Introduction to MapReduce/Hadoop

Thomas Heinis

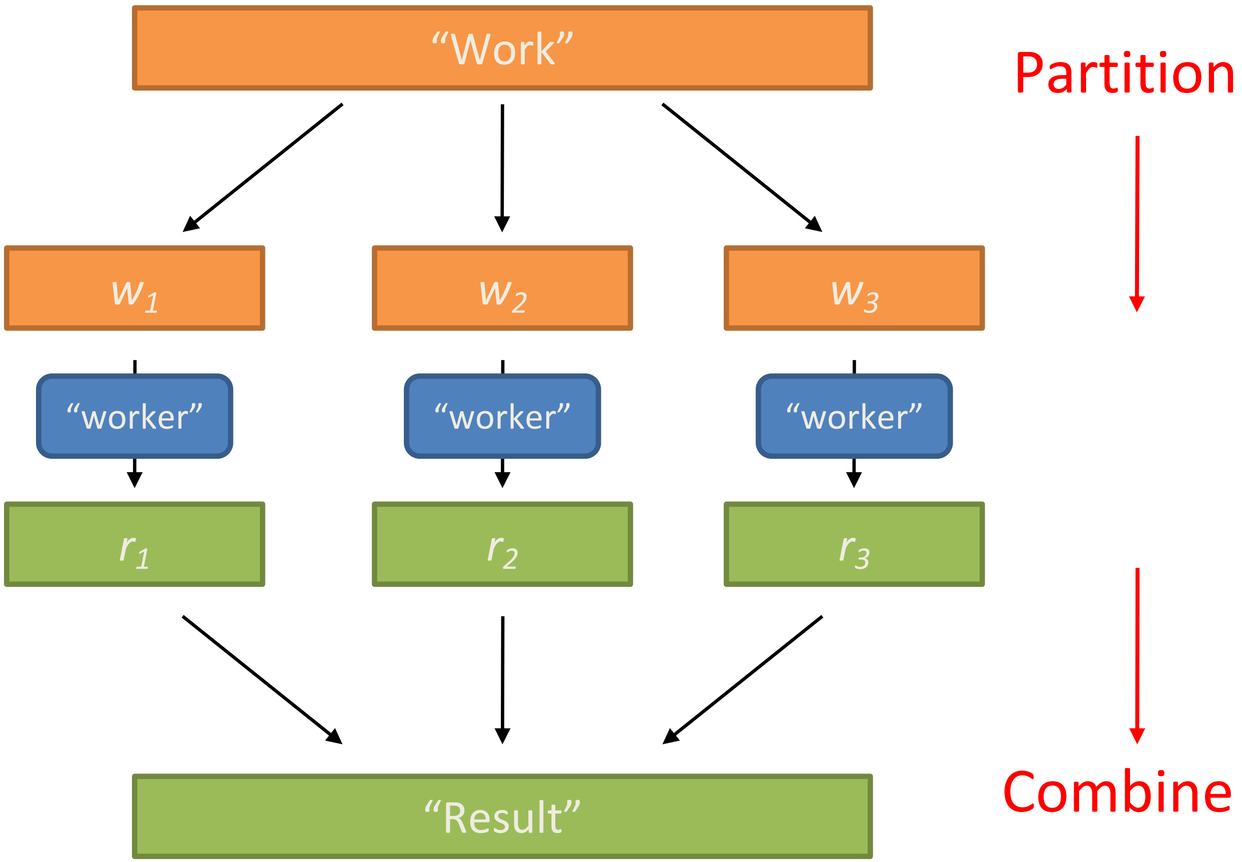
Imperial College London

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?

Parallelization

- 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

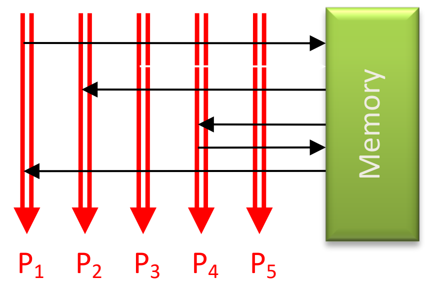
Managing Multiple Workers

- 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…

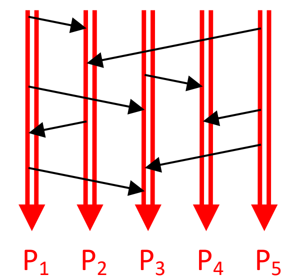
Current Tools

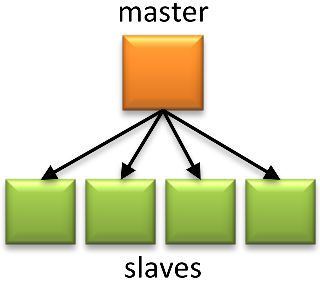
- Programming models
  - Shared memory (pthreads)
  - Message passing (MPI)
- Design Patterns
  - Master-slaves
  - Producer-consumer flows
  - Shared work queues

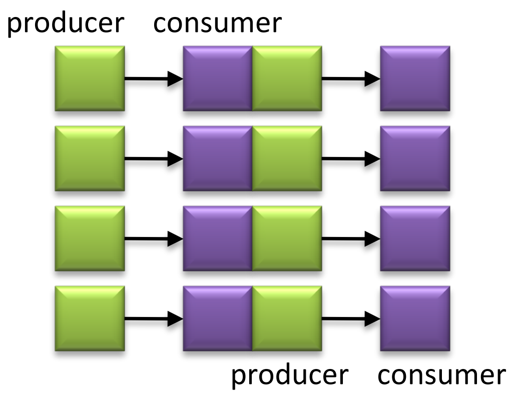
Shared Memory

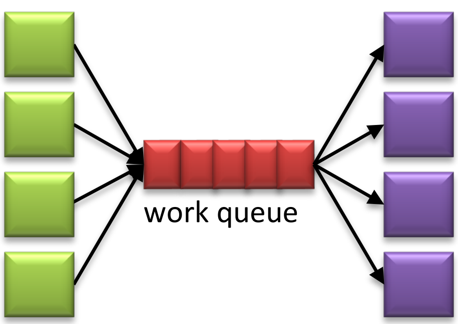


Message Passing









Concurrency Challenge!

- Concurrency is difficult to reason about
- Concurrency is even more difficult to reason about
  - At the scale of datacenters (even across 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 traditional 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

Key Ideas

- Scale “out”, not “up”
  - Limits of single machines and large shared-memory machines
- Move processing to the data
  - Cluster have limited bandwidth
- Process data sequentially, avoid random access
  - Seeks are expensive, disk throughput is reasonable
- Seamless scalability
  - From the mythical man-month to the tradable machine-hour

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

MapReduce: Big Data Processing Abstraction

Typical Large-Data Problem

Map

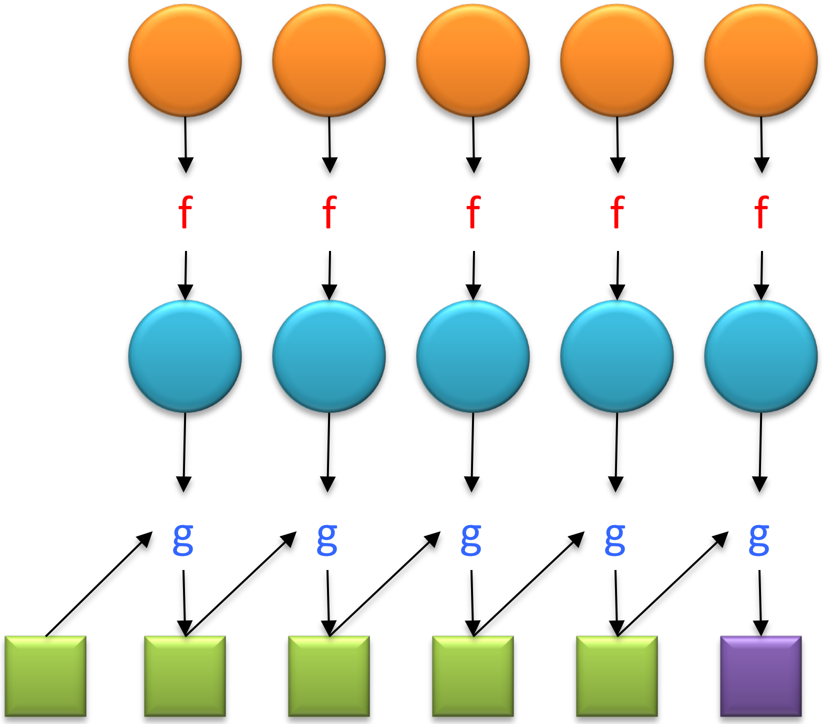
- Iterate over a large number of records
- Extract something of interest from each
- Shuffle and sort intermediate results

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’)]

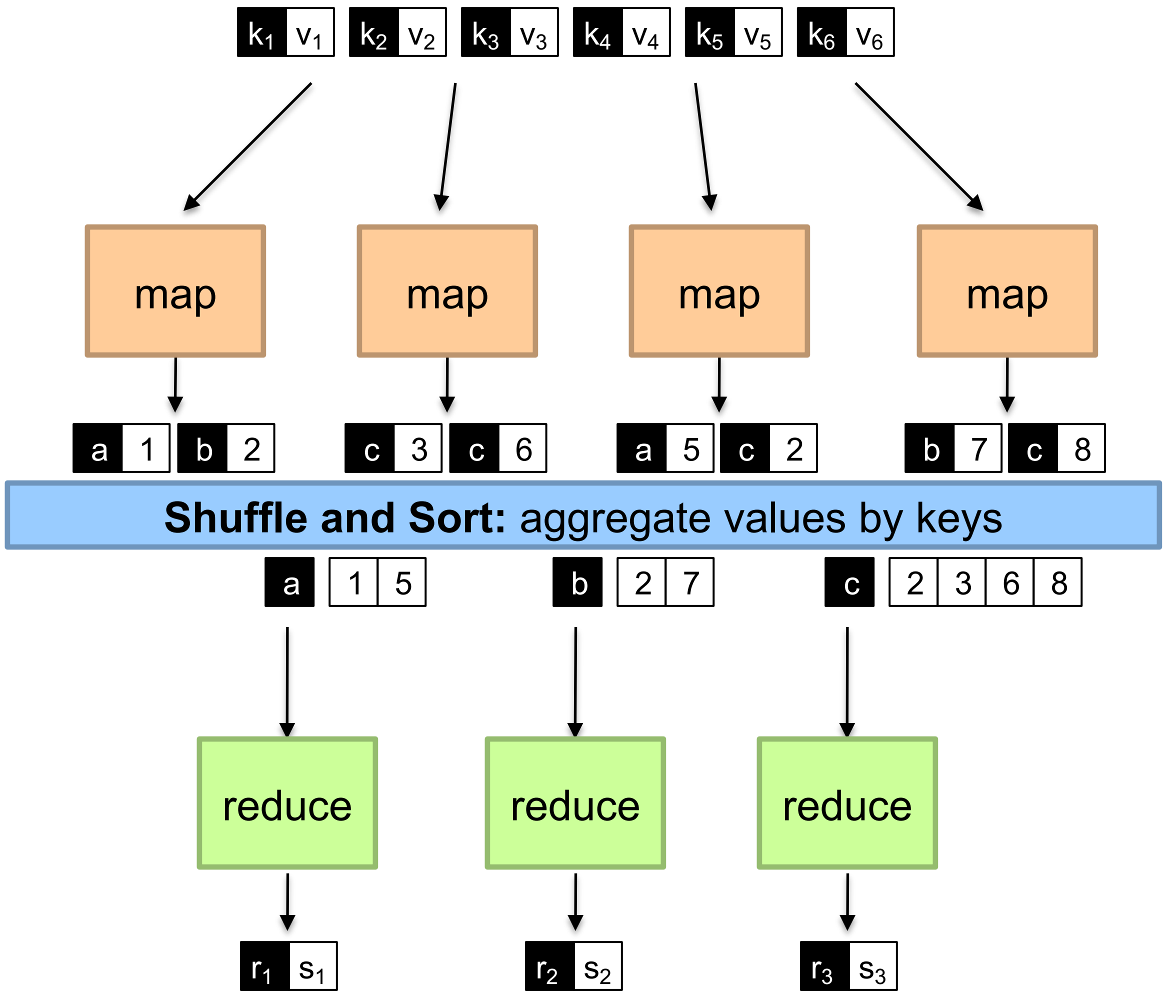
  – All values with the same key are sent to the same reducer

- The execution framework handles everything else…

Key Observation from Data Mining Algorithms

- Popular algorithms have a common loop
- Can be used as the basis for supporting a common middleware
- Target distributed memory parallelism, shared memory parallelism, and combination
- Ability to process large and disk-resident datasets

```
while( ) {
  forall( data instances d) {
    I  =  process(d)
    R(I)  = R(I) op   d
  }
  .......
}
```


k1 v1
k2 v2
k3 v3
k4 v4
k5 v5
k6 v6
map
map
map
map
a 1 b 2
c 3 c 6
a 5 c 2
b 7 c 8
Shuffle and Sort: aggregate values by keys
a 1 5
b 2 7
c 2 3 6 8
reduce
reduce
reduce
r1 s1
r2 s2
r3 s3

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...

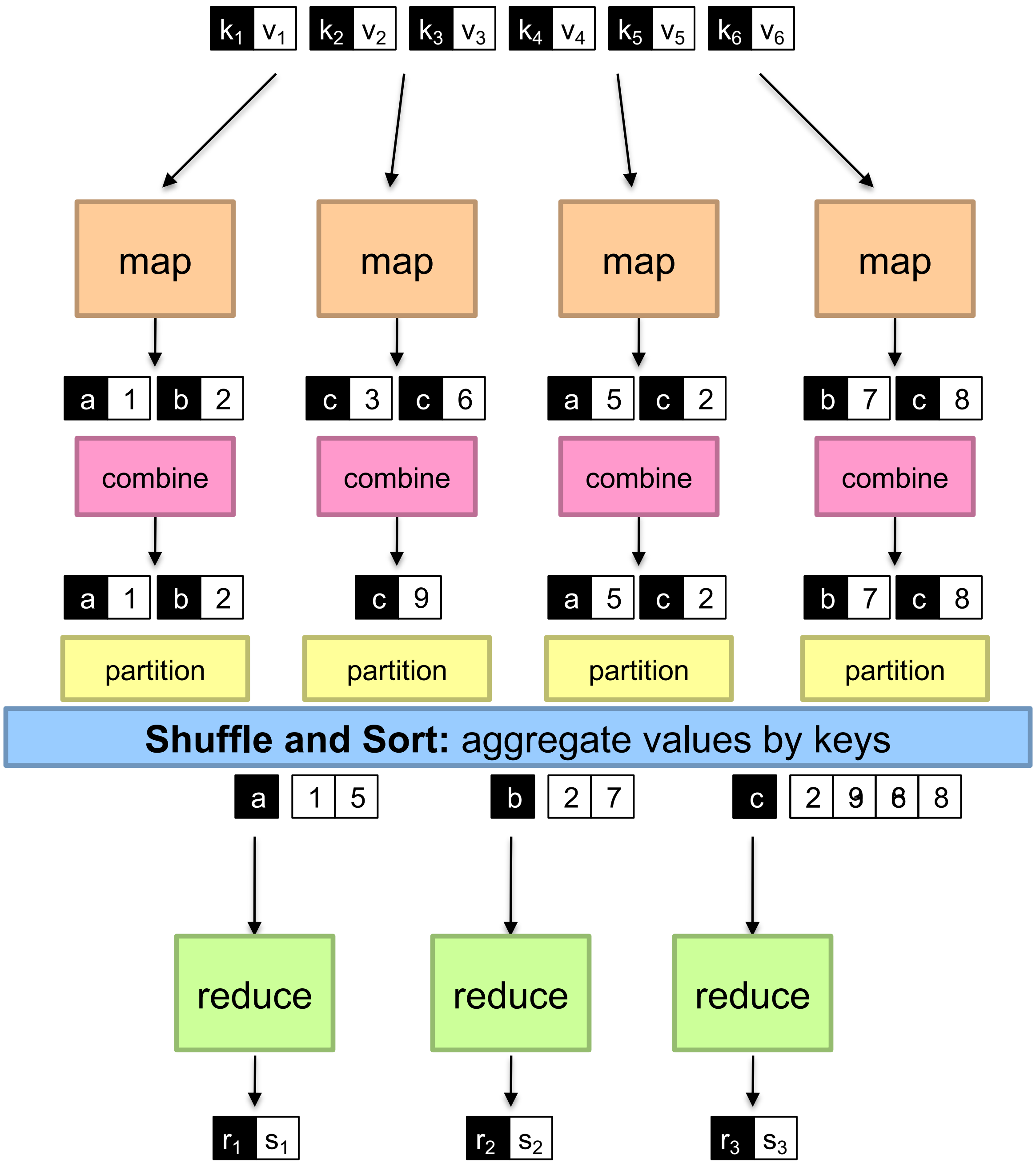
What's "everything else"?

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 filesystem (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 8 8
reduce
reduce
reduce
r_1 s_1
r_2 s_2
r_3 s_3

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.

Hadoop History

- **Dec 2004 –** Google GFS paper published
- **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.

Who uses Hadoop?

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

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)

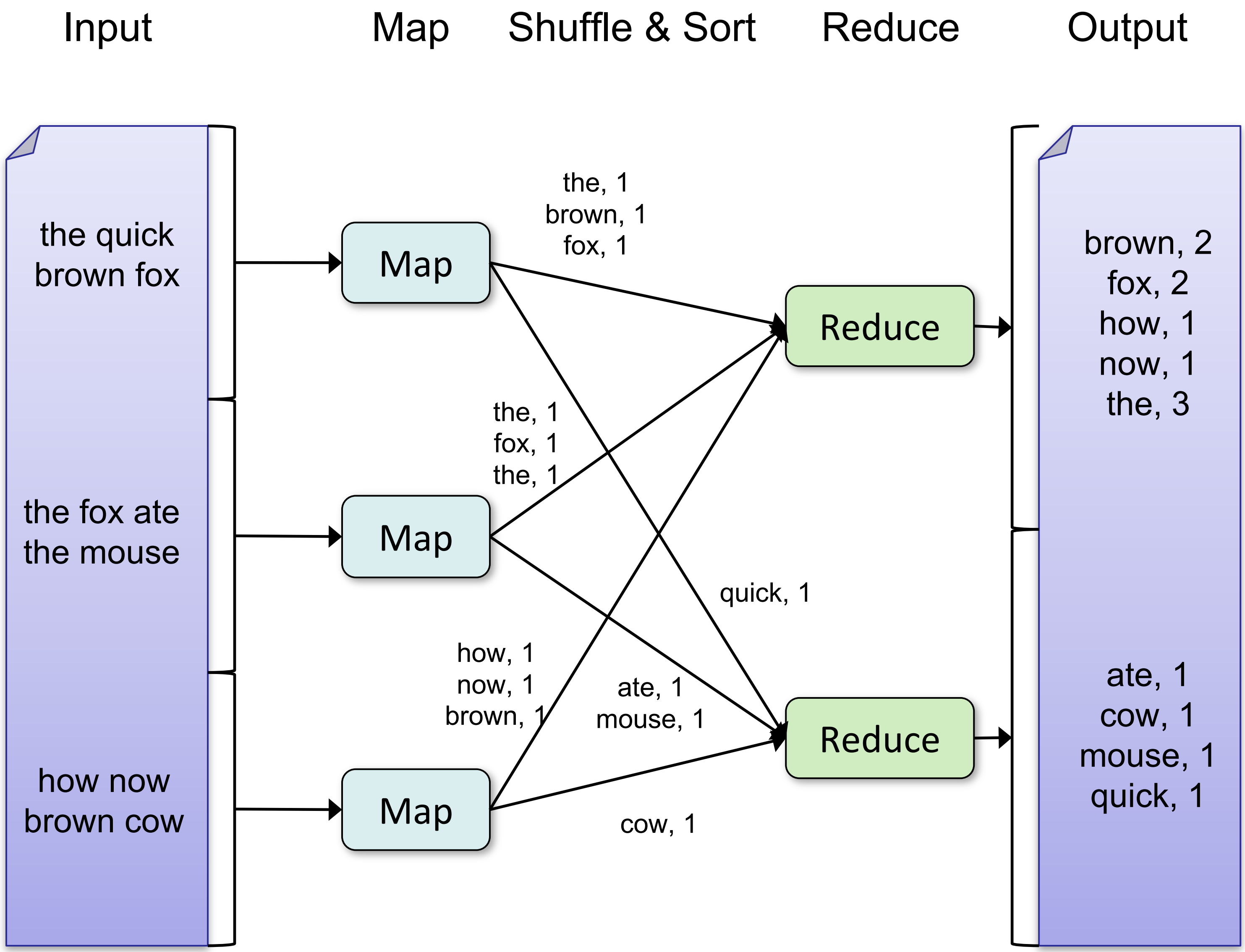
```
public static class IntSumReducer
     extends Reducer<Text,IntWritable,Text,IntWritable> {
  private IntWritable result = new IntWritable();

  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



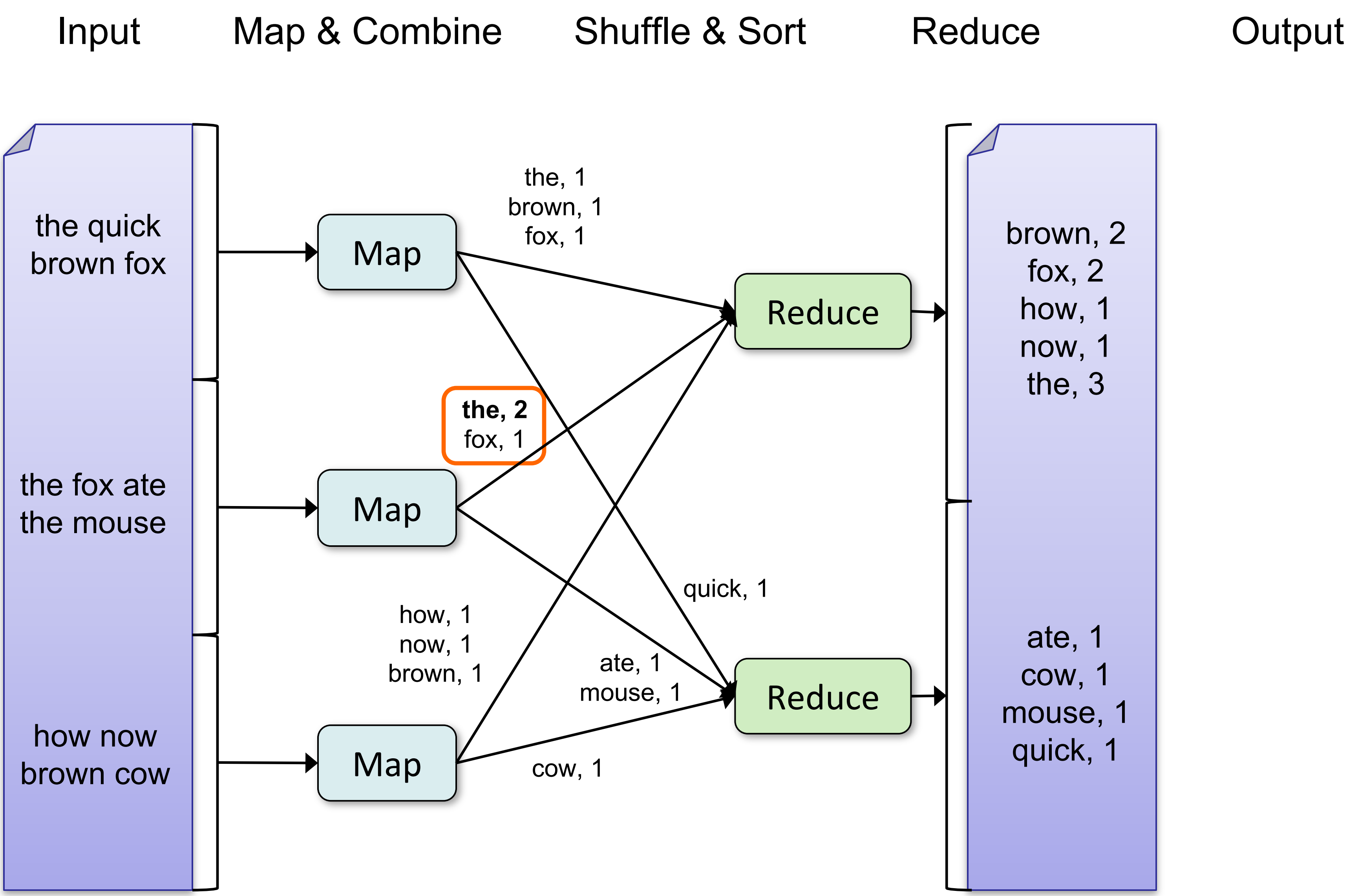
An Optimization: The Combiner

- 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



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)

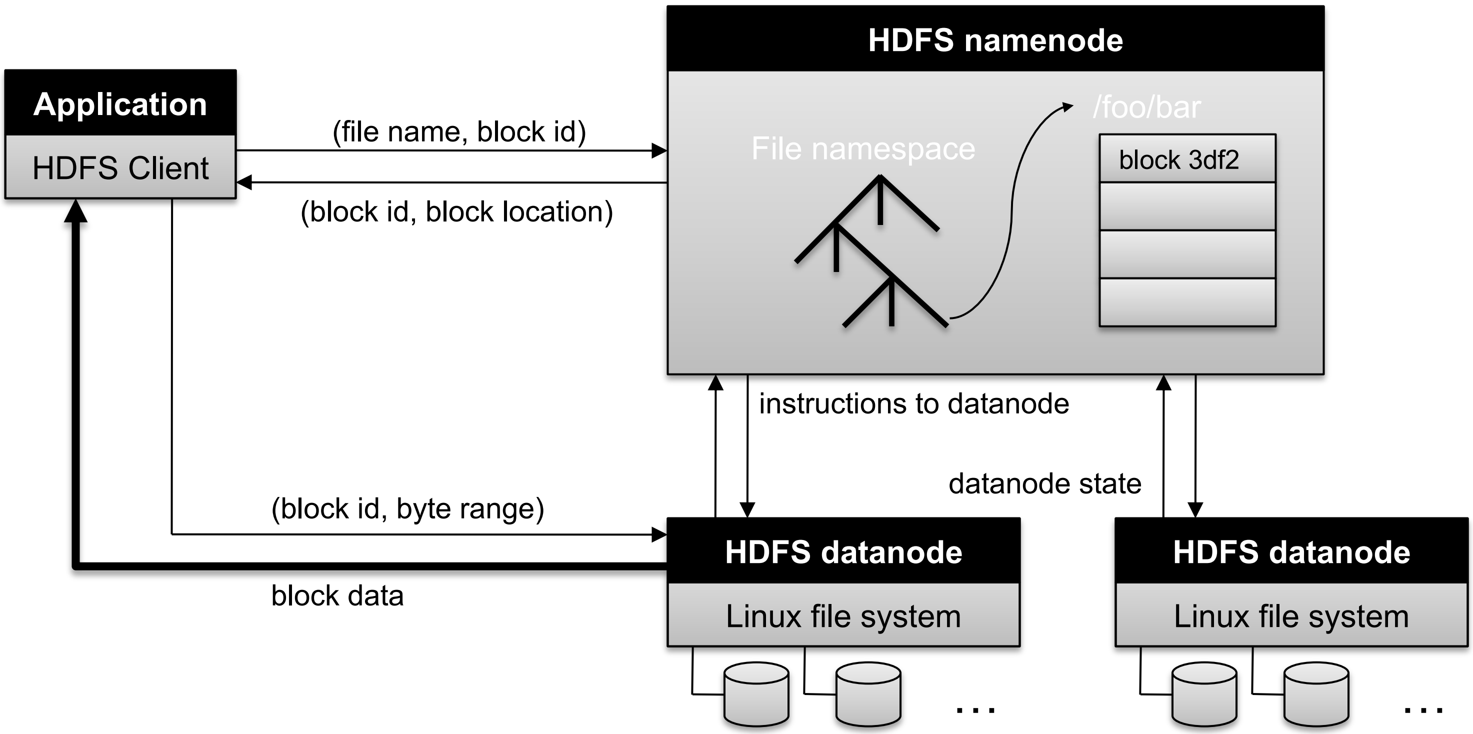
HDFS = GFS clone (same basic ideas)

From GFS to HDFS

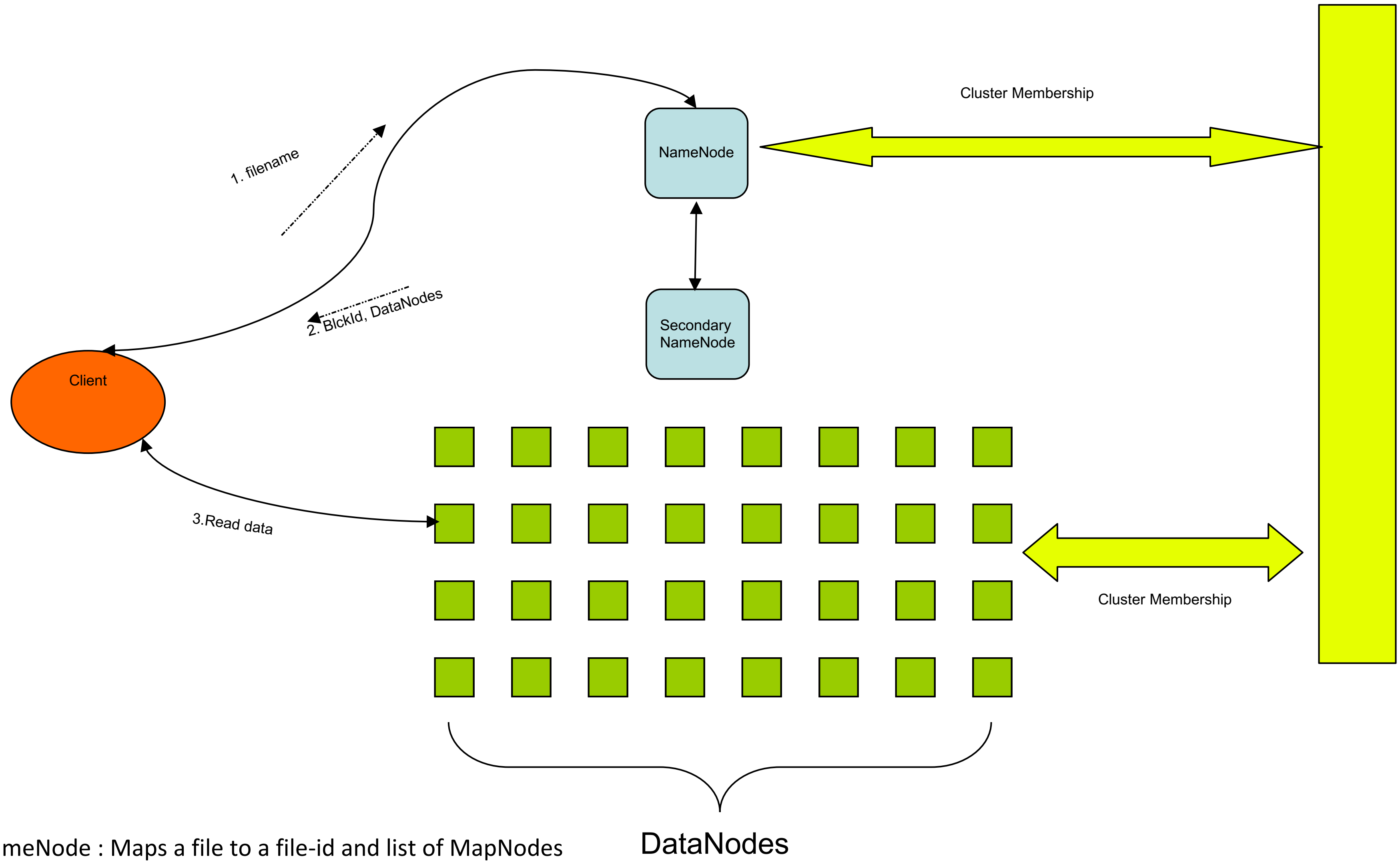
- Terminology differences:
  - GFS master = Hadoop namenode
  - GFS chunkservers = Hadoop datanodes
- Functional differences:
  - HDFS performance is (likely) slower

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

HDFS Workflow



HDFS Architecture

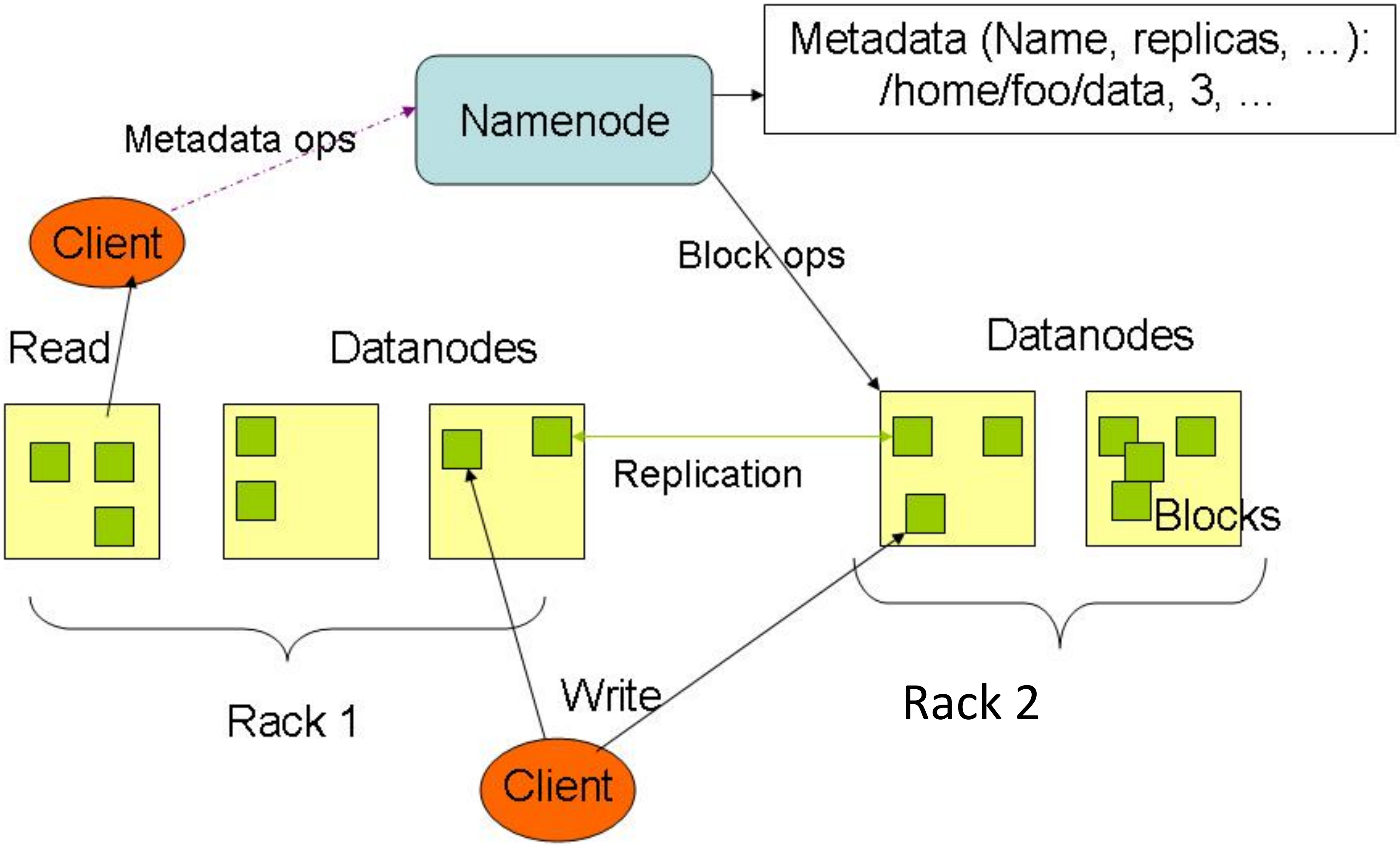


NameNode : Maps a file to a file-id and list of MapNodes
DataNode : Maps a block-id to a physical location on disk
SecondaryNameNode: Periodic merge of Transaction log

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 64MB 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**
- **Would like to make this policy pluggable**

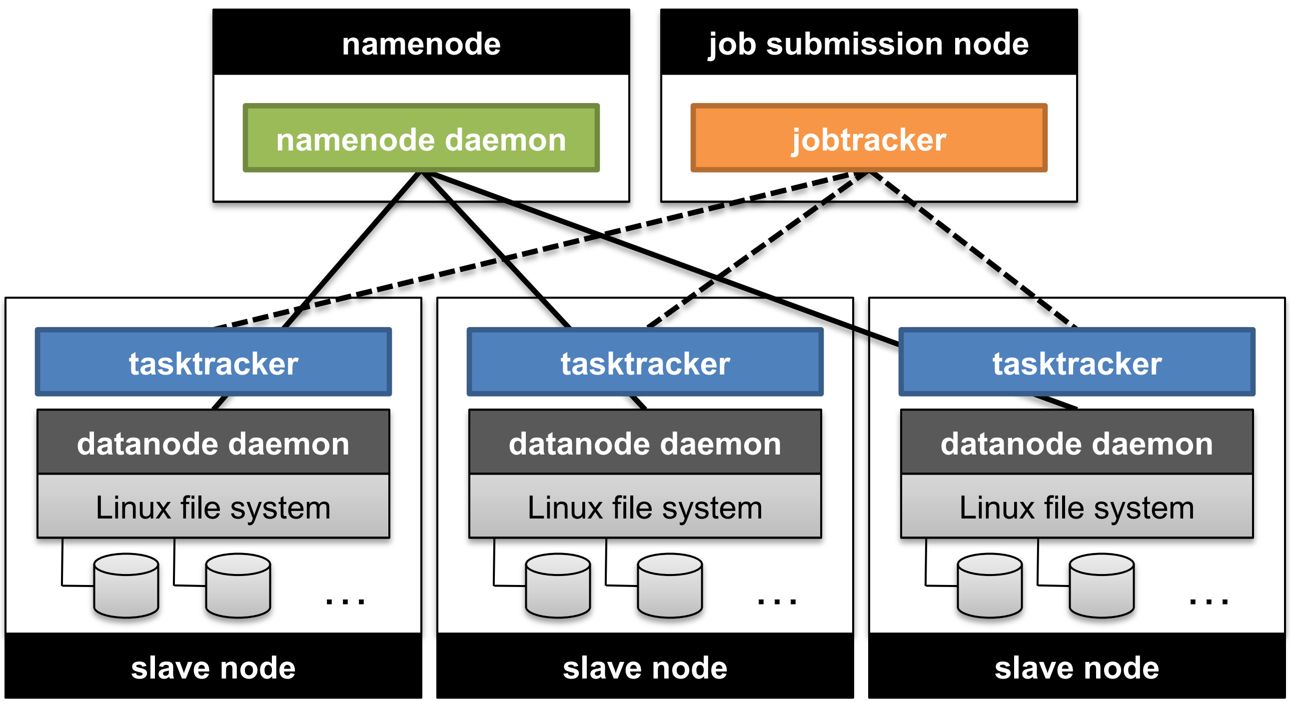
Data Correctness

- **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

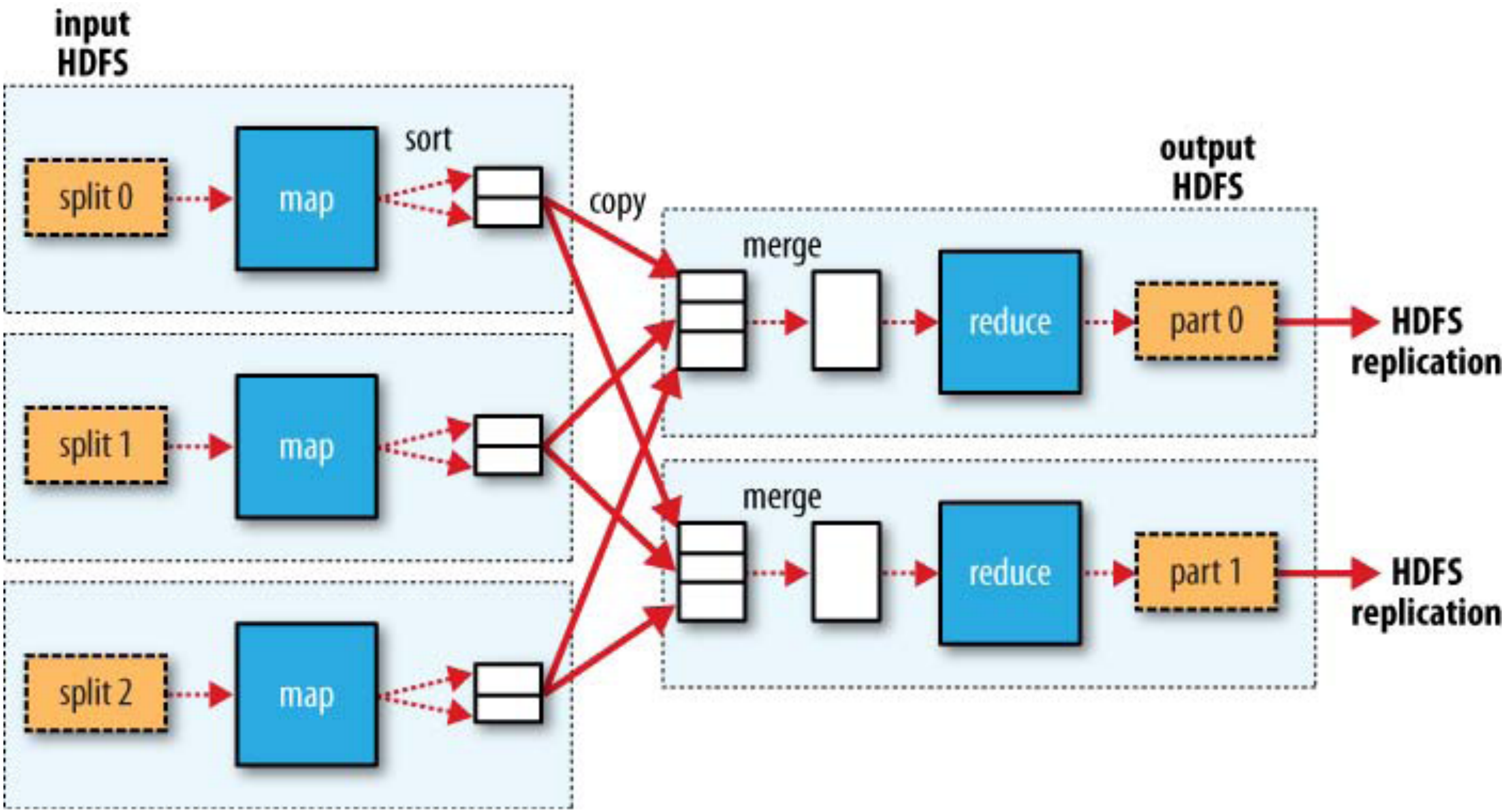
NameNode Failure

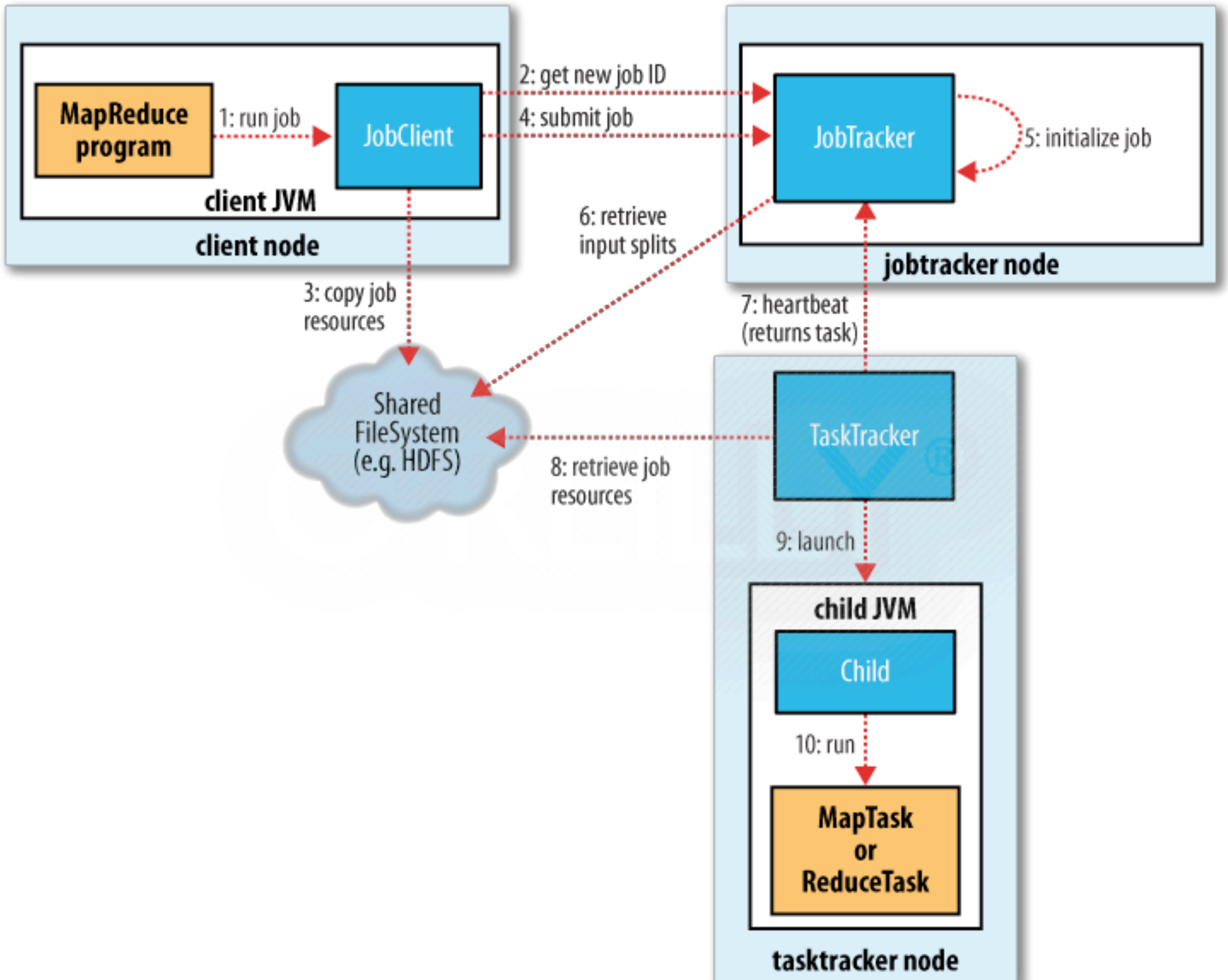
- **A single point of failure**
- **Transaction Log stored in multiple directories**
  - A directory on the local file system
  - A directory on a remote file system (NFS/CIFS)
- **Need to develop a real decentralized solution**

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