MapReduce:
Simplified Data Processing on Large Clusters

Jeff Dean, Sanjay Ghemawat
Google, Inc.

Motivation: Large Scale Data Processing

Many tasks: Process lots of data to produce other data

Want to use hundreds or thousands of CPUs

- ... but this needs to be easy

MapReduce provides:

- Automatic parallelization and distribution
- Fault-tolerance
- I/O scheduling
- Status and monitoring

Programming model

Input & Output: each a set of key/value pairs

Programmer specifies two functions:

```
map (in_key, in_value) -> list(out_key,
intermediate_value)
```

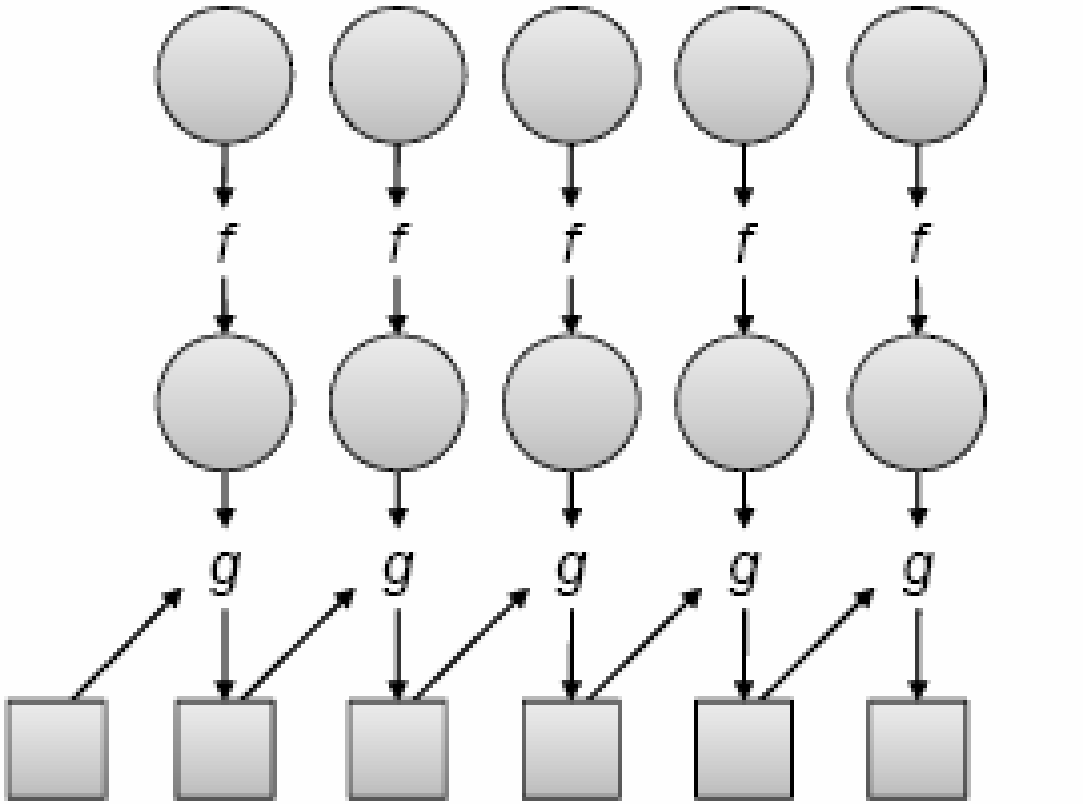
- Processes input key/value pair
- Produces set of intermediate pairs

```
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)

Inspired by similar primitives in LISP and other languages

Map y Fold



Data-Intensive Text Processing with MapReduce

Jimmy Lin and Chris Dyer. University of Maryland

Example: Count word occurrences

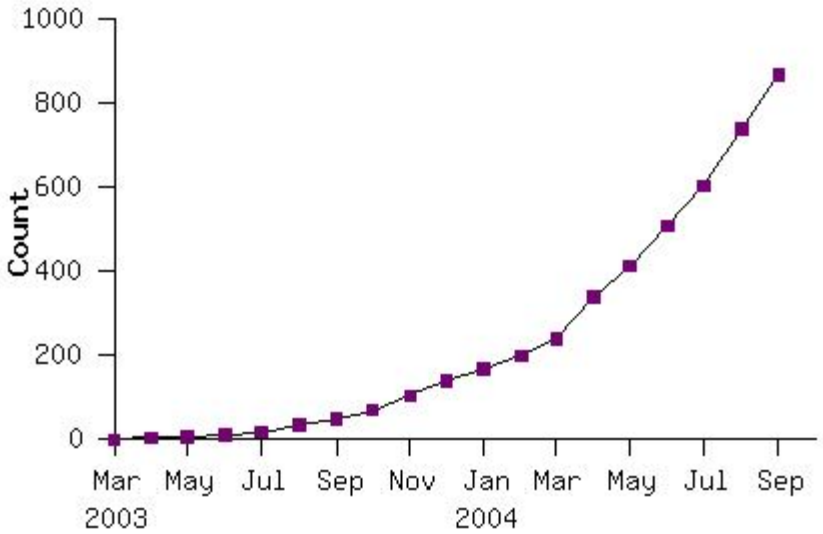
```
map(String input_key, String input_value):
  // input_key: document name
  // input_value: document contents
  for each word w in input_value:
    EmitIntermediate(w, "1");


reduce(String output_key, Iterator intermediate_values):
  // output_key: a word
  // output_values: a list of counts
  int result = 0;
  for each v in intermediate_values:
    result += ParseInt(v);
  Emit(AsString(result));
```

Pseudocode: See appendix in paper for real code

Model is Widely Applicable

MapReduce Programs In Google Source Tree



Example uses:

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

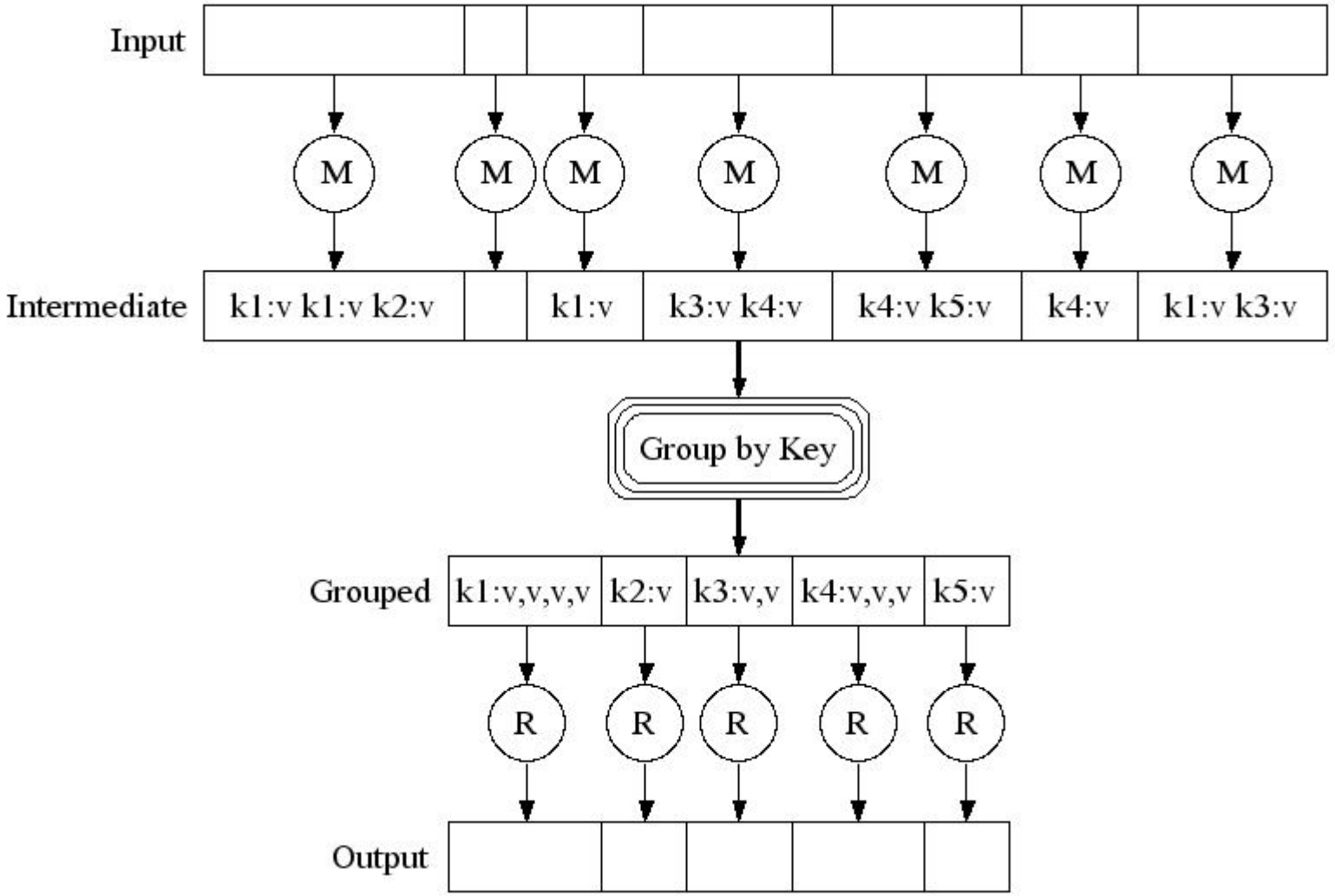
Implementation Overview

Typical cluster:

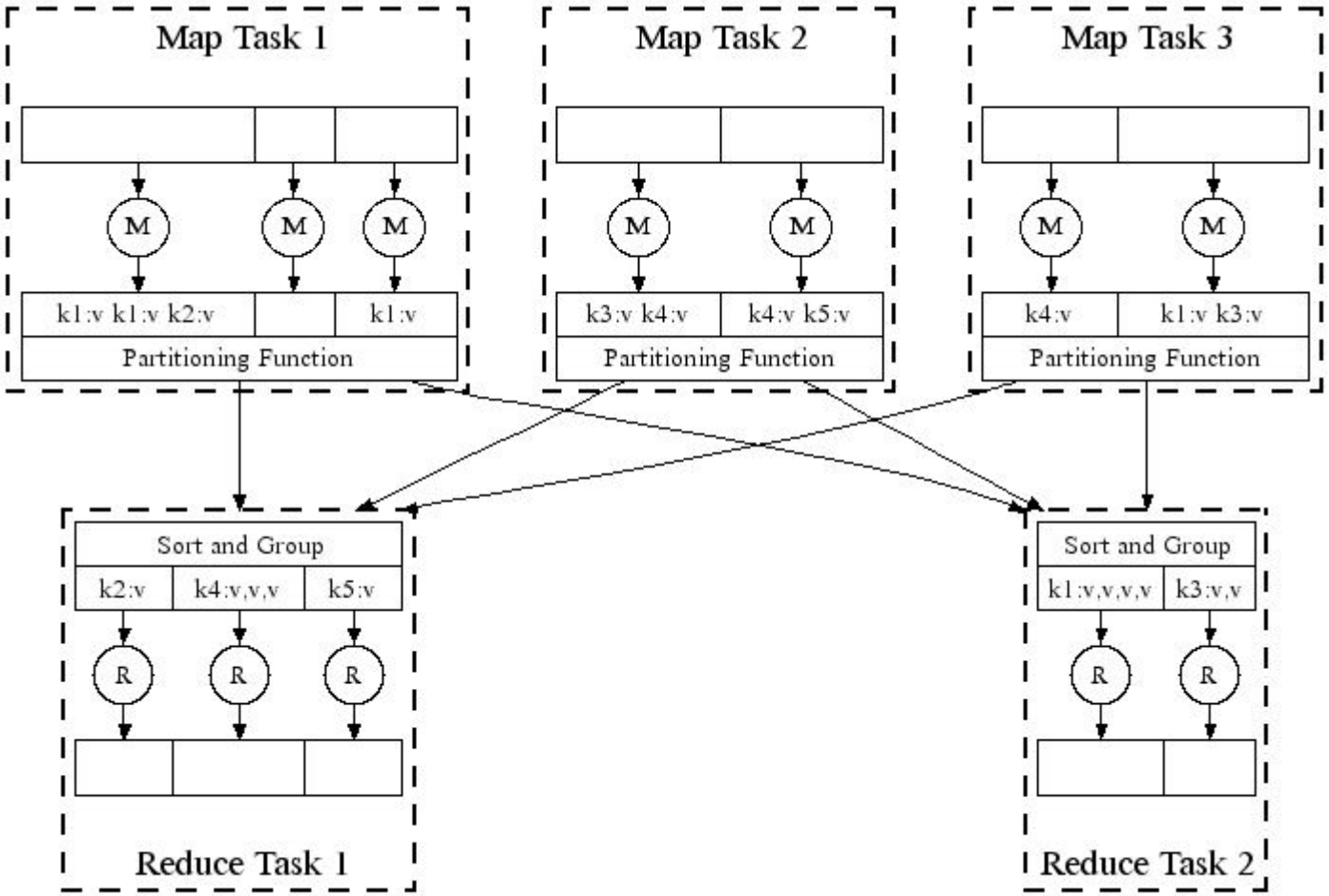
- 100s/1000s of 2-CPU x86 machines, 2-4 GB of memory
- Limited bisection bandwidth
- Storage is on local IDE disks
- GFS: distributed file system manages data (SOSP'03)
- Job scheduling system: jobs made up of tasks, scheduler assigns tasks to machines

Implementation is a C++ library linked into user programs

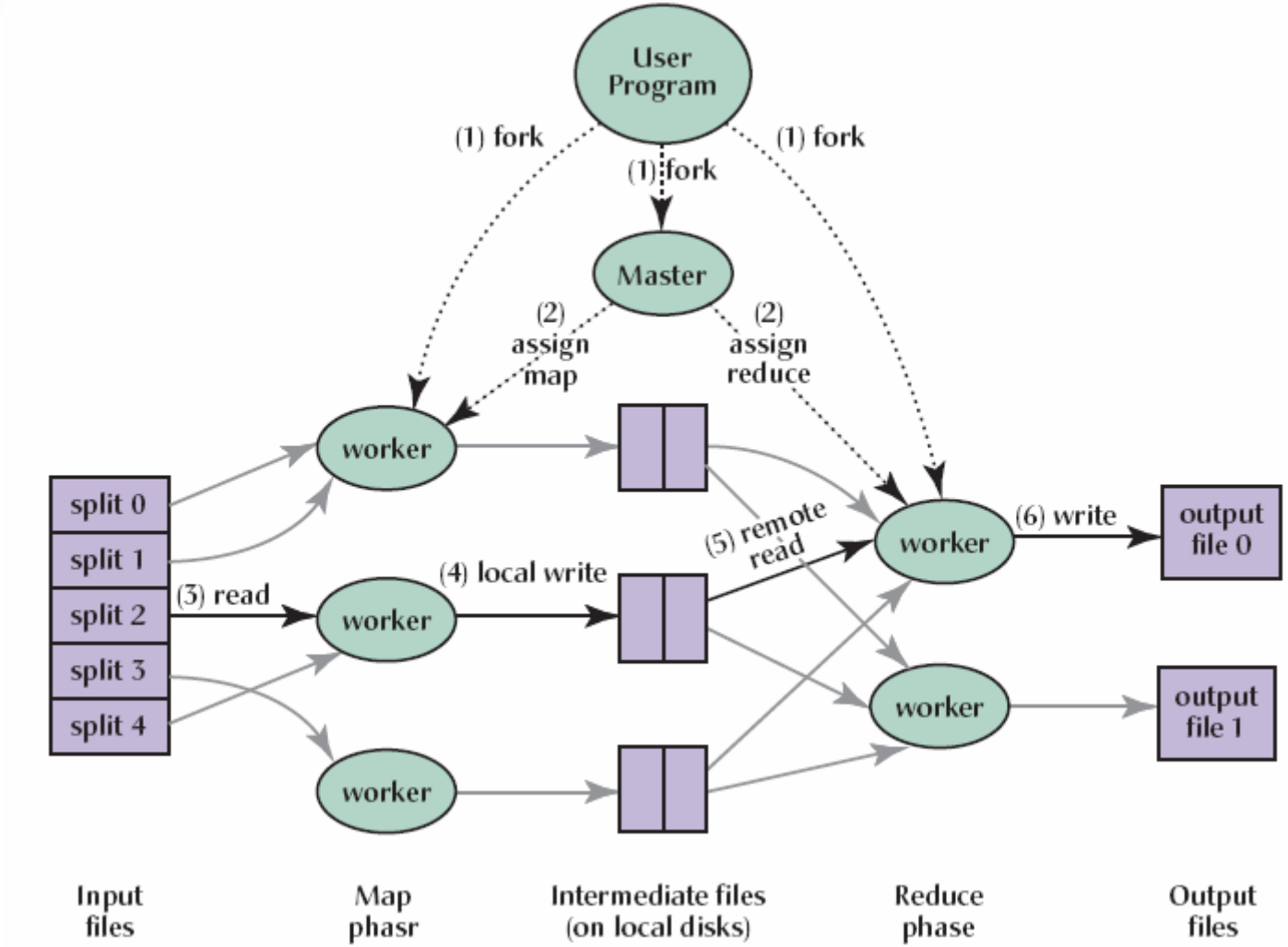
Execution


Input
M
M
M
M
M
M
M
Intermediate
k1:v k1:v k2:v
k1:v
k3:v k4:v
k4:v k5:v
k4:v
k1:v k3:v
Group by Key
Grouped
k1:v,v,v,v
k2:v
k3:v,v
k4:v,v,v
k5:v
R
R
R
R
R
Output

Parallel Execution



Visión física de ejecución Google-MR

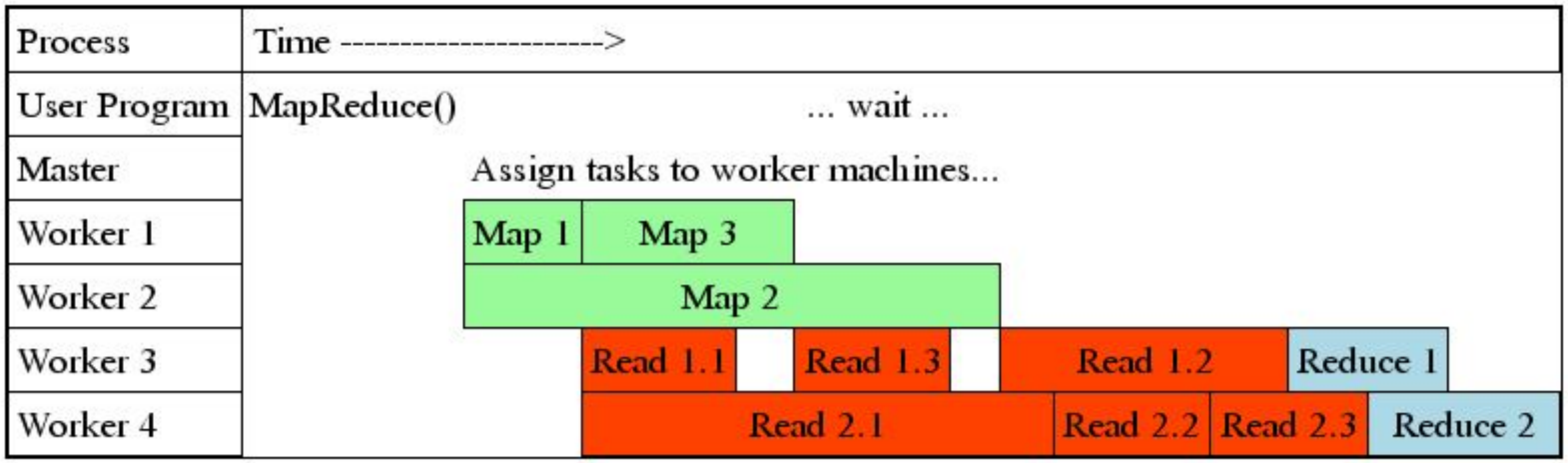


Task Granularity And Pipelining

Fine granularity tasks: many more map tasks than machines

- Minimizes time for fault recovery
- Can pipeline shuffling with map execution
- Better dynamic load balancing

Often use 200,000 map/5000 reduce tasks w/ 2000 machines



Fault tolerance: Handled via re-execution

- On worker failure:
  - Detect failure via periodic heartbeats
  - Re-execute completed and in-progress *map* tasks
  - Re-execute in progress *reduce* tasks
  - Task completion committed through master
- Master failure:
  - Could handle, but don't yet (master failure unlikely)

Robust: lost 1600 of 1800 machines once, but finished fine

Semantics in presence of failures: see paper

Refinement: Redundant Execution

Slow workers significantly lengthen completion time

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

Solution: Near end of phase, spawn backup copies of tasks

- Whichever one finishes first "wins"

Effect: 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, but not always possible

- On seg fault:
  - Send UDP packet to master from signal handler
  - Include sequence number of record being processed

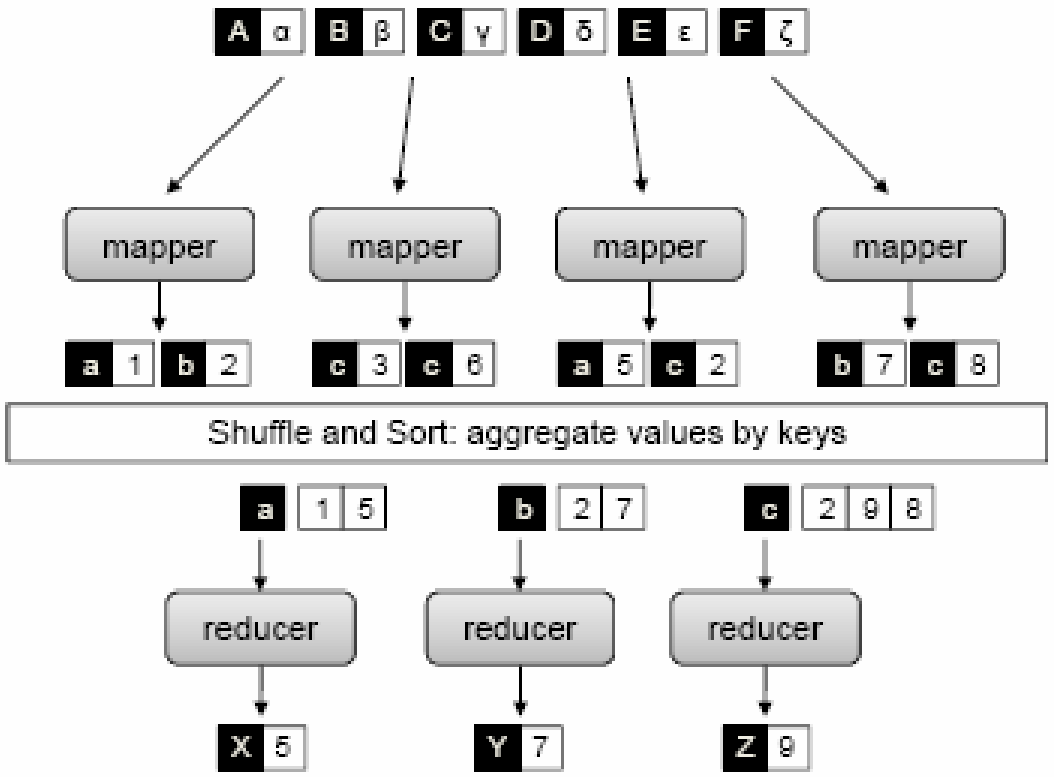
- If master sees two failures for same record:
  - Next worker is told to skip the record

Effect: Can work around bugs in third-party libraries

Other Refinements (see paper)

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

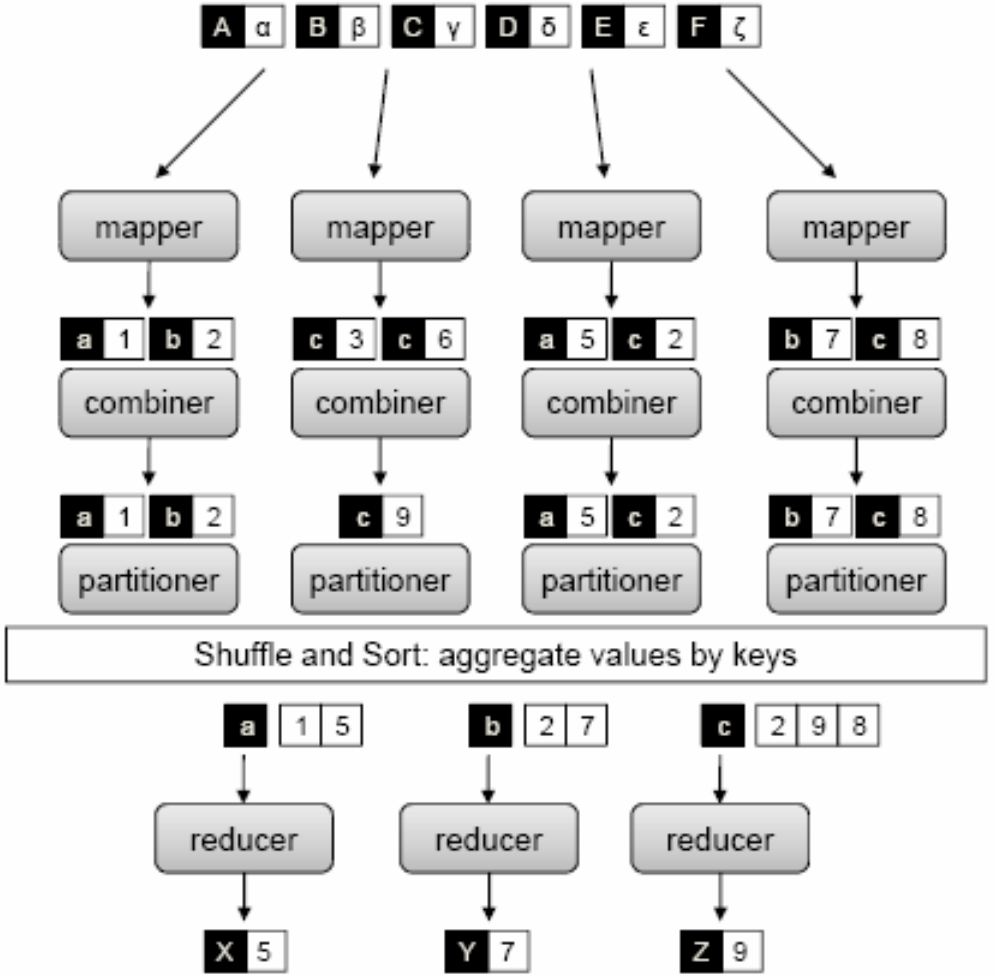
Visión lógica de ejecución Hadoop-MR



Data-Intensive Text Processing with MapReduce

Jimmy Lin and Chris Dyer. University of Maryland

Visión lógica Hadoop-MR con *combiners*



Data-Intensive Text Processing with MapReduce. Lin & Dyer.

Performance

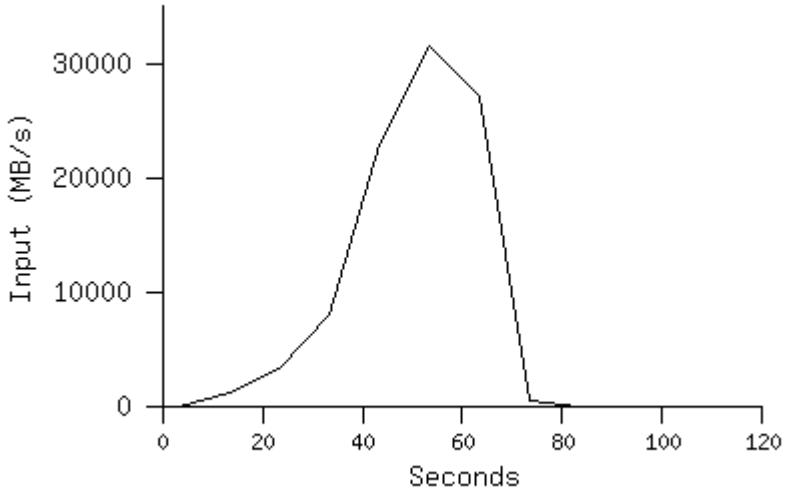
Tests run on cluster of 1800 machines:

- 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	Scan 10^{10} 100-byte records to extract records matching a rare pattern (92K matching records)
MR_Sort	Sort 10^{10} 100-byte records (modeled after TeraSort benchmark)

MR_Grep



Locality optimization helps:

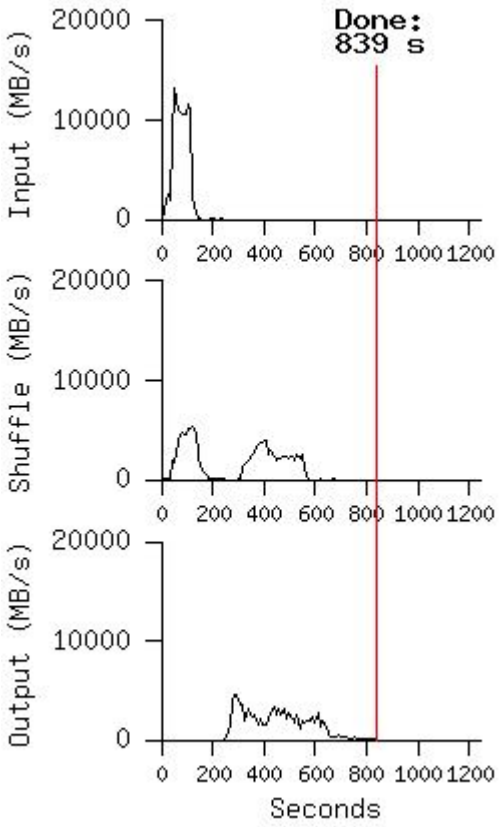
- 1800 machines read 1 TB of data at peak of ~31 GB/s
- Without this, rack switches would limit to 10 GB/s

Startup overhead is significant for short jobs

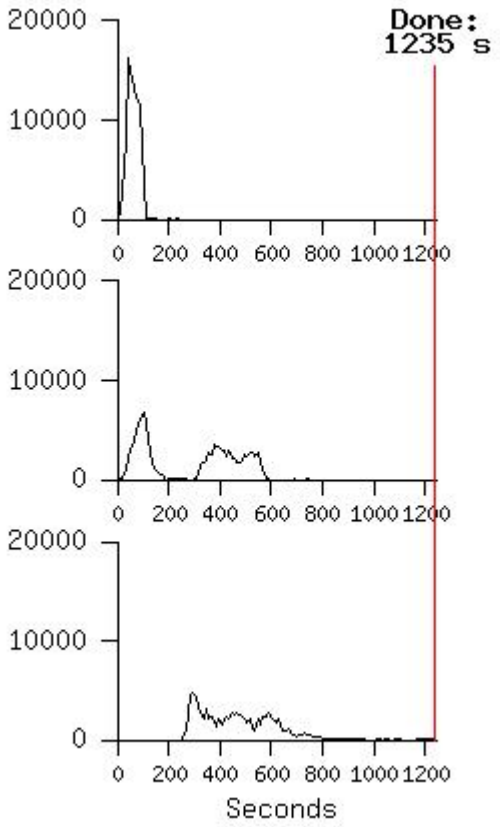
MR_Sort

- Backup tasks reduce job completion time significantly
- System deals well with failures

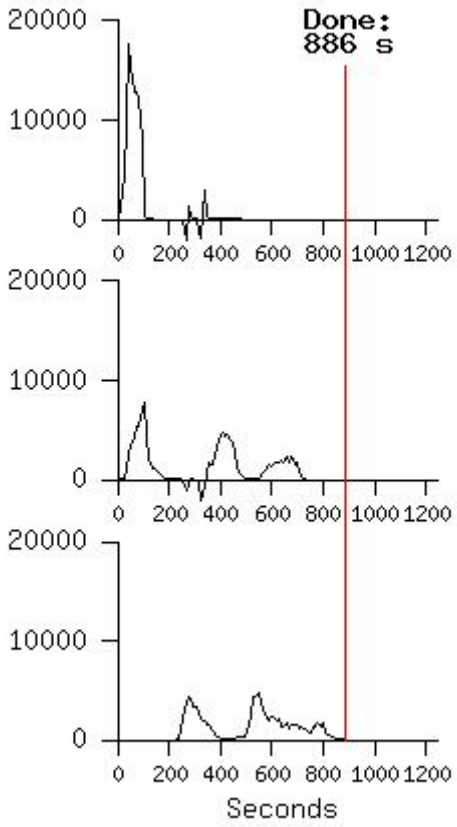
Normal



No backup tasks



200 processes killed



Experience: Rewrite of Production Indexing System

Rewrote Google's production indexing system using MapReduce

- Set of ~~10~~, ~~14~~, ~~17~~, ~~21~~, 24 MapReduce operations
- New code is simpler, easier to understand
- MapReduce takes care of failures, slow machines
- Easy to make indexing faster by adding more machines

Usage: MapReduce jobs run in August 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	

Conclusions

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

Thanks to Josh Levenberg, who has made many significant improvements and to everyone else at Google who has used and helped to improve MapReduce.