What happens at such a scale?

"In pioneer days they used oxen for heavy pulling, and when one ox couldn't budge a log, they didn't try to grow a larger ox. We shouldn't be trying for bigger computers, but for more systems of computers."

– Grace Hopper

MapReduce

Kaustubh Beedkar
kbeedkar@cse.iitd.ac.in

Department of Computer Science and Engineering
Indian Institute of Technology Delhi

Outline

1. Background
2. Programming Model
3. Architecture
4. Hadoop MapReduce
5. Hadoop HDFS
6. Algorithms & Data Management
7. Discussion

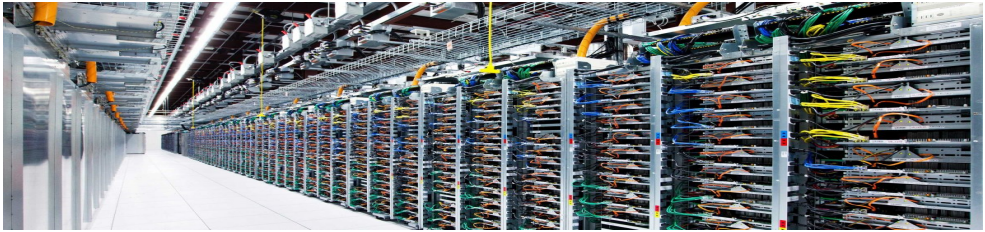
BigData management, storage, analytics — a technological challenge

Do existing approaches suffice to solve this challenge?

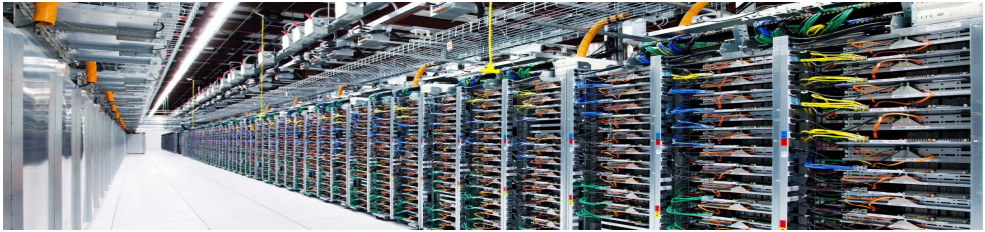
- Can we put the data in a relational database?
- Can we find an appropriate schema for the data?
- Can we process the data on a single machine?
- Can we process the data in Matlab?

Probably not!

We need a new set of tools

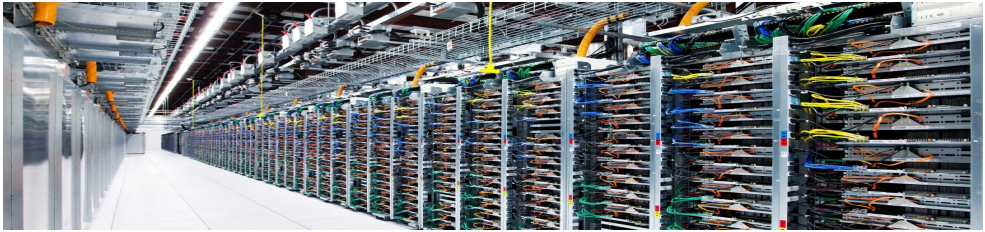


We need a new set of tools



- Distributed file systems
  - Store petabytes of data in the cluster
  - Transparently handle reads, writes, and replication

We need a new set of tools



- Distributed file systems
  - Store petabytes of data in the cluster
  - Transparently handle reads, writes, and replication
- Parallel processing platforms
  - Offer a programming model to allow developers to write distributed applications
  - Move computation to data, not data to computation
  - Relieve the developer from handling concurrency, network communication, and machine failures

MapReduce & Hadoop: History (1)

- 2003: Google publishes about its **cluster architecture** & **distributed file system** (GFS)
- 2004: Google publishes about its **MapReduce** programming model on top of GFS
  - C++, closed-source, Python and Java APIs on top
- 2006: Apache & Yahoo develop **Hadoop** & **HDFS**
  - Java, open source, various APIs on top
- 2007: Hadoop becomes an independent Apache project
  - Yahoo! uses Hadoop in production

MapReduce & Hadoop: History(2)

- Today: Hadoop is used as a **general-purpose storage and analysis platform** for Big Data
  - Hadoop distributions from several vendors including Amazon, Cloudera, EMC, Hortonworks, IBM, Microsoft, Oracle, etc.
  - Many users
  - Research and development actively continues
  - Hadoop not "just" MapReduce and HDFS, but large ecosystem (more later)

Google: The data challenge

- Jeffrey Dean, Google Fellow, PACT'06 keynote speech
  - 20+ billion web pages x 20KB = 400 TB
  - One computer can read 30-35 MB/sec from disk
    $\approx$ 4 months to read the web
  - $\approx$ 1,000 hard drives just to store the web
  - Even more to "do" something with the data
  - **But**: Same problem with 1,000 machines $<$ 3 hours
- MapReduce CACM'08 article
  - 100,000 MapReduce jobs executed in Google every day
  - Total data processed $>$ 20 PB of data per day

Google cluster architecture (1)

- Single-thread performance doesn't matter
  - For large problems, total throughput/$ more important than peak performance
- Stuff breaks
  - 1 server $\rightarrow$ may stay up three years (1,000 days)
  - 10,000 servers $\rightarrow$ expect to lose 10 per day
- Ultra-reliable hardware doesn't really help
  - At large scales, even most reliable hardware fails (albeit less often)
  - Software still needs to be fault-tolerant
  - **Commodity machines** give better performance/$

Dean, 2009

The Joys of Real Hardware

Typical first year for a new cluster:

~0.5 overheating (power down most machines in <5 mins, ~1-2 days to recover)
~1 PDU failure (~500-1000 machines suddenly disappear, ~6 hours to come back)
~1 rack-move (plenty of warning, ~500-1000 machines powered down, ~6 hours)
~1 network rewiring (rolling ~5% of machines down over 2-day span)
~20 rack failures (40-80 machines instantly disappear, 1-6 hours to get back)
~5 racks go wonky (40-80 machines see 50% packetloss)
~8 network maintenances (4 might cause ~30-minute random connectivity losses)
~12 router reloads (takes out DNS and external vips for a couple minutes)
~3 router failures (have to immediately pull traffic for an hour)
~dozens of minor 30-second blips for dns
~1000 individual machine failures
~thousands of hard drive failures
slow disks, bad memory, misconfigured machines, flaky machines, etc.

Long distance links: wild dogs, sharks, dead horses, drunken hunters, etc.

Google cluster architecture (2)

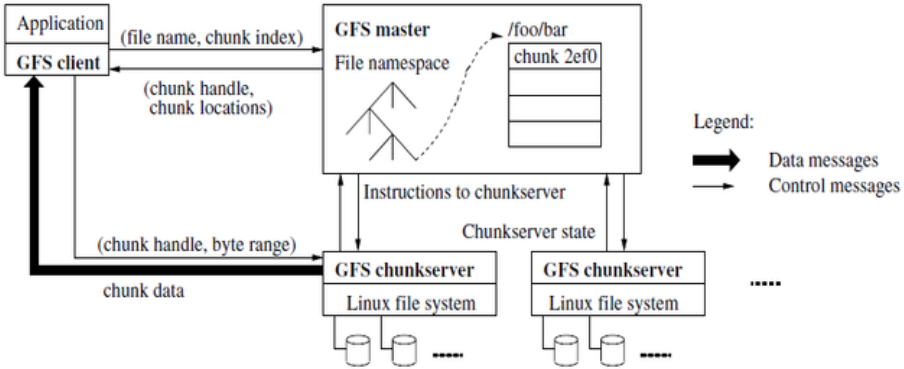
- Reliable computer infrastructure from clusters of **unreliable commodity PCs**
- **Replicate services** across many machines to increase request throughput and availability
- **Favor price/performance** over peak performances
- Such an approach is also suitable in many other situations.

Google File System (GFS)

- Stores very large data files efficiently and reliably
- Files divided into fixed-sized chunks (64MB)
  - Stored as Linux files
- Multiple **chunkservers**
  - Store chunks on local disks
  - Each chunk replicated to multiple chunkservers
  - No caching of remote chunks (not worth it)
- One **master**
  - Each chunk gets a chunk handle from the master
  - Maintains all file system metadata
  - Talks to chunkservers periodically
- Multiple **clients**
  - Talk to master for metadata operations
  - Metadata can be cached at clients
  - Read / write of data from chunkservers

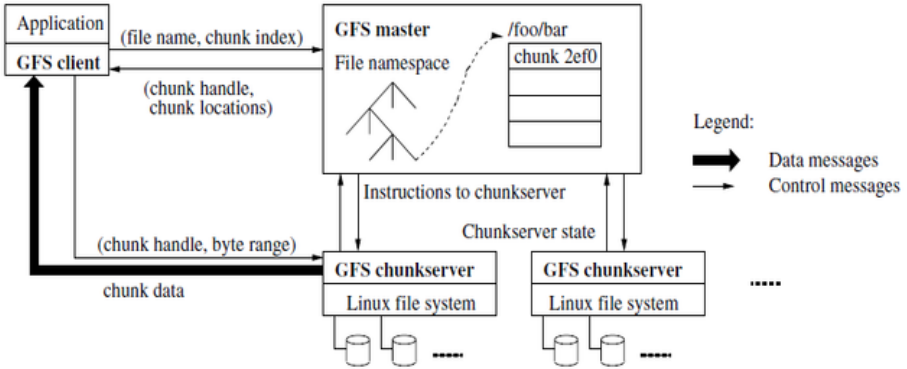
GFS architecture

- Single master, multiple chunk servers



GFS architecture

- Single master, multiple chunk servers
- Master is potential single point of failure/ scalability bottleneck
  - Use shadow masters
  - Minimize master involvements (large chunks, only metadata)



Outline

1. Background
2. Programming Model
3. Architecture
4. Hadoop MapReduce
5. Hadoop HDFS
6. Algorithms & Data Management
7. Discussion

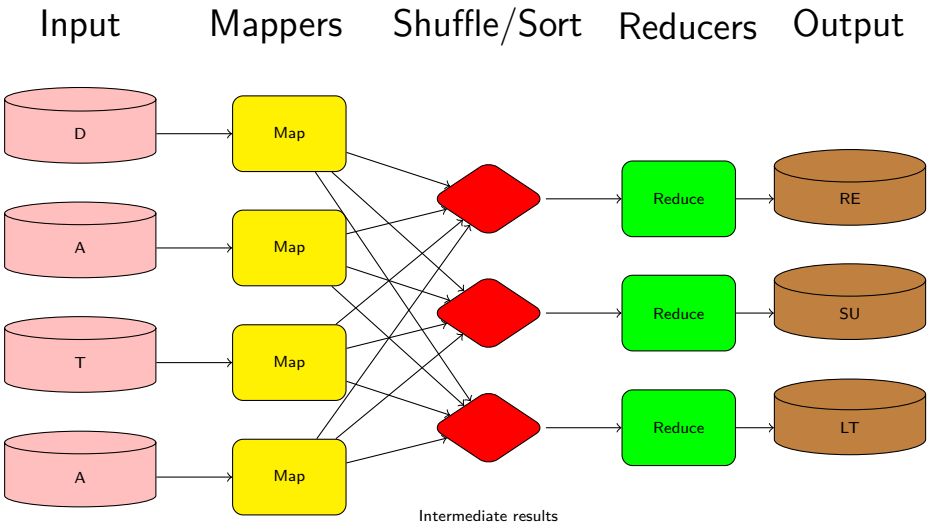
MapReduce

- **A programming model**
  - Functional style
  - Users specify Map and Reduce functions
  - Expressive
- **An associated implementation**
  - Automatically parallelized
  - Automatic partitioning, execution, failure handling, load balancing, communication
  - Can handle very large datasets of clusters of commodity computers

MapReduce in a Nutshell (1)

- Framework
  - Read lots of data
  - Map: process a data item / record
  - Shuffle and sort
  - Reduce: aggregate, summarize, filter, transform
  - Write results
- Computational structure fixed as above
- Map and Reduce are user-specified $\rightarrow$ used to model given problem

MapReduce in a Nutshell (2)



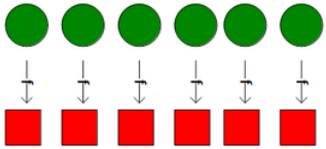
Functional programming foundations

Roughly

- Map in MapReduce $\iff$ map in FP
- Reduce in MapReduce $\iff$ fold in FP

map() in Haskell

- Take a list, apply function f to each element



- Definition
  - map :: $(a \to b) \to [a] \to [b]$ — type of map
  - map $f\,[] = []$ — empty list case
  - map $f\,(x : xs) = f\ x : map\ f\ xs$ — non-empty list case
- Example
  - $\text{map}(x \to x * x)[1, 2, 3, 4, 5] \to [1, 4, 9, 16, 25]$

Examples

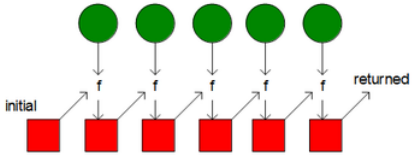
- Optimize a set of images
  - a = image
  - f = function to optimize image
  - b = image
- Detect sentiment in text
  - a = sentence
  - f = function to detect sentiment
  - b = set of sentiments
- Projection (w/o duplicate elimination)
  - a = a relational tuple
  - f = function that removes some attributes
  - b = a relational tuple (now with different schema)

Implicit parallelization in map()

- Application of function f on an element of the list often not influenced by computations on other elements
  - Implies that we can reorder or parallelize the execution
  - Note: not true when f has side-effects or state
- This property forms the basis for MapReduce!

fold() in Haskell

- Move through a list, apply a function f to each element plus an **accumulator**; f returns next accumulator, final accumulator output



- Two versions: fold left, fold right
- Definition
  - $\text{fold}_{left} :: (b \to a \to b) \to b \to [a] \to b$ — type
  - $\text{fold}_{left}\ f\ y\ [] = y$ — empty list
  - $\text{fold}_{left}\ f\ y(x : xs) = fold_{left}\ f(f\ y\ x)xs$ — non-empty list
- Example
  - $fold_{left}(+)0[1, 2, 3] \to (((0 + 1) + 2) + 3) \to 6$

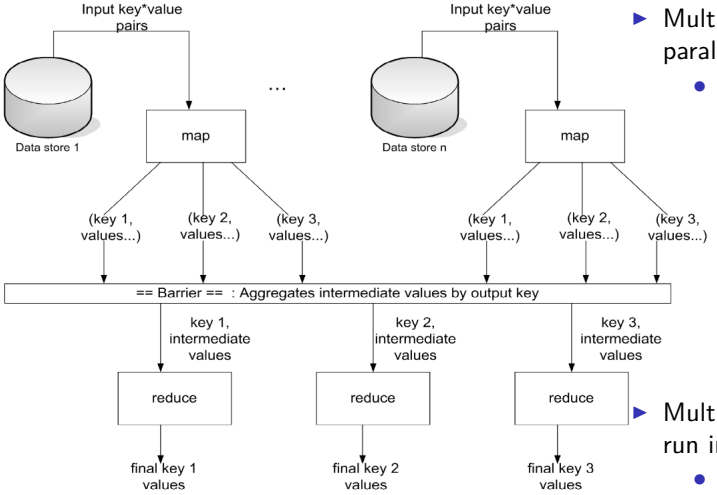
Examples

- Compute an aggregate
  - Count, sum, min, max, top-k, ...
  - Of squares, logs, ...
  - Individually, for subset, per group, ...
- Maximum salary for managers by department
  - a = person record
  - b = set of (department, max. salary)-pairs
  - initial = empty set
  - f = function that updates (dep,sal)-pairs with a given person; updates only if person is manager

MapReduce programming model

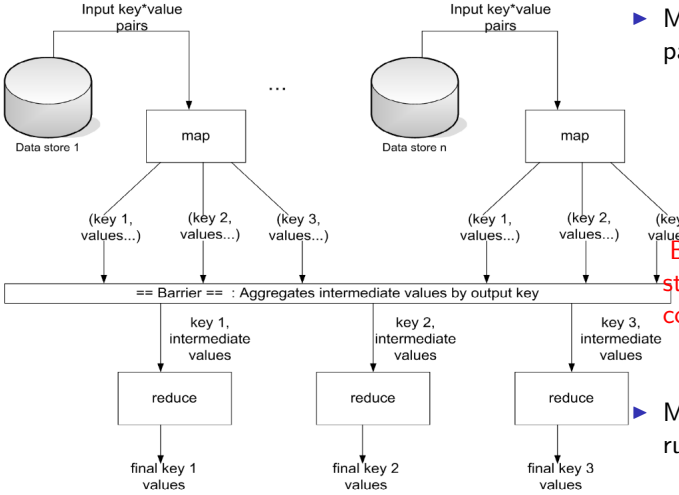
- Operates on collection of key-value pairs
  - Inputs, intermediate results, outputs
- **Map phase**: Apply user-defined Map function to each k/v pair
  - Map::(Key1, Value1) $\rightarrow$ list(Key2, Value2)
- **Shuffle phase**: Group all Map output pairs with the same key together (collect values)
- **Reduce phase**: Apply user-defined Reduce function for each key (on its values)
  - Reduce :: (Key2, list(Value2)) $\rightarrow$ list (Key3, Value3)

MapReduce parallelization



- Multiple Map functions run in parallel
  - Each working on one element of the data = k/v pair
- Multiple Reduce functions can run in parallel
  - Each on a different key

MapReduce parallelization



- Multiple Map functions run in parallel
  - Each working on one element of the data = k/v pair

Bottleneck: Reduce phase cannot start before map phase is completely finished

- Multiple Reduce functions can run in parallel
  - Each on a different key

WordCount Example

- Task: count how often each word occurs in a text collection
- **Map function**
  - Counts words in a single document
  - (document-id, text) *to* list(word, count)
  - Output list has one element per work in text
- MapReduce then groups counts per word
- **Reduce function**
  - Aggregates counts of word
  - (word, list(count)) $\rightarrow$ list(word, final-count)
  - Here: output list has just one element

WordCount Example: Map

Input

- (1, “One ring to rule them all, one ring to find them.”)
- (2, “One ring to bring them all and in the darkness bind them.”)

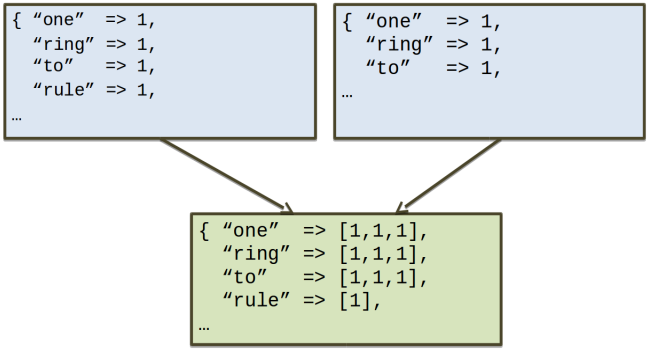
Line 1

```
{ “one”  => 1,
  “ring” => 1,
  “to”   => 1,
  “rule” => 1,
  “them” => 1,
  “all”  => 1,
  “one”  => 1,
  “ring” => 1,
  “to”   => 1,
  “find” => 1,
  “them” => 1 }
```

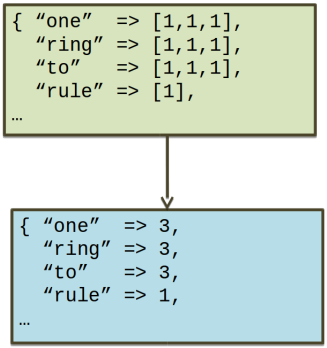
Line 2

```
{ “one”  => 1,
  “ring” => 1,
  “to”   => 1,
  “bring”=> 1,
  “them” => 1,
  “all”  => 1,
  “and”  => 1,
  “in”   => 1,
  “the”  => 1,
  “darkness” => 1,
  “bind” => 1,
  “them” => 1 }
```

WordCount Example: Group by word



WordCount Example: Reduce



WordCount Example: Pseudo code

- Map(Integer docId, String text)

  for each word w in text:
  emit$(w, 1)$

- Reduce(String word, Iterator<Integer> counts)

  int result = 0
  for each count **in** counts:
  result += count
  emit(word, result)

WordCount Example: Pseudo code

- Map(Integer docId, String text)

  for each word w in text:
  emit$(w, 1)$

- Reduce(String word, Iterator<Integer> counts)

  int result = 0
  for each count **in** counts:
  result += count
  emit(word, result)

Question: What are the differences to map() and fold()?

Basic MapReduce program design

- Tasks that can be performed independently on a (large number of) data objects: Map
- Tasks that require combining of multiple data objects: Reduce
- Sometimes it is easier to start program design with Map, sometimes with Reduce
- Select keys and values such that the right objects end up together in the same Reduce invocation
- Might have to partition a complex task into multiple MapReduce sub-tasks

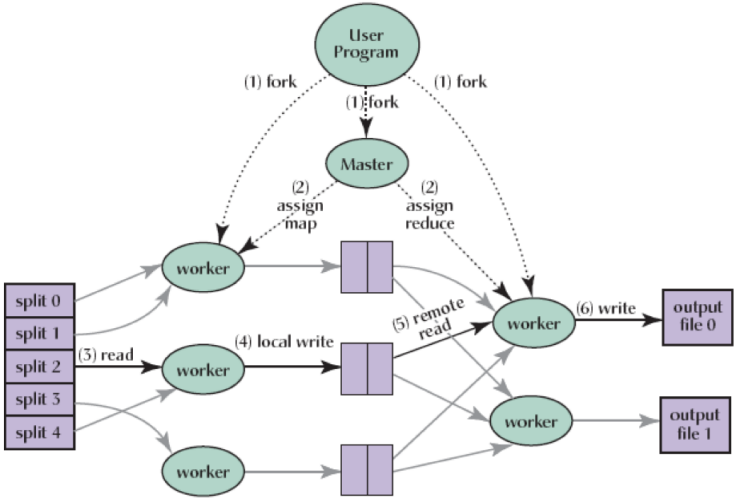
Outline

1. Background
2. Programming Model
3. Architecture
4. Hadoop MapReduce
5. Hadoop HDFS
6. Algorithms & Data Management
7. Discussion

Objectives of Google MapReduce

- Simple programming model
- Batch processing
  - High throughput
  - Often long running jobs
  - Not: low latency
- Scalability
  - 10s, 100s, 1000s of nodes
  - Additionally depends on the problem
- Fault tolerance
  - Cluster of commodity nodes
  - Handle failure of a node while join is running gracefully (**query fault tolerance**)

MapReduce execution overview



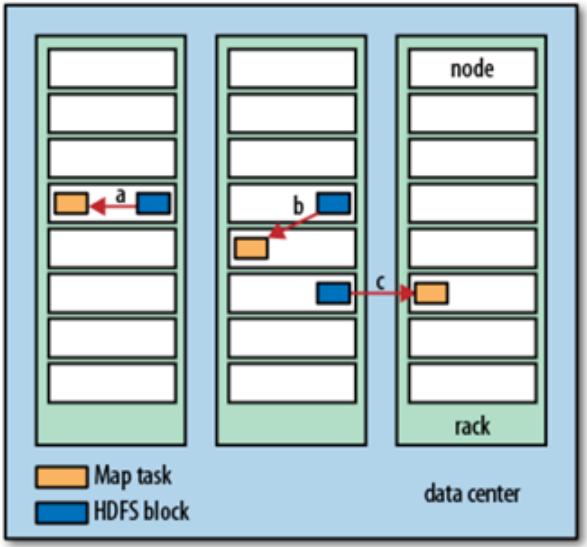
MapReduce execution overview

- Data (often) on distributed file system like GFS, HDFS
- One master, many workers
  - Input data **split** into M **map tasks** (typically 64 MB $\approx$ chunk size in GFS)
  - Reduce phase **partitioned** into R **reduce tasks** (hash(key) mod R)
  - Tasks are assigned to workers dynamically
- Master assigns each map task to a free worker
  - Considers **data locality** to worker when assigning a task
  - Worker reads task input (often from local files) and applies Map operation to each k/v pair
  - Worker produces R local files containing intermediate k/v pairs
- Master assigns each reduce task to a free worker
  - Worker reads intermediate k/v pairs from map workers
  - Worker sorts & applies user's Reduce operation to produce the output in global file system

Data locality

- Move computation to data
  - Small code, large data
  - Goal: Reduce network bandwidth
- Machines run both GFS chunkservers and MR workers
  - In GFS, each chunk stored at 3 machines
  - MR master schedules map tasks based on the location of these chunks
- Goal: run map task on machine that stores input chunk
  - Then map task reads data locally (**local fetch**)
  - Thousands of machines can read at local disk speed; read rate not limited by network

Local, rack-local, off-rack map tasks



Choosing M and R

- M = number of map tasks
- R = number of reduce tasks
- Larger M, R creates smaller tasks
  - Easier load balancing
  - Faster recovery
- But: higher cost at master & higher start-up cost
- Recommendation
  - Choose M so that split size is approximately 64MB
  - Choose R a small multiple of the number of workers; alternatively choose R a little smaller than #workers to finish reduce phase in one "wave"

Fault tolerance

- On worker failure
  - Master detects failure via periodic heartbeats
  - Both completed and in-progress map tasks on that worker should be re-executed
  - Only in-progress reduce tasks on that worker should be re-executed
  - All reduce workers will be notified about any map re-executions
- On master failure
  - State is check-pointed to GFS
  - New master recovers & continues

Stragglers

- Problem: **stragglers** (= slow workers) significantly lengthen the completion time
- Solution: Close to completion, spawn **backup copies** of the remaining in-progress tasks
  - Also called **speculative execution**
  - Whichever one finishes first, "wins"
- Additional cost: a few percent more resource usage
- Example: A sort program without backup = 44% longer

Practical extensions

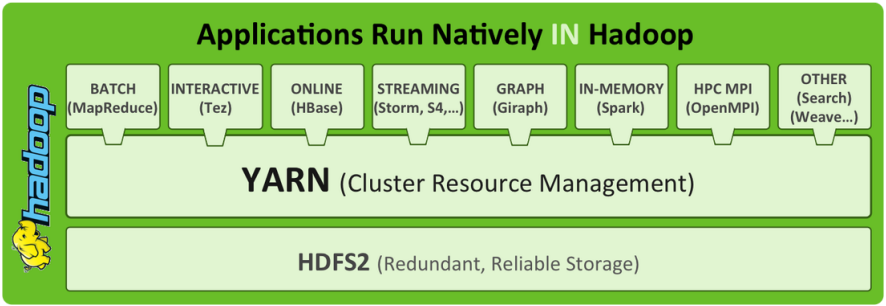
- Reduce phase is optional (if omitted, **Map-only job**)
- User-specified **combiner functions** for partial combination within a map task can save network bandwidth ($\sim$ mini-reduce)
- User-specified **partitioning functions** for mapping intermediate key values to reduce workers (by default: hash(key) mod R)
  - Example: hash(hostname(urlkey)) mod R
- Ordering guarantees: Processing intermediate k/v pairs in increasing order
  - Example: reduce of WordCount outputs ordered results
- Custom input and output format handlers
- Single-machine execution option for testing & debugging
- More later ...

Outline

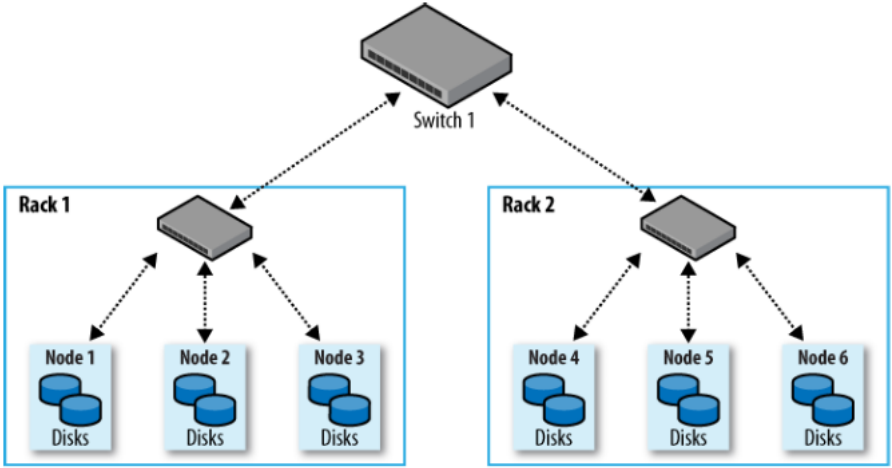
1. Background
2. Programming Model
3. Architecture
4. Hadoop MapReduce
5. Hadoop HDFS
6. Algorithms & Data Management
7. Discussion

What is Hadoop?

- Framework for distributed processing of large datasets
  - Originally, Hadoop 1: MapReduce + HDFS (**our focus**)
  - Hadoop 2: many applications + YARN
  - Now: Hadoop 3

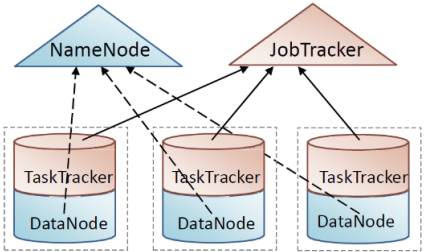


Typical MapReduce cluster configuration

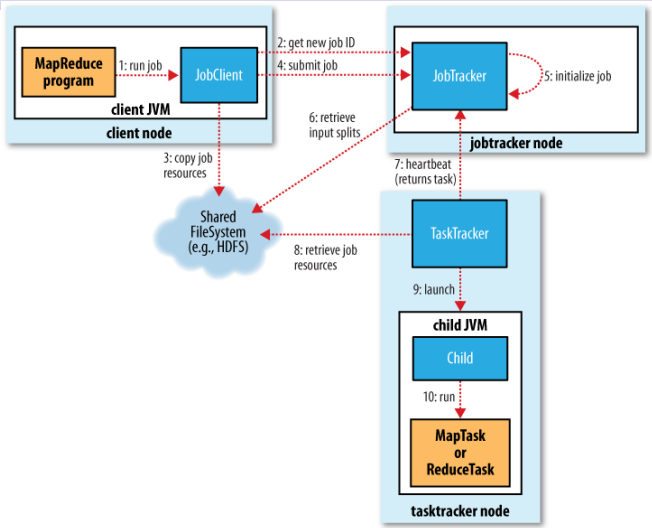


Hadoop main services

- Hadoop DFS (HDFS)
  - **Namenode**: namespace and block management
  - **Datanode**: block replica container
- MapReduce
  - **Jobtracker**: client communication, job scheduling, resource management, lifecycle coordination
  - **Tasktracker**: task execution and management



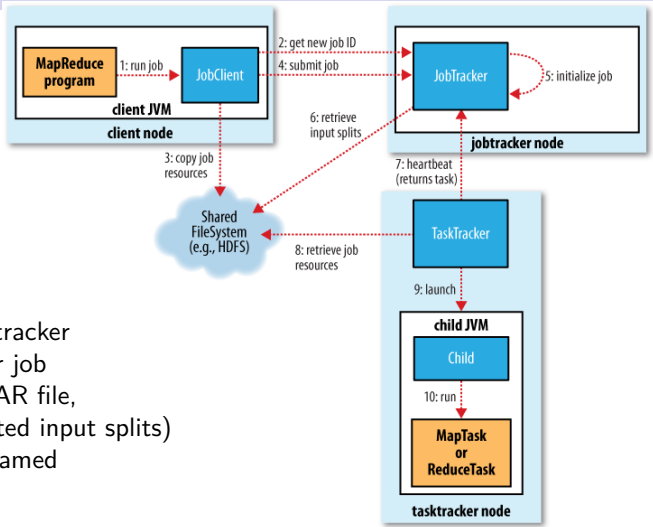
Hadoop job execution



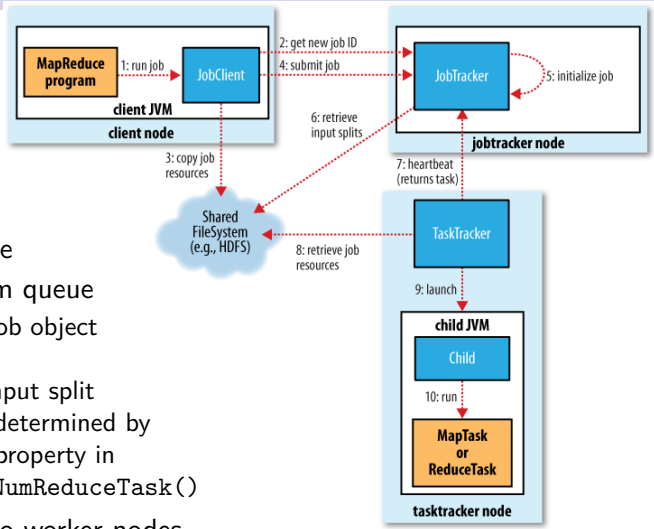
Hadoop job execution

Job initialization (1–4)

- Client submits MapReduce job through `job.submit()` call
- **Job submission process**
  - Get new `jobId` from jobtracker
  - Determine input splits for job
  - Copy job resources(job JAR file, configuration file, computed input splits) to HDFS into directory named after the `jobId`
  - Inform jobtracker that job is ready for execution



Hadoop job execution



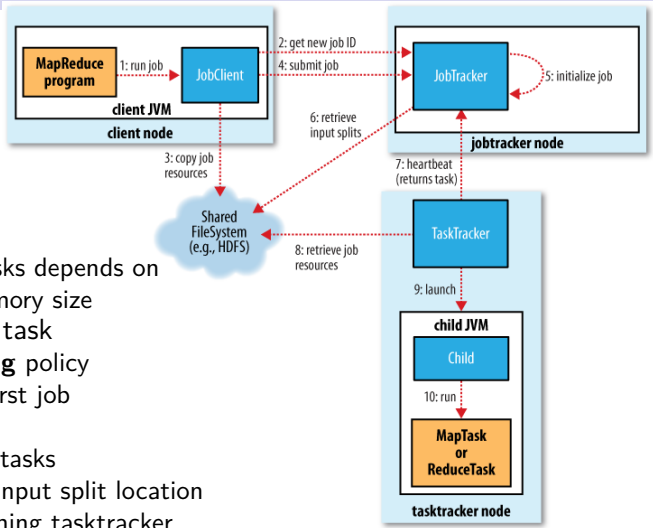
Job initialization (5–6)

- Jobtracker puts ready job into internal queue
- **Job scheduler** pics job from queue
  - Initializes it by creating job object
  - Creates list of task
  - One map task for each input split
  - Number of reduce tasks determined by `mapred.reduce.tasks` property in Job, which is set by `setNumReduceTask()`
- Tasks need to be assigned to worker nodes

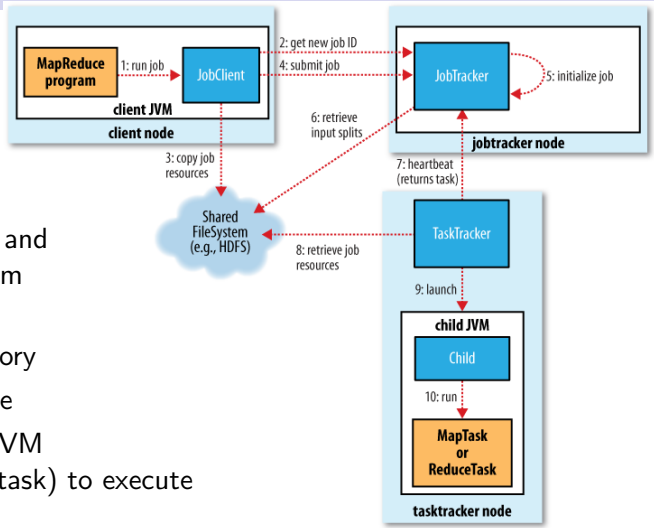
Hadoop job execution

Task assignment (7)

- Tasktrackers send heartbeats to jobtrackers
  - Also indicate if ready to run new tasks
  - Number of "slots" for tasks depends on number of cores and memory size
- Jobtracker replies with new task
  - Based on **task scheduling** policy (e.g., choose task from first job in priority-queue)
  - Map tasks before reduce tasks
  - Choose Map task where input split location is closest to machine running tasktracker instance (data-local < rack-local < off-rack)
  - Choose reduce task arbitrary (no data locality)



Hadoop job execution



Task execution (8–10)

- Tasktracker copies job JAR and other configuration data from HDFS to local disk
- Creates local working directory
- Creates TaskRunner instance
- TaskRunner launches new JVM (or reuses on from another task) to execute the JAR

Hadoop job execution

Progress monitoring

- Task report **progress** to tasktracker
  - As a part of hearbeat
  - Indicates that task is still alive
  - Less frequently: sends "counters" (e.g., #output Map records written, user defined)
- Jobtracker computes **global status** of job progress
  - Job client polls jobtracker regularly for status
  - Visible on console and web UI

Hadoop job execution

Handling task failures

- Failures
  - **Task errors** reported to tasktracker and logged
  - **Hanging task** detected through timeout (then killed)
- Jobtracker will automatically **reschedule** failed task
  - Tries up to `mapred.map.max.attempts` many times (similar for reduce)
  - Job is aborted when task failure rate exceeds `mapred.max.map.failures.percent` (similar for reduce)

Hadoop job execution

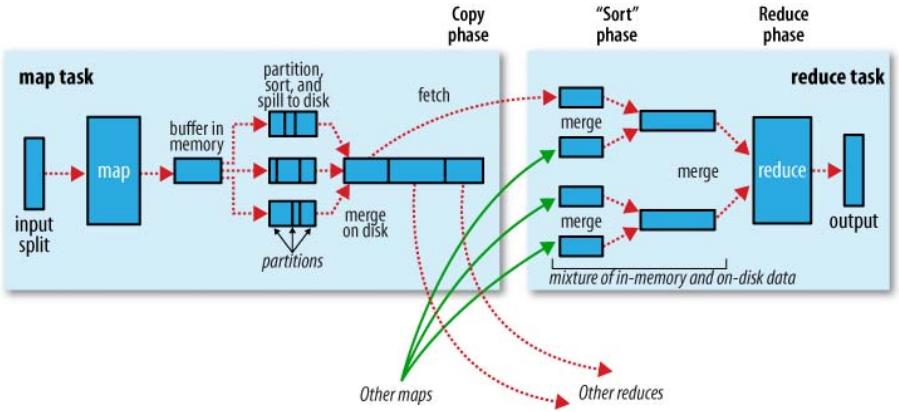
Handling TaskTracker/JobTracker failures

- **Tasktracker failure** detected by jobtracker from missing heartbeat messages
  - Jobtracker re-schedules map tasks and incomplete reduce tasks from that tasktracker
- Hadoop cannot deal with **jobtracker failure**
  - Single point of failure
  - Unlikely to happen (just one machine)
  - Improvements in YARN (hadoop 2/3)

Moving data from mappers to reducers

- **Shuffle and Sort** phase
  - Synchronization barrier between map and reduce phase
  - Often one of the most expensive parts of a MapReduce execution
- Mappers need to separate (and pre-sort) output intended for different reducers
- Reducers need to
  - Collect their data from all mappers
  - Group it by key
  - Sort: Keys at each reducer are processed in order
- Partitioning and sorting can be customized

Shuffle & sort overview



Shuffle & sort example

- Example MapReduce program

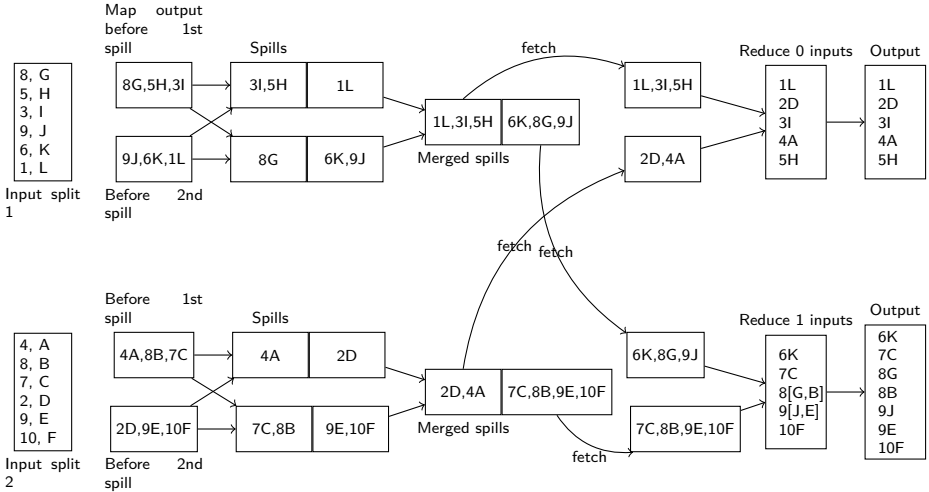
```
1: MAP(Key k, Value v)
2: EMIT(k, v)
3:
4: REDUCE(Key k, list⟨Value⟩ V)
5: for each v ∈ V do
6:     EMIT(k, v)
7: end for
```

- To assign key-value pairs to partitions (reduce tasks), we use the following hash function

$$\text{hash}(k) = \begin{cases} 0 & \text{if } k \leq 5, \\ 1 & \text{if } k > 5. \end{cases}$$

- Consider two input splits (6 k-v pairs each), in-memory buffer of a Map task can hold up to 3 k-v pairs, and two reduce task (0 and 1)

Shuffle & sort example



MapReduce development steps

- Write Map and Reduce functions
  - Create unit tests
- Write driver program to run a job
  - Can run from IDE with small data subset for testing
  - If test fails, use IDE for debugging
  - Update unit tests and Map/Reduce if necessary
- Once program works on small test set, run it on full data set
  - If there are problems, update tests and code accordingly
- Fine-tune code, do some profiling

Outline

1. Background
2. Programming Model
3. Architecture
4. Hadoop MapReduce
5. Hadoop HDFS
6. Algorithms & Data Management
7. Discussion

Hadoop Distributed File System (HDFS)

- Distributed file systems manage the storage across a network of machines
- HDFS is Hadoop's flagship filesystem
- Hadoop has a general-purpose file system abstraction
  - Can work with several storage systems
  - E.g., local file system, HDFS, Amazon S3, ...

HDFS design objective

- Very large files
- Streaming data access
  - Write once, read-many-times
  - Time to read whole file is important
  - Reads largely sequentially
- Commodity hardware
  - Fault-tolerance needed
- Not well-suited for
  - Low-latency data access
  - Lots of small files
  - Multiple writers, arbitrary file modifications

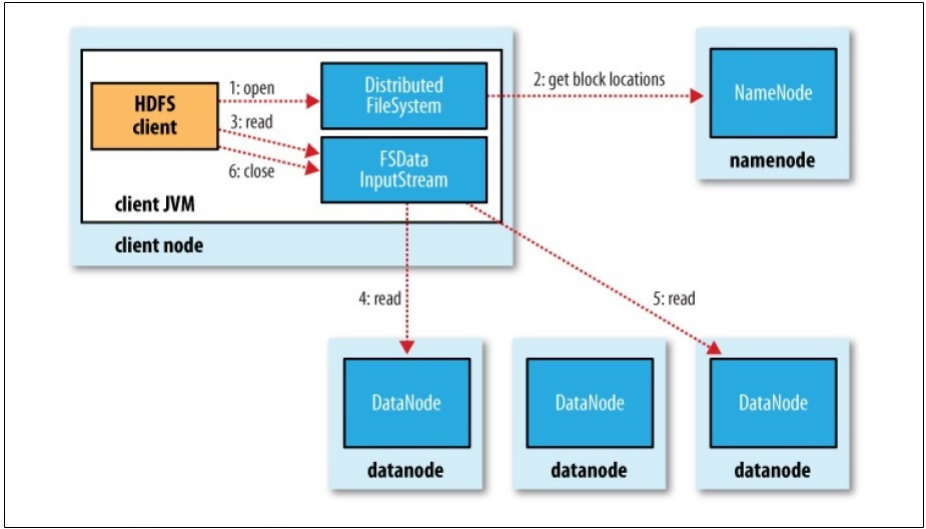
Blocks

- HDFS files are broken into block-sized chunks
  - 64MB by default
- With this block abstraction
  - A file can be larger than a single disk in the network
  - Storage system is simplified (metadata for blocks)
  - Replication for fault-tolerance and availability is facilitated

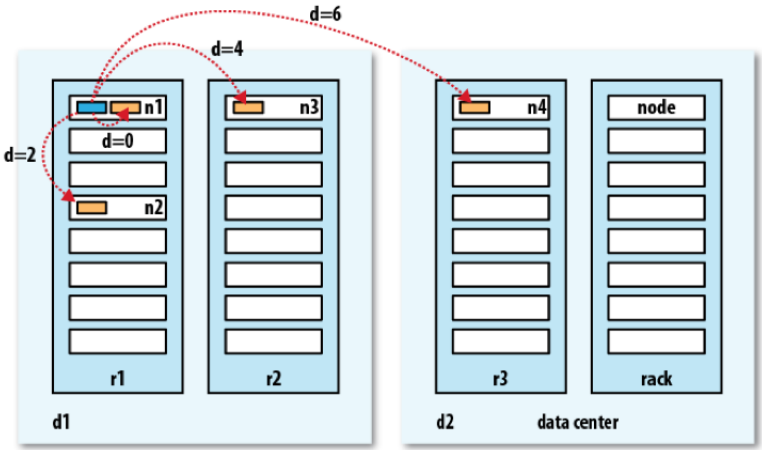
Namdenodes and Datanodes

- Two types of HDFS nodes
  - One **namenode** (the master)
  - Multiple **datanodes** (workers)
- Namenode manages the filesystem namespace
  - File system tree and metadata, stored persistently
  - Block locations, stored transiently
- Datanodes store and retrieve data blocks
  - When requested from clients or namenode
  - Also send list of stored blocks periodically to namenode

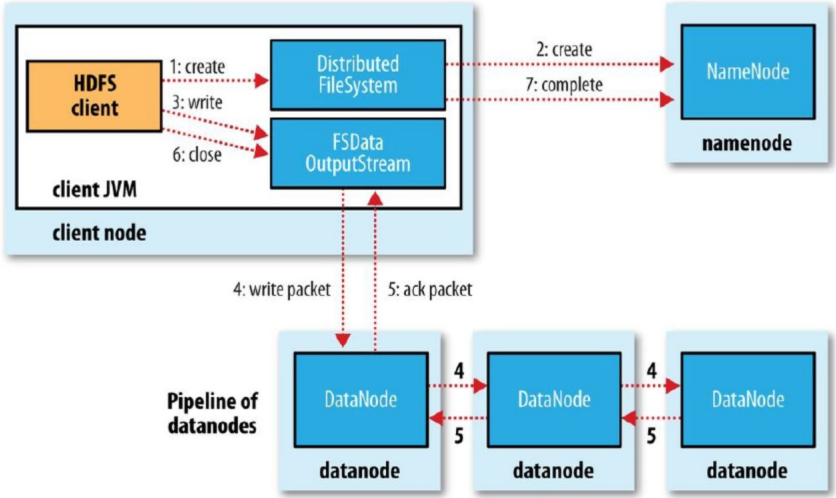
Reading from HDFS



Network distance in Hadoop

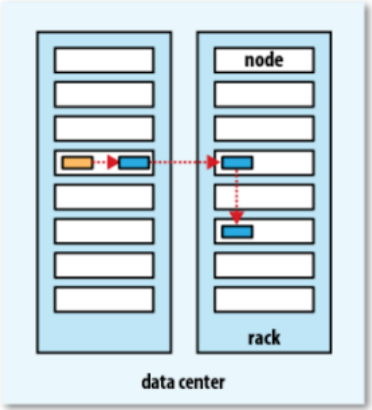


Writing to HDFS



Replica placement

- Issues
  - Reliability
  - Write and read bandwidth
  - Block distribution
- Default strategy
  - First replica: on the client node
  - Second: random, off rack
  - Third: same rack as second, different node
  - More: random



Outline

1. Background
2. Programming Model
3. Architecture
4. Hadoop MapReduce
5. Hadoop HDFS
6. Algorithms & Data Management
7. Discussion

Combiner functions

- **Combiners** optionally "pre-reduce" (or "merge") mapper output
  - Can save network bandwidth + work at reducers
  - Same input type as Reduce / output type as Map
  - Usually same as reduce function
  - But does not replace reduce step (since it sees only local data)
  - Does not change program semantics
- Works well esp. for distributive functions

WordCount Example: Map

Input

- (1, “One ring to rule them all, one ring to find them.”)
- (2, “One ring to bring them all and in the darkness bind them.”)

Line 1 (Map task 1)

```
{ “one”  => 1,
  “ring” => 1,
  “to”   => 1,
  “rule” => 1,
  “them” => 1,
  “all”  => 1,
  “one”  => 1,
  “ring” => 1,
  “to”   => 1,
  “find” => 1,
  “them” => 1 }
```

Line 2 (Map task 2)

```
{ “one”  => 1,
  “ring” => 1,
  “to”   => 1,
  “bring”=> 1,
  “them” => 1,
  “all”  => 1,
  “and”  => 1,
  “in”   => 1,
  “the”  => 1,
  “darkness” => 1,
  “bind” => 1,
  “them” => 1 }
```

WordCount Example: Combine input

Partition 1

```
{ “one”  => [1,1]
  “ring” => [1,1],
  “to”   => [1,1],
  “rule” => [1],
  “them” => [1,1],
  “all”  => [1],
  “find” => [1] }
```

Partition 2

```
{ “one”  => [1],
  “ring” => [1],
  “to”   => [1],
  “bring”=> [1],
  “them” => [1,1],
  “all”  => [1],
  “and”  => [1],
  “in”   => [1],
  “the”  => [1],
  “darkness” => [1],
  “bind” => [1] }
```

WordCount Example: Combine output

Partition 1

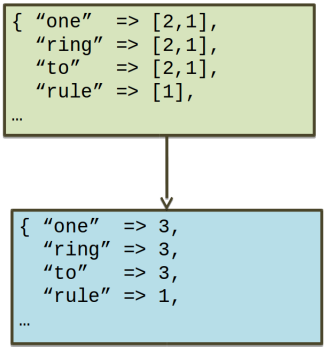
```
{ “one”  => 2
  “ring” => 2,
  “to”   => 2,
  “rule” => 1,
  “them” => 2,
  “all”  => 1,
  “find” => 1 }
```

Partition 2

```
{ “one”  => 1,
  “ring” => 1,
  “to”   => 1,
  “bring”=> 1,
  “them” => 2,
  “all”  => 1,
  “and”  => 1,
  “in”   => 1,
  “the”  => 1,
  “darkness” => 1,
  “bind” => 1 }
```

Here: Combine function = Reduce function

WordCount Example: Reduce



WordCount Example: Psuedo code

- Map(Integer docId, String text)

  for each word w in text:
  emit$(w, 1)$

- Combine(String word, Iterator<Integer> counts)

  int result = 0
  for each count **in** counts:
  result += count
  emit(word, result)

- Reduce(String word, Iterator<Integer> counts)

  int result = 0
  for each count **in** counts:
  result += count
  emit(word, result)

Combiner caveats

- Some tasks do not admit a (useful) Combiner
- Distributive and algebraic aggregates (per key) usually lead to useful Combiners
  - Example: average
  - Example: output if sum(values)>threshold
    - Can't decide whether threshold is met based on local info
    - Combiners sum up (useful), reducers sum up and check threshold
- Holistic aggregates also work, but Combiner may not be beneficial
  - Example: median

Partitioning, grouping, sorting

Hadoop allows flexibility in

- How to **partition** keys to Reduce tasks
  - Which key goes to which partition?
  - HashPartitioner
- How to **sort** keys within a partition
  - In what order are keys processed?
  - Default: sort by key
- How to **group** keys for Reduce function
  - Which k/v pairs go to the same Reduce invocation?
  - Default: same key

Example

- Weather data
  - Input: Temperature for each day
  - Desired output: Temperatures per year in decreasing order

    1900 35C ← max for year 1900
    1900 34C
    1900 34C
    ...
    1901 36C ← max for year 1901
    1901 35C

  - I.e., (1900,[35,34,34...]),(1901,[...]),...

Example: Options

Option 1

- Use year as key, temperature as value
- Sort in Reduce code
- Works, but: high memory consumption in Reduce

Example: Options

Option 1

- Use year as key, temperature as value
- Sort in Reduce code
- Works, but: high memory consumption in Reduce

Option 2

- Use custom partitioning, grouping, sorting
- Now Hadoop does all the work (can use IdentityReducer)

Example: Composite keys

- We want to sort by values, which Hadoop cannot do
- Idea: Use **composite key** (year, temp)
  - But also: output temperature as value
- Example Map output

Key	**Value**	**Partition**
(1900, 35C)	35C	1
(1900, 34C)	34C	2
(1900, 34C)	34C	2
.	.	.
.	.	.
.	.	.
(1901, 36C)	36C	1
(1901, 35C)	35C	2

Example: Composite keys

- We want to sort by values, which Hadoop cannot do
- Idea: Use **composite key** (year, temp)
  - But also: output temperature as value
- Example Map output

Key	**Value**	**Partition**
(1900, 35C)	35C	1
(1900, 34C)	34C	2
(1900, 34C)	34C	2
.	.	.
.	.	.
.	.	.
(1901, 36C)	36C	1
(1901, 35C)	35C	2

- Problem: Map output not partitioned by year

Example: Custom partitioner/sort

- Solution
  - Write a custom **partitioner** that partitions based on year only (recall: year = part of key)
  - Write a custom **key comparator** that "sorts" by year ascending and temperature descending
- Example

Key	**Value**	**Partition**	**Group**
(1900, 35C)	35C	1	1
(1900, 34C)	34C	1	2
(1900, 34C)	34C	1	2
.	.	.	.
.	.	.	.
.	.	.	.
(1901, 36C)	36C	2	1
(1901, 35C)	35C	2	2

Example: Custom partitioner/sort

- Solution
  - Write a custom **partitioner** that partitions based on year only (recall: year = part of key)
  - Write a custom **key comparator** that "sorts" by year ascending and temperature descending
- Example

Key	**Value**	**Partition**	**Group**
(1900, 35C)	35C	1	1
(1900, 34C)	34C	1	2
(1900, 34C)	34C	1	2
.	.	.	.
.	.	.	.
.	.	.	.
(1901, 36C)	36C	2	1
(1901, 35C)	35C	2	2

- Problem! Partitions correct, but not grouped by year

Example: Custom grouping

- Solution (contd.)
  - Define a custom **grouping comparator** method that considers only the year
  - Example

Key	**Value**	**Partition**	**Group**	**Reduce input**
(1900, 35C)	35C	1	1	(1900,35C)[35C,34C,...]
(1900, 34C)	34C	1	1	
(1900, 34C)	34C	1	1	
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
(1901, 36C)	36C	2	1	(1901,36C)[36C,35C,...]
(1901, 35C)	35C	2	1	

Example: Custom grouping

- Solution (contd.)
  - Define a custom **grouping comparator** method that considers only the year
  - Example

Key	Value	Partition	Group	Reduce input
(1900, 35C)	35C	1	1	(1900,35C)[35C,34C,...]
(1900, 34C)	34C	1	1	
(1900, 34C)	34C	1	1	
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
(1901, 36C)	36C	2	1	(1901,36C)[36C,35C,...]
(1901, 35C)	35C	2	1	

- In general: grouping comparator should be consistent with key comparator
- Note: similar trick for median temp per year

SQL in MapReduce

- MapReduce is powerful enough to run basic SQL queries
  - Although that's not its original intention

SQL in MapReduce

- MapReduce is powerful enough to run basic SQL queries
  - Although that's not its original intention
- Easy case: single table, no sorting
  - `select prodId, sum(amt) from sales where depId = 10 group by prodId`

SQL in MapReduce

- MapReduce is powerful enough to run basic SQL queries
  - Although that's not its original intention
- Easy case: single table, no sorting
  - `select prodId, sum(amt) from sales where depId = 10 group by prodId`
  - Map: selection, projection w/o duplicate elimination
  - MapReduce framework: grouping
  - Reduce: aggregation, having, duplicate elimination

Sorting

- How to produce a total order?
  - `select prodId, amt order by amt`

Sorting

- How to produce a total order?
  - `select prodId, amt order by amt`
- Naïve: use a single reducer
  - Looses parallelism

Sorting

- How to produce a total order?
  - `select prodId, amt order by amt`
- Naïve: use a single reducer
  - Looses parallelism
- Better: **parallel external merge sort**
  - Mapper: output (sort attribute, tuple)-pairs
  - Use custom partitioner to run **range partitioning**
  - Use custom key comparator for **sort order**
  - Reducer: outputs all tuples
  - Problem: need partitioning vector
    - Often determined from sample of tuples
    - Hadoop directly supports sampling

Joins (1)

- Possible, but fairly involved
- `select * from R, S where R.key = S.key`

Joins (1)

- Possible, but fairly involved
- `select * from R, S where R.key = S.key`

Map-side join (Broadcast join)

- Great when one relation is small (say, S)
- Map over tuples of R
- Map function "knows" S entirely; e.g., broadcast via Hadoop's **distributed cache**
- Map joins each input tuple with S and outputs results
- No Reduce

Joins (2)

- Possible, but fairly involved
- `select * from R, S where R.key = S.key`

Reduce side join

- Map over tuples from R and from S, i.e., use **multiple inputs**
- Annotate each tuple with its source relation, i.e., Mapper outputs: (join key, (relation name, tuple))
- For each join key, Reduce obtains all tuples with that key (from both relations) $\rightarrow$ can join
- That's a **parallel hash join**
- Can use Hadoop's custom partitioning, sorting, grouping facilities to improve efficiency

Outline

1. Background
2. Programming Model
3. Architecture
4. Hadoop MapReduce
5. Hadoop HDFS
6. Algorithms & Data Management
7. Discussion

MapReduce and parallel DBMS

	Traditional RDBMS	MapReduce
Data size	GBs	PBs
Access	Interactive and batch	Batch
Updates	Read and write many times	Write once, read many times
Structure	Static schema	Dynamic schema
Integrity	High	Low
Scaling	Nonlinear	Linear

Benefits of MapReduce

- Simple model (but: see criticism)
- Scalable (but: depends on task)
- High throughput
- Query fault tolerance
- Supports “data first, purpose later” paradigm of Big Data
  - Dynamic schema, schema-on-read

Criticism

- Low level, restricted programming model
- No query language
- No indices, no updates, no transactions
- Slow
- Reinvents many DBMS technologies

Judge for yourself!

For which tasks is a DBMS the right tool, for which tasks MapReduce, and for which neither?

Literature

- T.White Hadoop – The Definitive Guide O'Reilly
- J.Lin, C. Dyer Data-Intensive Text Processing with MapReduce Morgan and Claypool