Basics of Cloud Computing – Lecture 4

Introduction to MapReduce

Pelle Jakovits

Satish Srirama

Outline

- Economics of Cloud Providers recap
- MapReduce model
- Hadoop MapReduce framework
- Hadoop Distributed File System (HDFS)
- Hadoop v2.0: YARN

Economics of Cloud Providers – Failures

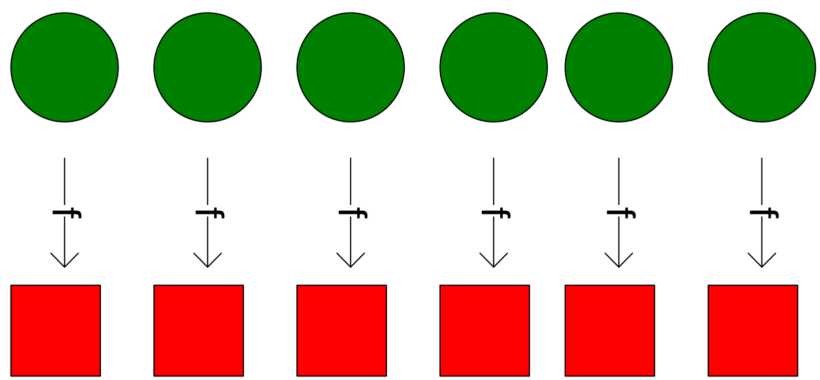
- Cloud computing can greatly simplify large scale data processing by providing virtually unlimited access to storage and computing resources
- However, Cloud Computing brought a shift from highly reliable servers to commodity servers
- High number of servers means that failures are common.
  - Software must adapt to failures
- Solution: Replicate data and computations
  - Distributed File System & MapReduce
- MapReduce = functional programming meets distributed processing on steroids
  - Not a new idea - dates to the 50's

Functional Programming -> MapReduce

- Two important concepts in functional programming:
  - **Map:** Apply a user defined function on every element of the list
  - **Fold:** Apply a user defined aggregation function on a list to reduce it into a single value

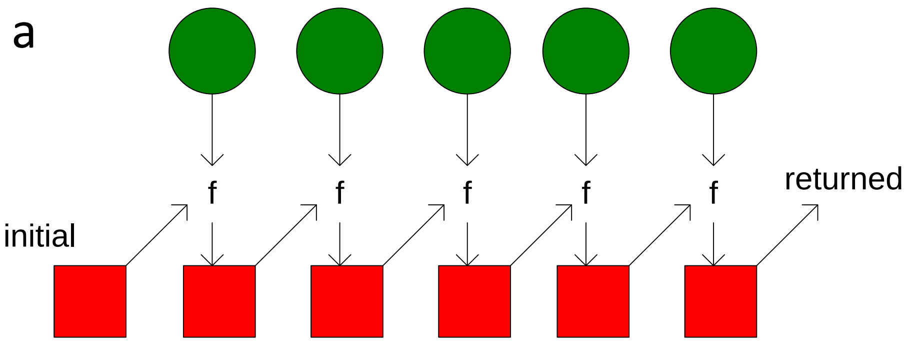
Map

- Map is a higher-order function
  - **map f Lst:** (a->b) -> [a] -> [b]
    - **f** - user defined function
    - **Lst** - a list of values
- Function **f** is applied to every element in the **input list**
- Result is a **new list**



Fold (FoldL)

- Fold is also a higher-order function
- **fold f x Lst**: (a -> b -> a) -> a -> [b] -> a
  - **f** - user defined function
  - **x** – initial accumulator value
  - **Lst** - a list of values



1. **Accumulator** is set to initial value **x**
2. Function **f** is applied to the first **list element** and the current value of the **accumulator**
3. Result is stored in the **accumulator**
4. Steps 2 and 3 are repeated for every following **item in the list**
5. The final value in the **accumulator** is returned as the result

Map/Fold in Action

- Simple map example:

```
square x = x * x
map square [1,2,3,4,5] → [1,4,9,16,25]
```

- Fold examples:

```
fold (+) 0 [1,2,3,4,5] → 15
fold (*) 1 [1,2,3,4,5] → 120
```

- Sum of squares:

```
fold (+) 0 (map square [1,2,3,4,5])) ->
fold (+) 0 [1,4,9,16,25] -> 55
```

Implicit parallelism

- In a purely functional setting, operations inside **map** can be performed independently and easily parallelised
  - We can partition the input list between multiple computers
  - We can apply the **map** operation on each partition separately
  - **Fold** provides a mechanism for combining **map** results back together
- If function **f** is ***associative***, then we can also compute **fold** operations on different partitions independently

```
f(f(a, b), c) = f(a, f(b, c))

(a + b) + c = a + (b + c)

fold f x [1,2,3,4] = f(fold f x [1,2], fold f x [3,4])
```

- This is the *"implicit parallelism"* of functional programming that MapReduce aims to exploit

Typical Large-Data Problem

Map

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

Reduce

Key idea: provide a functional abstraction for these two operations – MapReduce

MapReduce

- Designed for processing very large-scale data (Petabytes)
- Deployed on a large cluster of servers
- Horizontally scalable - Add more servers to handle more data and speed up processing
- Data is stored in a distributed file system and replicated
- Parallelism is achieved by executing many Map and Reduce tasks concurrently

- **NB!** MapReduce **map** and **reduce** functions are not exactly the same as **map** and **fold** functions from functional programming!

MapReduce model

- Input is a list of Key and Value pairs: [(k, v)]
  - For example: (**LineNr**, **LineString**) when processing text files
- Programmers only have to specify two functions:

  map (k, v) → (k', v')*

  reduce (k', [v']) → (k'', v'')*
- The execution framework handles everything else:
  - Data partitioning, distribution, synchronization, fault recovery, etc.

Map function

- Map function is applied to every (Key, Value) pair in the input list
- Input to the user defined map functions is a single (Key, Value) pair
- Output is zero or more key and value pairs.
  - In functional programming, map function always had to return exactly one value.

map (k, v) → (k', v')*

Reduce function

- All KeyPairs produced in the map step are grouped by the keys and values are combined into a list
  - This happens between Map and Reduce stages
- Input to a reduce function is a **unique key** and a list of **values**: (Key, [Value])
- Reduce function is applied on the key and list of values
  - Typically an aggregation function is applied on the list
- Output is zero or more key and value pairs

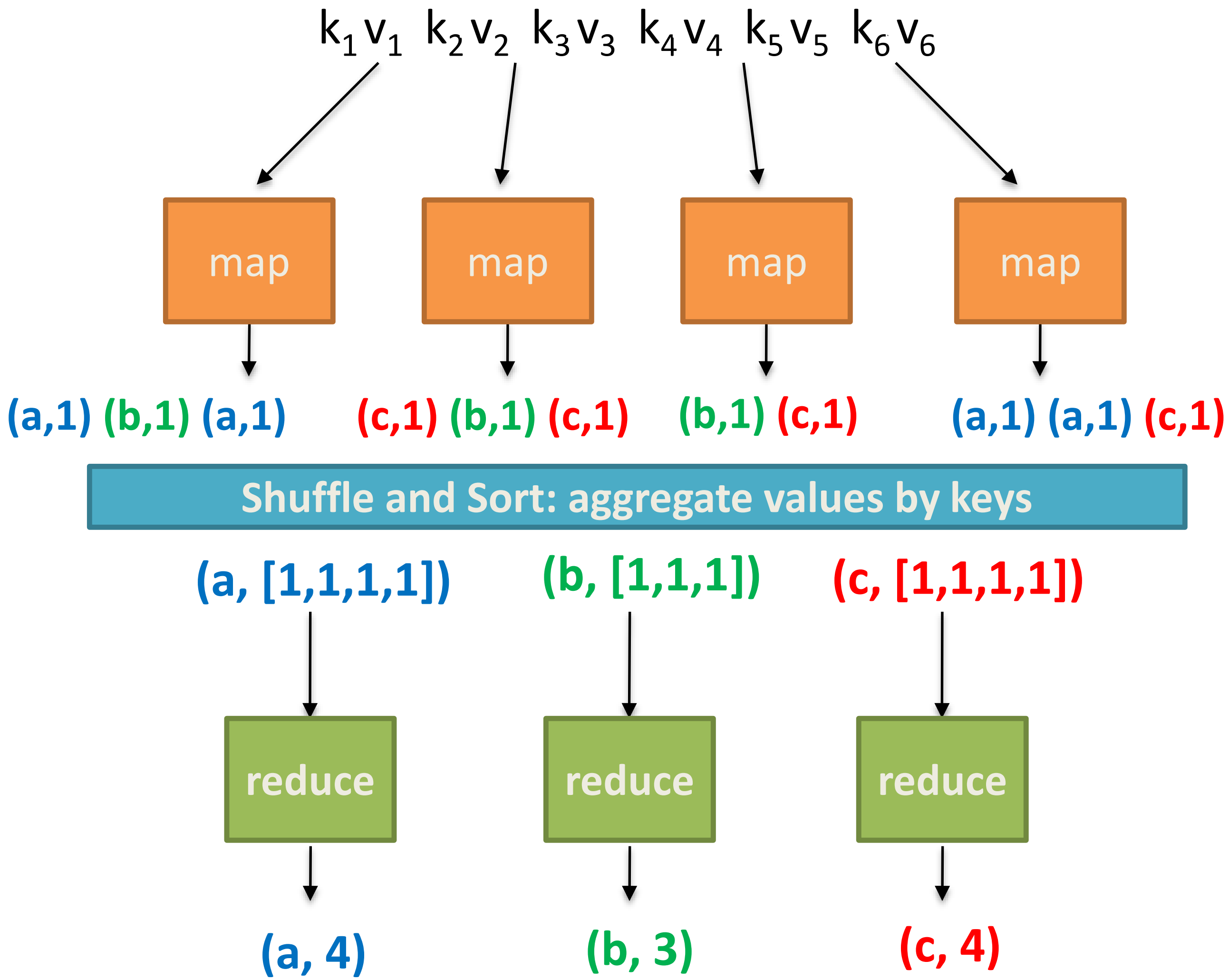
reduce (k’, [v’]) → (k’’, v’’)*

Example: Word Count

```
Map(String lineNr, String line):
    for each word w in line:
        Emit(w, 1);

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

MapReduce



k1 v1 k2 v2 k3 v3 k4 v4 k5 v5 k6 v6
map
map
map
map
(a,1) (b,1) (a,1)
(c,1) (b,1) (c,1)
(b,1) (c,1)
(a,1) (a,1) (c,1)
Shuffle and Sort: aggregate values by keys
(a, [1,1,1,1])
(b, [1,1,1])
(c, [1,1,1,1])
reduce
reduce
reduce
(a, 4)
(b, 3)
(c, 4)

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 “data distribution”
  - Partition and replicate data
  - Moves processes to data
- Handles scheduling
  - Assigns workers to map and reduce tasks
- Handles synchronization
  - Gathers, sorts, and shuffles intermediate data
- Handles errors and faults
  - Detects worker failures and automatically restarts
- Handles speculative execution
  - Detects “slow” workers and re-executes work

Sounds simple, but many challenges!

MapReduce - extended

- 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…

- Users can change the behaviour of the framework by overwriting certain components, such as:
  – **Partitioner**
  – **Combiner**

Partitioner

- Controls how the key space is distributed into different partitions.

  `Partition(key, number of partitions) → partition for key`
- A simple hash of the key can be used, e.g.:

  `Partition(Key, partitions) = hashFun(Key) mod partitions`
- Programmers can overwrite paritioner function to force specific keys to be located in the same parition
  - Example: **Global sorting order**
  - Partition keys that start with **0** into partition **0**, **1** into **1**, **2** into **2**, …, **9** into **9**.
  - Makes sure that every previous partition has smaller keys than the next one.

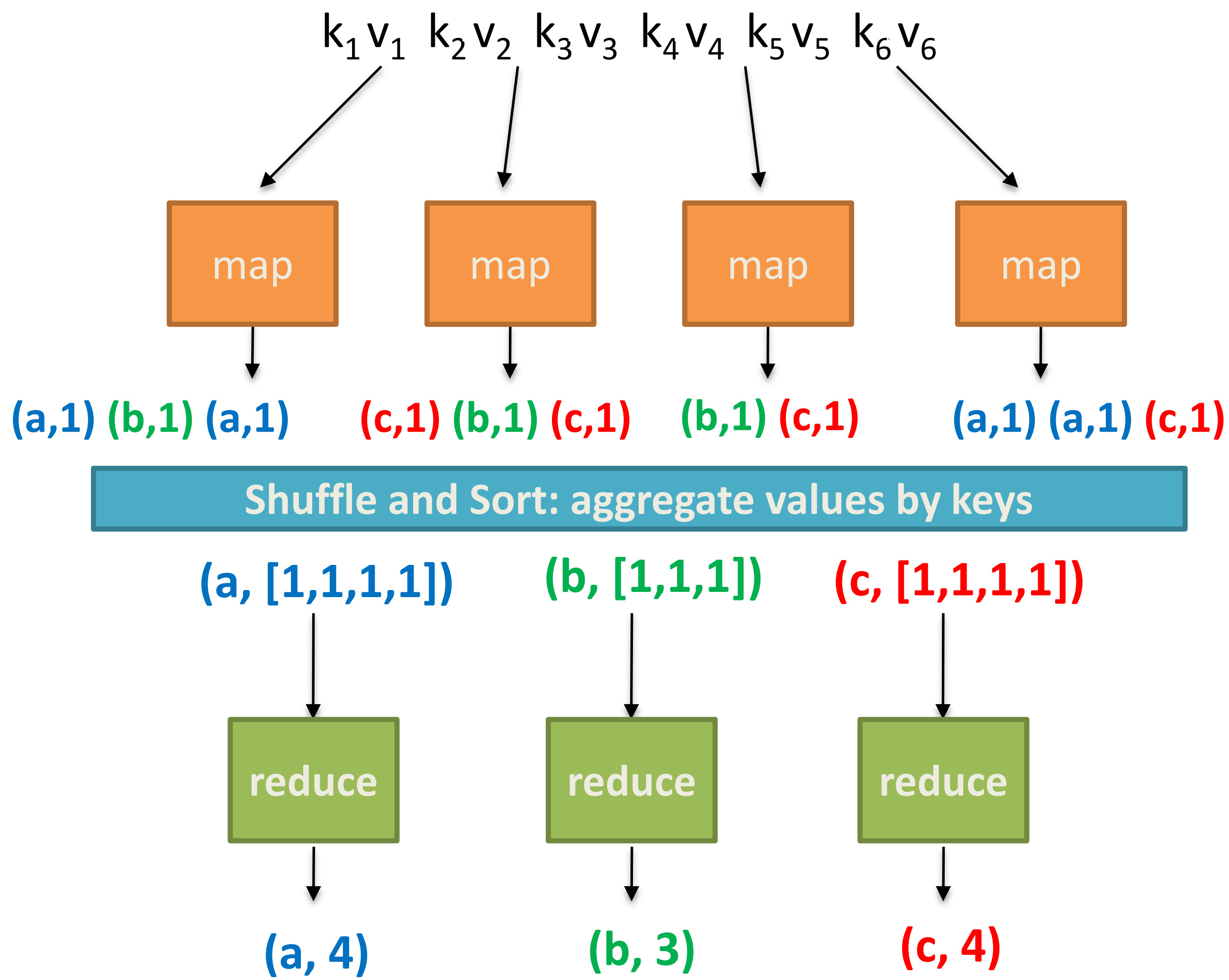
Combiner

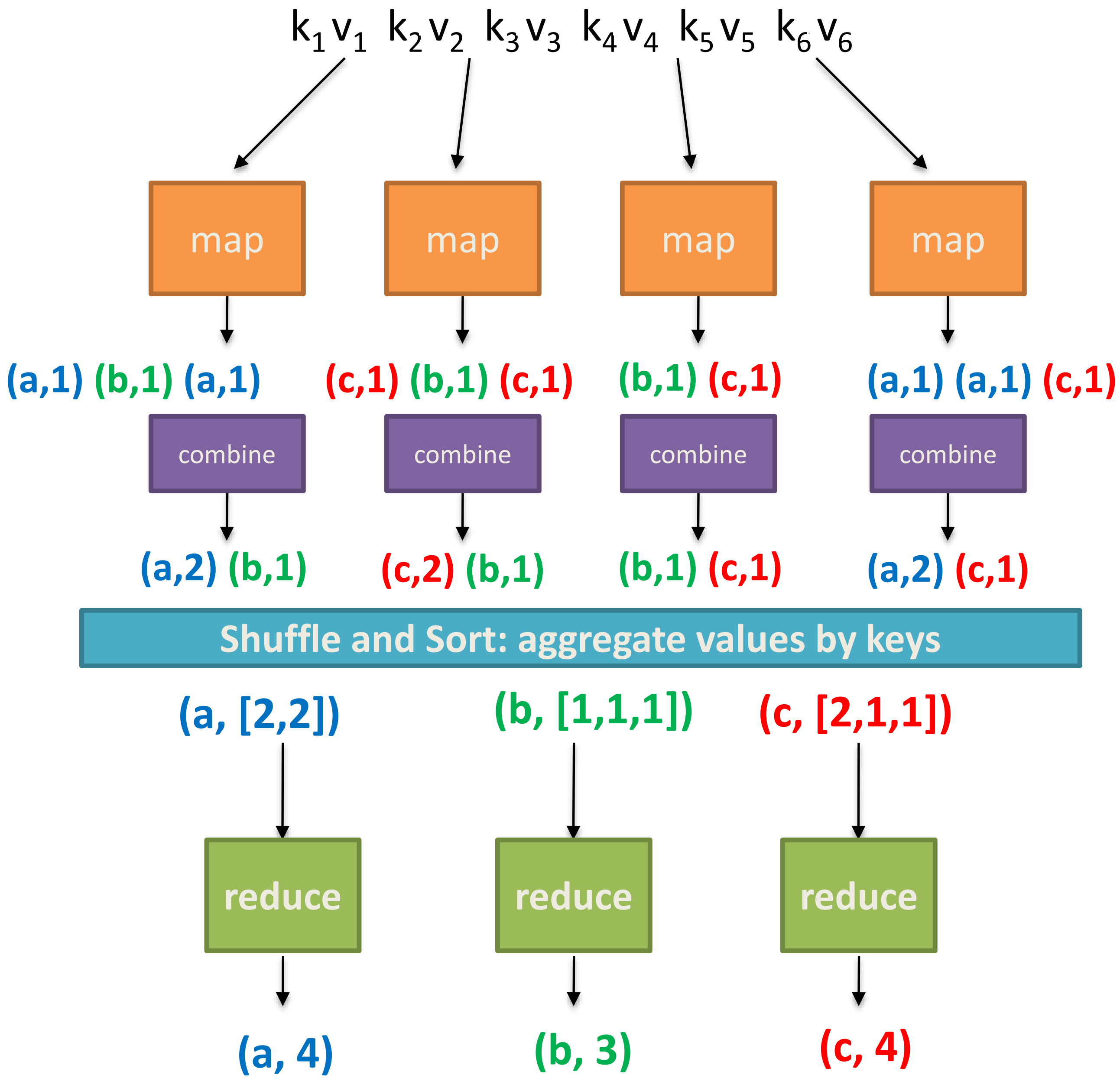
- In many cases, Map stage produces too many KeyValue pairs.
  – WordCount produces a KeyValue for every single word.
- Combine can be used as a mini-reducer stage that runs in memory after the map phase

```
combine (k', [v']) → (k'', v'')*
```

- Used as an optimization to reduce network traffic
- Often the Reduce function can be used directly as a combiner
  – But not always!
  – Why?

WordCount without Combiner

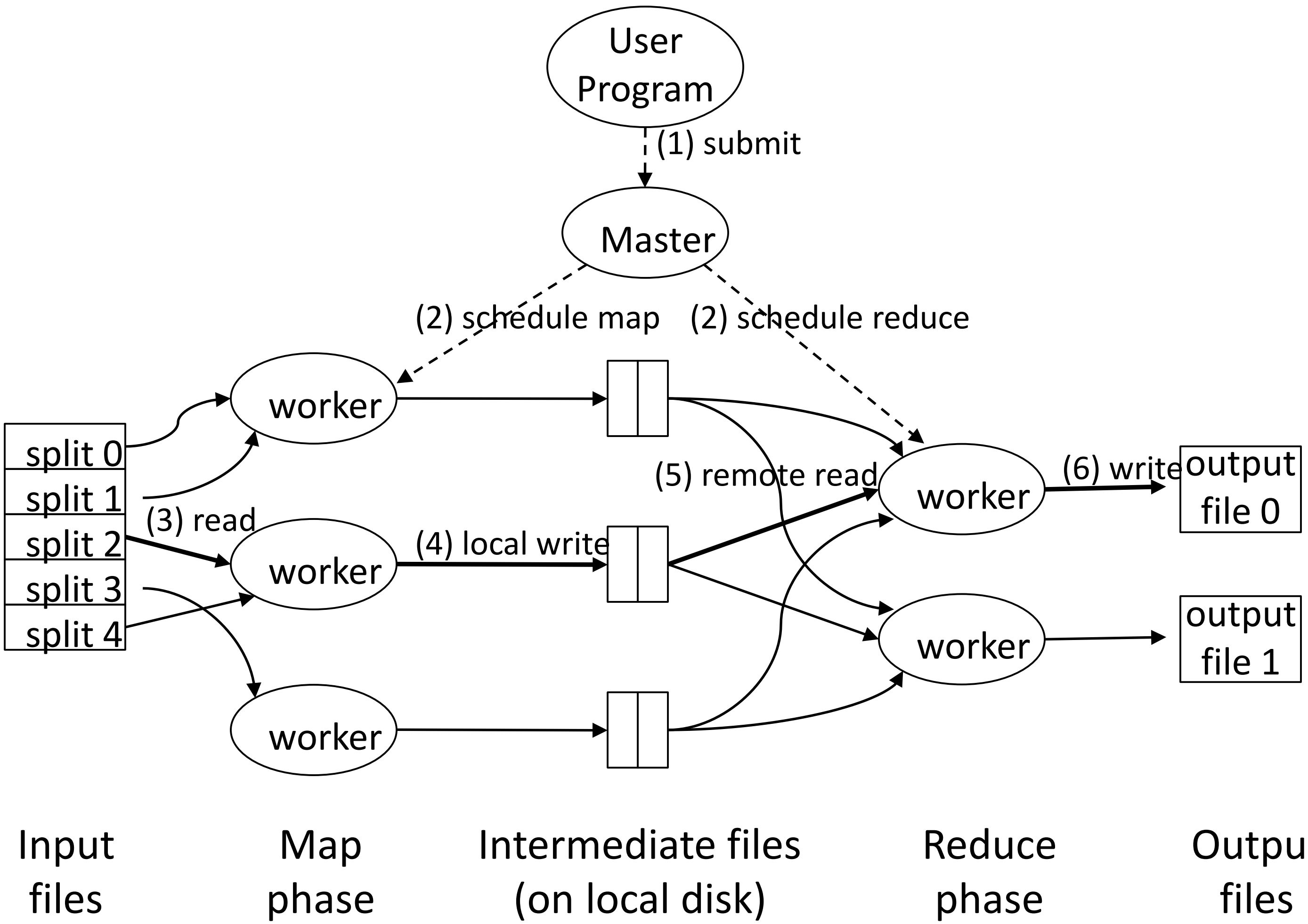



k1 v1 k2 v2 k3 v3 k4 v4 k5 v5 k6 v6
map
map
map
map
(a,1) (b,1) (a,1)
(c,1) (b,1) (c,1)
(b,1) (c,1)
(a,1) (a,1) (c,1)
combine
combine
combine
combine
(a,2) (b,1)
(c,2) (b,1)
(b,1) (c,1)
(a,2) (c,1)
Shuffle and Sort: aggregate values by keys
(a, [2,2])
(b, [1,1,1])
(c, [2,1,1])
reduce
reduce
reduce
(a, 4)
(b, 3)
(c, 4)

Two more important details…

- Barrier between map and reduce phases
  - Reduce tasks wait until map tasks are finished
  - But we can begin copying intermediate data earlier
- Data is ordered by keys before reduce operation is performed
  - But no enforced ordering *across* reducers
    - Unless using a specially designed partitioner!

MapReduce Overall Architecture



Adapted from (Dean and Ghemawat, OSDI 2004)

MapReduce can refer to…

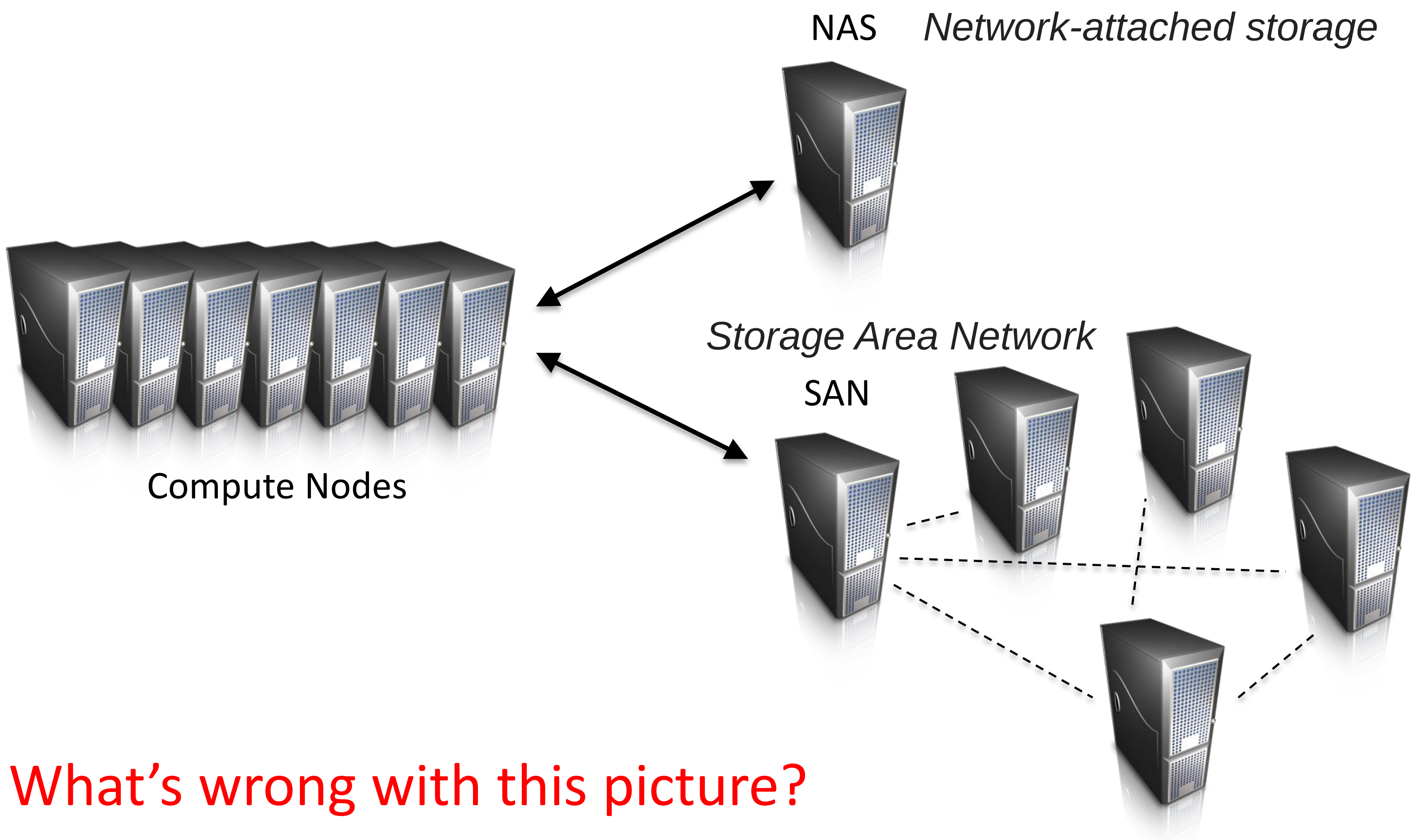
- The programming model
- The execution framework (aka “runtime”)
- The specific implementation

Usage is usually understandable from context!

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 implementations
  - For GPUs, cell processors, etc.
  - MapReduce as database query engine (CouchDB)

Cloud Computing Storage: How to move data to the workers?



What's wrong with this picture?

Managing Peta-scale data

- Network bandwidth is limited
- Not enough RAM to hold all the data in memory
- Disk access is slow, but disk throughput is reasonable
- 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 where the data is locally stored
- A distributed file system is the answer
  - GFS (Google File System) for Google’s MapReduce
  - HDFS (Hadoop Distributed File System) for Hadoop

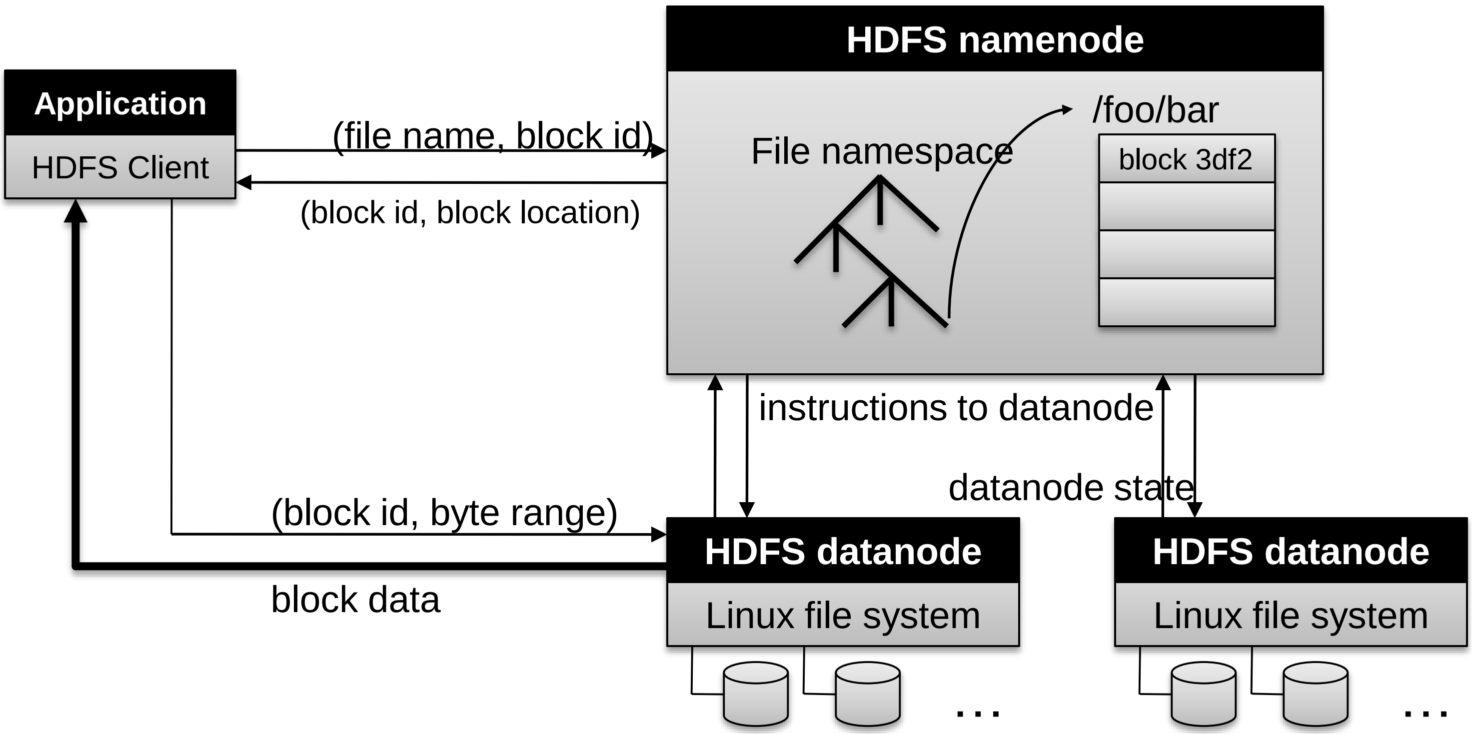
Distributed File System - Assumptions

- Choose cheaper 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

Distributed File System - Design Decisions

- Files stored as chunks
  - Fixed size (64MB)
- Reliability through replication
  - Each chunk replicated across 3+ nodes
- 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)
- Hadoop Distributed File System (HDFS) is the implementation of Google File System (GFS)

HDFS Architecture



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

Hadoop Processing Model

- Create or allocate a cluster
- Put data onto the file system
  - Data is split into blocks
  - Replicated and stored in the cluster
- Run your job
  - Copy Map code to the allocated nodes
    - Move computation to data, not data to computation
  - Gather output of Map, sort and partition on key
  - Run Reduce tasks
- Results are stored in the HDFS

MapReduce Terminology

- **Job** – A “full program” - an execution of a Mapper and Reducer across a data set
- **Task** – An execution of a Mapper or a Reducer on a data chunk
  – Also known as Task-In-Progress (TIP)
- **Task Attempt** – A particular instance of an attempt to execute a task on a machine

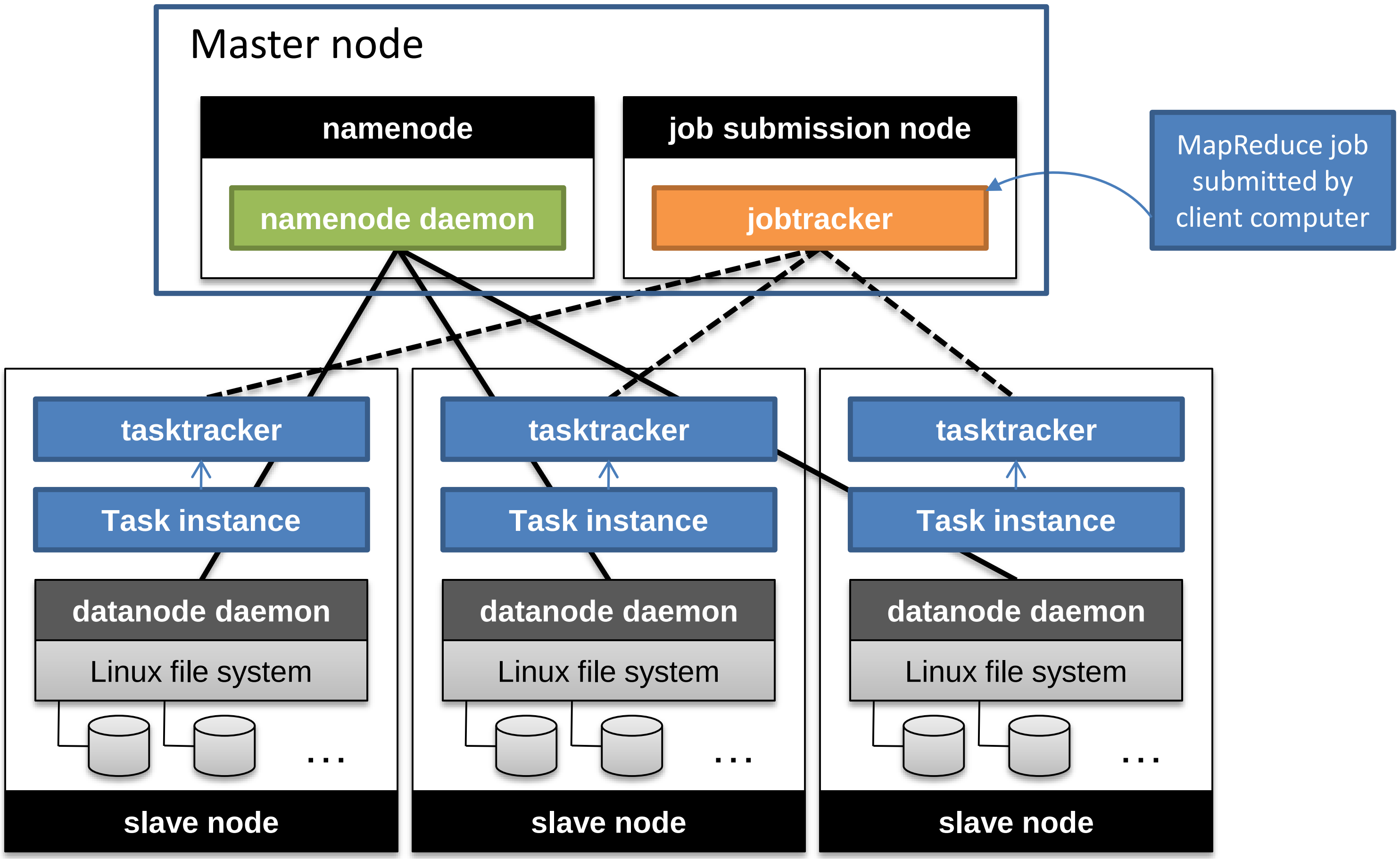
Terminology example

- Running “Word Count” across 20 files is one **job**
- 20 files to be mapped imply 20 map **tasks**
  + some number of reduce **tasks**
- At least 20 **map task attempts** will be performed
  - more if a machine crashes or slow, etc.

Task Attempts

- A particular task will be attempted at least once, possibly more times if it crashes
  - If the same input causes crashes over and over, that input will eventually be abandoned
- Multiple attempts at one task may occur in parallel with speculative execution turned on

Hadoop MapReduce Architecture : High Level



MapReduce Summary

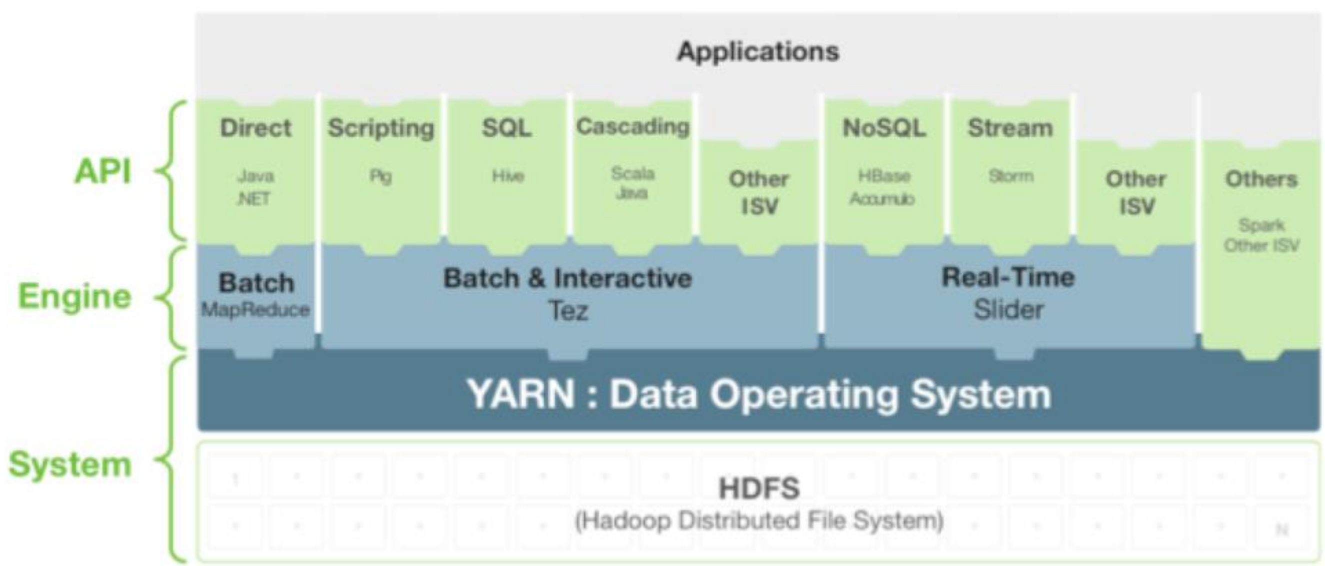
- Simple, but powerful programming model
- Scales to handle petabyte+ workloads
  - Google: six hours and two minutes to sort 1PB (10 trillion 100-byte records) on 4,000 computers
  - Yahoo!: 16.25 hours to sort 1PB on 3,800 computers
- Incremental performance improvement with more nodes
- Seamlessly handles failures, but possibly with performance penalties

Limitations with MapReduce V1

- Master node has too many responsibilities!
- Scalability issues
  - Maximum Cluster Size – 4000 Nodes
  - Maximum Concurrent Tasks – 40000
- Coarse synchronization in Job Tracker
  - Single point of failure
  - Failure kills all queued and running jobs
- Jobs need to be resubmitted by users
  - Restart is very tricky due to complex state
- Problems with resource utilization

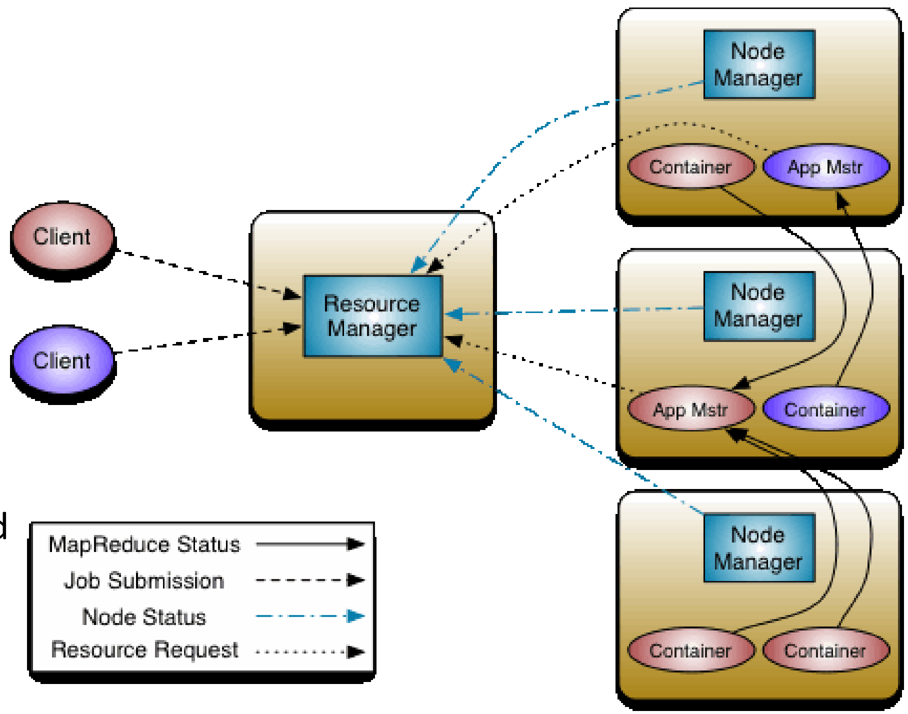
MapReduce NextGen aka YARN aka MRv2

- New architecture introduced in hadoop-0.23
- Divides two major functions of the JobTracker
  - Resource management and job life-cycle management are divided into separate components
- An application is either a single job in the sense of classic MapReduce jobs or a DAG of such jobs



YARN Architecture

- RescourceManager:
  - Arbitrates resources among all the applications in the system
  - Has two main components: Scheduler and ApplicationsManager
- NodeManager:
  - Per-machine slave
  - Responsible for launching the applications' containers, monitoring their resource usage
- ApplicationMaster:
  - Negotiate appropriate resource containers from the Scheduler, tracking their status and monitoring for progress
- Container:
  - Unit of allocation incorporating resource elements such as memory, cpu, disk, network etc.
  - To execute a specific task of the application
  - Similar to map/reduce slots in MRv1



Execution Sequence with YARN

- A **client** program submits the application
- **ResourceManager** allocates a specified container to start the **ApplicationMaster**
- **ApplicationMaster**, on boot-up, registers with **ResourceManager**
- **ApplicationMaster** negotiates with **ResourceManager** for appropriate resource containers
- On successful container allocations, **ApplicationMaster** contacts **NodeManager** to launch the container
- Application code is executed within the container, and then **ApplicationMaster** is responded with the execution status
- During execution, the client communicates directly with **ApplicationMaster** or **ResourceManager** to get status, progress updates etc.
- Once the application is complete, **ApplicationMaster** unregisters with **ResourceManager** and shuts down, freeing its own container process

Next Lab

- Set up IDE for creating Hadoop MapReduce applications
  - Run Hadoop MapReduce code in your computer without installing Hadoop
- Try out MapReduce WordCount example
- Improve the WordCount example

Next Lecture

- We will take a look at different MapReduce algorithms
- Learn how to design MapReduce applications

References

- Hadoop wiki http://wiki.apache.org/hadoop/
- Data-Intensive Text Processing with MapReduce
  - Authors: Jimmy Lin and Chris Dyer
  - http://lintool.github.io/MapReduceAlgorithms/MapReduce-book-final.pdf
- White, Tom. Hadoop: the definitive guide. O'Reilly, 2012.
- J. Dean and S. Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters”, OSDI'04: Sixth Symposium on Operating System Design and Implementation, Dec, 2004.
- Apache Hadoop YARN architecture - https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html