Cloud & Cluster Data Management

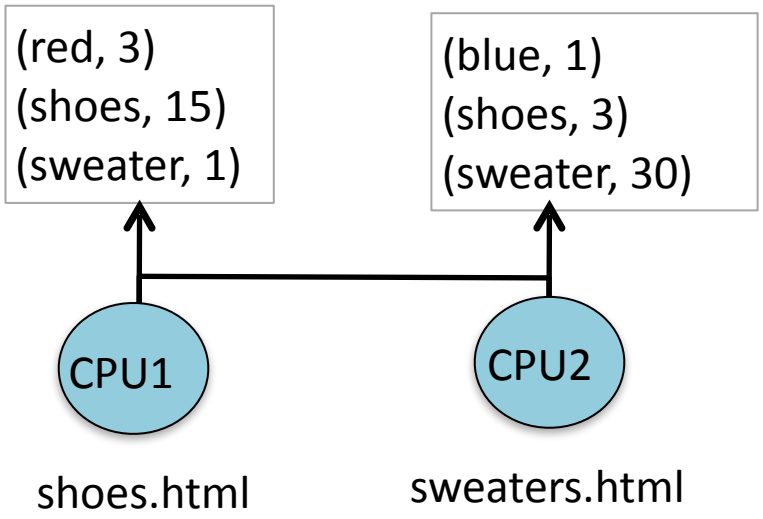
MAP/REDUCE

Example: Count Word Occurrences

- Problem: Process html files and and count the number of times words occur in those documents
- Three options
  - Write a program
  - Use Map Reduce
  - Use a database

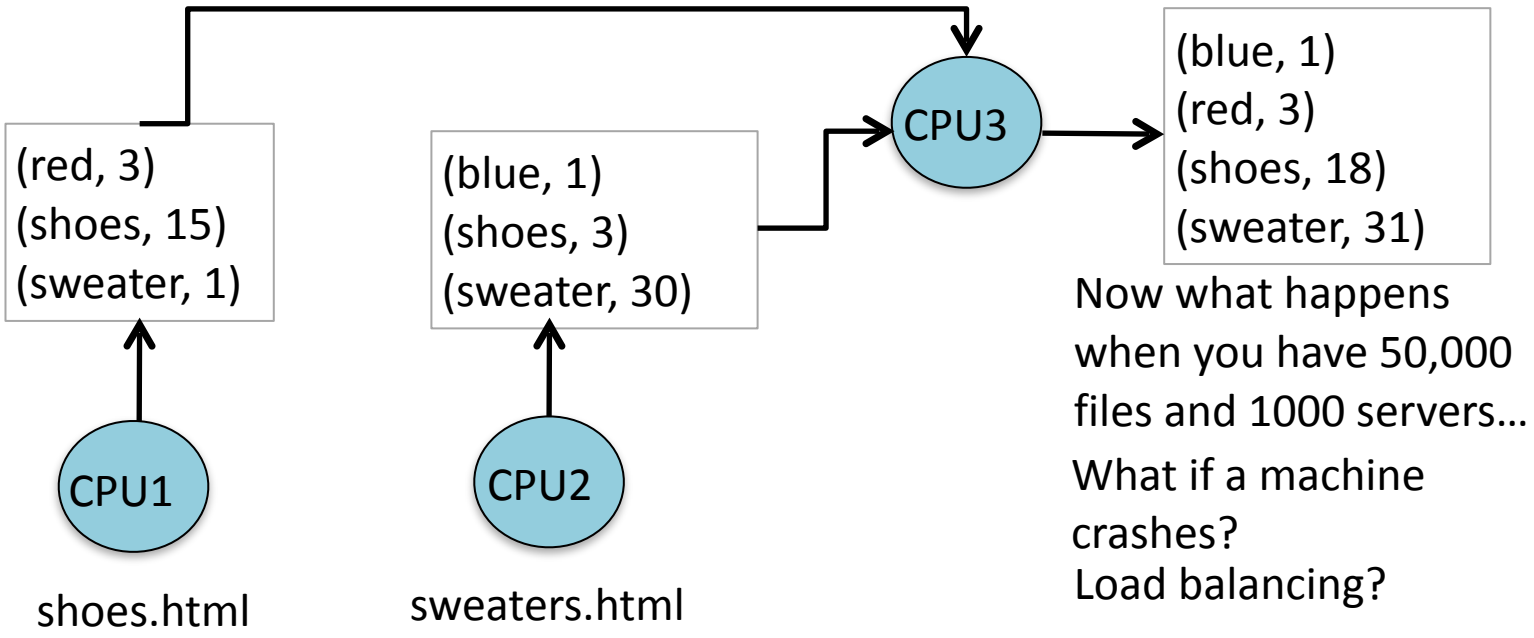
Count Word Occurrences – Write a Program

- Count word occurrences in large collection of documents
- For Starters: Count word occurrences in two files: shoes.html, sweaters.html



Count Word Occurrences – Write a Program

- Count word occurrences in large collection of documents
- For Starters: Count word occurrences in two files: shoes.html, sweaters.html



Now what happens when you have 50,000 files and 1000 servers…

What if a machine crashes?

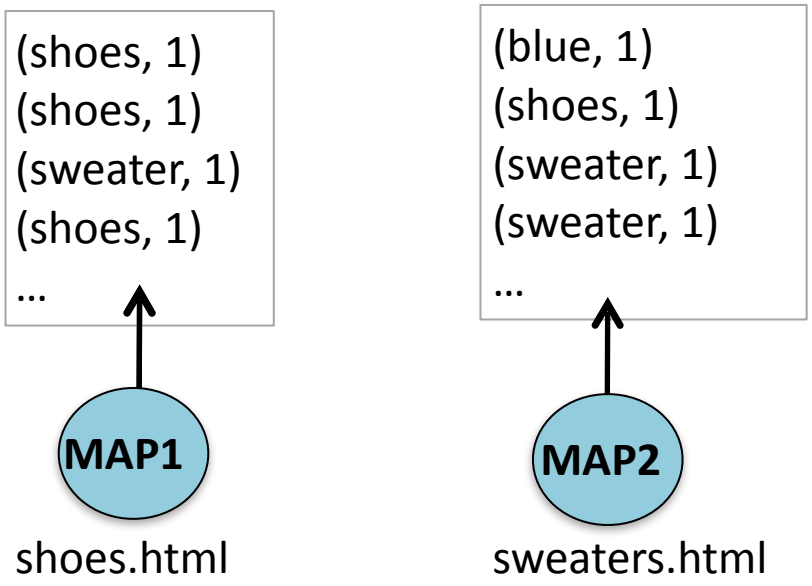
Load balancing?

Count Word Occurrences – Map Reduce

- Map:
  - Input: (document name, document contents) pairs
  - Output: (word, “1”) for each word in the contents
- Reduce:
  - Input: (word, [1,1,1,1]) (list of counts)
  - Output: 4 (sum of counts)

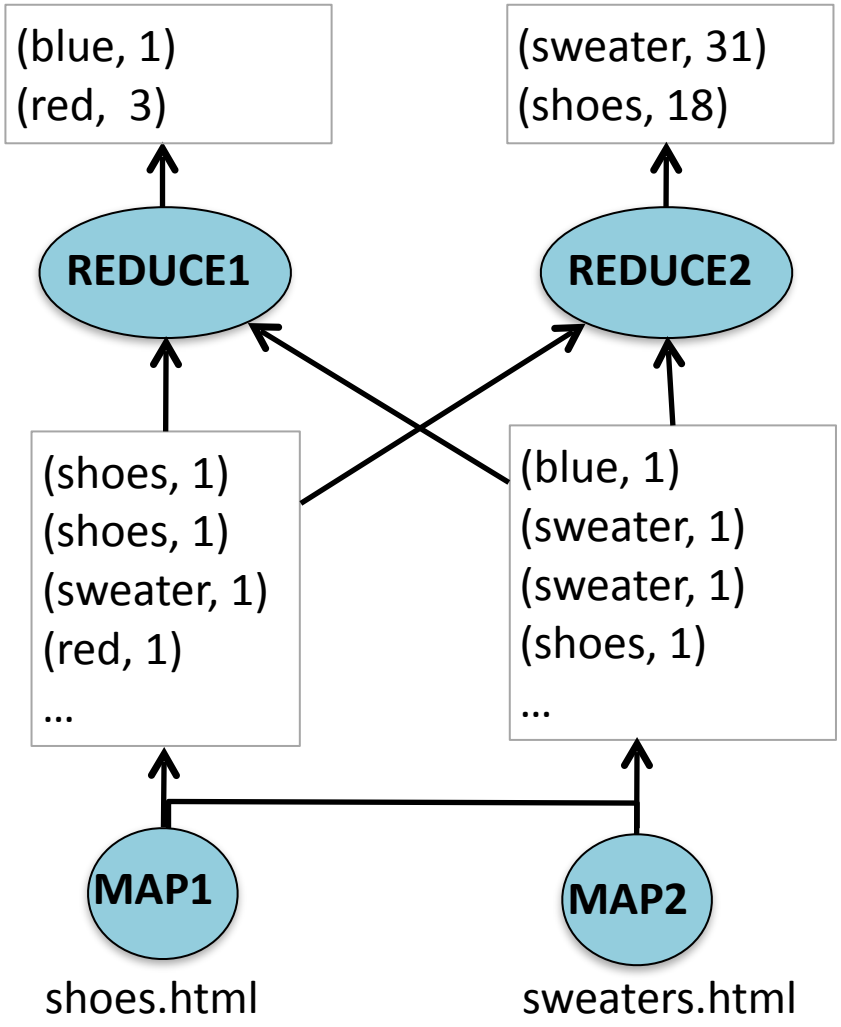
Count Word Occurrences – Map Reduce

- Map:
  - Input: (document name, document contents) pairs
  - Output: (word, “1”) for each word in the contents
- Reduce:
  - Input: (word, [1,1,1,1]) (list of counts)
  - Output: 4 (sum of counts)



Count Word Occurrences – Map Reduce

- Map:
  - Input: (document name, document contents) pairs
  - Output: (word, “1”) for each word in the contents
- Reduce:
  - Input: (word, [1,1,1,1]) (list of counts)
  - Output: 4 (sum of counts)



Now what happens when you have 50,000 files and 1000 servers…

Count Word Occurrences - Database

- Step 1: Pick schema and load database
- Step 2: Query

STEP 1:

```
CREATE TABLE words
(docname text, word text, lineno integer)

COPY words FROM shoes.csv with csv
COPY words FROM sweaters.csv with csv
```

→

```
(shoes.csv, red, 23)
(sweaters.csv, blue, 3)
(shoes.csv, shoes, 29)
(shoes.csv, shoes, 76)
….
```

STEP 2:

```
SELECT word, count(*)
FROM words
GROUP BY word
```

→

```
(blue, 1)
(red, 3)
(shoes, 18)
(sweater, 31)
```

Count Word Occurrences - Databases

- But, now, you can also do…

```
SELECT word, count (*)
FROM words
WHERE count > 100
GROUP BY word
```

```
SELECT MAX(lineno)
FROM words
WHERE word = ‘shoes’
```

```
SELECT count(*)
FROM words
WHERE word LIKE “s%”
```

```
SELECT word, count(*)
FROM words
WHERE word LIKE “s%”
GROUP BY word
```

- And, if it’s a parallel database, the scaling and parallelization will all be handled automatically…

Count Word Occurrences - Observations

- Write a Program
  - Requires high-level of programmer ability – scalability & fault-tolerance must be incorporated into the code
  - Limited code re-use
- MapReduce
  - Requires lower level of programmer ability – scalability & fault-tolerance handled by the MapReduce framework
  - Some code re-use
- Database
  - Requires load phase (and schema)
  - Declarative query language (SQL)
    - Query language is powerful, does not require parallel programming skills (but you have to know SQL)
  - Scalability and fault-tolerance handled by (parallel) database

MapReduce: A major step backwards

…

As both educators and researchers, we are amazed at the hype that the MapReduce proponents have spread about how it represents a paradigm shift in the development of scalable, data-intensive applications. MapReduce may be a good idea for writing certain types of general-purpose computations, but to the database community, it is:

1. A giant step backward in the programming paradigm for large-scale data intensive applications

2. A sub-optimal implementation, in that it uses brute force instead of indexing

3. Not novel at all -- it represents a specific implementation of well known techniques developed nearly 25 years ago

4. Missing most of the features that are routinely included in current DBMS

5. Incompatible with all of the tools DBMS users have come to depend on

….

https://homes.cs.washington.edu/~billhowe/mapreduce_a_major_step_backwards.html

Let's Look at Map Reduce …

- Background and Requirements
  - computations are conceptually straightforward
  - input data is (very) large
  - distribution over hundreds or thousands of nodes
  - Had implementations of hundreds of special-purpose computations
    - inverted indices, number pages crawled per host, most frequent queries all executed over web logs
    - Computations simple, but parallelization makes things complex…

Map Reduce - Motivation

- Programming model for processing of large data sets
  - abstraction to express simple computations
  - claim: many real-world tasks are expressible in this model
  - automatic parallelization and scalability (commodity shared-nothing machines)
  - hide details of parallelization, data distribution, fault-tolerance, and load-balancing
  - programmers don’t need experience in parallel programming…
  - re-execution is primary mechanism for fault-tolerance

Programming Model

- Inspired by primitives from functional programming languages such as Lisp, Scheme, and Haskell
- Input and output are sets of key/value pairs
- Programmer specifies two functions
  - **map** $(k_1, v_1) \rightarrow list(k_2, v_2)$
  - **reduce** $(k_2, list(v_2)) \rightarrow list(v_2)$
- Key and value domains
  - input keys and values are drawn from a different domain than intermediate and output keys and values
  - intermediate keys and values are drawn from the same domain as output keys and values

Map Function

- User-defined function
  - processes input key/value pair
  - produces a set of *intermediate* key/value pairs
- Map function I/O
  - **input:** read from GFS file (chunk)
  - **output:** written to intermediate file on local disk
- Map/reduce library
  - executes map function
  - groups together all intermediate values with the same key
  - “passes” these values to reduce functions
- Effect of map function
  - processes and partitions input data
  - builds distributed map (transparent to user)
  - similar to “group by” operation in SQL (but with a list as output)

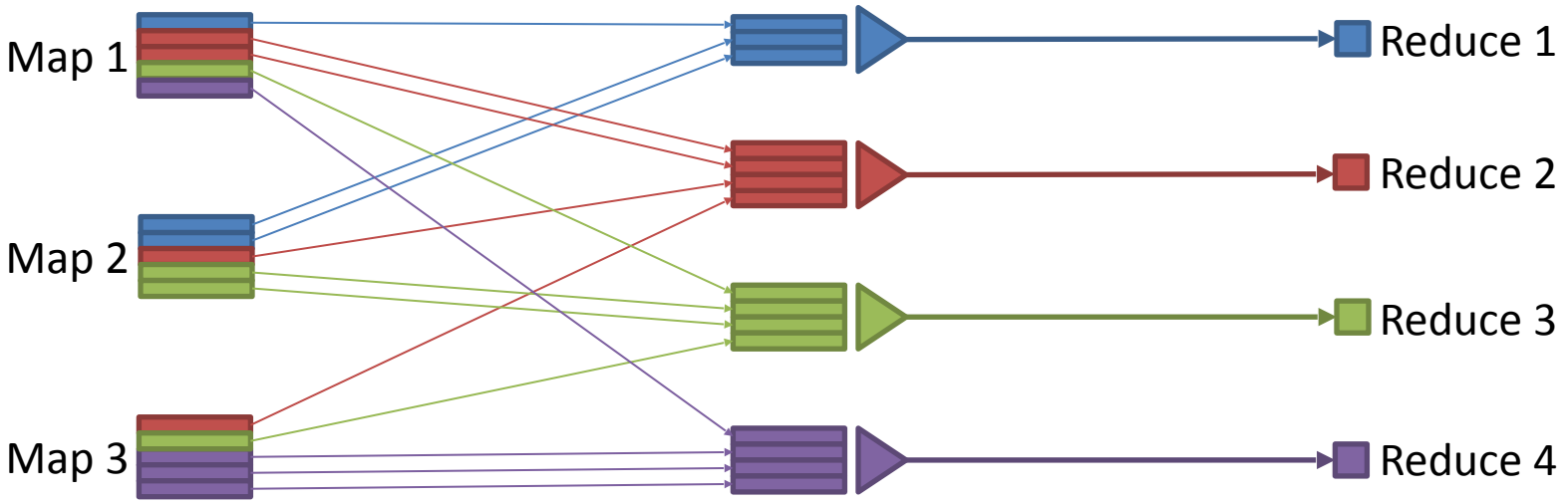
Reduce Function

- User-defined function
  - accepts *one* intermediate key and a set of values for that key
  - merges these values together to form a (possibly) smaller set
  - typically, zero or one output value is generated per invocation
- Reduce function I/O
  - **input:** read from intermediate files using remote reads on local files of corresponding mapper nodes
  - **output:** each reducer writes its output as a file back to GFS
- Effect of reduce function
  - similar to aggregation operation in SQL

Map Reduce Specification

- Names input/output files
- Optional tuning parameters

Map/Reduce Interaction



- Map functions create a user-defined “index” from source data
- Reduce functions compute grouped aggregates based on index
- Flexible framework
  - users can cast raw original data in any model that they need
  - wide range of tasks can be expressed in this simple framework

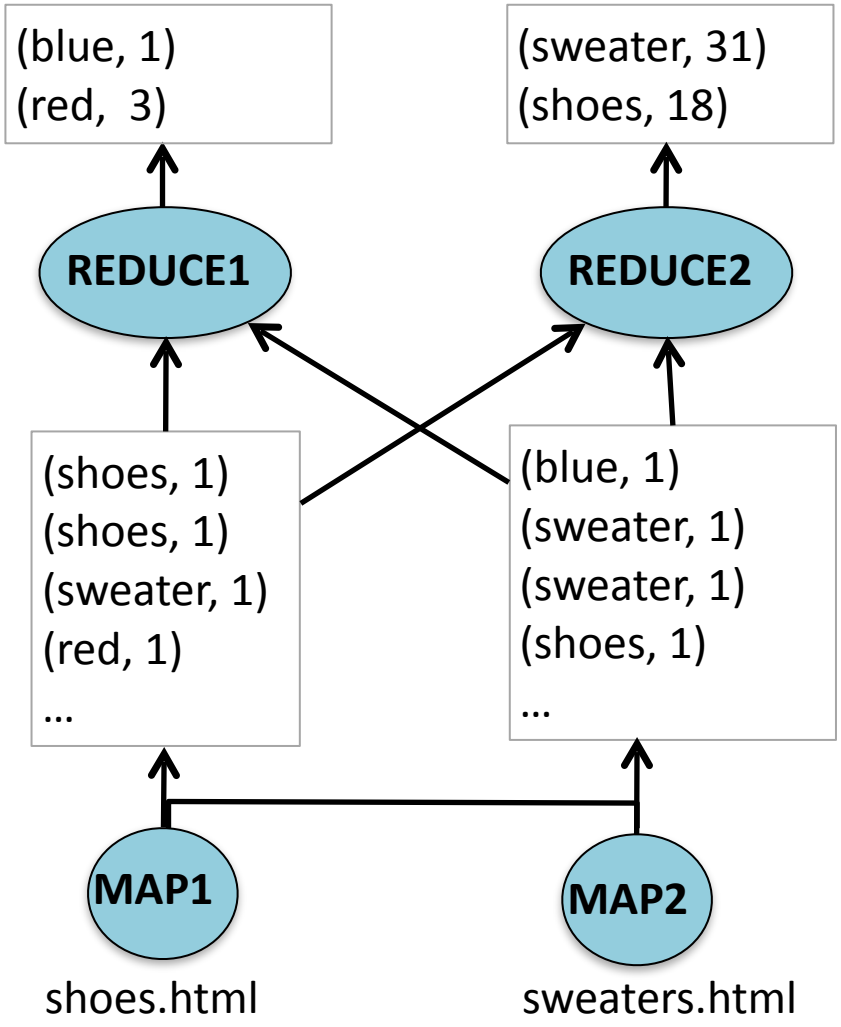
MapReduce Example

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

reduce(String key, Iterator values):
   // key:    word
   // values:
   int result = 0;
   for each v in values:
      result += ParseInt(v);
   Emit(AsString(result));
```

Count Word Occurrences – Map Reduce

- Map:
  - Input: (document name, document contents) pairs
  - Output: (word, "1") for each word in the contents
- Reduce:
  - Input: (word, [1,1,1,1]) (list of counts)
  - Output: 4 (sum of counts)



Now what happens when you have 50,000 files and 1000 servers…

More Examples

- Distributed “grep”
  - **goal:** find positions of a pattern in a set of files
  - **map:** (File, String) → list(Integer, String), emits a <line#, line> pair for every line that matches the pattern
  - **reduce:** identity function that simply outputs intermediate values
- Count of URL access frequency
  - **goal:** analyze Web logs and count page requests
  - **map:** (URL, String) → list(URL, Integer), emits <URL, 1> for every occurrence of a URL
  - **reduce:** (URL, list(Integer)) → list(Integer), sums the occurrences of each URL
- Workload of first example is in map function, whereas it is on the reduce in the second example

More Examples

- Reverse Web-link graph
  - **goal:** find which source pages link to a target page
  - **map:** (URL, CLOB) → list(URL, URL), parses the page content and emits one <target, source> pair for every target URL found in the source page
  - **reduce:** (URL, list(URL)) → list(URL), concatenates all lists for one source URL
- Term-vector per host
  - **goal:** for each host, construct its term vector as a list of <word, frequency> pairs
  - **map:** (URL, CLOB) → list(String, List), parses the page content (CLOB) and emits a <hostname, term vector> pair for each document
  - reduce: (String, list(List<String, Integer>)) → list(List<String, Integer>), combines all per-document term vectors and emits final <hostname, term vector> pairs

More Examples

- Inverted index
  - **goal:** create an index structure that maps search terms (words) to document identifiers (URLs)
  - **map:** (URL, CLOB) → list(String, URL), parses document content and emits a sequence of <word, document id> pairs
  - **reduce:** (String, list(URL)) → list(URL), accepts all pairs for a given word, and sorts and combines the corresponding document ids
- Distributed sort
  - **goal:** sort "records" according to a user-defined key
  - **map:** (? , Object) → list(Key, Record), extracts the key from each "record" and emits <key, record> pairs
  - **reduce:** emits all pairs unchanged
  - Map/reduce guarantees that pairs in each partition are processed ordered by key, but still requires clever *partitioning function* to work!

Discussion Question

Pick an online application and a query and think about how you would implement in Map Reduce

Example Applications: Twitter, Amazon, Instagram,, Snapchat, gmail, FaceBook, Minecraft, Healthcare.gov, Dropbox, Flickr, Instagram, Ebay, Yelp, TripAdvisor, Zillow, E*TRADE, iTunes, online banking

Example Queries: Twitter: count tweets about a hashtag; Instagram: top 10 hashtags in the past day; Amazon: average order price per customer; Facebook: find all of your friends

Map Reduce:

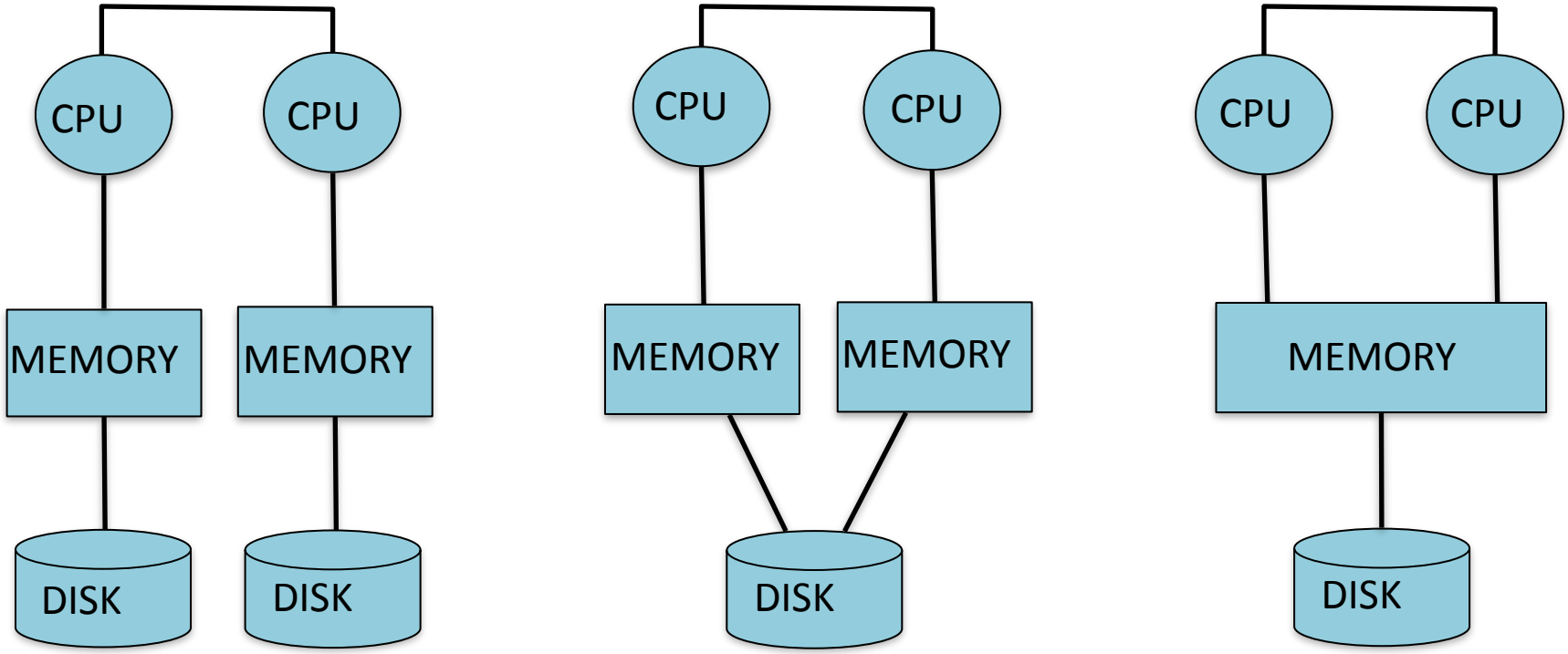
Map Function:

Reduce Function:

Implementation Architecture

- Based on the “Google computing environment”
  - same assumptions and properties as GFS
  - builds on top of GFS
- Architecture
  - one master, many workers
  - users submit jobs consisting of a set of tasks to a scheduling system
  - tasks are mapped to available workers within the cluster by master

Let's Remember HW Architectures…



Easy/cheap to build (commodity hw)

Expensive to build (custom hw)

Difficult to program

Easier to program

Implementation Execution

- Execution overview
  - map invocations are distributed across multiple machines by automatically partitioning the input data into a set of *M* splits
  - input splits can be processed in parallel
  - reduce invocations are distributed by partitioning the intermediate key space into *R* pieces using a partitioning function, e.g. "*hash(key)* **mod** *R*"

Execution Overview

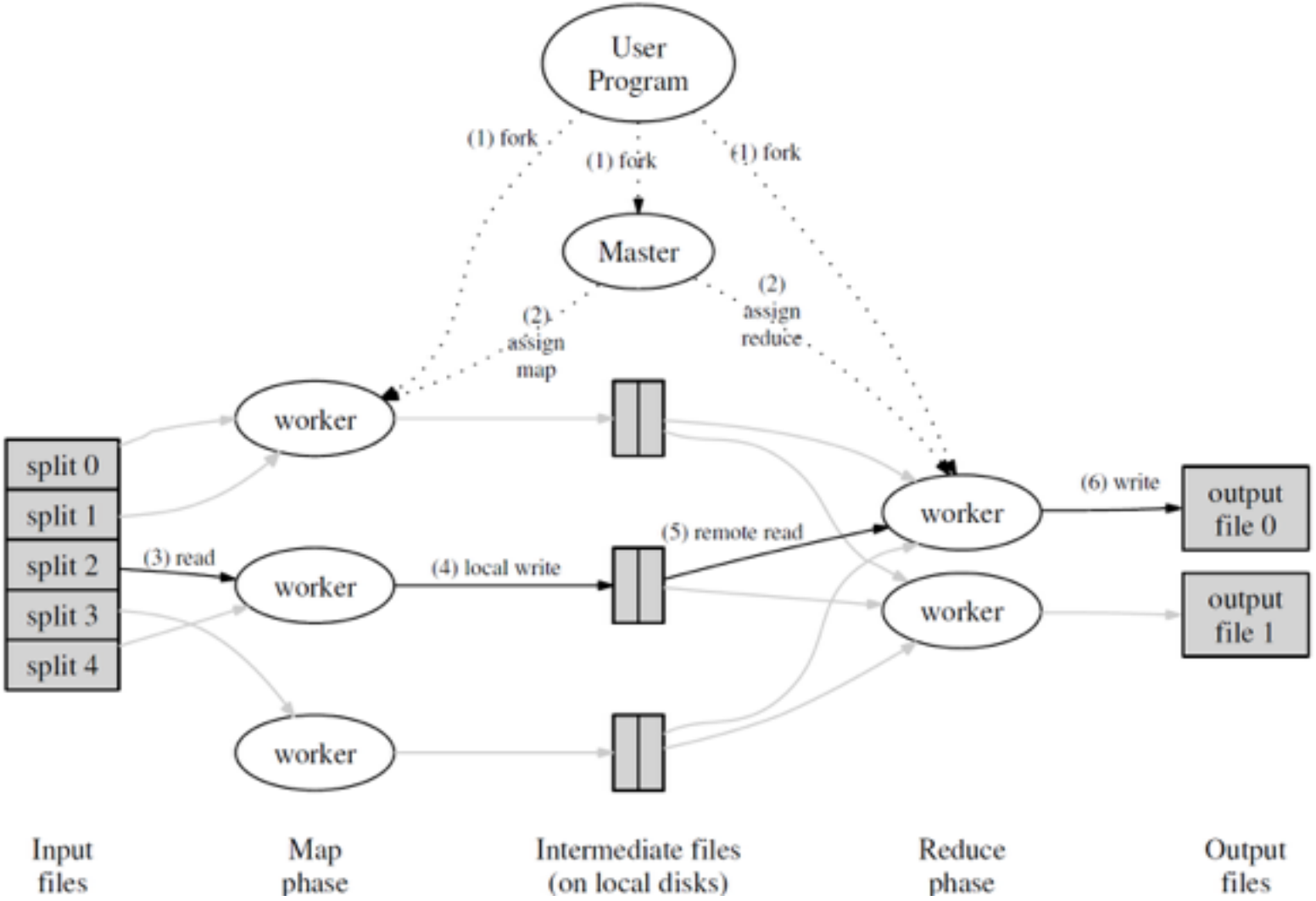


Figure Credit: "MapReduce: Simplified Data Processing on Large Clusters" by J. Dean and S. Ghemawat, 2004

Execution Overview

1. Map/reduce library splits input files into *M* pieces and then starts copies of the program on a cluster of machines
2. One copy is the master, the rest are workers; master assigns *M* map and *R* reduce tasks to idle workers
3. Map worker reads its input split, parses out key/value pairs and passes them to user-defined map function
4. Buffered pairs are written to local disk, partitioned into *R* regions; location of pairs passed back to master
5. Reduce worker is notified by master with pair locations; uses RPC to read intermediate data from local disk of map workers and sorts it by intermediate key to group tuples by key
6. Reduce worker iterates over sorted data and for each unique key, it invokes user-defined reduce function; result appended to reduce partition
7. Master wakes up user program after all map and reduce tasks have been completed

Master Data Structures

- Information about all map and reduce task
  - **worker state:** idle, in-progress, or completed
  - **identity** of the worker machine (for non-idle tasks)
- Intermediate file regions
  - propagates intermediate file locations from map to reduce tasks
  - stores locations and sizes of the *R* intermediate file regions produced by each map task
  - updates to this location and size information are received as map tasks are completed
  - information pushed incrementally to workers that have in-progress reduce tasks

Fault Tolerance

- Worker failure
  - master pings workers periodically; assumes failure if no response
  - completed/in-progress map and in-progress reduce tasks on failed worker are rescheduled on a different worker node
    - output of map is stored locally, reduce output is stored in GFS
  - resilient to large-scale worker failures
- Master failure
  - master fails -> restart map-reduce operation
  - “given that there is only a single master, failure is unlikely”

Fault Tolerance

- Failure semantics
  - if user-defined functions are *deterministic*, execution with faults produces the same result as execution without faults
  - rely on atomic commits of map and reduce tasks
  - map & reduce write to private temp files
  - map produces R files, reduce produces one file
  - map completes, sends list of files to Master (if task already completed, message ignored), Master records names of data files
  - reduce completes, renames temp file to final output file – conflicts resolved by atomic rename operation

More Implementation Aspects

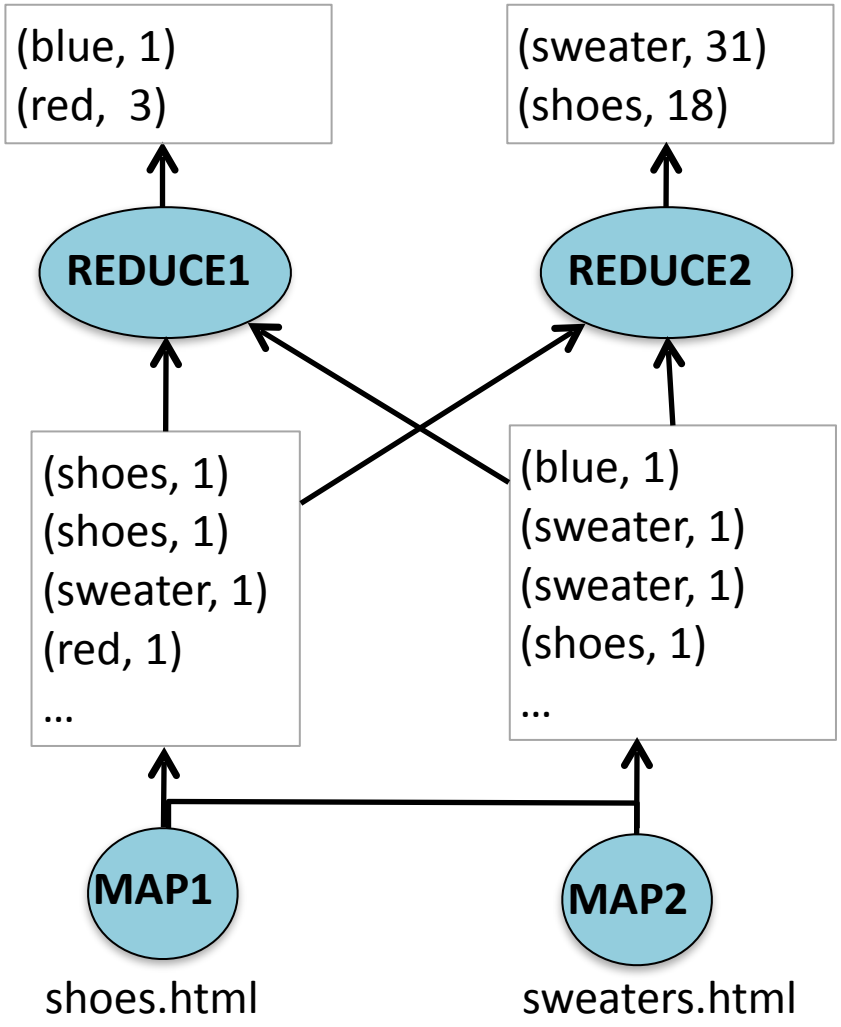
- Locality
  - network bandwidth is scarce resource
  - move computation close to data
  - master takes GFS metadata into consideration (location of replicas)
  - vast majority of input data is ready locally
- Task granularity
  - *M* map tasks, *R* reduce tasks
  - *M* and *R* much larger than number of worker machines (dynamic load balancing, speeds up recovery)
  - Sizes of *M* and *R* limited by
    - master makes *O(M + R)* scheduling decisions
    - master stores *O(M * R)* states in memory (constant factors small)
  - R constrained by users (produces R files)
  - M selected so each task uses 16MB-64MB (chunk size) of input data
  - R small multiple of # of worker machines (M=200,000, R=5,000, 2,000 machines)

More Implementation Aspects

- Backup Tasks
  - "stragglers" are a common cause for suboptimal performance
    - Bad disk, overloaded machine, no processor cache
  - as a map/reduce computation comes close to completion, master assigns the same task to multiple workers
  - Significantly reduces time to complete, at the cost of a few percent execution time increase

Count Word Occurrences – Map Reduce

- Map:
  - Input: (document name, document contents) pairs
  - Output: (word, "1") for each word in the contents
- Reduce:
  - Input: (word, [1,1,1,1]) (list of counts)
  - Output: 4 (sum of counts)



Now what happens when you have 50,000 files and 1000 servers…

Map Reduce – Key Features

- Single Master
  - tracks mapper and reducer status and intermediate file locations
- Fault Tolerance – monitor workers and restart jobs (also used for stragglers)
- Locality – move computation to disk where possible
- Task granularity
  - M selected so each task uses 16MB-64MB (chunk size) of input data
  - R selected based on user needs
  - R small multiple of # of worker machines (M=200,000, R=5,000, 2,000 machines)

Refinements

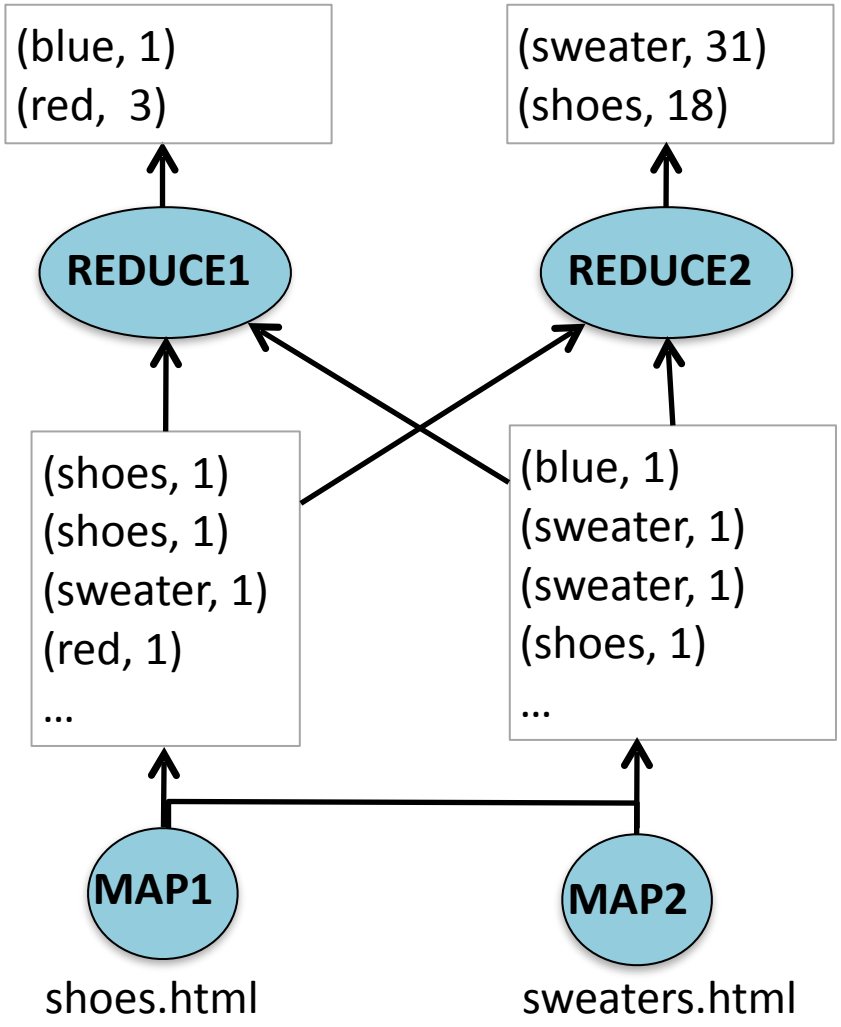
- Partitioning function
  - default function can be replaced by user; “application-specific” partitioning
- Ordering guarantees
  - within a given partition, intermediate key/value pairs are processed in increasing key order
- Combiner function
  - Effectively: partially apply the reducer at the mapper
  - Addresses significant key repetitions (zipf distribution < the, 1>)
- Input and output types
  - Default input reader - key: offset in file, value: contents of line
  - user can define their own “readers” and “writers”

Refinements

- Skipping bad records (a.k.a. dealing with bugs in the code)
  - Skip records on which tasks have failed many times
- Status Information
  - Progress & debug info provided to user over HTTP
- Counters (good for sanity checking)
  - counter facility to count occurrences of various events (workers count, send to master)

Count Word Occurrences – Map Reduce

- Map:
  - Input: (document name, document contents) pairs
  - Output: (word, “1”) for each word in the contents
- Reduce:
  - Input: (word, [1,1,1,1]) (list of counts)
  - Output: 4 (sum of counts)



Now what happens when you have 50,000 files and 1000 servers…

Performance

- Tasks: Grep & Sort over 10^{10} 100-byte records (~1TB data)
- Grep – extracts small amount of data from large data set
- Sort – shuffles data from one representation to another
- Cluster
  - 1800 machines
    - 2GHz Intel Xeon, Hyperthreading enabled, 4GB memory (1-1.5GB for other tasks)
  - Weekend afternoon (mostly idle disks & network)

Grep Set-up

- Scans through 10^{10} 100-byte records
- Relatively rare three-character pattern (pattern occurs in 92,337 records).
- The input is split into approximately 64MB pieces (M = 15000)
- Entire output is placed in one file (R = 1).

Performance Experiments - Grep

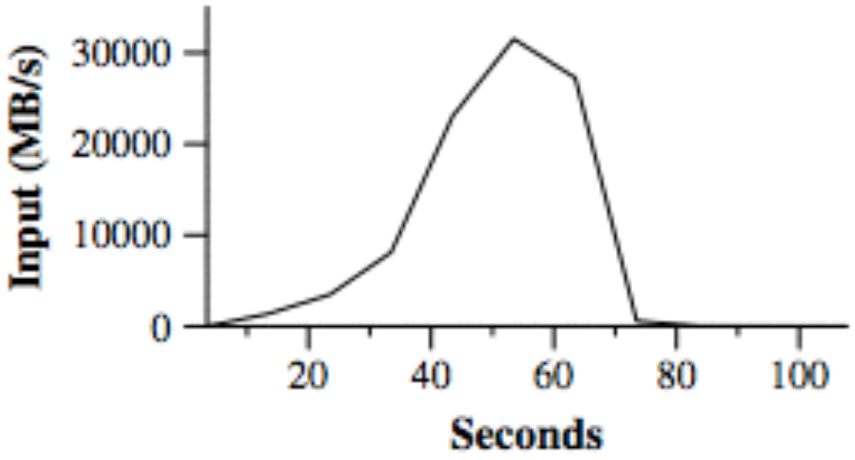


Figure 2: Data transfer rate over time

- Shows progress of computation over time
  - Rate of input data is scanning peaks at > 30 GB/s (1764 workers)
- Computation takes ~150 seconds - includes ~60sec of startup overhead
  - Propagation of the program to all worker machines
  - Interaction with GFS – open 1000 input files, get info for locality

Figure Credit: "MapReduce: Simplified Data Processing on Large Clusters" by J. Dean and S. Ghemawat, 2004

Performance Setup - Sort

- Sort 10^{10} 100-byte records (~1TB data)
  - 3-line **map** function extracts a 10-byte sorting key
  - **Reduce** is the identity function (built-in)
  - < 50 lines of user code
  - 64MB input data pieces, M = 15,000
  - R = 4,000 (partition on initial bytes of key)
    - Partitioning function knows distribution of keys
  - Output is 2-way replicated (writes ~2TB)

Performance Experiments - Sort

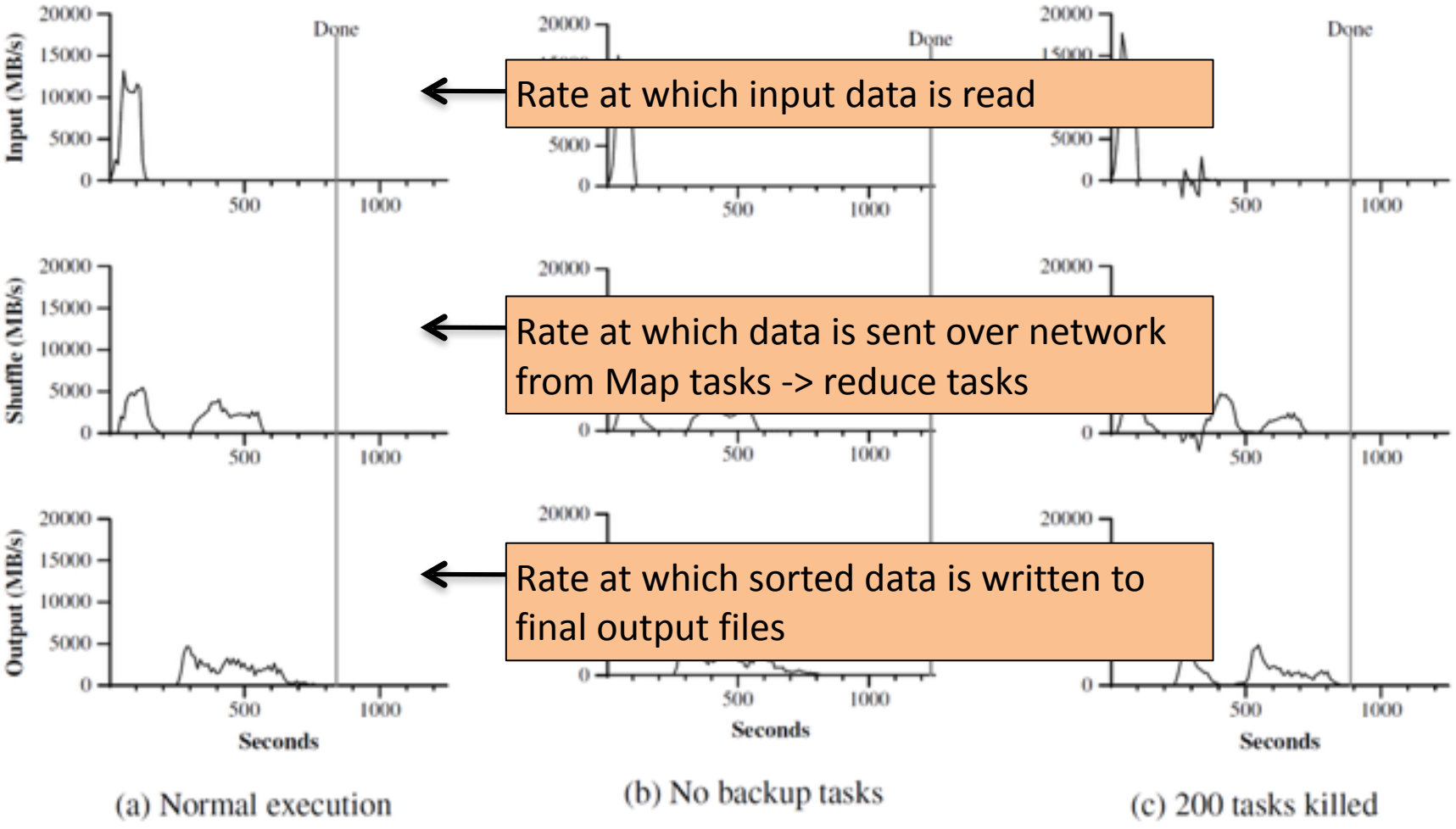
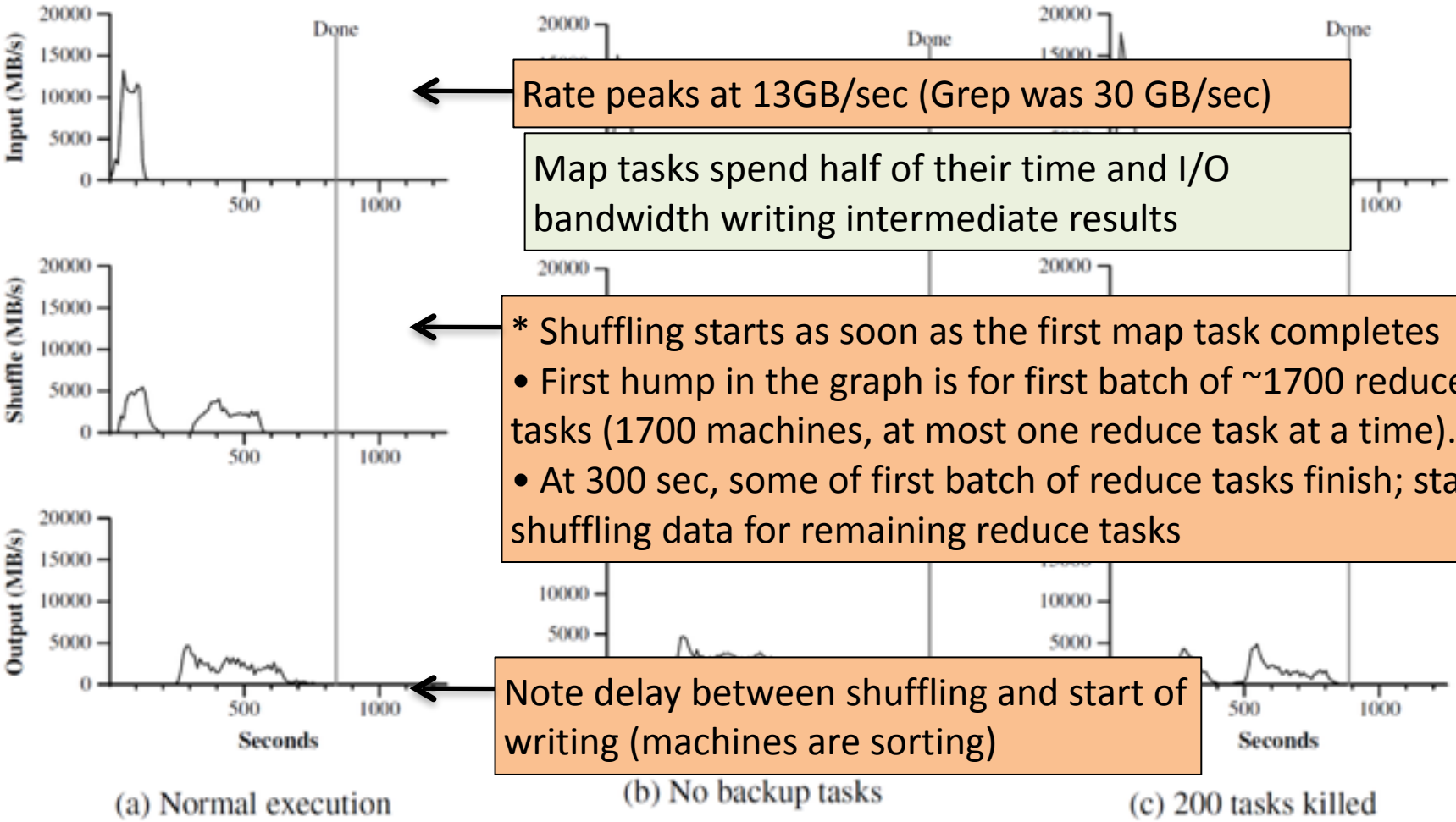


Figure Credit: "MapReduce: Simplified Data Processing on Large Clusters" by J. Dean and S. Ghemawat, 2004

Performance Experiments - Sort



Rate peaks at 13GB/sec (Grep was 30 GB/sec)

Map tasks spend half of their time and I/O bandwidth writing intermediate results

* Shuffling starts as soon as the first map task completes
• First hump in the graph is for first batch of ~1700 reduce tasks (1700 machines, at most one reduce task at a time).
• At 300 sec, some of first batch of reduce tasks finish; start shuffling data for remaining reduce tasks

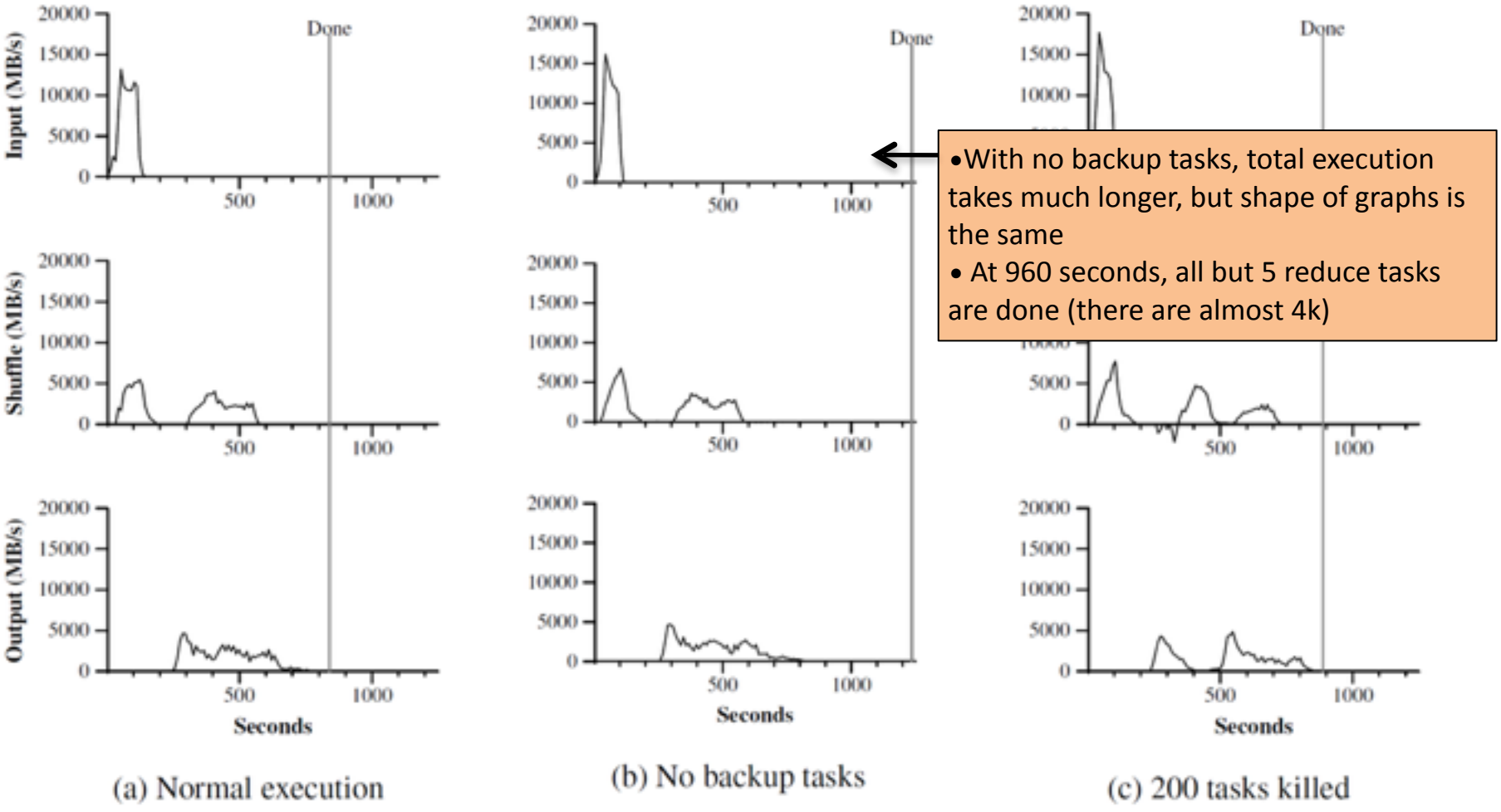
Note delay between shuffling and start of writing (machines are sorting)

Figure Credit: "MapReduce: Simplified Data Processing on Large Clusters" by J. Dean and S. Ghemawat, 2004

Sort - Comments

- Input rate is higher than the shuffle rate and the output rate because of locality optimization
  - Input data is read mostly locally
- Shuffle rate is higher than the output rate because the output phase writes two copies of the sorted data
- Two replicas because that replication is the mechanism for reliability and availability

Effect of Backup Tasks



- With no backup tasks, total execution takes much longer, but shape of graphs is the same
- At 960 seconds, all but 5 reduce tasks are done (there are almost 4k)

Figure Credit: "MapReduce: Simplified Data Processing on Large Clusters" by J. Dean and S. Ghemawat, 2004

Machine Failures

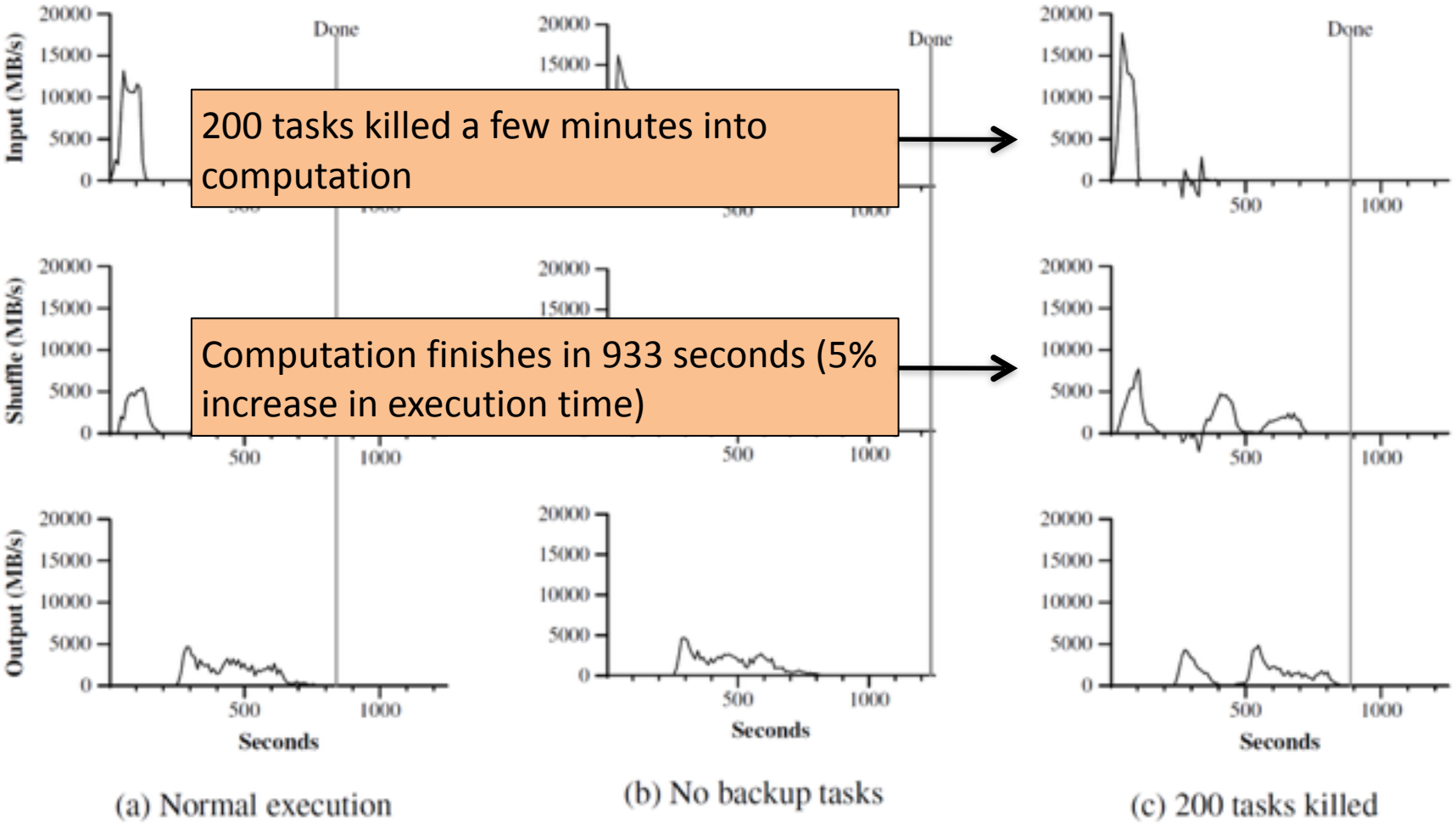


Figure Credit: "MapReduce: Simplified Data Processing on Large Clusters" by J. Dean and S. Ghemawat, 2004

Experience

- Written in Feb 2003
- Enhancements in Aug 2003
  - Locality optimization
  - Dynamic load balancing of task execution
- Usage
  - Large-scale machine learning
  - Clustering for Google News
  - Extraction of data to produce reports of popular queries
  - Extraction of web page properties
  - Large-scale graph computations
- Successful because:
  - “makes it possible to write a simple program and run it efficiently on a thousand machines in the course of half an hour”
  - “allows programmers who have no experience with distributed and/or parallel systems to exploit large amounts of resources easily”

Experience - Statistics

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

Table 1: MapReduce jobs run in August 2004

Figure Credit: "MapReduce: Simplified Data Processing on Large Clusters" by J. Dean and S. Ghemawat, 2004

Experience - Indexing

- Rewrite of production indices for Google web search
- 20TB raw data (2004)
- Indexing is a sequence of 5-10 MapReduce operations
  - was ad-hoc distributed passes in prior version
- Benefits
  - Indexing code is simpler, smaller, easier to understand – fault tolerance, distribution and parallelization hidden in MR library
  - Keep conceptually unrelated computation separate – not mixed together to avoid multiple passes over the data – making changes is easier (few months to few days)
  - Indexing process easier to operate: machine failures, slow machines, networking hiccups are dealt with automatically & easy to add new machines

References

- J. Dean and S. Ghemawat: **MapReduce: Simplified Data Processing on Large Clusters.** *Proc. Symp. on Opearting Systems Design & Implementation (OSDI), pp. 137-149, 2004.*