

MapReduce

Thoai Nam

High Performance Computing Lab (HPC Lab)

Faculty of Computer Science and Engineering

HCMC University of Technology

Ref

- MapReduce algorithm design, Jimmy Lin



processes 20 PB a day (2008)
crawls 20B web pages a day (2012)



>10 PB data, 75B DB
calls per day (6/2012)

>100 PB of user data +
500 TB/day (8/2012)



S3: 449B objects, peak 290k
request/second (7/2011)
IT objects (6/2012)



640K ought to be
enough for anybody.

HPC Lab-CSE-HCMUT



150 PB on 50k+ servers
running 15k apps (6/2011)



Wayback Machine: 240B web
pages archived, 5 PB (1/2013)

LHC: ~15 PB a year



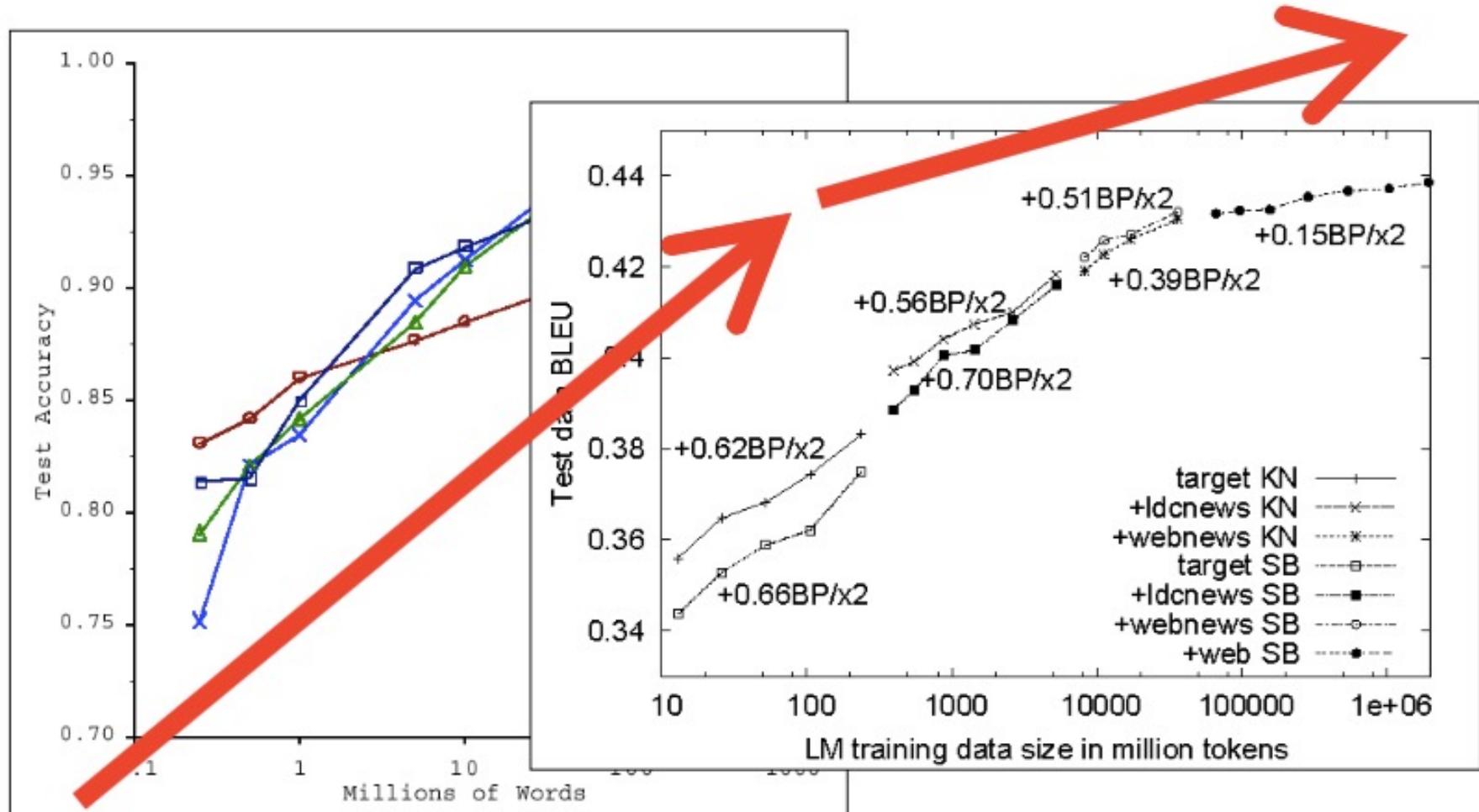
LSST: 6-10 PB a year
(~2015)



How much data?

No data like more data!

s/knowledge/data/g;



A wide-angle photograph of a massive server room. The floor is a polished grey tile. Numerous server racks are arranged in long rows, stretching into the distance. The racks are illuminated from within, with many glowing blue and yellow lights. The ceiling is a complex network of steel beams and pipes, also bathed in a blue light. The overall atmosphere is one of a high-tech, industrial data center.

MapReduce

Typical Big Data Problem

- Iterate over a large number of records

Map Extract something of interest from each

- Shuffle and sort intermediate results
- Aggregate intermediate results
- Generate final output

Reduce

Key idea: provide a functional abstraction for these two operations

MapReduce: A Real World Analogy

Coins Deposit



MapReduce: A Real World Analogy

Coins Deposit



Coins Counting Machine

MapReduce: A Real World Analogy

Coins Deposit

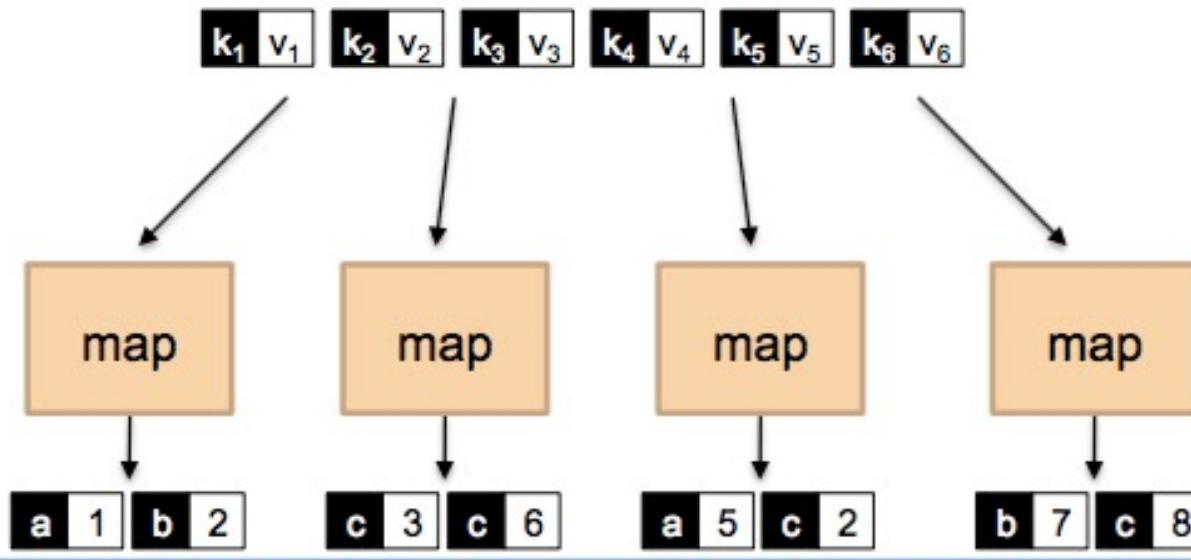


Mapper: Categorize coins by their face values

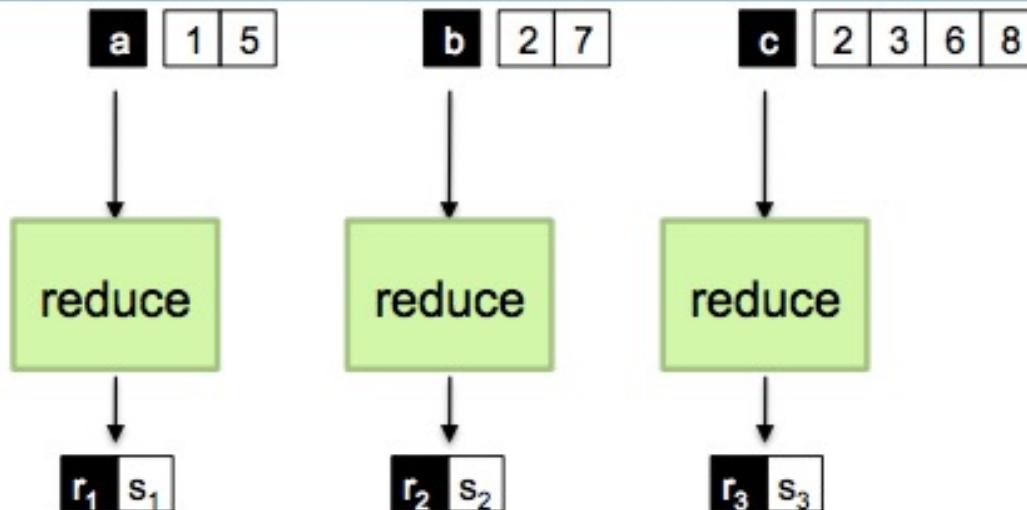
Reducer: Count the coins in each face value *in parallel*

MapReduce

- Programmers specify two functions:
map (k_1, v_1) \rightarrow [$<k_2, v_2>$]
reduce ($k_2, [v_2]$) \rightarrow [$<k_3, v_3>$]
(All values with the **same key** are sent to the same reducer)
- The execution framework handles everything else...



Shuffle and Sort: aggregate values by keys



MapReduce

- Programmers specify two functions:

Map (k_1, v_1) $\rightarrow \langle k_2, v_2 \rangle^*$

Reduce ($k_2, \text{list } (v_2)$) $\rightarrow \text{list } (v_3)$

(All values with the **same key** are sent to the same reducer)

- The execution framework handles everything else...

MapReduce “runtime”

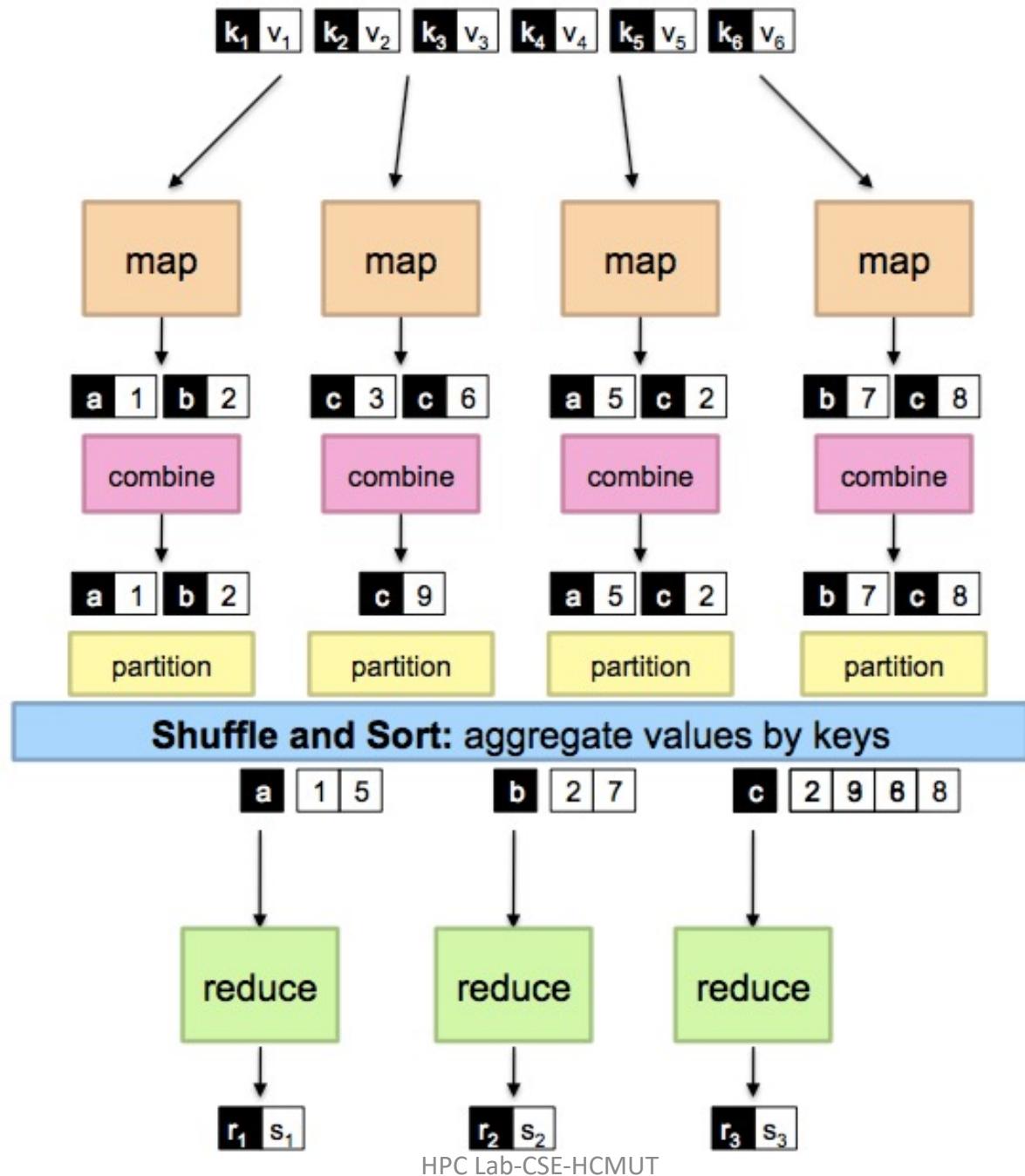
- Handles scheduling
 - Assigns workers to map and reduce tasks
- Handles “data distribution”
 - Moves processes to data
- Handles synchronization
 - Gathers, sorts, and shuffles intermediate data
- Handles errors and faults
 - Detects worker failures and restarts
- Everything happens on top of a distributed file system

Synchronization & ordering

- Barrier between map and reduce phases
 - But intermediate data can be copied over as soon as mappers finish
- Keys arrive at each reducer in sorted order
 - No enforced ordering *across* reducers

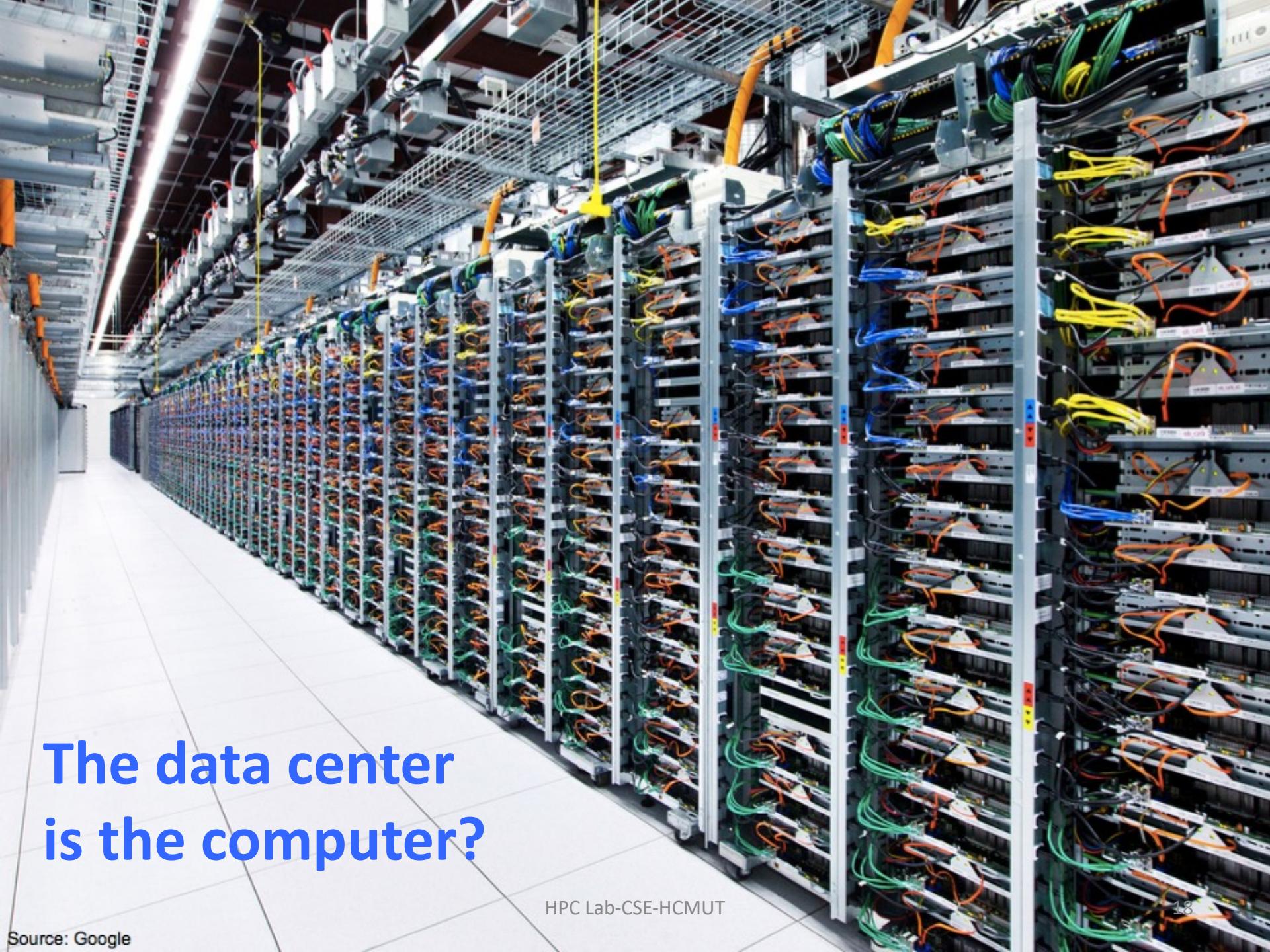
MapReduce

- Programmers specify two functions:
Map (k_1, v_1) \rightarrow $\langle k_2, v_2 \rangle^*$
Reduce ($k_2, \text{list}(v_2)$) \rightarrow $\text{list}(v_3)$
(All values with the **same key** are sent to the same reducer)
- The execution framework handles everything else...
- Not quite...usually, programmers also specify:
partition ($k_2, \text{number of partitions}$) \rightarrow **partition for** k_2
 - Often a simple hash of the key, e.g., $\text{hash}(k') \bmod n$
 - Divides up key space for parallel reduce operations
combine (k_2, v_2) \rightarrow $\langle k_2, v_2 \rangle^*$
 - Mini-reducers that run in memory after the map phase
 - Used as an optimization to reduce network traffic



What's the big deal?

- Developers need the right level of abstraction
 - Moving beyond the von Neumann architecture
 - We need better programming models
- Abstractions hide low-level details from the developers
 - No more race conditions, lock contention, etc.
- MapReduce separating the *what* from *how*
 - Developer specifies the computation that need to be performed
 - Execution framework (“runtime”) handles actual execution



The data center
is the computer?

MapReduce can refers to...

- The programming model
- The execution framework (aka “runtime”)
- The specific implementation

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, etc.
 - The *de facto* big data processing platform
 - Rapidly expanding software ecosystem
- Lots of custom research implementations
 - For GPUs, cell processors, etc.



MapReduce algorithm design

- The execution framework handles “everything else”...
 - Scheduling: assigns workers to map and reduce tasks
 - “Data distribution”: moves processes to data
 - Synchronization: gathers, sorts, and shuffles intermediate data
 - Errors and faults: detects worker failures and restarts
- Limited control over data and execution flow
 - All algorithms must expressed in m,r,c,p
- You don’t know:
 - Where mappers and reducers run
 - When a mapper or reducer begins or finishes
 - Which input a particular mapper is processing
 - Which intermediate key a particular reducer is processing

Apache Hadoop

Data volumes: Google Example

- Analyze 10 billion web pages
- Average size of a webpage: 20KB
- Size of the collection: 10 billion x 20KBs = 200TB
- HDD hard disk read bandwidth: 150MB/sec
- Time needed to read all web pages (without analyzing them): 2 million seconds = more than 15 days
- A single node architecture is not adequate

Data volumes: Google Example with SSD

- Analyze 10 billion web pages
- Average size of a webpage: 20KB
- Size of the collection: 10 billion x 20KBs = 200TB
- SSD hard disk read bandwidth: 550MB/sec
- Time needed to read all web pages (without analyzing them): 2 million seconds = more than 4 days
- A single node architecture is not adequate

Apache Hadoop

- Scalable fault-tolerant distributed system for Big Data
 - Distributed Data Storage
 - Distributed Data Processing
 - Borrowed concepts/ideas from the systems designed at Google (Google File System for Google's MapReduce)
 - Open source project under the Apache license
 - But there are also many commercial implementations (e.g., Cloudera, Hortonworks, MapR)

Hadoop History

- Dec 2004 - Google published a paper about GFS
- July 2005 - Nutch uses MapReduce
- Feb 2006 - Hadoop becomes a Lucene subproject
- Apr 2007 - Yahoo! runs it on a 1000-node cluster
- Jan 2008 - Hadoop becomes an Apache Top Level Project
- Jul 2008 - Hadoop is tested on a 4000 node cluster
- Feb 2009 - The Yahoo! Search Webmap is a Hadoop application that runs on more than 10,000 core Linux cluster
- June 2009 - Yahoo! made available the source code of its production version of Hadoop
- In 2010 Facebook claimed that they have the largest Hadoop cluster in the world with 21 PB of storage
 - On July 27, 2011 they announced the data has grown to 30PB.

Hadoop vs. HPC

- **Hadoop**
 - Designed for **Data intensive workloads**
 - Usually, no CPU demanding/intensive tasks
- **HPC** (High-performance computing)
 - A supercomputer with a high-level computational capacity
 - Performance of a supercomputer is measured in floating-point operations per second (FLOPS)
 - Designed for **CPU intensive tasks**
 - Usually it is used to process "small" data sets

Hadoop: main components

- Core components of Hadoop:
 - Distributed Big Data Processing Infrastructure based on the MapReduce programming paradigm
 - Provides a high-level abstraction view
 - Programmers do not need to care about task scheduling and synchronization
 - Fault-tolerant
 - Node and task failures are automatically managed by the Hadoop system
 - HDFS (Hadoop Distributed File System)
 - High availability distributed storage
 - Fault-tolerant

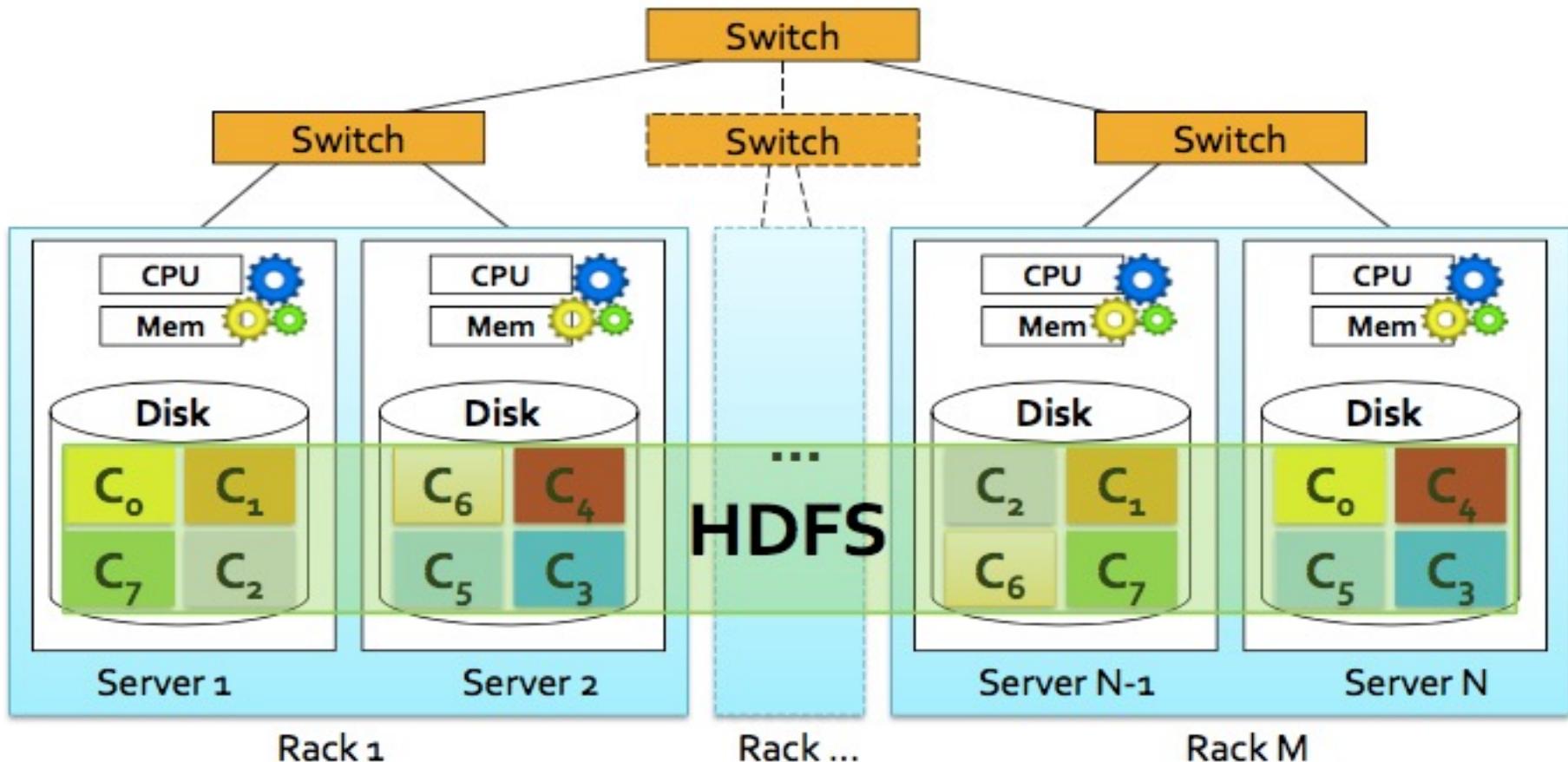
HDFS

(Hadoop Distributed File System)

HDFS

- HDFS is a distributed file system that is fault tolerant, scalable and extremely easy to expand
- HDFS is the primary distributed storage for Hadoop applications
- HDFS provides interfaces for applications to move themselves closer to data
- HDFS is designed to ‘just work’, however a working knowledge helps in diagnostics and improvements

HDFS: a distributed file system

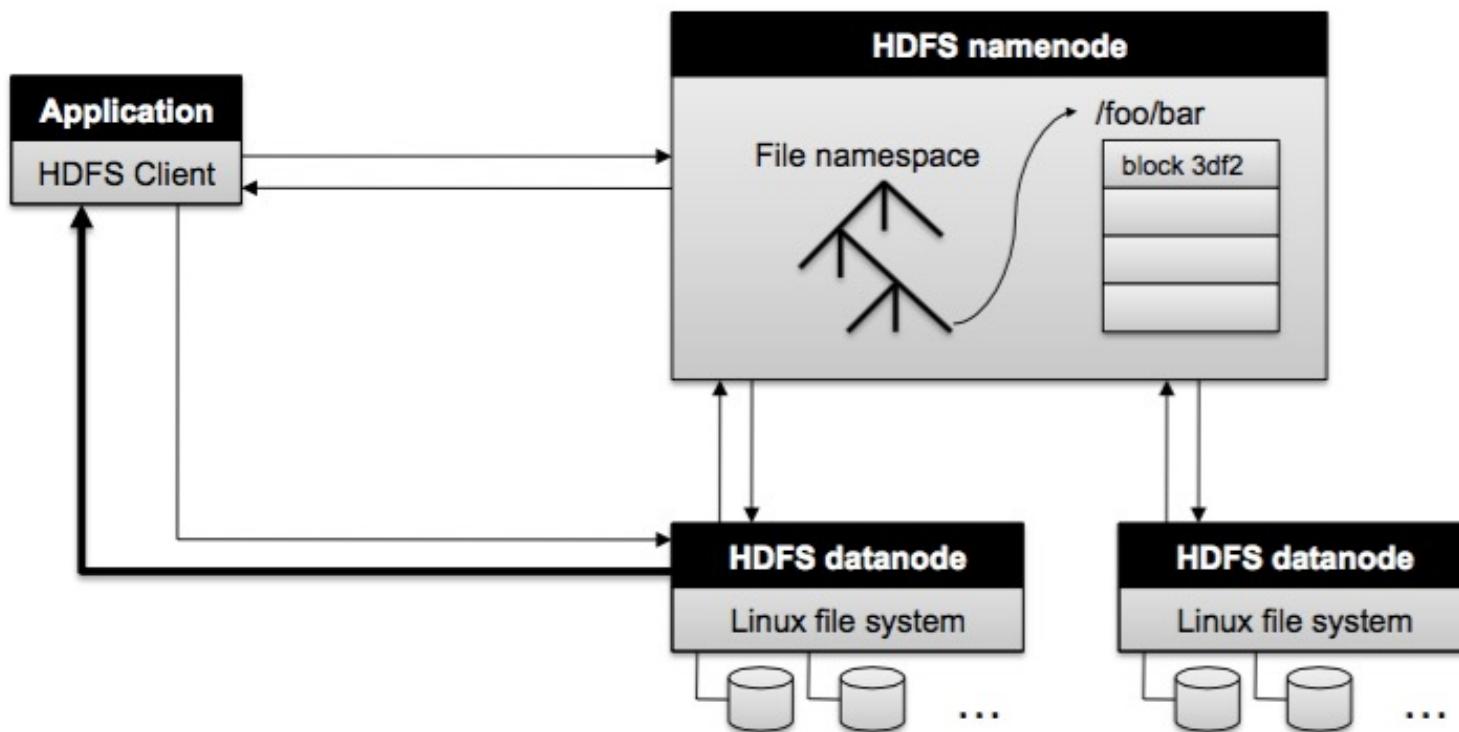


Example with number of replicas per chunk = 2

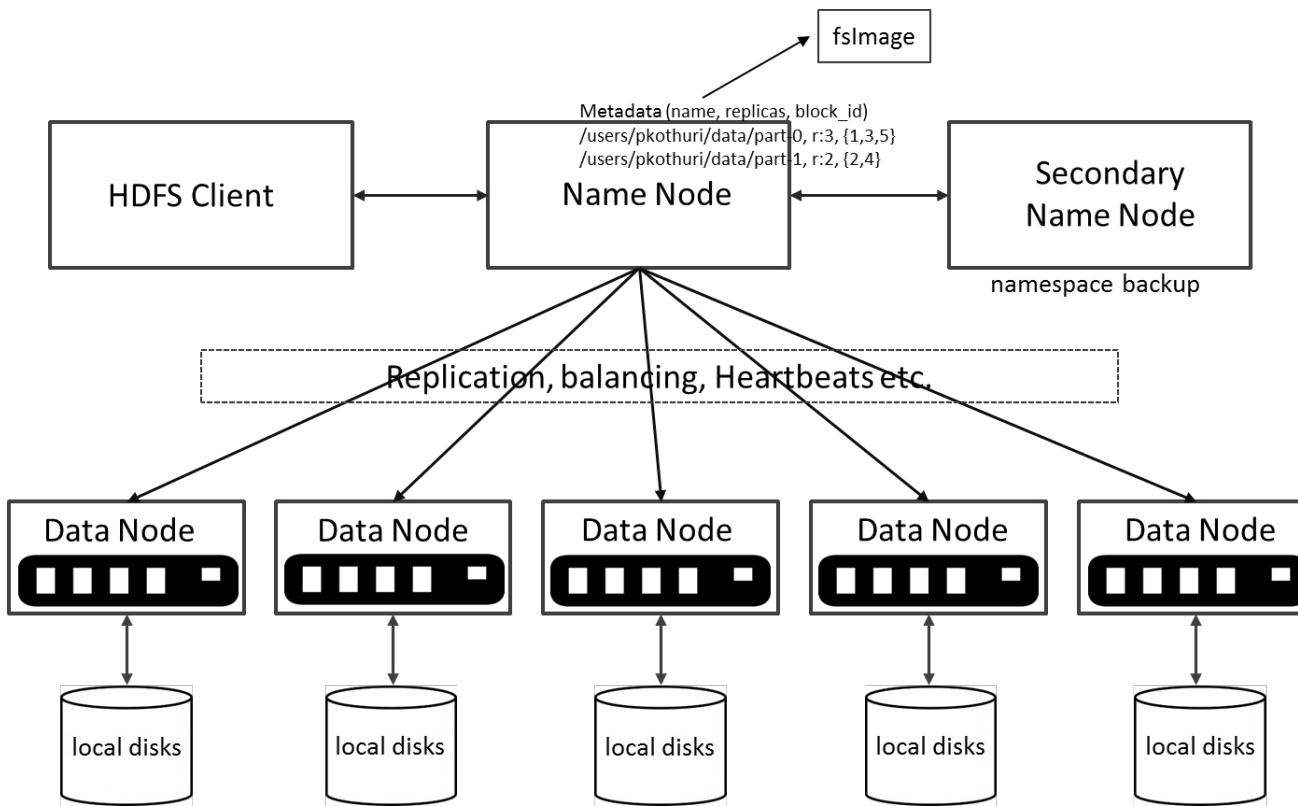
HDFS – Data Organization

- Each file written into HDFS is split into data blocks
- Each block is stored on one or more nodes
- Each copy of the block is called replica
- Block placement policy
 - First replica is placed on the local node
 - Second replica is placed in a different rack
 - Third replica is placed in the same rack as the second replica

HDFS architecture (1)



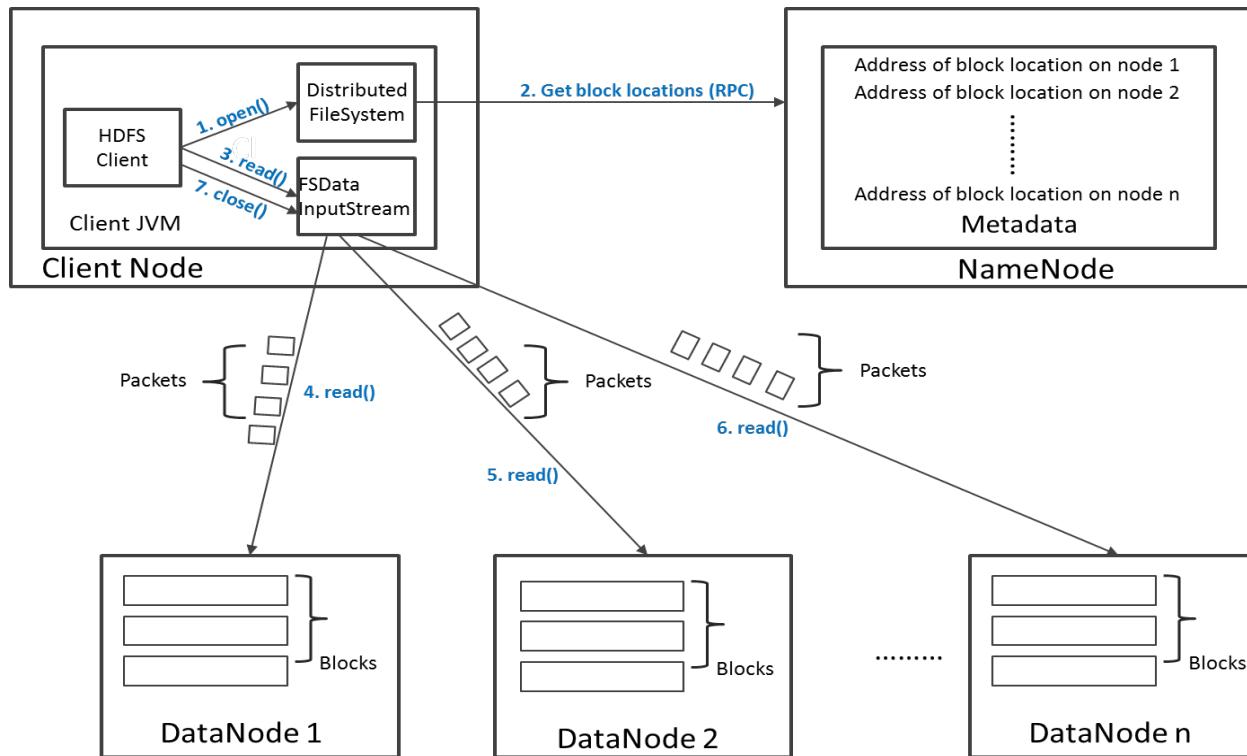
HDFS architecture (2)



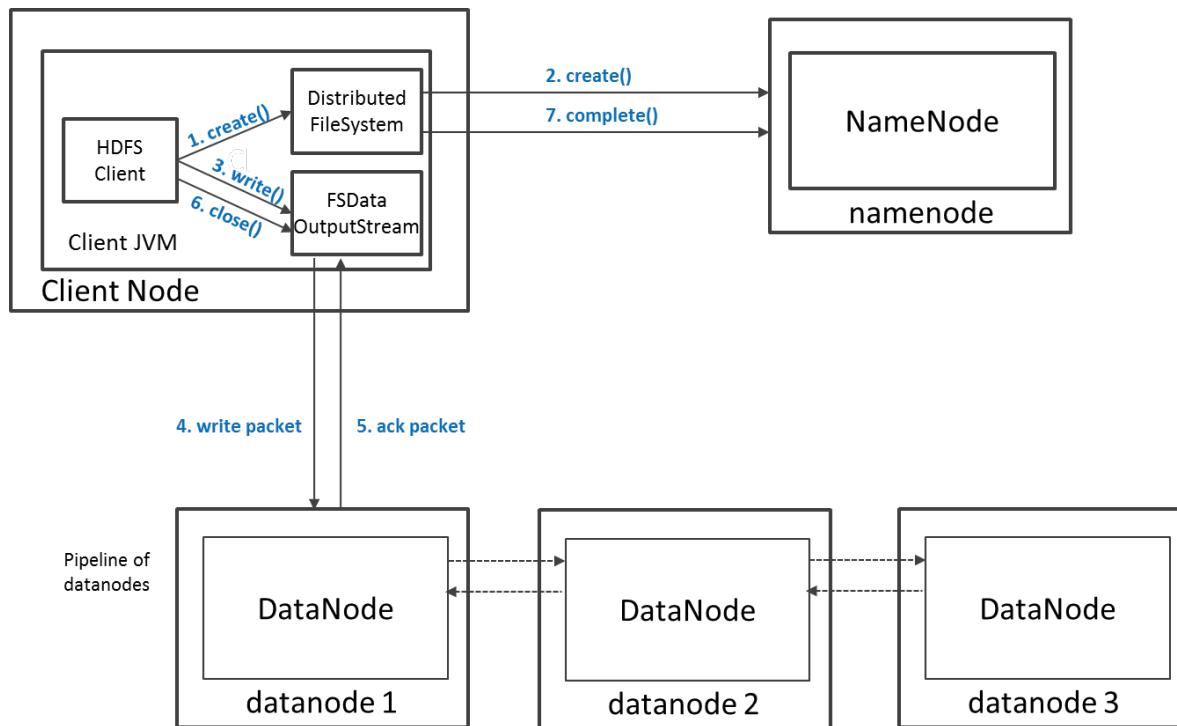
There are two (*and a half*) types of machines in a HDFS cluster

- **NameNode** is the heart of an HDFS filesystem, it maintains and manages the file system metadata. E.g; what blocks make up a file, and on which datanodes those blocks are stored
- **DataNode** where HDFS stores the actual data, there are usually quite a few of these

Read operation in HDFS



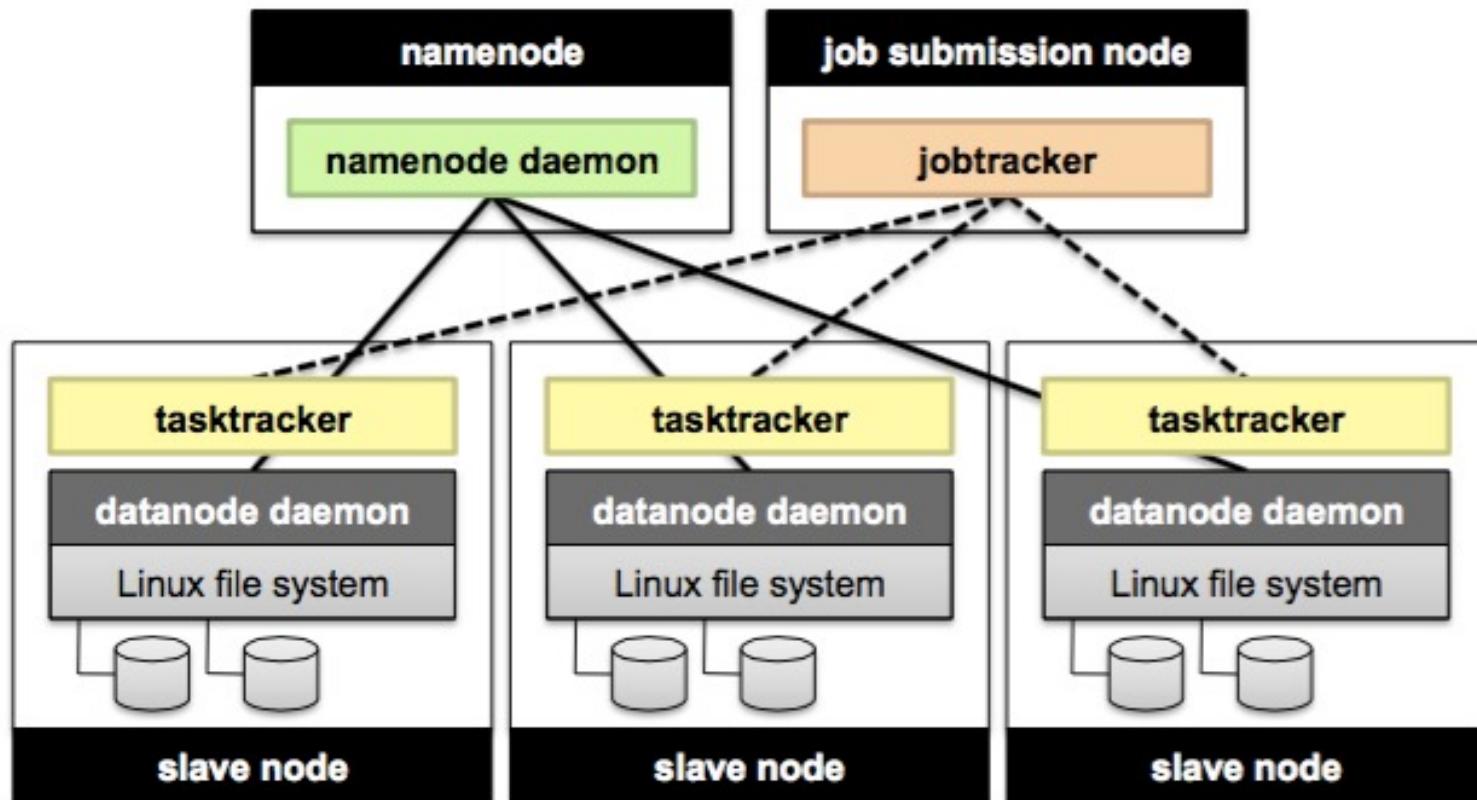
Write operation in HDFS



Unique features of HDFS

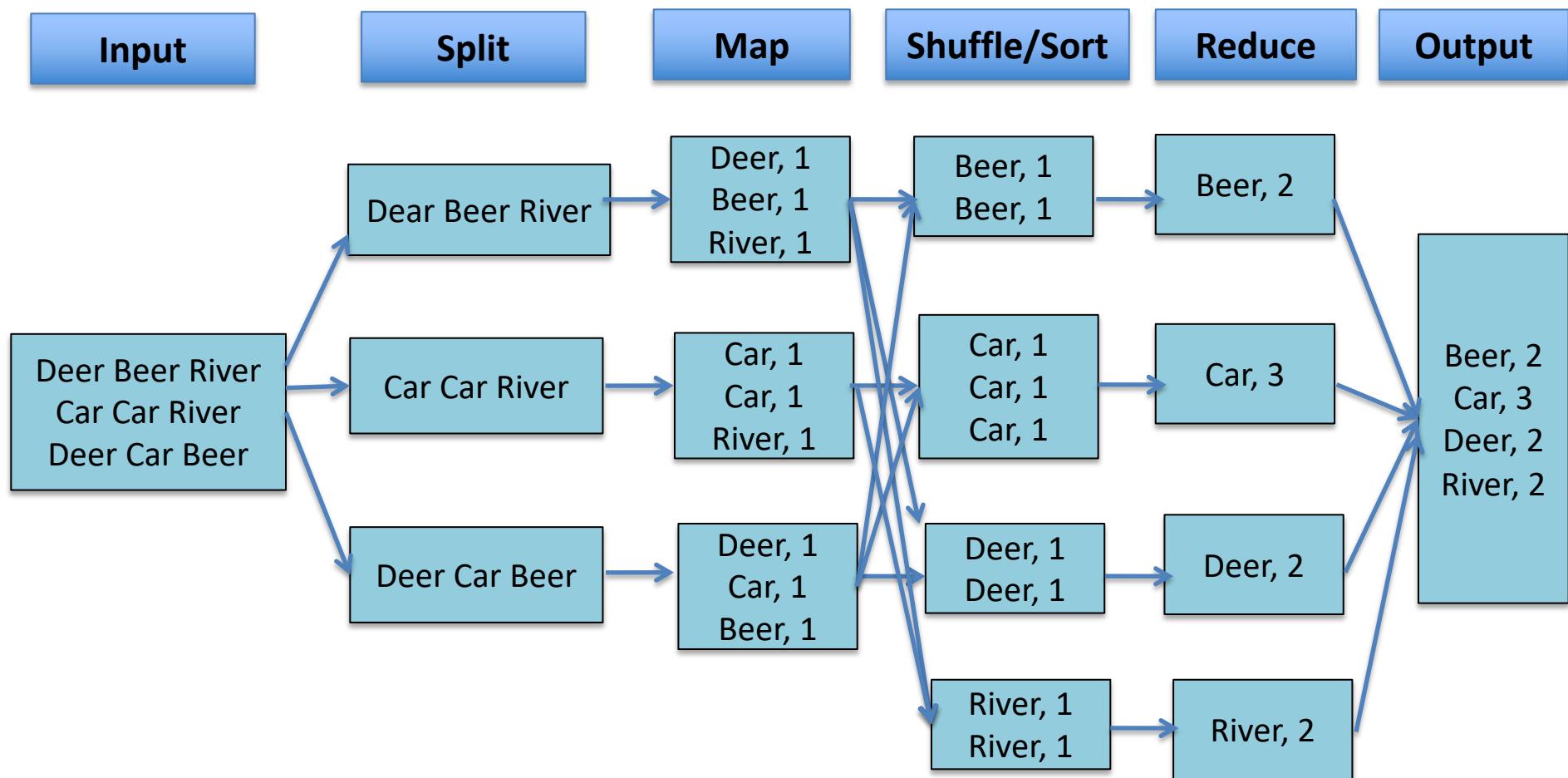
- HDFS also has a bunch of unique features that make it ideal for distributed systems:
 - **Failure tolerant** - data is duplicated across multiple DataNodes to protect against machine failures. The default is a replication factor of 3 (every block is stored on three machines).
 - **Scalability** - data transfers happen directly with the DataNodes so your read/write capacity scales fairly well with the number of DataNodes
 - **Space** - need more disk space? Just add more DataNodes and re-balance
 - **Industry standard** - Other distributed applications are built on top of HDFS (HBase, Map-Reduce)
- HDFS is designed to process large data sets with **write-once-read-many**, **it is not for low latency access**

MapReduce & HDFS

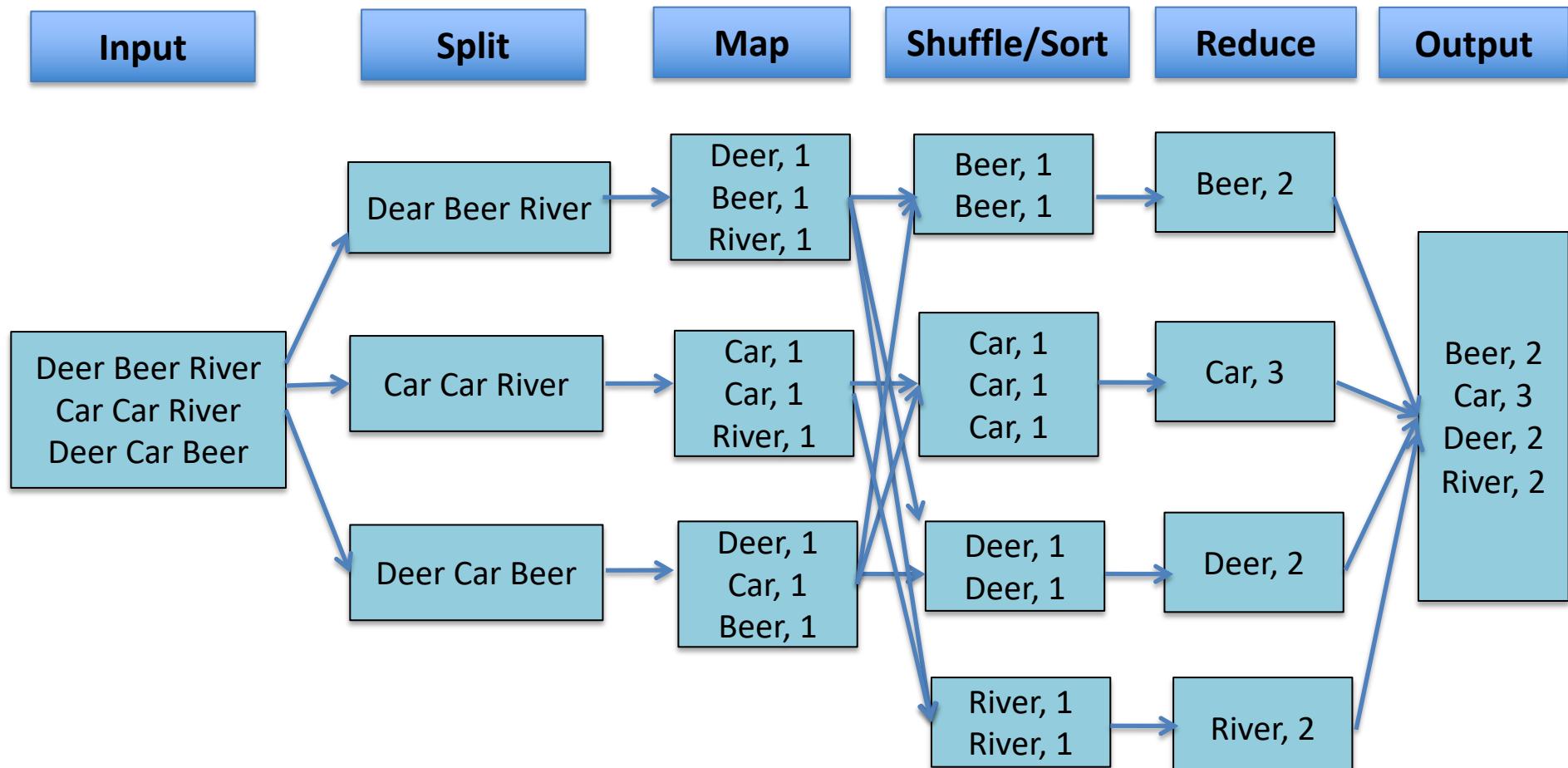


Algorithm & programming

MapReduce Example: Word Count



MapReduce Example: Word Count



Q: What are the Key and Value Pairs of Map and Reduce?

Map: Key=word, Value=1

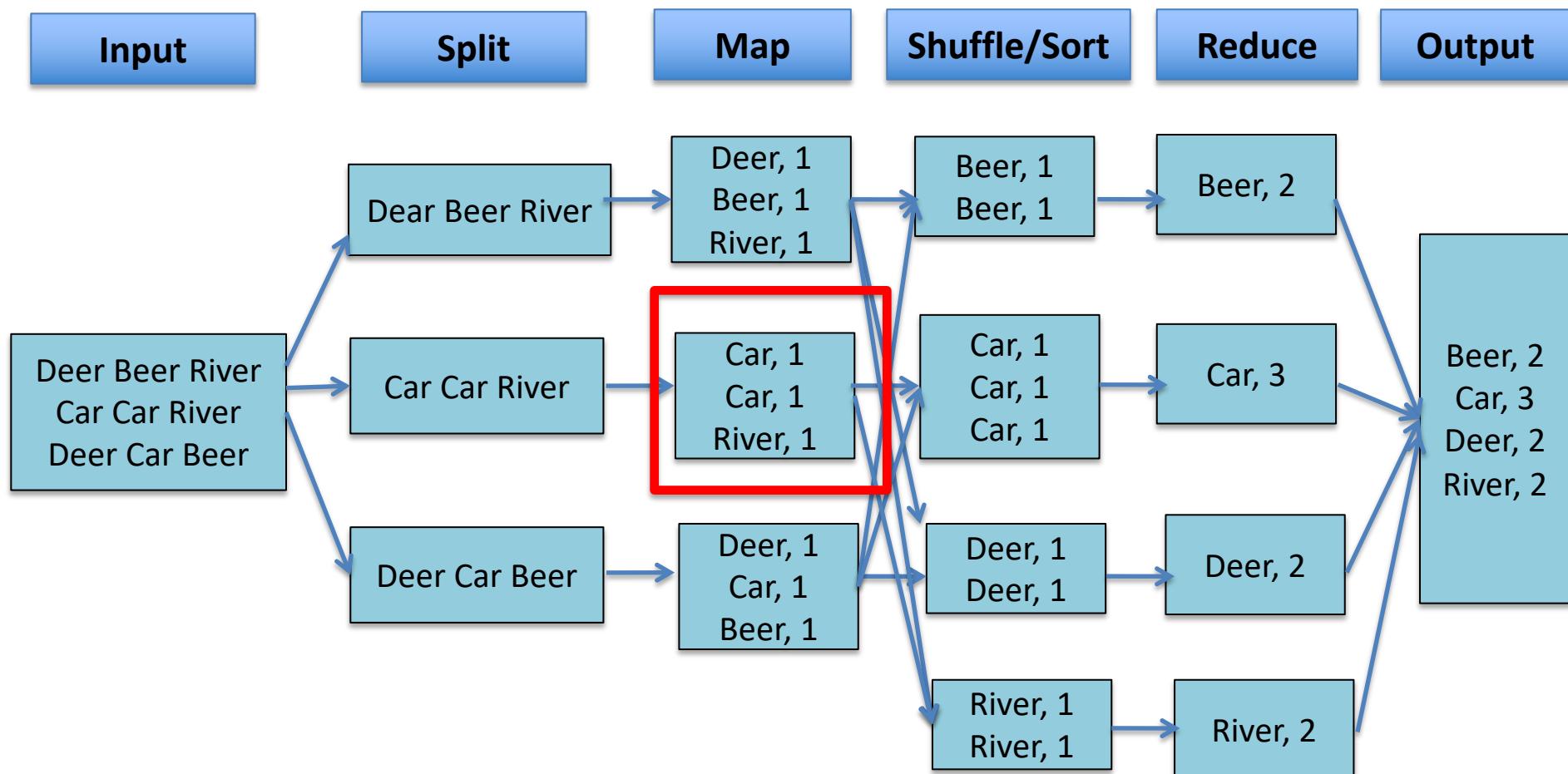
Reduce: Key=word, Value=aggregated count

Word Count: baseline

```
1: class MAPPER
2:   method MAP(docid a, doc d)
3:     for all term t  $\in$  doc d do
4:       EMIT(term t, count 1)

1: class REDUCER
2:   method REDUCE(term t, counts [c1, c2, ...])
3:     sum  $\leftarrow$  0
4:     for all count c  $\in$  counts [c1, c2, ...] do
5:       sum  $\leftarrow$  sum + c
6:     EMIT(term t, count s)
```

MapReduce Example: Word Count

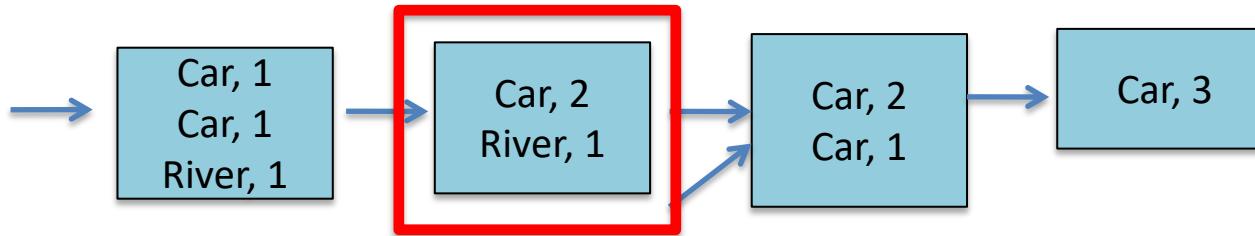


Q: Do you see any place we can improve the efficiency?

**Local aggregation at mapper will be able to improve
MapReduce efficiency.**

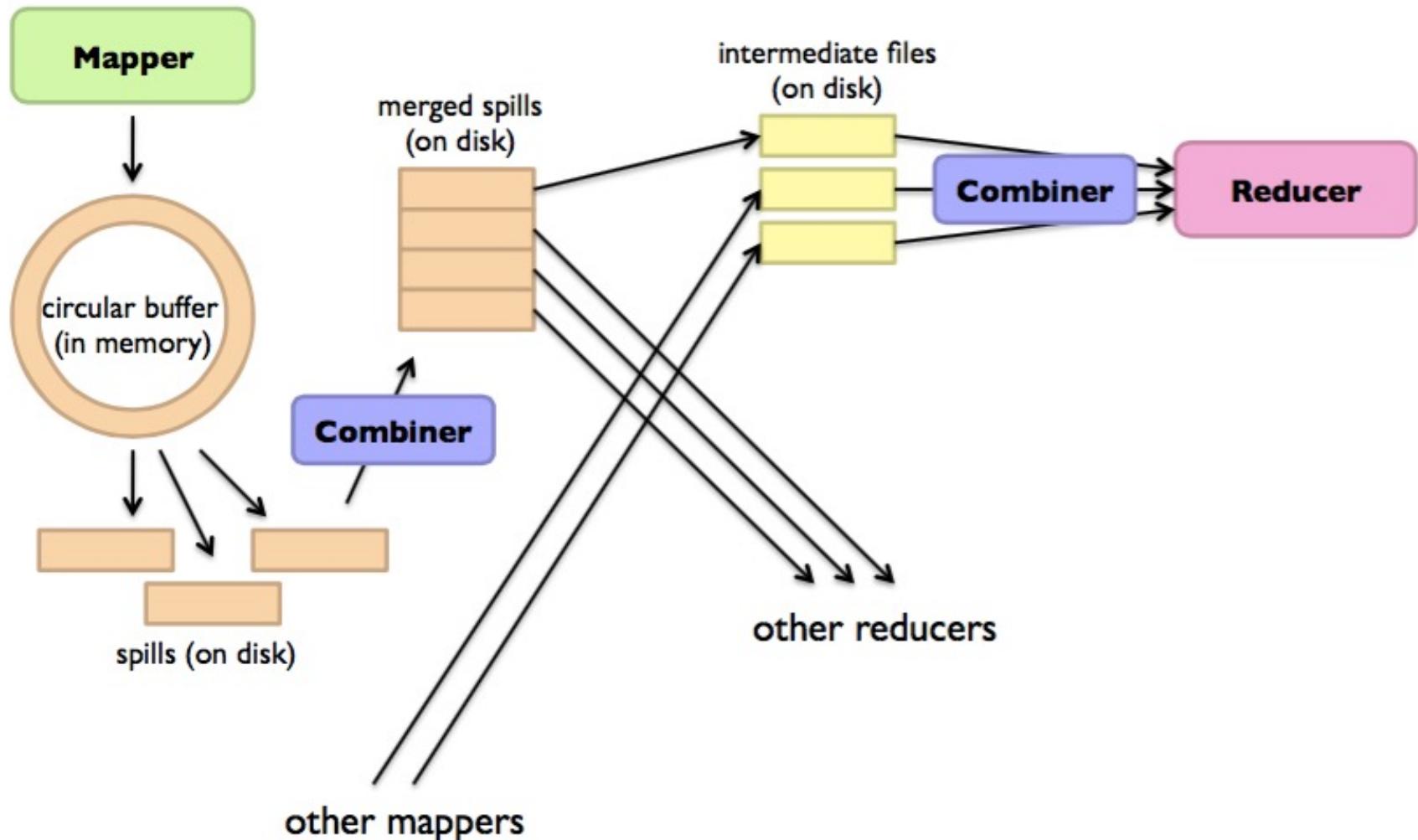
MapReduce: Combiner

- **Combiner:** do local aggregation/combine task at mapper

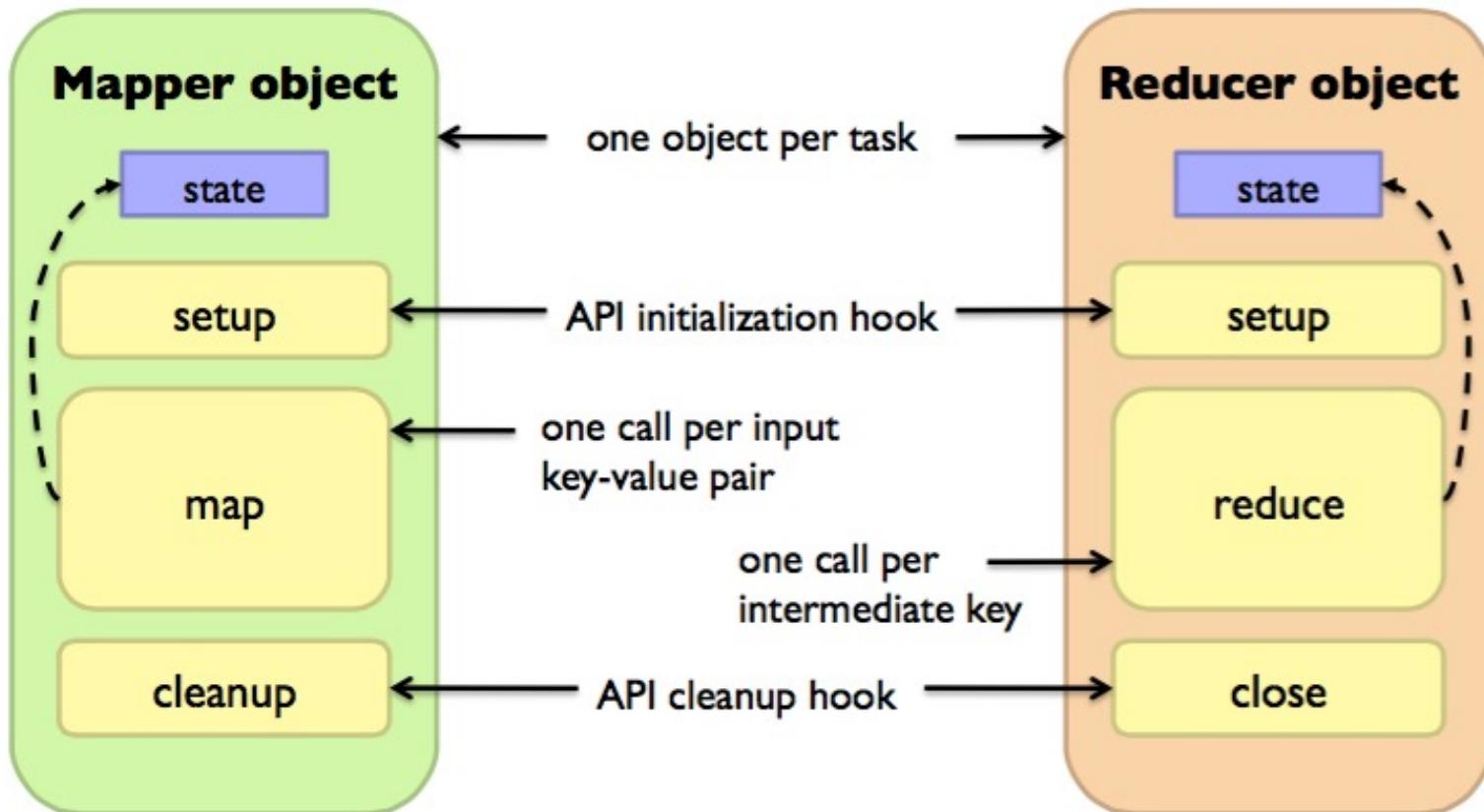


- **Q: What are the benefits of using combiner:**
 - Reduce memory/disk requirement of Map tasks
 - Reduce network traffic
- **Q: Can we remove the reduce function?**
 - No, reducer still needs to process records with same key but from *different mappers*
- **Q: How would you implement combiner?**
 - It is the same as Reducer!

Shuffle and sort



Preserving state



Implementation **don't**

- Don't unnecessarily create objects
 - Object creation is costly
 - Garbage collection is costly
- Don't buffer objects
 - Processes have limited heap size (remember, commodity machines)
 - May work for small datasets, but won't scale!

Word Count: version 1

```
1: class MAPPER
2:   method MAP(docid a, doc d)
3:     H  $\leftarrow$  new ASSOCIATIVEARRAY
4:     for all term t  $\in$  doc d do
5:       H{t}  $\leftarrow$  H{t} + 1            $\triangleright$  Tally counts for entire document
6:     for all term t  $\in$  H do
7:       EMIT(term t, count H{t})
```

Word Count: version 2

```
1: class MAPPER
2:   method INITIALIZE
3:      $H \leftarrow$  new ASSOCIATIVEARRAY
4:   method MAP(docid  $a$ , doc  $d$ )
5:     for all term  $t \in$  doc  $d$  do
6:        $H\{t\} \leftarrow H\{t\} + 1$ 
7:   method CLOSE
8:     for all term  $t \in H$  do
9:       EMIT(term  $t$ , count  $H\{t\}$ )
```

Key idea: preserve state across
input key-value pairs!

▷ Tally counts *across* documents

Are combiners still need?

Design pattern for local aggregation

- “In-mapper combining”
 - Fold the functionality of the combiner into the mapper by preserving state across multiple map calls
- Advantages
 - Speed
 - [Why is this faster than actual combiners?](#)
- Disadvantages
 - Explicit memory management required
 - Potential for order-dependent bugs

Combiner design

- Combiners and reducers share same method signature
 - Sometimes, reducers can serve as combiners
 - Often, not...
- Remember: combiner are optional optimizations
 - Should not affect algorithm correctness
 - May be run 0, 1, or multiple times
- Example: find average of integers associated with the same key

Computing the Mean: version 1

```
1: class MAPPER
2:     method MAP(string t, integer r)
3:         EMIT(string t, integer r)

1: class REDUCER
2:     method REDUCE(string t, integers [r1, r2, ...])
3:         sum ← 0
4:         cnt ← 0
5:         for all integer r ∈ integers [r1, r2, ...] do
6:             sum ← sum + r
7:             cnt ← cnt + 1
8:             ravg ← sum/cnt
9:             EMIT(string t, integer ravg)
```

Why can't we use Reducer as Combiner?

Computing the Mean: version 2

```
1: class MAPPER
2:     method MAP(string t, integer r)
3:         EMIT(string t, integer r)
4:
5: class COMBINER
6:     method COMBINE(string t, integers [r1, r2, ...])
7:         sum ← 0
8:         cnt ← 0
9:         for all integer r ∈ integers [r1, r2, ...] do
10:             sum ← sum + r
11:             cnt ← cnt + 1
12:         EMIT(string t, pair (sum, cnt))           ▷ Separate sum and count
13:
14: class REDUCER
15:     method REDUCE(string t, pairs [(s1, c1), (s2, c2) ...])
16:         sum ← 0
17:         cnt ← 0
18:         for all pair (s, c) ∈ pairs [(s1, c1), (s2, c2) ...] do
19:             sum ← sum + s
20:             cnt ← cnt + c
21:         ravg ← sum/cnt
22:         EMIT(string t, integer ravg)
```

Why doesn't this work?

Computing the Mean: version 3

```
1: class MAPPER
2:     method MAP(string t, integer r)
3:         EMIT(string t, pair (r, 1))
4:
5: class COMBINER
6:     method COMBINE(string t, pairs [(s1, c1), (s2, c2) ...])
7:         sum ← 0
8:         cnt ← 0
9:         for all pair (s, c) ∈ pairs [(s1, c1), (s2, c2) ...] do
10:             sum ← sum + s
11:             cnt ← cnt + c
12:         EMIT(string t, pair (sum, cnt))
13:
14: class REDUCER
15:     method REDUCE(string t, pairs [(s1, c1), (s2, c2) ...])
16:         sum ← 0
17:         cnt ← 0
18:         for all pair (s, c) ∈ pairs [(s1, c1), (s2, c2) ...] do
19:             sum ← sum + s
20:             cnt ← cnt + c
21:         ravg ← sum/cnt
22:         EMIT(string t, pair (ravg, cnt))
```

Computing the Mean: version 4

```
1: class MAPPER
2:   method INITIALIZE
3:     S ← new ASSOCIATIVEARRAY
4:     C ← new ASSOCIATIVEARRAY
5:   method MAP(string t, integer r)
6:     S{t} ← S{t} + r
7:     C{t} ← C{t} + 1
8:   method CLOSE
9:     for all term t ∈ S do
10:       EMIT(term t, pair (S{t}, C{t}))
```

Are combiners still need?

Word Count & sorting

- **New Goal:** output all words sorted by their frequencies (total counts) in a document.
- **Question:** How would you adopt the basic word count program to solve it?
- **Solution:**
 - Sort words by their counts in the reducer
 - Problem: what happens if we have more than one reducer?

Word Count & sorting

- **New Goal:** output all words sorted by their frequencies (total counts) in a document.
- **Question:** How would you adopt the basic word count program to solve it?
- **Solution:**
 - Do two rounds of MapReduce
 - In the 2nd round, take the output of WordCount as input but switch key and value pair!
 - Leverage the sorting capability of *shuffle/sort* to do the global sorting!

Word Count & top K words

- **New Goal:** output the top K words sorted by their frequencies (total counts) in a document.
- **Question:** How would you adopt the basic word count program to solve it?
- **Solution:**
 - Use the solution of previous problem and only grab the top K in the final output
 - Problem: is there a more efficient way to do it?

Word Count & top K words

- **New Goal:** output the top K words sorted by their frequencies (total counts) in a document.
- **Question:** How would you adopt the basic word count program to solve it?
- **Solution:**
 - Add a sort function to the *reducer* in the first round and only output the top K words
 - Intuition: the global top K must be a local top K in any reducer!

MapReduce In-class Exercise

- **Problem:** Find the maximum monthly temperature for each year from weather reports
- **Input:** A set of records with format as:
 $\langle \text{Year}/\text{Month}, \text{Average Temperature of that month} \rangle$
 - **(200707,100), (200706,90)**
 - **(200508, 90), (200607,100)**
 - **(200708, 80), (200606,80)**
- **Question:** write down the Map and Reduce function to solve this problem
 - Assume we split the input by line

Mapper and Reducer of Max Temperature

- Map(key, value){
 // key: line number
 // value: tuples in a line
 for each tuple t in value:
 Emit(t->year, t->temperature);}
- Reduce(key, list of values){
 // key: year
 //list of values: a list of monthly temperature
 int max_temp = -100;
 for each v in values:
 max_temp= max(v, max_temp);
 Emit(key, max_temp);}

Combiner is the same as Reducer

MapReduce Example: Max Temperature

Input

(200707,100), (200706,90)
(200508, 90), (200607,100)
(200708, 80), (200606,80)

Map

(2007,100), (2007,90)

(2005, 90), (2006,100)

(2007, 80), (2006, 80)

Combine

(2007,100)

(2005, 90), (2006,100)

(2007, 80), (2006, 80)

Shuttle/Sort

(2005,[90])

(2006,[100, 80])

(2007,[100, 80])

Reduce

(2005,90)

(2006,100)
HPC Lab-CSE-HCMUT

(2007,100)

MapReduce In-class Exercise

- **Key-Value Pair of Map and Reduce:**
 - **Map:** (year, temperature)
 - **Reduce:** (year, maximum temperature of the year)
- **Question: How to use the above Map Reduce program (*that contains the combiner*) with slight changes to find the average monthly temperature of the year?**

Mapper and Reducer of Average Temperature

- Map(key, value){
 // key: line number
 // value: tuples in a line
 for each tuple t in value:
 Emit(t->year, t->temperature);}
 - Reduce(key, list of values){
 // key: year
 // list of values: a list of monthly temperatures
 int total_temp = 0;
 for each v in values:
 total_temp= total_temp+v;
 Emit(key, total_temp/size_of(values));}
- Combiner is the same as Reducer

MapReduce Example: Average Temperature

Input

(200707,100), (200706,90)
(200508, 90), (200607,100)
(200708, 80), (200606,80)

Real average of
2007: 90

Map

(2007,100), (2007,90)

(2005, 90), (2006,100)

(2007, 80), (2006,80)

Combine

(2007,95)

(2005, 90), (2006,100)

(2007, 80), (2006,80)

Shuttle/Sort

(2005,[90])

(2006,[100, 80])

(2007,[95, 80])

Reduce

(2005,90)

(2006,90)
HPC Lab-CSE-HCMUT

(2007,87.5)

MapReduce In-class Exercise

- The problem is with the combiner!
- Here is a simple counterexample:
 - $(2007, 100), (2007, 90) \rightarrow (2007, 95)$
 $(2007, 80) \rightarrow (2007, 80)$
 - Average of the above is: $(2007, 87.5)$
 - However, the real average is: $(2007, 90)$
- However, we can do a small trick to get around this
 - Mapper: $(2007, 100), (2007, 90) \rightarrow (2007, <190, 2>)$
 $(2007, 80) \rightarrow (2007, <80, 1>)$
 - Reducer: $(2007, <270, 3>) \rightarrow (2007, 90)$

MapReduce Example: Average Temperature

Input

(200707,100), (200706,90)
(200508, 90), (200607,100)
(200708, 80), (200606,80)

Map

(2007,100), (2007,90)

(2005, 90), (2006,100)

(2007, 80), (2006,80)

Combine

(2007,<190,2>)

(2005, <90,1>),
(2006, <100,1>)

(2007, <80,1>),
(2006,<80,1>)

Shuttle/Sort

(2005,[<90,1>])

(2006,[<100,1>, <80,1>])

(2007,[<190,2>, <80,1>])

Reduce

(2005,90)

(2006,90)
HPC Lab-CSE-HCMUT

(2007,90)

Mapper and Reducer of Average Temperature

- **Map(key, value){**
 // key: line number
 // value: tuples in a line
 for each tuple t in value:
 Emit(t->year, t->temperature);}
- **Reduce (key, list of values){**
 // key: year
 // list of values: a list of <temperature sums, counts> tuples

 int total_temp = 0;
 int total_count=0;
 for each v in values:
 total_temp= total_temp+v->sum;
 total_count=total_count+v->count;

 Emit(key,**total_temp/total_count**);}
- **Combine(key, list of values){**
 // key: year
 // list of values: a list of monthly temperature

 int total_temp = 0;
 for each v in values:
 total_temp= total_temp+v;

 Emit(key,<total_temp,size_of(values)>);}

MapReduce In-class Exercise

- Functions that can use combiner are called *distributive*:
 - Distributive: Min/Max(), Sum(), Count(), TopK()
 - Non-distributive: Mean(), Median(), Rank()

Gray, Jim*, et al. "Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals." Data Mining and Knowledge Discovery 1.1 (1997): 29-53.

*Jim Gray received Turing Award in 1998

Map Reduce Problems Discussion

- **Problem 1:** Find Word Length Distribution
- **Statement:** Given a set of documents, use MapReduce to find the length distribution of all words contained in the documents
- **Question:**
 - What are the Mapper and Reducer Functions?

This is a test data for
the word length
distribution problem



12: 1
7: 1
6: 1
4: 4
3: 2
2: 1
1: 1

Mapper and Reducer of Word Length Distribution

- Map(key, value){

// key: document name

// value: words in a document

for each word w in value:

Emit(length(w), w);}

- Reduce(key, list of values){

// key: length of a word

// list of values: a list of words with the same length

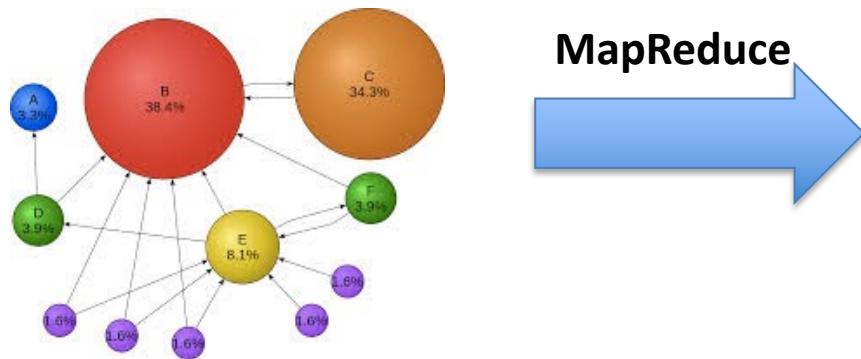
Emit(key, size_of(values));}

Map Reduce Problems Discussion

- **Problem 1:** Find Word Length Distribution
- **Mapper and Reducer:**
 - Mapper(document)
 { Emit (Length(word), word) }
 - Reducer(output of map)
 { Emit (Length(word), Size of (List of words at a particular length))}

Map Reduce Problems Discussion

- **Problem 2:** Indexing & Page Rank
- **Statement:** Given a set of web pages, each page has a **page rank** associated with it, use Map-Reduce to find, for each word, a list of pages (sorted by rank) that contains that word
- **Question:**
 - What are the Mapper and Reducer Functions?



Word 1: [page x1,
page x2, ...]

Word 2: [page y1,
page y2, ...]

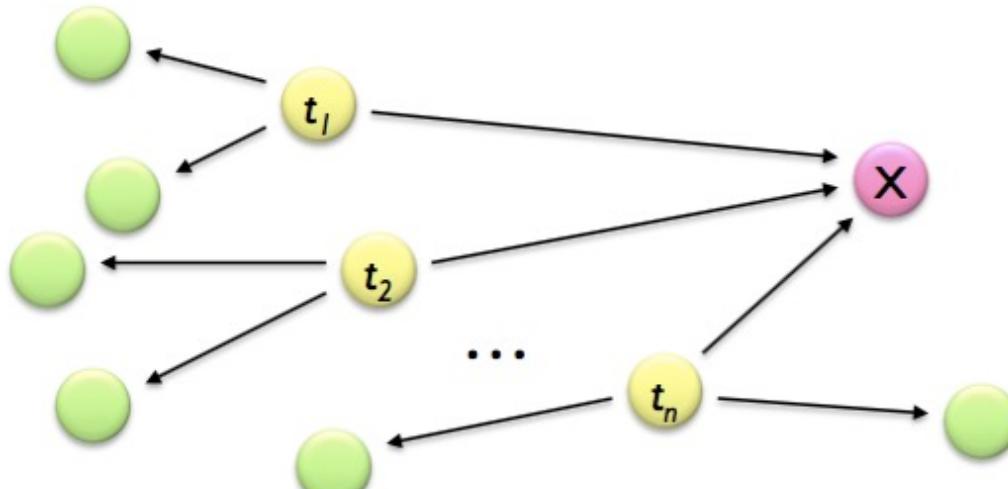
...

Page Rank

Given page x with inlinks t_1, \dots, t_n , where

- $C(t)$ is the out-degree of t
- α is probability of random jump
- N is the total number of nodes in the graph

$$PR(x) = \alpha \left(\frac{1}{N} \right) + (1 - \alpha) \sum_{i=1}^n \frac{PR(t_i)}{C(t_i)}$$



Mapper and Reducer of Indexing and PageRank

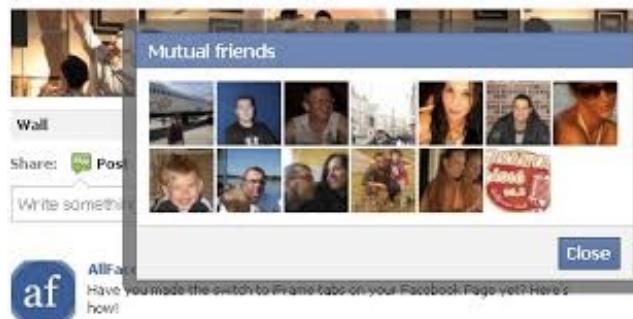
- Map(key, value){
 // key: a page
 // value: words in a page
 for each word w in value:
 Emit(w, <page_id, page_rank>);}
- Reduce(key, list of values){
 // key: a word
 // list of values: a list of pages containing that word
 sorted_pages=sort(values, page_rank)
 Emit(key, sorted_pages);}

Map Reduce Problems Discussion

- **Problem 2:** Indexing and Page Rank
- **Mapper and Reducer:**
 - Mapper(page_id, <page_text, page_rank>)
 { Emit (word, <page_id, page_rank>) }
 - Reducer(output of map)
 { Emit (word, List of pages contains the word sorted by their page_ranks)}

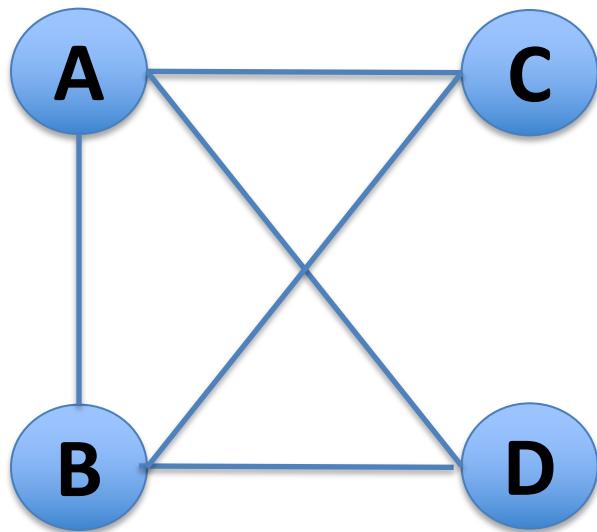
Map Reduce Problems Discussion

- **Problem 3: Find Common Friends**
- **Statement:** Given a group of people on online social media (e.g., Facebook), each has a list of friends, use Map-Reduce to find common friends of any two persons who are friends
- **Question:**
 - What are the Mapper and Reducer Functions?



Map Reduce Problems Discussion

- Problem 3: Find Common Friends
- Simple example:



Input:

A -> B,C,D

B-> A,C,D

C-> A,B

D->A,B

MapReduce

Output:

(A ,B) -> C,D

(A,C) -> B

(A,D) -> ..

Mapper and Reducer of Common Friends

- Map(key, value){
 // key: person_id
 // value: the list of friends of the person
 for each friend f_id in value:
 Emit(<person_id, f_id>, value);}
- Reduce(key, list of values){
 // key: <friend pair>
 // list of values: a set of friend lists related with the friend pair
 for v1, v2 in values:
 common_friends = v1 intersects v2;
 Emit(key, common_friends);}

Map Reduce Problems Discussion

- **Problem 3:** Find Common Friends
- **Mapper and Reducer:**
 - Mapper(friend list of a person)
 - { for each person in the friend list:
 Emit (<friend pair>, <list of friends>) }
 - Reducer(output of map)
 - { Emit (<friend pair>, **Intersection of two (i.e, the one in friend pair) friend lists**) }

Map Reduce Problems Discussion

- Problem 3: Find Common Friends
- Mapper and Reducer:

Input:

A -> B,C,D
B-> A,C,D
C-> A,B
D->A,B

Map:

(A,B) -> B,C,D
(A,C) -> B,C,D
(A,D) -> B,C,D
(A,B) -> A,C,D
(B,C) -> A,C,D
(B,D) -> A,C,D
(A,C) -> A,B
(B,C) -> A,B
(A,D) -> A,B
(B,D) -> A,B

Reduce:

(A,B) -> C,D
(A,C) -> B
(A,D) -> B
(B,C) -> A
(B,D) -> A

*Suggest
Friends ☺*

Enjoy MR and Hadoop ☺

