Parallel Programming with Hadoop/MapReduce

CS 240A, Tao Yang, Winter 2013

Overview

- **What is MapReduce?**
- **Related technologies**
  - Hadoop/Google file system
- **MapReduce applications**

Motivations

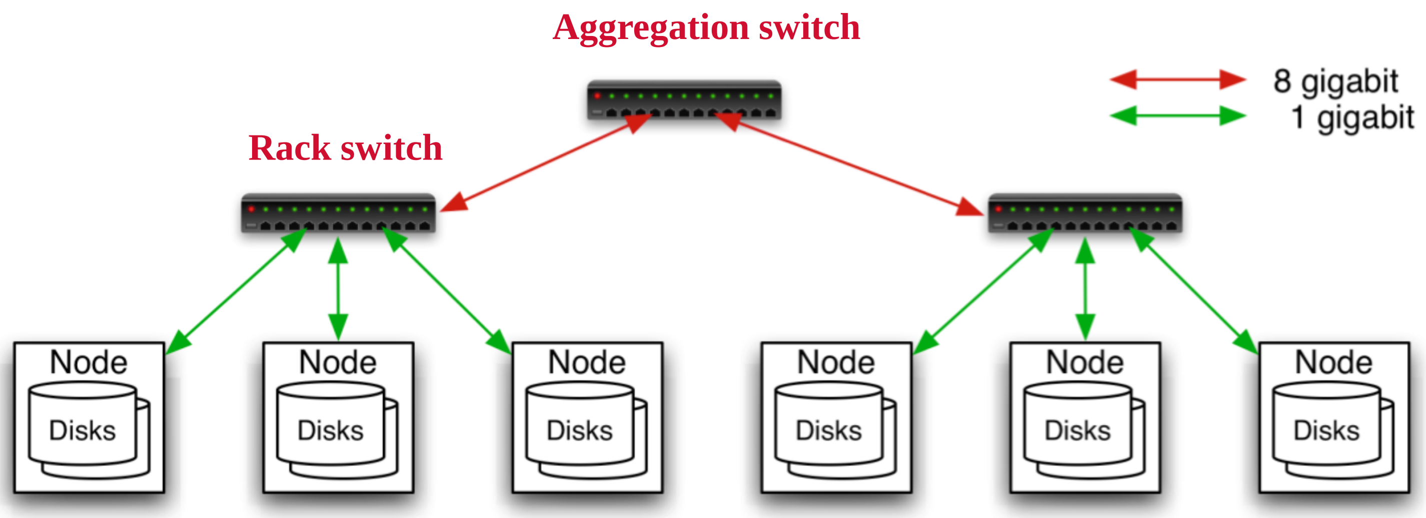
- **Motivations**
  - Large-scale data processing on clusters
  - Massively parallel (hundreds or thousands of CPUs)
  - Reliable execution with easy data access
- **Functions**
  - Automatic parallelization & distribution
  - Fault-tolerance
  - Status and monitoring tools
  - A clean abstraction for programmers
    - Functional programming meets distributed computing
    - A batch data processing system

Parallel Data Processing in a Cluster

- **Scalability to large data volumes:**
  - Scan 1000 TB on 1 node @ 100 MB/s = 24 days
  - Scan on 1000-node cluster = 35 minutes

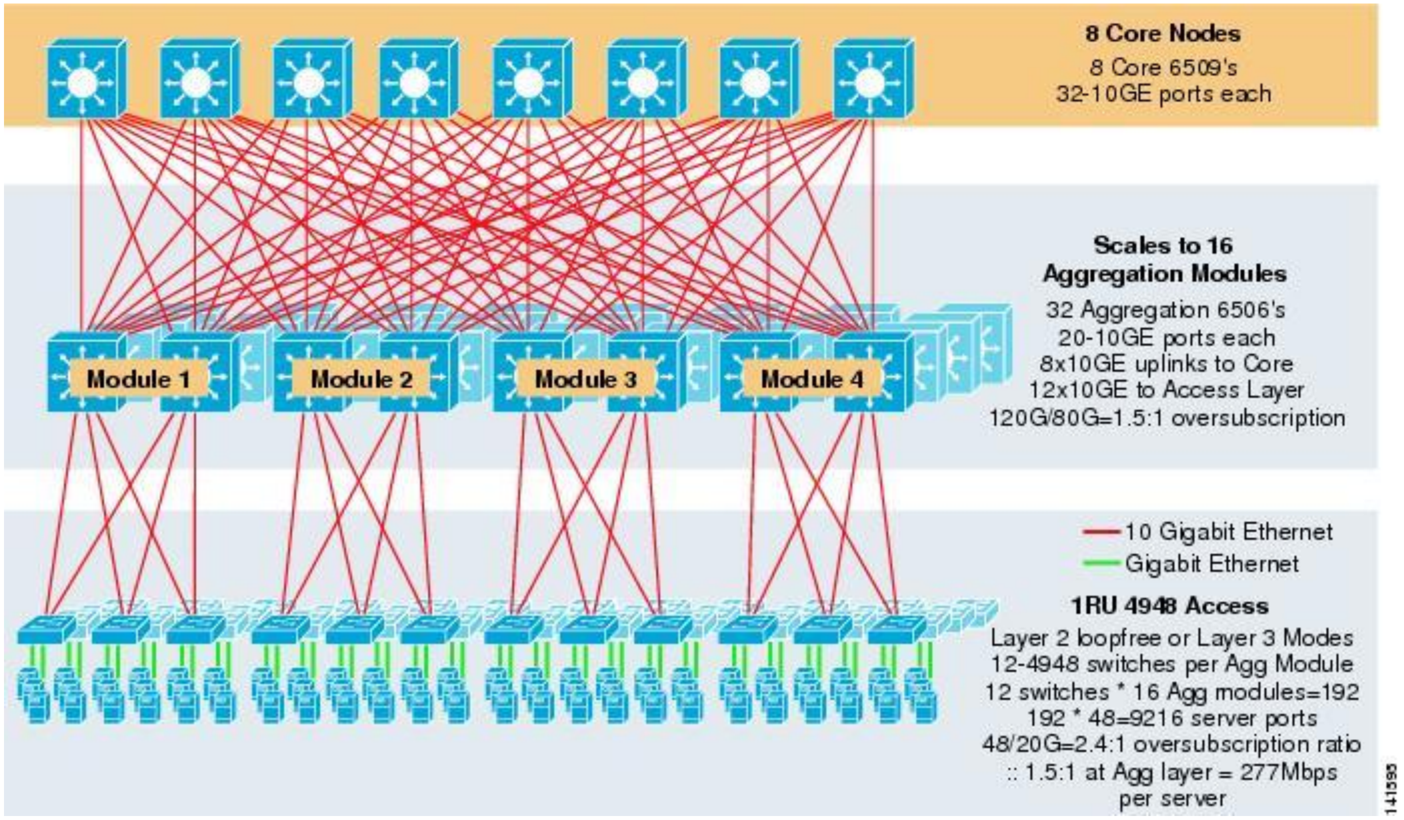
- **Cost-efficiency:**
  - Commodity nodes /network
    - » Cheap, but not high bandwidth, sometime unreliable
  - Automatic fault-tolerance (fewer admins)
  - Easy to use (fewer programmers)

Typical Hadoop Cluster



- 40 nodes/rack, 1000-4000 nodes in cluster
- 1 Gbps bandwidth in rack, 8 Gbps out of rack
- Node specs :
  8-16 cores, 32 GB RAM, 8 × 1.5 TB disks

Layered Network Architecture in Conventional Data Centers



- **A layered example from Cisco: core, aggregation, the edge or top-of-rack switch.**
- **http://www.cisco.com/en/US/docs/solutions/Enterprise/Data_Center/DC_Infra2_5/DCInfra_3a.html**

MapReduce Programming Model

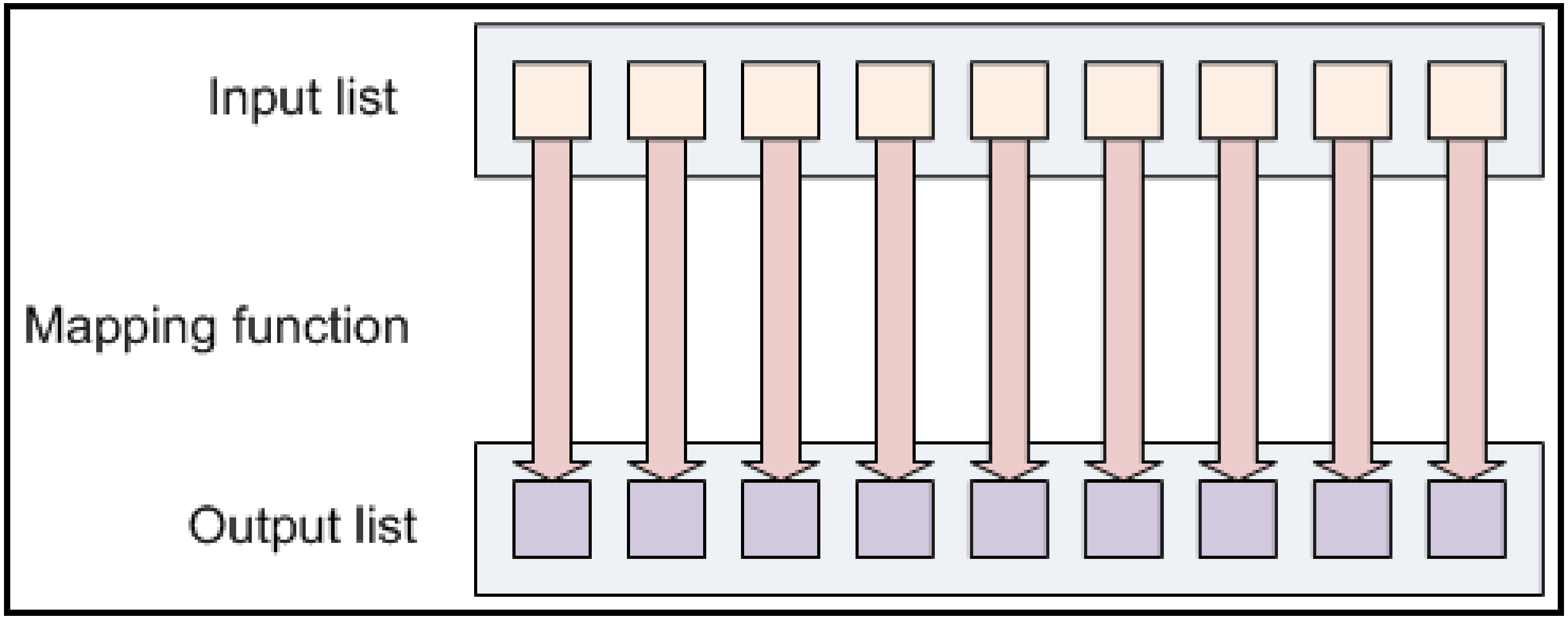
- **Inspired from map and reduce operations commonly used in functional programming languages like Lisp.**
- **Have multiple map tasks and reduce tasks**
- **Users implement interface of two primary methods:**
  - Map: (key1, val1) → (key2, val2)
  - Reduce: (key2, [val2]) → [val3]

Example: Map Processing in Hadoop

- **Given a file**
  - A file may be divided into multiple parts (splits).
- **Each record (line) is processed by a Map function,**
  - written by the user,
  - takes an input key/value pair
  - produces a set of intermediate key/value pairs.
  - e.g. (doc—id, doc-content)
- **Draw an analogy to SQL *group-by* clause**

map

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



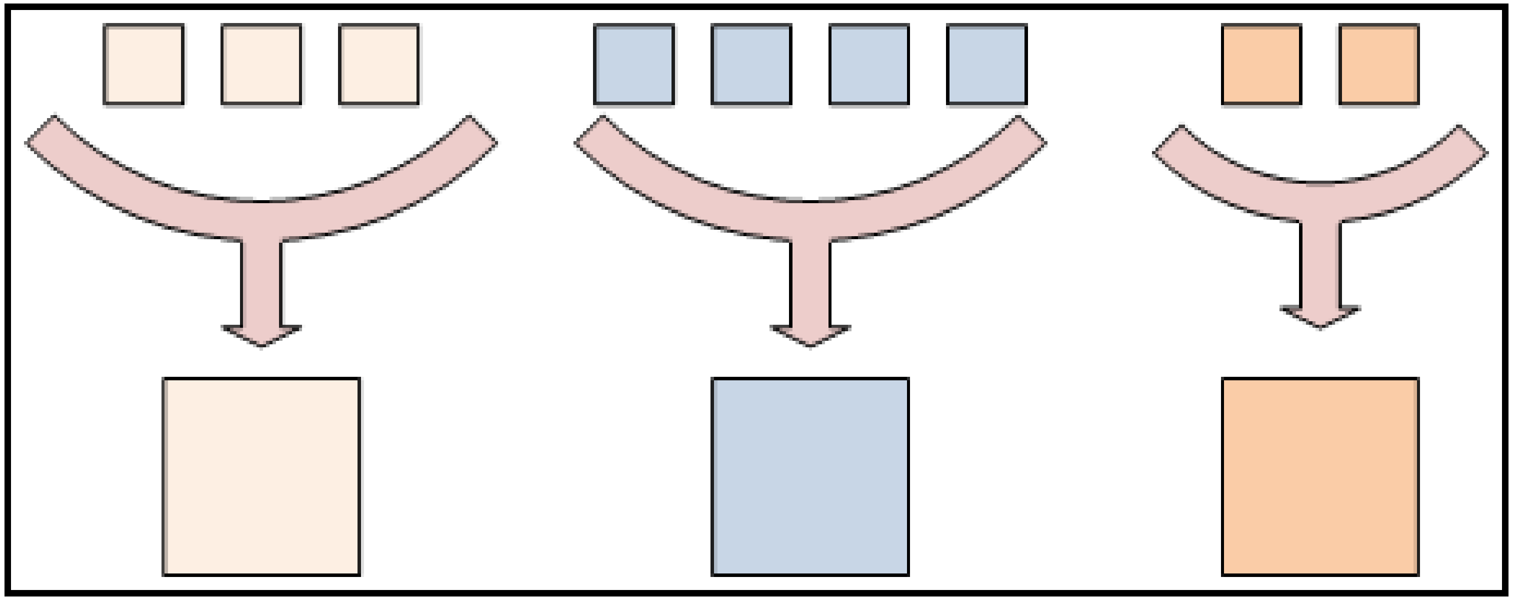
Processing of Reducer Tasks

- **Given a set of (key, value) records produced by map tasks.**
  - all the intermediate values for a given output key are combined together into a list and given to a reducer.
  - Each reducer further performs (key2, [val2]) $\rightarrow$ [val3]

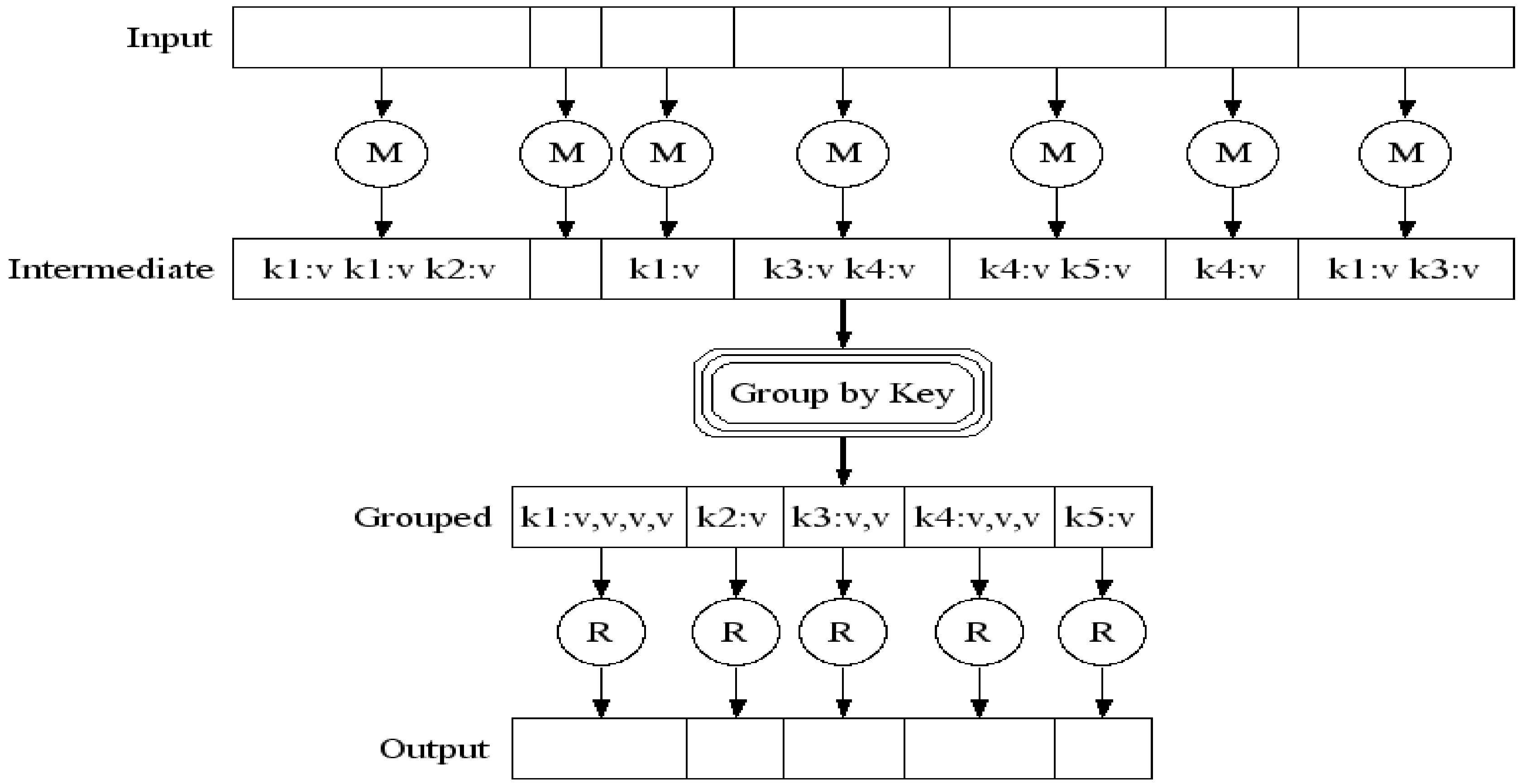
- **Can be visualized as *aggregate* function (e.g., average) that is computed over all the rows with the same group-by attribute.**

Reduce

```
reduce (out_key, intermediate_value list) ->
                 out_value list
```

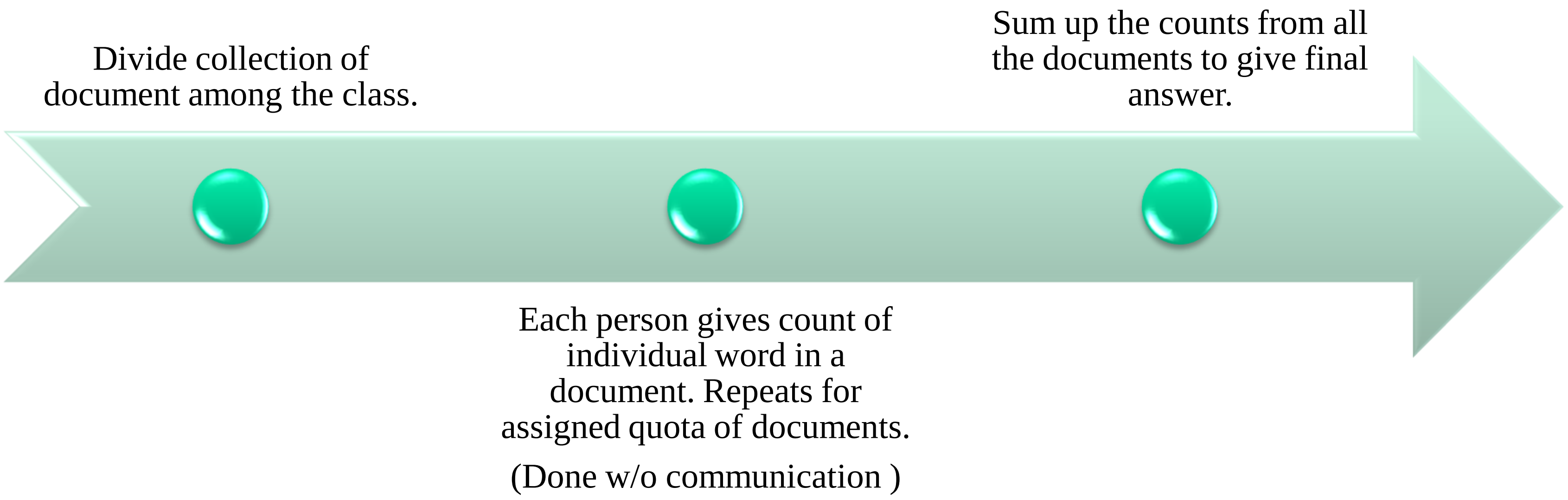


Put Map and Reduce Tasks Together



Example: Word counting

- ”Consider the problem of counting the number of occurrences of each word in a large collection of documents”



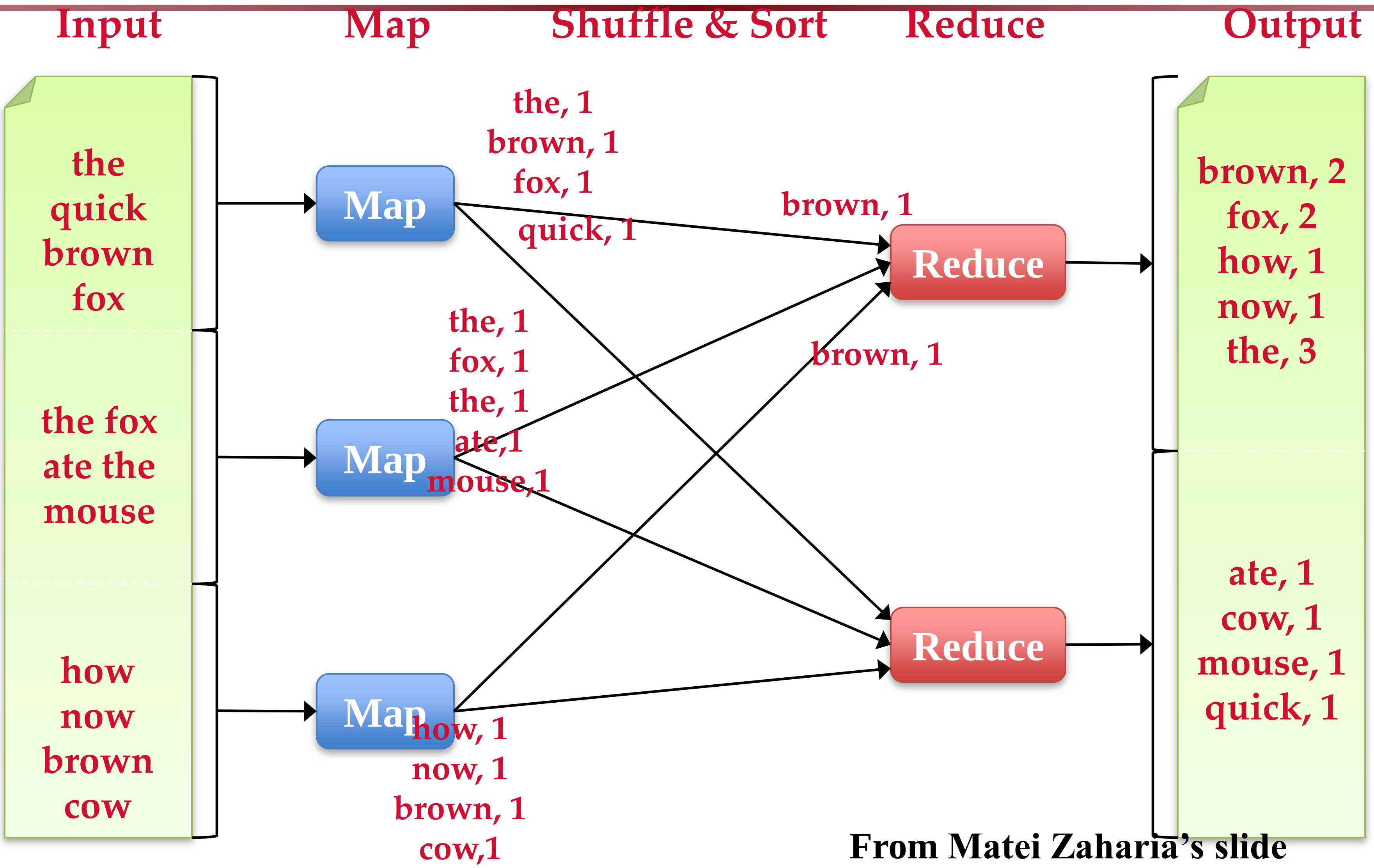
Divide collection of document among the class.

Each person gives count of individual word in a document. Repeats for assigned quota of documents.

(Done w/o communication)

Sum up the counts from all the documents to give final answer.

Word Count Execution



From Matei Zaharia's slide

Pseudo-code

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

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

MapReduce WordCount.java

Hadoop distribution: **src/examples/org/apache/hadoop/examples/WordCount.java**

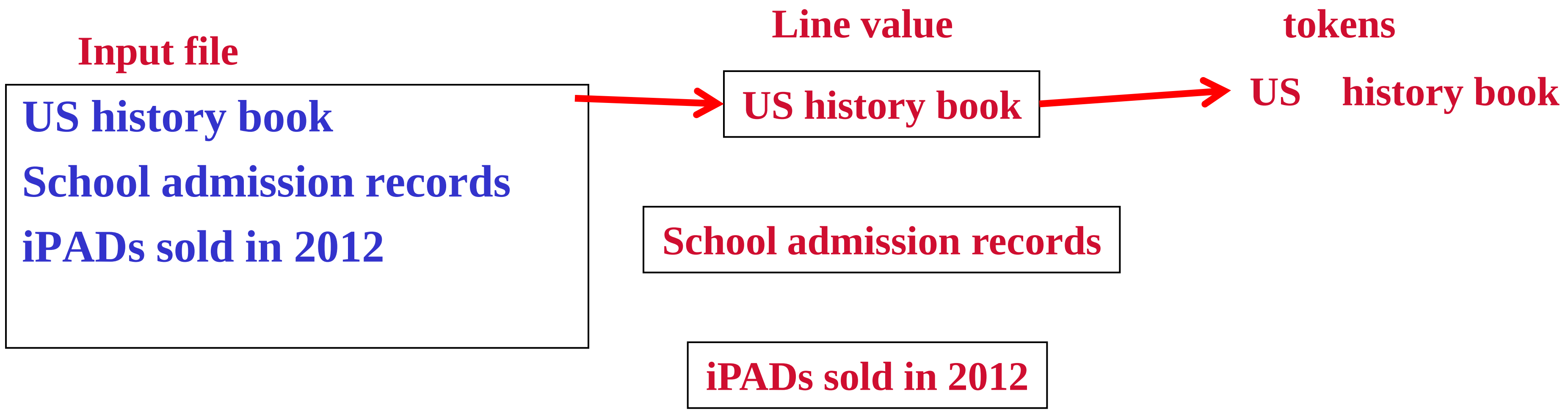
```
public static class TokenizerMapper
     extends Mapper<Object, Text, Text, IntWritable>{

  private final static IntWritable one = new IntWritable(1);
  private Text word = new Text();

  public void map(Object key, Text value, Context context
                 ) throws IOException, InterruptedException {
    StringTokenizer itr = new StringTokenizer(value.toString());
    while (itr.hasMoreTokens()) {
         word.set(itr.nextToken());
        context.write(word, one);
    }
  }
}
```

MapReduce WordCount.java

map() gets a key, value, and context

- key - "bytes from the beginning of the line?“
- value - the current line;

in the while loop, each token is a "word" from the current line



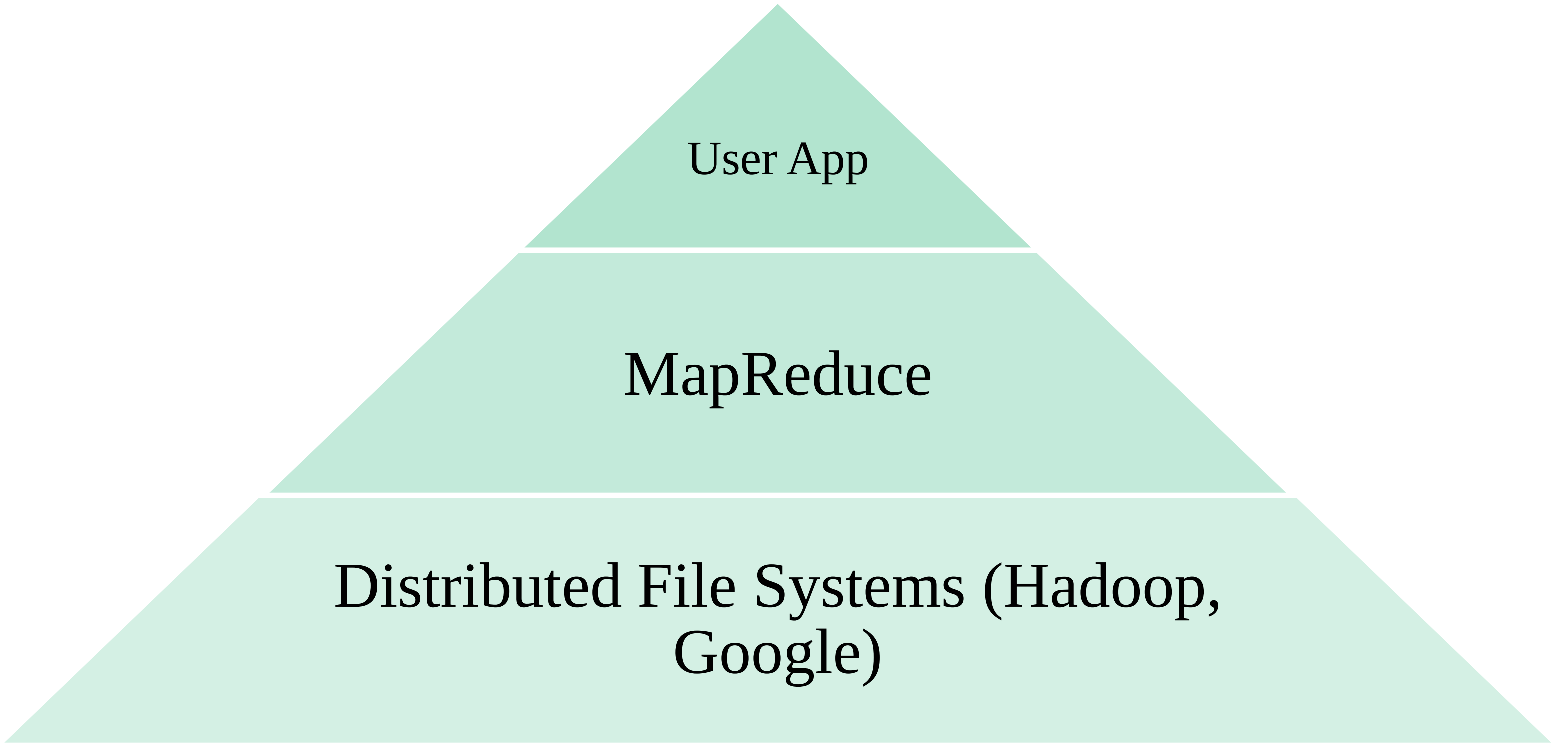
Reduce code in WordCount.java

```
public static class IntSumReducer
       extends Reducer<Text,IntWritable,Text,IntWritable> {
    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> values,
                       Context context
                       ) throws IOException, InterruptedException {
      int sum = 0;
      for (IntWritable val : values) {
        sum += val.get();
      }
      result.set(sum);
      context.write(key, result);
    }
  }
```

The driver to set things up and start

```
//   Usage: wordcount <in> <out>
 public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
    Job job = new Job(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
    FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
```

Systems Support for MapReduce



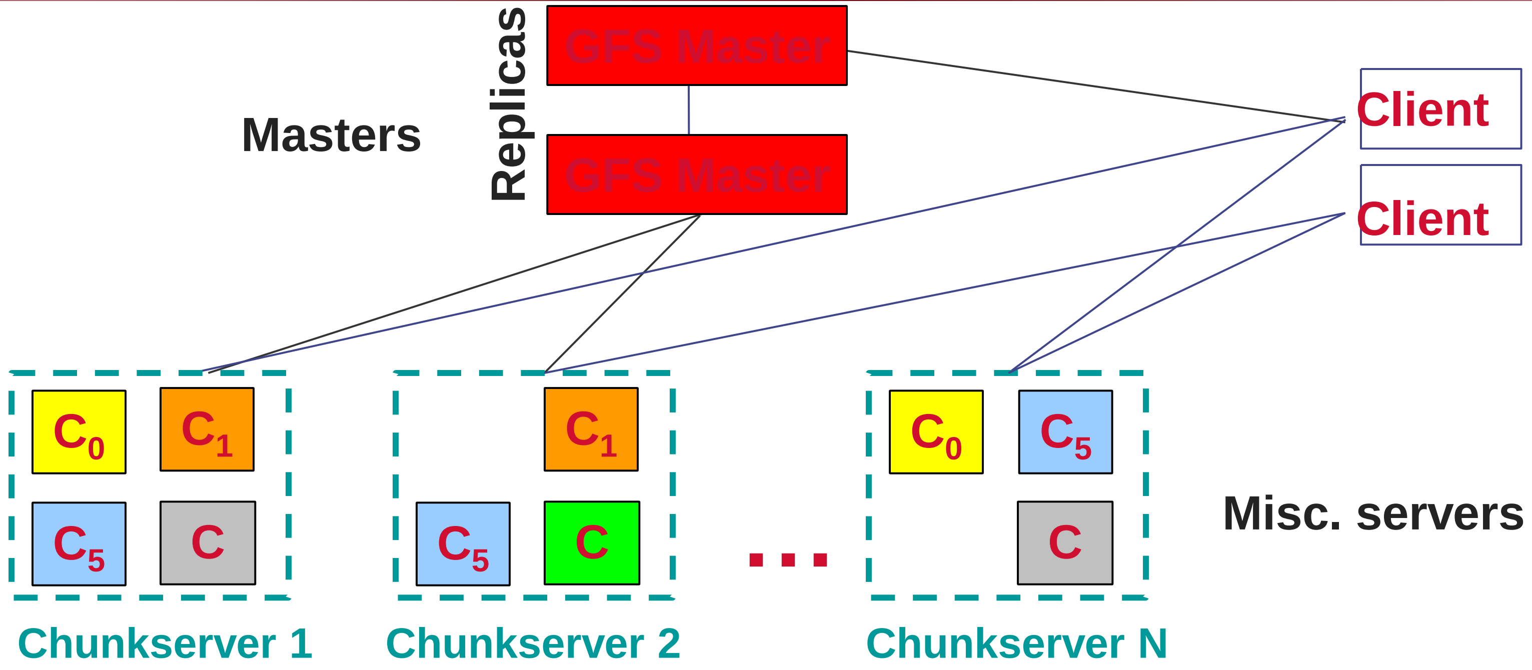
Distributed Filesystems

- **The interface is the same as a single-machine file system**
  - create(), open(), read(), write(), close()
- **Distribute file data to a number of machines (storage units).**
  - Support replication
- **Support concurrent data access**
  - Fetch content from remote servers. Local caching
- **Different implementations sit in different places on complexity/feature scale**
  - Google file system and Hadoop HDFS
    - » Highly scalable for large data-intensive applications.
    - » Provides redundant storage of massive amounts of data on cheap and unreliable computers

Assumptions of GFS/Hadoop DFS

- **High component failure rates**
  - Inexpensive commodity components fail all the time
- **“Modest” number of HUGE files**
  - Just a few million
  - Each is 100MB or larger; multi-GB files typical
- **Files are write-once, mostly appended to**
  - Perhaps concurrently
- **Large streaming reads**
- **High sustained throughput favored over low latency**

GFS Design

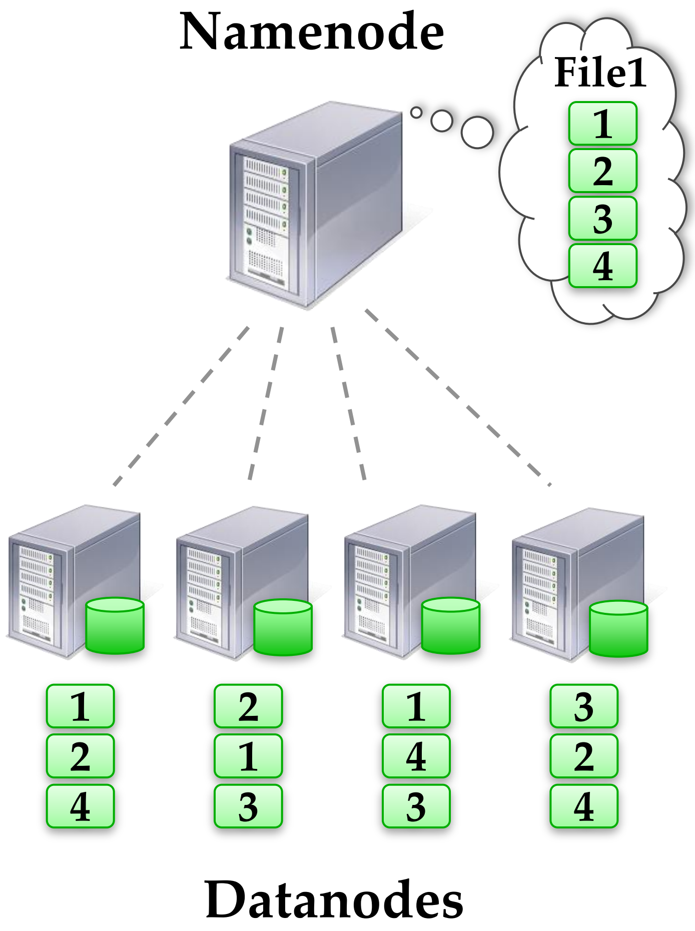


- Files are broken into chunks (typically 64 MB) and serve in chunk servers
- Master manages metadata, but clients may cache meta data obtained.
  - Data transfers happen directly between clients/chunk-servers
- Reliability through replication

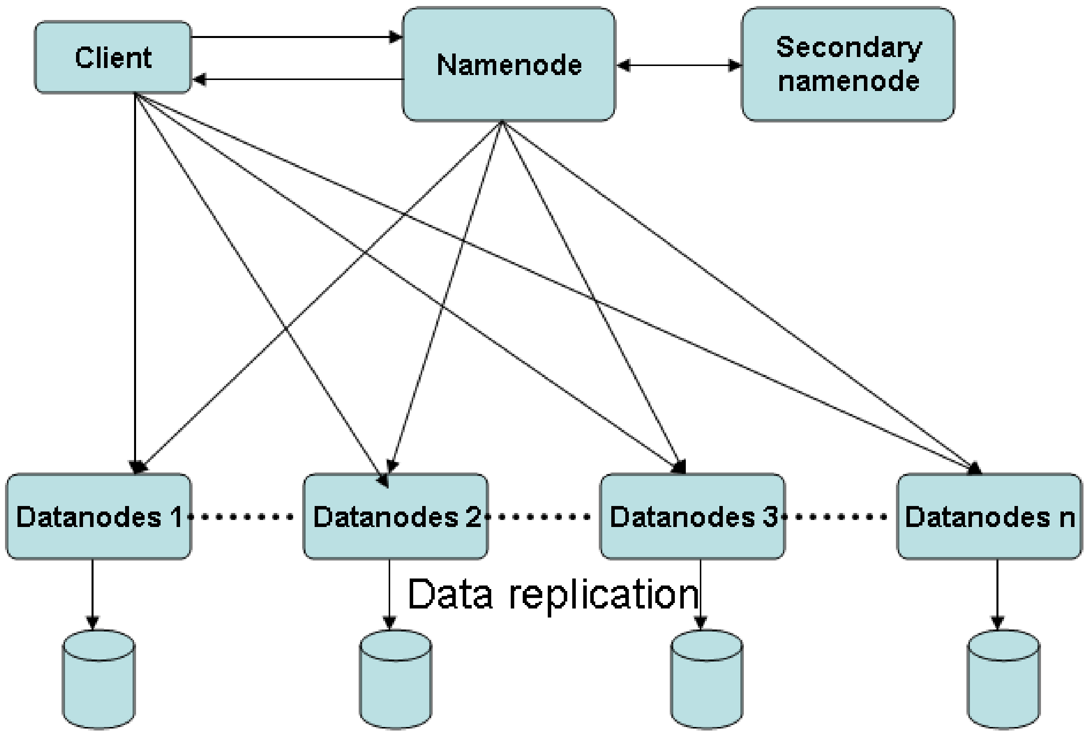
  Each chunk replicated across 3+ *chunk-servers*

Hadoop Distributed File System

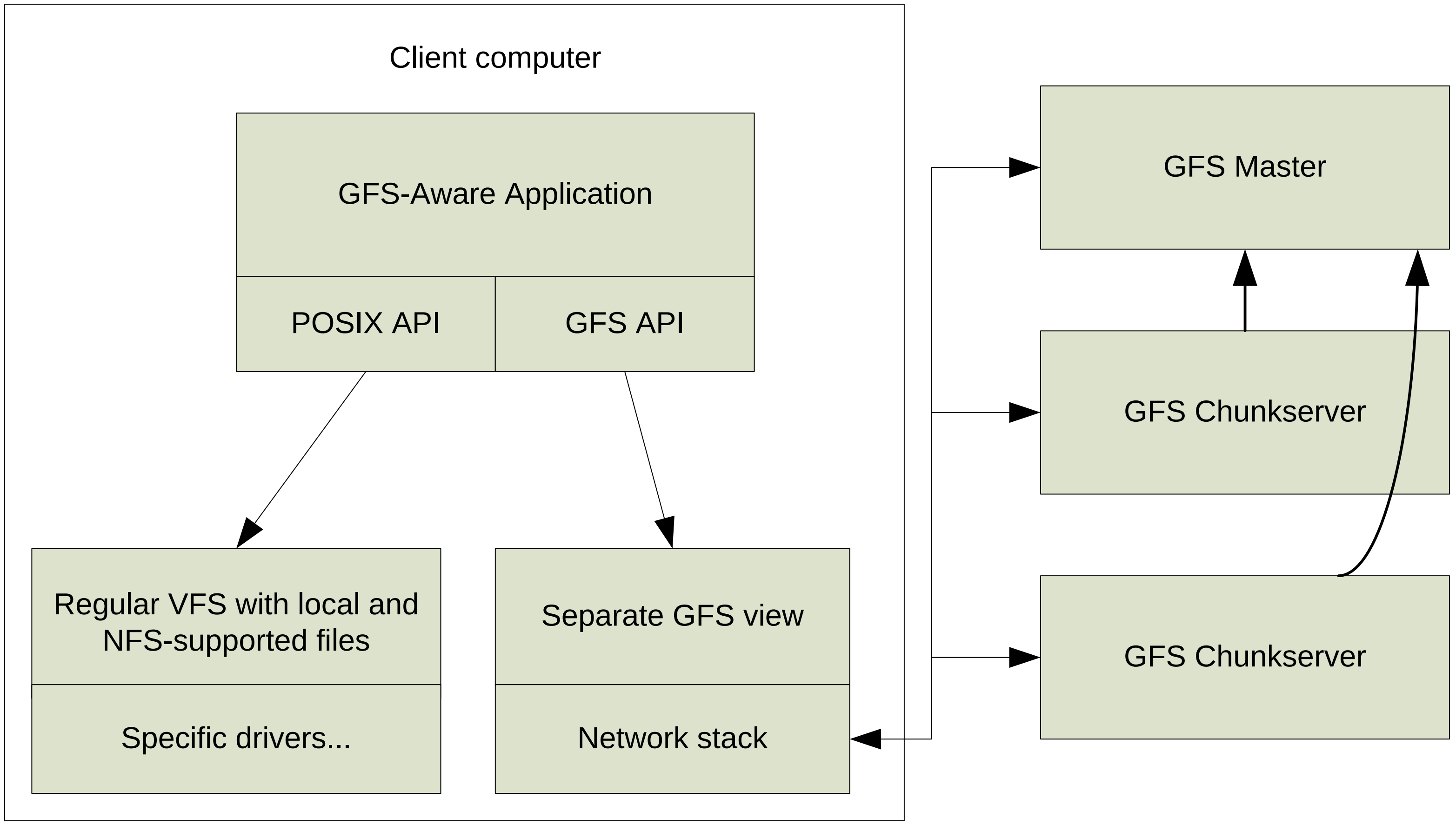
- Files split into 128MB blocks
- Blocks replicated across several datanodes (often 3)
- Namenode stores metadata (file names, locations, etc)
- Optimized for large files, sequential reads
- Files are append-only



Hadoop DFS

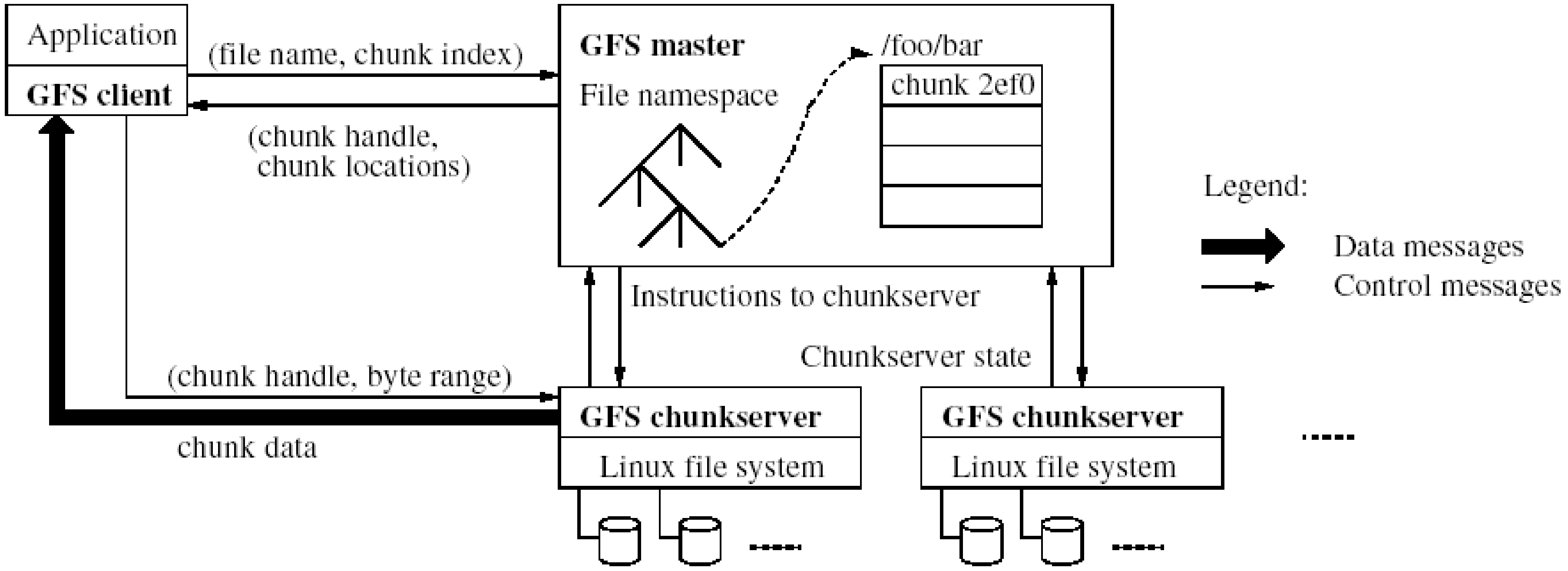

Client
Namenode
Secondary namenode
Datanodes 1
Datanodes 2
Datanodes 3
Datanodes n
Data replication

GFS Client Block Diagram

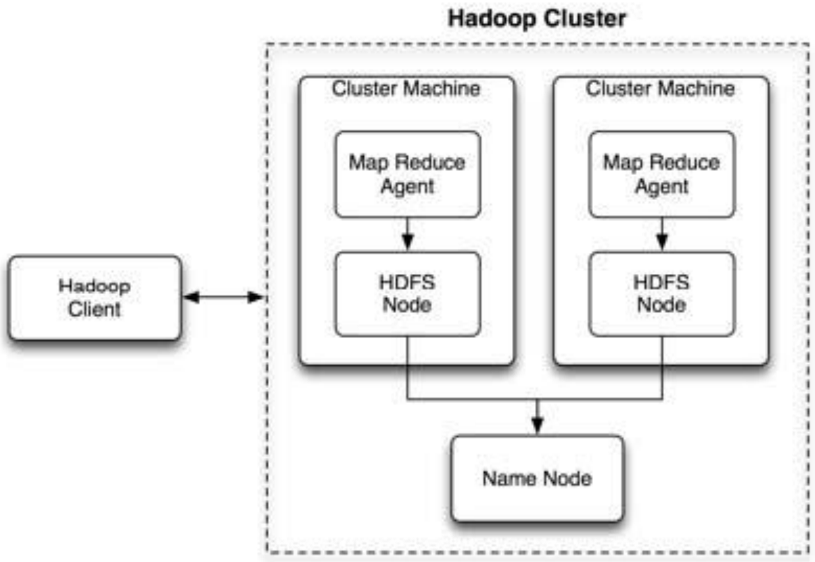


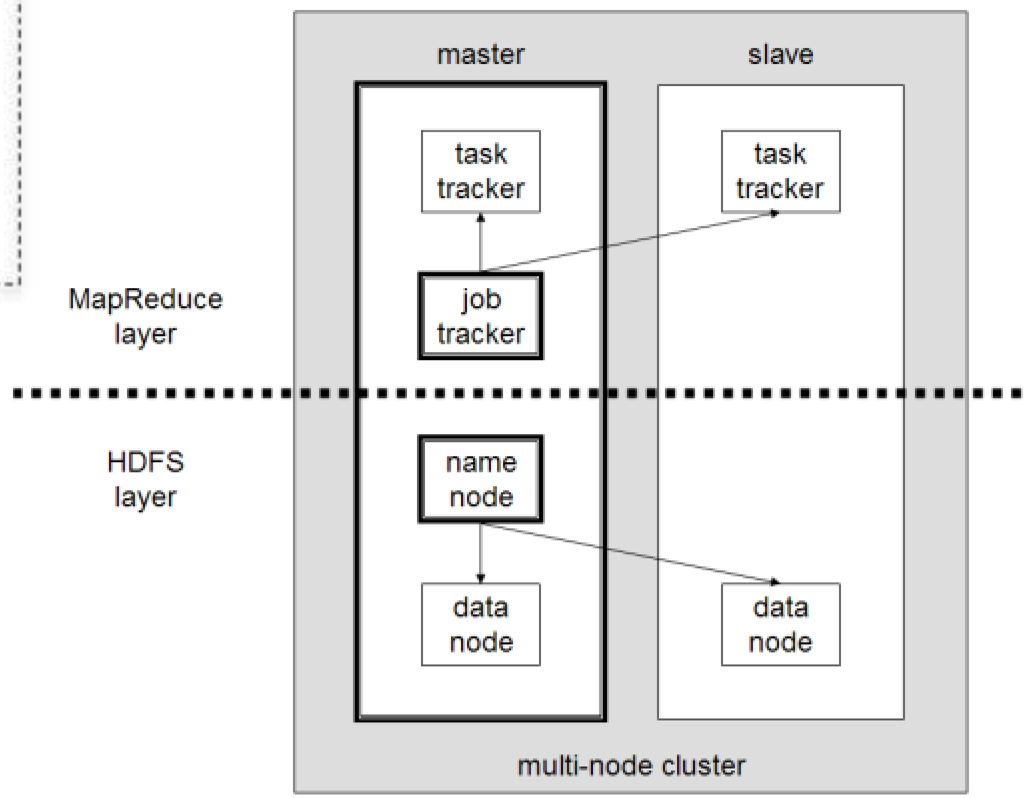
- **Provide both POSIX standard file interface, and costumed API**
- **Can cache meta data for direct client-chunk server access**

Read/write access flow in GFS



Hadoop DFS with MapReduce





MapReduce: Execution overview

Master Server distributes M map tasks to machines and monitors their progress.

↓

Map task reads the allocated data, saves the map results in local buffer.

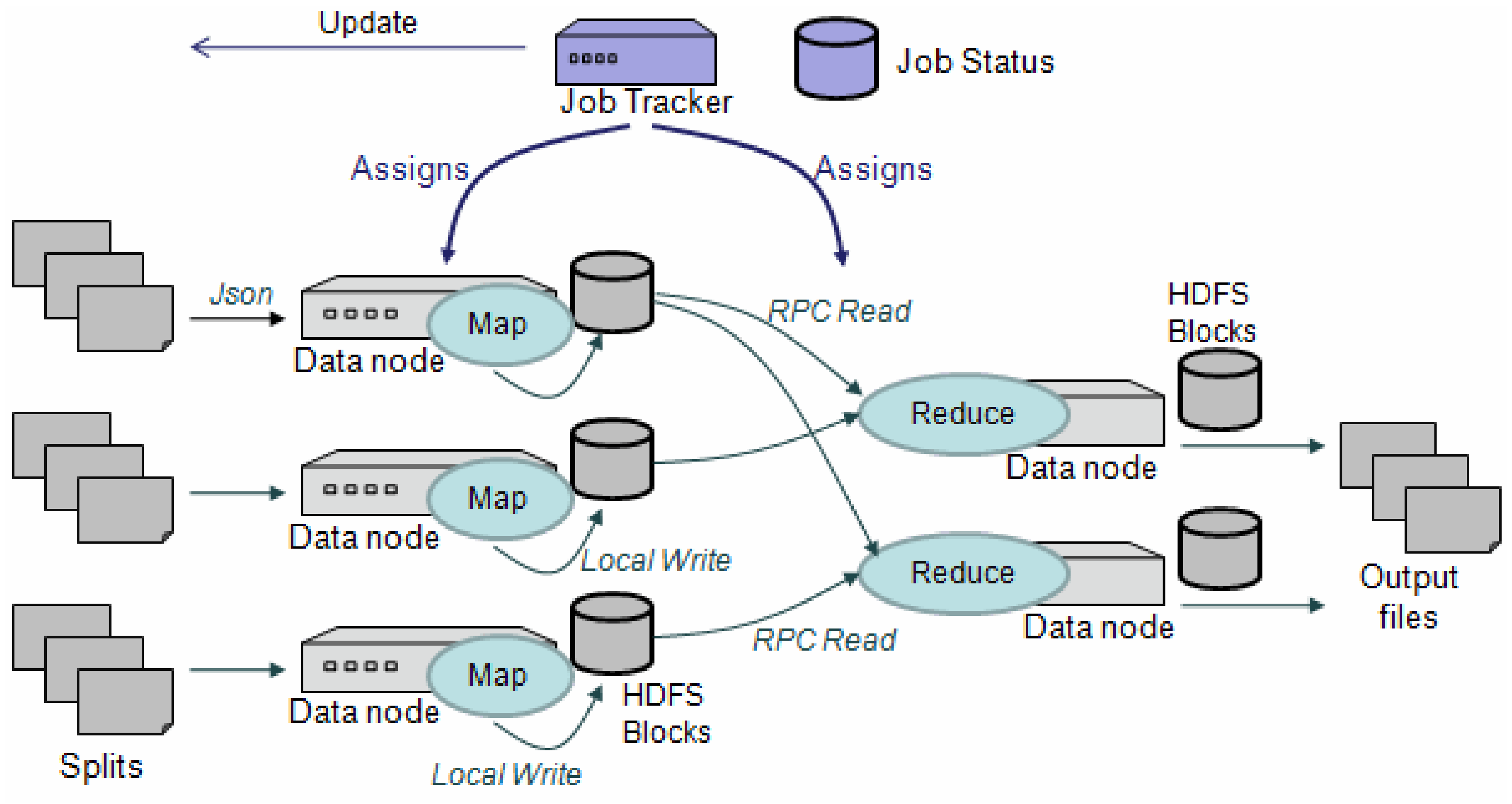
↓

Shuffle phase assigns reducers to these buffers, which are remotely read and processed by reducers.

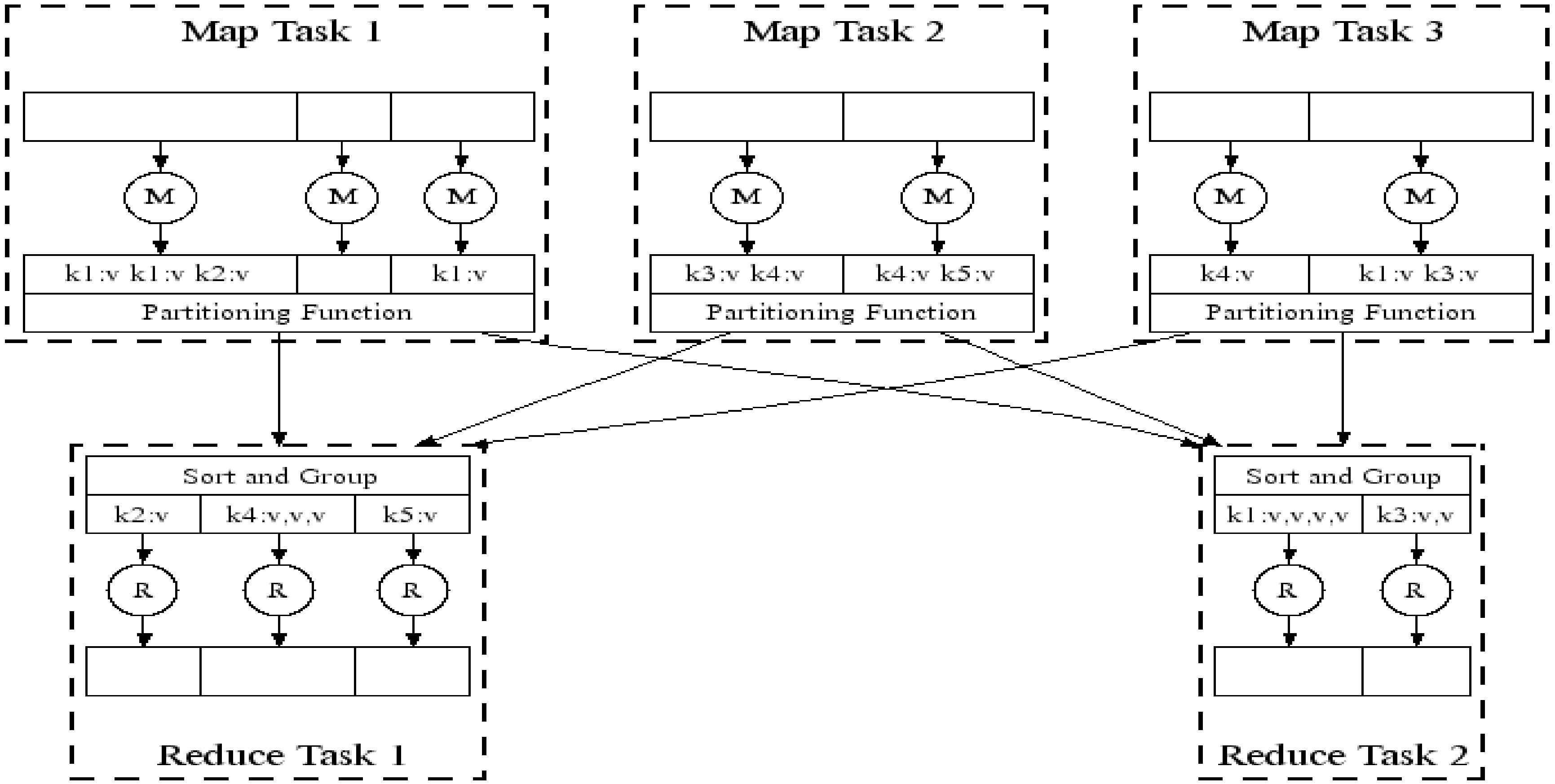
↓

Reducers output the result on stable storage.

Execute MapReduce on a cluster of machines with Hadoop DFS



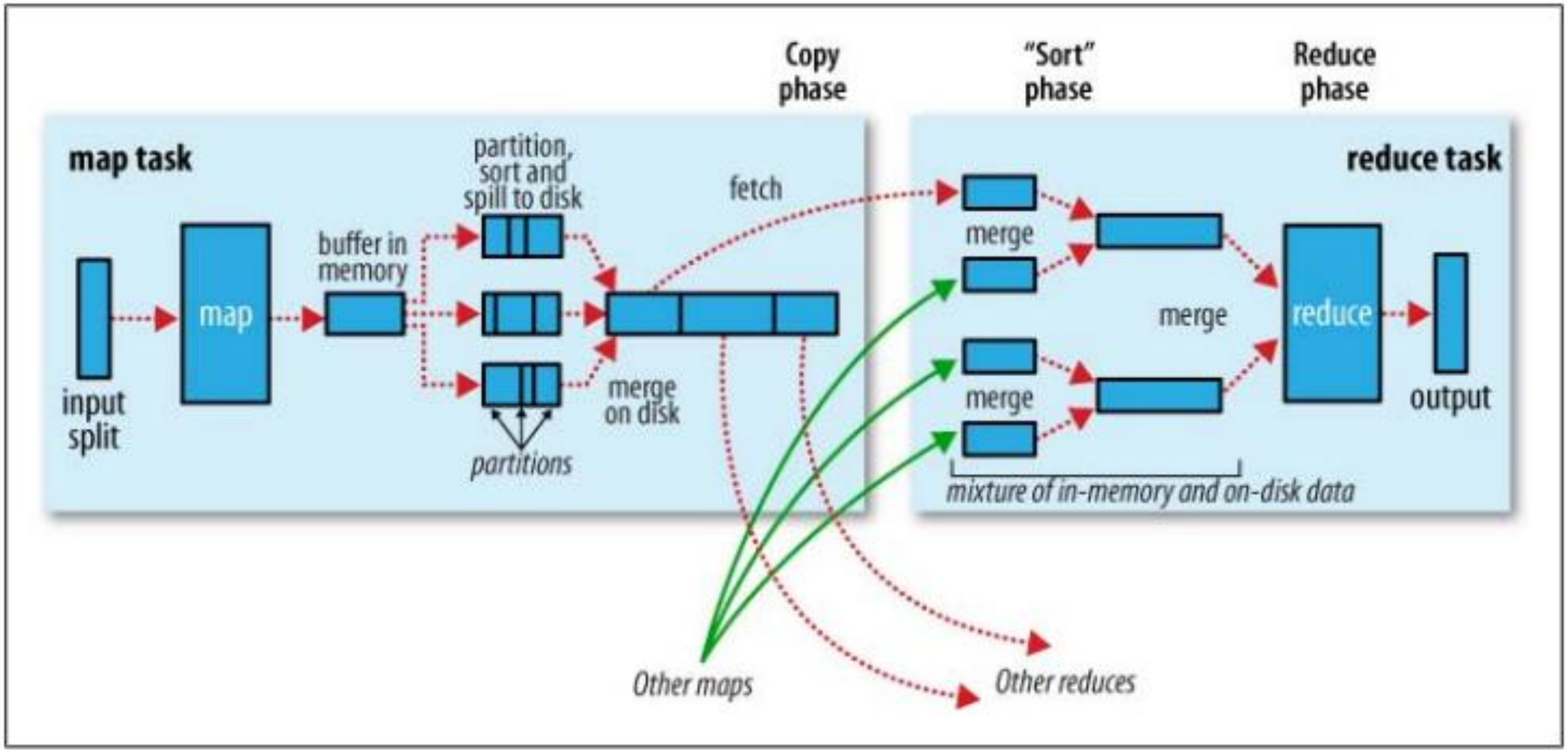
MapReduce in Parallel: Example



MapReduce: Execution Details

- **Input reader**
  - Divide input into splits, assign each split to a Map task
- **Map task**
  - Apply the Map function to each record in the split
  - Each Map function returns a list of (key, value) pairs
- **Shuffle/Partition and Sort**
  - Shuffle distributes sorting & aggregation to many reducers
  - All records for key *k* are directed to the same reduce processor
  - Sort groups the same keys together, and prepares for aggregation
- **Reduce task**
  - Apply the Reduce function to each key
  - The result of the Reduce function is a list of (key, value) pairs

MapReduce with data shuffling& sorting

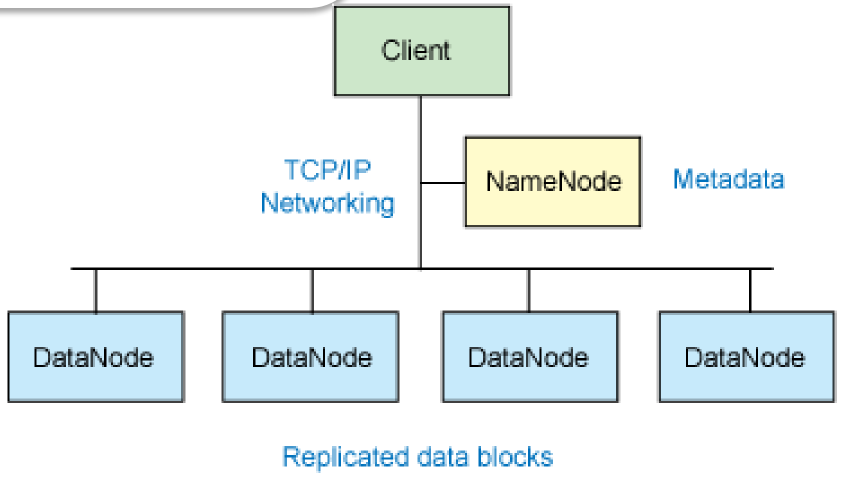


Tom White, *Hadoop: The Definitive Guide*

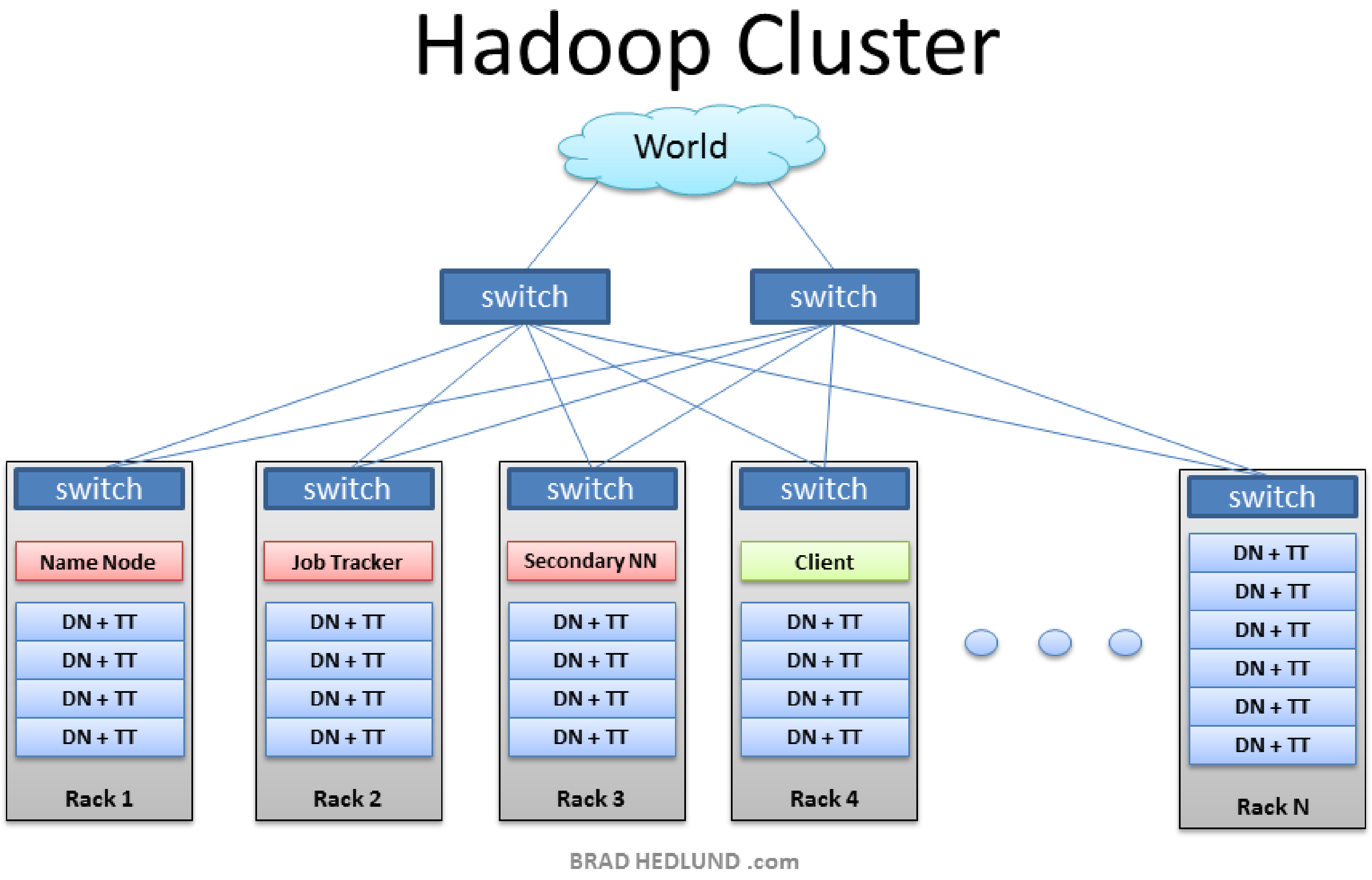
MapReduce: Runtime Environment &Hadoop

MapReduce Runtime Environment

- Partitioning the input data.
- Scheduling program across cluster of machines, Locality Optimization and Load balancing
- Dealing with machine failure
- Managing Inter-Machine communication



Hadoop Cluster with MapReduce



MapReduce: Fault Tolerance

- **Handled via re-execution of tasks.**
  - Task completion committed through master
- **Mappers save outputs to local disk before serving to reducers**
  - Allows recovery if a reducer crashes
  - Allows running more reducers than # of nodes

•**If a task crashes:**

- Retry on another node
  - OK for a map because it had no dependencies
  - OK for reduce because map outputs are on disk
- If the same task repeatedly fails, fail the job or ignore that input block
- : For the fault tolerance to work, *user tasks must be deterministic and side-effect-free*

2. If a node crashes:

- Relaunch its current tasks on other nodes
- Relaunch any maps the node previously ran
  - Necessary because their output files were lost along with the crashed node

MapReduce:
Locality Optimization

- Leverage the distributed file system to schedule a map task on a machine that contains a replica of the corresponding input data.
- Thousands of machines read input at local disk speed
- Without this, rack switches limit read rate

MapReduce: Redundant Execution

- Slow workers are source of bottleneck, may delay completion time.
- Near end of phase, spawn backup tasks, one to finish first wins.
- Effectively utilizes computing power, reducing job completion time by a factor.

MapReduce: Skipping Bad Records

- Map/Reduce functions sometimes fail for particular inputs.
- Fixing the Bug might not be possible : Third Party Libraries.
- On Error
  - Worker sends signal to Master
  - If multiple error on same record, skip record

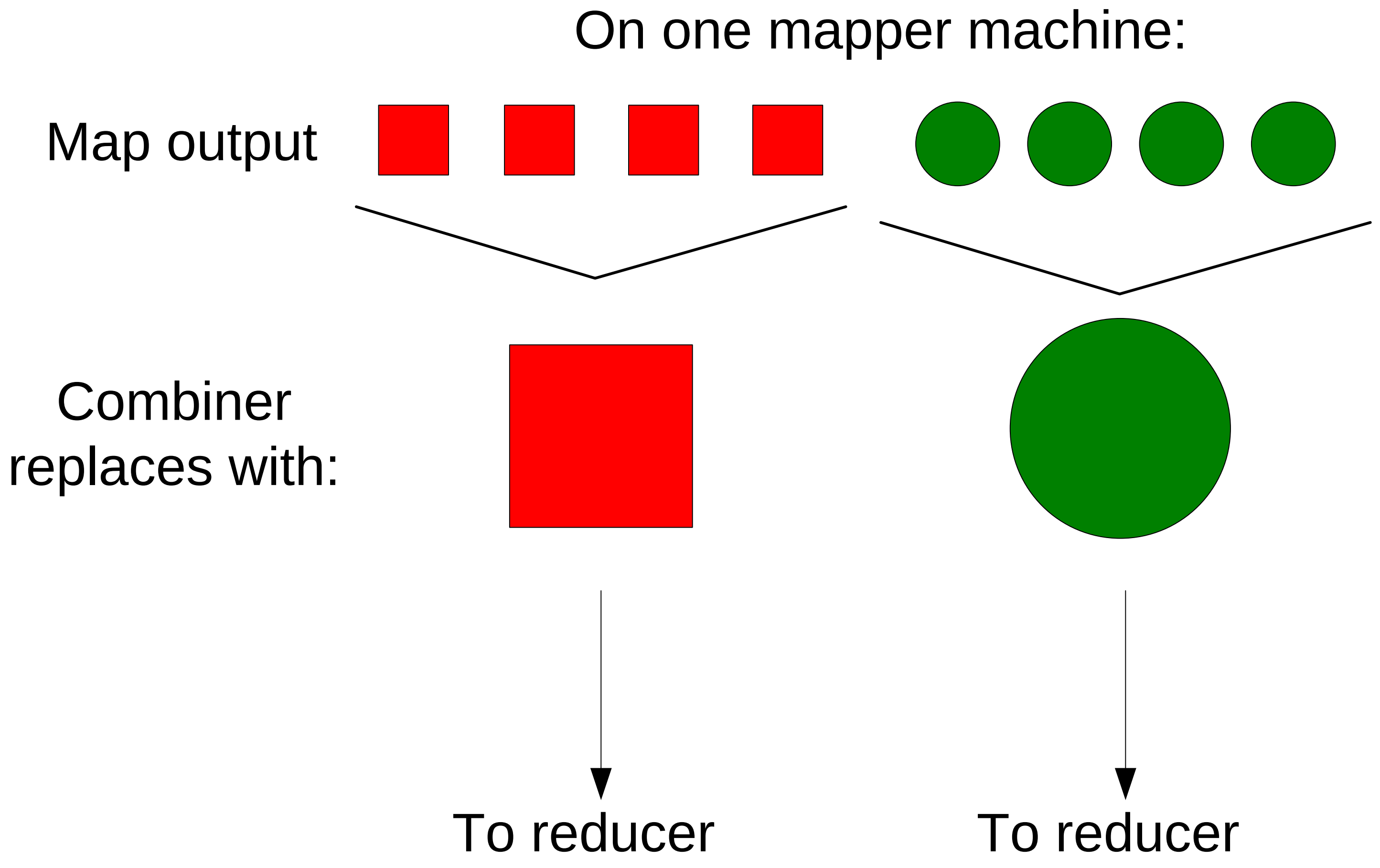
MapReduce: Miscellaneous Refinements

- Combiner function at a map task
- Sorting Guarantees within each reduce partition.
- Local execution for debugging/testing
- User-defined counters

Combining Phase

- Run on map machines after map phase
- “Mini-reduce,” only on local map output
- Used to save bandwidth before sending data to full reduce tasks
- Reduce tasks can be combiner if commutative & associative

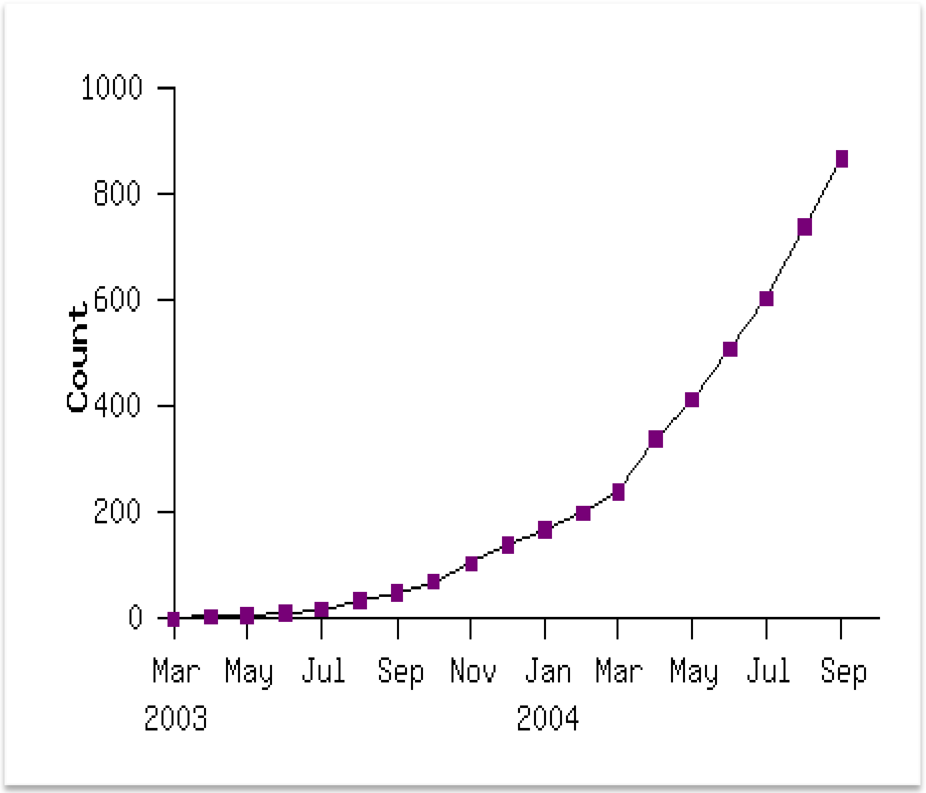
Combiner, graphically



Examples of MapReduce Usage in Web Applications

- Distributed Grep.
- Count of URL Access Frequency.
- Clustering (K-means)
- Graph Algorithms.
- Indexing Systems

MapReduce Programs In Google Source Tree



Hadoop and Tools

- **Various Linux Hadoop clusters around**
  - Cluster +Hadoop
    - » http://hadoop.apache.org
  - Amazon EC2
- **Winows and other platforms**
  - The NetBeans plugin simulates Hadoop
  - The workflow view works on Windows
- **Hadoop-based tools**
  - For Developing in Java, NetBeans plugin
- **Pig Latin,** a SQL-like high level data processing script language
- **Hive,** Data warehouse, SQL
- **Mahout,** Machine Learning algorithms on Hadoop
- **HBase,** Distributed data store as a large table

More MapReduce Applications

- Map Only processing
- Filtering and accumulation
- Database join
- Reversing graph edges
- Producing inverted index for web search
- PageRank graph processing

MapReduce Use Case 1: Map Only

Data distributive tasks – Map Only

- **E.g. classify individual documents**
- **Map does everything**
  - Input: (docno, doc_content), …
  - Output: (docno, [class, class, …]), …
- **No reduce tasks**

MapReduce Use Case 2: Filtering and Accumulation

Filtering & Accumulation – Map and Reduce

- **E.g. Counting total enrollments of two given student classes**
- **Map** selects records and outputs initial counts
  - In: (Jamie, 11741), (Tom, 11493), …
  - Out: (11741, 1), (11493, 1), …
- **Shuffle/Partition** by class_id
- **Sort**
  - In: (11741, 1), (11493, 1), (11741, 1), …
  - Out: (11493, 1), …, (11741, 1), (11741, 1), …
- **Reduce accumulates counts**
  - In: (11493, [1, 1, …]), (11741, [1, 1, …])
  - Sum and Output: (11493, 16), (11741, 35)

MapReduce Use Case 3: Database Join

- A JOIN is a means for combining fields from two tables by using values common to each.
- Example :For each employee, find the department he works in

Employee Table

LastName	DepartmentID
Rafferty	31
Jones	33
Steinberg	33
Robinson	34
Smith	34

JOIN

Pred:

EMPLOYEE.DepID= DEPARTMENT.DepID

Department Table

DepartmentID	DepartmentName
31	Sales
33	Engineering
34	Clerical
35	Marketing

JOIN RESULT

LastName	DepartmentName
Rafferty	Sales
Jones	Engineering
Steinberg	Engineering
…	…

MapReduce Use Case 3 – Database Join

Problem: Massive lookups

- Given two large lists: (URL, ID) and (URL, doc_content) pairs
- Produce (URL, ID, doc_content) or (ID, doc_content)

Solution:

- **Input stream**: both (URL, ID) and (URL, doc_content) lists
  - (http://del.icio.us/post, 0), (http://digg.com/submit, 1), …
  - (http://del.icio.us/post, <html0>), (http://digg.com/submit, <html1>), …
- **Map** simply passes input along,
- **Shuffle and Sort on URL** (group ID & doc_content for the same URL together)
  - Out: (http://del.icio.us/post, 0), (http://del.icio.us/post, <html0>), (http://digg.com/submit, <html1>), (http://digg.com/submit, 1), …
- **Reduce** outputs result stream of (ID, doc_content) pairs
  - In: (http://del.icio.us/post, [0, html0]), (http://digg.com/submit, [html1, 1]), …
  - Out: (0, <html0>), (1, <html1>), …

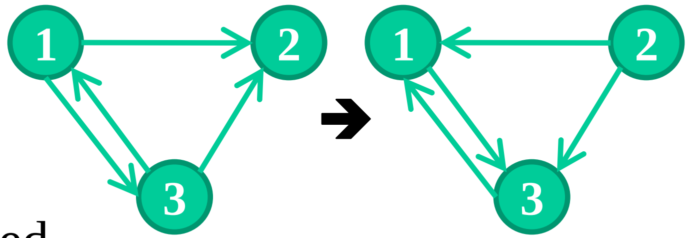
MapReduce Use Case 4: Reverse graph edge directions & output in node order

- **Input example: adjacency list of graph (3 nodes and 4 edges)**

  (3, [1, 2]) (1, [3])

  (1, [2, 3]) ➔ (2, [1, 3])

  (3, [1])



- node_ids in the output **values** are also sorted. But Hadoop only sorts on keys!
- **MapReduce format**
  - Input: (3, [1, 2]), (1, [2, 3]).
  - Intermediate: (1, [3]), (2, [3]), (2, [1]), (3, [1]). (reverse edge direction)
  - Out: (1,[3]) (2, [1, 3]) (3, [[1]).

MapReduce Use Case 5: Inverted Indexing Preliminaries

Construction of inverted lists for document search

- Input: documents: (docid, [term, term..]), (docid, [term, ..]), ..
- Output: (term, [docid, docid, …])
  - E.g., (apple, [1, 23, 49, 127, …])

A document id is an <u>internal document id</u>, e.g., a unique integer

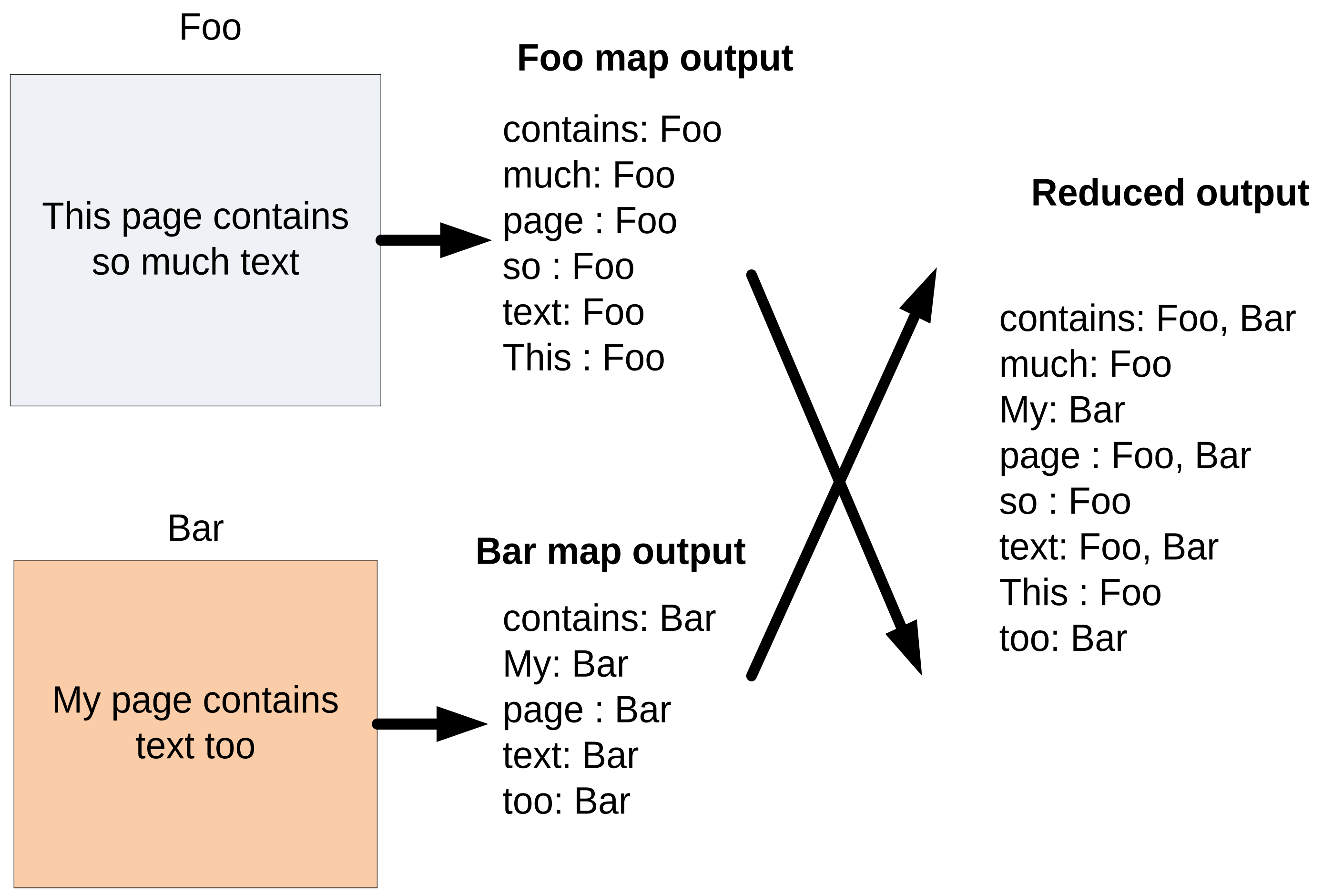
- <u>Not</u> an external document id such as a url

Using MapReduce to Construct Indexes: A Simple Approach

A simple approach to creating inverted lists

- **Each Map task is a document parser**
  - Input: A stream of documents
  - Output: A stream of (term, docid) tuples
    - » (long, 1) (ago, 1) (and, 1) … (once, 2) (upon, 2) …
    - » We may create internal IDs for words.
- **Shuffle sorts tuples by key and routes tuples to Reducers**
- **Reducers convert streams of keys into streams of inverted lists**
  - Input: (long, 1) (long, 127) (long, 49) (long, 23) …
  - The reducer sorts the values for a key and builds an inverted list
  - Output: (long, [df:492, docids:1, 23, 49, 127, …])

Inverted Index: Data flow



Foo map output

contains: Foo
much: Foo
page : Foo
so : Foo
text: Foo
This : Foo

Bar map output

contains: Bar
My: Bar
page : Bar
text: Bar
too: Bar

Reduced output

contains: Foo, Bar
much: Foo
My: Bar
page : Foo, Bar
so : Foo
text: Foo, Bar
This : Foo
too: Bar

Processing Flow Optimization

A more detailed analysis of processing flow

- **Map:** (docid$_1$, content$_1$) → (t$_1$, docid$_1$) (t$_2$, docid$_1$) …
- **Shuffle** by t, prepared for map-reducer communication
- **Sort** by t, conducted in a reducer machine

  (t$_5$, docid$_1$) (t$_4$, docid$_3$) … → (t$_4$, docid$_3$) (t$_4$, docid$_1$) (t$_5$, docid$_1$) …
- **Reduce:** (t$_4$, [docid$_3$ docid$_1$ …]) → (t, ilist)

docid:	a unique integer
t:	a term, e.g., “apple”
ilist:	a complete inverted list

but a) inefficient, b) docids are sorted in reducers, and c) assumes ilist of a word fits in memory

Using Combine () to Reduce Communication

- **Map:** (docid$_1$, content$_1$) $\rightarrow$ (t$_1$, ilist$_{1,1}$) (t$_2$, ilist$_{2,1}$) (t$_3$, ilist$_{3,1}$) …
  - Each output inverted list covers just <u>one document</u>
- **Combine locally**

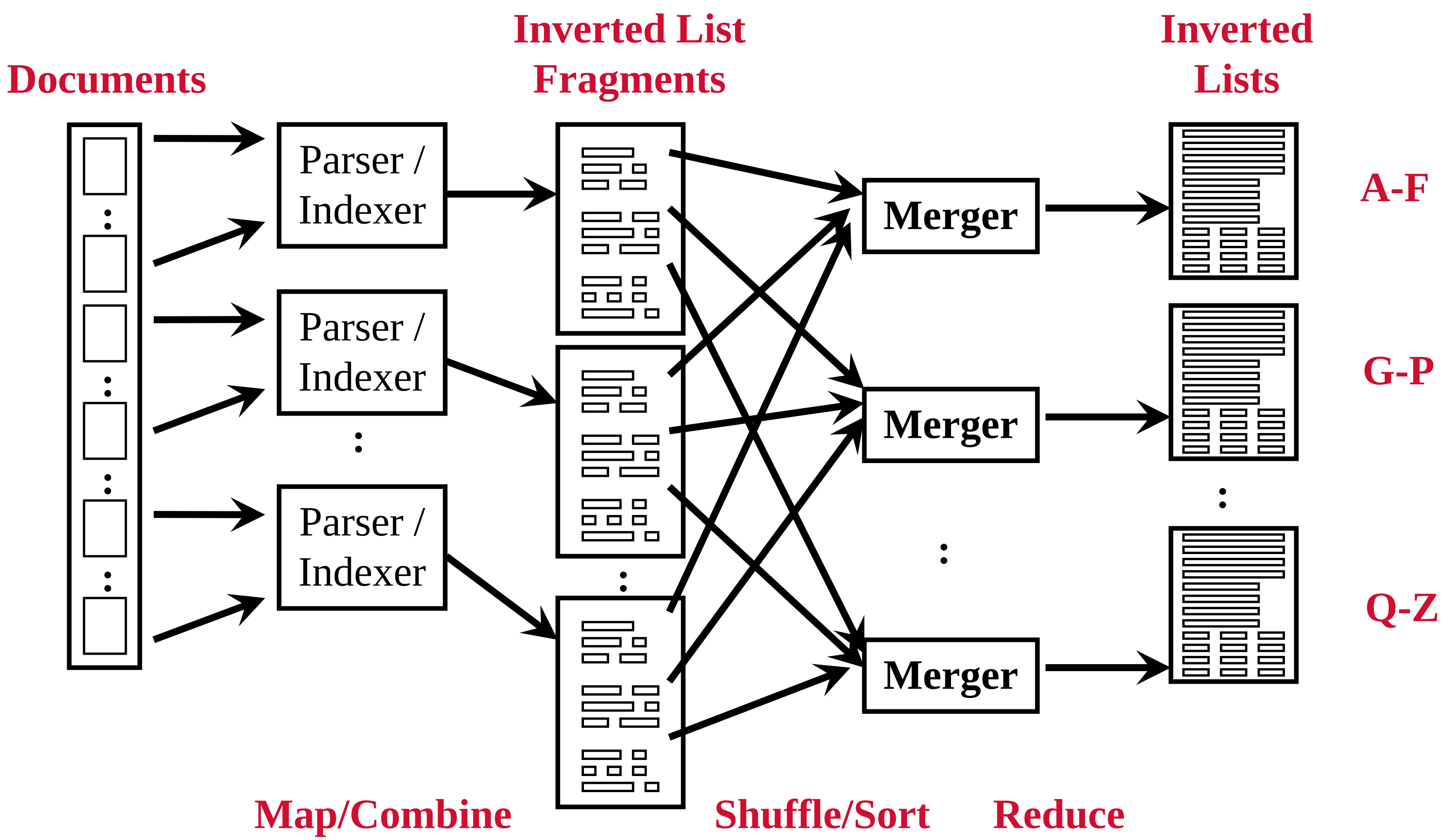
  Sort by t

  Combine: (t$_1$ [ilist$_{1,2}$ ilist$_{1,3}$ ilist$_{1,1}$ …]) $\rightarrow$ (t$_1$, ilist$_{1,27}$)
  - Each output inverted list covers a <u>sequence of documents</u>
- **Shuffle** by t
- **Sort** by t

  (t$_4$, ilist$_{4,1}$) (t$_5$, ilist$_{5,3}$) … $\rightarrow$ (t$_4$, ilist$_{4,2}$) (t$_4$, ilist$_{4,4}$) (t$_4$, ilist$_{4,1}$) …
- **Reduce:** (t$_7$, [ilist$_{7,2}$, ilist$_{3,1}$, ilist$_{7,4}$, …]) $\rightarrow$ (t$_7$, ilist$_{final}$)

ilist$_{i,j}$: the j’th inverted list fragment for term i

Using MapReduce to Construct Indexes



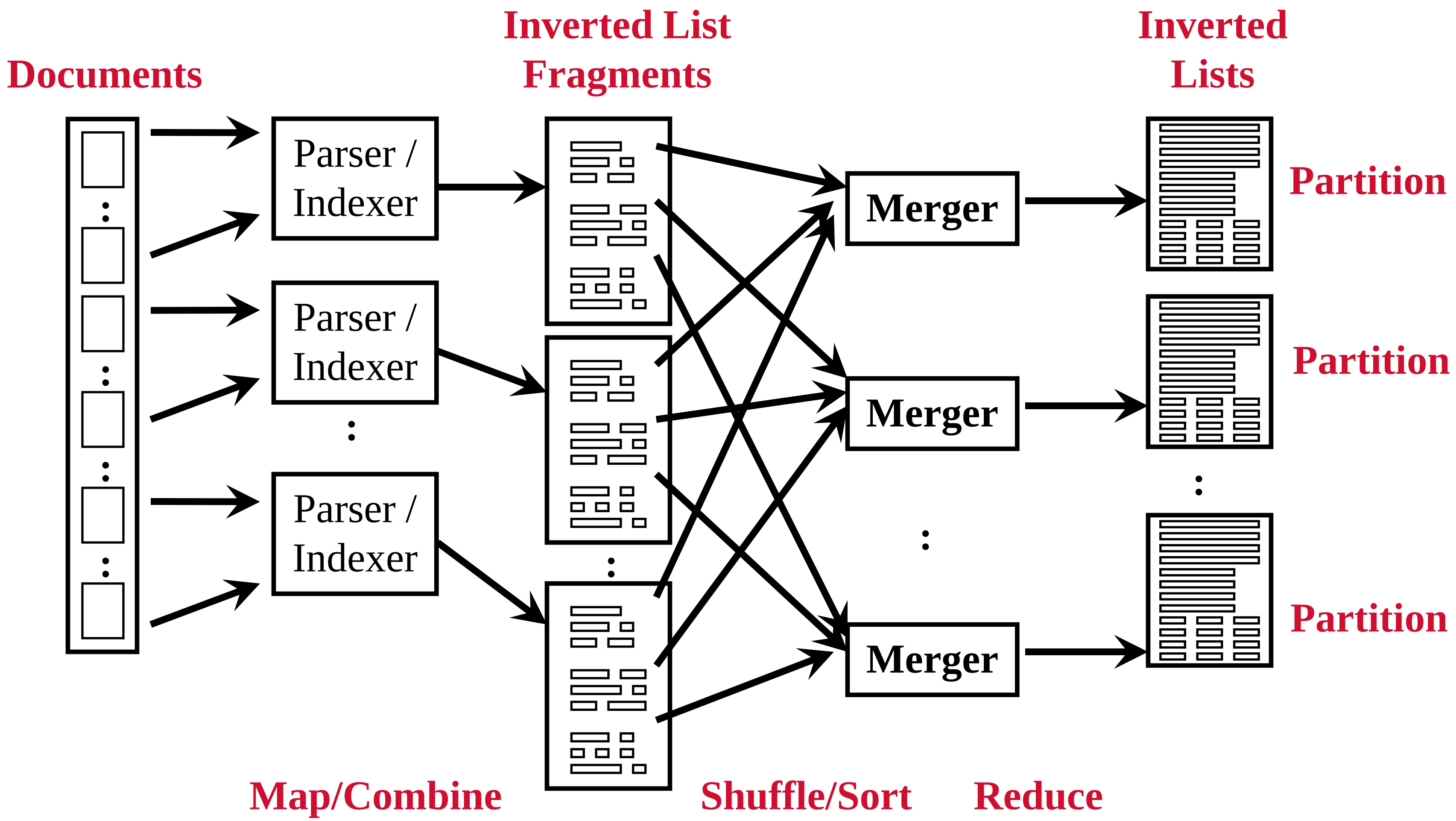
Construct Partitioned Indexes

- **Useful when the document list of a term does not fit memory**
- **Map:** (docid$_1$, content$_1$) → ([p, t$_1$], ilist$_{1,1}$)
- **Combine** to sort and group values

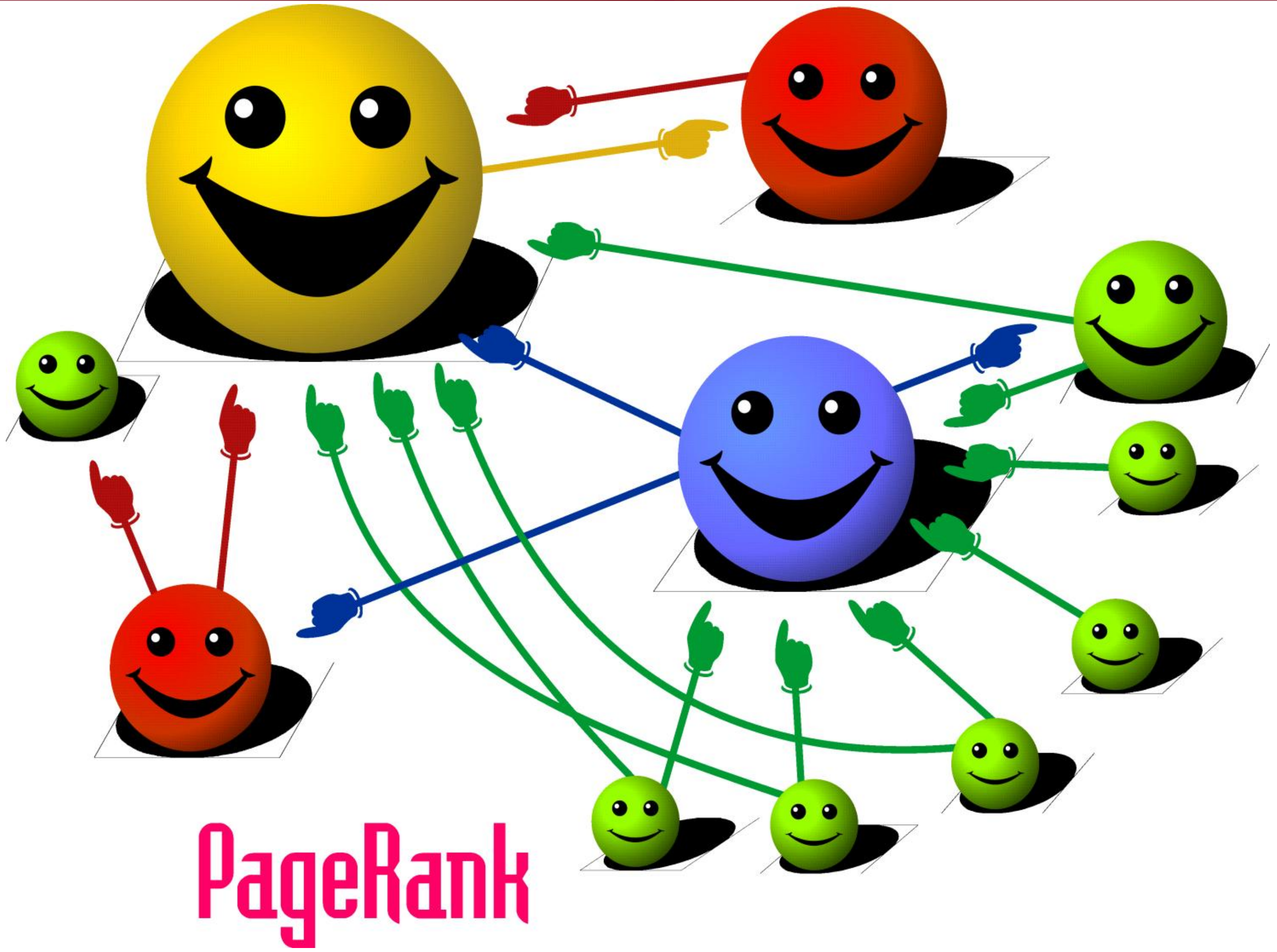
  ([p, t$_1$] [ilist$_{1,2}$ ilist$_{1,3}$ ilist$_{1,1}$ …]) → ([p, t$_1$], ilist$_{1,27}$)
- **Shuffle** by p
- **Sort** values by [p, t]
- **Reduce:** ([p, t$_7$], [ilist$_{7,2}$, ilist$_{7,1}$, ilist$_{7,4}$, …]) → ([p, t$_7$], ilist$_{final}$)

p: partition (shard) id

Generate Partitioned Index



MapReduce Use Case 6: PageRank

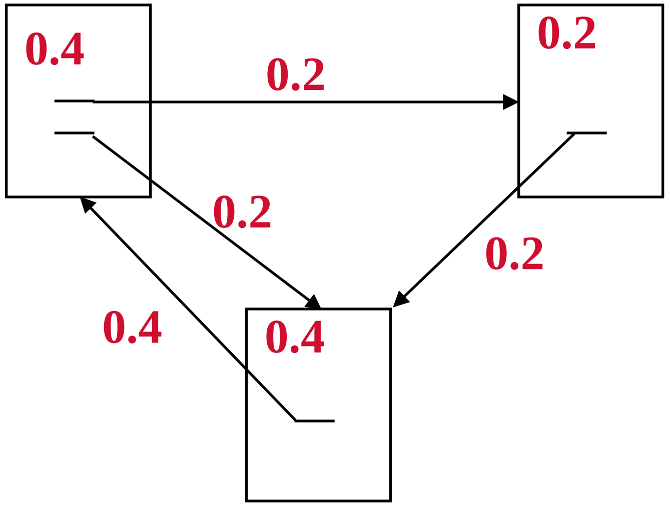


PageRank

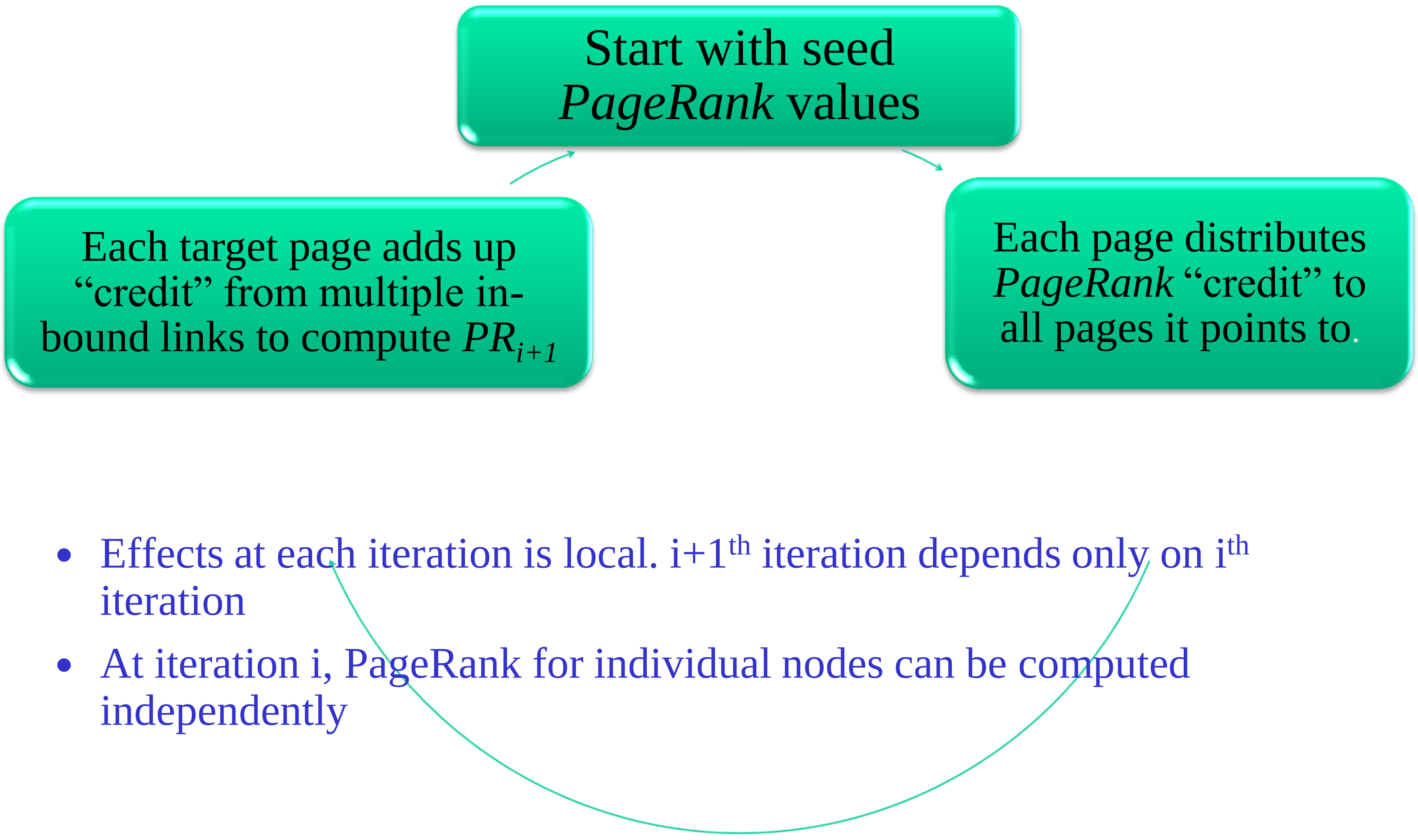
- Model page reputation on the web

$$PR(x) = (1-d) + d\sum_{i=1}^{n} \frac{PR(t_i)}{C(t_i)}$$

- i=1,n lists all parents of page x.
- PR(x) is the page rank of each page.
- C(t) is the out-degree of t.
- d is a damping factor .



Computing PageRank Iteratively



Start with seed *PageRank* values

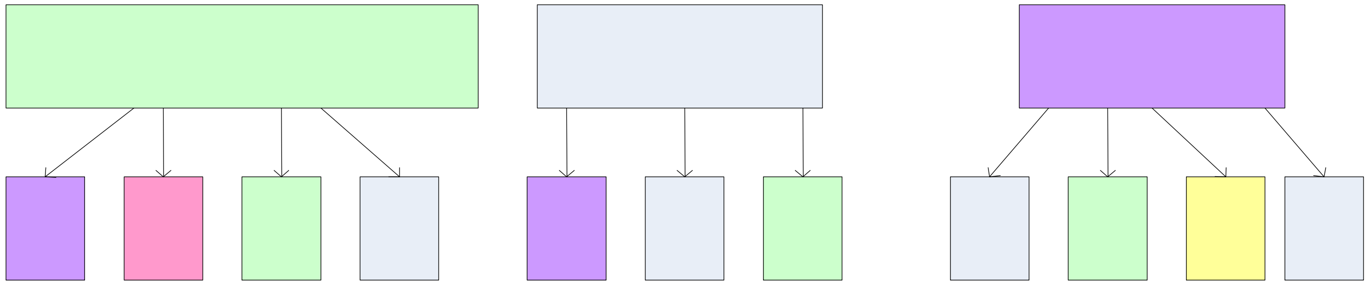
Each page distributes *PageRank* "credit" to all pages it points to.

Each target page adds up "credit" from multiple in-bound links to compute PR_{i+1}

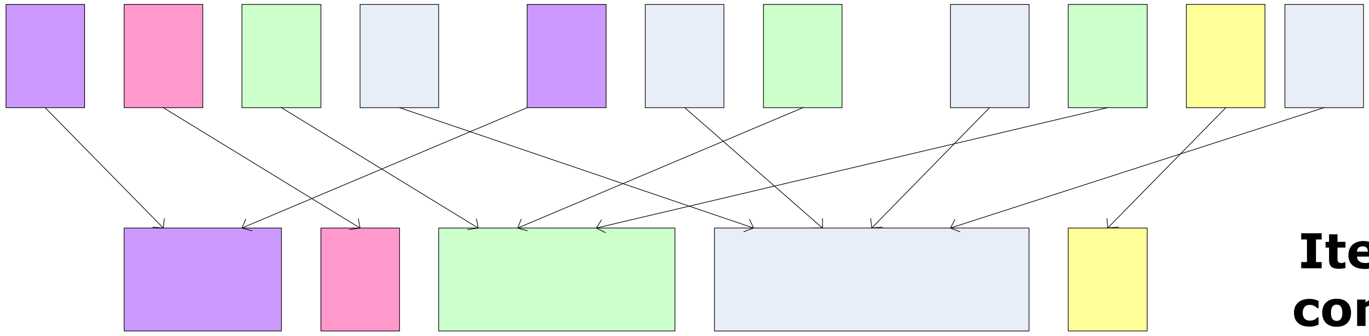
- Effects at each iteration is local. i+1th iteration depends only on ith iteration
- At iteration i, PageRank for individual nodes can be computed independently

PageRank using MapReduce

Map: distribute PageRank "credit" to link targets



Reduce: gather up PageRank "credit" from multiple sources to compute new PageRank value



Iterate until convergence

Source of Image: Lin 2008

PageRank Calculation: Preliminaries

One PageRank iteration:

- Input:
  - (id$_1$, [score$_1^{(t)}$, out$_{11}$, out$_{12}$, ..]), (id$_2$, [score$_2^{(t)}$, out$_{21}$, out$_{22}$, ..]) ..
- Output:
  - (id$_1$, [score$_1^{(t+1)}$, out$_{11}$, out$_{12}$, ..]), (id$_2$, [score$_2^{(t+1)}$, out$_{21}$, out$_{22}$, ..]) ..

MapReduce elements

- Score distribution and accumulation
- Database join

PageRank: Score Distribution and Accumulation

- **Map**
  - In: (id_1, [$score_1^{(t)}$, out_{11}, out_{12}, ..]), (id_2, [$score_2^{(t)}$, out_{21}, out_{22}, ..]) ..
  - Out: (out_{11}, $score_1^{(t)}/n_1$), (out_{12}, $score_1^{(t)}/n_1$) .., (out_{21}, $score_2^{(t)}/n_2$), ..
- **Shuffle & Sort by node_id**
  - In: (id_2, $score_1$), (id_1, $score_2$), (id_1, $score_1$), ..
  - Out: (id_1, $score_1$), (id_1, $score_2$), .., (id_2, $score_1$), ..
- **Reduce**
  - In: (id_1, [$score_1$, $score_2$, ..]), (id_2, [$score_1$, ..]), ..
  - Out: (id_1, $score_1^{(t+1)}$), (id_2, $score_2^{(t+1)}$), ..

PageRank: Database Join to associate outlinks with score

- **Map**
  - In & Out: (id$_1$, score$_1^{(t+1)}$), (id$_2$, score$_2^{(t+1)}$), .., (id$_1$, [out$_{11}$, out$_{12}$, ..]), (id$_2$, [out$_{21}$, out$_{22}$, ..]) ..
- **Shuffle & Sort by node_id**
  - Out: (id$_1$, score$_1^{(t+1)}$), (id$_1$, [out$_{11}$, out$_{12}$, ..]), (id$_2$, [out$_{21}$, out$_{22}$, ..]), (id$_2$, score$_2^{(t+1)}$), ..
- **Reduce**
  - In: (id$_1$, [score$_1^{(t+1)}$, out$_{11}$, out$_{12}$, ..]), (id$_2$, [out$_{21}$, out$_{22}$, .., score$_2^{(t+1)}$]), ..
  - Out: (id$_1$, [score$_1^{(t+1)}$, out$_{11}$, out$_{12}$, ..]), (id$_2$, [score$_2^{(t+1)}$, out$_{21}$, out$_{22}$, ..]) ..

Conclusions

- **MapReduce advantages**
- **Application cases**
  - – Map only: for totally distributive computation
  - – Map+Reduce: for filtering & aggregation
  - – Database join: for massive dictionary lookups
  - – Secondary sort: for sorting on values
  - – Inverted indexing: combiner, complex keys
  - – PageRank: side effect files

For More Information

- J. Dean and S. Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters.” *Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI 2004)*, pages 137-150. 2004.
- S. Ghemawat, H. Gobioff, and S.-T. Leung. “The Google File System.” *OSDI 200?*
- http://hadoop.apache.org/common/docs/current/mapred_tutorial.html. “Map/Reduce Tutorial”. Fetched January 21, 2010.
- Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media. June 5, 2009
- http://developer.yahoo.com/hadoop/tutorial/module4.html
- J. Lin and C. Dyer. *Data-Intensive Text Processing with MapReduce*. Book Draft. February 7, 2010.