تحلیلها و سیستمهای داده های حجیم

مدرس: سید حسین خواسته

دانشکده مهندسی کامپیوتر

تهران – خیابان شریعتی – نرسیده به پل سید خندان
صندوق پستی ۱۶۳۱۵– ۱۳۵۵ کدپستی ۱۶۳۱۴ تلفن: ۸۸۴۶۲۴۹۴

دانشگاه صنعتی خواجه نصیرالدین طوسی

K.N Toosi University Of Technology

تاسیس ۱۳۰۷

Introduction to
Big Data Analytics Platforms

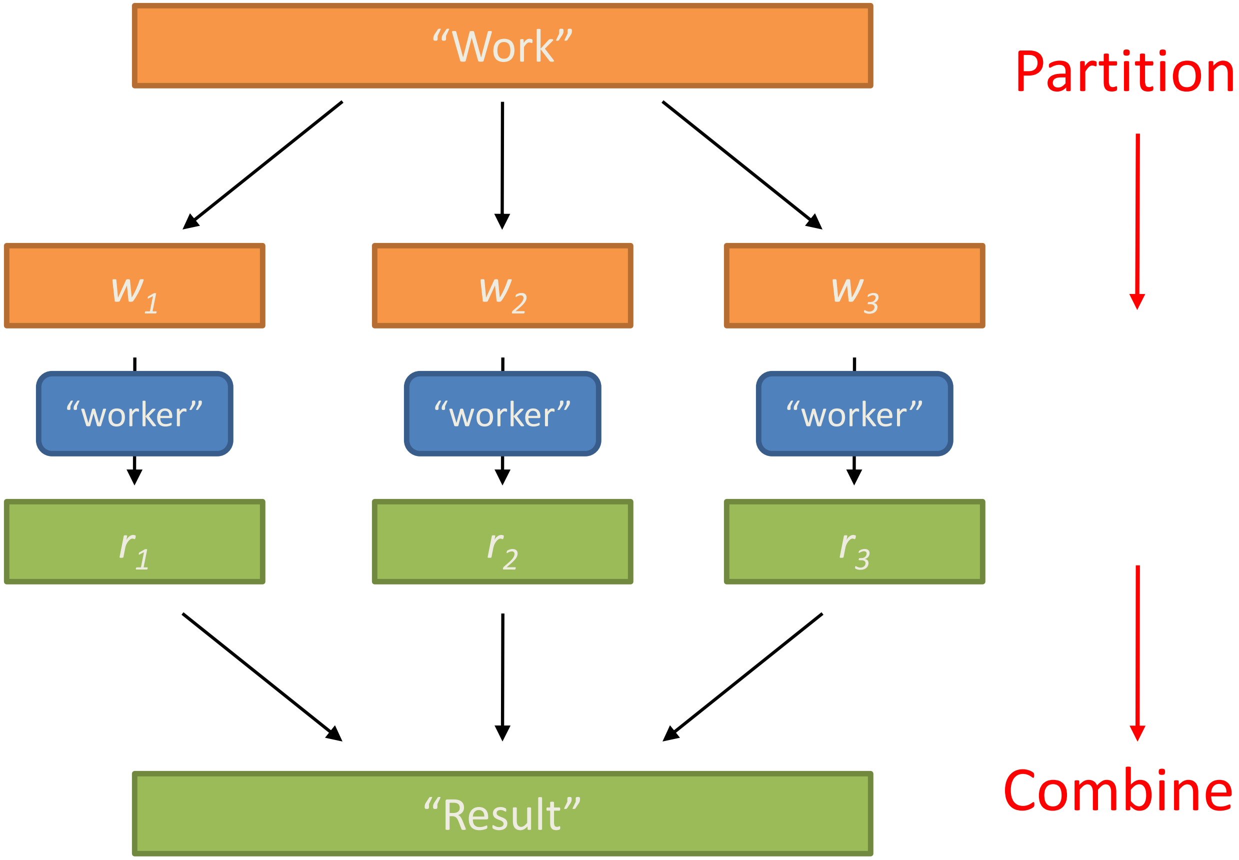
Typical Large-Data Problem

- Iterate over a large number of records
- Extract something of interest from each
- Shuffle and sort intermediate results
- Aggregate intermediate results
- Generate final output

- The problem:
  - Diverse input format (data diversity & heterogeneity)
  - Large Scale: Terabytes, Petabytes
  - Parallelization

How to leverage a number of cheap off-the-shelf computers?

Divide and Conquer



Parallelization Challenges

- How do we assign work units to workers?
- What if we have more work units than workers?
- What if workers need to share partial results?
- How do we aggregate partial results?
- How do we know all the workers have finished?
- What if workers die?

What is the common theme of all of these problems?

Common Theme?

- Parallelization problems arise from:
  - Communication between workers (e.g., to exchange state)
  - Access to shared resources (e.g., data)
- Thus, we need a synchronization mechanism

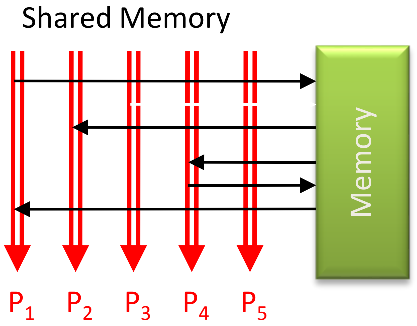
BANCO REAL
11455

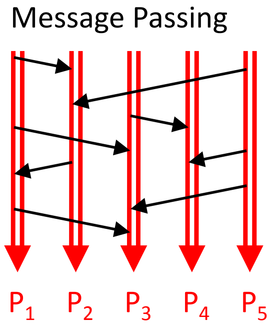
Managing Multiple Workers

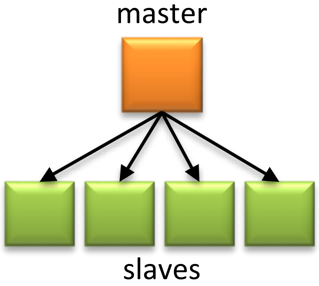
- Difficult because
  - We don’t know the order in which workers run
  - We don’t know when workers interrupt each other
  - We don’t know the order in which workers access shared data
- Thus, we need:
  - Semaphores (lock, unlock)
  - Conditional variables (wait, notify, broadcast)
  - Barriers
- Still, lots of problems:
  - Deadlock, livelock, race conditions...
  - Dining philosophers, sleeping barbers, ...
- Moral of the story: be careful!

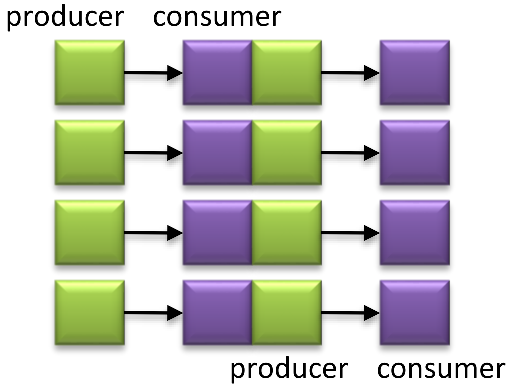
Current Tools

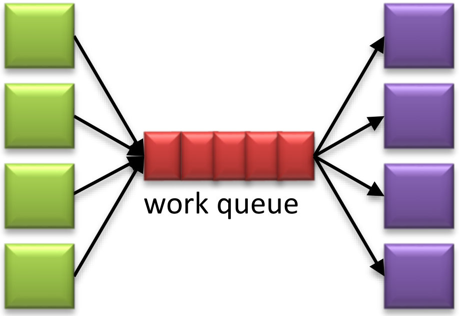
- Programming models
  - Shared memory (pthreads)
  - Message passing (MPI)
- Design Patterns
  - Master-slaves
  - Producer-consumer flows
  - Shared work queues











Concurrency Challenge!

- Concurrency is difficult to reason about
- Concurrency is even more difficult to reason about
  - At the scale of datacenters
  - In the presence of failures
  - In terms of multiple interacting services
- Not to mention debugging…
- The reality:
  - Lots of one-off solutions, custom code
  - Write you own dedicated library, then program with it
  - Burden on the programmer to explicitly manage everything

What’s the point?

- **It’s all about the right level of abstraction**
  - **The von Neumann architecture has served us well, but is no longer appropriate for the multi-core/cluster environment**
- Hide system-level details from the developers
  - No more race conditions, lock contention, etc.
- **Separating the *what* from *how***
  - Developer specifies the computation that needs to be performed
  - Execution framework (“runtime”) handles actual execution

The datacenter *is* the computer!

Image from http://wiki.apache.org/hadoop-data/attachments/HadoopPresentations/attachments/aw-apachecon-eu-2009.pdf

Apache Hadoop

- The Apache™ Hadoop® project develops open-source software for reliable, scalable, distributed computing.
- The Apache Hadoop software library is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models. It is designed to scale up from single servers to thousands of machines, each offering local computation and storage. Rather than rely on hardware to deliver high-availability, the library itself is designed to detect and handle failures at the application layer, so delivering a highly-available service on top of a cluster of computers, each of which may be prone to failures.
- The project includes these modules:
- **Hadoop Common**: The common utilities that support the other Hadoop modules.
- **Hadoop Distributed File System (HDFS™)**: A distributed file system that provides high-throughput access to application data.
- **Hadoop YARN**: A framework for job scheduling and cluster resource management.
- **Hadoop MapReduce**: A YARN-based system for parallel processing of large data sets.

Apache Hadoop

- **Ambari™**: A web-based tool for provisioning, managing, and monitoring Apache Hadoop clusters which includes support for Hadoop HDFS, Hadoop MapReduce, Hive, HCatalog, HBase, ZooKeeper, Oozie, Pig and Sqoop. Ambari also provides a dashboard for viewing cluster health such as heatmaps and ability to view MapReduce, Pig and Hive applications visually alongwith features to diagnose their performance characteristics in a user-friendly manner.
- **Avro™**: A data serialization system.
- **Cassandra™**: A scalable multi-master database with no single points of failure.
- **Chukwa™**: A data collection system for managing large distributed systems.
- **HBase™**: A scalable, distributed database that supports structured data storage for large tables.
- **Hive™**: A data warehouse infrastructure that provides data summarization and ad hoc querying.
- **Mahout™**: A Scalable machine learning and data mining library.
- **Pig™**: A high-level data-flow language and execution framework for parallel computation.
- **Spark™**: A fast and general compute engine for Hadoop data. Spark provides a simple and expressive programming model that supports a wide range of applications, including ETL, machine learning, stream processing, and graph computation.
- **Tez™**: A generalized data-flow programming framework, built on Hadoop YARN, which provides a powerful and flexible engine to execute an arbitrary DAG of tasks to process data for both batch and interactive use-cases. Tez is being adopted by Hive™, Pig™ and other frameworks in the Hadoop ecosystem, and also by other commercial software (e.g. ETL tools), to replace Hadoop™ MapReduce as the underlying execution engine.
- **ZooKeeper™**: A high-performance coordination service for distributed applications.

Apache Hadoop

- Scalable fault-tolerant distributed system for Big Data:
  - Data Storage
  - Data Processing
  - A virtual Big Data machine
  - Borrowed concepts/Ideas from Google; Open source under the Apache license

- Core Hadoop has two main systems:
  - **Hadoop/MapReduce**: distributed big data processing infrastructure (abstract/paradigm, fault-tolerant, schedule, execution)
  - **HDFS (Hadoop Distributed File System)**: fault-tolerant, high-bandwidth, high availability distributed storage

Hadoop History

- **Dec 2004 –** Google GFS paper published
- **July 2005 –** Nutch uses MapReduce
- **Feb 2006 –** Becomes Lucene subproject
- **Apr 2007 –** Yahoo! on 1000-node cluster
- **Jan 2008 –** An Apache Top Level Project
- **Jul 2008 –** A 4000 node test cluster
- **Sept 2008 –** Hive becomes a Hadoop subproject
- **Feb 2009 –** The Yahoo! Search Webmap is a Hadoop application that runs on more than 10,000 core Linux cluster and produces data that is now used in every Yahoo! Web search query.
- **June 2009 –** On June 10, 2009, Yahoo! made available the source code to the version of Hadoop it runs in production.
- **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 30 PB.

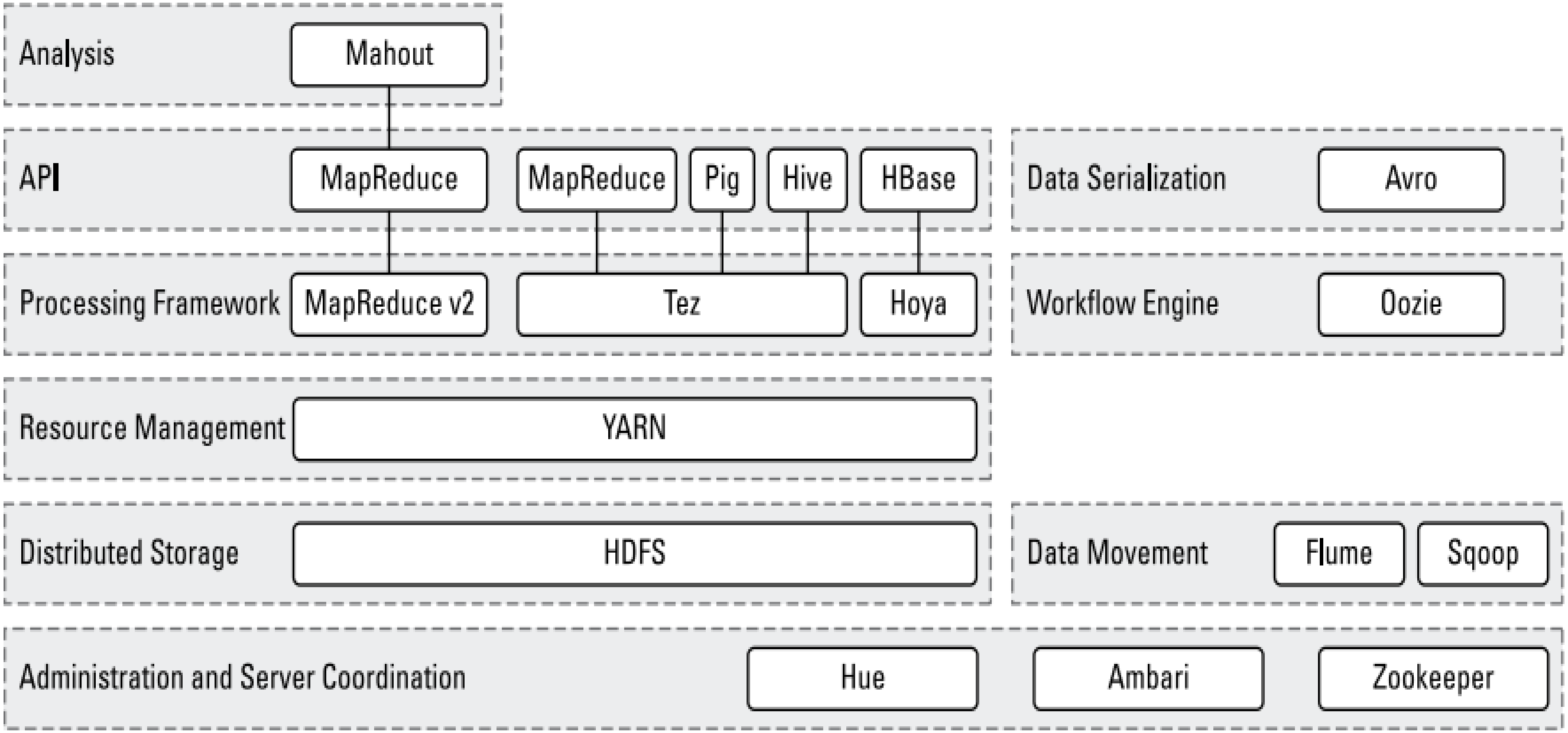
Hadoop History

- 27 dec 2011: Apache Hadoop release 1.0.0
- June 2012: Facebook claim “biggest Hadoop cluster”, totalling more than 100 PetaBytes in HDFS
- 2013: Yahoo runs Hadoop on 42,000 nodes, computing about 500,000 MapReduce jobs per day
- 15 oct 2013: Apache Hadoop release 2.2.0 (YARN)

Who uses Hadoop?

- Amazon/A9
- Facebook
- Google
- IBM
- Joost
- Last.fm
- New York Times
- PowerSet
- Veoh
- Yahoo!

Hadoop Ecosystem



Analysis: Mahout

API: MapReduce, MapReduce, Pig, Hive, HBase

Data Serialization: Avro

Processing Framework: MapReduce v2, Tez, Hoya

Workflow Engine: Oozie

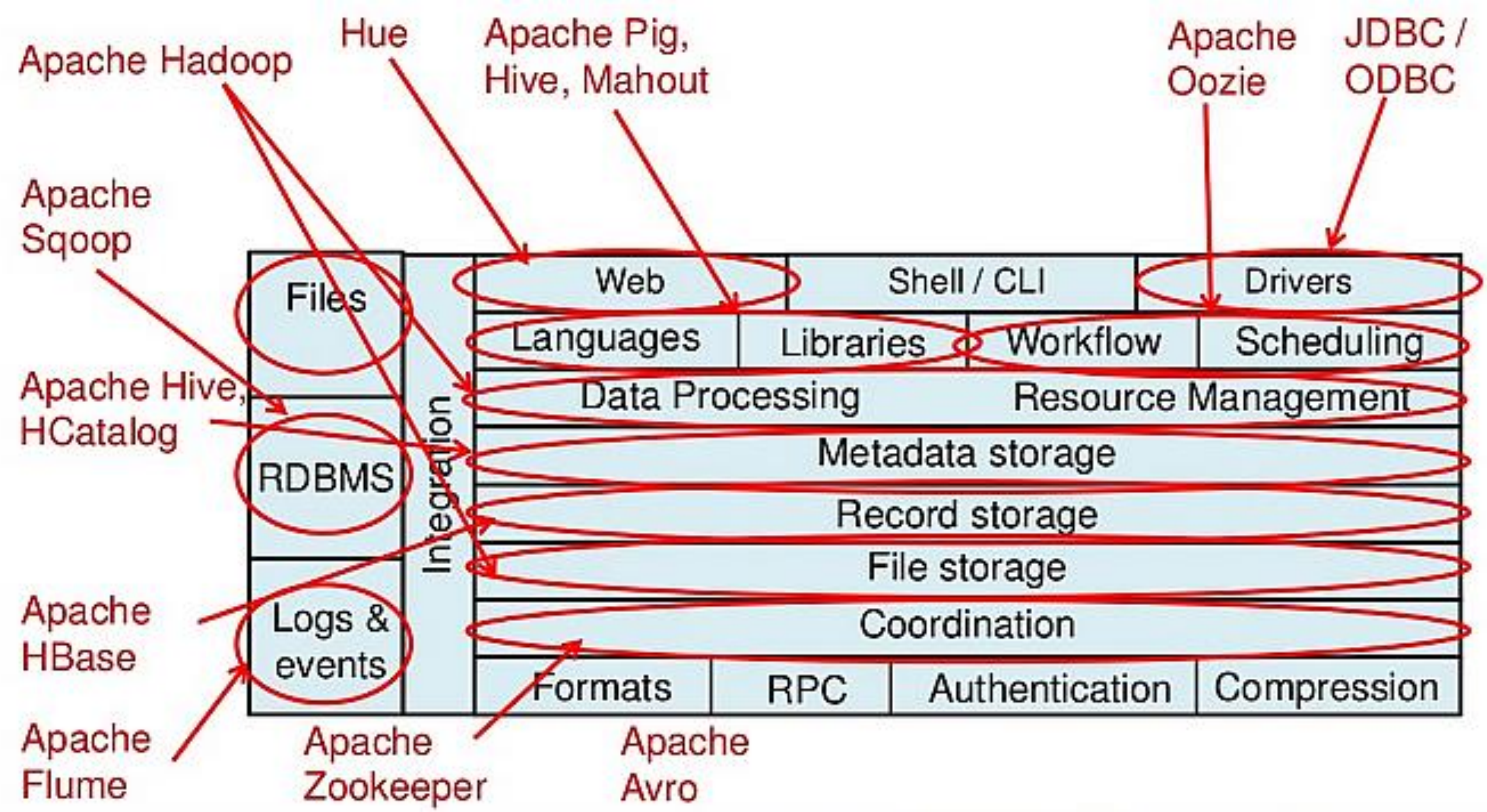
Resource Management: YARN

Distributed Storage: HDFS

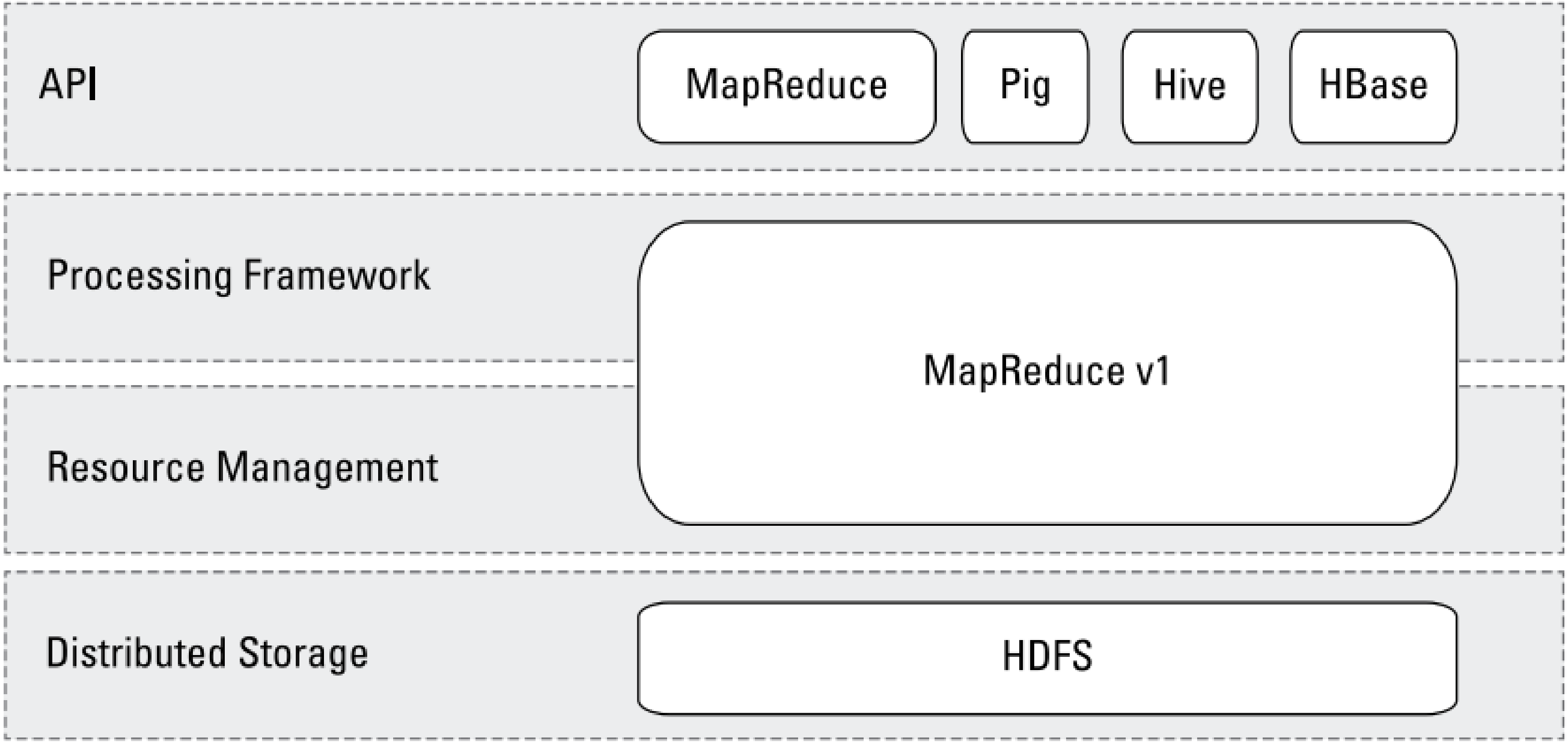
Data Movement: Flume, Sqoop

Administration and Server Coordination: Hue, Ambari, Zookeeper

The wider Hadoop Ecosystem



Hadoop1 Data Processing Architecture



Hadoop1 Data Processing Architecture

- **Distributed storage**: The Hadoop Distributed File System (HDFS) is the storage layer where the data, interim results, and final result sets are stored.

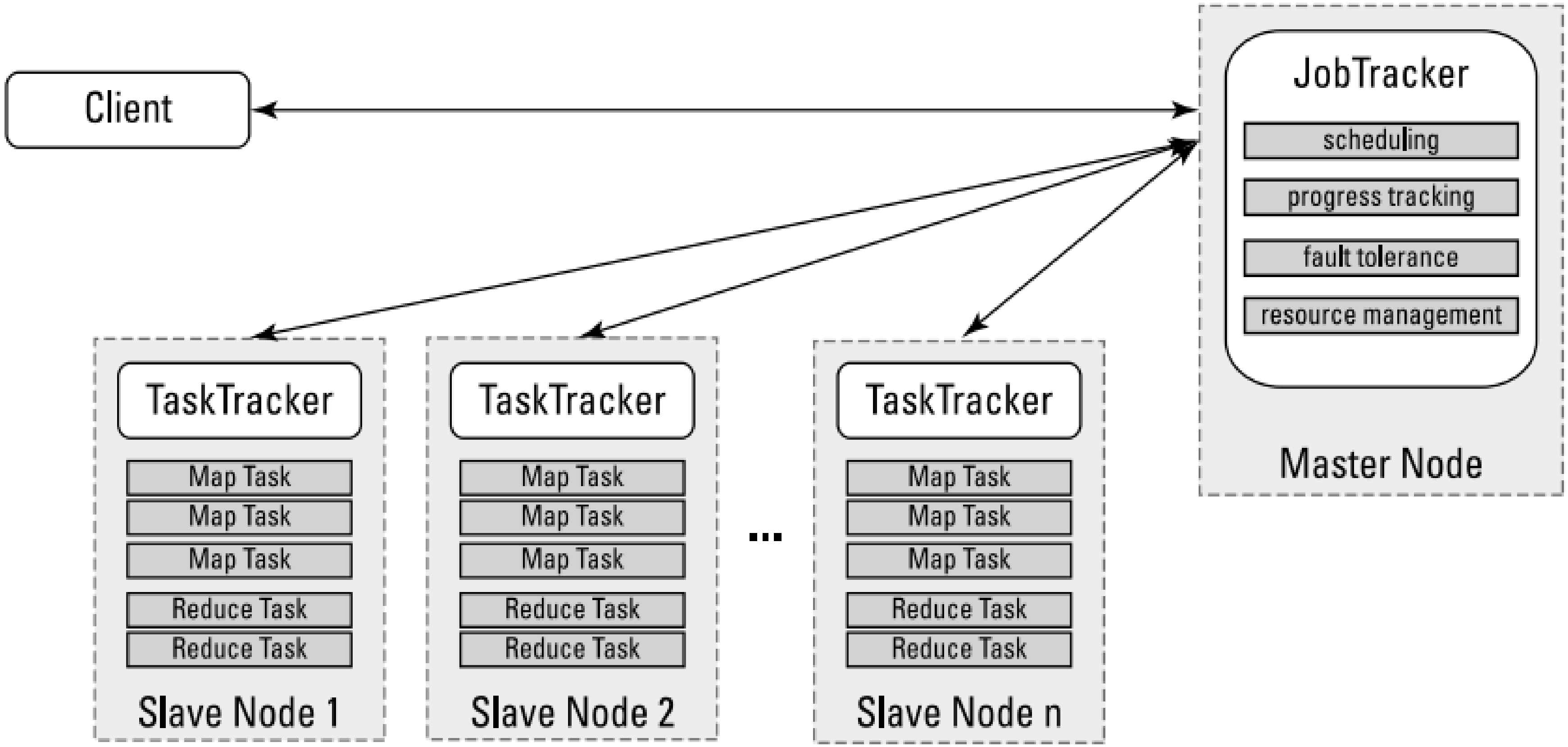
- **Resource management**: In addition to disk space, all slave nodes in the Hadoop cluster have CPU cycles, RAM, and network bandwidth. A system such as Hadoop needs to be able to parcel out these resources so that multiple applications and users can share the cluster in predictable and tunable ways. This job is done by the JobTracker daemon.

Hadoop1 Data Processing Architecture

- **Processing framework:** The MapReduce process flow defines the execution of all applications in Hadoop 1. this begins with the map phase; continues with aggregation with shuffle, sort, or merge; and ends with the reduce phase. In Hadoop 1, this is also managed by the JobTracker daemon, with local execution being managed by TaskTracker daemons running on the slave nodes.

- **Application Programming Interface (API)**: Applications developed for Hadoop 1 needed to be coded using the MapReduce API. In Hadoop 1, the Hive and Pig projects provide programmers with easier interfaces for writing Hadoop applications, and underneath the hood, their code compiles down to MapReduce.

Hadoop1 Execution



Hadoop1 Execution

1. The client application submits an application request to the JobTracker.

2. The JobTracker determines how many processing resources are needed to execute the entire application. This is done by requesting the locations and names of the files and data blocks that the application needs from the NameNode, and calculating how many map tasks and reduce tasks will be needed to process all this data.

3. The JobTracker looks at the state of the slave nodes and queues all the map tasks and reduce tasks for execution.

4. As processing slots become available on the slave nodes, map tasks are deployed to the slave nodes. Map tasks assigned to specific blocks of data are assigned to nodes where that same data is stored.

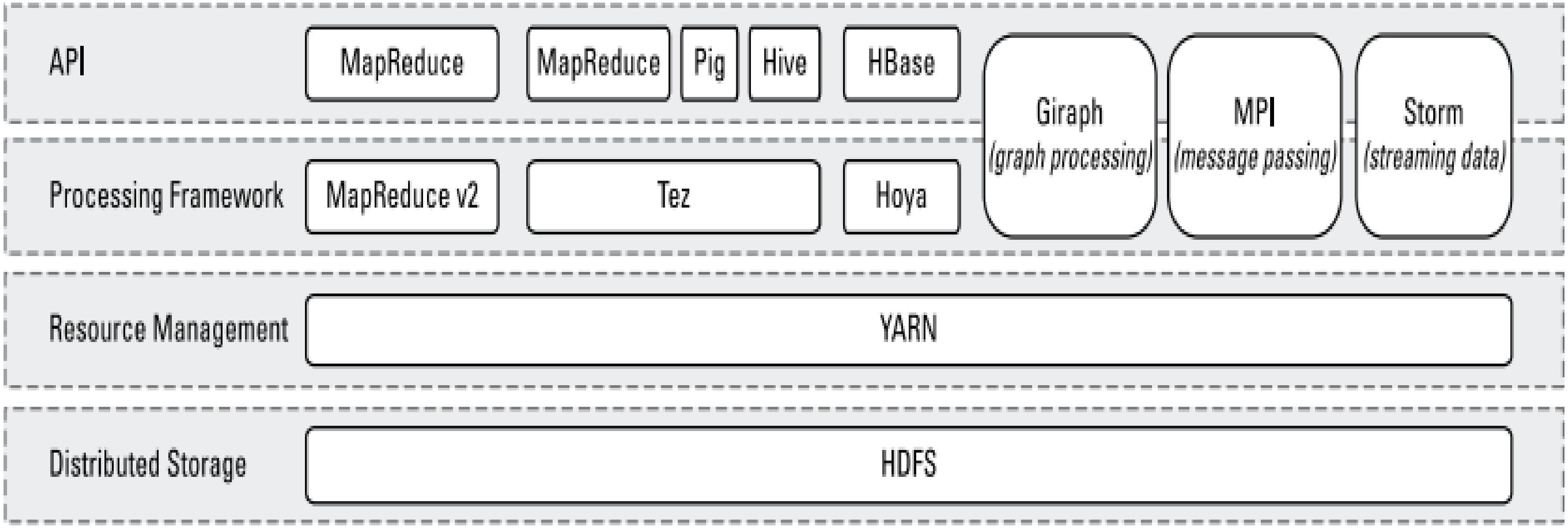
Hadoop1 Execution

5. The JobTracker monitors task progress, and in the event of a task failure or a node failure, the task is restarted on the next available slot. If the same task fails after four attempts (which is a default value and can be customized), the whole job will fail.

6. After the map tasks are finished, reduce tasks process the interim result sets from the map tasks.

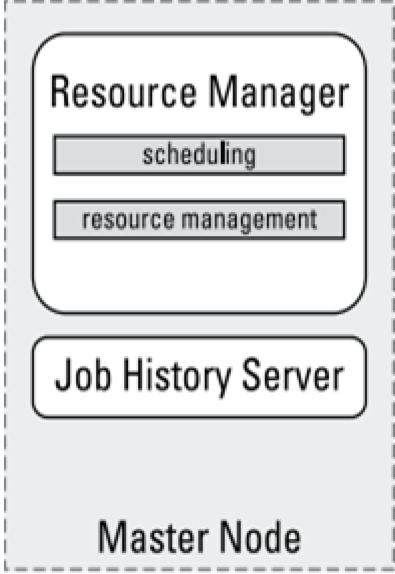
7. The result set is returned to the client application.

Hadoop Data Processing Architecture With Yarn



- YARN – Yet Another Resource Negotiator:
  - A Tool that enables the other processing framworks to run on Hadoop.
  - A general-purpose resource management facility that can schedule and assign CPU cycles and memory (and in the future, other resources, such as network bandwidth) from the Hadoop cluster to waiting applications.

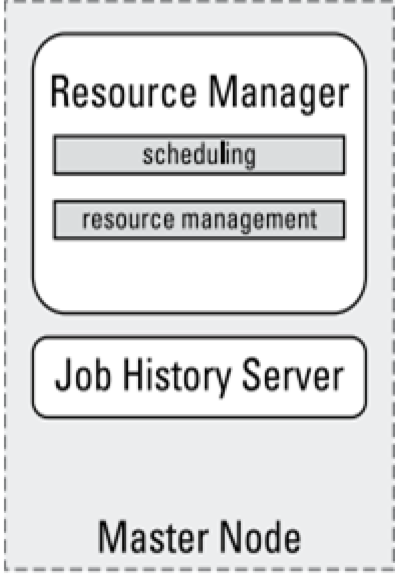
YARN



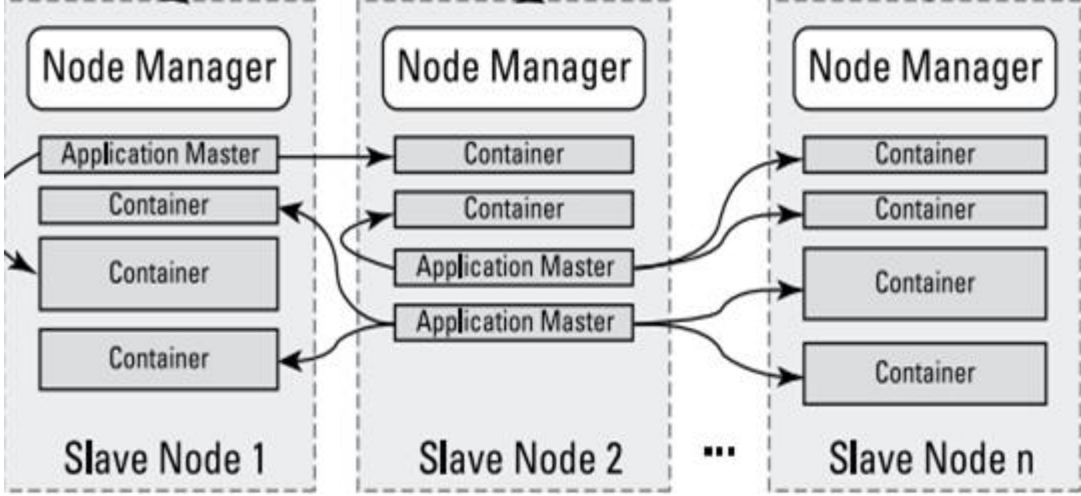
- **Resource Manager**
  - governs all the data processing resources in the Hadoop cluster
  - The Resource Manager is a dedicated scheduler that assigns resources to requesting applications
  - Its only tasks are to maintain a global view of all resources in the cluster, handling resource requests, scheduling the request, and then assigning resources to the requesting application
  - The Resource Manager, a critical component in a Hadoop cluster, should run on a dedicated master node

YARN

- **Job History Server**
  - a function that the JobTracker used to handle
  - Any client requests for a job history or the status of current jobs are served by the Job History Server.

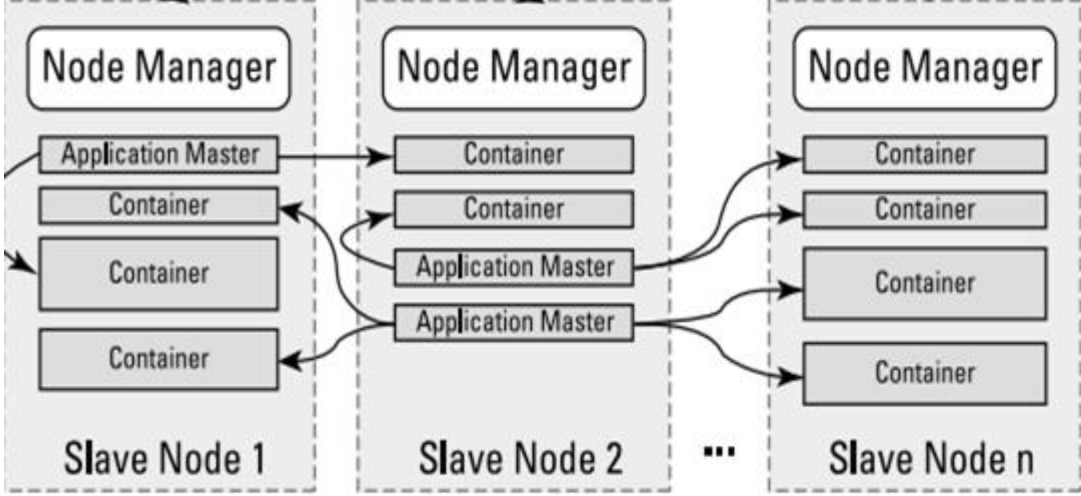


YARN



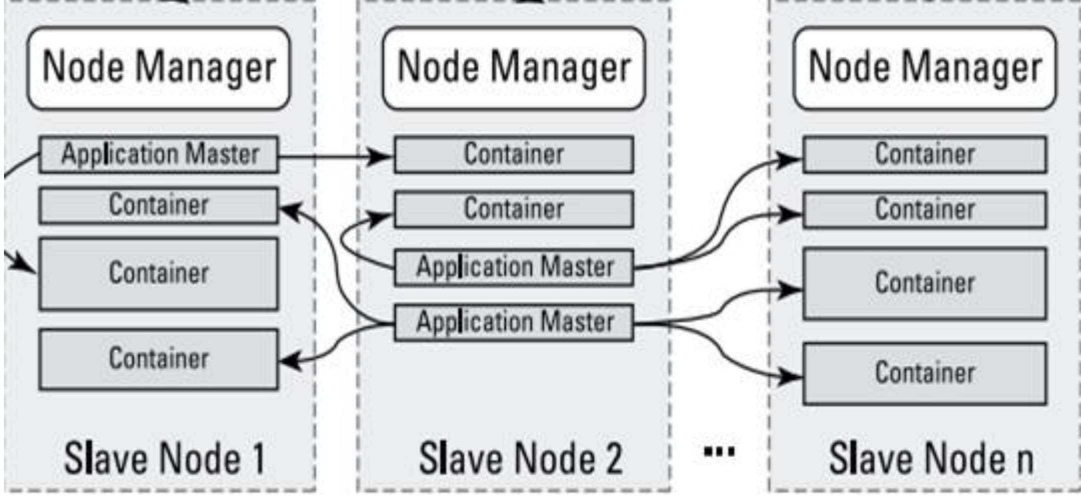
- **Node Manager**
  - Each slave node has a Node Manager
  - Acts as a slave for the Resource Manager.
  - Each Node Manager tracks the available data processing resources on its slave node and sends regular reports to the Resource Manager.
  - The processing resources in a Hadoop cluster are consumed in bite-size pieces called containers

YARN



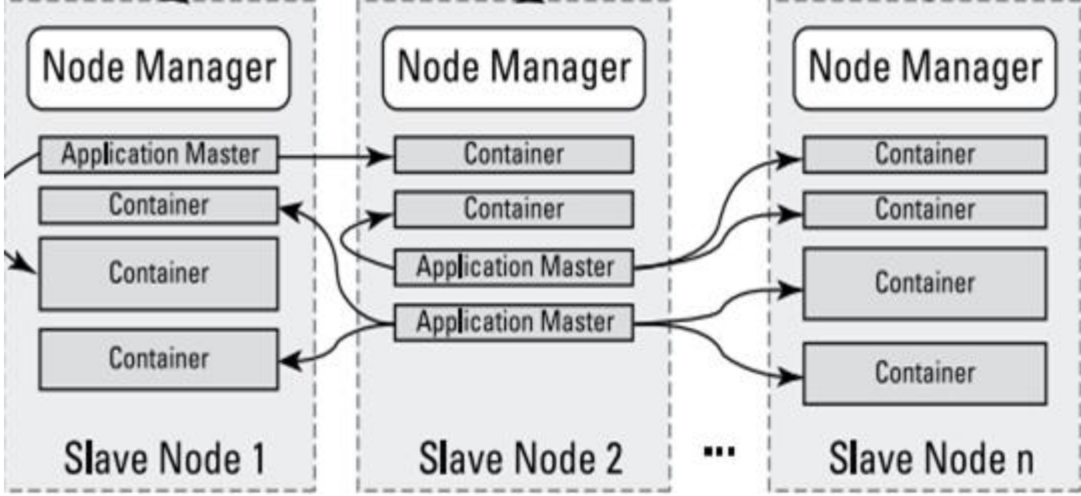
- **Container**
  - A container is a collection of all the resources necessary to run an application: CPU cores, memory, network bandwidth, and disk space
  - A deployed container runs as an individual process on a slave node in a Hadoop cluster.
  - containers can be requested with custom amounts of resources.
  - All container processes running on a slave node are initially provisioned, monitored, and tracked by that slave node's Node Manager

YARN



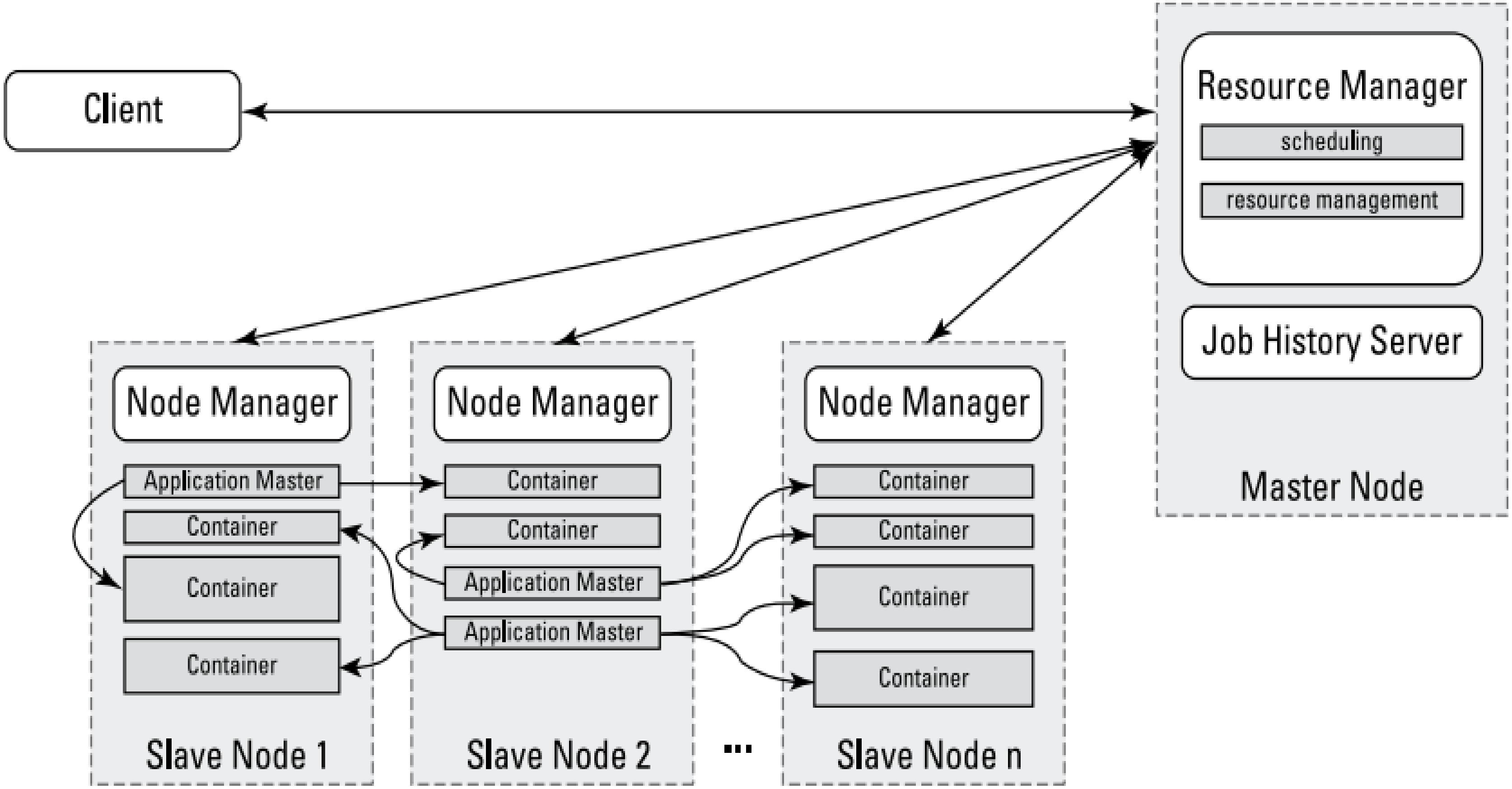
- **Application Master**
  - Each application running on the Hadoop cluster has its own, dedicated Application Master instance, which actually runs in a container process on a slave node
  - This is work that the JobTracker did for every application in Hadoop1
  - Application Master sends heartbeat messages to the Resource Manager with its status and the state of the application's resource needs

YARN



- **Application Master**
  - Based on the results of the Resource Manager's scheduling, it assigns container resource leases — basically reservations for the resources containers need — to the Application Master on specific slave nodes
  - The Application Master oversees the full lifecycle of an application, all the way from requesting the needed containers from the Resource Manager to submitting container lease requests to the NodeManager
  - Each application framework that's written for Hadoop must have its own Application Master implementation. MapReduce, for example, has a specific Application Master that's designed to execute map tasks and reduce tasks in sequence

YARN's Application Execution



YARN’s Application Execution

1. The client application submits an application request to the Resource Manager.

2. The Resource Manager asks a Node Manager to create an Application Master instance for this application. The Node Manager gets a container for it and starts it up.

3. This new Application Master initializes itself by registering itself with the Resource Manager.

4. The Application Master figures out how many processing resources are needed to execute the entire application. This is done by requesting from the NameNode the names and locations of the files and data blocks the application needs and calculating how many map tasks and reduce tasks are needed to process all this data.

YARN’s Application Execution

5. The Application Master then requests the necessary resources from the Resource Manager. The Application Master sends heartbeat messages to the Resource Manager throughout its lifetime, with a standing list of requested resources and any changes (for example, a kill request).

6. The Resource Manager accepts the resource request and queues up the specific resource requests alongside all the other resource requests that are already scheduled.

7. As the requested resources become available on the slave nodes, the Resource Manager grants the Application Master leases for containers on specific slave nodes.

8. The Application Master requests the assigned container from the Node Manager and sends it a Container Launch Context (CLC). The CLC includes everything the application task needs in order to run: environment variables, authentication tokens, local resources needed at runtime (for example, additional data files, or application logic in JARs), and the command string necessary to start the actual process. The Node Manager then creates the requested container process and starts it.

YARN’s Application Execution

9. The application executes while the container processes are running. The Application Master monitors their progress, and in the event of a container failure or a node failure, the task is restarted on the next available slot. If the same task fails after four attempts (a default value which can be customized), the whole job will fail. During this phase, the Application Master also communicates directly with the client to respond to status requests.

10. Also, while containers are running, the Resource Manager can send a kill order to the Node Manager to terminate a specific container. This can be as a result of a scheduling priority change or a normal operation, such as the application itself already being completed.

11. In the case of MapReduce applications, after the map tasks are finished, the Application Master requests resources for a round of reduce tasks to process the interim result sets from the map tasks.

YARN’s Application Execution

12. When all tasks are complete, the Application Master sends the result set to the client application, informs the Resource Manager that the application has successfully completed, deregisters itself from the Resource Manager, and shuts itself down.

MapReduce: Big Data Processing Abstraction

Typical Large-Data Problem

Map

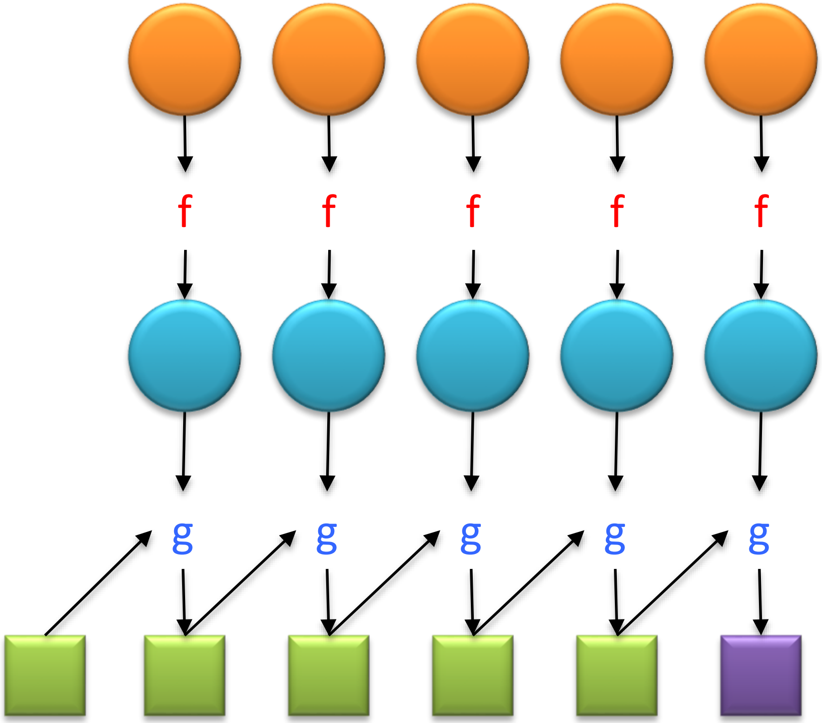
- Iterate over a large number of records
- Extract something of interest from each
- Shuffle and sort intermediate results

Reduce

- Aggregate intermediate results
- Generate final output

Key idea: provide a functional abstraction for these two operations

Roots in Functional Programming



MapReduce

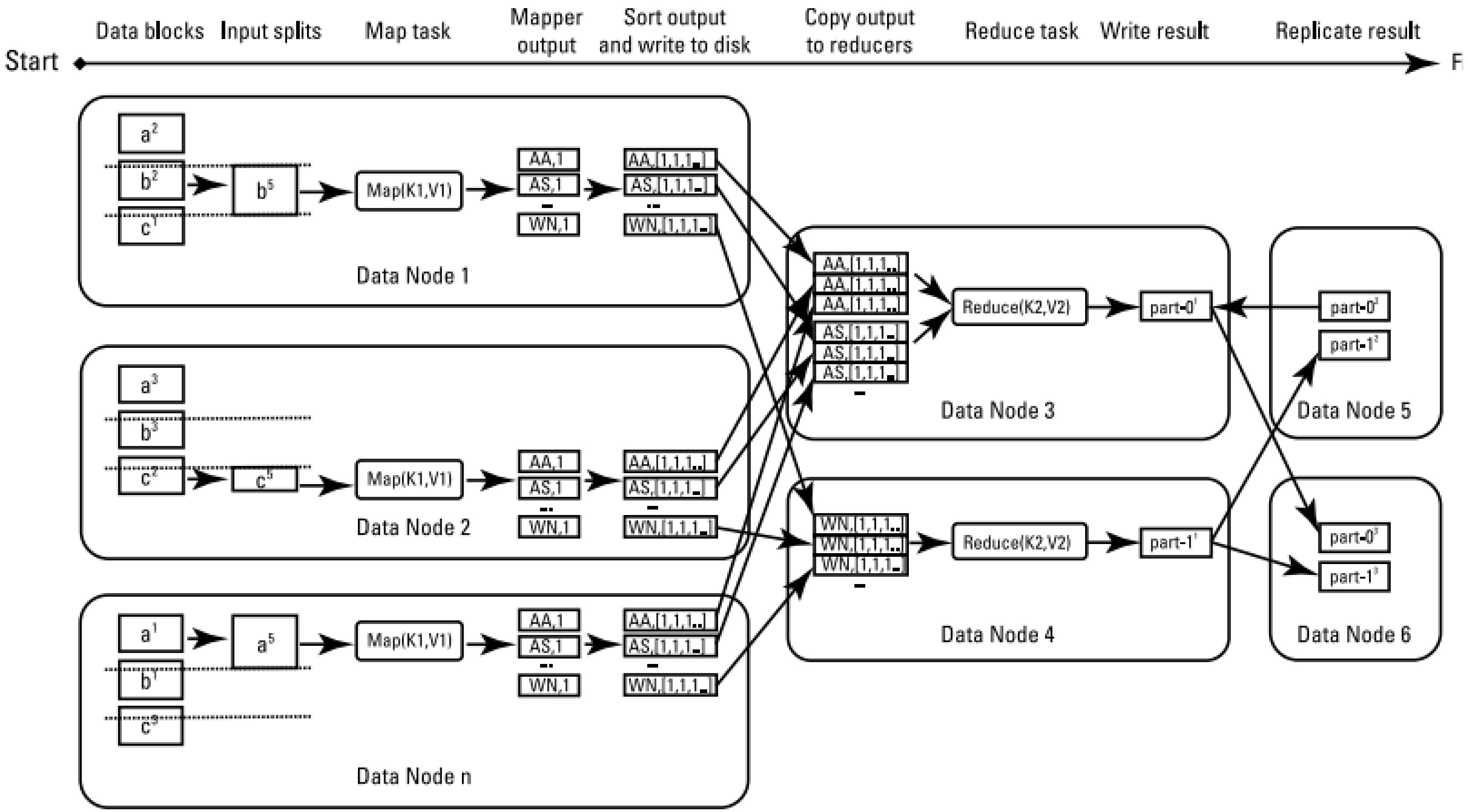
- Programmers specify two functions:

  map (k, v) → [(k’, v’)]

  reduce (k’, [v’]) → [(k’, v’)]

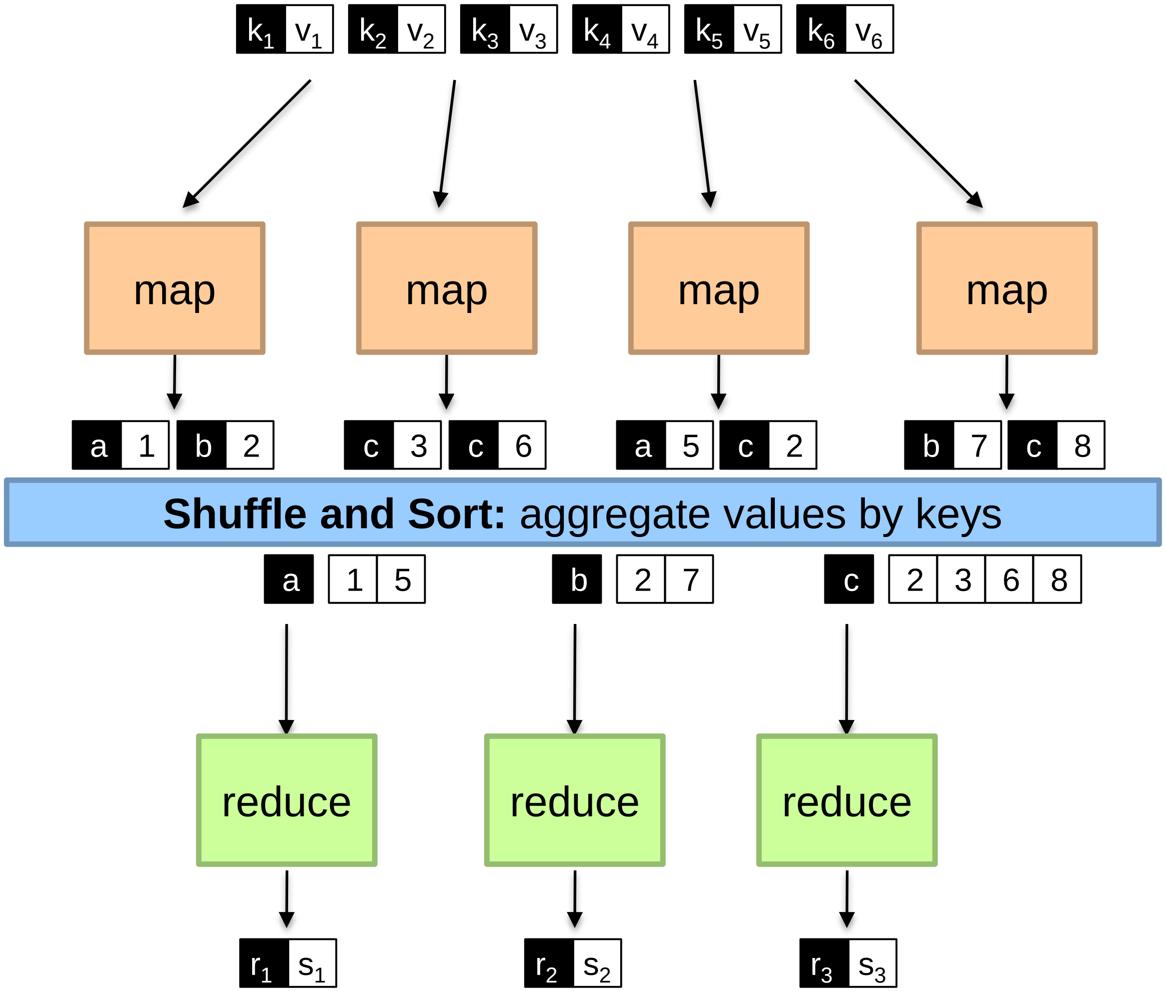
  - All values with the same key are sent to the same reducer
- The execution framework handles everything else…

MapReduce



map: (K1, V1) → list(K2, V2)

reduce: (K2, list(V2)) → list(K3, V3)



Shuffle and Sort: aggregate values by keys

MapReduce

- Programmers specify two functions:

  map (k, v) → <k', v'>*

  reduce (k', v') → <k', v'>*

  – All values with the same key are sent to the same reducer

- The execution framework handles everything else…

What's "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 FS (later)

MapReduce

- Programmers specify two functions:

  map (k, v) → [(k’, v’)]

  reduce (k’, [v’]) → [(k’, v’)]

  – All values with the same key are reduced together
- The execution framework handles everything else…
- Not quite…usually, programmers also specify:

  partition (k’, number of partitions) → partition for k’

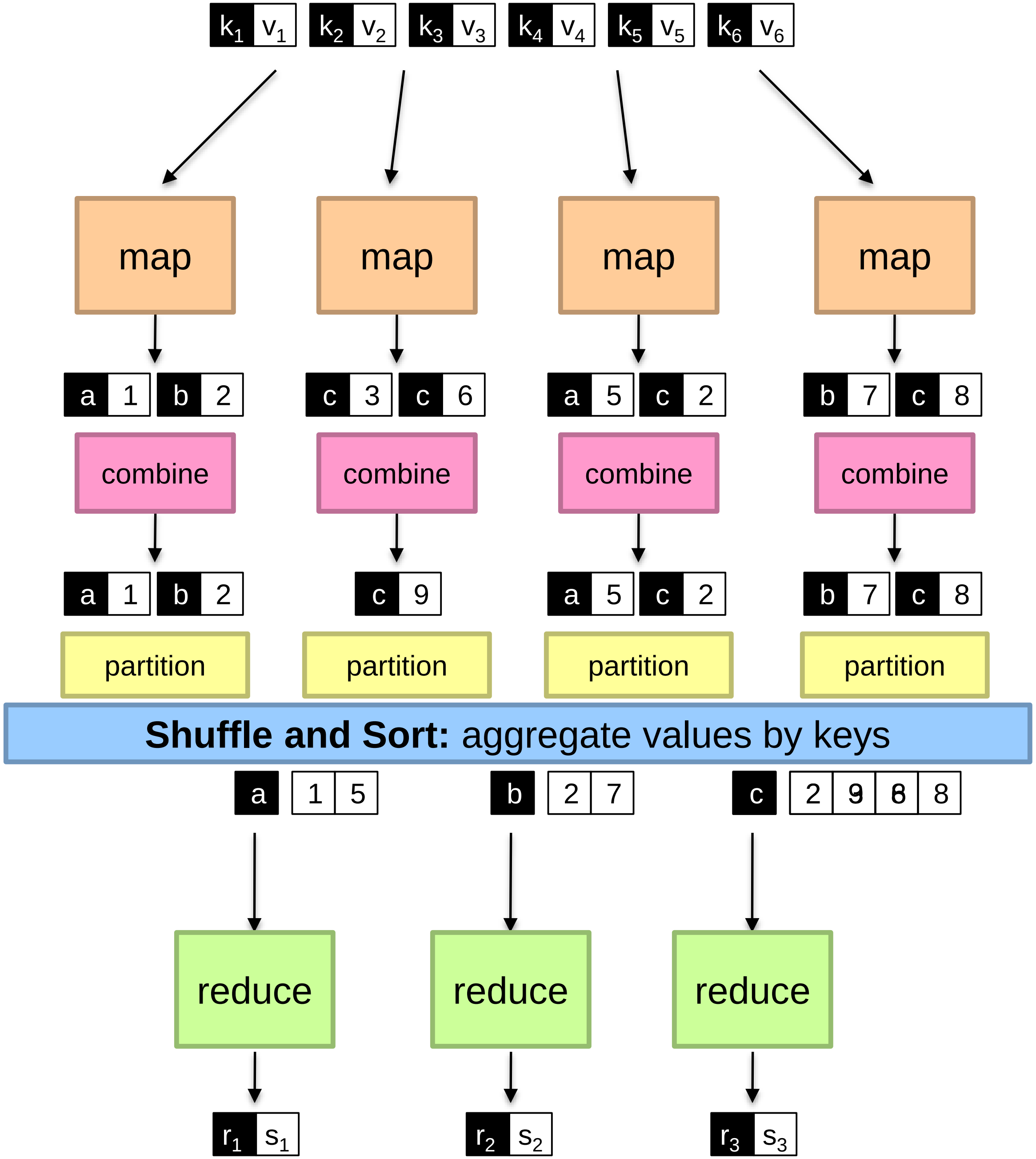
  – Often a simple hash of the key, e.g., hash(k’) mod n

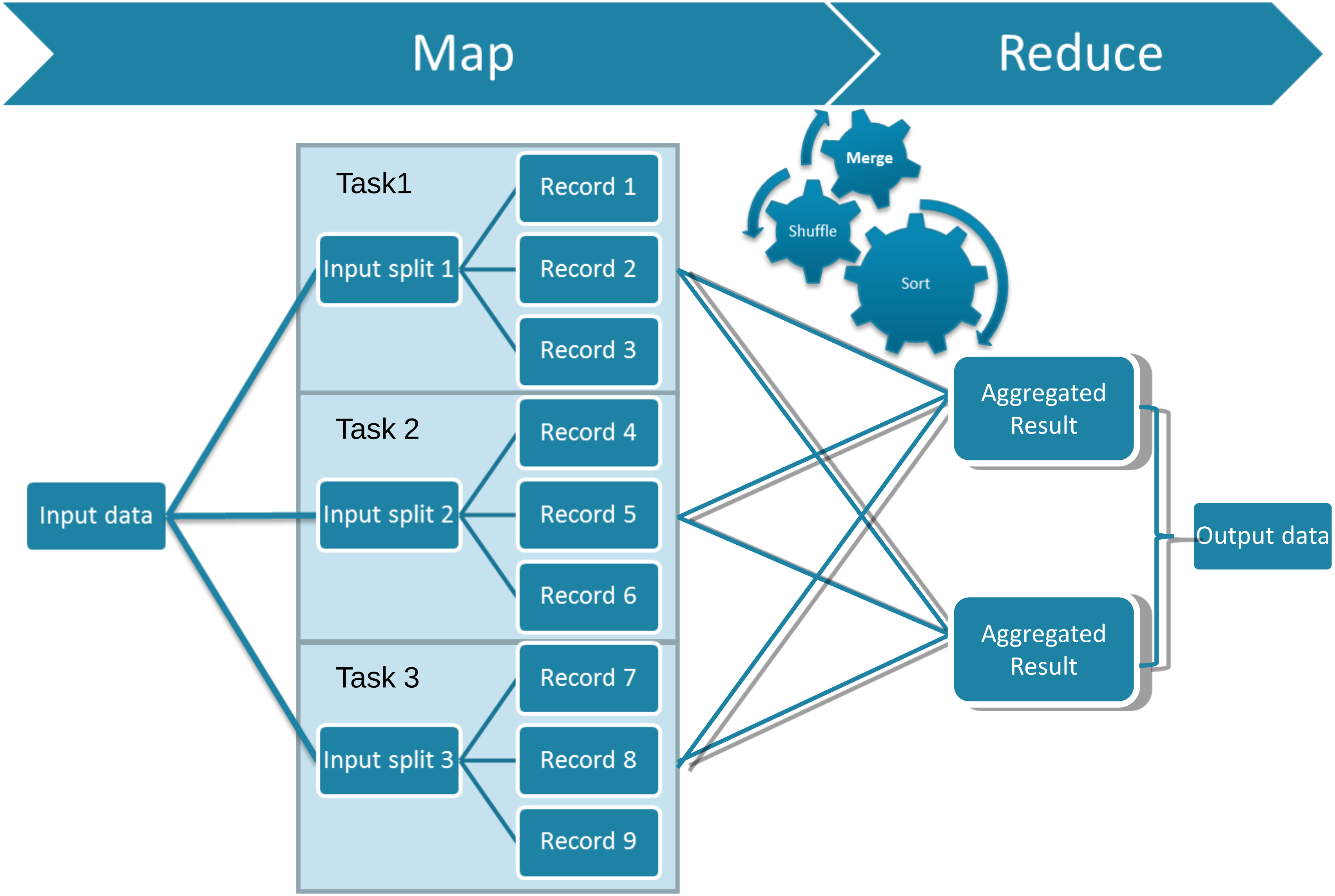
  – Divides up key space for parallel reduce operations

  combine (k’, [v’]) → [(k’, v’’)]

  – Mini-reducers that run in memory after the map phase

  – Used as an optimization to reduce network traffic


k_1 v_1
k_2 v_2
k_3 v_3
k_4 v_4
k_5 v_5
k_6 v_6
map
map
map
map
a 1 b 2
c 3 c 6
a 5 c 2
b 7 c 8
combine
combine
combine
combine
a 1 b 2
c 9
a 5 c 2
b 7 c 8
partition
partition
partition
partition
Shuffle and Sort: aggregate values by keys
a 1 5
b 2 7
c 2 9 6 8
reduce
reduce
reduce
r_1 s_1
r_2 s_2
r_3 s_3



Map

Reduce

Merge

Shuffle

Sort

Task1

Input split 1

Record 1

Record 2

Record 3

Task 2

Input split 2

Record 4

Record 5

Record 6

Task 3

Input split 3

Record 7

Record 8

Record 9

Input data

Aggregated Result

Aggregated Result

Output data

Two more details…

- Barrier between map and reduce phases
  - But we can begin copying intermediate data earlier
- Keys arrive at each reducer in sorted order
  - No enforced ordering *across* reducers

MapReduce can refer to…

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

Usage is usually clear from context!

“Hello World”: Word Count

```
Map(String docid, String text):
    for each word w in text:
        Emit(w, 1);

Reduce(String term, Iterator<Int> values):
    int sum = 0;
    for each v in values:
        sum += v;
    Emit(term, sum);
```

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, used in production
  - Now an Apache project
  - Rapidly expanding software ecosystem
- Lots of custom research implementations
  - For GPUs, cell processors, etc.

Example Word Count (Map)

```
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);
     }
   }
 }
```

Example Word Count (Reduce)

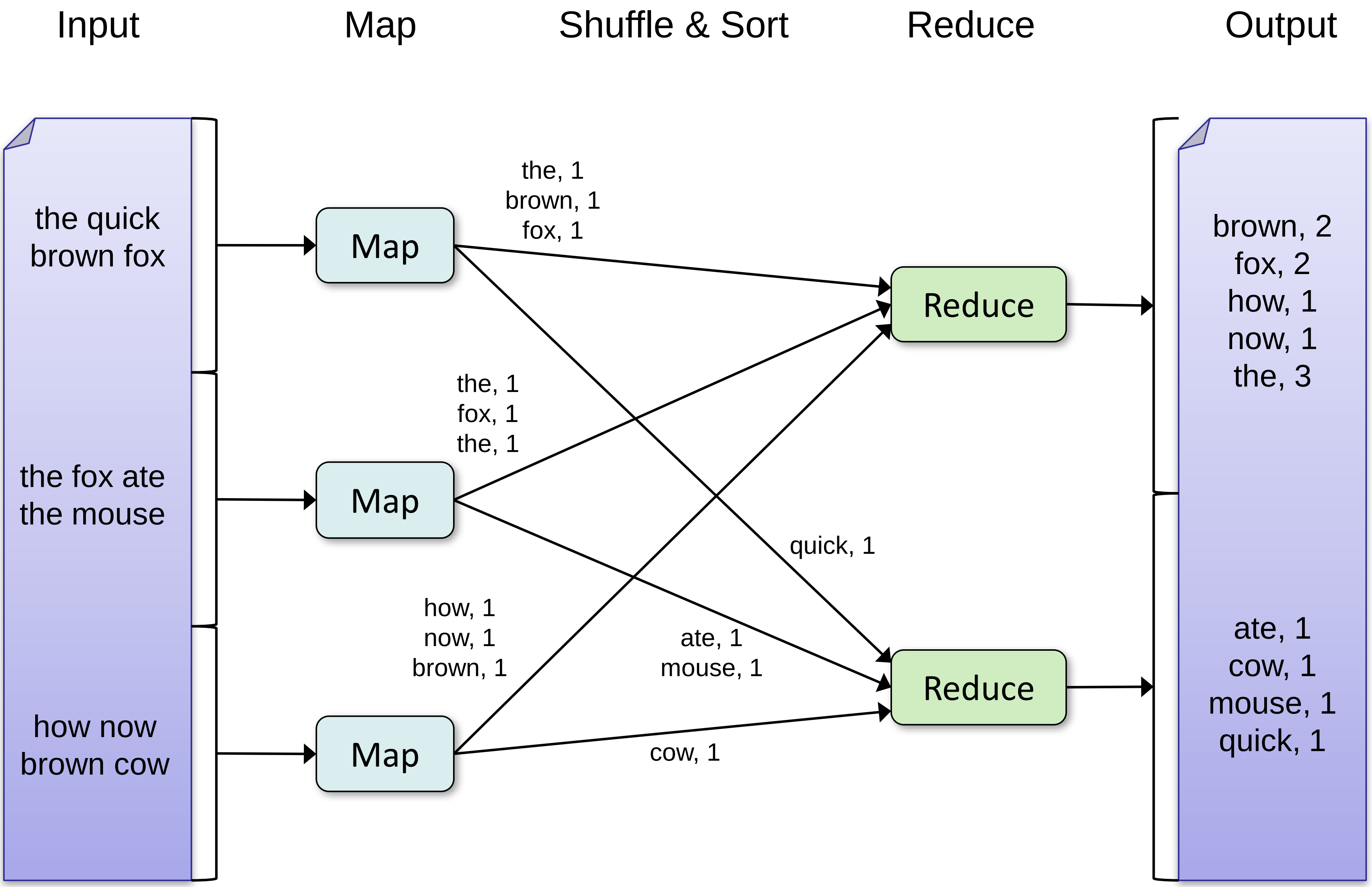
```
public static class IntSumReducer
      extends Reducer<Text,IntWritable,Text,IntWritable> {
  private IntWritable result = new IntWritable();

  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);
  }
 }
```

Example Word Count (Driver)

```
 public static void main(String[] args) throws Exception {
   Configuration conf = new Configuration();
   String[] otherArgs = new GenericOptionsParser(conf,
args).getRemainingArgs();
   if (otherArgs.length != 2) {
     System.err.println("Usage: wordcount <in> <out>");
     System.exit(2);
   }
   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);
 }
```

Word Count Execution



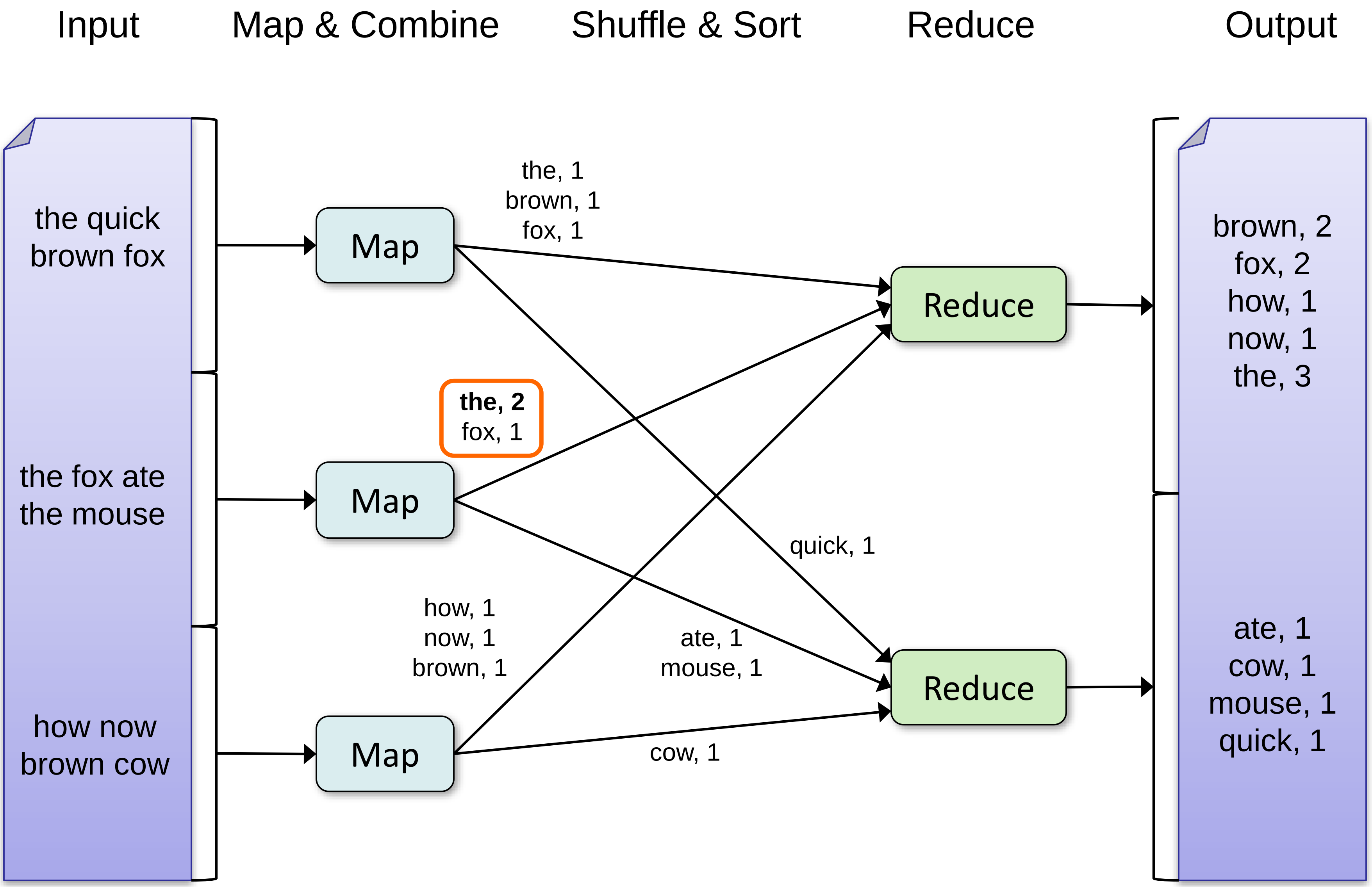
An Optimization: The Combiner

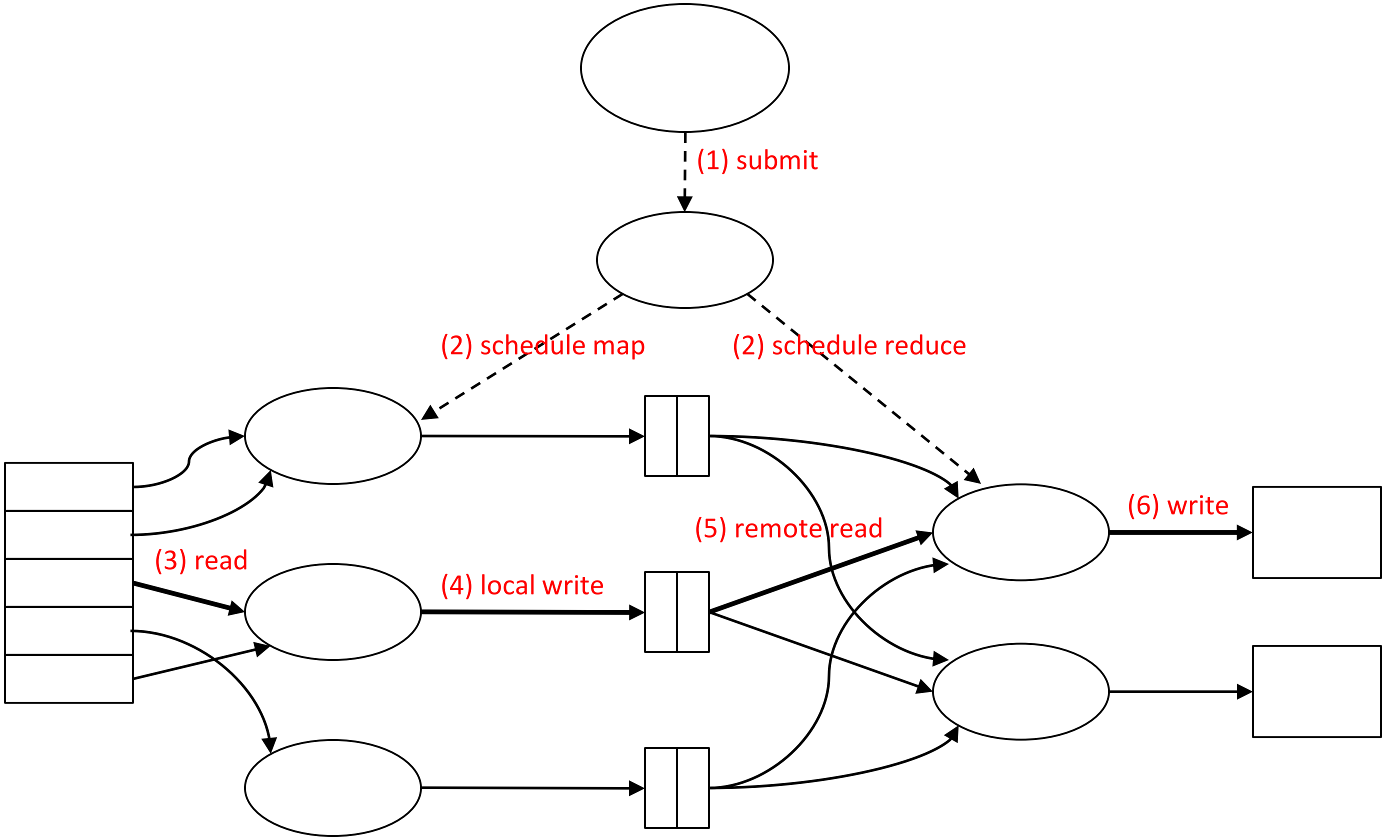
- A combiner is a local aggregation function for repeated keys produced by same map
- For associative ops. like sum, count, max
- Decreases size of intermediate data

- Example: local counting for Word Count:

```
def combiner(key, values):
  output(key, sum(values))
```

Word Count with Combiner




(1) submit
(2) schedule map
(2) schedule reduce
(3) read
(4) local write
(5) remote read
(6) write

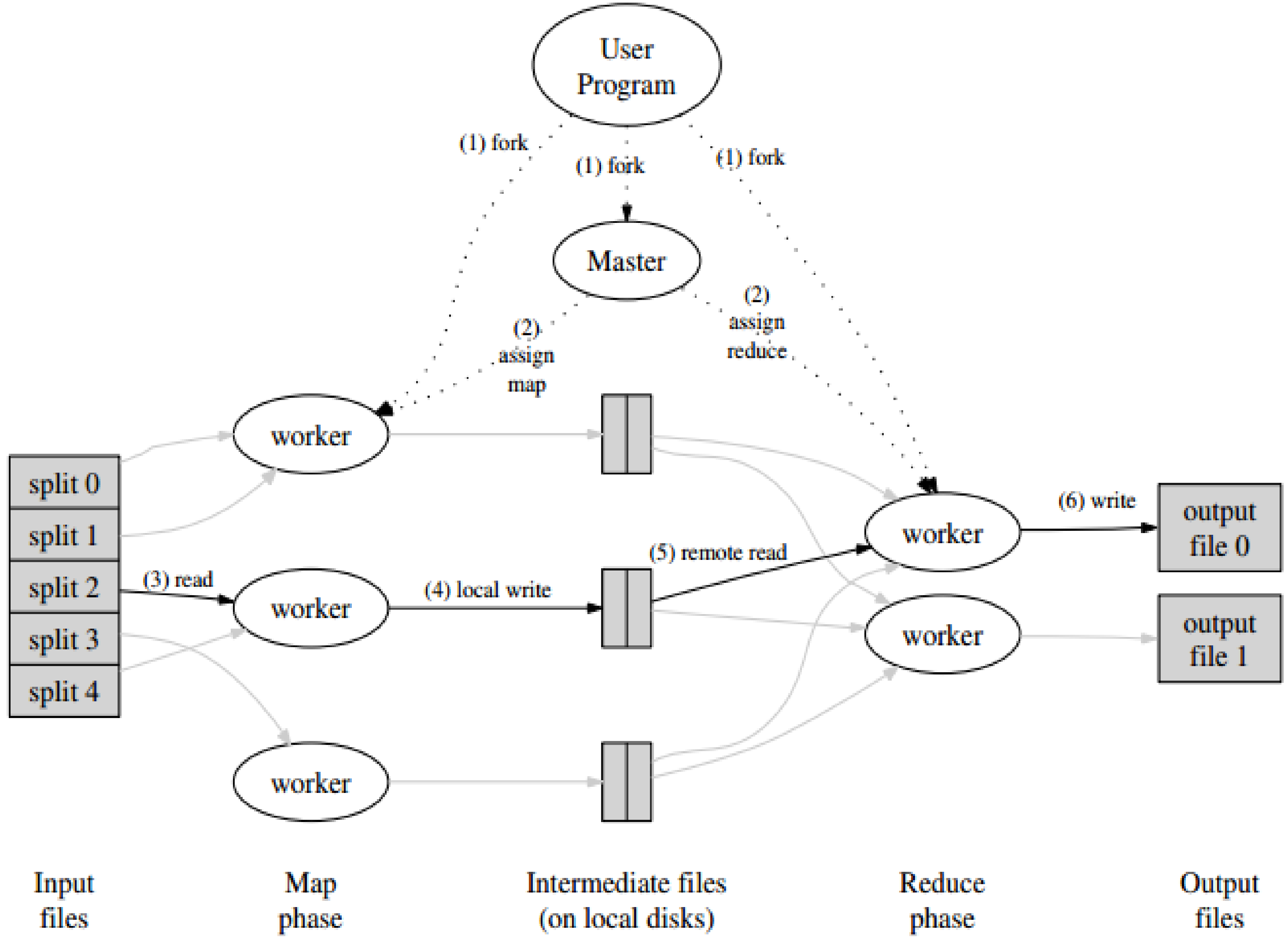


Figure 1: Execution overview

MapReduce

1. کتابخانه MapReduce در برنامه کاربر داده ها را به M قسمت تقسیم می کند. هر قسمت می تواند ۱۶ یا ۶۴ یا (در هدوپ ۱۲۸) مگابایت باشد و توسط کاربر قابل تغییر است و بعد از این برنامه را در کامپیوترهای کلاستر کپی می کند.

2. یکی از کپی های برنامه بخصوص است و Masterنام دارد. بقیه Wokerنامیده می شوند که توسط Masterبه آنها کار سپرده می شود. تعداد Mوظیفه Mapو Rوظیفه Reduceوجود دارند که می بایست به Workerها تخصیص داده شوند، Master، Workerهای بیکار را انتخاب می کند و به هریک وظیفه Mapیا Reduceرا اختصاص می دهد

3. Workerای که وظیفه Mapبه آن تخصیص یافته محتویات متناسب با قسمت داده اش را می خواند، زوج های key/valueرا از درون قسمت داده استخراج می کند و هریک را به عنوان پارامترورودی به تابع Mapتعریف شده توسط کاربر ارسال می کند. زوج key/valueمیانی تولید شده توسط تابع Mapدر حافظه بافر می شود

MapReduce

4. در ترتیب زمانی مشخصی، زوج key/valueهای میانی بافرشده در دیسک ذخیره می شوند سپس توسط تابع تقسیم بندی به Rقسمت تقسیم می شوند. مکان این زوج کلیدمقدار روی دیسک به Master بازگردانده می شود و Masterبنا به درخواست Workerهای Reduceمحل آنها را نشان می دهد

5. زمانی که پاسخ درخواست محل یک Worker Reduceتوسط Masterداده شد، از یک روال Remoteجهت خواندن داده های بافرشده در دیسک استفاده می کند. زمانی که Reduce Workerتمام داده های مورد نیازش را خواند آنها را با کلیدهای واسط مرتب سازی می کند ، سپس همگی کلیدهای یکسان با هم در یک گروه قرار می گیرند. مرتب سازی به این دلیل نیاز است که تعدادی از کلیدهای مختلف به یک وظیفه Reduceیکسان نگاشت می شوند. اگر تعداد کلیدهای میانی خیلی زیاد باشد و اینکار در حافظه صورت نگیرد از مرتب سازی externalاستفاده می شود

MapReduce

6. Reduce Workerدر حلقه ای کلیدهای میانی مرتب سازی شده را بررسی می کند و به هر یک از کلیدهای یکتا که برخورد کند،کلید و مقادیر مرتبط با آن را به تابع Reduceکاربر به عنوان پارامتر ورودی ارسال می کند. خروجی تابع Reduceبه یک فایل خروجی برای Reduceاین قسمت اضافه می شود.

7. زمانی که همگی وظیفه های Mapو Reduceبه پایان رسیدند، Master برنامه کاربر را فراخوانی می کند. در این بخش، MapReduceبه مد کاربر باز می گردد.

- بعد از اتمام موفقیت آمیز، خروجی اجرای MapReduceدر داخل Rفایل خروجی موجود می باشد (به ازای هر وظیفه یک فایل و نام فایل توسط کاربر مشخص می شود). معمولا کاربر نیازی به ترکیب این فایل ها در یک فایل ندارد، اغلب خروجی این فایل ها به عنوان ورودی MapReduceدیگری داده مي شود یا از آنها برای برنامه های توزیع شده دیگر استفاده می شود که نیاز به قسمت بندی فایل به تعدادی فایل دیگر دارند.

Geographical Data

- Large data sets including road, intersection, and feature data
- Problems that Google Maps has used MapReduce to solve
  - Locating roads connected to a given intersection
  - Rendering of map tiles
  - Finding nearest feature to a given address or location

Geographical Data

Example 1

- Input: List of roads and intersections
- Map: Creates pairs of connected points (road, intersection) or (road, road)
- Sort: Sort by key
- Reduce: Get list of pairs with same key
- Output: List of all points that connect to a particular road

Geographical Data

Example 2

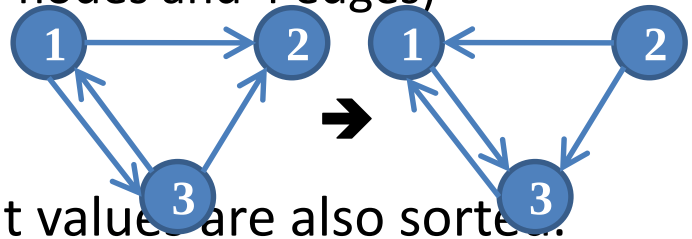
- Input: Graph describing node network with all gas stations marked
- Map: Search five mile radius of each gas station and mark distance to each node
- Sort: Sort by key
- Reduce: For each node, emit path and gas station with the shortest distance
- Output: Graph marked and nearest gas station to each node

Secondary Sort

Problem: Sorting on values

- E.g. Reverse graph edge directions & output in node order
  - Input: adjacency list of graph (3 nodes and 4 edges)

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



- Note, the node_ids in the output values are also sorted. But Hadoop only sorts on keys!

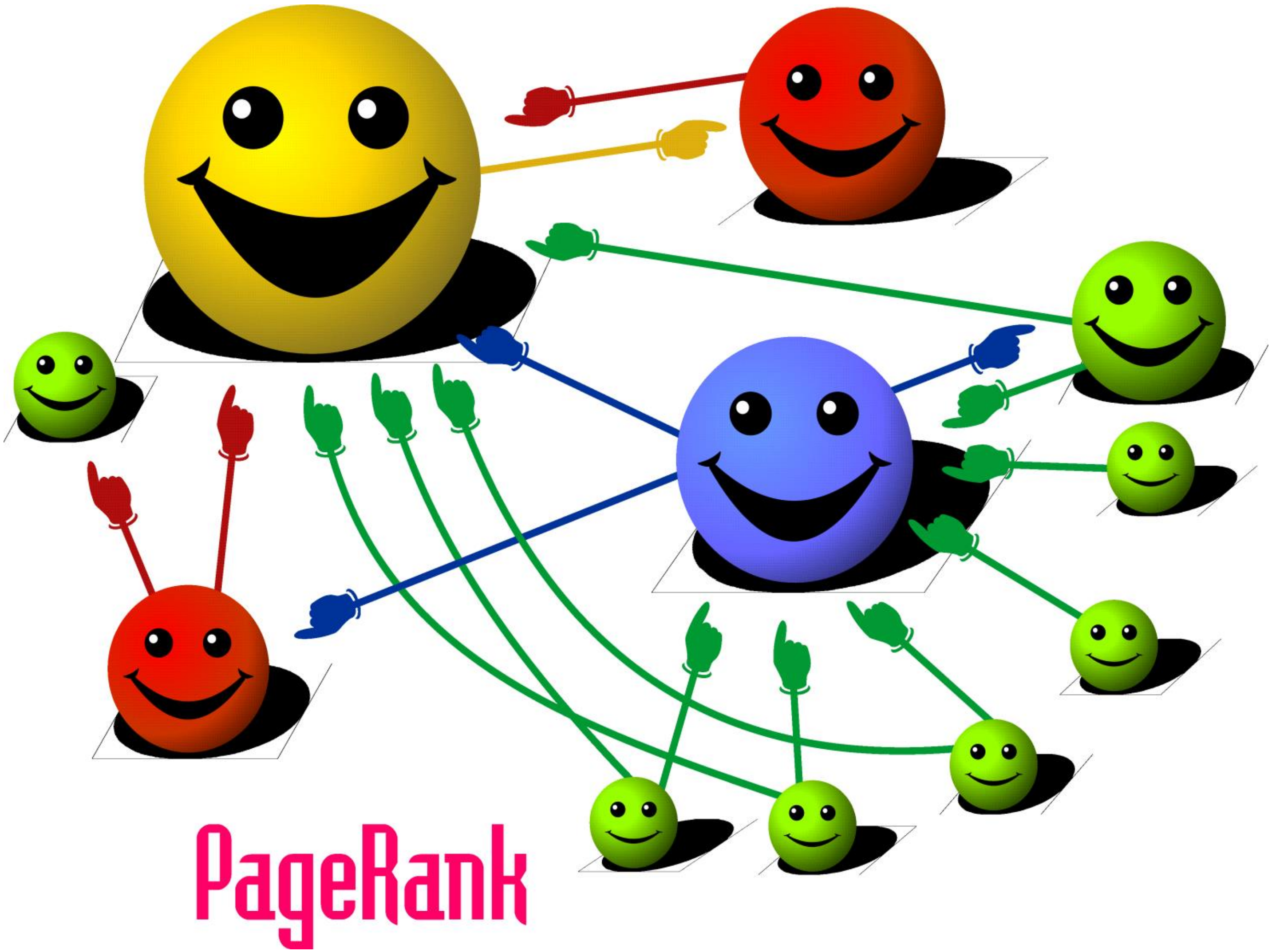
Solution: Secondary sort

- Map
  - In: (3, [1, 2]), (1, [2, 3]).
  - Intermediate: (1, [3]), (2, [3]), (2, [1]), (3, [1]). (reverse edge direction)
  - Out: (<1, 3>, [3]), (<2, 3>, [3]), (<2, 1>, [1]), (<3, 1>, [1]).
  - Copy node_ids from value to key.

Secondary Sort

Secondary Sort (ctd.)

- Shuffle on Key.field1, and Sort on whole Key (both fields)
  - In: (<1, 3>, [3]), (<2, 3>, [3]), (<2, 1>, [1]), (<3, 1>, [1])
  - Out: (<1, 3>, [3]), (<2, 1>, [1]), (<2, 3>, [3]), (<3, 1>, [1])
- Grouping comparator
  - Merge according to part of the key
  - Out: (<1, 3>, [3]), (<2, 1>, [1, 3]), (<3, 1>, [1])
    this will be the reducer's input
- Reduce
  - Merge & output: (1, [3]), (2, [1, 3]), (3, [1])



PageRank

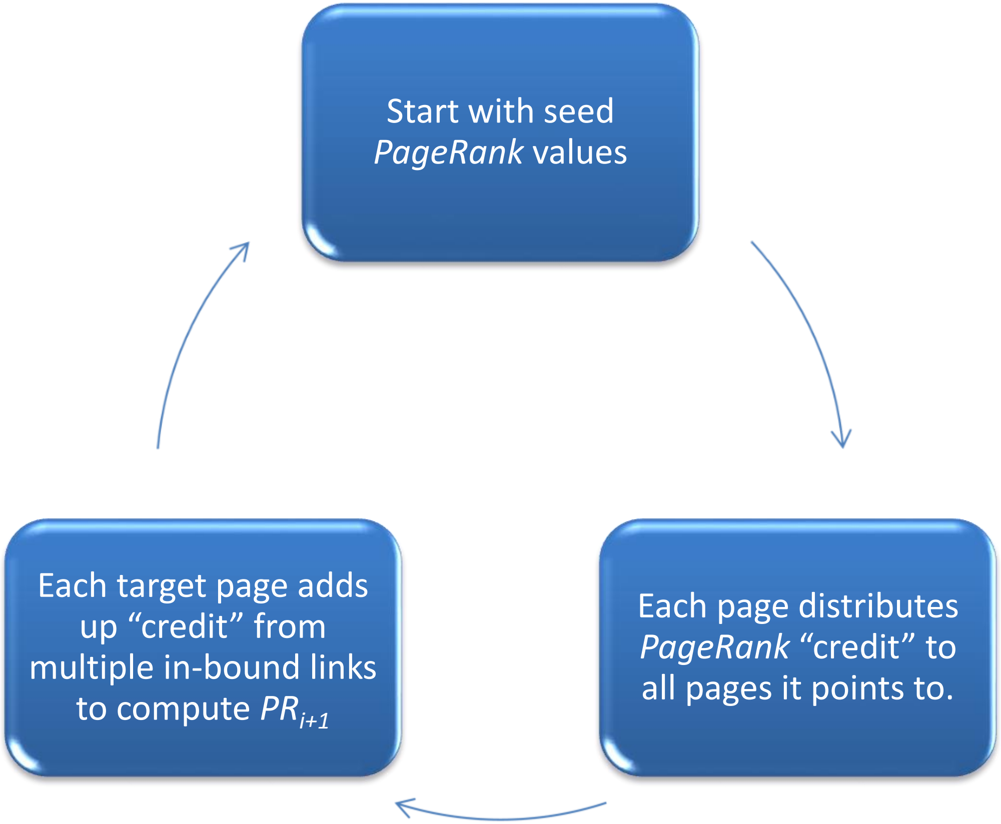
MapReduce : PageRank

- PageRank models the behavior of a “random surfer”.

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

- C(t) is the out-degree of t, and (1-d) is a damping factor (random jump)
- The “random surfer” keeps clicking on successive links at random not taking content into consideration.
- Distributes its pages rank equally among all pages it links to.
- The dampening factor takes the surfer “getting bored” and typing arbitrary URL.

Computing PageRank



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}

PageRank : Key Insights

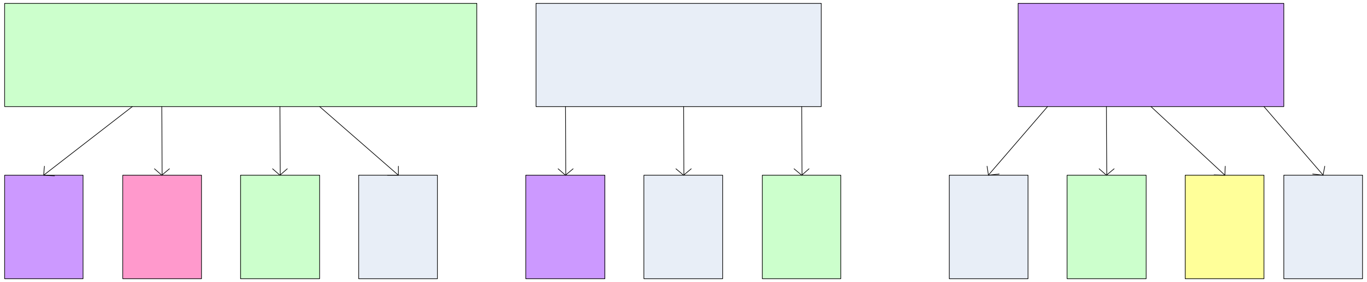
- Effects at each iteration is local. i+1th iteration depends only on ith iteration
- At iteration i, PageRank for individual nodes can be computed independently

PageRank using MapReduce

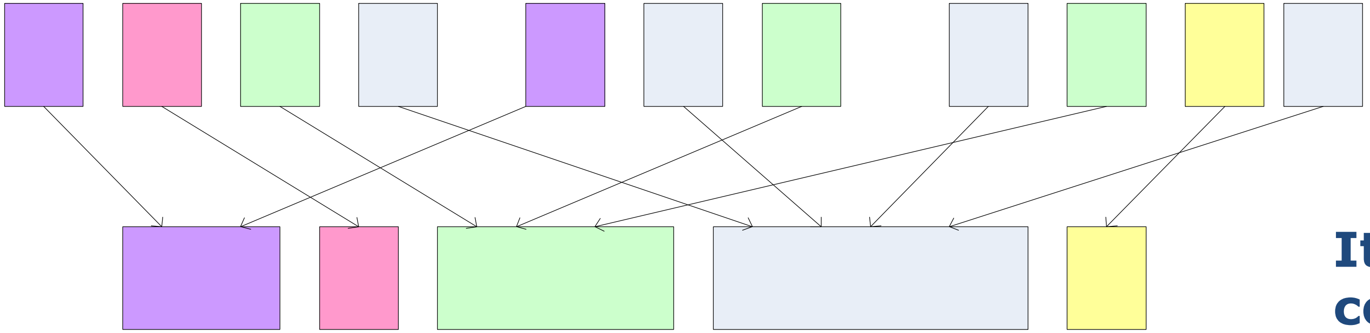
- Use Sparse matrix representation (M)
- Map each row of M to a list of PageRank "credit" to assign to out link neighbours.
- These prestige scores are *reduced* to a single PageRank value for a page by aggregating over them.

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

Phase 1: Process HTML

- Map task takes (URL, page-content) pairs and maps them to (URL, (PR_{init}, list-of-urls))
  - PR_{init} is the "seed" PageRank for URL
  - list-of-urls contains all pages pointed to by URL
- Reduce task is just the identity function

Phase 2: PageRank Distribution

- Reduce task gets (URL, url_list) and many (URL, *val*) values
  - Sum *val*s and fix up with *d to get new PR*
  - Emit (URL, (new_rank, url_list))

- Check for convergence using non parallel component

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
- Side-effect files

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, $score_2^{(t+1)}$), ..

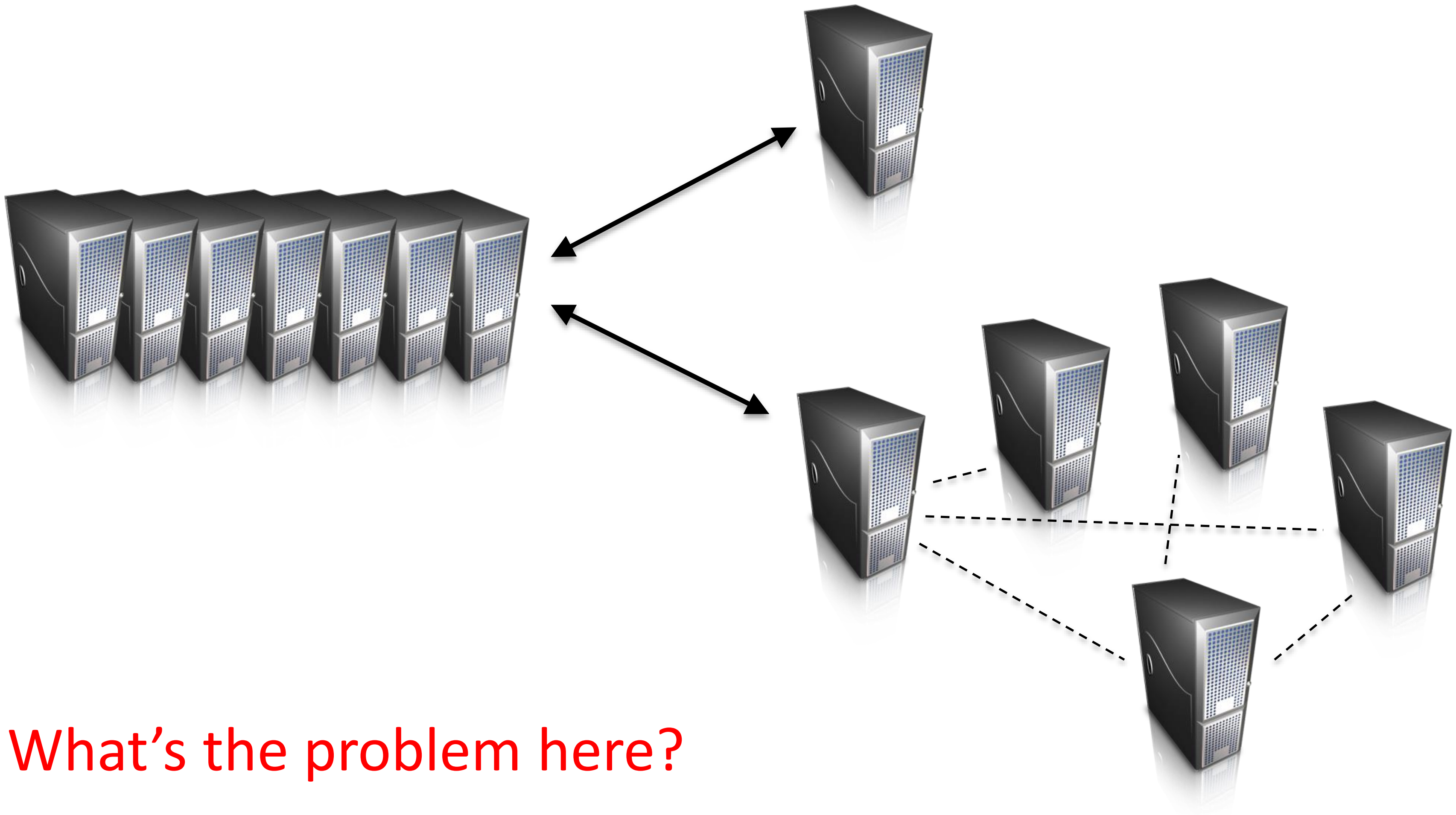
PageRank: Database Join to associate outlinks with score

- 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}, ..]) ..

PageRank:
Side Effect Files for dangling nodes

- Dangling Nodes
  - Nodes with no outlinks (observed but not crawled URLs)
  - Score has no outlet
    - need to distribute to all graph nodes evenly
- Map for dangling nodes:
  - In: .., (id_3, [score_3]), ..
  - Out: .., ("*", 0.85×score_3), ..
- Reduce
  - In: .., ("*", [score_1, score_2, ..]), ..
  - Out: .., everything else, ..
  - Output to side-effect: ("*", score), fed to Mapper of next iteration

How do we get data to the workers?



What's the problem here?

Distributed File System

- Don’t move data to workers… move workers to the data!
  - Store data on the local disks of nodes in the cluster
  - Start up the workers on the node that has the data local
- Why?
  - Not enough RAM to hold all the data in memory
  - Disk access is slow, but disk throughput is reasonable
- A distributed file system is the answer
  - GFS (Google File System) for Google’s MapReduce
  - HDFS (Hadoop Distributed File System) for Hadoop

GFS: Assumptions

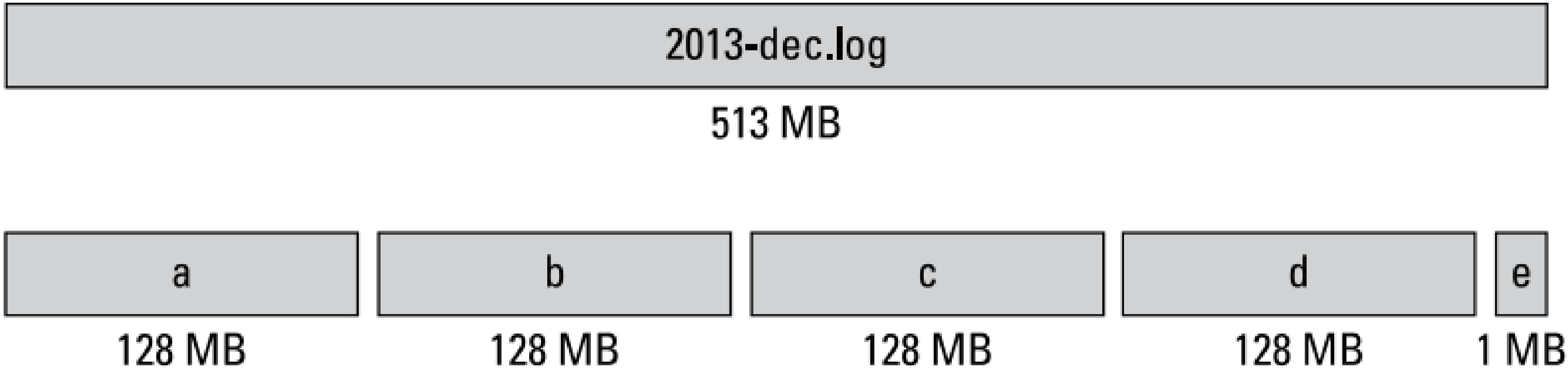
- Commodity hardware over “exotic” hardware
  - Scale “out”, not “up”
- High component failure rates
  - Inexpensive commodity components fail all the time
- “Modest” number of huge files
  - Multi-gigabyte files are common, if not encouraged
- Files are write-once, mostly appended to
  - Perhaps concurrently
- Large streaming reads over random access
  - High sustained throughput over low latency

GFS: Design Decisions

- Files stored as chunks
  - Fixed size (64MB)
- Reliability through replication
  - Each chunk replicated across 3+ chunkservers
- Single master to coordinate access, keep metadata
  - Simple centralized management
- No data caching
  - Little benefit due to large datasets, streaming reads
- Simplify the API
  - Push some of the issues onto the client (e.g., data layout)

HDFS = GFS clone (same basic ideas)

Files stored as chunks



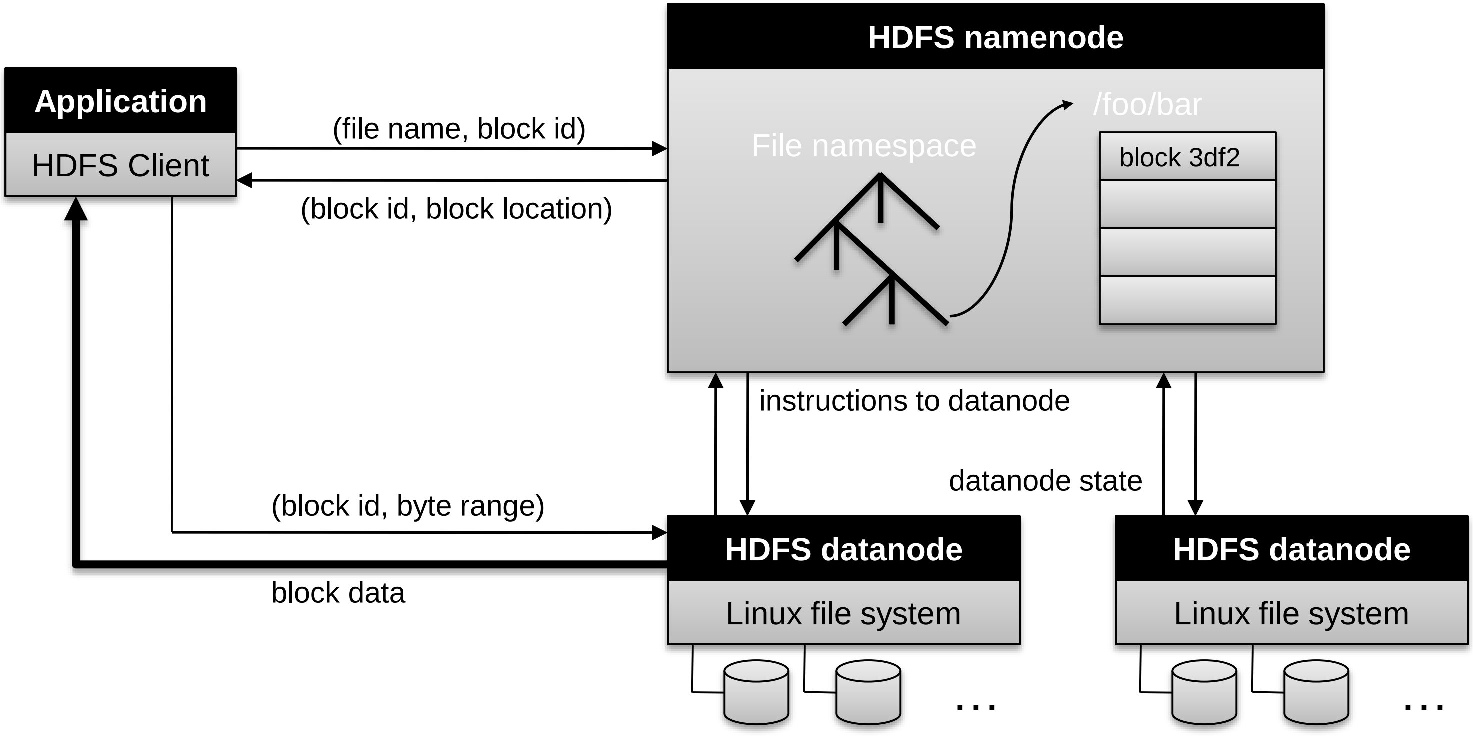
Not every file you need to store is an exact multiple of your system's block size, so the final data block for a file uses only as much space as is needed. In the case of Figure 4-1, the final block of data is 1MB.

From GFS to HDFS

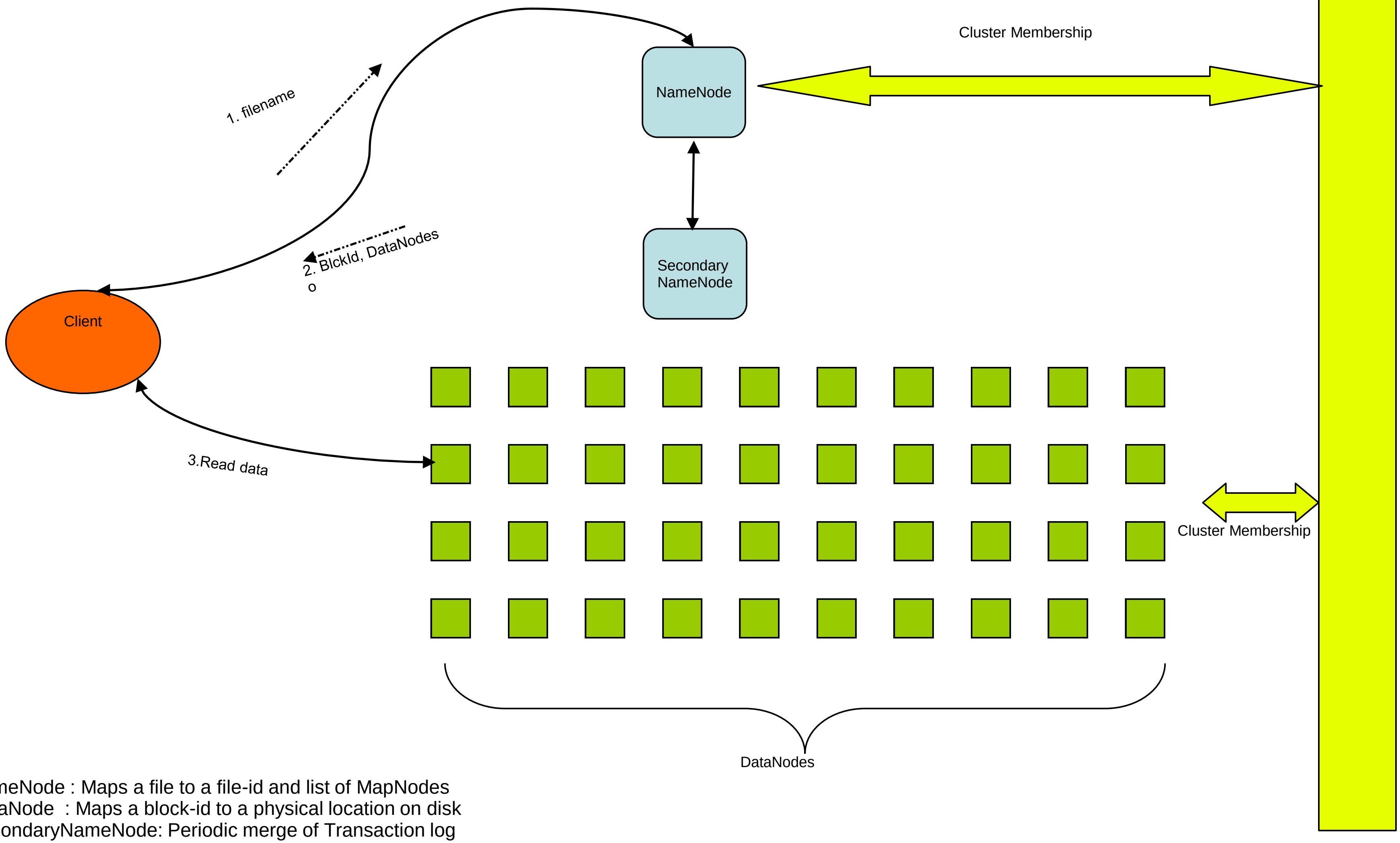
- Terminology differences:
  - GFS master = Hadoop namenode
  - GFS chunkservers = Hadoop datanodes
- Functional differences:
  - HDFS performance is (likely) slower

For the most part, we'll use the Hadoop terminology…

HDFS Working Flow



HDFS Architecture

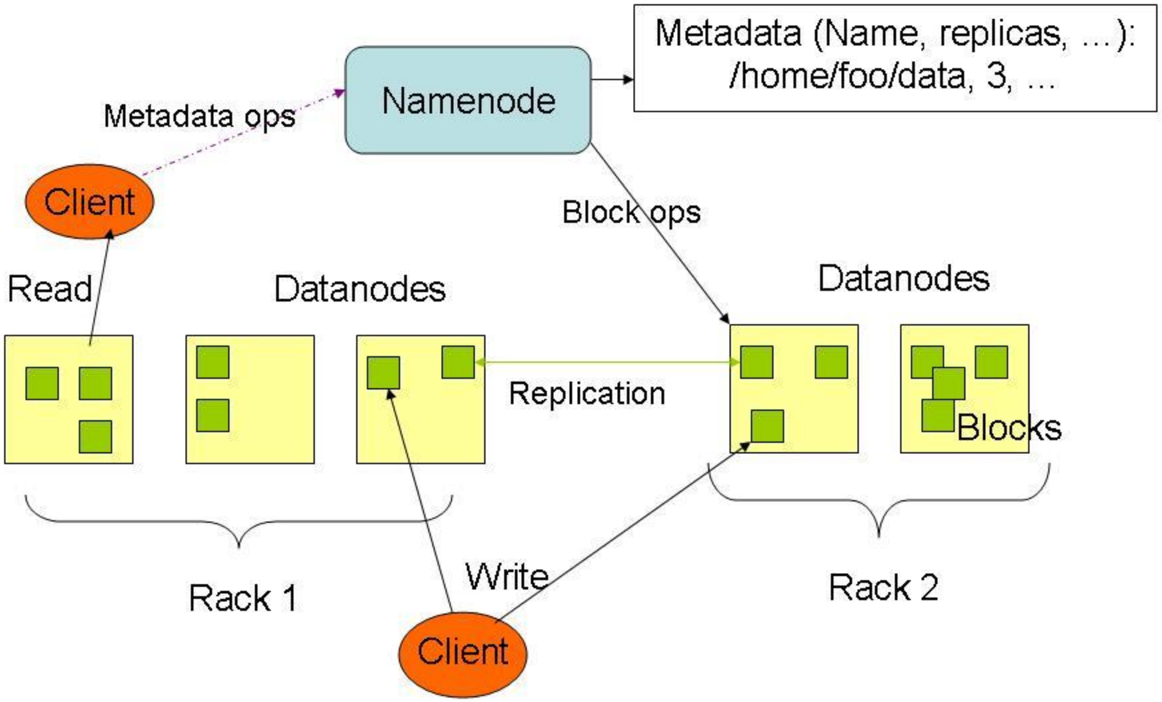


NameNode : Maps a file to a file-id and list of MapNodes
DataNode : Maps a block-id to a physical location on disk
SecondaryNameNode: Periodic merge of Transaction log

Distributed File System

- **Single Namespace for entire cluster**
- **Data Coherency**
  - Write-once-read-many access model
  - Client can only append to existing files
- **Files are broken up into blocks**
  - Typically 128 MB block size
  - Each block replicated on multiple DataNodes
- **Intelligent Client**
  - Client can find location of blocks
  - Client accesses data directly from DataNode

HDFS Architecture



NameNode Metadata

- **Meta-data in Memory**
  - The entire metadata is in main memory
  - No demand paging of meta-data
- **Types of Metadata**
  - List of files
  - List of Blocks for each file
  - List of DataNodes for each block
  - File attributes, e.g creation time, replication factor
- **A Transaction Log**
  - Records file creations, file deletions. etc

Namenode Responsibilities

- Managing the file system namespace:
  - Holds file/directory structure, metadata, file-to-block mapping, access permissions, etc.
- Coordinating file operations:
  - Directs clients to datanodes for reads and writes
  - No data is moved through the namenode
- Maintaining overall health:
  - Periodic communication with the datanodes
  - Block re-replication and rebalancing
  - Garbage collection

DataNode

- **A Block Server**
  - Stores data in the local file system (e.g. ext3)
  - Stores meta-data of a block (e.g. CRC)
  - Serves data and meta-data to Clients
- **Block Report**
  - Periodically sends a report of all existing blocks to the NameNode
- **Facilitates Pipelining of Data**
  - Forwards data to other specified DataNodes

Block Placement

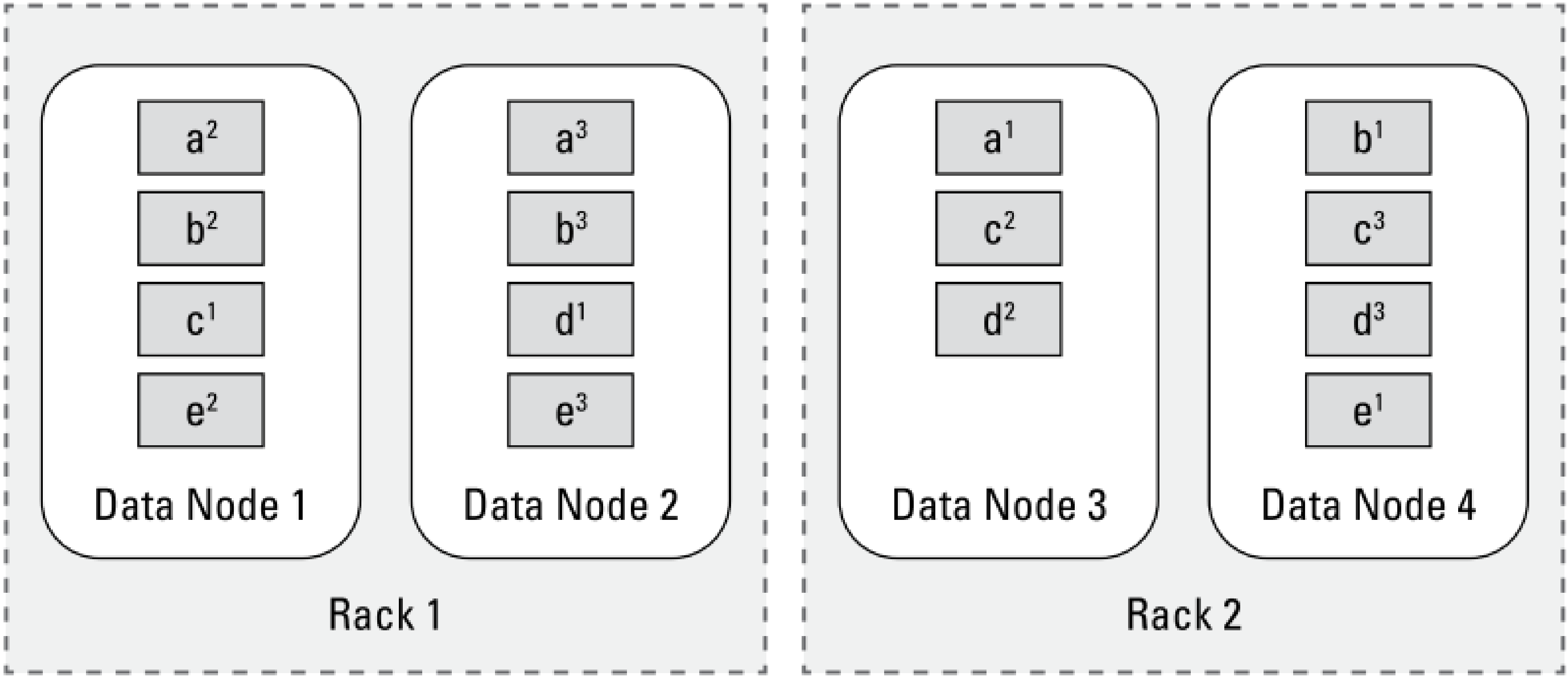
- **Current Strategy**

  -- One replica on local node

  -- Second replica on a remote rack

  -- Third replica on same remote rack

  -- Additional replicas are randomly placed
- **Clients read from nearest replica**



The file shown in Figure 4-2 has five data blocks, labeled a, b, c, d, and e. If you take a closer look, you can see this particular cluster is made up of two racks with two nodes apiece, and that the three copies of each data block have been spread out across the various slave nodes.

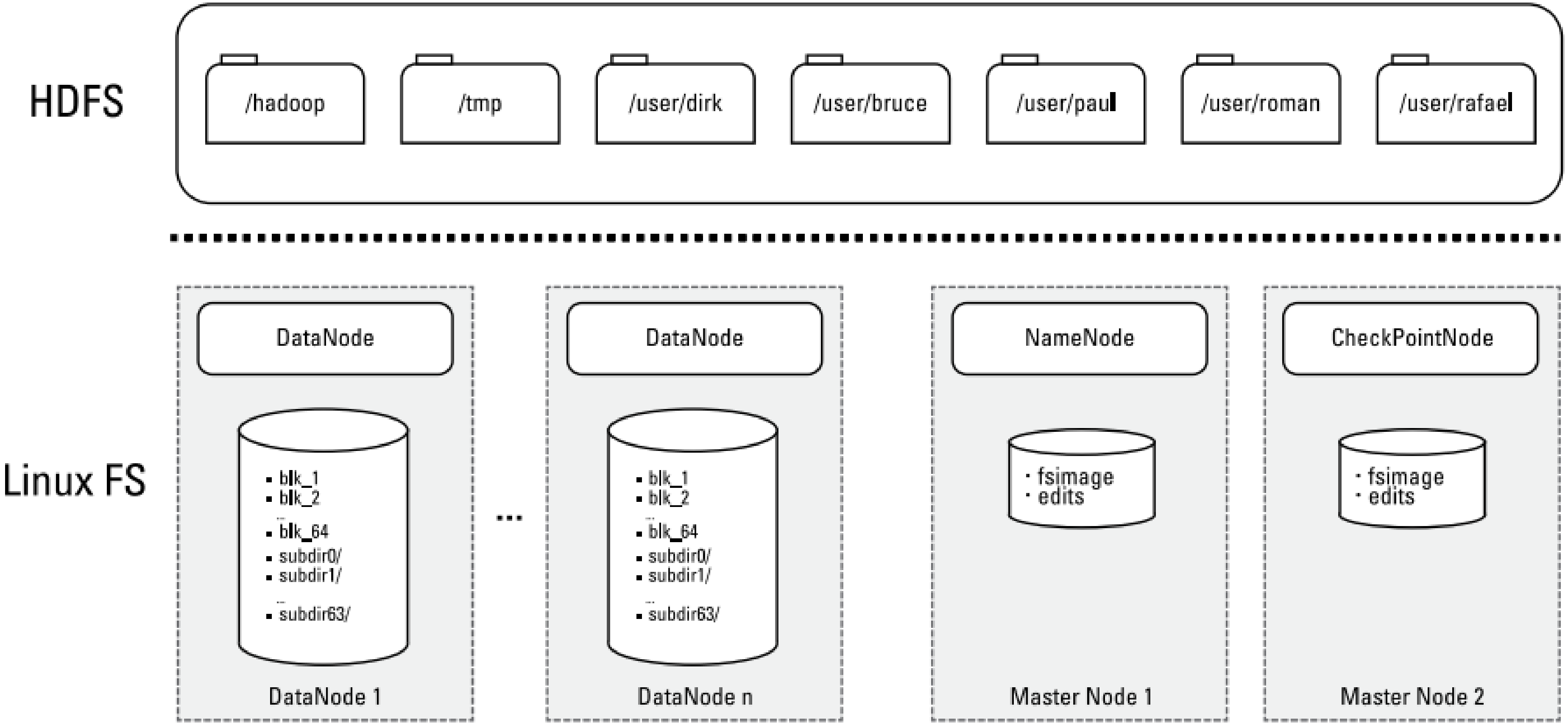


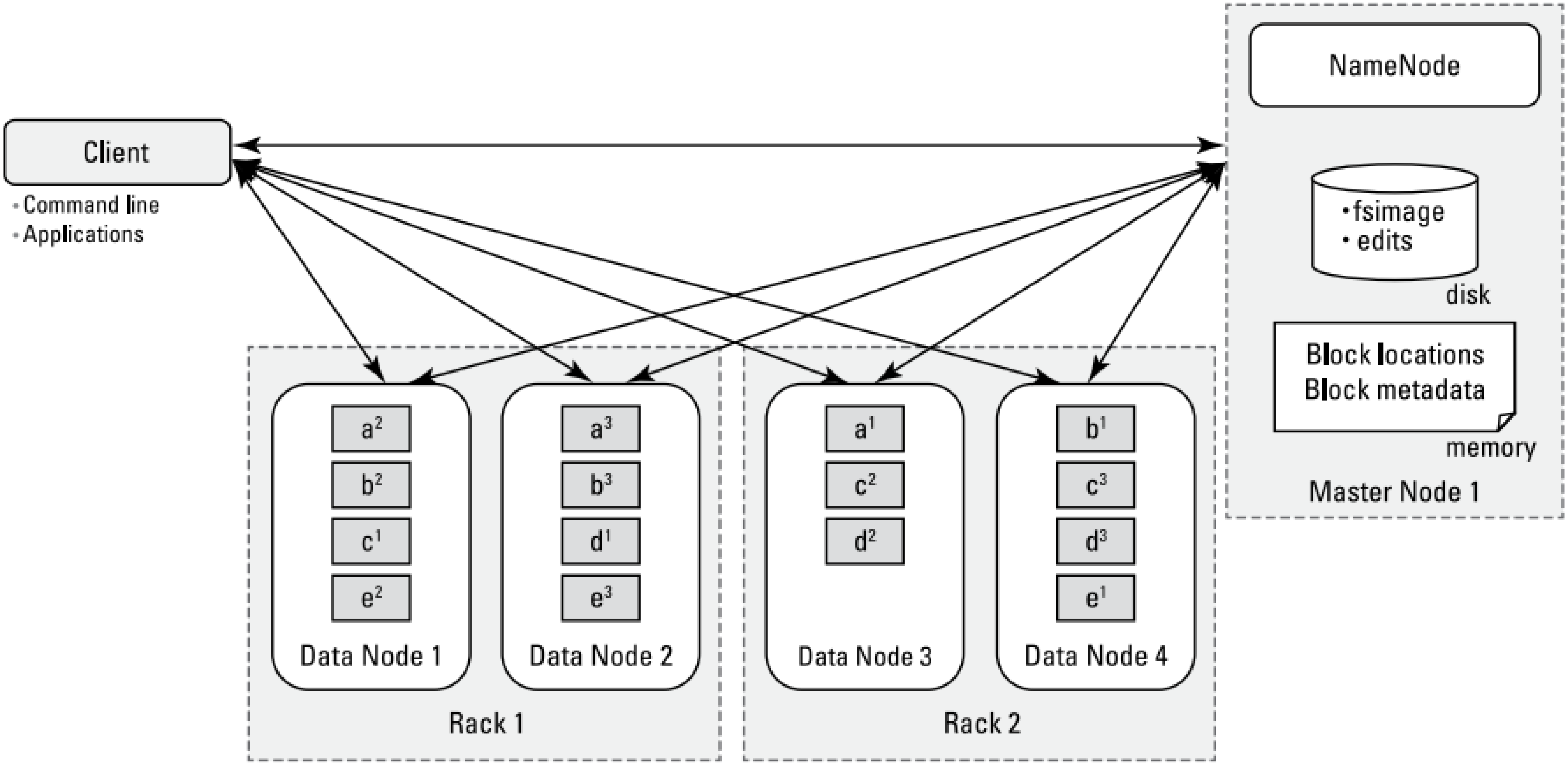
Figure 4-3 shows that a Hadoop cluster is made up of two classes of servers: *slave nodes*, where the data is stored and processed, and *master nodes*, which govern the management of the Hadoop cluster. On each of the master nodes and slave nodes, HDFS runs special services and stores raw data to capture the state of the file system. In the case of the slave nodes, the raw data consists of the blocks stored on the node, and with the master nodes, the raw data consists of metadata that maps data blocks to the files stored in HDFS.

Data Correctness

- **Use Checksums to validate data**
  - Use CRC32
- **File Creation**
  - Client computes checksum per 512 byte
  - DataNode stores the checksum
- **File access**
  - Client retrieves the data and checksum from DataNode
  - If Validation fails, Client tries other replicas

NameNode Failure

- **A single point of failure**
- **Transaction Log stored in multiple directories**
  - A directory on the local file system
  - A directory on a remote file system (NFS/CIFS)
- **Need to develop a real HA solution**

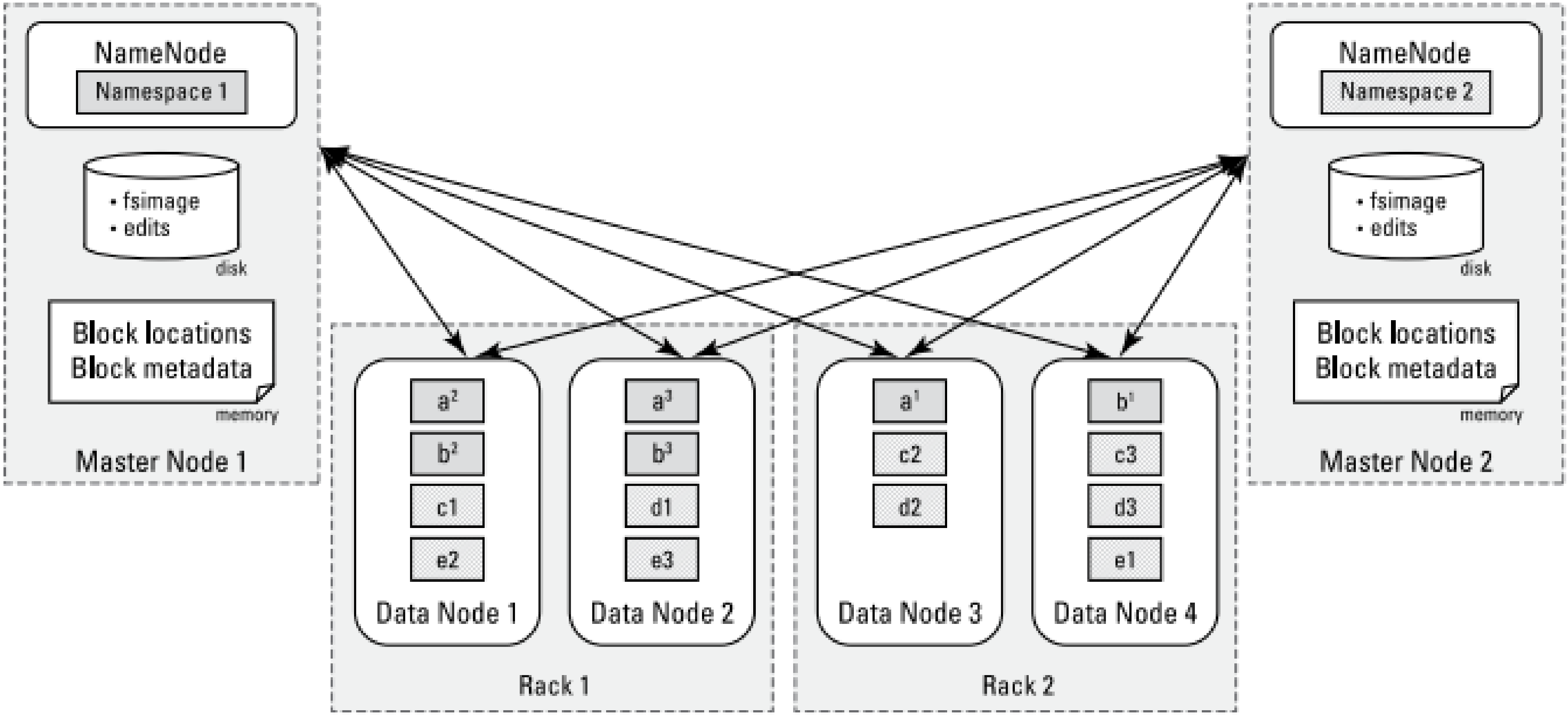


Throughout the life of the cluster, the DataNode daemons send the NameNode heartbeats (a quick signal) every three seconds, indicating they're active. (This default value is configurable.) Every six hours (again, a configurable default), the DataNodes send the NameNode a block report outlining which file blocks are on their nodes. This way, the NameNode always has a current view of the available resources in the cluster.

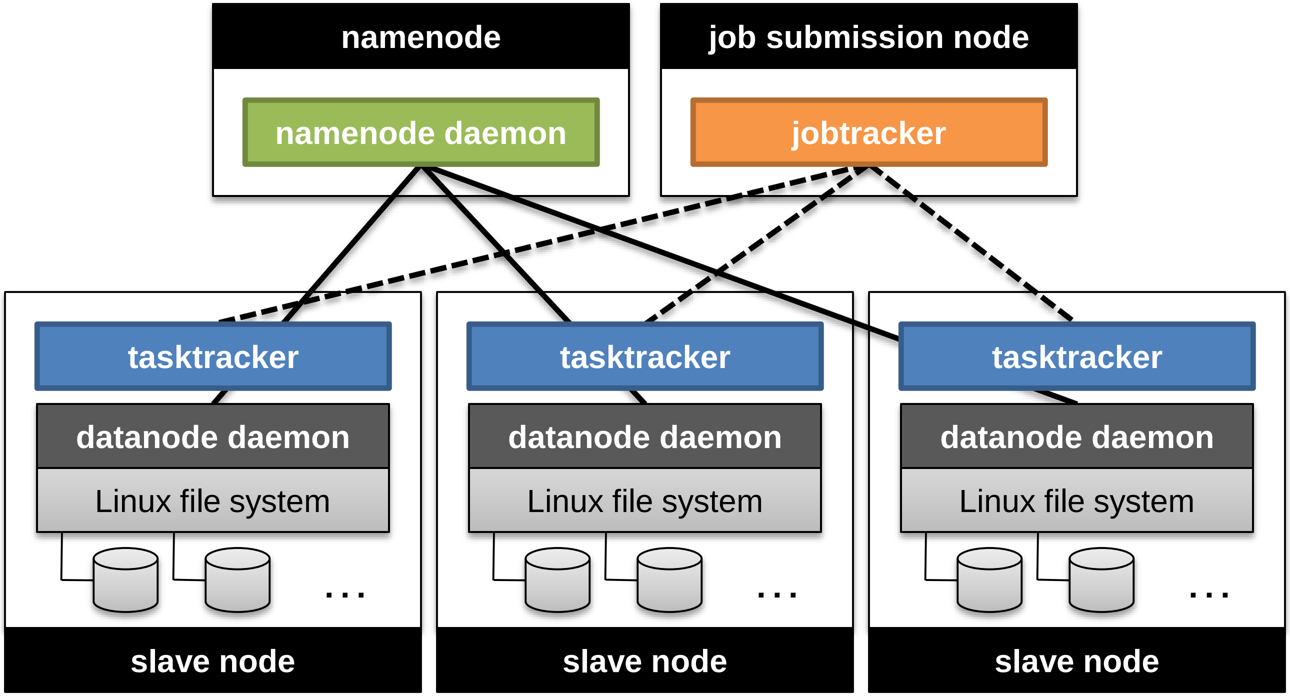
HDFS Federation

- **Before Hadoop 2 NameNode placed limits on the degree to which Hadoop clusters could scale**
- **Few clusters were able to scale beyond 3,000 or 4,000 nodes**
- **The solution to expanding Hadoop clusters indefinitely is to federate the NameNode**
- **Have multiple NameNode**
  - Each NameNode responsible only for the file blocks in its own name space
  - The slave nodes all contain blocks from both name spaces

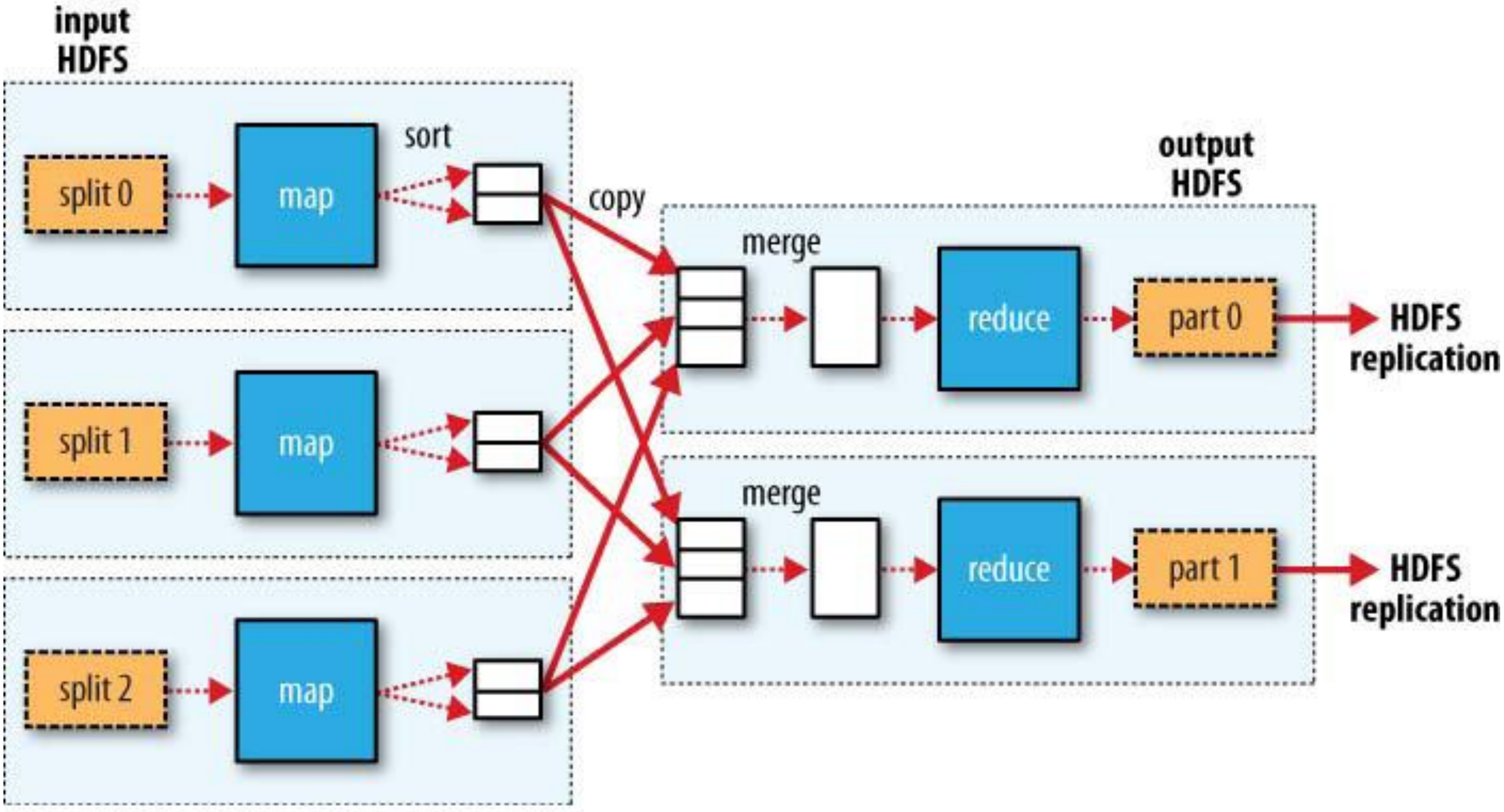
HDFS Federation

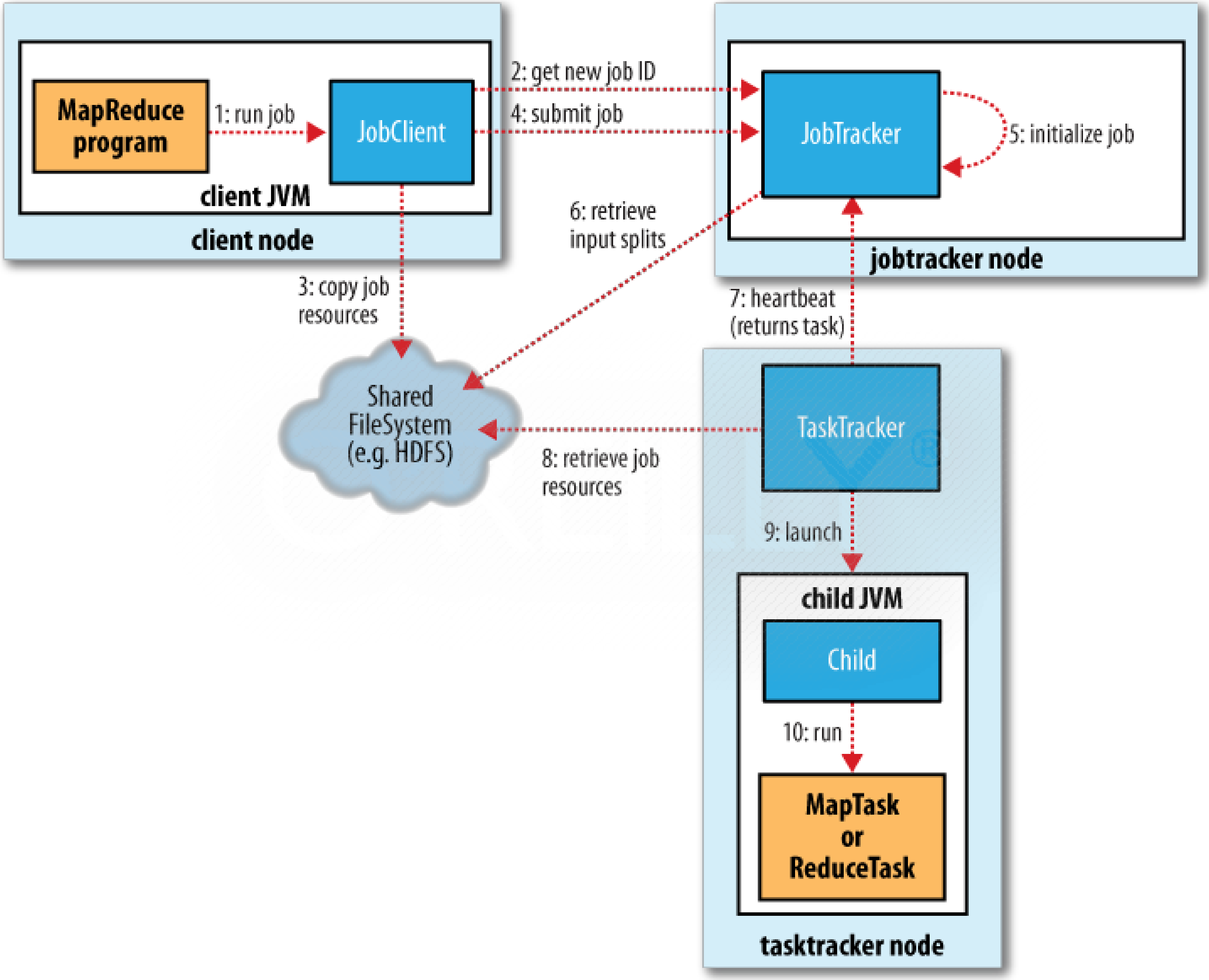


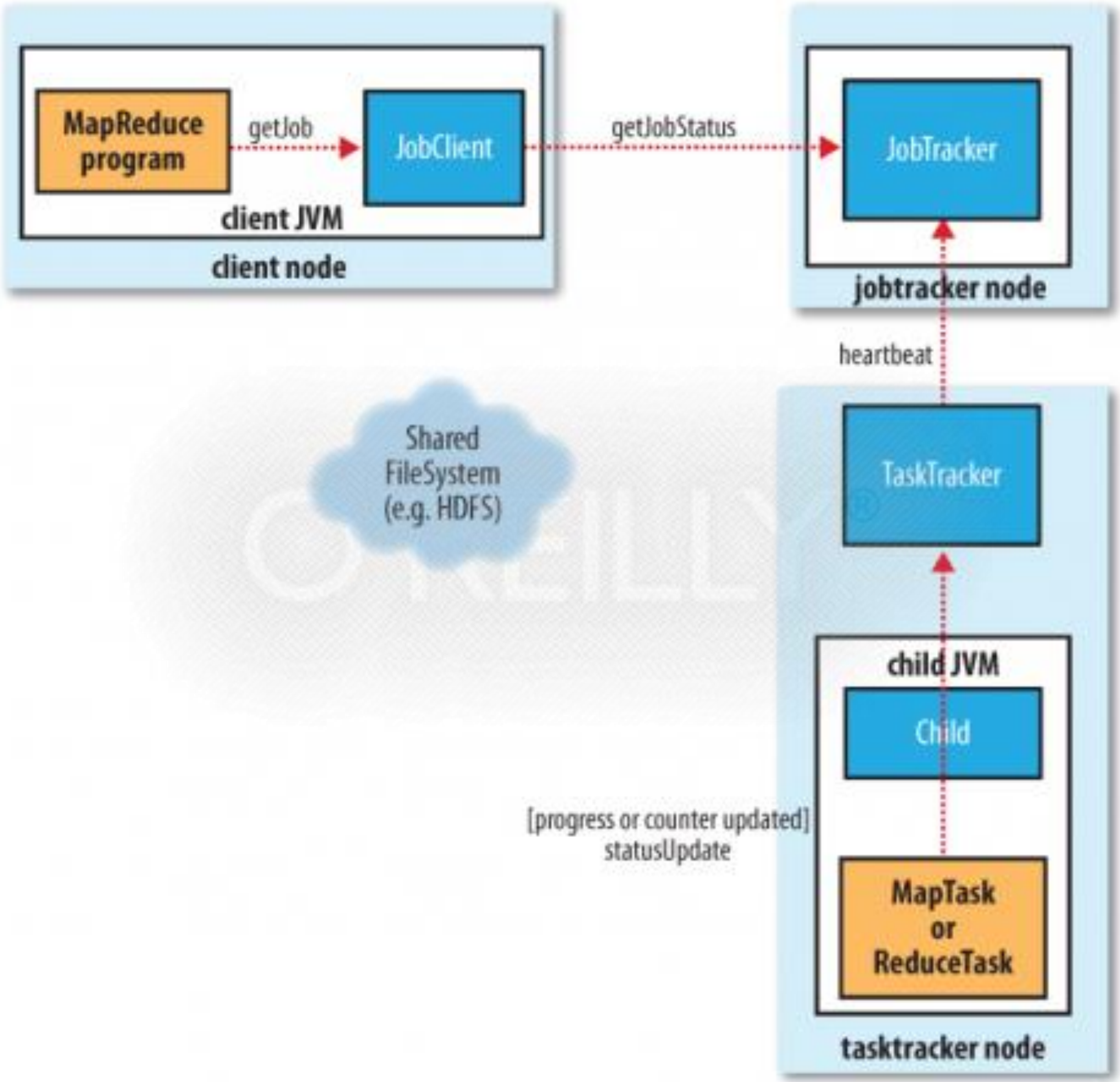
Putting everything together…



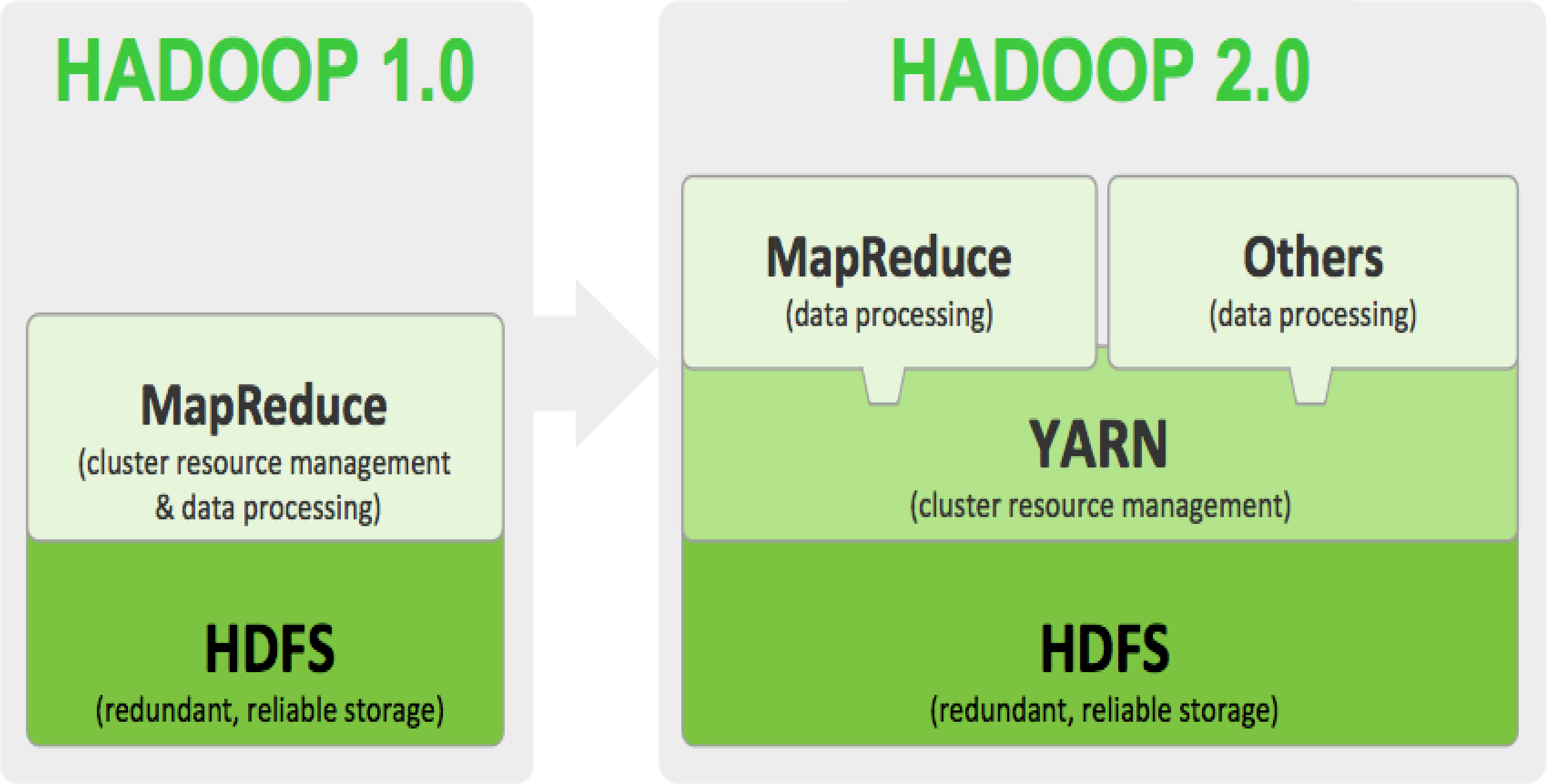
MapReduce Data Flow




MapReduce program
1: run job
JobClient
client JVM
client node
2: get new job ID
4: submit job
JobTracker
5: initialize job
jobtracker node
3: copy job resources
6: retrieve input splits
7: heartbeat (returns task)
Shared FileSystem (e.g. HDFS)
TaskTracker
8: retrieve job resources
9: launch
child JVM
Child
10: run
MapTask or ReduceTask
tasktracker node


MapReduce program
getJob
JobClient
client JVM
client node
getJobStatus
JobTracker
jobtracker node
heartbeat
Shared FileSystem (e.g. HDFS)
TaskTracker
child JVM
Child
[progress or counter updated]
statusUpdate
MapTask or ReduceTask
tasktracker node

Hadoop 1 vs. Hadoop 2


HADOOP 1.0
MapReduce
(cluster resource management
& data processing)
HDFS
(redundant, reliable storage)
HADOOP 2.0
MapReduce
(data processing)
Others
(data processing)
YARN
(cluster resource management)
HDFS
(redundant, reliable storage)

Hadoop 1 vs. Hadoop 2

- HDFS federation
  - In Hadoop 1, a single Namenode managed the entire namespace for a Hadoop cluster
  - With HDFS federation, multiple Namenode servers manage namespaces and this allows for horizontal scaling

- resource manager YARN
  - YARN is a resource manager that was created by separating the processing engine and resource management capabilities of MapReduce as it was implemented in Hadoop 1
  - brings significant performance improvements for some applications, supports additional processing models, and implements a more flexible execution engine

Happy Hadooping!