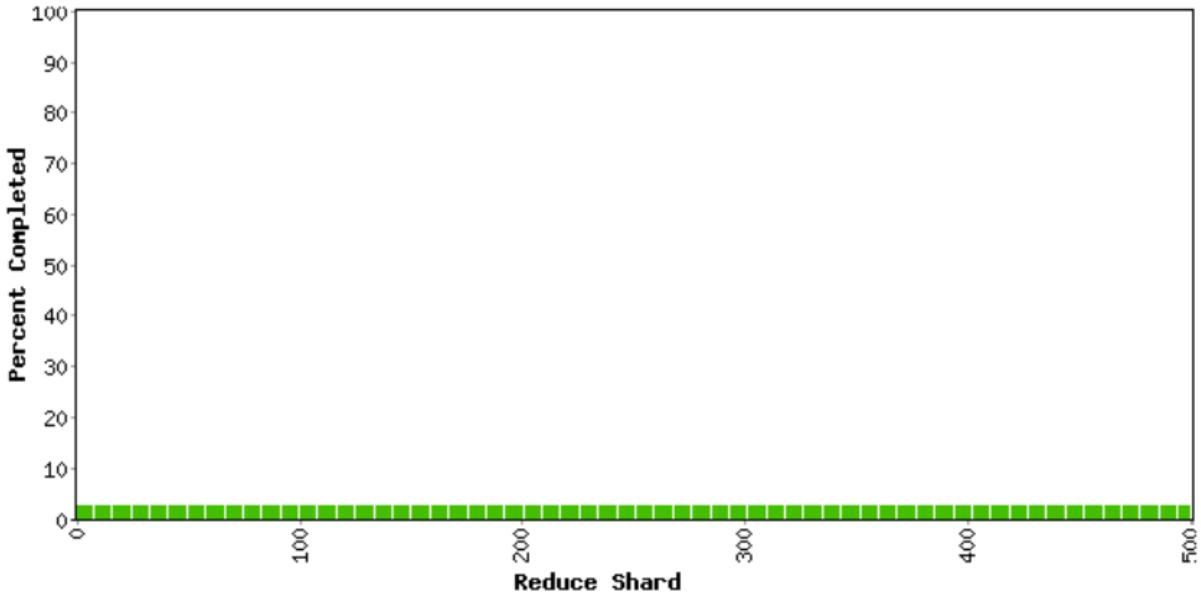


MapReduce status: MR_Indexer-beta6-large-2003_10_28_00_03

Started: Fri Nov 7 09:51:07 2003 -- up 0 hr 00 min 18 sec

323 workers; 0 deaths

Type	Shards	Done	Active	Input(MB)	Done(MB)	Output(MB)
Map	13853	0	323	878934.6	1314.4	717.0
Shuffle	500	0	323	717.0	0.0	0.0
Reduce	500	0	0	0.0	0.0	0.0



Counters

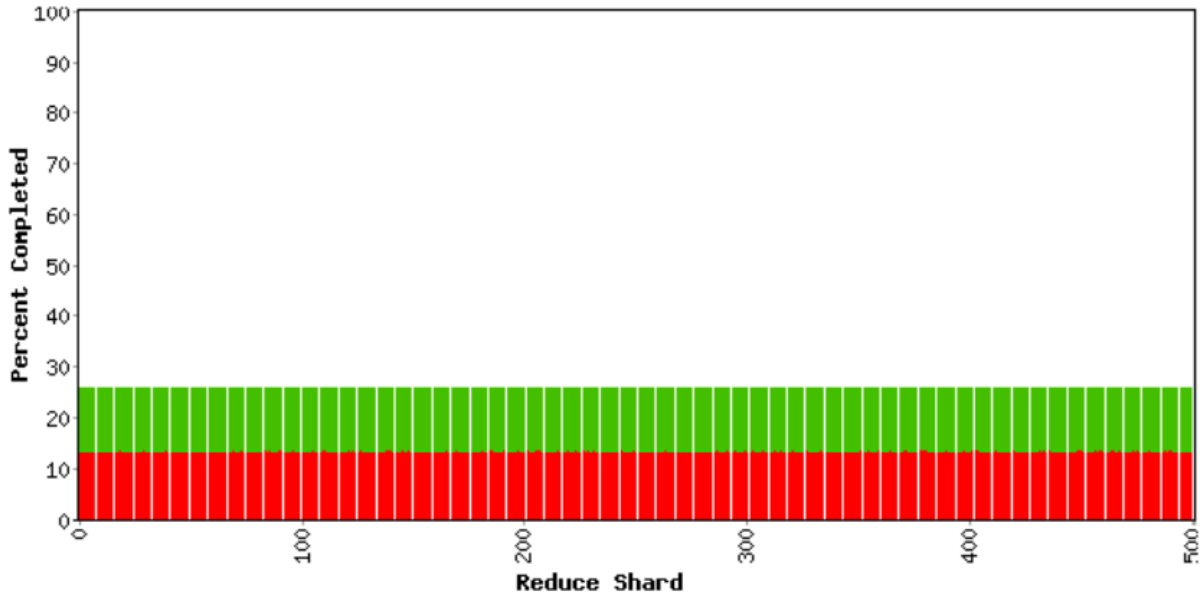
Variable	Minute
Mapped (MB/s)	72.5
Shuffle (MB/s)	0.0
Output (MB/s)	0.0
doc-index-hits	145825686
docs-indexed	506631
dups-in-index-merge	0
mr-operator-calls	508192
mr-operator-	506631

MapReduce status: MR_Indexer-beta6-large-2003_10_28_00_03

Started: Fri Nov 7 09:51:07 2003 -- up 0 hr 05 min 07 sec

1707 workers; 1 deaths

Type	Shards	Done	Active	Input(MB)	Done(MB)	Output(MB)
Map	13853	1857	1707	878934.6	191995.8	113936.6
Shuffle	500	0	500	113936.6	57113.7	57113.7
Reduce	500	0	0	57113.7	0.0	0.0



Counters

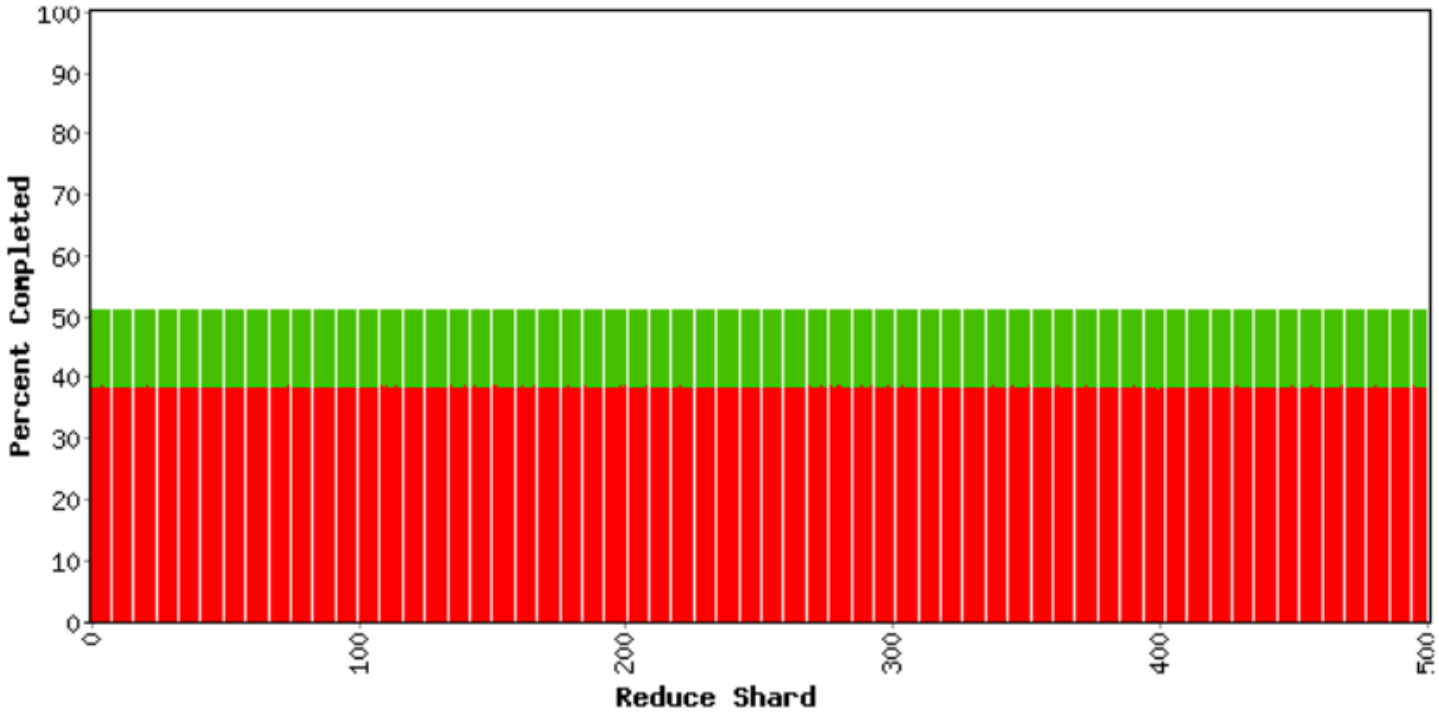
Variable	Minute
Mapped (MB/s)	699.1
Shuffle (MB/s)	349.5
Output (MB/s)	0.0
doc-index-hits	5004411944
docs-indexed	17290135
dups-in-index-merge	0
mr-operator-calls	17331371
mr-operator-outputs	17290135

MapReduce status: MR_Indexer-beta6-large-2003_10_28_00_03

Started: Fri Nov 7 09:51:07 2003 -- up 0 hr 10 min 18 sec

1707 workers; 1 deaths

Type	Shards	Done	Active	Input(MB)	Done(MB)	Output(MB)
Map	13853	5354	1707	878934.6	406020.1	241058.2
Shuffle	500	0	500	241058.2	196362.5	196362.5
Reduce	500	0	0	196362.5	0.0	0.0



Counters

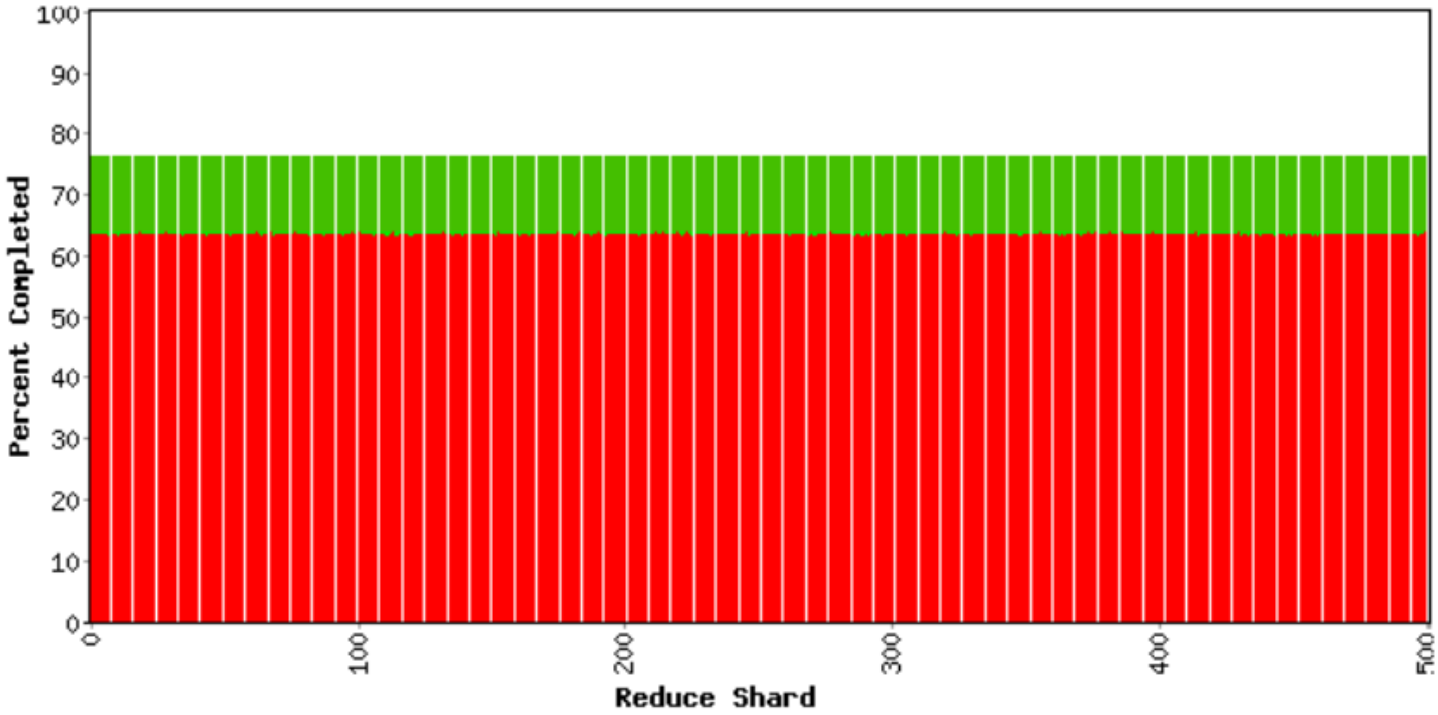
Variable	Minute
Mapped (MB/s)	704.4
Shuffle (MB/s)	371.9
Output (MB/s)	0.0
doc-index-hits	5000364228
docs-indexed	17300709
dups-in-index-merge	0
mr-operator-calls	17342493
mr-operator-outputs	17300709

MapReduce status: MR_Indexer-beta6-large-2003_10_28_00_03

Started: Fri Nov 7 09:51:07 2003 -- up 0 hr 15 min 31 sec

1707 workers; 1 deaths

Type	Shards	Done	Active	Input(MB)	Done(MB)	Output(MB)
Map	13853	8841	1707	878934.6	621608.5	369459.8
Shuffle	500	0	500	369459.8	326986.8	326986.8
Reduce	500	0	0	326986.8	0.0	0.0



Counters

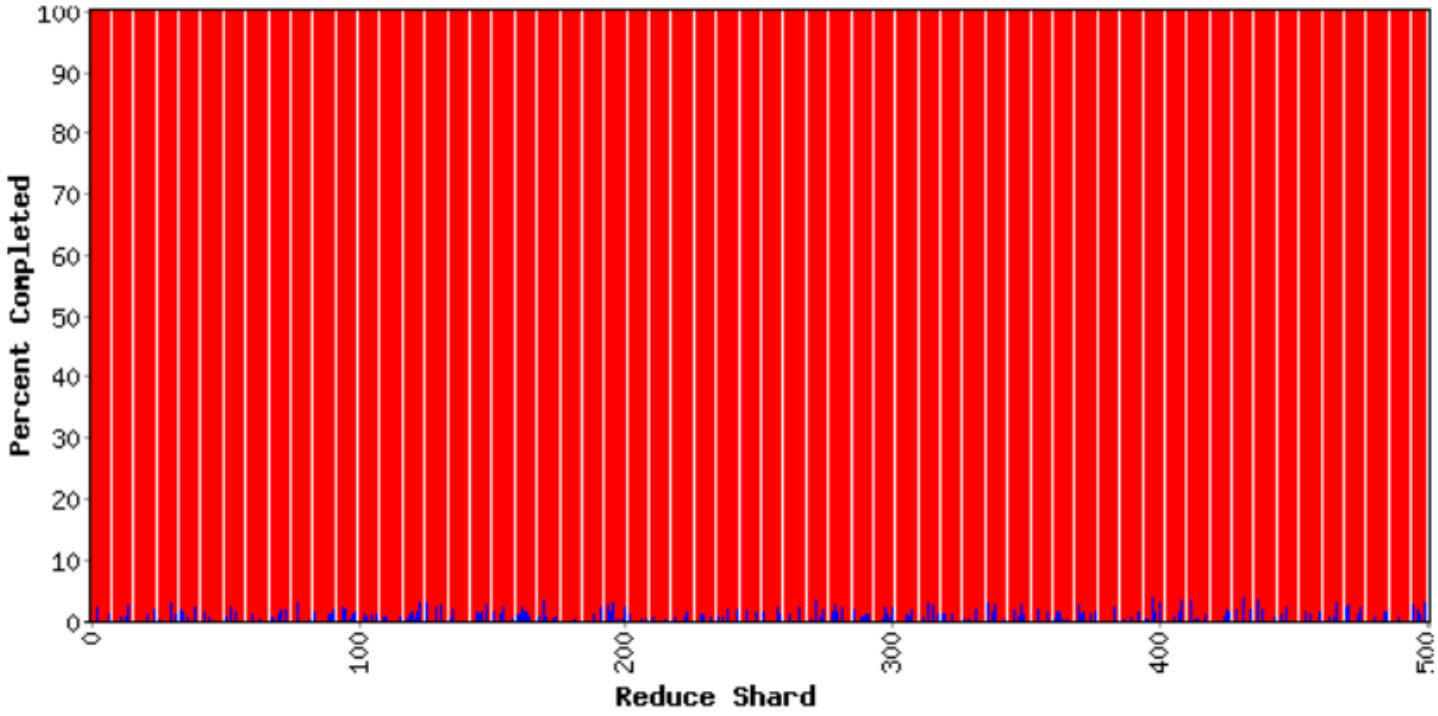
Variable	Minute
Mapped (MB/s)	706.5
Shuffle (MB/s)	419.2
Output (MB/s)	0.0
doc-index-hits	4982870667
docs-indexed	17229926
dups-in-index-merge	0
mr-operator-calls	17272056
mr-operator-outputs	17229926

MapReduce status: MR_Indexer-beta6-large-2003_10_28_00_03

Started: Fri Nov 7 09:51:07 2003 -- up 0 hr 29 min 45 sec

1707 workers; 1 deaths

Type	Shards	Done	Active	Input(MB)	Done(MB)	Output(MB)
Map	13853	13853	0	878934.6	878934.6	523499.2
Shuffle	500	195	305	523499.2	523389.6	523389.6
Reduce	500	0	195	523389.6	2685.2	2742.6



Counters

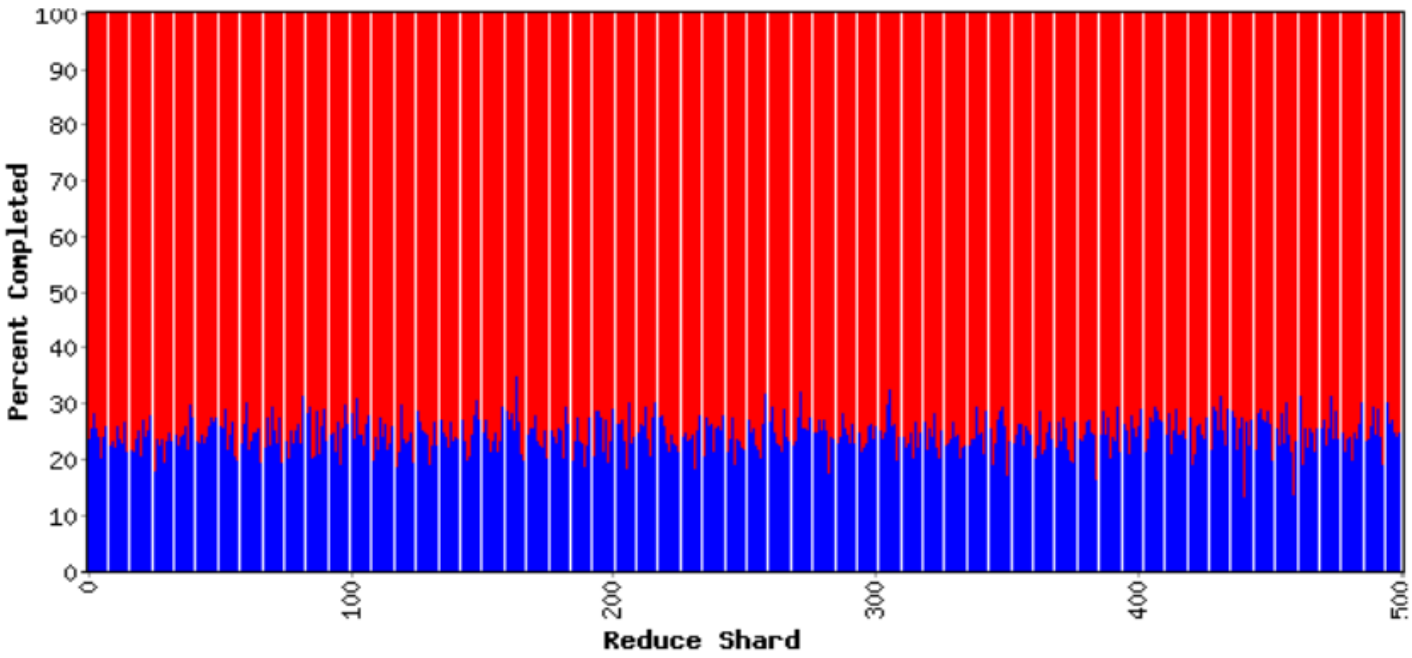
Variable	Minute	
Mapped (MB/s)	0.3	
Shuffle (MB/s)	0.5	
Output (MB/s)	45.7	
doc-index-hits	2313178	105
docs-indexed	7936	
dups-in-index-merge	0	
mr-merge-calls	1954105	
mr-merge-outputs	1954105	

MapReduce status: MR_Indexer-beta6-large-2003_10_28_00_03

Started: Fri Nov 7 09:51:07 2003 -- up 0 hr 31 min 34 sec

1707 workers; 1 deaths

Type	Shards	Done	Active	Input(MB)	Done(MB)	Output(MB)
Map	13853	13853	0	878934.6	878934.6	523499.2
Shuffle	500	500	0	523499.2	523499.5	523499.5
Reduce	500	0	500	523499.5	133837.8	136929.6



Counters

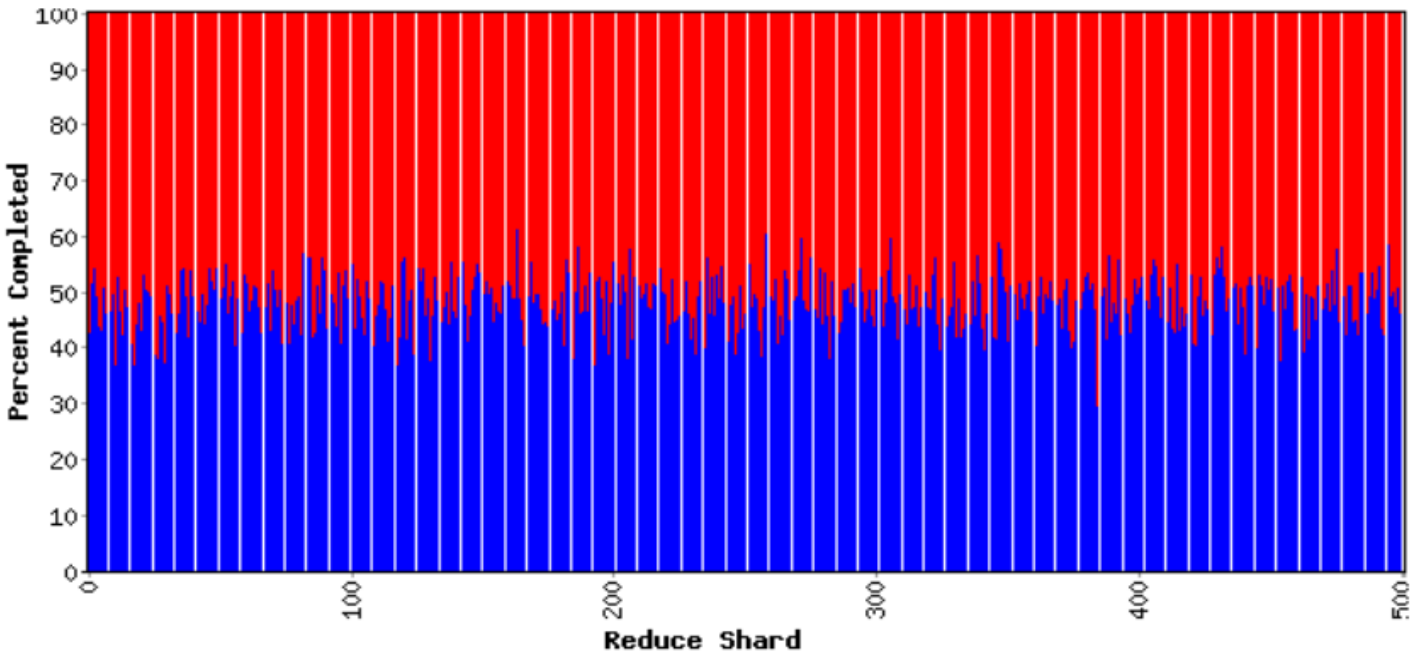
Variable	Minute	
Mapped (MB/s)	0.0	
Shuffle (MB/s)	0.1	
Output (MB/s)	1238.8	
doc-index-hits	0	1(
docs-indexed	0	
dups-in-index-merge	0	
mr-merge-calls	51738599	
mr-merge-outputs	51738599	

MapReduce status: MR_Indexer-beta6-large-2003_10_28_00_03

Started: Fri Nov 7 09:51:07 2003 -- up 0 hr 33 min 22 sec

1707 workers; 1 deaths

Type	Shards	Done	Active	Input(MB)	Done(MB)	Output(MB)
Map	13853	13853	0	878934.6	878934.6	523499.2
Shuffle	500	500	0	523499.2	523499.5	523499.5
Reduce	500	0	500	523499.5	263283.3	269351.2



Counters

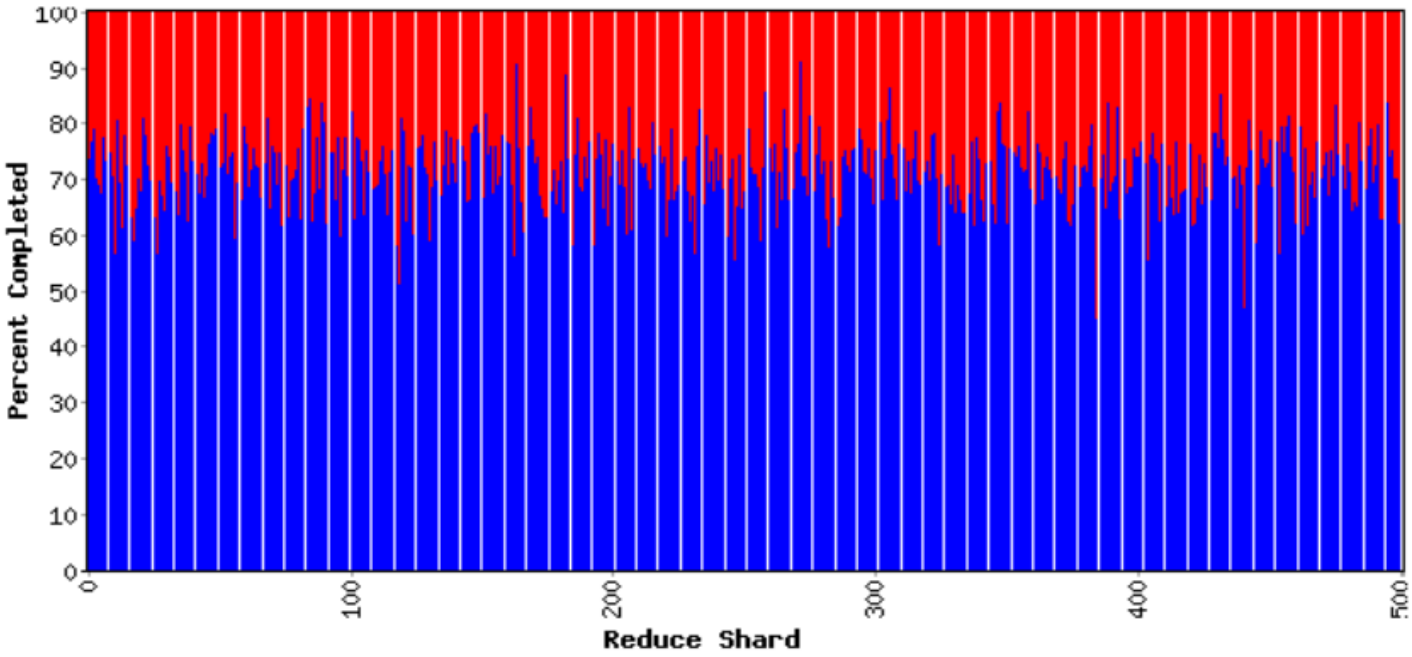
Variable	Minute	
Mapped (MB/s)	0.0	
Shuffle (MB/s)	0.0	
Output (MB/s)	1225.1	
doc-index-hits	0	1(
docs-indexed	0	
dups-in-index-merge	0	
mr-merge-calls	51842100	
mr-merge-outputs	51842100	

MapReduce status: MR_Indexer-beta6-large-2003_10_28_00_03

Started: Fri Nov 7 09:51:07 2003 -- up 0 hr 35 min 08 sec

1707 workers; 1 deaths

Type	Shards	Done	Active	Input(MB)	Done(MB)	Output(MB)
Map	13853	13853	0	878934.6	878934.6	523499.2
Shuffle	500	500	0	523499.2	523499.5	523499.5
Reduce	500	0	500	523499.5	390447.6	399457.2



Counters

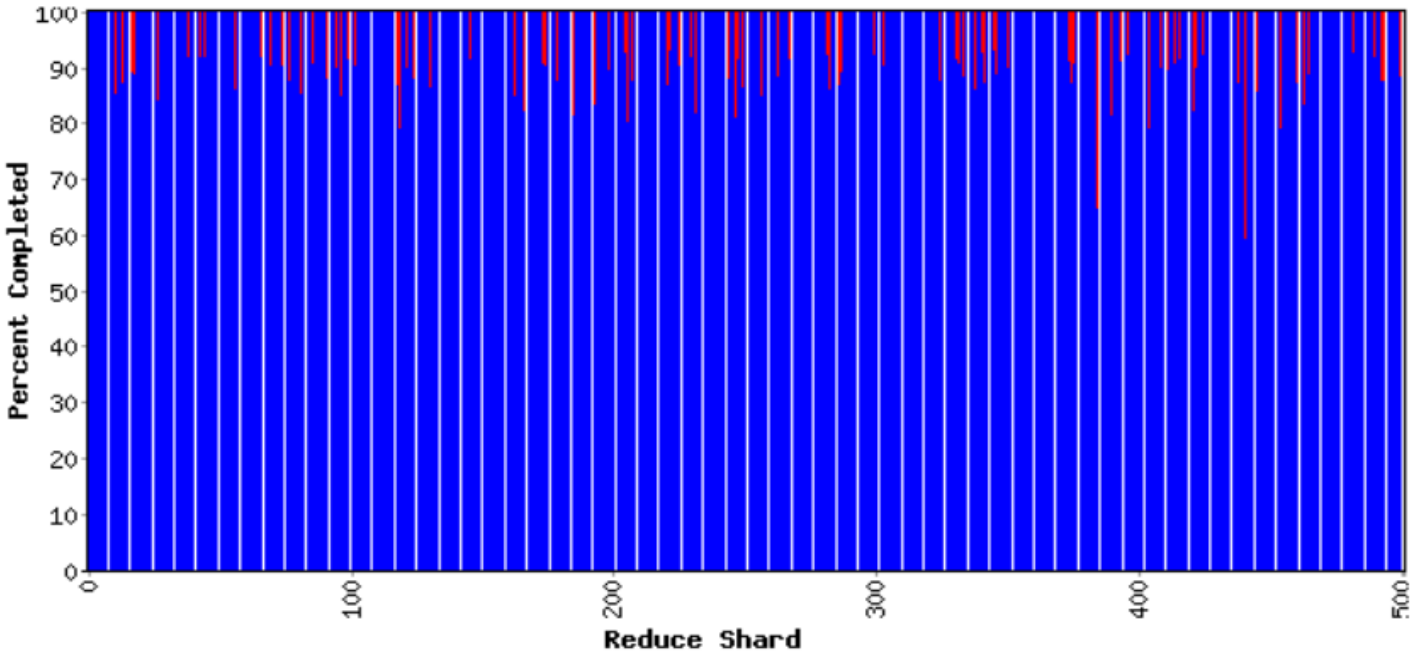
Variable	Minute	
Mapped (MB/s)	0.0	
Shuffle (MB/s)	0.0	
Output (MB/s)	1222.0	
doc-index-hits	0	1(
docs-indexed	0	
dups-in-index-merge	0	
mr-merge-calls	51640600	
mr-merge-outputs	51640600	

MapReduce status: MR_Indexer-beta6-large-2003_10_28_00_03

Started: Fri Nov 7 09:51:07 2003 -- up 0 hr 37 min 01 sec

1707 workers; 1 deaths

Type	Shards	Done	Active	Input(MB)	Done(MB)	Output(MB)
Map	13853	13853	0	878934.6	878934.6	523499.2
Shuffle	500	500	0	523499.2	520468.6	520468.6
Reduce	500	406	94	520468.6	512265.2	514373.3



Counters

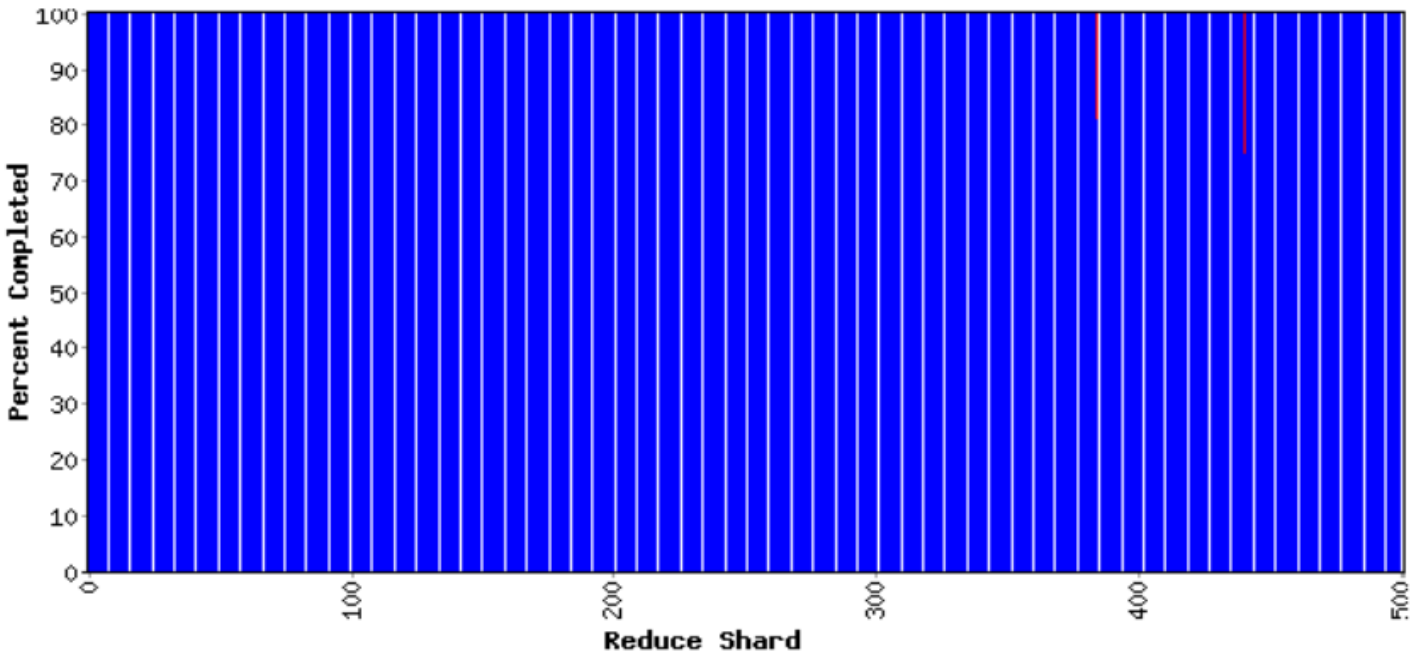
Variable	Minute	
Mapped (MB/s)	0.0	
Shuffle (MB/s)	0.0	
Output (MB/s)	849.5	
doc-index-hits	0	1(
docs-indexed	0	
dups-in-index-merge	0	
mr-merge-calls	35083350	
mr-merge-outputs	35083350	

MapReduce status: MR_Indexer-beta6-large-2003_10_28_00_03

Started: Fri Nov 7 09:51:07 2003 -- up 0 hr 38 min 56 sec

1707 workers; 1 deaths

Type	Shards	Done	Active	Input(MB)	Done(MB)	Output(MB)
Map	13853	13853	0	878934.6	878934.6	523499.2
Shuffle	500	500	0	523499.2	519781.8	519781.8
Reduce	500	498	2	519781.8	519394.7	519440.7



Counters

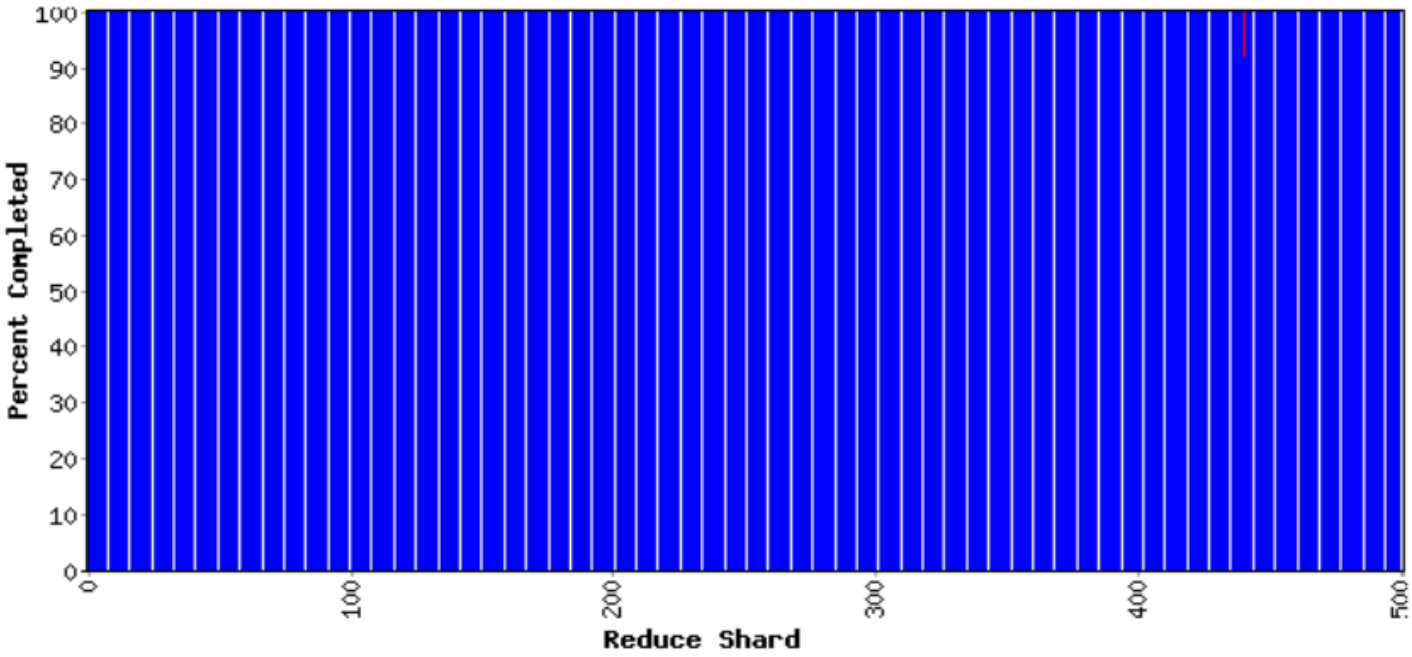
Variable	Minute	
Mapped (MB/s)	0.0	
Shuffle (MB/s)	0.0	
Output (MB/s)	9.4	
doc-index-hits	0	1056
docs-indexed	0	3
dups-in-index-merge	0	
mr-merge-calls	394792	3
mr-merge-outputs	394792	3

MapReduce status: MR_Indexer-beta6-large-2003_10_28_00_03

Started: Fri Nov 7 09:51:07 2003 -- up 0 hr 40 min 43 sec

1707 workers; 1 deaths

Type	Shards	Done	Active	Input(MB)	Done(MB)	Output(MB)
Map	13853	13853	0	878934.6	878934.6	523499.2
Shuffle	500	500	0	523499.2	519774.3	519774.3
Reduce	500	499	1	519774.3	519735.2	519764.0



Counters

Variable	Minute	
Mapped (MB/s)	0.0	
Shuffle (MB/s)	0.0	
Output (MB/s)	1.9	
doc-index-hits	0	105
docs-indexed	0	
dups-in-index-merge	0	
mr-merge-calls	73442	
mr-merge-outputs	73442	

Fault Tolerance / Workers

Handled via re-execution

- Detect failure via periodic heartbeats
- Re-execute completed + in-progress *map* tasks
- Re-execute in progress *reduce* tasks
- Task completion committed through master
-

Robust: lost 1600/1800 machines once → finished ok

Semantics in presence of failures: see paper

Master Failure

- Could handle, … ?
- But don't yet
  - (master failure unlikely)

Refinement: Redundant Execution

Slow workers significantly delay completion time

- Other jobs consuming resources on machine
- Bad disks w/ soft errors transfer data slowly
- Weird things: processor caches disabled (!!)

Solution: Near end of phase, spawn backup tasks

- Whichever one finishes first "wins"

Dramatically shortens job completion time

Refinement: Locality Optimization

- Master scheduling policy:
  - Asks GFS for locations of replicas of input file blocks
  - Map tasks typically split into 64MB (GFS block size)
  - Map tasks scheduled so GFS input block replica are on same machine or same rack
- Effect
  - Thousands of machines read input at local disk speed
    - Without this, rack switches limit read rate

Refinement
Skipping Bad Records

- Map/Reduce functions sometimes fail for particular inputs
  - Best solution is to debug & fix
    - Not always possible ~ third-party source libraries
  - On segmentation fault:
    - Send UDP packet to master from signal handler
    - Include sequence number of record being processed
  - If master sees two failures for same record:
    - Next worker is told to skip the record

Other Refinements

- Sorting guarantees
  - within each reduce partition
- Compression of intermediate data
- Combiner
  - Useful for saving network bandwidth
- Local execution for debugging/testing
- User-defined counters

Performance

Tests run on cluster of 1800 machines:

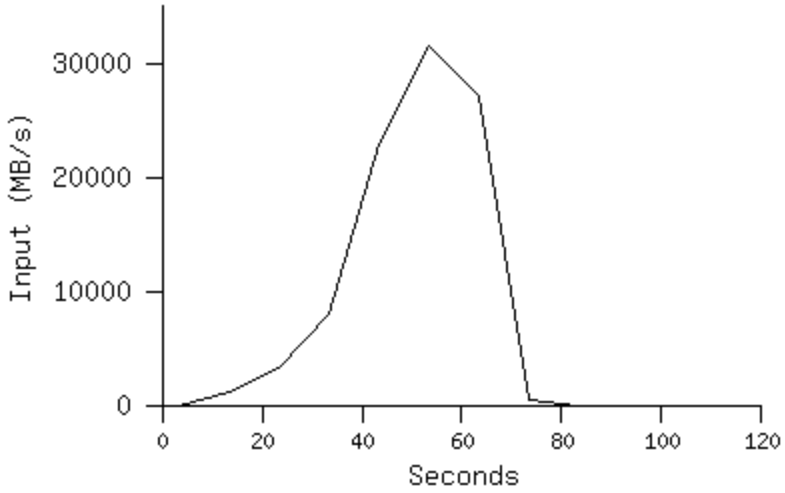
- 4 GB of memory
- Dual-processor 2 GHz Xeons with Hyperthreading
- Dual 160 GB IDE disks
- Gigabit Ethernet per machine
- Bisection bandwidth approximately 100 Gbps

Two benchmarks:

MR_Grep — 10^{10} 100-byte records to extract records matching a rare pattern (92K matching records)

MR_Sort — 10^{10} 100-byte records (modeled after TeraSort benchmark)

MR_Grep



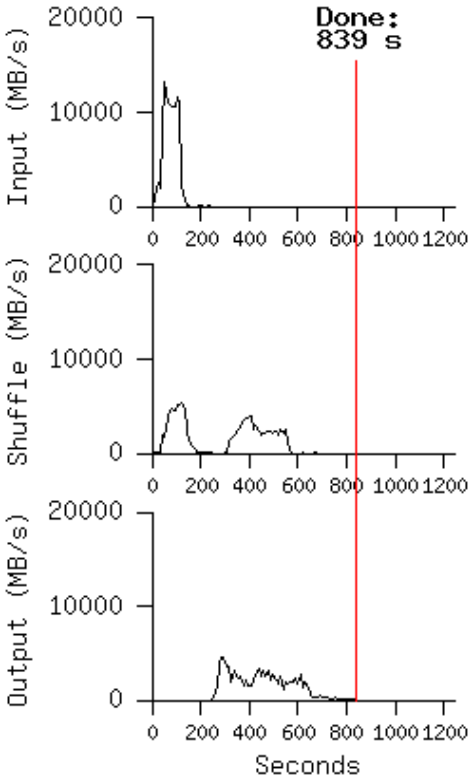
Locality optimization helps:

- 1800 machines read 1 TB at peak ~31 GB/s
- W/out this, rack switches would limit to 10 GB/s

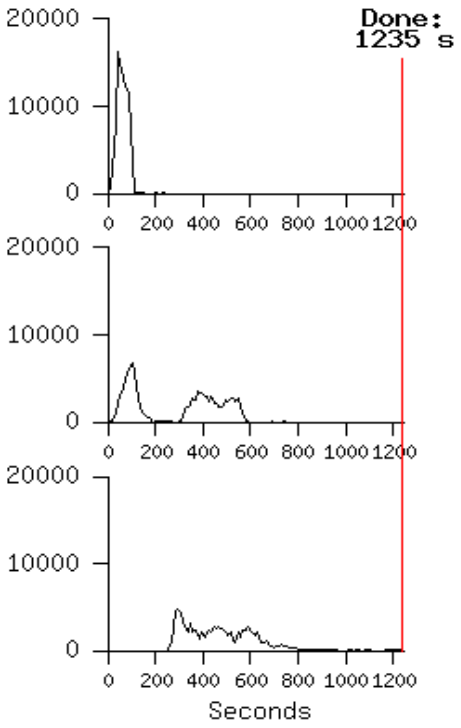
Startup overhead is significant for short jobs

MR_Sort

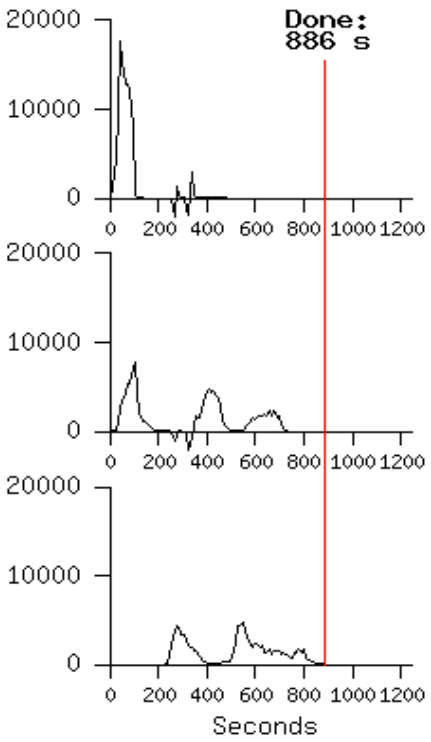
Normal



No backup tasks



200 processes killed



- Backup tasks reduce job completion time a lot!
- System deals well with failures

Experience

Rewrote Google's production indexing System using MapReduce

- Set of 10, 14, 17, 21, 24 MapReduce operations
- New code is simpler, easier to understand
  - 3800 lines C++ → 700

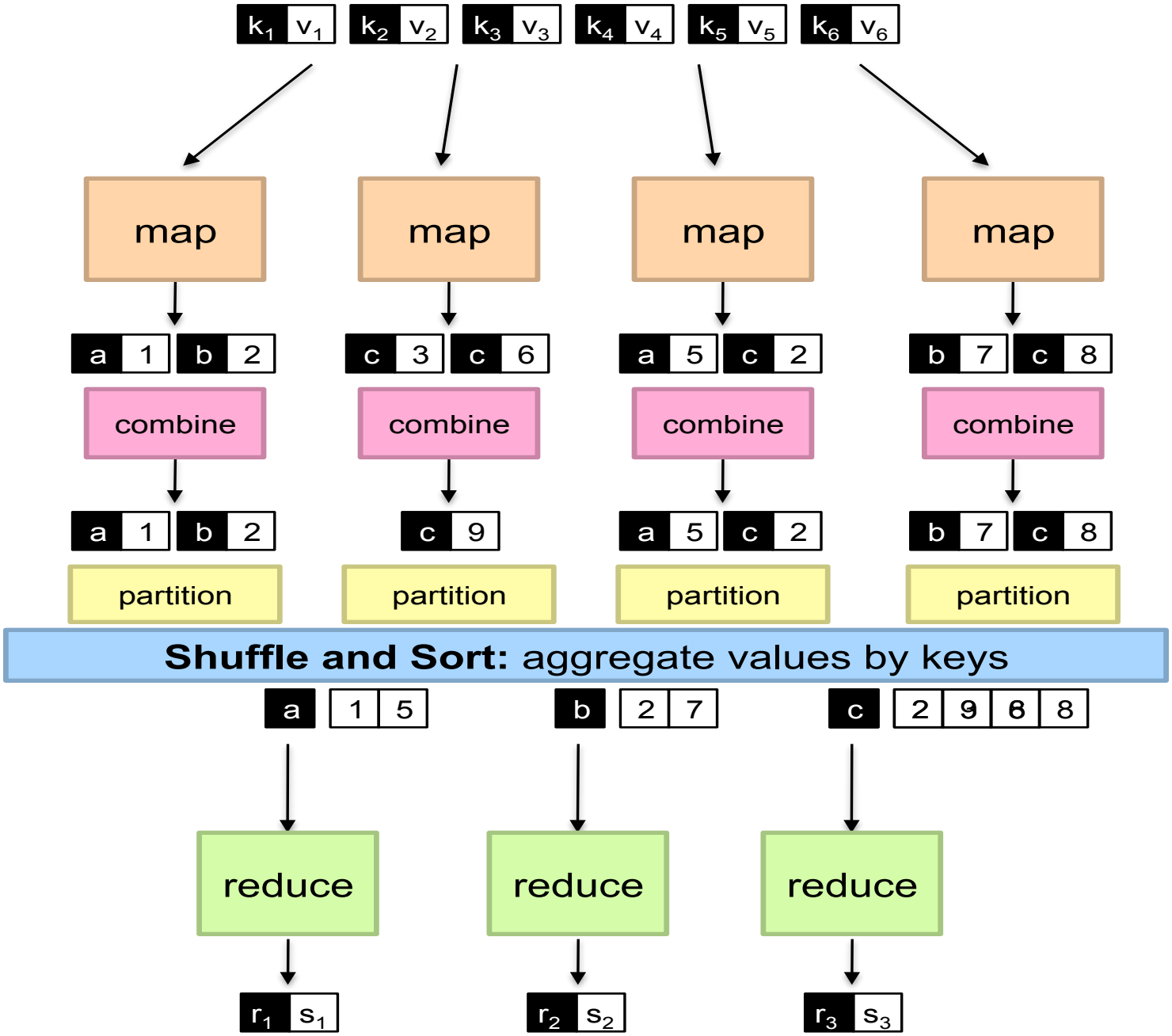
- MapReduce handles failures, slow machines
- Easy to make indexing faster by adding more machines

Usage in Aug 2004

Number of jobs	29,423	
Average job completion time	634	secs
Machine days used	79,186	days
Input data read	3,288	TB
Intermediate data produced	758	TB
Output data written	193	TB
Average worker machines per job	157	
Average worker deaths per job	1.2	
Average map tasks per job	3,351	
Average reduce tasks per job	55	
Unique *map* implementations	395	
Unique *reduce* implementations	269	
Unique *map*/*reduce* combinations	426	

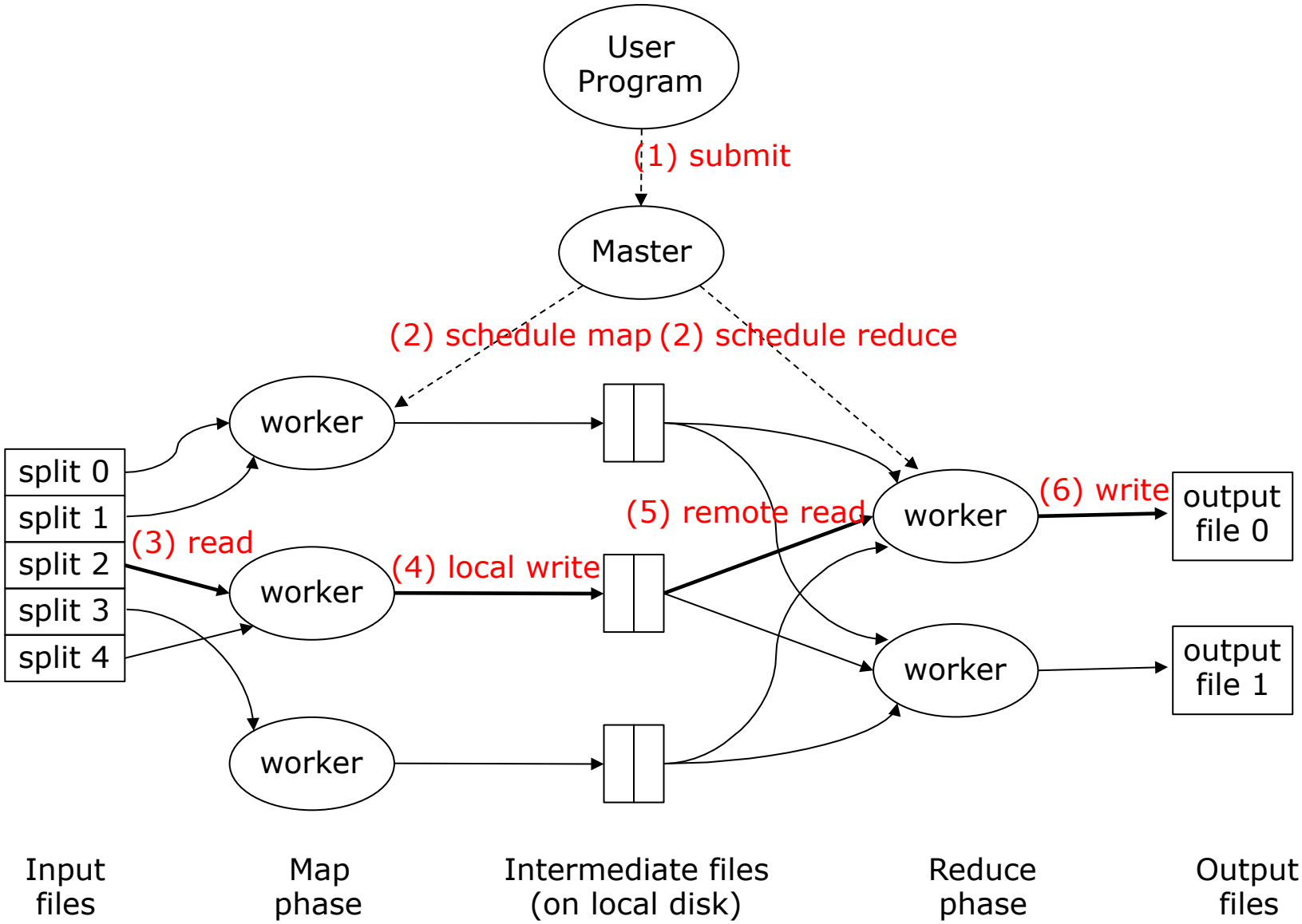
Related Work

- Programming model inspired by functional language primitives
- Partitioning/shuffling similar to many large-scale sorting systems
  - NOW-Sort ['97]
- Re-execution for fault tolerance
  - BAD-FS ['04] and TACC ['97]
- Locality optimization has parallels with Active Disks/Diamond work
  - Active Disks ['01], Diamond ['04]
- Backup tasks similar to Eager Scheduling in Charlotte system
  - Charlotte ['96]
- Dynamic load balancing solves similar problem as River's distributed queues
  - River ['99]


k1 v1
k2 v2
k3 v3
k4 v4
k5 v5
k6 v6
map
map
map
map
a 1
b 2
c 3
c 6
a 5
c 2
b 7
c 8
combine
combine
combine
combine
a 1
b 2
c 9
a 5
c 2
b 7
c 8
partition
partition
partition
partition
Shuffle and Sort: aggregate values by keys
a 1 5
b 2 7
c 2 9 6 8
reduce
reduce
reduce
r1 s1
r2 s2
r3 s3

MapReduce Implementations

- Google has a proprietary implementation in C++
  - Bindings in Java, Python
- Hadoop is an open-source implementation in Java
  - Development led by Yahoo, now an Apache project
  - Used in production at Yahoo, Facebook, Twitter, LinkedIn, Netflix, …
  - The *de facto* big data processing platform
  - Rapidly expanding software ecosystem
- Lots of custom research implementations
  - For GPUs, cell processors, etc.



Adapted from (Dean and Ghemawat, OSDI 2004)

Conclusions

- MapReduce proven to be useful abstraction
- Greatly simplifies large-scale computations
- Fun to use:
  - focus on problem,
  - let library deal w/ messy details