Big Data and Internet Thinking

Chentao Wu
Associate Professor
Dept. of Computer Science and Engineering
wuct@cs.sjtu.edu.cn

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

Download lectures

- ftp://public.sjtu.edu.cn
- User: wuct
- Password: wuct123456

- http://www.cs.sjtu.edu.cn/~wuct/bdit/

Schedule

- lec1: Introduction on big data, cloud computing & IoT
- lec2: Parallel processing framework (e.g., MapReduce)
- lec3: Advanced parallel processing techniques (e.g., YARN, Spark)
- lec4: Cloud & Fog/Edge Computing
- lec5: Data reliability & data consistency
- lec6: Distributed file system & objected-based storage
- lec7: Metadata management & NoSQL Database
- lec8: Big Data Analytics

Collaborators

Google
谷歌

阿里云
aliyun.com

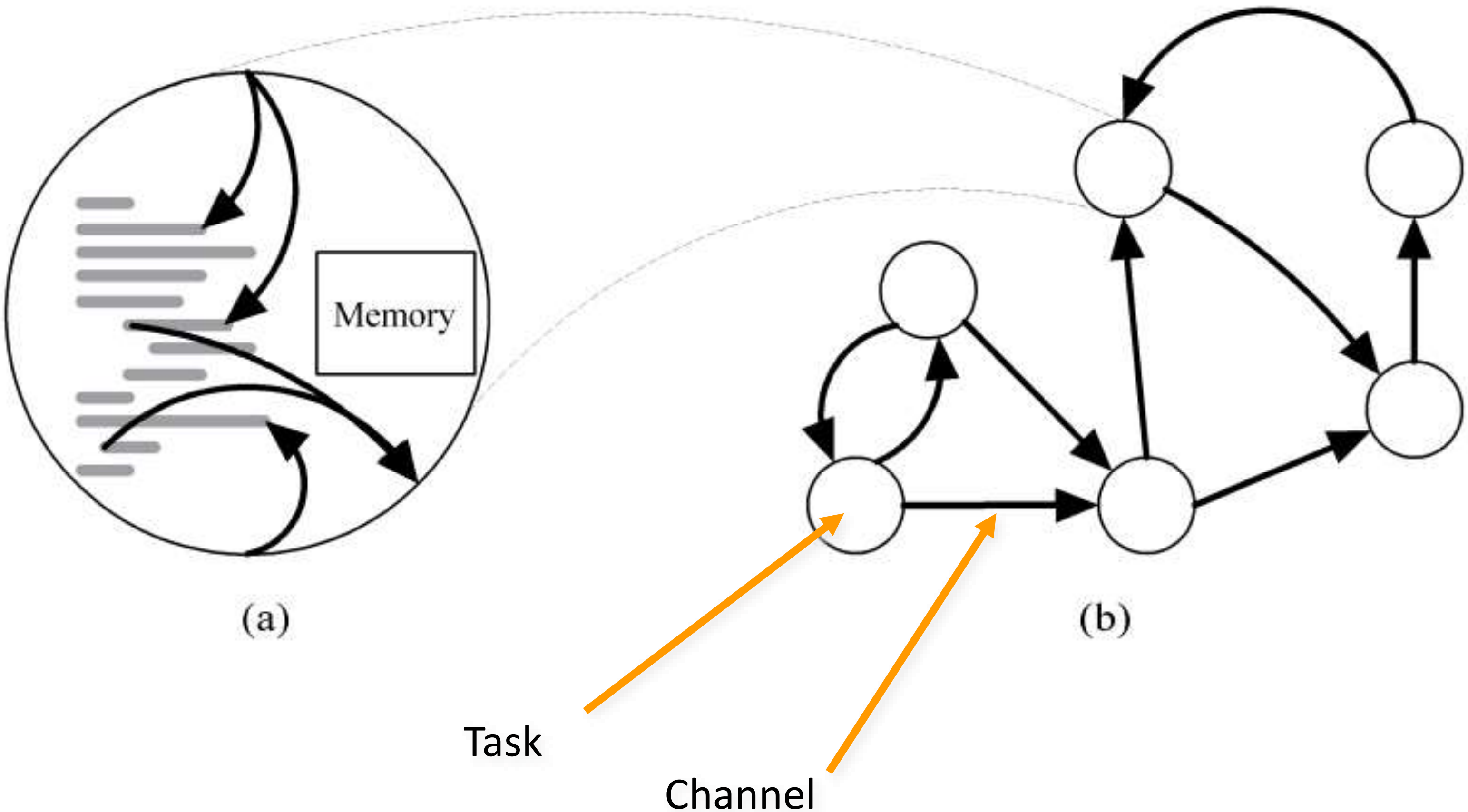
DELL EMC

1 Parallel Programming Basic

Task/Channel Model

- Parallel computation = set of tasks
- Task
  - Program
  - Local memory
  - Collection of I/O ports
- Tasks interact by sending messages through channels

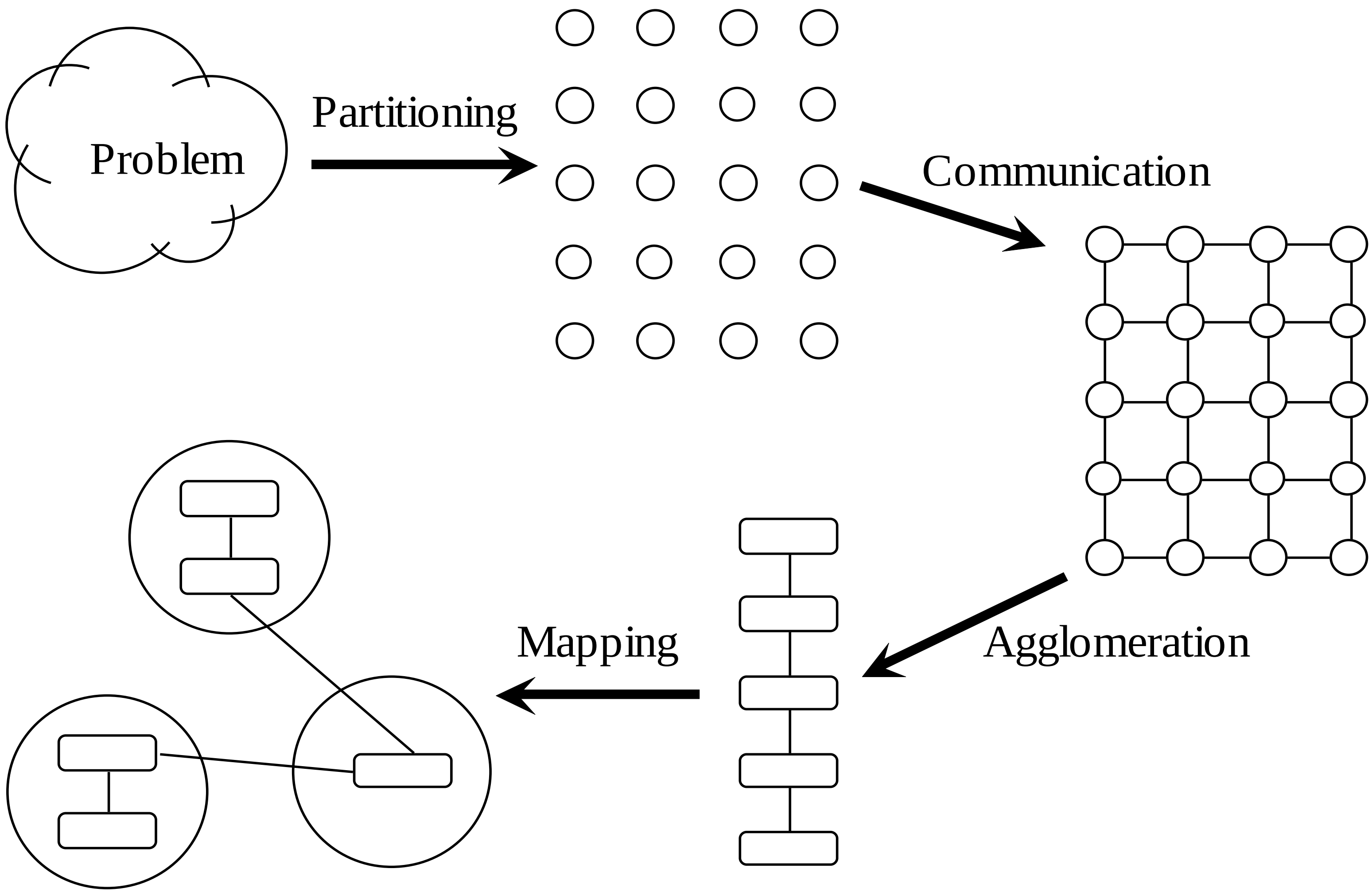
Task/Channel Model



Foster's Design Methodology

- Partitioning
- Communication
- Agglomeration
- Mapping

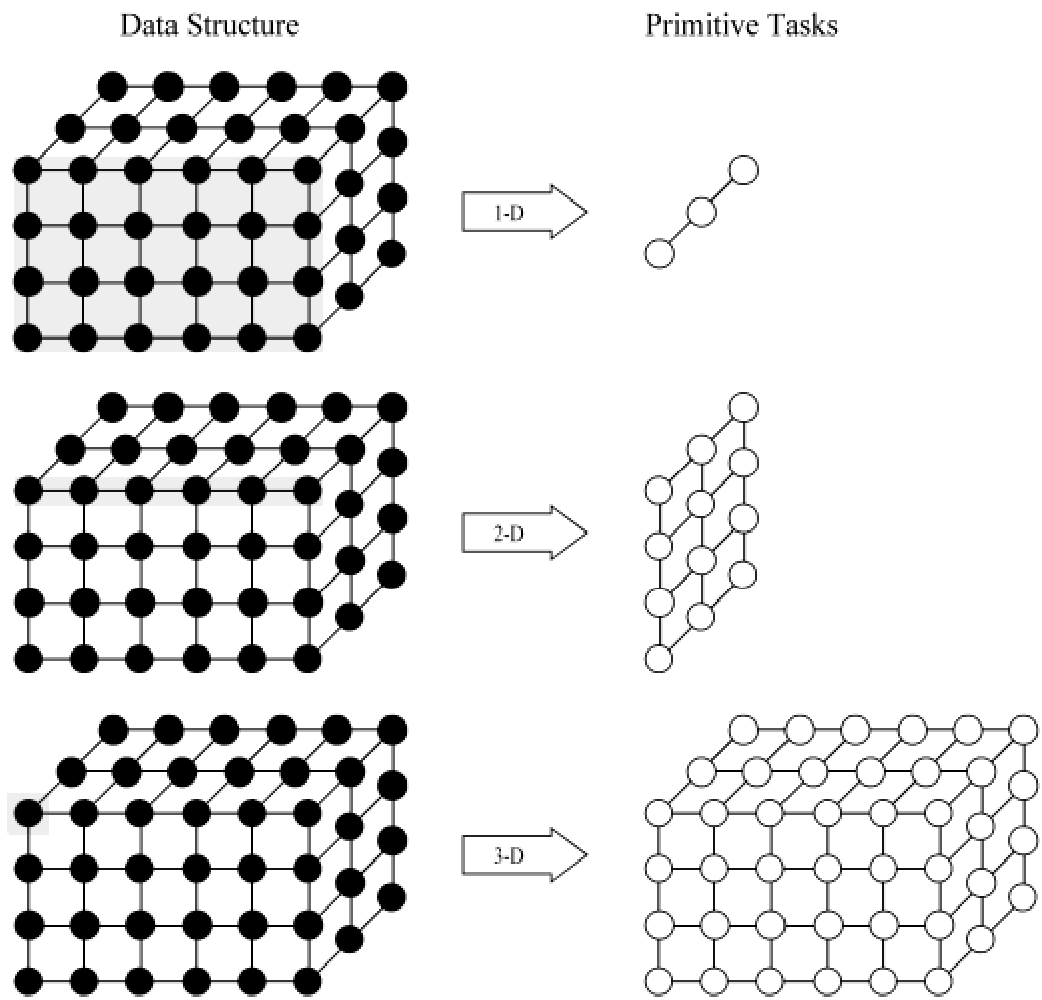
Foster's Design Methodology



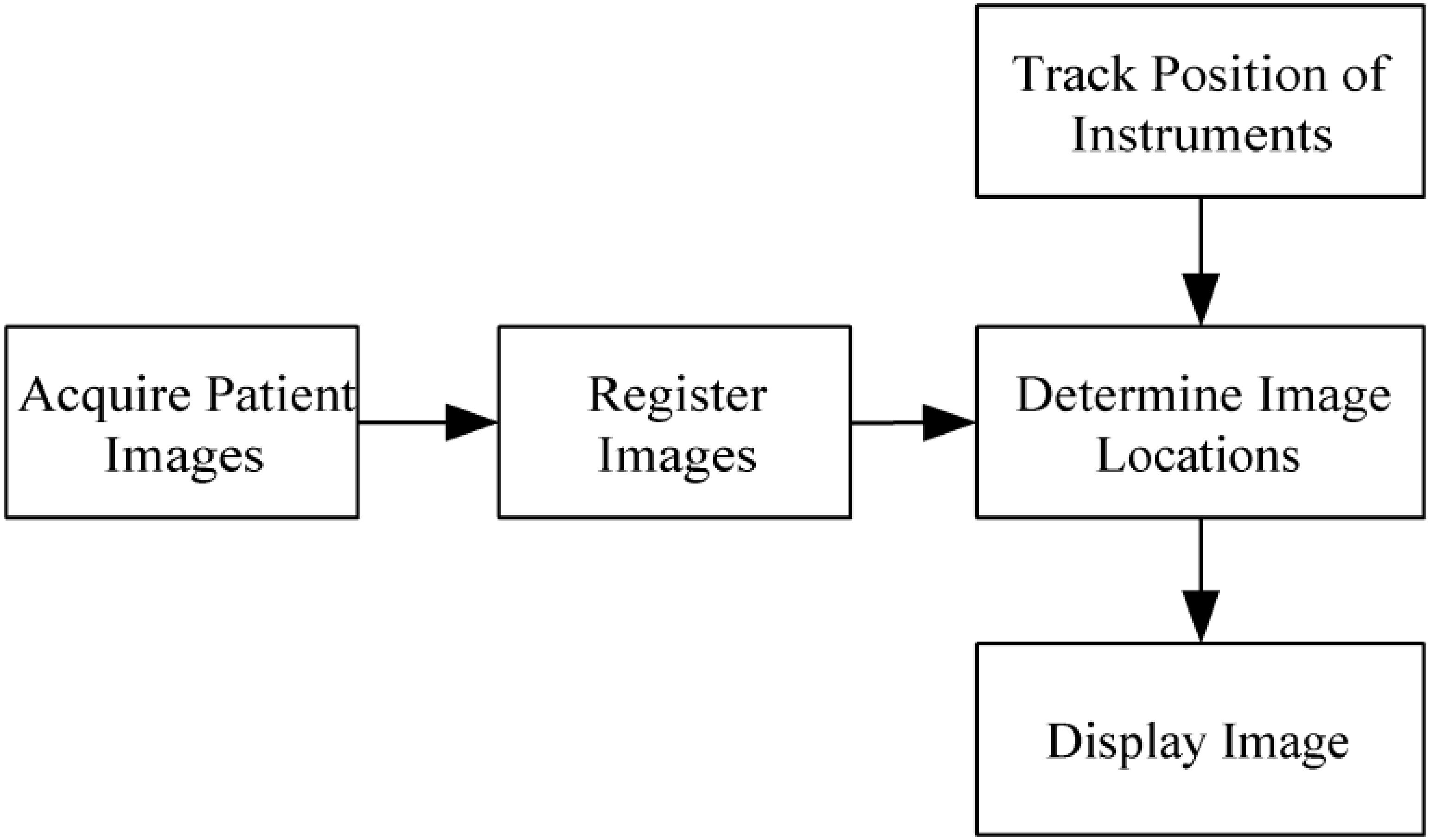
Partitioning

- Dividing computation and data into pieces
- Domain decomposition
  - Divide data into pieces
  - Determine how to associate computations with the data
- Functional decomposition
  - Divide computation into pieces
  - Determine how to associate data with the computations

Example Domain Decompositions



Example Functional Decomposition



Partitioning Checklist

- At least 10x more primitive tasks than processors in target computer
- Minimize redundant computations and redundant data storage
- Primitive tasks roughly the same size
- Number of tasks an increasing function of problem size

Communication

- Determine values passed among tasks
- Local communication
  - Task needs values from a small number of other tasks
  - Create channels illustrating data flow
- Global communication
  - Significant number of tasks contribute data to perform a computation
  - Don’t create channels for them early in design

Communication Checklist

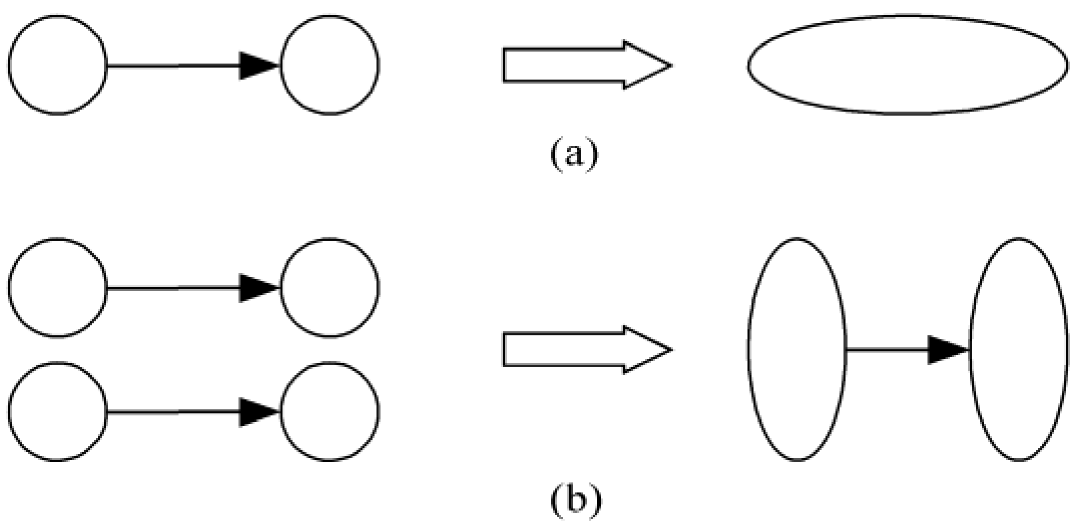
- Communication operations balanced among tasks
- Each task communicates with only small group of neighbors
- Tasks can perform communications concurrently
- Task can perform computations concurrently

Agglomeration

- Grouping tasks into larger tasks
- Goals
  - Improve performance
  - Maintain scalability of program
  - Simplify programming
- In MPI programming, goal often to create one agglomerated task per processor

Agglomeration Can Improve Performance

- Eliminate communication between primitive tasks agglomerated into consolidated task
- Combine groups of sending and receiving tasks



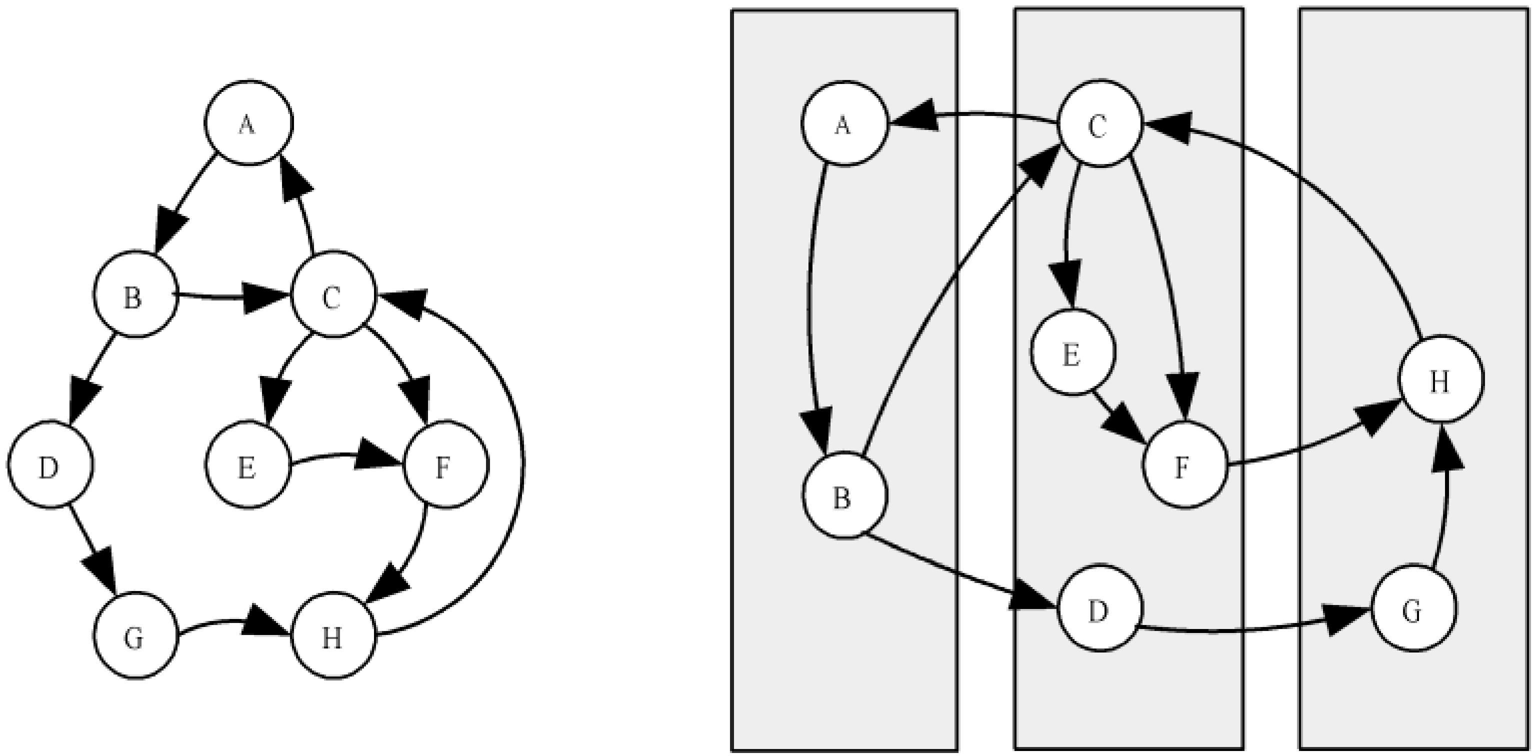
Agglomeration Checklist

- Locality of parallel algorithm has increased
- Replicated computations take less time than communications they replace
- Data replication doesn't affect scalability
- Agglomerated tasks have similar computational and communications costs
- Number of tasks increases with problem size
- Number of tasks suitable for likely target systems
- Tradeoff between agglomeration and code modifications costs is reasonable

Mapping

- Process of assigning tasks to processors
- Centralized multiprocessor: mapping done by operating system
- Distributed memory system: mapping done by user
- Conflicting goals of mapping
  - Maximize processor utilization
  - Minimize interprocessor communication

Mapping Example



(a) (b)

Optimal Mapping

- Finding optimal mapping is NP-hard
- Must rely on heuristics

Mapping Decision Tree

- Static number of tasks
  - Structured communication
    - Constant computation time per task
      - Agglomerate tasks to minimize comm
      - Create one task per processor
    - Variable computation time per task
      - Cyclically map tasks to processors
  - Unstructured communication
      - Use a static load balancing algorithm
- Dynamic number of tasks

Mapping Strategy

- Static number of tasks
- Dynamic number of tasks
  - Frequent communications between tasks
    - Use a dynamic load balancing algorithm
  - Many short-lived tasks
    - Use a run-time task-scheduling algorithm

Mapping Checklist

- Considered designs based on one task per processor and multiple tasks per processor
- Evaluated static and dynamic task allocation
- If dynamic task allocation chosen, task allocator is not a bottleneck to performance
- If static task allocation chosen, ratio of tasks to processors is at least 10:1

2

Map-Reduce Framework

MapReduce Programming Model

- Inspired from map and reduce operations commonly used in functional programming languages like Lisp.

- Have multiple map tasks and reduce tasks

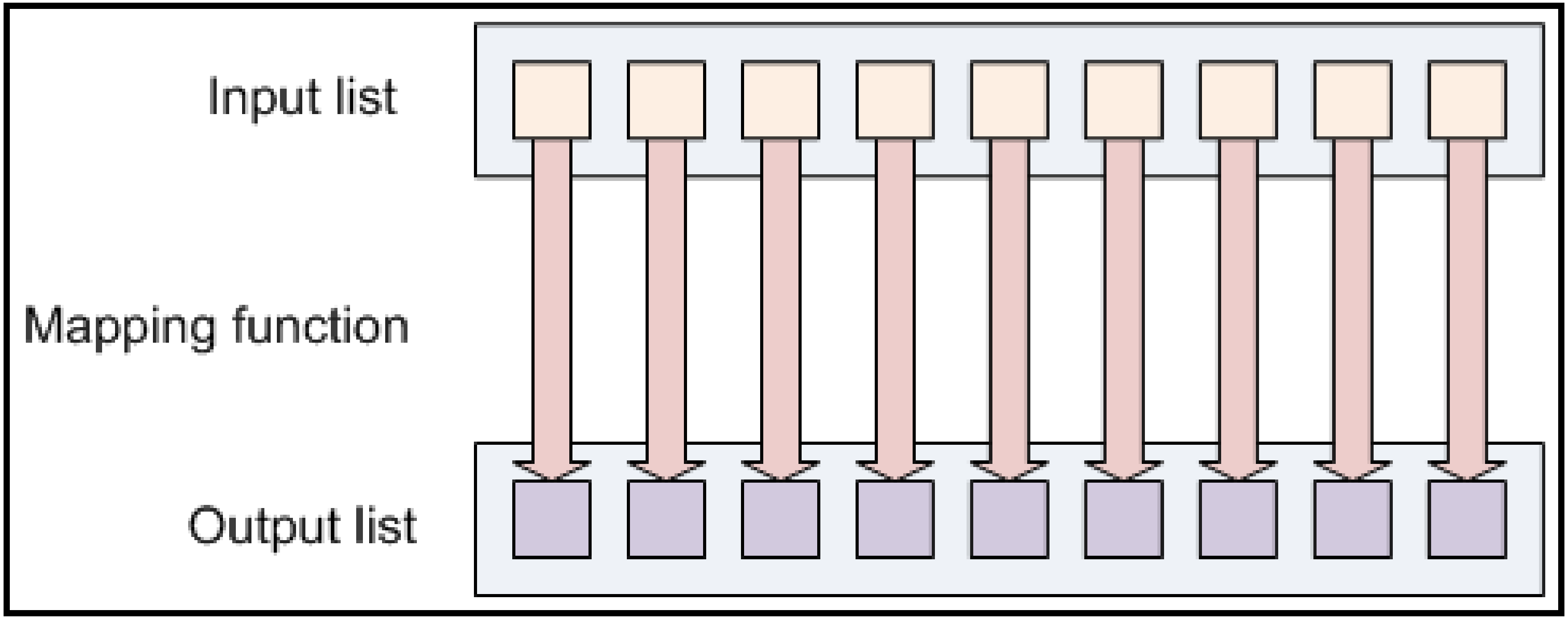
- Users implement interface of two primary methods:
  - Map: (key1, val1) → (key2, val2)
  - Reduce: (key2, [val2]) → [val3]

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
```



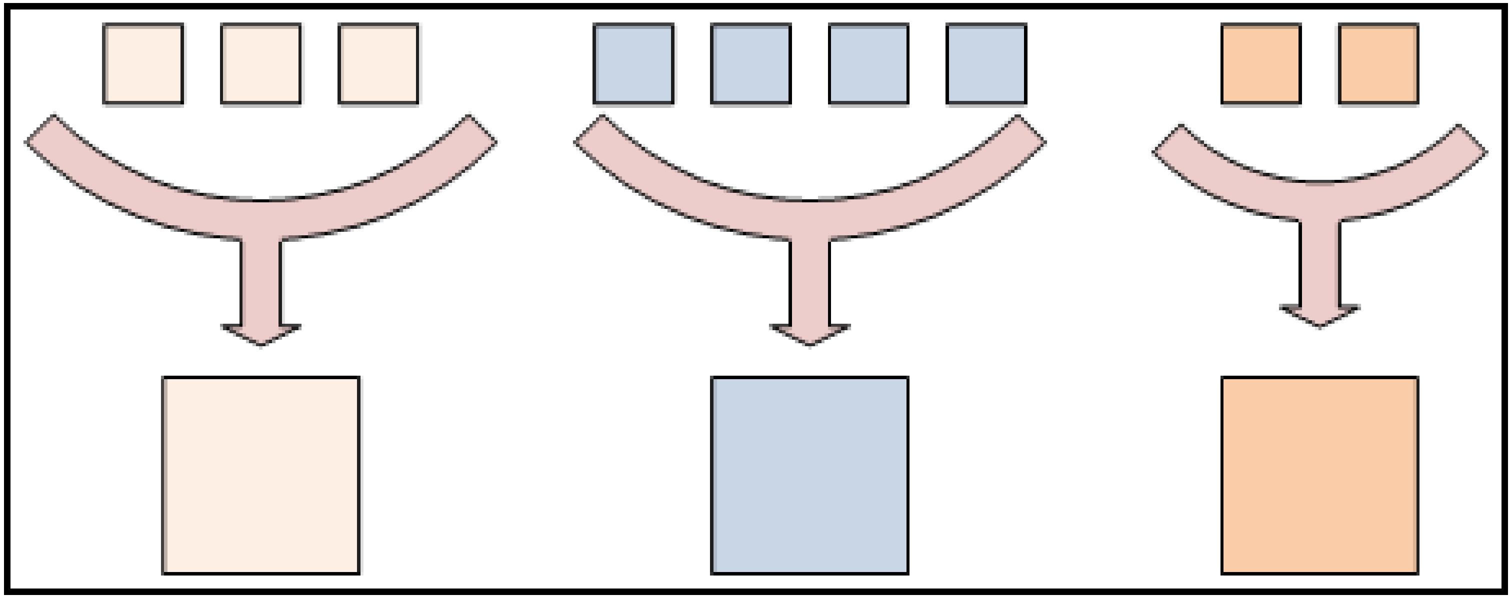
Processing of Reducer Tasks

- 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]) → [val3]

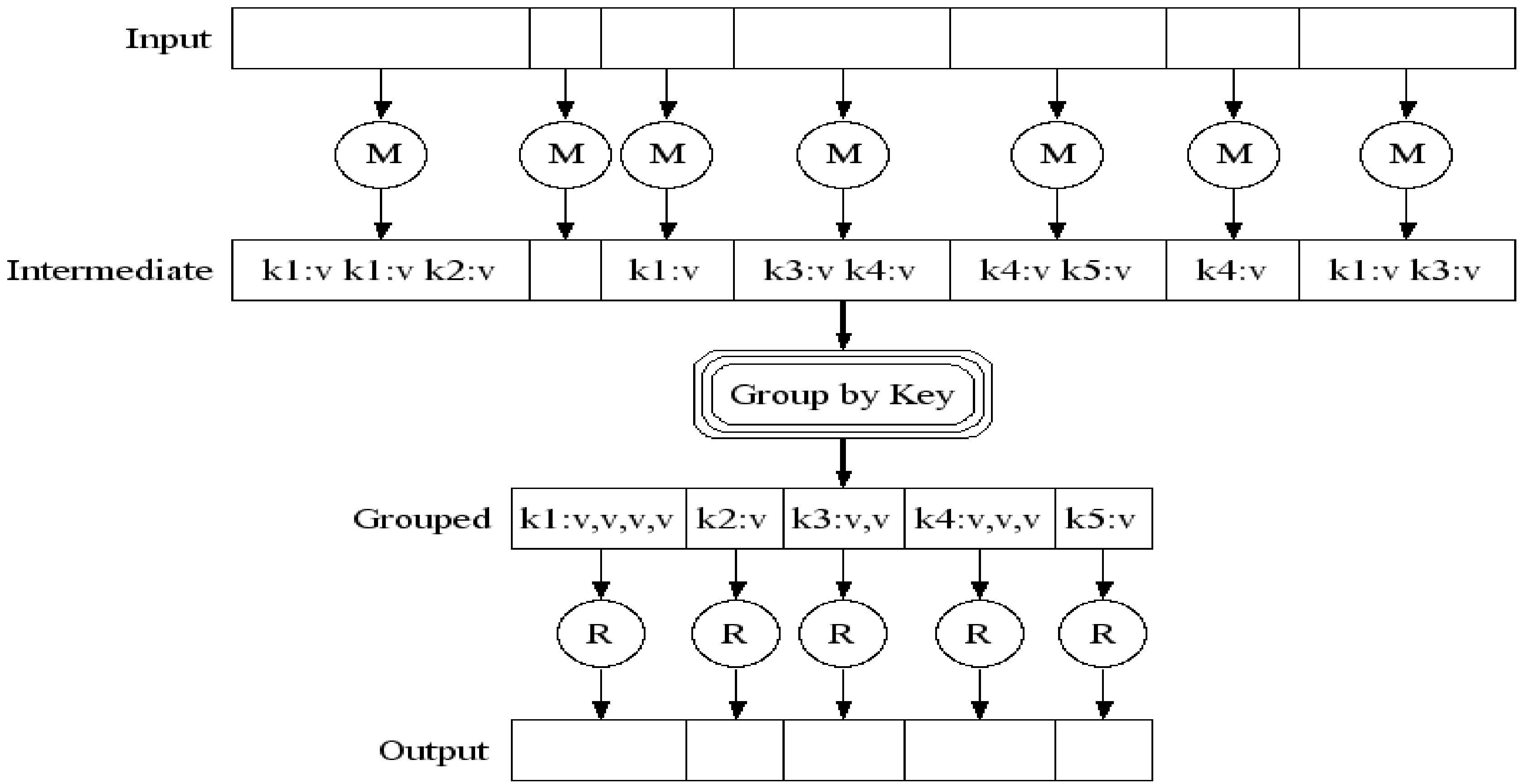
- Can be visualized as *aggregate* function (e.g., average) that is computed over all the rows with the same group-by attribute.

Reduce

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



Put Map and Reduce Tasks Together



Example: Wordcount (1)

Map

// assume input is a
// set of text files
// k is a line offset
// v is the line for that offset

let map(k, v) =
foreach word in v:
emit(word, 1)

Reduce

// k is a word
// vals is a list of 1s

let reduce(k, vals) =
emit(k, vals.length())

Example: Wordcount (2) Input/Output for a Map-Reduce Job

File1.txt

California is a great place

File2.txt

Los_Angeles is the biggest city in California

California 2
great 1
place 1
Los_Angeles 1
biggest 1
city 1
is 2
a 1
in 1

Example: Wordcount (3)
Map

Mapper

California is a great place

California

is

a

great

place

Key	Value
California	1
is	1
a	1
great	1
place	1

Example: Wordcount (4)
Map

Mapper

Key	Value
California	1
is	1
a	1
great	1
place	1

Key	Value
Los_Angeles	1
is	1
the	1
biggest	1
city	1
in	1
California	1

Example: Wordcount (5)
Map→Reduce

Mapper to Reducer

Key	Value
California	1
is	1
a	1
great	1
place	1

Key	Value
Los_Angeles	1
is	1
the	1
biggest	1
city	1
in	1
California	1

Example: Wordcount (6)
Input to Reduce

Key	Values
California	{1,1}
Los_Angeles	1

Key	Values
a	1
is	{1,1}
the	1
biggest	1
city	1
in	1
great	1
place	1

Example: Wordcount (7)
Reduce Output

Key	Values
California	2
Los_Angeles	1

Key	Values
a	1
is	2
the	1
biggest	1
city	1
in	1
great	1
place	1

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.

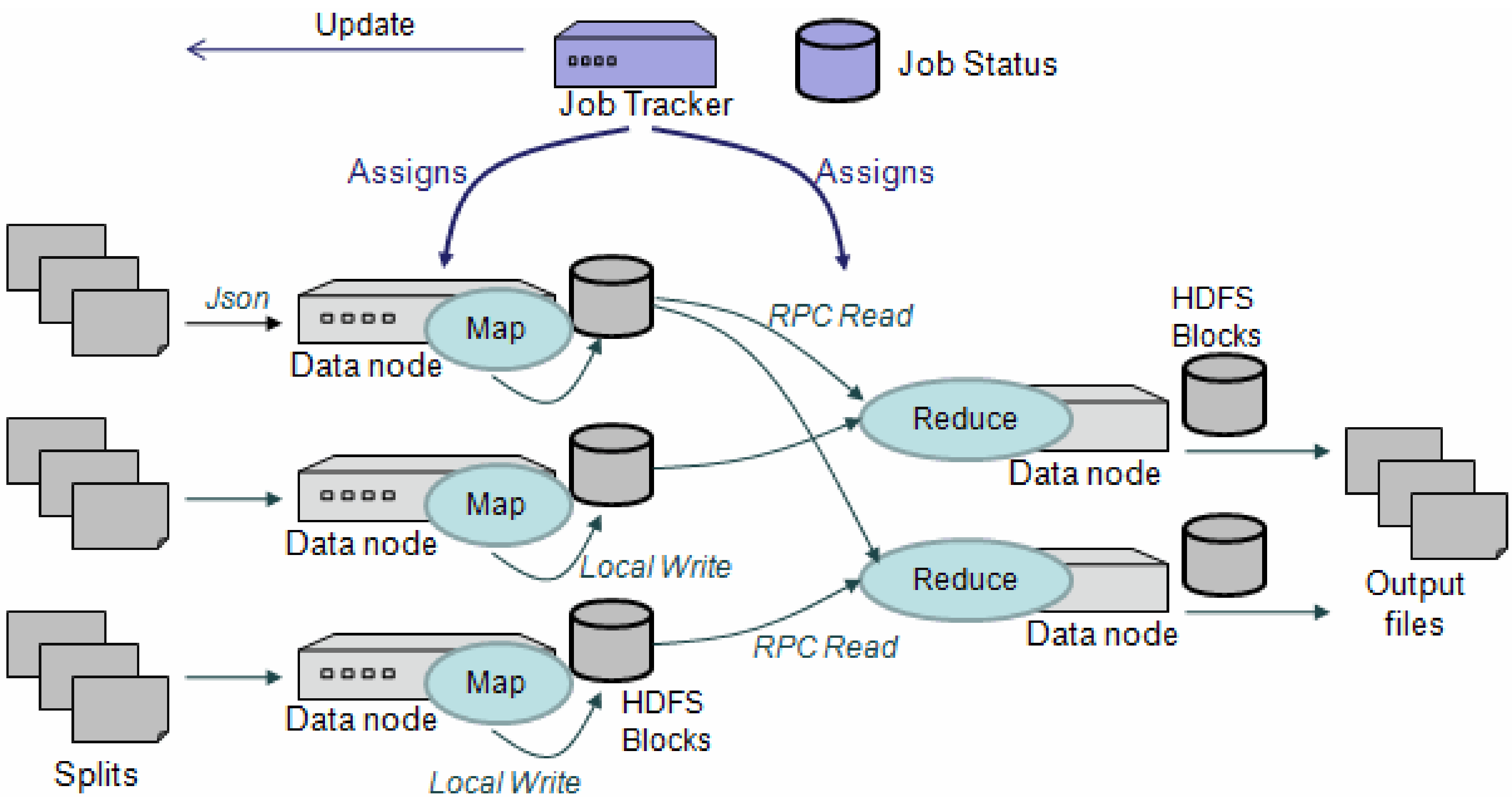
↓

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

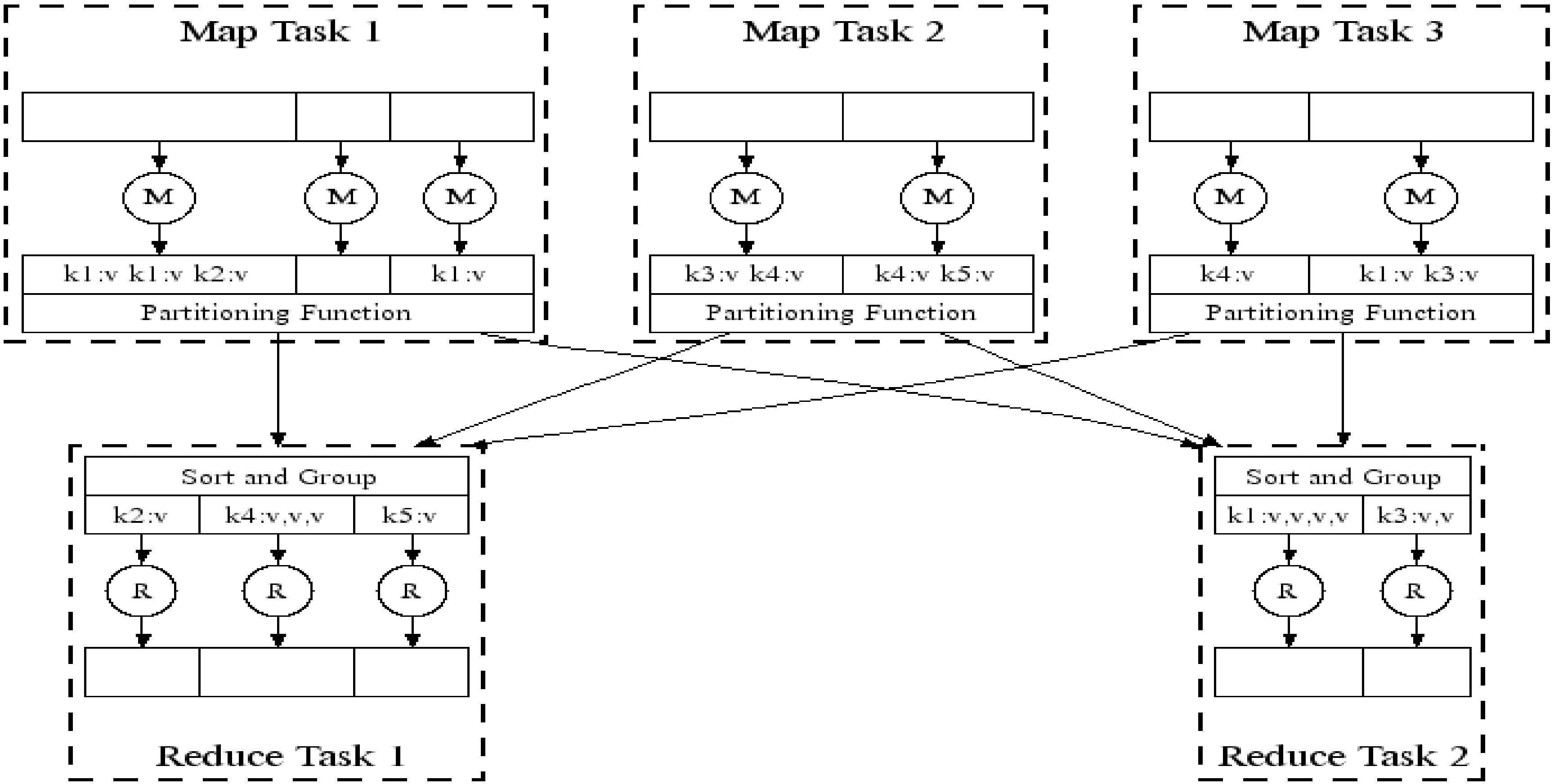
↓

Reducers output the result on stable storage.

Execute MapReduce on a cluster of machines with HDFS



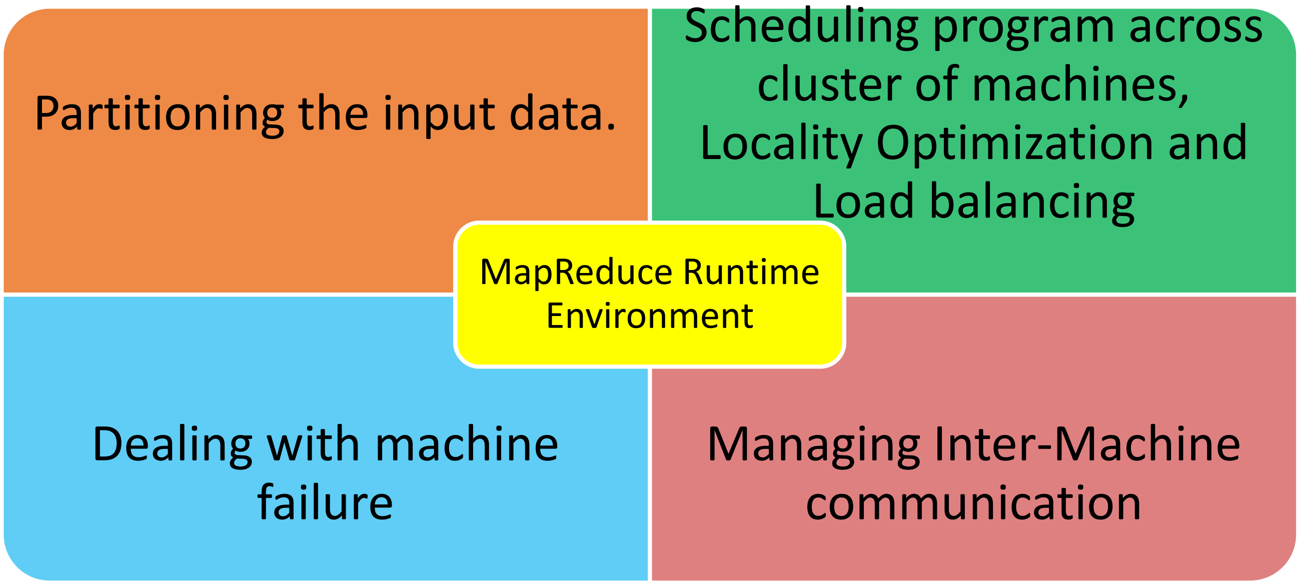
MapReduce in Parallel: Example

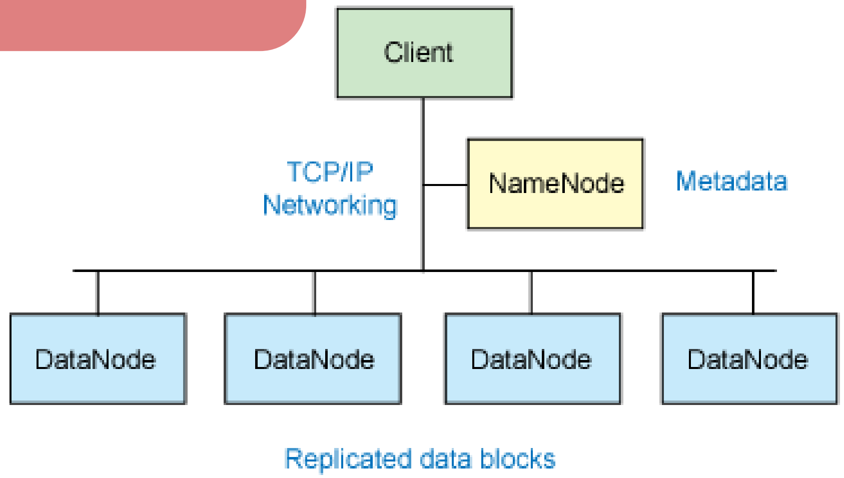


MapReduce: Execution Details

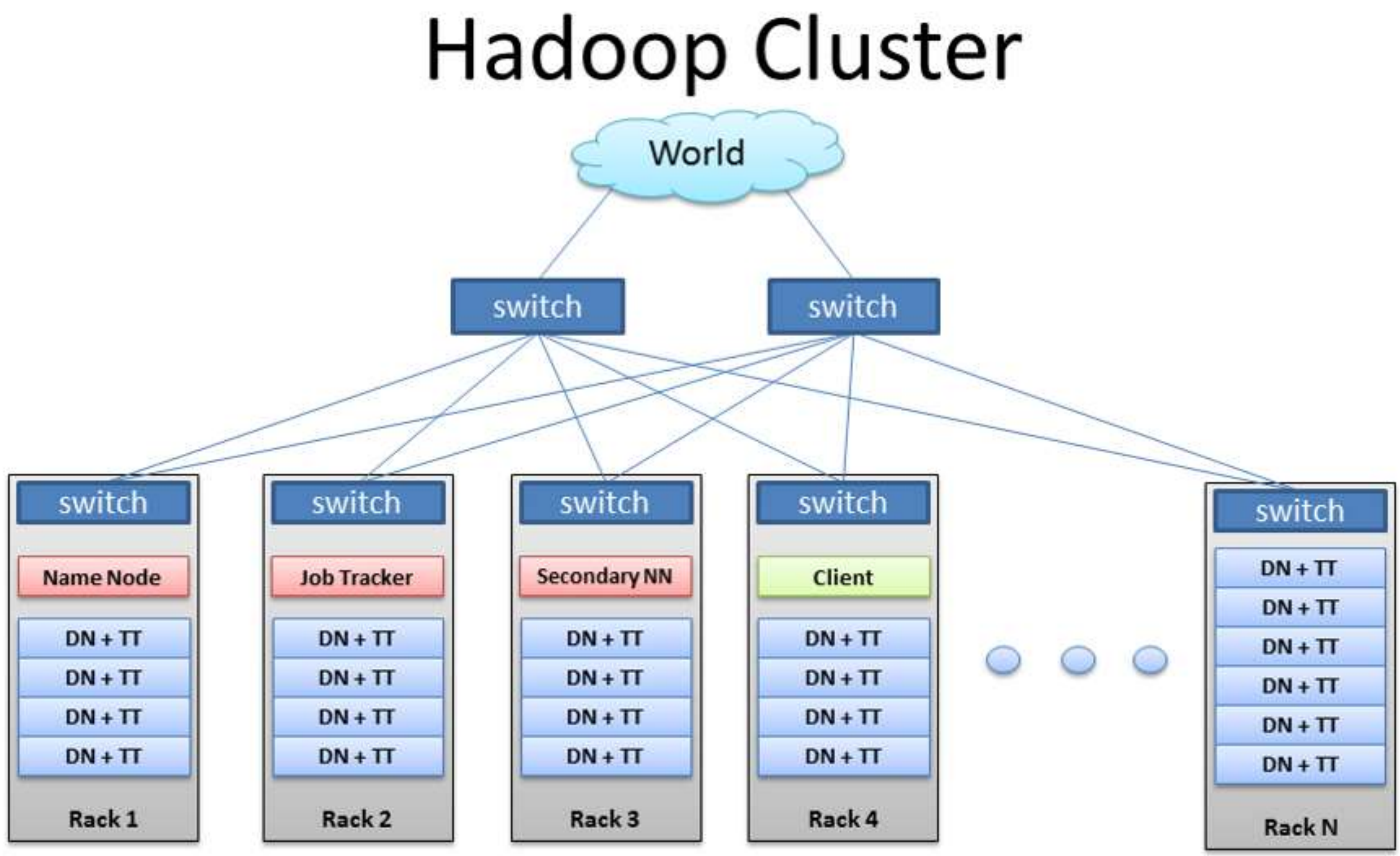
- Input reader
  - Divide input into splits, assign each split to a Map task
- Map task
  - Apply the Map function to each record in the split
  - Each Map function returns a list of (key, value) pairs
- Shuffle/Partition and Sort
  - Shuffle distributes sorting & aggregation to many reducers
  - All 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: Runtime Environment

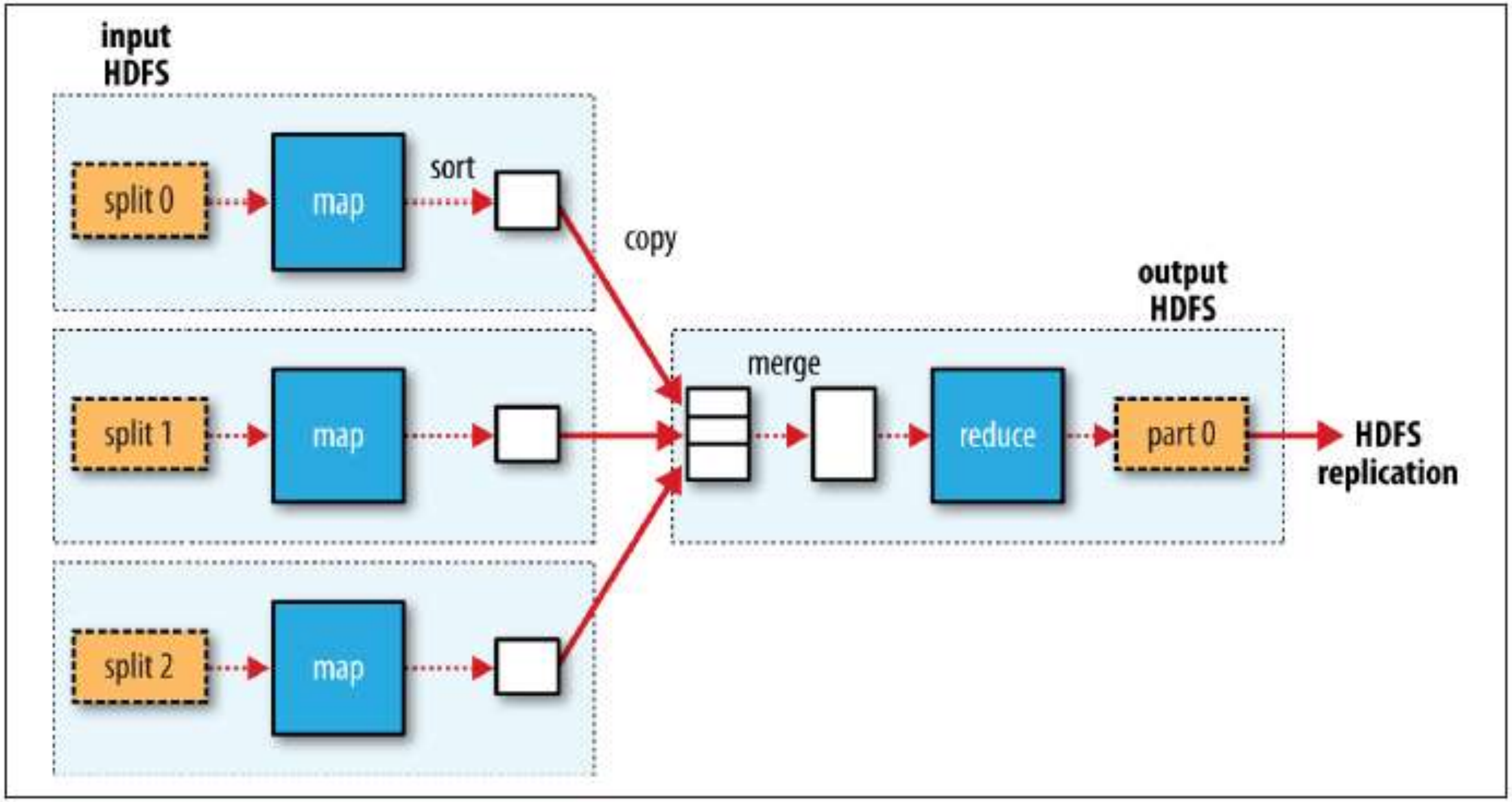




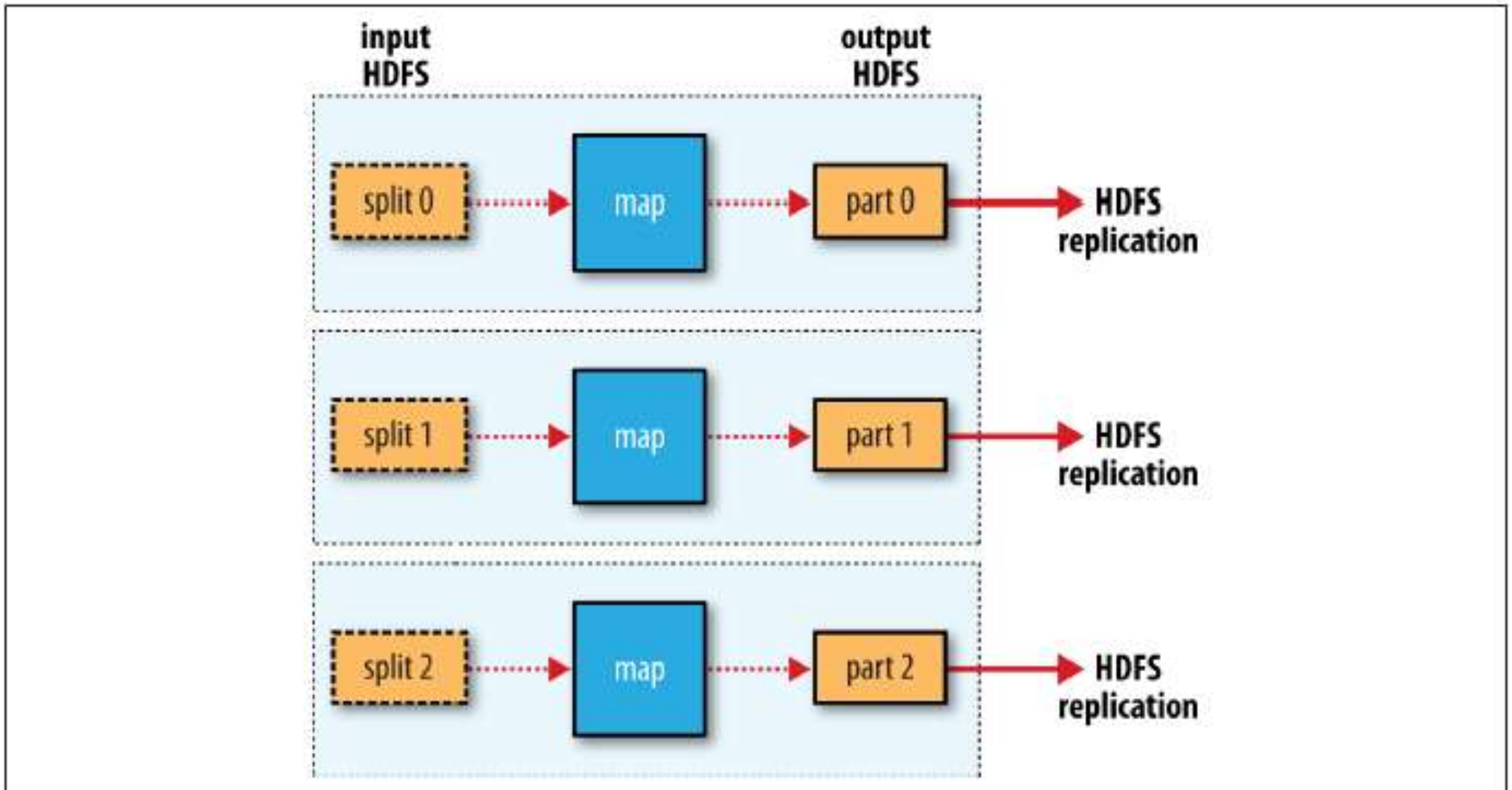
Hadoop Cluster with MapReduce



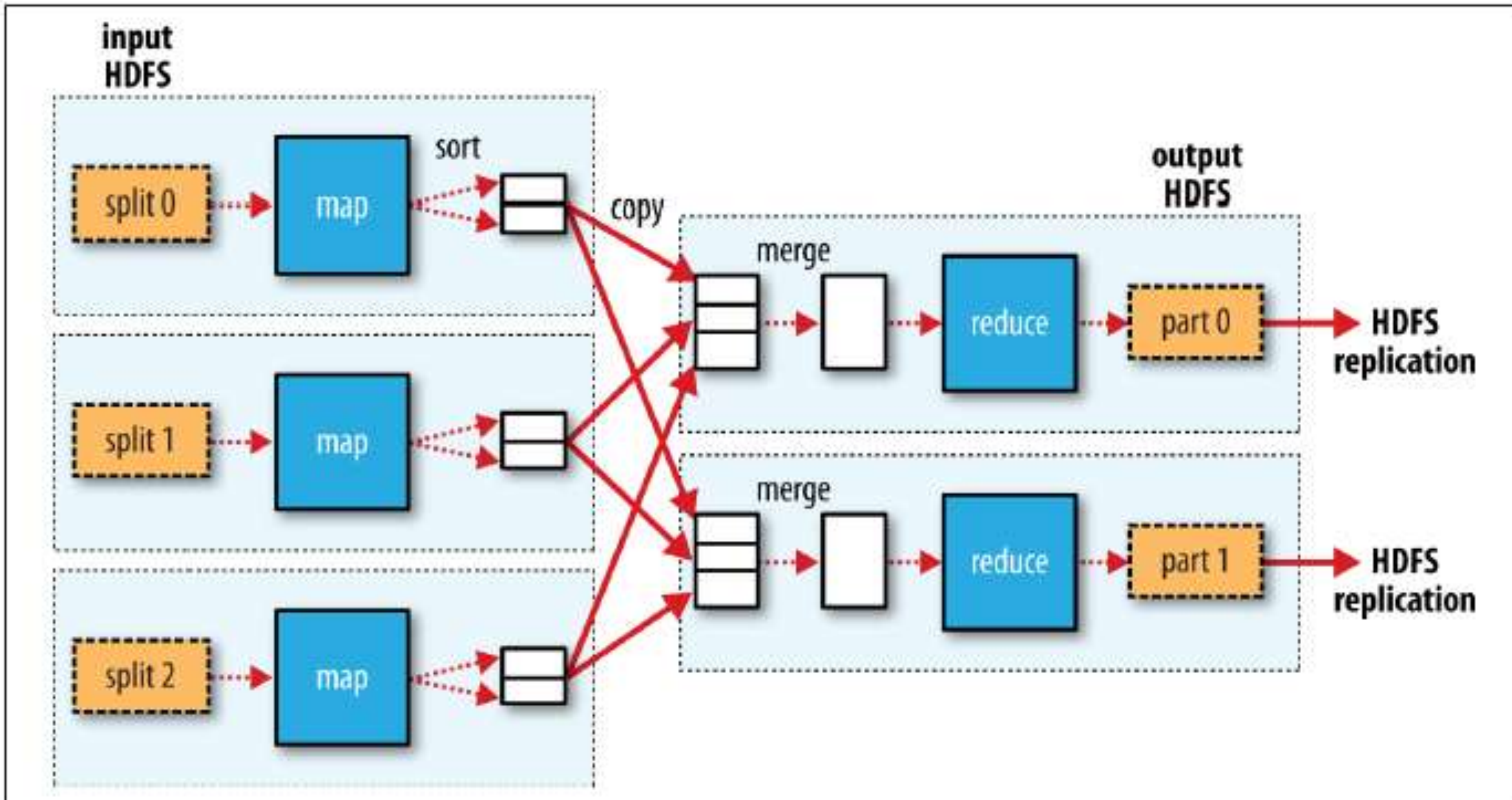
MapReduce (Single Reduce Task)



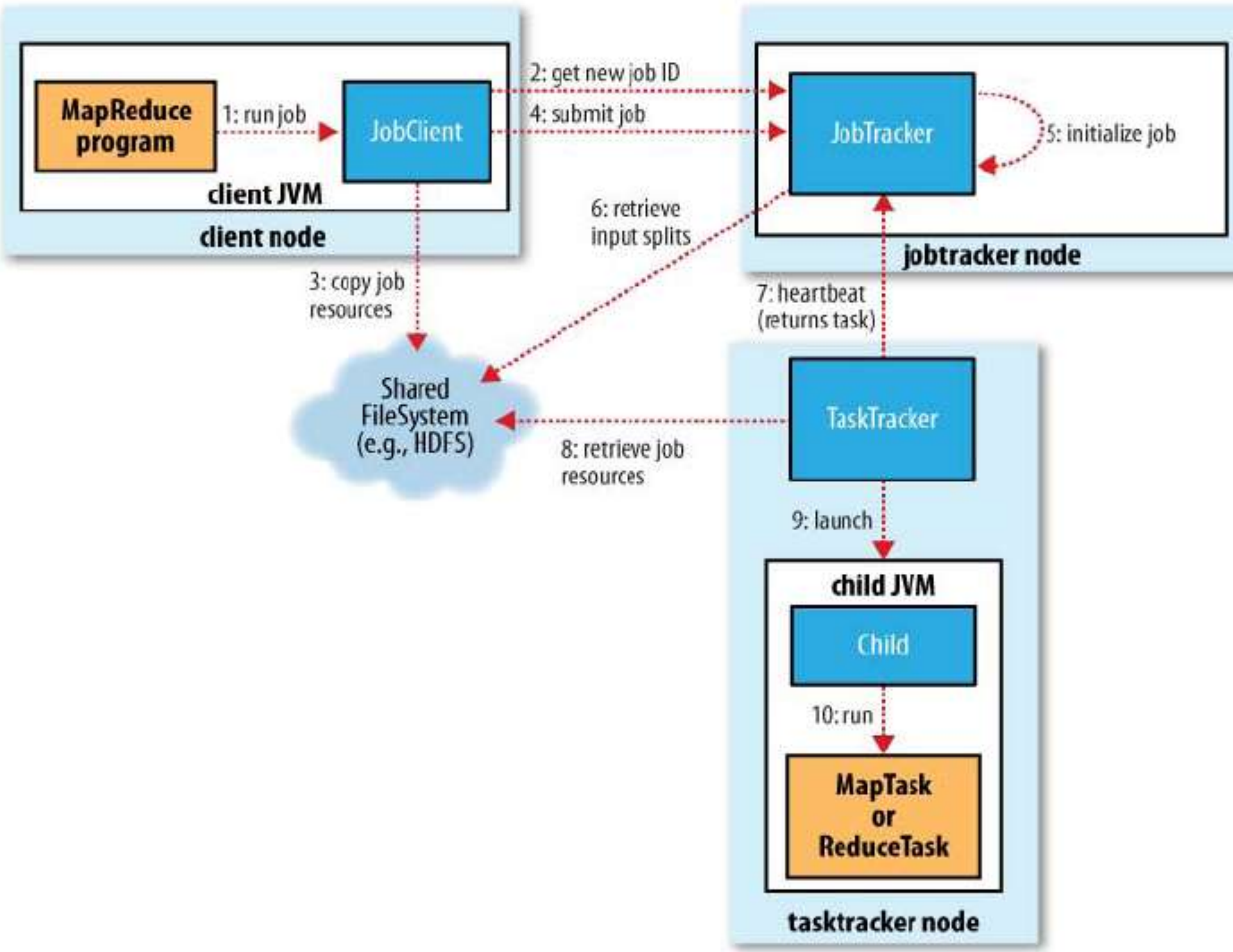
MapReduce (No Reduce Task)



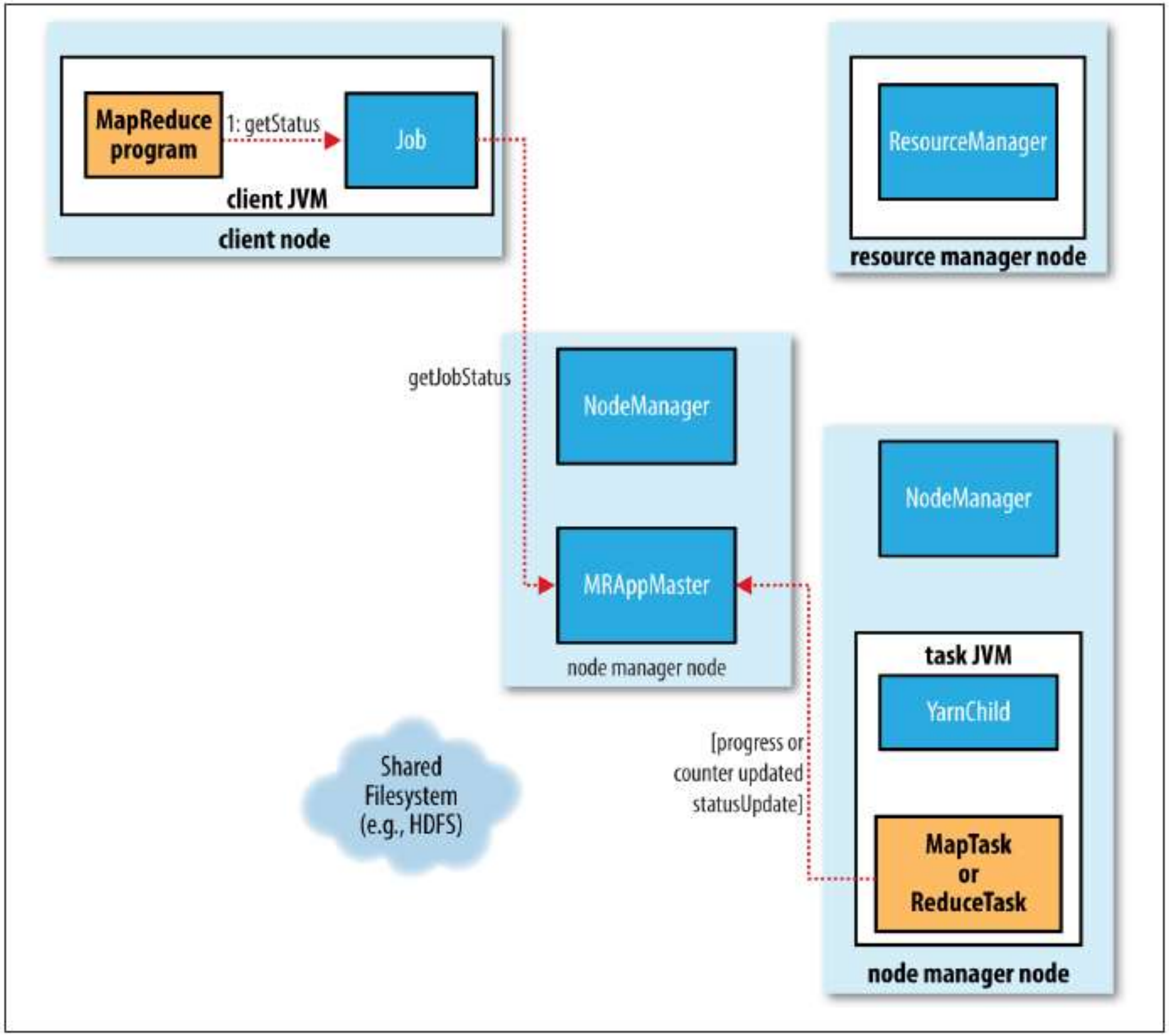
MapReduce (Multiple Reduce Tasks)



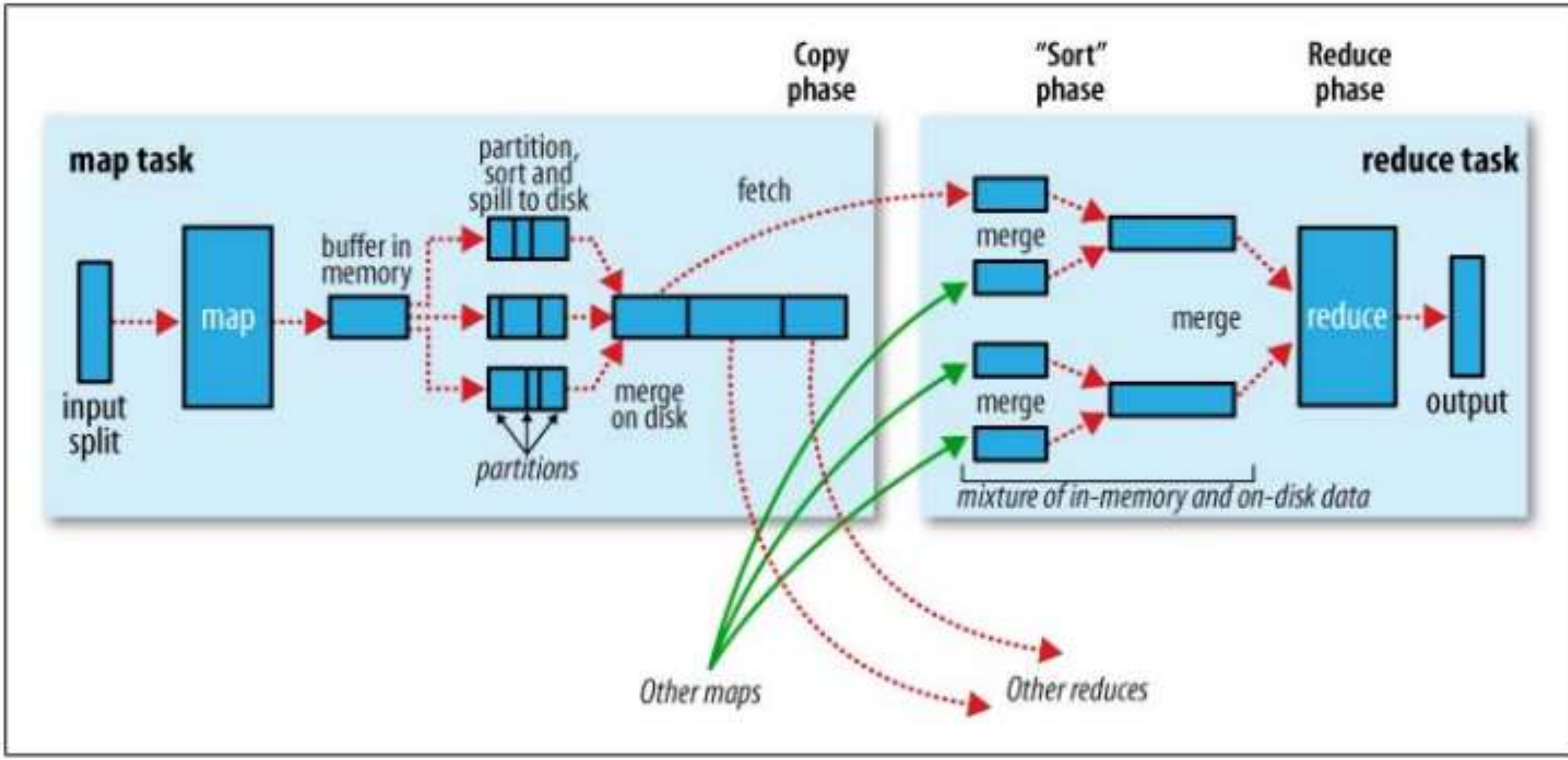
High Level of Map-Reduce in Hadoop



Status Update



MapReduce with data shuffling & sorting



Lifecycle of a MapReduce Job

```
File  Edit  Options  Buffers  Tools  Java  Help

public class WordCount {

  public static class Map extends MapReduceBase implements
               Mapper<LongWritable, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(LongWritable key, Text value, OutputCollector<Text, IntWritable>
                 output, Reporter reporter) throws IOException {
      String line = value.toString();
      StringTokenizer tokenizer = new StringTokenizer(line);
      while (tokenizer.hasMoreTokens()) {
        word.set(tokenizer.nextToken());
        output.collect(word, one);
  }}}

  public static class Reduce extends MapReduceBase implements
               Reducer<Text, IntWritable, Text, IntWritable> {
    public void reduce(Text key, Iterator<IntWritable> values, OutputCollector<Text,
                   IntWritable> output, Reporter reporter) throws IOException {
      int sum = 0;
      while (values.hasNext()) { sum += values.next().get(); }
      output.collect(key, new IntWritable(sum));
  }}

  public static void main(String[] args) throws Exception {
    JobConf conf = new JobConf(WordCount.class);
    conf.setJobName("wordcount");
    conf.setOutputKeyClass(Text.class);
    conf.setOutputValueClass(IntWritable.class);
    conf.setMapperClass(Map.class);
    conf.setCombinerClass(Reduce.class);
    conf.setReducerClass(Reduce.class);
    conf.setInputFormat(TextInputFormat.class);
    conf.setOutputFormat(TextOutputFormat.class);
    FileInputFormat.setInputPaths(conf, new Path(args[0]));
    FileOutputFormat.setOutputPath(conf, new Path(args[1]));

    JobClient.runJob(conf);
  }}
--:---  mapreduce.java    All L9     (Java/l Abbrev)----------------------------------------
Wrote /home/shivnath/Desktop/mapreduce.java
```

Map function

Reduce function

Run this program as a MapReduce job

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*
- 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

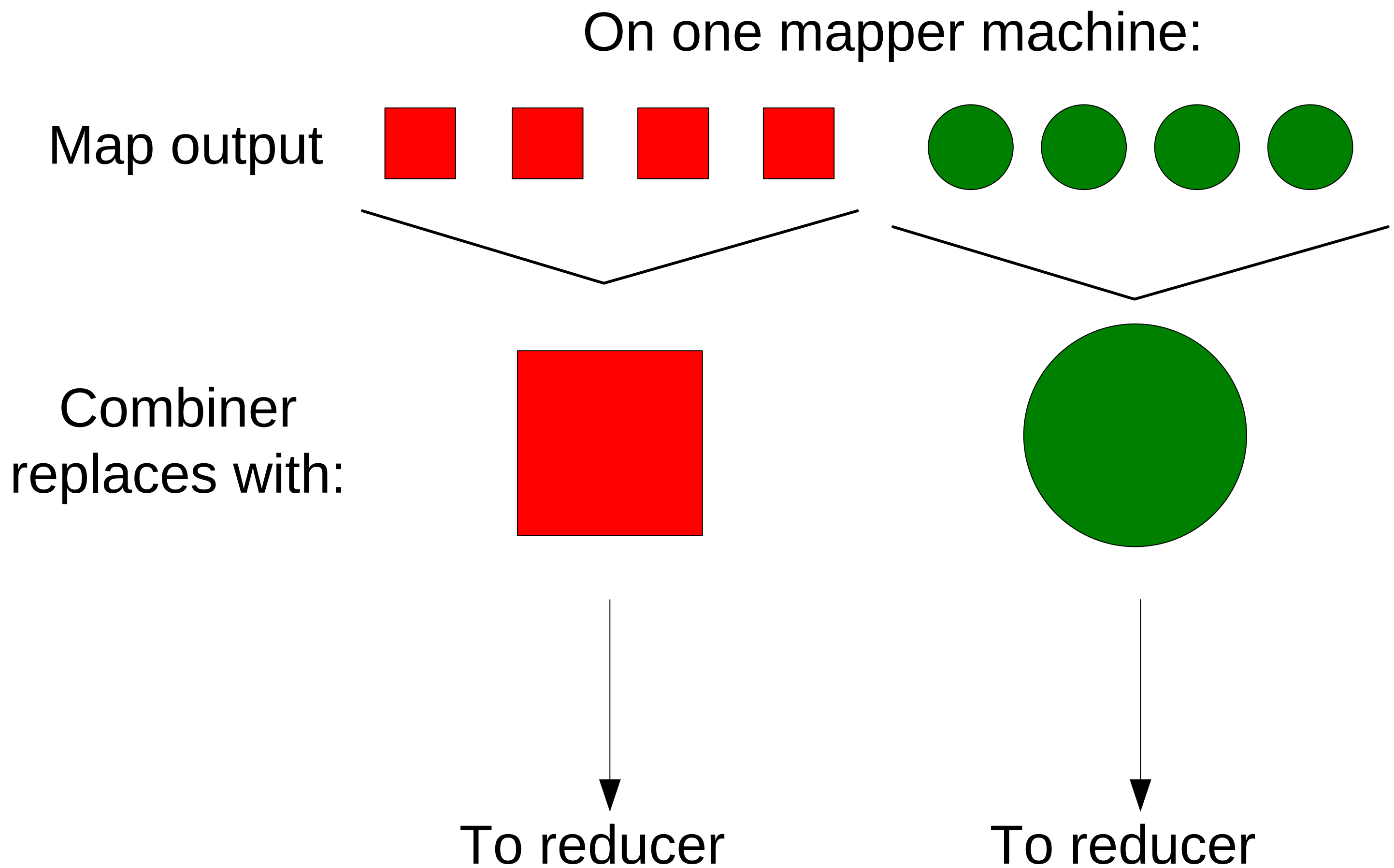
MapReduce: Miscellaneous Refinements

- 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

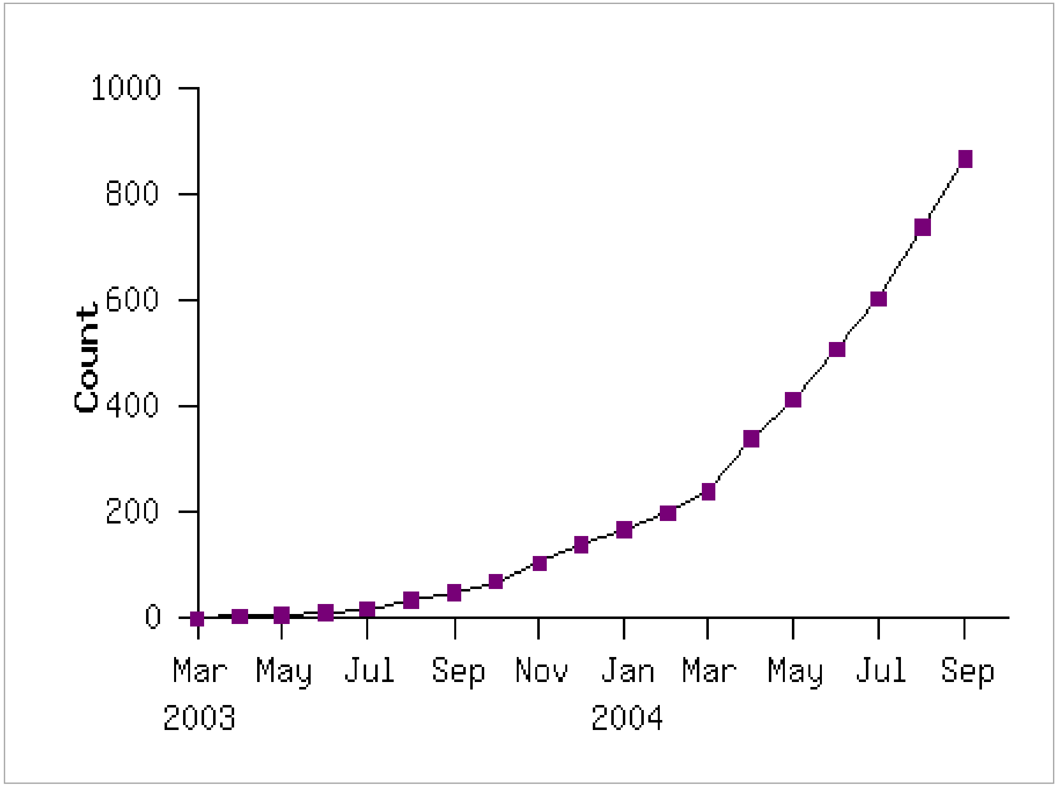
Combiner, graphically



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



3 Applications Using Map-Reduce

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)

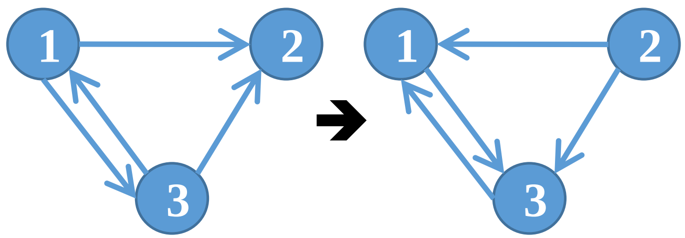
Solution:

- 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 internal document id, e.g., a unique integer

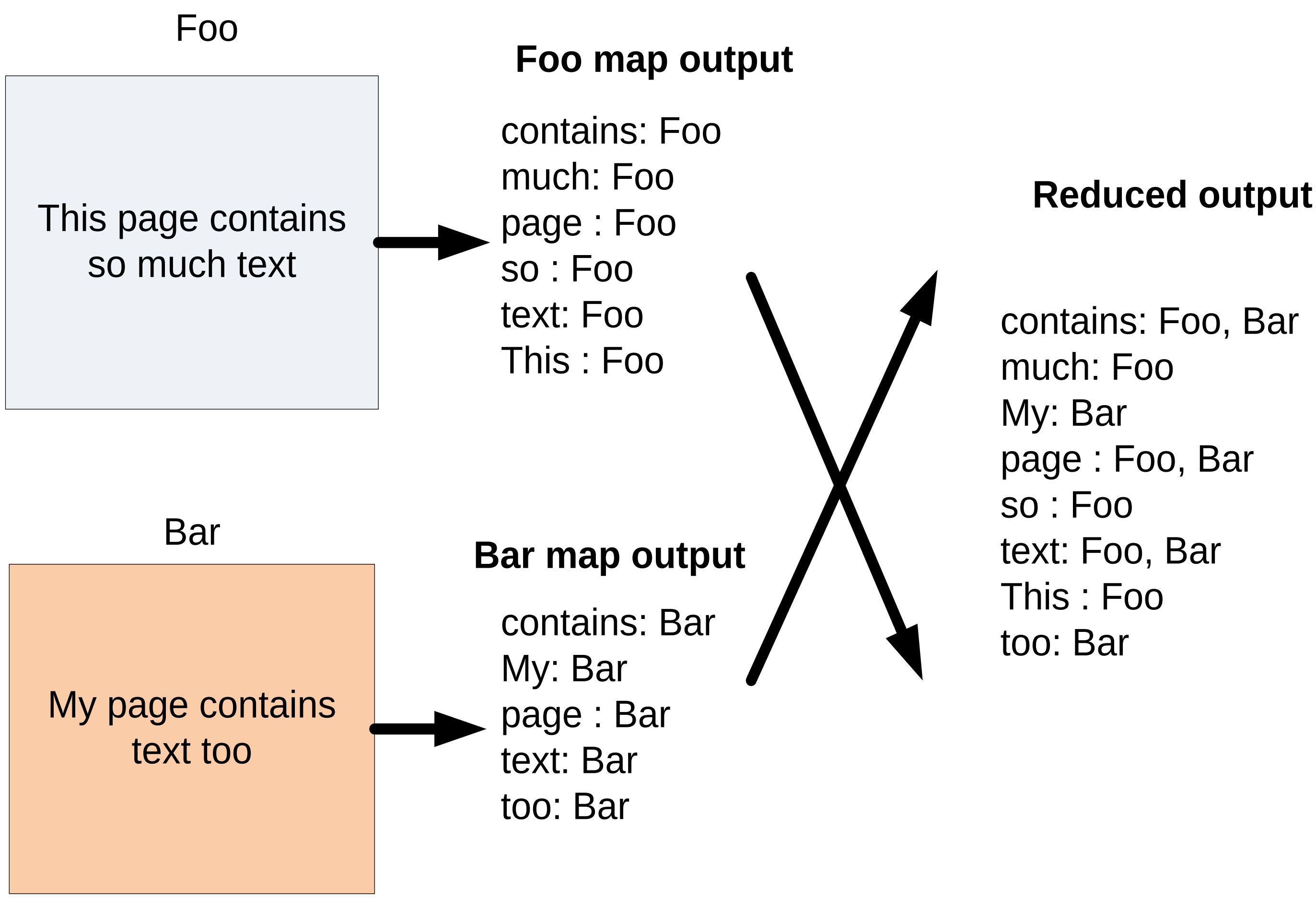
- Not an external document id such as a url

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



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

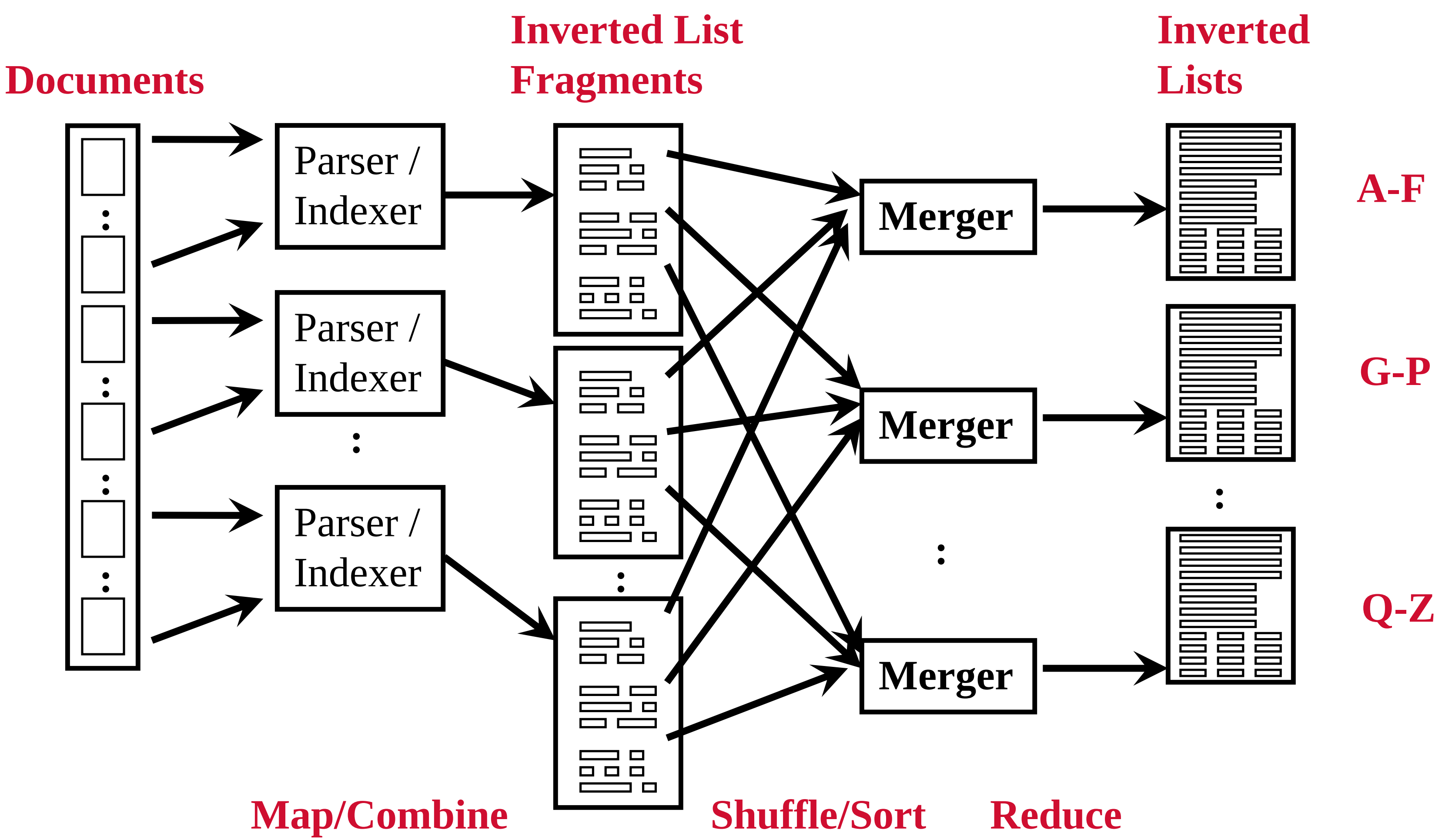
  Sort by t

  Combine: $(t_1\ [ilist_{1,2}\ ilist_{1,3}\ ilist_{1,1}\ \ldots]) \rightarrow (t_1, ilist_{1,27})$
  - Each output inverted list covers a <u>sequence of documents</u>
- Shuffle by t
- Sort by t

  $(t_4, ilist_{4,1})\ (t_5, ilist_{5,3})\ \ldots \rightarrow (t_4, ilist_{4,2})\ (t_4, ilist_{4,4})\ (t_4, ilist_{4,1})$ …
- Reduce: $(t_7, [ilist_{7,2}, ilist_{3,1}, ilist_{7,4}, \ldots]) \rightarrow (t_7, ilist_{final})$

$ilist_{i,j}$: the j'th inverted list fragment for term i

Using MapReduce to Construct Indexes



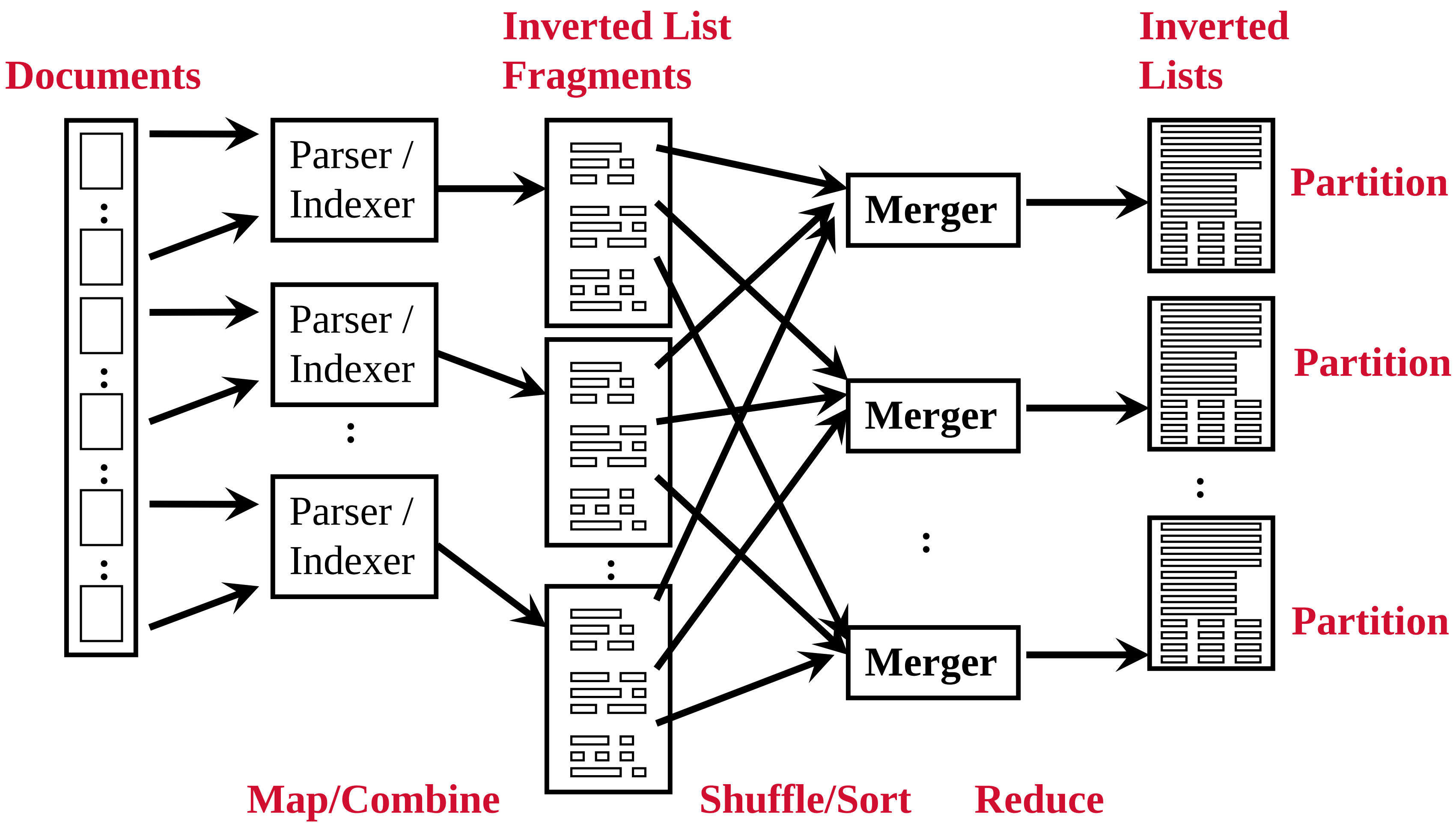
Construct Partitioned Indexes

- Useful when the document list of a term does not fit memory
- Map: $(\text{docid}_1, \text{content}_1) \rightarrow ([p, t_1], \text{ilist}_{1,1})$
- Combine to sort and group values

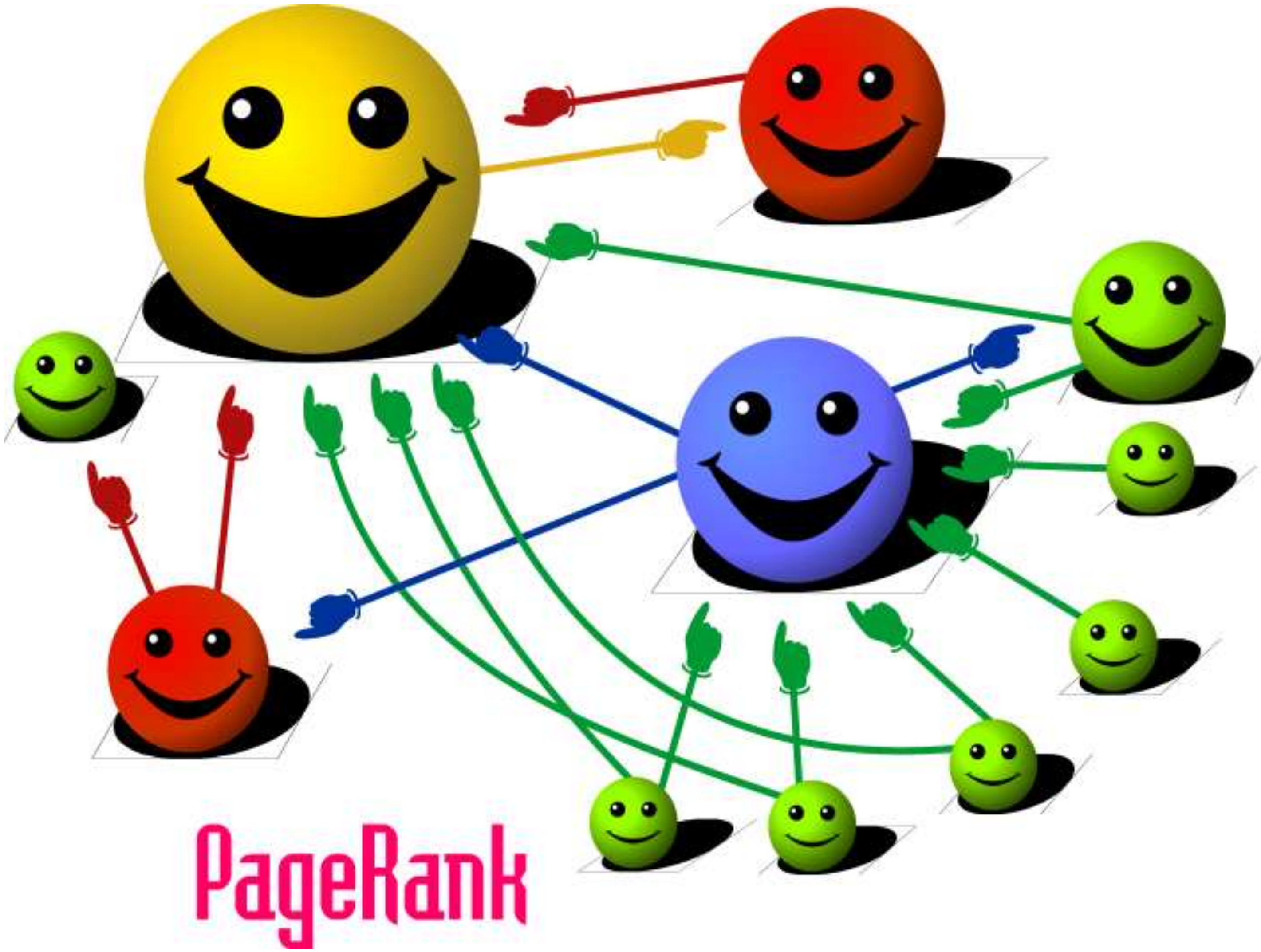
  $([p, t_1]\ [\text{ilist}_{1,2}\ \text{ilist}_{1,3}\ \text{ilist}_{1,1}\ \ldots]) \rightarrow ([p, t_1], \text{ilist}_{1,27})$
- Shuffle by p
- Sort values by [p, t]
- Reduce: $([p, t_7], [\text{ilist}_{7,2}, \text{ilist}_{7,1}, \text{ilist}_{7,4}, \ldots]) \rightarrow ([p, t_7], \text{ilist}_{final})$

p: partition (shard) id

Generate Partitioned Index



MapReduce Use Case 6: PageRank

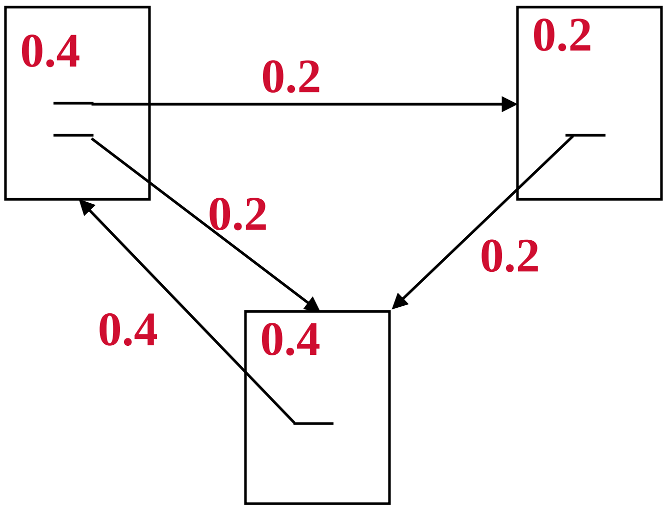


PageRank

- 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

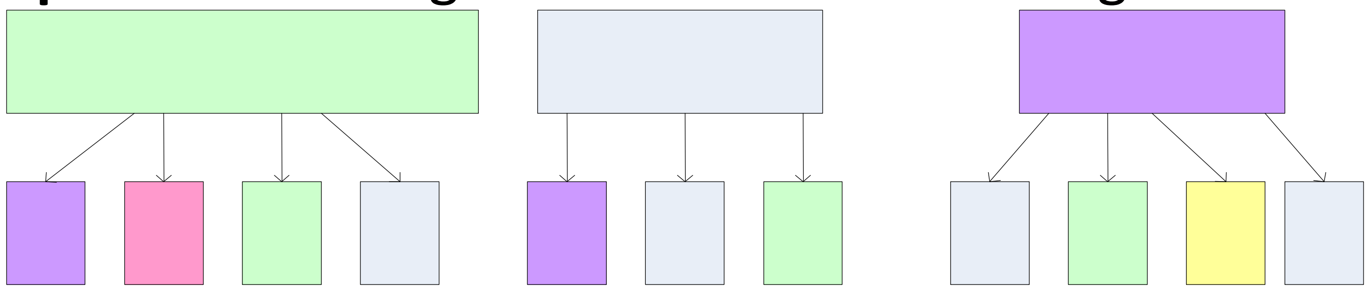
Each page distributes *PageRank* "credit" to all pages it points to.

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

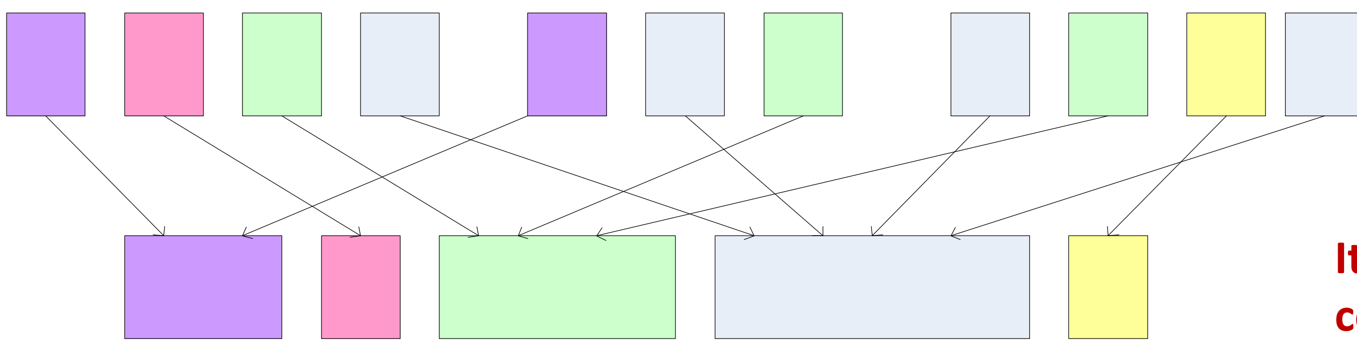
- Effects at each iteration is local. i+1th iteration depends only on ith iteration
- At iteration i, PageRank for individual nodes can be computed independently

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

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}, ..]) ..

Conclusion

- 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

References

- J. Dean and S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters." *In Proc. of OSDI 2004*.
- S. Ghemawat, H. Gobioff, and S.-T. Leung. "The Google File System." *In Proc. of SOSP 2003.*
- 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. 2013.
- http://developer.yahoo.com/hadoop/tutorial/module4.html
- J. Lin and C. Dyer. *Data-Intensive Text Processing with MapReduce*, Book Draft. February 7, 2010.

Thank you!