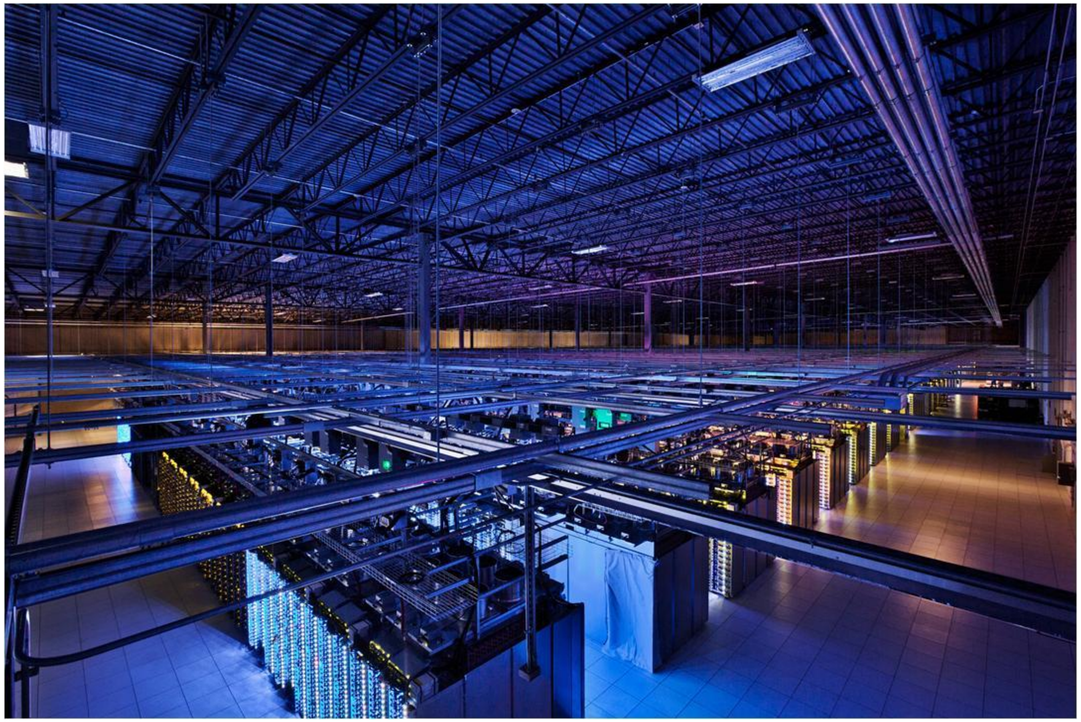
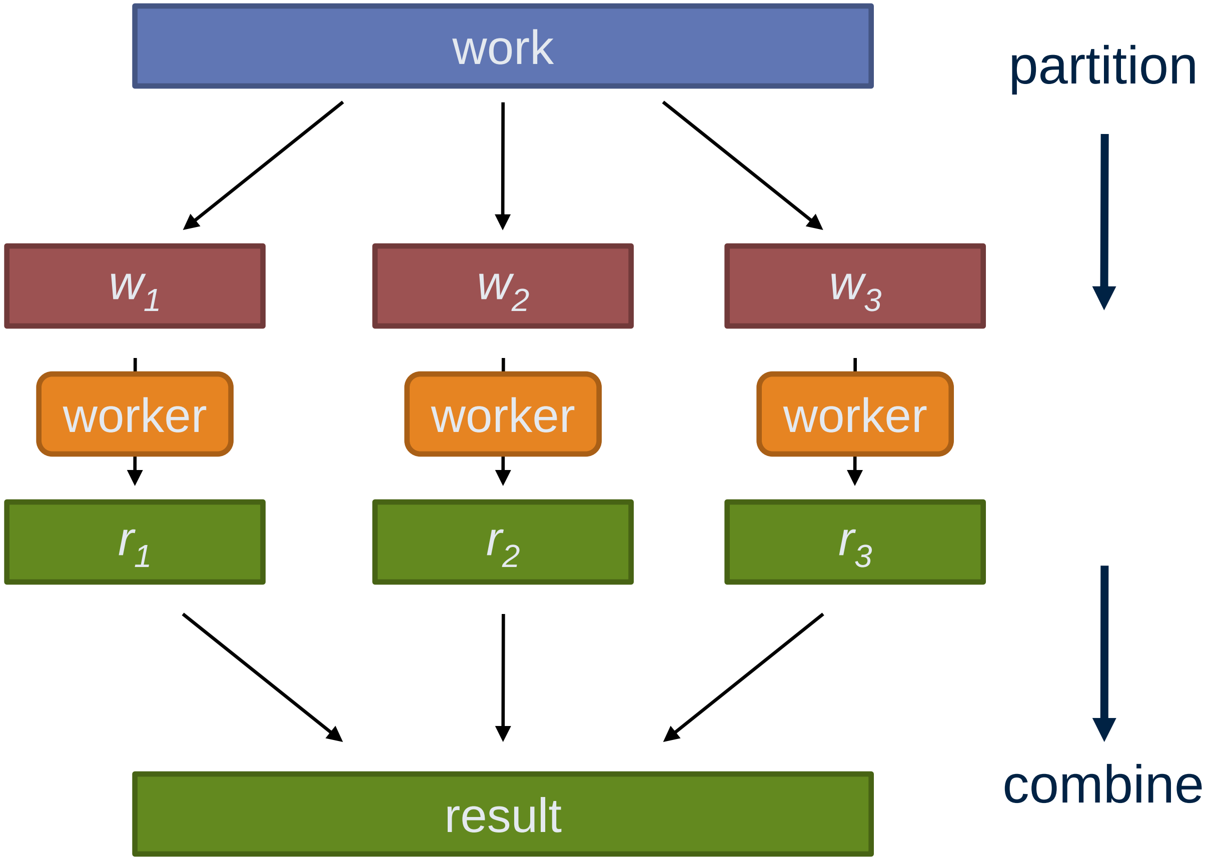
Laarge-Scale Data Engineering

The MapReduce Framework & Hadoop



event.cwi.nl/lsde

Key premise: divide and conquer



Parallelisation 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 know all the workers have finished?
- What if workers die?
- What if data gets lost while transmitted over the network?

What’s the common theme of all of these problems?

Common theme?

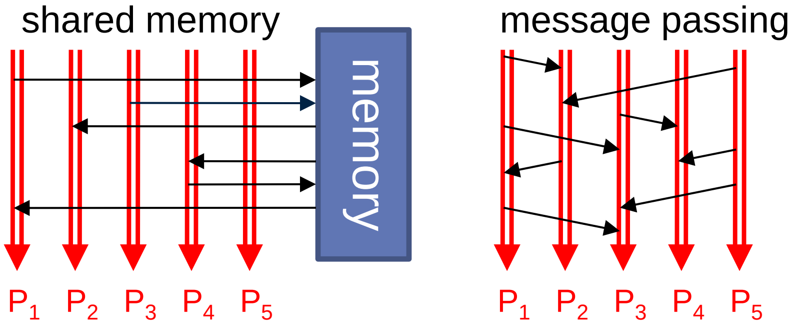
- Parallelization problems arise from:
  - Communication between workers (e.g., to exchange state)
  - Access to shared resources (e.g., data)
- Thus, we need a synchronization mechanism

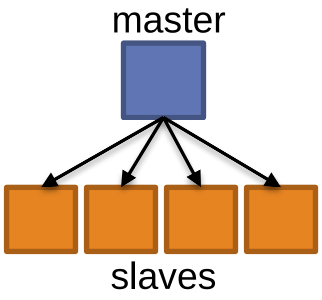
Managing multiple workers

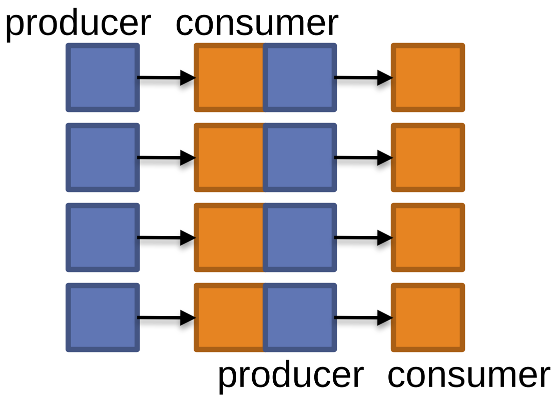
- Difficult because
  - We don’t know the order in which workers run
  - We don’t know when workers interrupt each other
  - We don’t know when workers need to communicate partial results
  - We don’t know the order in which workers access shared data
- Thus, we need:
  - Semaphores (lock, unlock)
 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

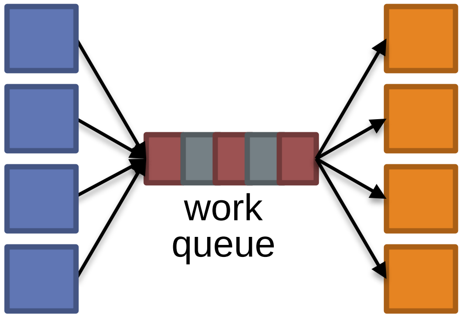
Current tools

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









Parallel programming: human bottleneck

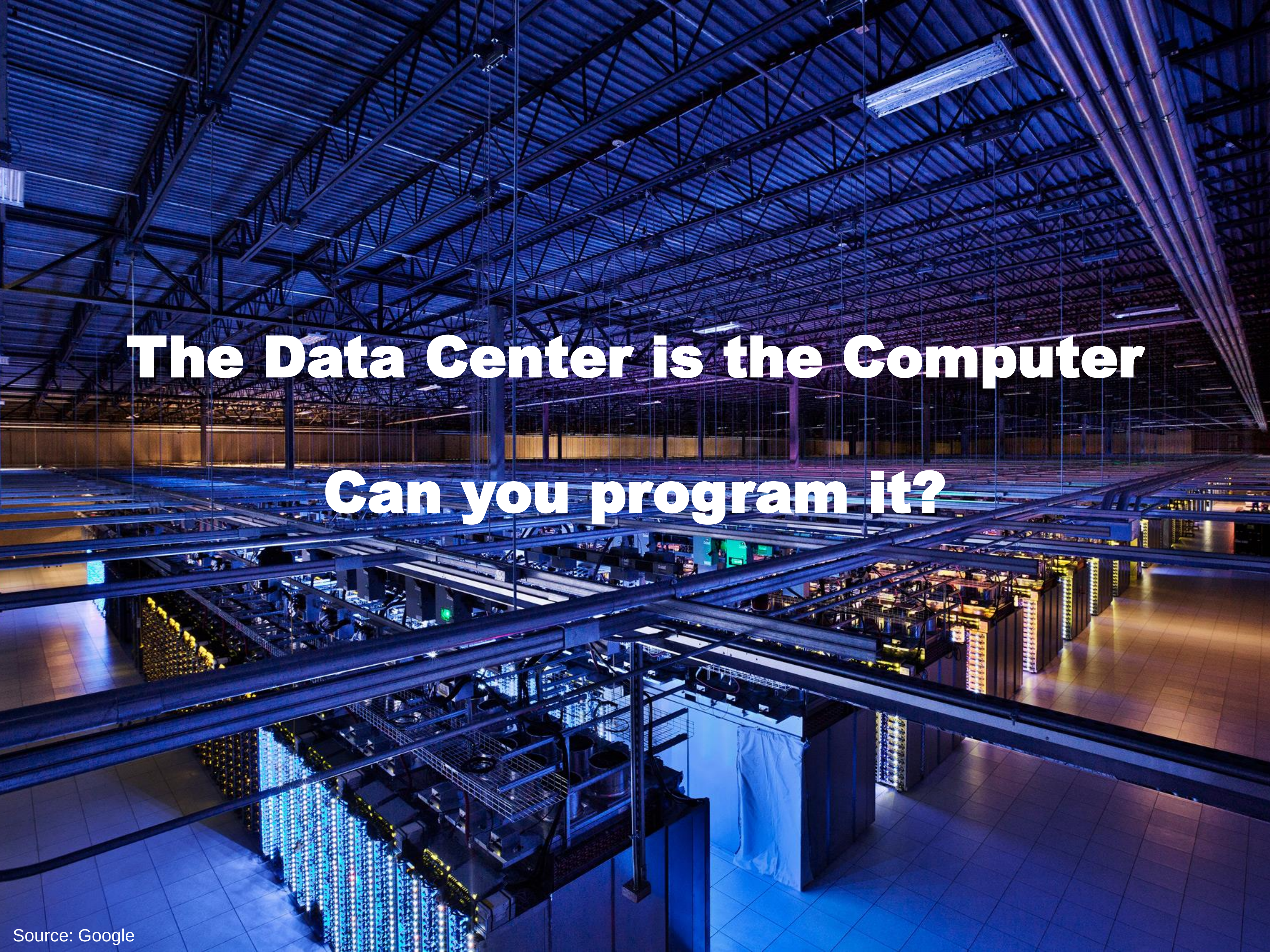
- Concurrency is difficult to reason about
- Concurrency is even more difficult to reason about
  - At the scale of datacenters and across datacenters
  - In the presence of failures
  - In terms of multiple interacting services
- Not to mention debugging…
- The reality:
  - Lots of one-off solutions,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

- The MapReduce Framework alleviates this
  - making this easy is what gave Google the advantage

What's the point?

- It's all about the right level of abstraction
  - Moving beyond the von Neumann architecture
  - We need better programming models
- 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 (aka runtime) handles actual execution

The data center *is* the computer!



The Data Center is the Computer

Can you program it?

Source: Google

MAPREDUCE AND HDFS

Big data needs big ideas

- Scale “out”, not “up”
  - Limits of SMP and large shared-memory machines
- Move processing to the data
  - Cluster has limited bandwidth, cannot waste it shipping data around
- Process data sequentially, avoid random access
  - Seeks are expensive, disk throughput is reasonable, memory throughput is even better
- Seamless scalability
  - From the mythical man-month to the tradable machine-hour
- Computation is still big
  - But if efficiently scheduled and executed to solve bigger problems we can throw more hardware at the problem and use the same code
  - Remember, the datacenter is the computer

Typical Big Data Problem

- Iterate over a large number of records
- Extract something of interest from each

Map

- Shuffle and sort intermediate results
- Aggregate intermediate results

Reduce

- Generate final output

Key idea: provide a functional abstraction for these two operations

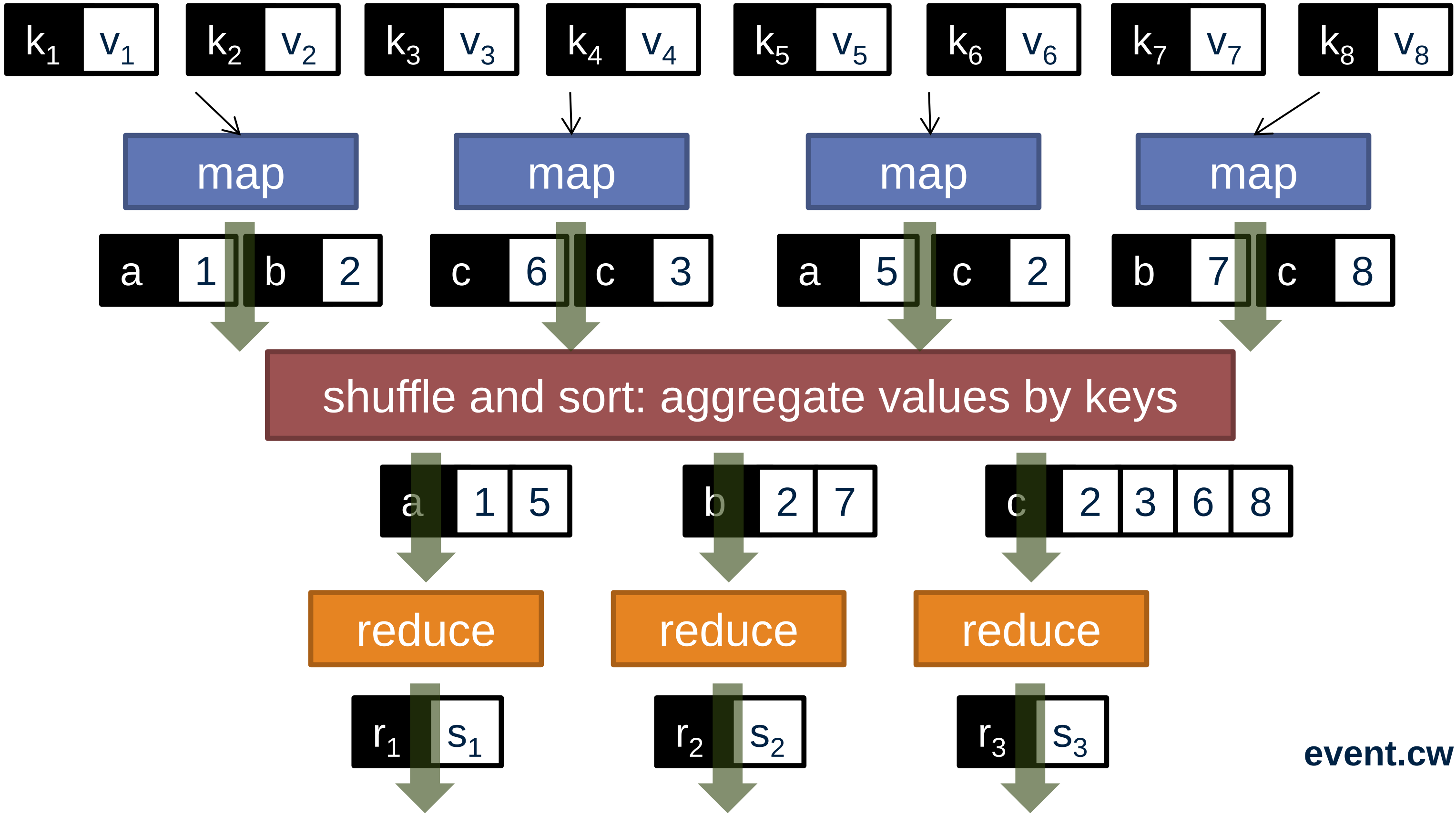
MapReduce

- Programmers specify two functions:

  map $(k_1, v_1) \rightarrow [<k_2, v_2>]$

  reduce $(k_2, [v_2]) \rightarrow [<k_3, v_3>]$

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



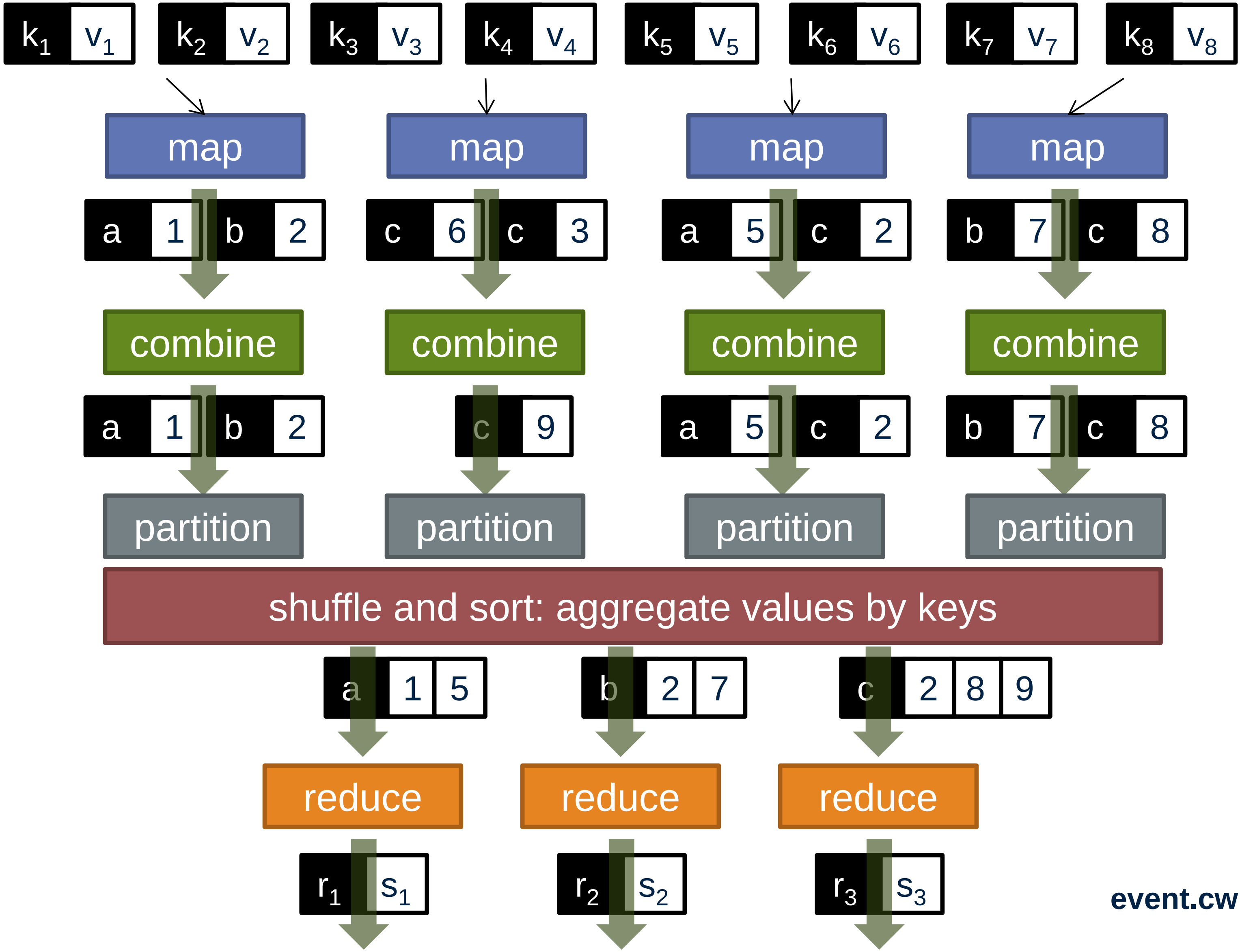
MapReduce runtime

- Orchestration of the distributed computation
- 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 file system (more information 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
- This is the minimal set of information to provide
- 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

Putting it all together



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

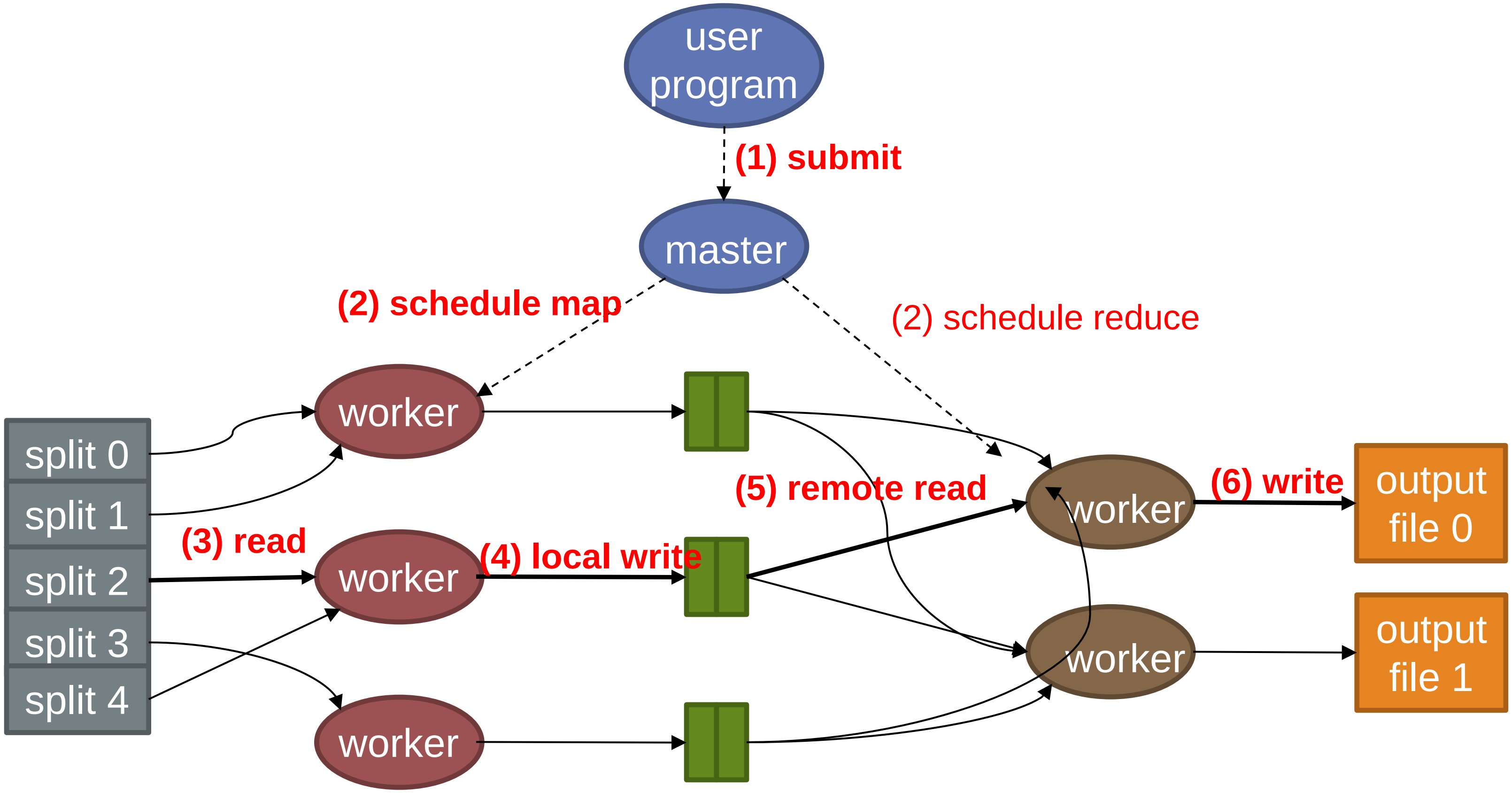
“Hello World”: Word Count

```
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, now an Apache project
  - Used in production at Yahoo, Facebook, Twitter, LinkedIn, Netflix, …
  - The *de facto* big data processing platform
  - Rapidly expanding software ecosystem
- Lots of custom research implementations
  - For GPUs, cell processors, etc.



Input files

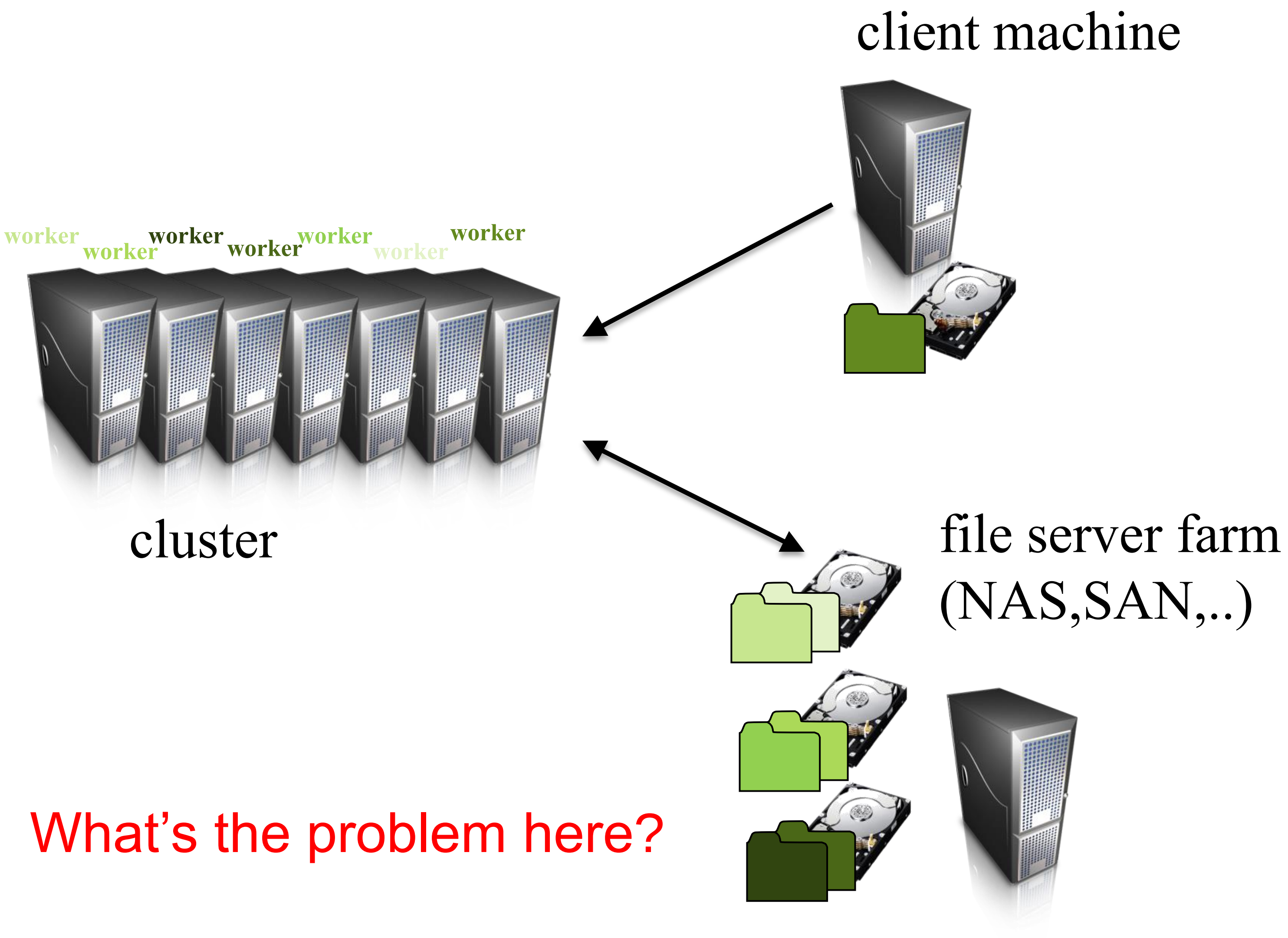
Map phase

Intermediate files (on local disk)

Reduce phase

Output files

How do we get data to the workers?



client machine

worker worker worker worker worker worker worker

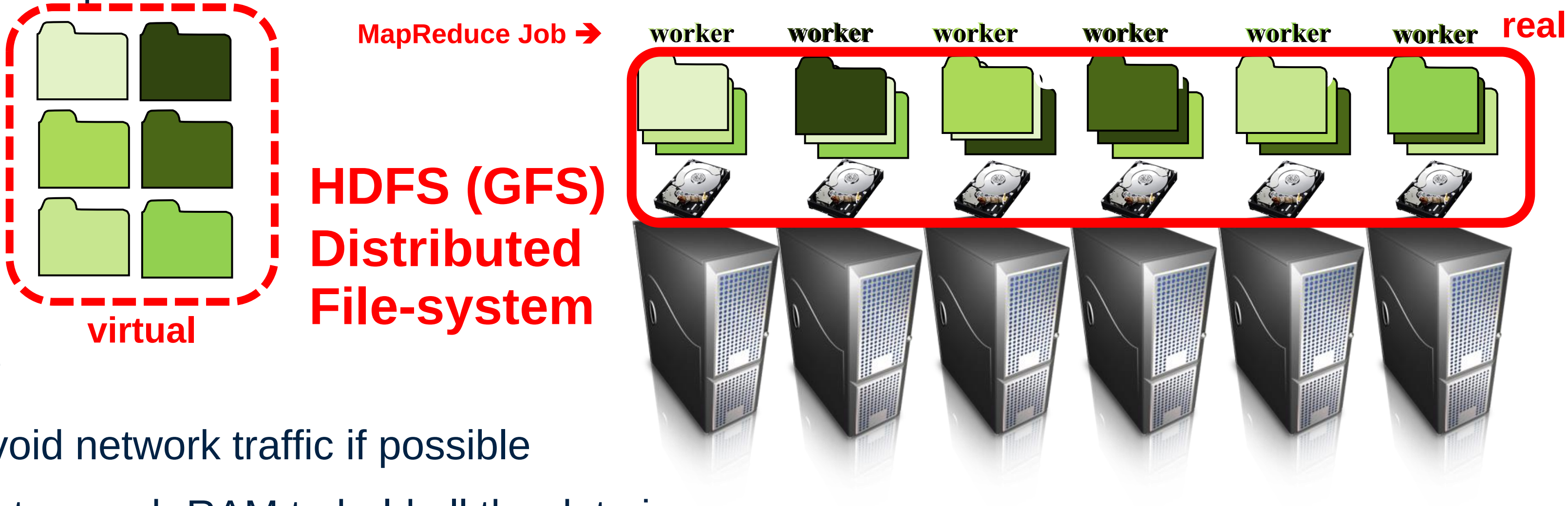
cluster

file server farm
(NAS,SAN,..)

What's the problem here?

Distributed file system

- Do not move data to workers, but 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



virtual

MapReduce Job ➔

worker worker worker worker worker worker

real

HDFS (GFS)
Distributed
File-system

- Why?
  - Avoid network traffic if possible
  - 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

Note: all data is replicated for fault-tolerance (HDFS default:3x)

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)

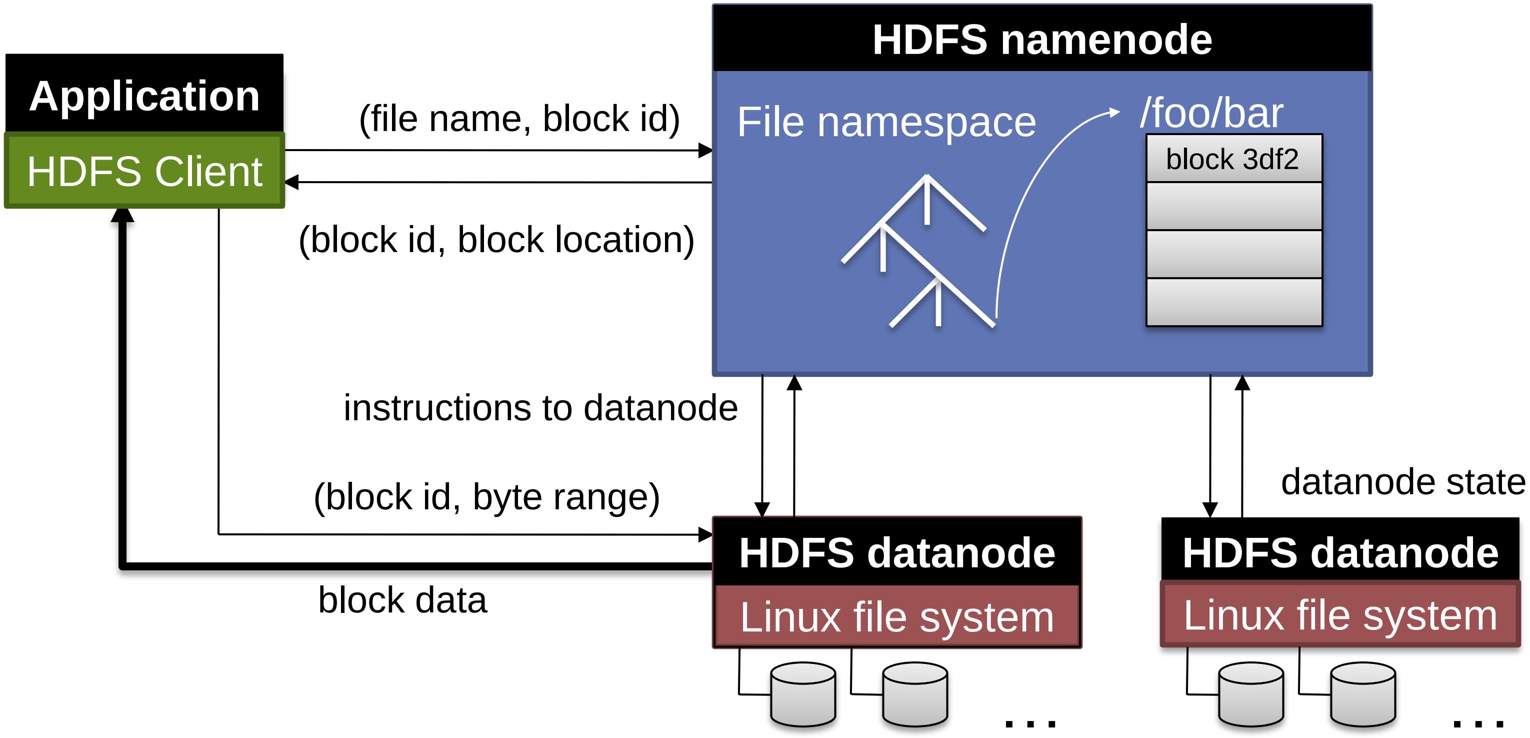
HDFS = GFS clone (same basic ideas)

From GFS to HDFS

- Terminology differences:
  - GFS master = Hadoop namenode
  - GFS chunkservers = Hadoop datanodes
- Differences:
  - Different consistency model for file appends
  - Implementation
  - Performance

For the most part, we'll use Hadoop terminology

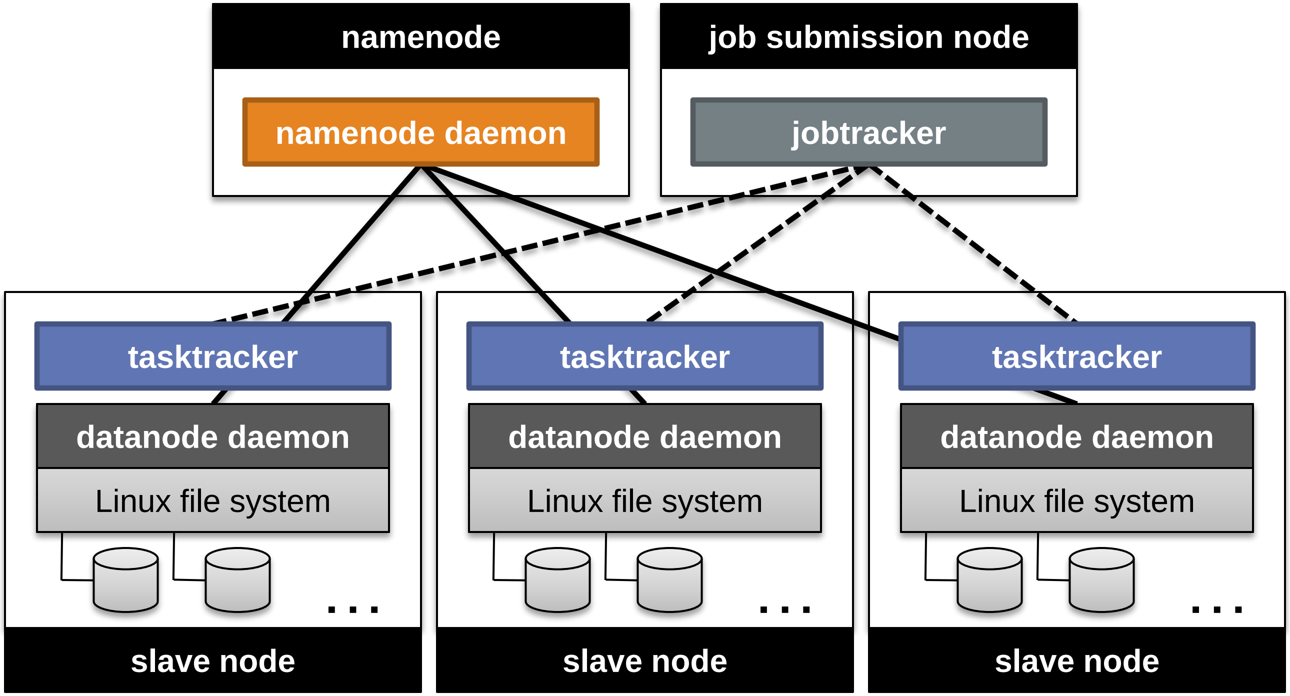
HDFS architecture



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

Putting everything together

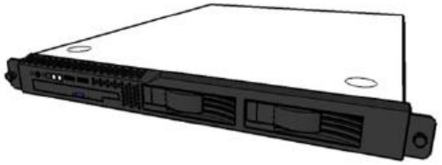


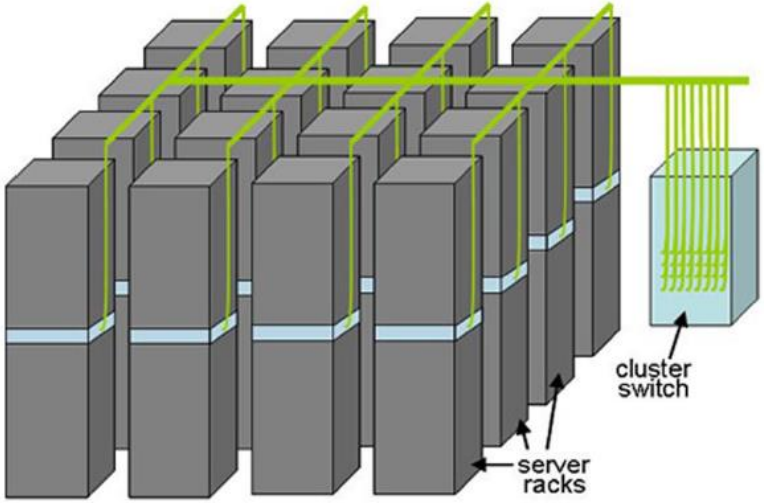
PROGRAMMING FOR A DATA CENTRE

Programming for a data centre

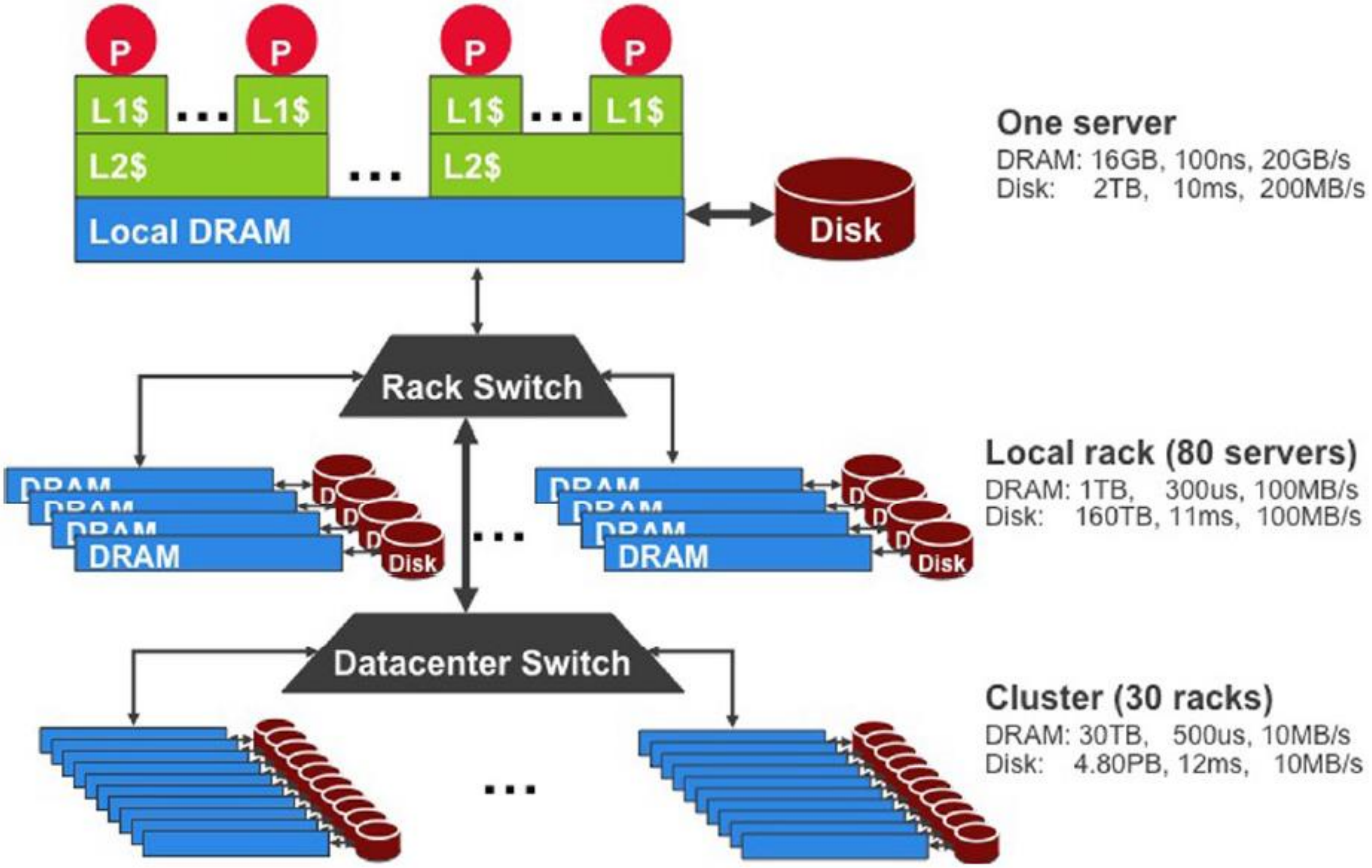
- Understanding the design of warehouse-sized computes
  - Different techniques for a different setting
  - Requires quite a bit of rethinking
- MapReduce algorithm design
  - How do you express everything in terms of `map()`, `reduce()`, `combine()`, and `partition()`?
  - Are there any design patterns we can leverage?

Building Blocks





Storage Hierarchy



One server
DRAM: 16GB, 100ns, 20GB/s
Disk: 2TB, 10ms, 200MB/s

Local rack (80 servers)
DRAM: 1TB, 300us, 100MB/s
Disk: 160TB, 11ms, 100MB/s

Cluster (30 racks)
DRAM: 30TB, 500us, 10MB/s
Disk: 4.80PB, 12ms, 10MB/s

Scaling up vs. out

- No single machine is large enough
  - Smaller cluster of large SMP machines vs. larger cluster of commodity machines (e.g., 8 128-core machines vs. 128 8-core machines)
- Nodes need to talk to each other!
  - Intra-node latencies: ~100 ns
  - Inter-node latencies: ~100 µs
- Let’s model communication overhead

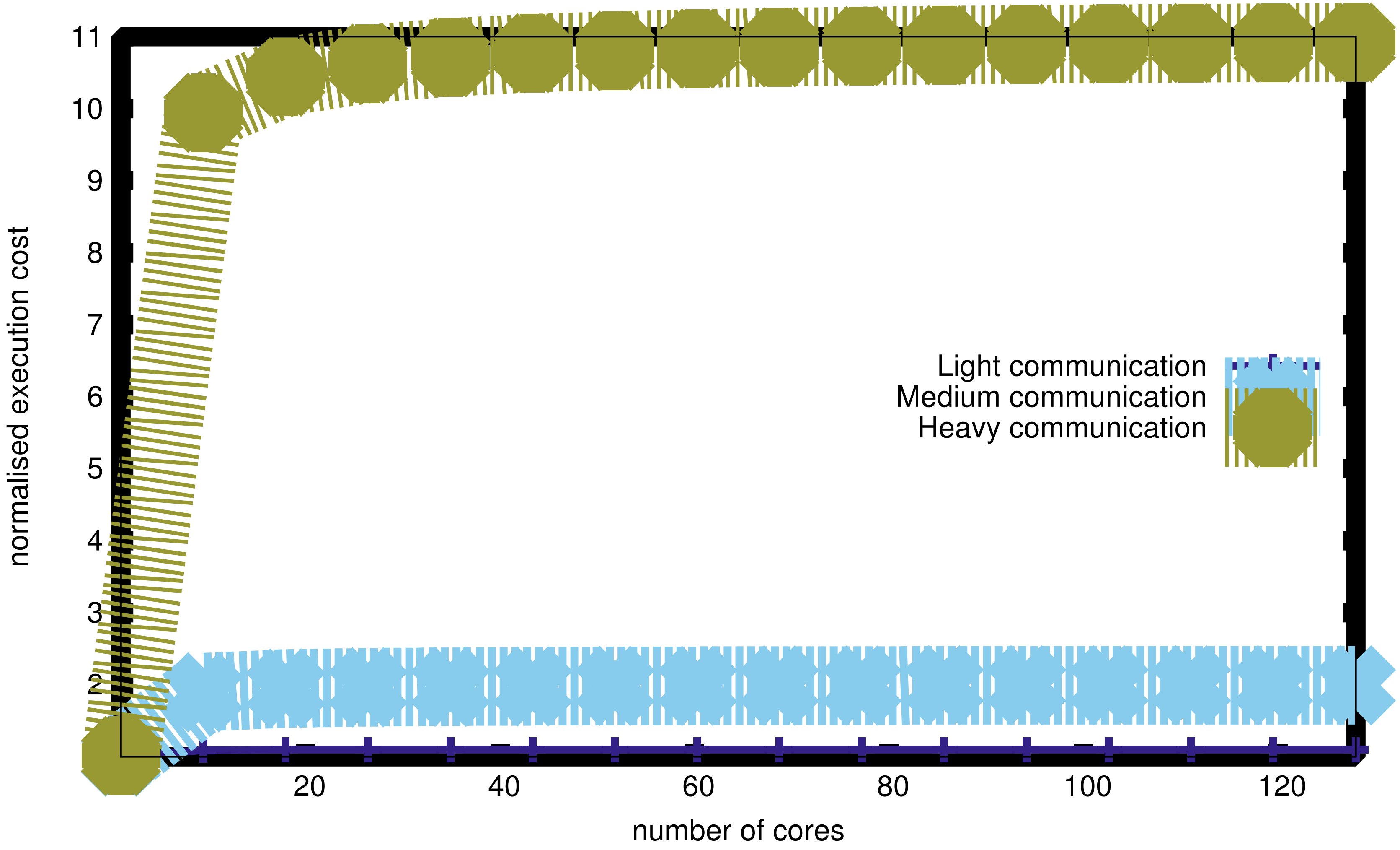
Modelling communication overhead

- Simple execution cost model:
  - Total cost = cost of computation + cost to access global data
  - Fraction of **local access** is **inversely proportional** to size of cluster
    - $1/n$ of the work is local
  - n nodes (ignore cores for now)

$$1\ \text{ms} + f \times [100\ \text{ns} \times (1/n) + 100\ \mu\text{s} \times (1 - 1/n)]$$

  - Three scenarios:
    - Light communication: f =1
    - Medium communication: f =10
    - Heavy communication: f =100
- What is the cost of communication?

Overhead of communication



Seeks vs. scans

- Consider a 1TB database with 100 byte records
  - We want to update 1 percent of the records
- Scenario 1: random access
  - Each update takes ~30 ms (seek, read, write)
  - 10^8 updates = ~35 days
- Scenario 2: rewrite all records
  - Assume 100MB/s throughput
  - Time = 5.6 hours(!)
- Lesson: avoid random seeks!

Important Latencies

L1 cache reference	0.5 ns
L2 cache reference	7 ns
Main memory reference	100 ns
Send 2K bytes over 1 Gbps network	20,000 ns
SSD read one page (random)	100,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Read 1MB sequentially from SSD	2,000,000 ns
Magnetic Disk read one page (random)	10,000,000 ns
Read 1 MB sequentially from magnetic disk	20,000,000 ns
Send packet CA → Netherlands → CA	150,000,000 ns
Read 100MB sequentiall from disk	1,000,000,000 ns

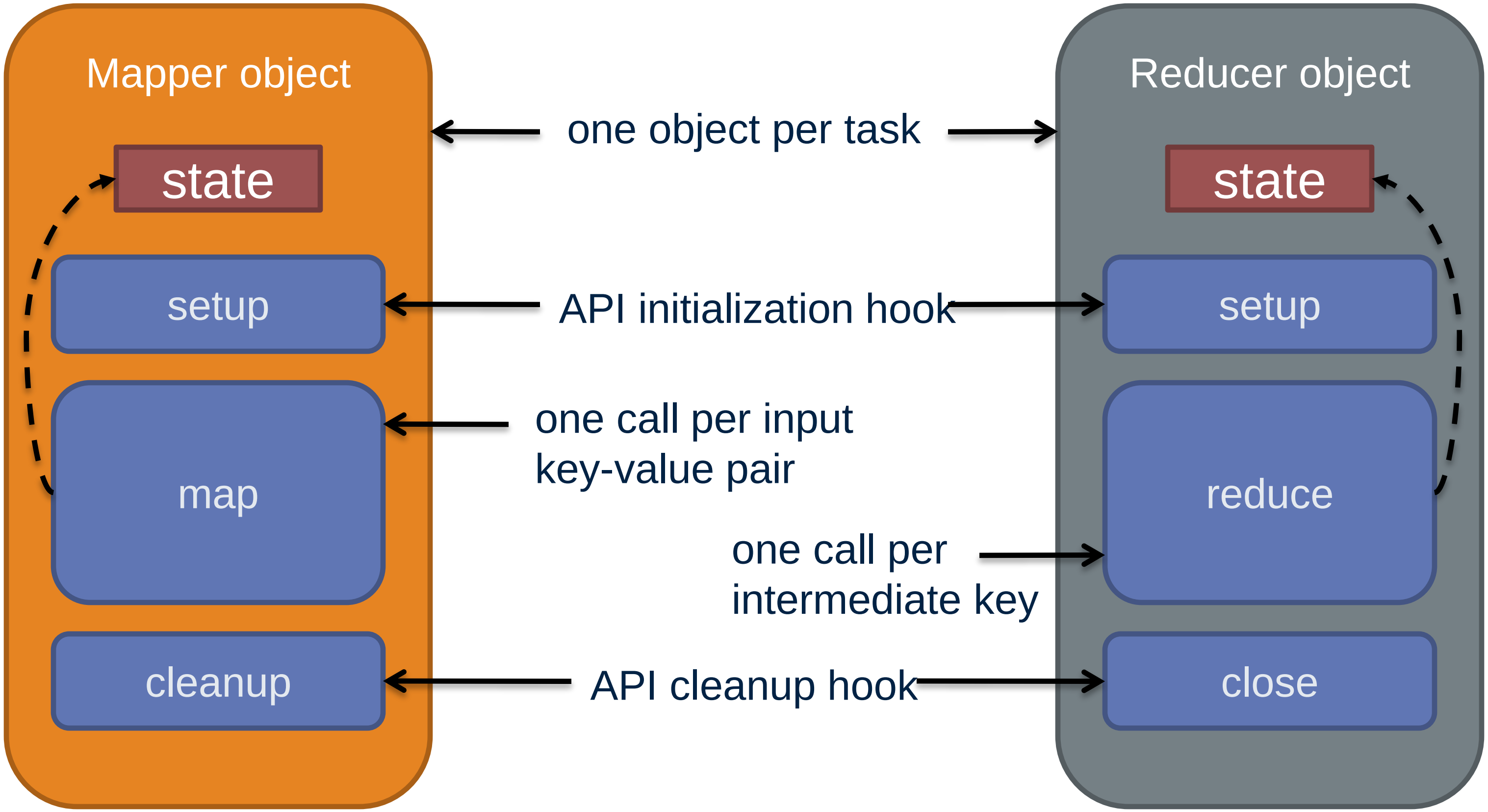
0.4MB/s

DEVELOPING ALGORITHMS

Optimising computation

- The cluster management software orchestrates the computation
- But we can still optimise the computation
  - Just as we can write better code and use better algorithms and data structures
  - At all times confined within the capabilities of the framework
- Cleverly-constructed data structures
  - Bring partial results together
- Sort order of intermediate keys
  - Control order in which reducers process keys
- Partitioner
  - Control which reducer processes which keys
- Preserving state in mappers and reducers
  - Capture dependencies across multiple keys and values

Preserving State



Importance of local aggregation

- Ideal scaling characteristics:
  - Twice the data, twice the running time
  - Twice the resources, half the running time
- Why can’t we achieve this?
  - Synchronization requires communication
  - Communication kills performance
- Thus… avoid communication!
  - Reduce intermediate data via local aggregation
  - Combiners can help

Word count: baseline

```
class Mapper
  method map(docid a, doc d)
    for all term t in d do
      emit(t, 1);

class Reducer
  method reduce(term t, counts [c1, c2, …])
    sum = 0;
    for all counts c in [c1, c2, …] do
      sum = sum + c;
    emit(t, sum);
```

Word count: introducing combiners

```
class Mapper
  method map(docid a, doc d)
    H = associative_array(term → count;)
    for all term t in d do
      H[t]++;
    for all term t in H[t] do
      emit(t, H[t]);
```

Local aggregation inside one document reduces Map output
(the many duplicate occurrences of the word “the” now produce 1 output pair)

Word count: introducing combiners

```
class Mapper
  method initialise()
    H = associative_array(term → count);

  method map(docid a, doc d)
    for all term t in d do
      H[t]++;

  method close()
    for all term t in H[t] do
      emit(t, H[t]);
```

Compute sums *across* documents!
(HashMap H is alive for the entire Map Job, which processes many documents)

Design pattern for local aggregation

- In-mapper combining
  - Fold the functionality of the combiner into the mapper by preserving state across multiple map calls
- Advantages
  - Speed
  - Why is this faster than actual combiners?
- Disadvantages
  - Explicit memory management required
  - Potential for order-dependent bugs

Combiner design

- Combiners and reducers share same method signature
  - Effectively they are map-side reducers
  - Sometimes, reducers can serve as combiners
  - Often, not…
- Remember: combiners are optional optimisations
  - Should not affect algorithm correctness
  - May be run 0, 1, or multiple times
- Example: find average of integers associated with the same key

Computing the mean: version 1

```
class Mapper
  method map(string t, integer r)
    emit(t, r);

class Reducer
  method reduce(string, integers [r1, r2, …])
    sum = 0;    count = 0;
    for all integers r in [r1, r2, …] do
      sum = sum + r;    count++
    $r_{avg}$ = sum / count;
    emit(t, $r_{avg}$);
```

Can we use a reducer as the combiner?

Computing the mean: version 2

```
class Mapper
  method map(string t, integer r)
    emit(t, r);
class Combiner
  method combine(string t, integers [r1, r2, …])
    sum = 0;    count = 0;
    for all integers r in [r1, r2, …] do
      sum = sum + r;    count++;
    emit(t, pair(sum, count));
class Reducer
  method reduce(string t, pairs [(s1, c1), (s2, c2), …])
    sum = 0;    count = 0;
    for all pair(s, c) r in [(s1, c1), (s2, c2), …] do
      sum = sum + s;    count = count + c;
    $r_{avg}$ = sum / count;
    emit(t, $r_{avg}$);
```

Wrong!

Computing the mean: version 3

```
class Mapper
  method map(string t, integer r)
    emit(t, pair(t, 1));
class Combiner
  method combine(string t, pairs [(s1, c1), (s2, c2), …])
    sum = 0;    count = 0;
    for all pair(s, c) in [(s1, c1), (s2, c2), …] do
      sum = sum + s;    count = count + c;
     emit(t, pair(sum, count));
class Reducer
  method reduce(string t, pairs [(s1, c1), (s2, c2), …])
    sum = 0;    count = 0;
    for all pair(s, c) in [(s1, c1), (s2, c2), …] do
      sum = sum + s;    count = count + c;
    $r_{avg}$ = sum / count;
    emit(t, $r_{avg}$);
```

Fixed!

Combiner must have input and output format = Reducer input format

Basic Hadoop API

Mapper

- **`void setup(Mapper.Context context)`**
  Called once at the beginning of the task
- **`void map(K key, V value, Mapper.Context context)`**
  Called once for each key/value pair in the input split
- **`void cleanup(Mapper.Context context)`**
  Called once at the end of the task

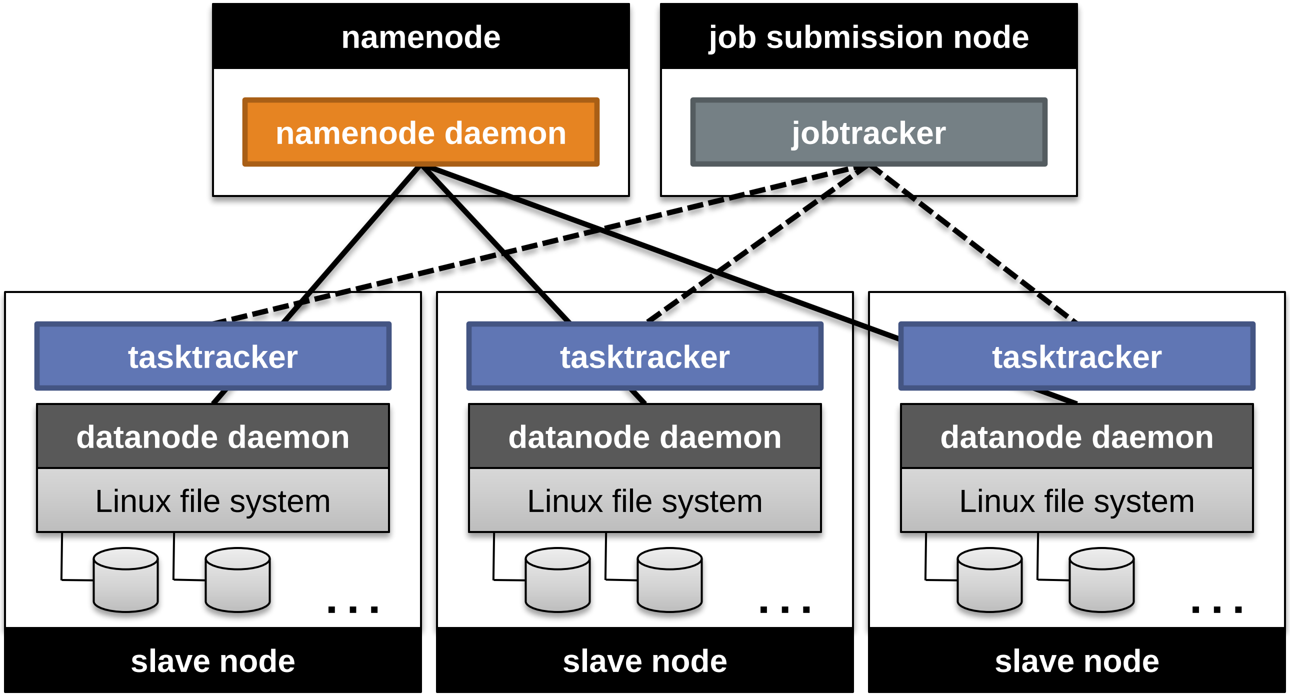
Reducer/Combiner

- **`void setup(Reducer.Context context)`**
  Called once at the start of the task
- **`void reduce(K key, Iterable<V> values, Reducer.Context ctx)`**
  Called once for each key
- **`void cleanup(Reducer.Context context)`**
  Called once at the end of the task

Basic cluster components

- One of each:
  - Namenode (NN): master node for HDFS
  - Jobtracker (JT): master node for job submission
- Set of each per slave machine:
  - Tasktracker (TT): contains multiple task slots
  - Datanode (DN): serves HDFS data blocks

Recap

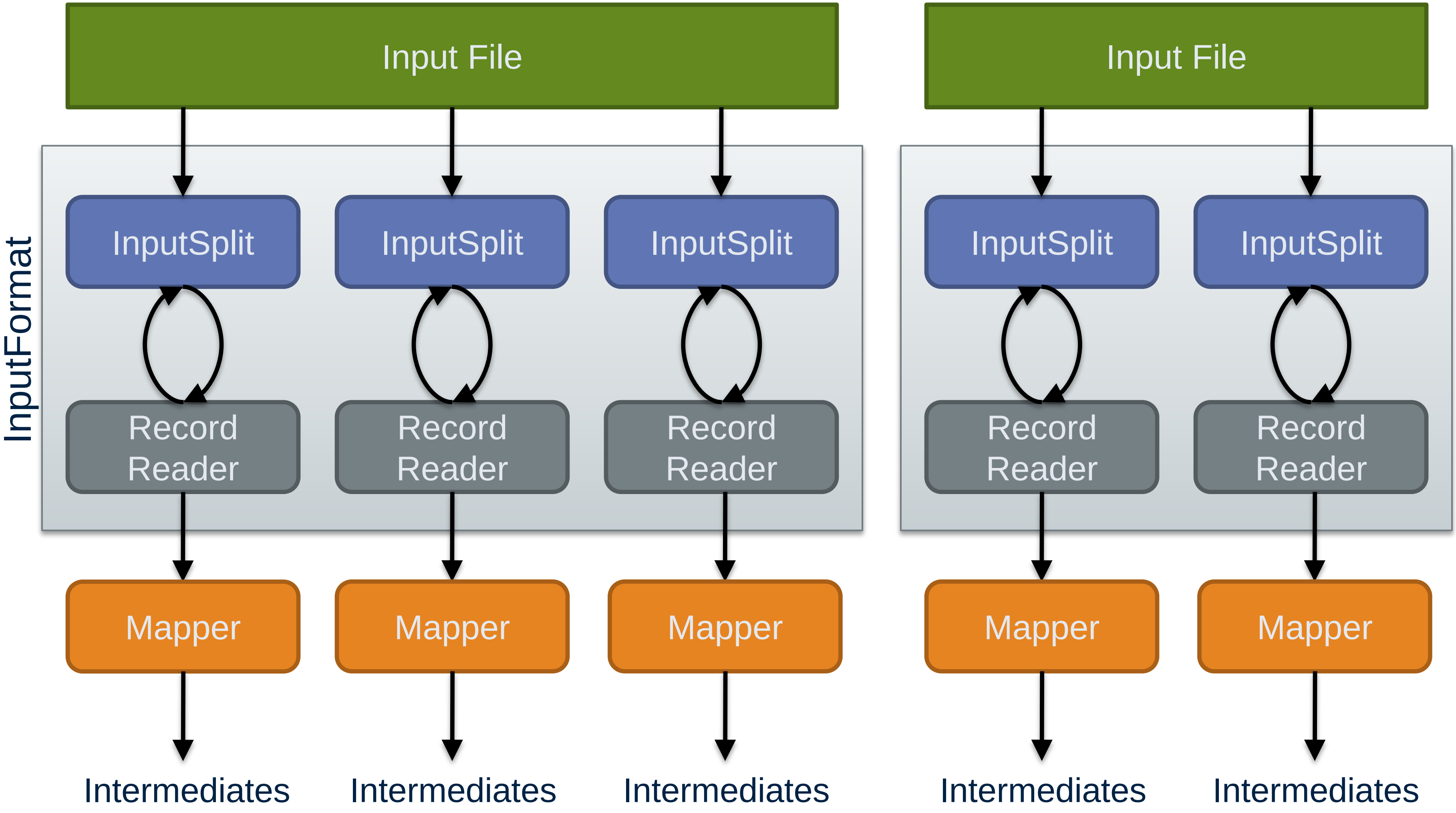

namenode
namenode daemon
job submission node
jobtracker
tasktracker
datanode daemon
Linux file system
...
slave node
tasktracker
datanode daemon
Linux file system
...
slave node
tasktracker
datanode daemon
Linux file system
...
slave node

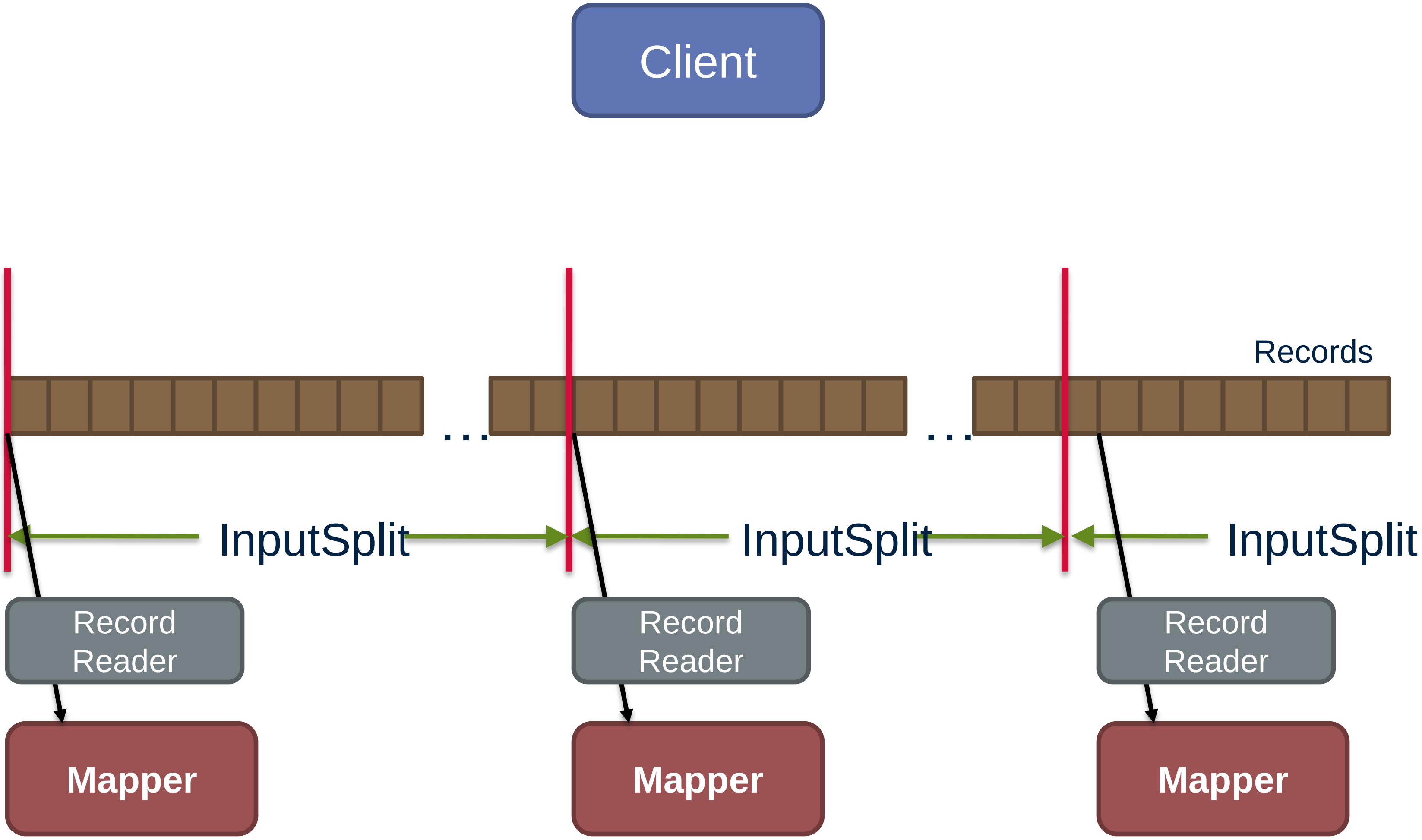
Anatomy of a job

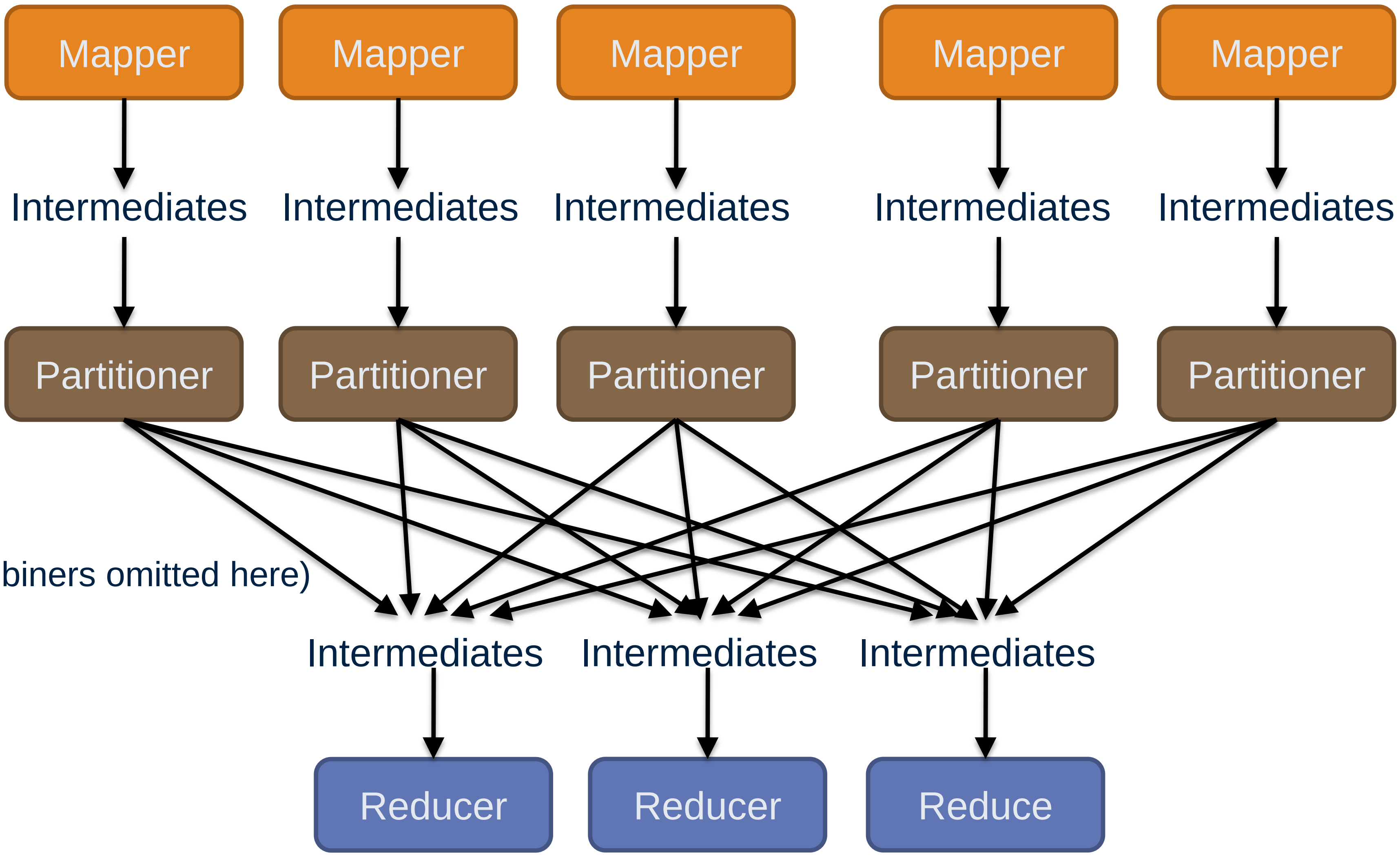
- MapReduce program in Hadoop = Hadoop job
  - Jobs are divided into map and reduce tasks
  - An instance of running a task is called a task attempt (occupies a slot)
  - Multiple jobs can be composed into a workflow
- Job submission:
  - Client (i.e., driver program) creates a job, configures it, and submits it to jobtracker
  - That's it! The Hadoop cluster takes over

Anatomy of a job

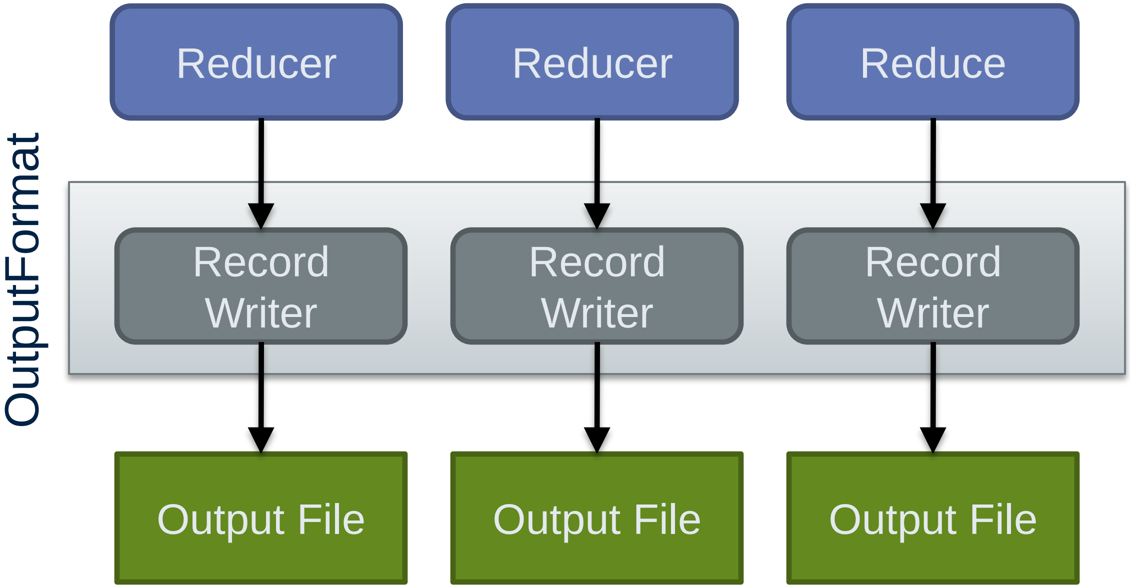
- Behind the scenes:
  - Input splits are computed (on client end)
  - Job data (jar, configuration XML) are sent to JobTracker
  - JobTracker puts job data in shared location, enqueues tasks
  - TaskTrackers poll for tasks
  - Off to the races


Input File
Input File
InputFormat
InputSplit
InputSplit
InputSplit
InputSplit
InputSplit
Record Reader
Record Reader
Record Reader
Record Reader
Record Reader
Mapper
Mapper
Mapper
Mapper
Mapper
Intermediates
Intermediates
Intermediates
Intermediates
Intermediates


Client
Records
. . .
. . .
InputSplit
InputSplit
InputSplit
Record Reader
Record Reader
Record Reader
Mapper
Mapper
Mapper



(combiners omitted here)


Reducer
Reducer
Reduce
OutputFormat
Record Writer
Record Writer
Record Writer
Output File
Output File
Output File

Input and output

- InputFormat:
  - TextInputFormat
  - KeyValueTextInputFormat
  - SequenceFileInputFormat
  - …
- OutputFormat:
  - TextOutputFormat
  - SequenceFileOutputFormat
  - …

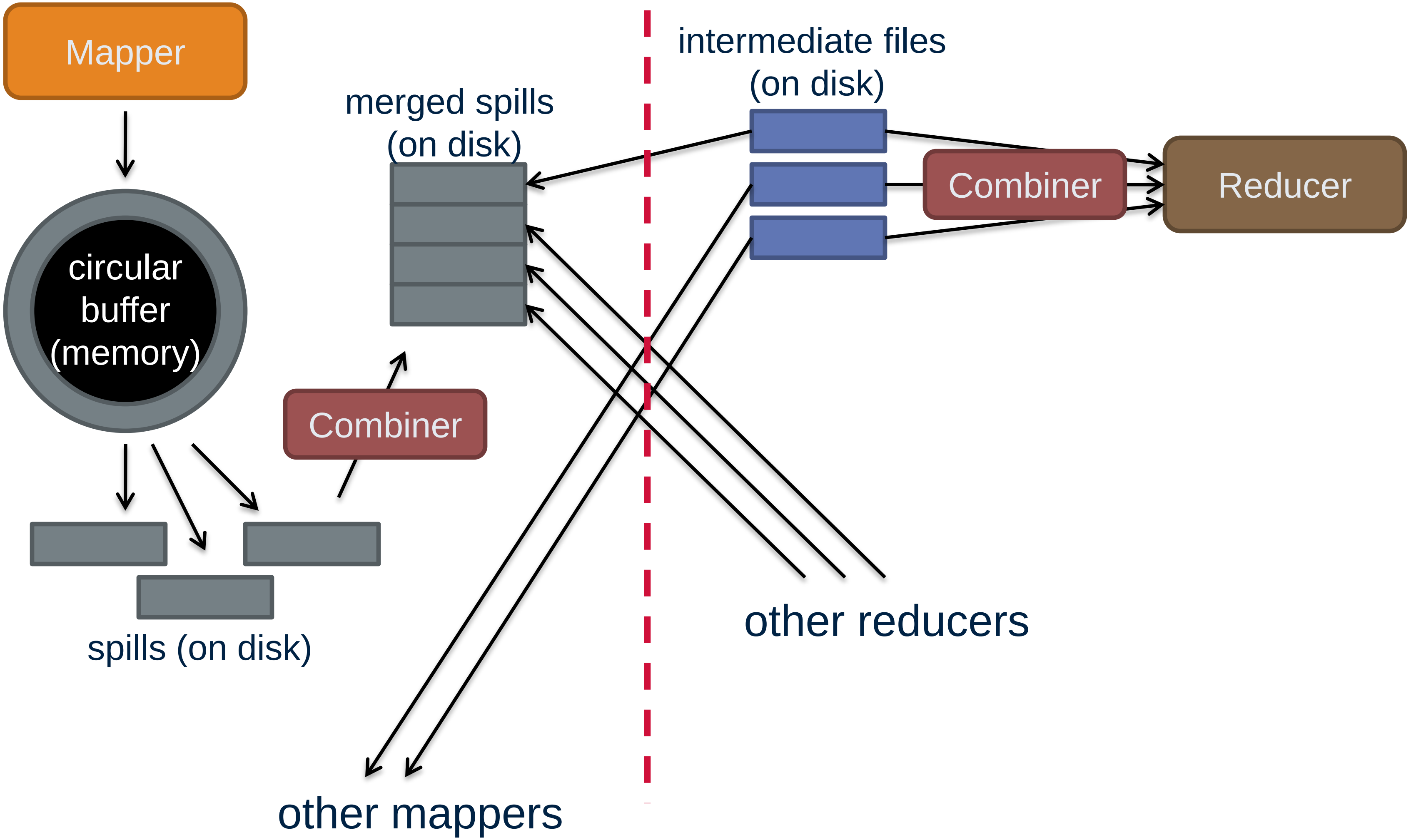
Complex data types in Hadoop

- How do you implement complex data types?
- The easiest way:
  - Encoded it as Text, e.g., (a, b) = “a:b”
  - Use regular expressions to parse and extract data
  - Works, but pretty hack-ish
- The hard way:
  - Define a custom implementation of Writable(Comparable)
  - Must implement: readFields, write, (compareTo)
  - Computationally efficient, but slow for rapid prototyping
  - Implement WritableComparator hook for performance
- Somewhere in the middle:
  - Some frameworks offers JSON support and lots of useful Hadoop types

Shuffle and sort in Hadoop

- Probably the most complex aspect of MapReduce
- Map side
  - Map outputs are buffered in memory in a circular buffer
  - When buffer reaches threshold, contents are spilled to disk
  - Spills merged in a single, partitioned file (sorted within each partition): combiner runs during the merges
- Reduce side
  - First, map outputs are copied over to reducer machine
  - Sort is a multi-pass merge of map outputs (happens in memory and on disk): combiner runs during the merges
  - Final merge pass goes directly into reducer

Shuffle and sort



Summary

- The difficulties of parallel programming
  - High-level frameworks to the rescue (Google MapReduce)
- Hadoop Architecture
  - MapReduce
  - HDFS
- MapReduce Programming
  - Word Count Examples
  - Optimization with combiners
  - Sequential access, Bulk Transfers
    - Vs small random accesses (memory, network,disk bottleneck)