Cristina Nita-Rotaru

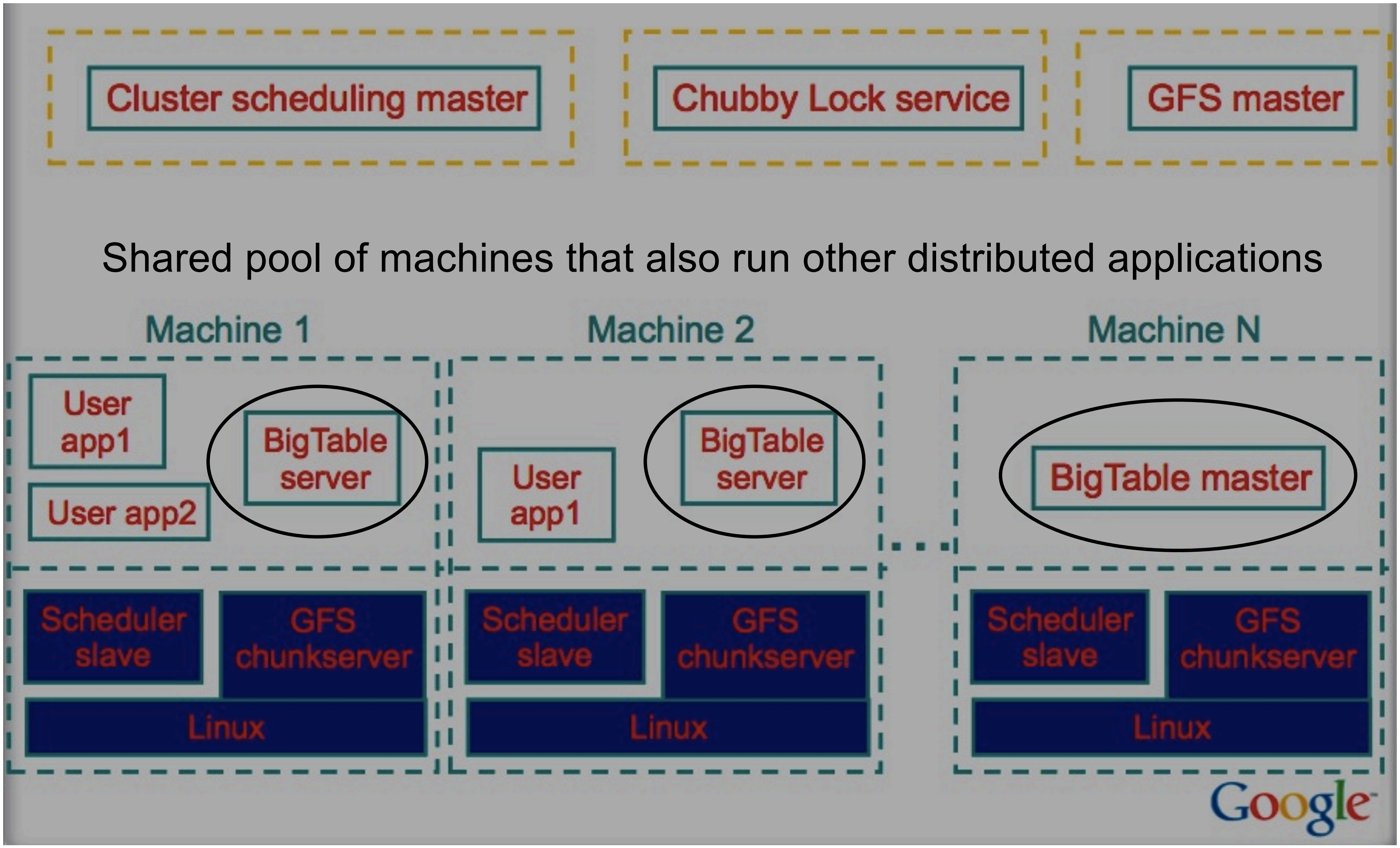
7610: Distributed Systems

MapReduce. Hadoop. Spark. Mesos. Yarn

REQUIRED READING

- MapReduce: Simplified Data Processing on Large Clusters OSDI 2004
- Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center, NSDI 2011
- Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing, NSDI 2012, best paper
- Apache Hadoop YARN: Yet Another Resource Negotiator SOCC 2013 (best paper)
- Omega: flexible, scalable schedulers for large compute clusters, EuroSys 2013 (best paper)

Typical Google Cluster



1: MapReduce

These are slides from Dan Weld's class at U. Washington (who in turn made his slides based on those by Jeff Dean, Sanjay Ghemawat, Google, Inc.)

Motivation

- Large-Scale Data Processing
  - Want to use 1000s of CPUs
    - But don't want hassle of managing things

- MapReduce provides
  - Automatic parallelization & distribution
  - Fault tolerance
  - I/O scheduling
  - Monitoring & status updates

Map/Reduce

- Map/Reduce
  - Programming model from Lisp
  - (and other functional languages)
- Many problems can be phrased this way
- Easy to distribute across nodes
- Nice retry/failure semantics

Map in Lisp (Scheme)

- (map f list [list2 list3 …])
- (map square ‘(1 2 3 4))
  - (1 4 9 16)
- (reduce + ‘(1 4 9 16))
  - (+ 16 (+ 9 (+ 4 1)))
- (reduce + (map square (map – l1 l2))))

Unary operator

Binary operator

Map/Reduce ala Google

- map(key, val) is run on each item in set
  - emits new-key / new-val pairs

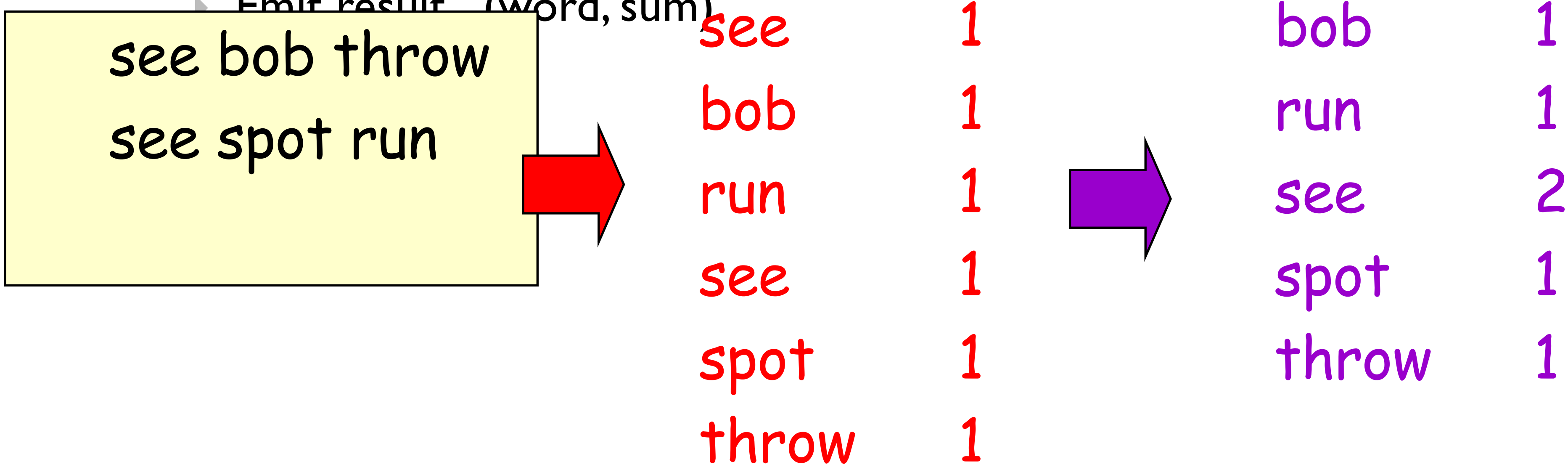
- reduce(key, vals) is run for each unique key emitted by map()
  - emits final output

count words in docs

- Input consists of (url, contents) pairs
- map(key = url, val = contents):
  - For each word w in contents, emit (w, "1")
- reduce(key = word, values = uniq_counts):
  - Sum all "1"s in values list
  - Emit result "(word, sum)"

Count, Illustrated

- map(key=url, val=contents):
  - For each word w in contents, emit (w, “1”)
- reduce(key=word, values=uniq_counts):
  - Sum all “1”s in values list
  - Emit result “(word, sum)”



Grep

- Input consists of (url+offset, single line)
- map(key=url+offset, val=line):
  - If contents matches regexp, emit (line, "1")
- reduce(key=line, values=uniq_counts):
  - Don't do anything; just emit line

Reverse Web-Link Graph

- Map
  - For each URL linking to target, …
  - Output <target, source> pairs
- Reduce
  - Concatenate list of all source URLs
  - Outputs: <target, list (source)> pairs

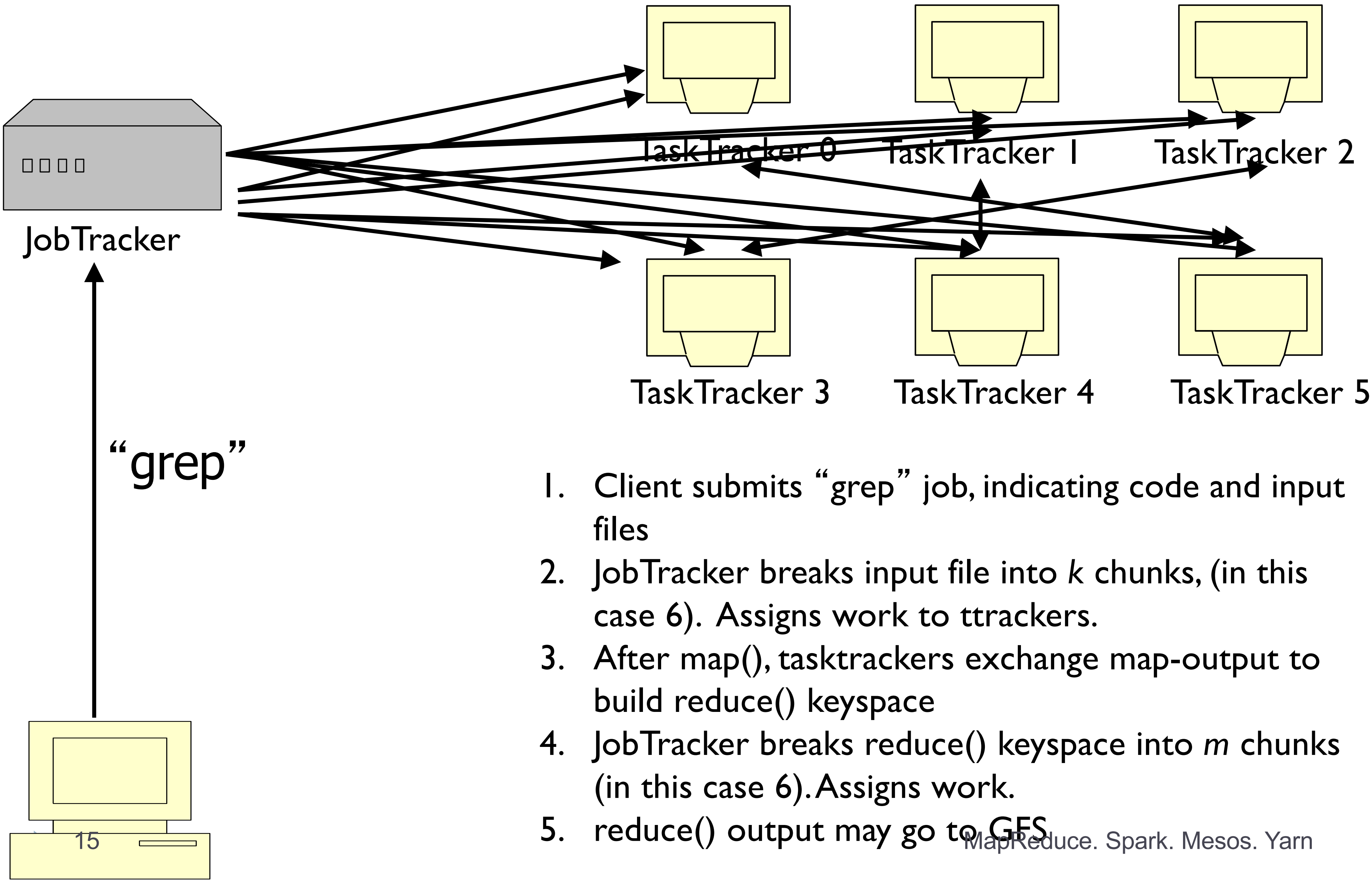
Implementation

- Typical cluster:
  - 100s/1000s of 2-CPU x86 machines, 2-4 GB of memory
  - Limited bisection bandwidth
  - Storage is on local IDE disks
  - GFS: distributed file system manages data
- Job scheduling system: jobs made up of tasks, scheduler assigns tasks to machines
- Implementation is a C++ library linked into user programs

Execution

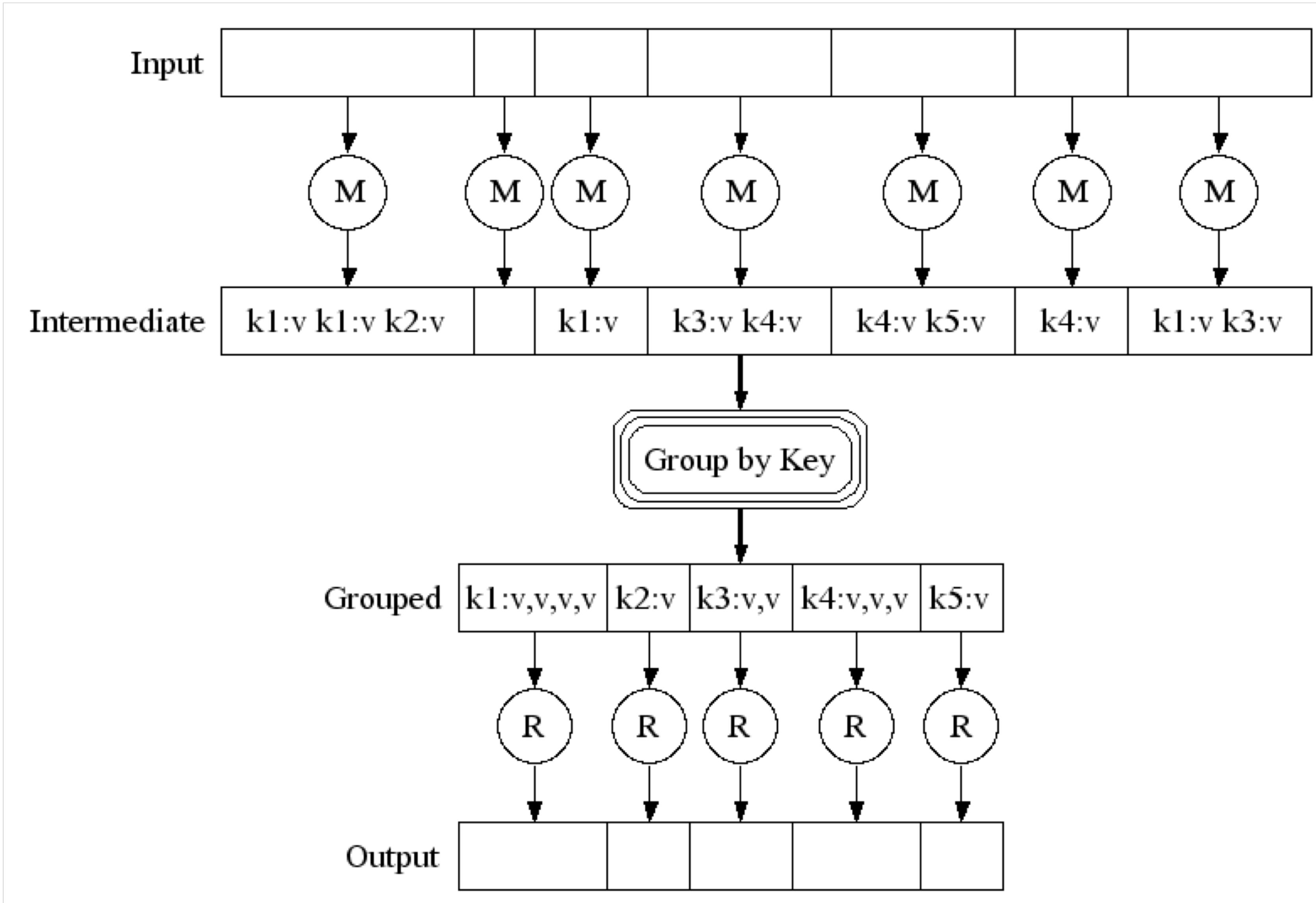
- How is this distributed?
  - Partition input key/value pairs into chunks, run map() tasks in parallel
  - After all map()s are complete, consolidate all emitted values for each unique emitted key
  - Now partition space of output map keys, and run reduce() in parallel
- If map() or reduce() fails, reexecute!

Job Processing

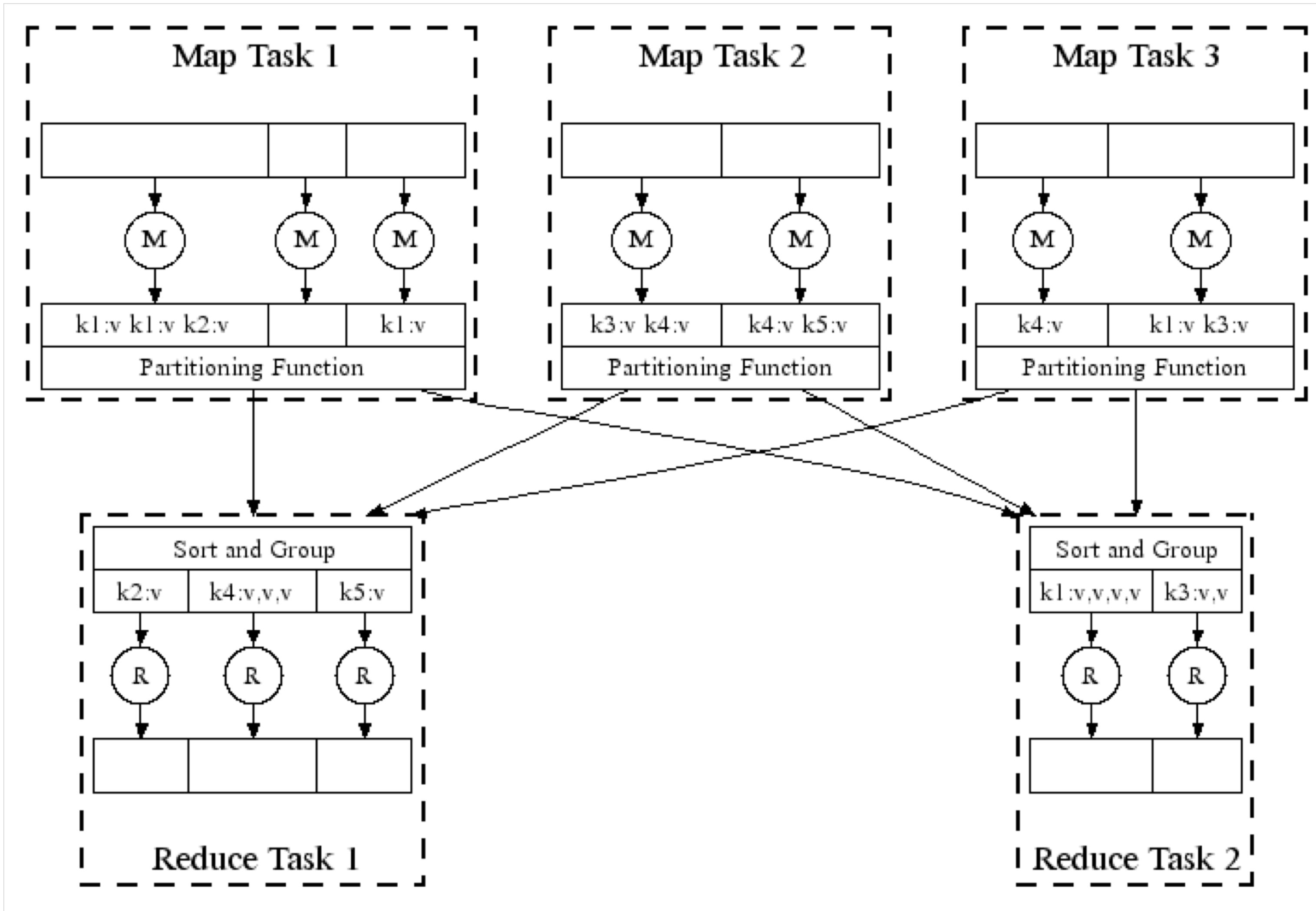


1. Client submits "grep" job, indicating code and input files
2. JobTracker breaks input file into *k* chunks, (in this case 6). Assigns work to ttrackers.
3. After map(), tasktrackers exchange map-output to build reduce() keyspace
4. JobTracker breaks reduce() keyspace into *m* chunks (in this case 6). Assigns work.
5. reduce() output may go to GFS

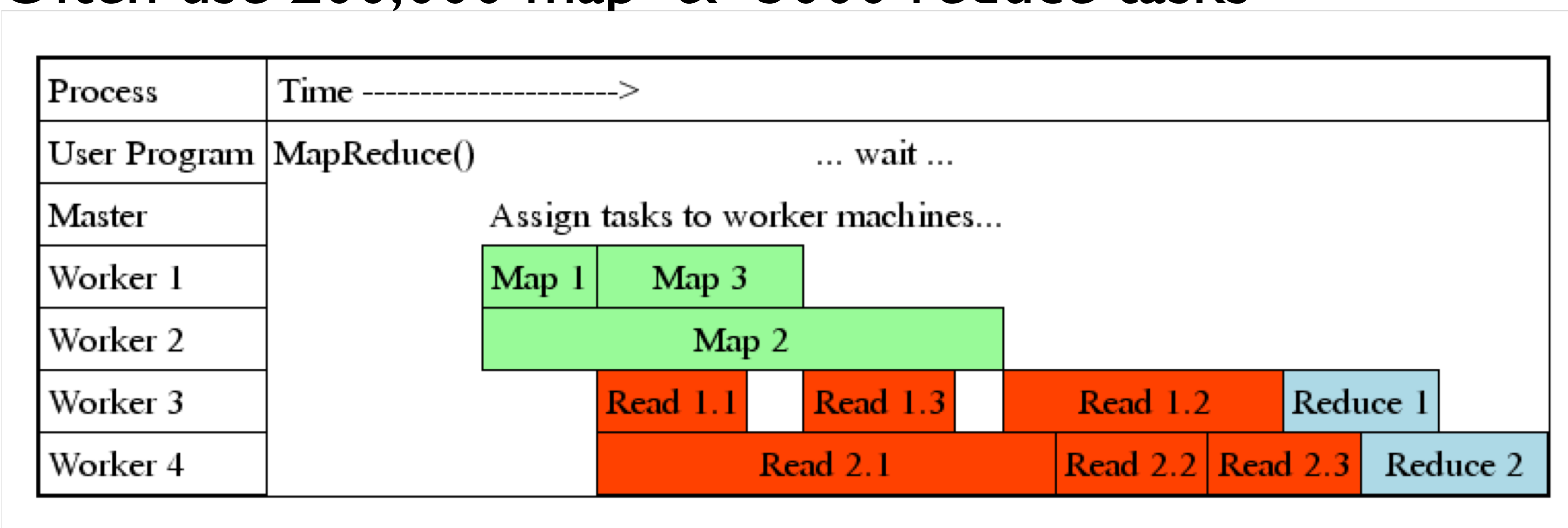
Execution


Input
M
M
M
M
M
M
M
Intermediate
k1:v k1:v k2:v
k1:v
k3:v k4:v
k4:v k5:v
k4:v
k1:v k3:v
Group by Key
Grouped
k1:v,v,v,v
k2:v
k3:v,v
k4:v,v,v
k5:v
R
R
R
R
R
Output

Parallel Execution



Task Granularity & Pipelining

- Fine granularity tasks: map tasks >> machines
  - Minimizes time for fault recovery
  - Can pipeline shuffling with map execution
  - Better dynamic load balancing
- Often use 200,000 map & 5000 reduce tasks
- 

Fault Tolerance / Workers

- Handled via re-execution
  - Detect failure via periodic heartbeats
  - Re-execute completed + in-progress map tasks
  - Re-execute in progress reduce tasks
  - Task completion committed through master
- Robust: lost 1600/1800 machines once → finished ok

Master Failure

- Could handle, … ?
- But don't yet
  - (master failure unlikely)

Refinement: Redundant Execution

Slow workers significantly delay completion time

- Other jobs consuming resources on machine
- Bad disks w/ soft errors transfer data slowly
- Weird things: processor caches disabled (!!)

Solution: Near end of phase, spawn backup tasks

- Whichever one finishes first "wins"

Dramatically shortens job completion time

Refinement: Locality Optimization

- Master scheduling policy:
  - Asks GFS for locations of replicas of input file blocks
  - Map tasks typically split into 64MB (GFS block size)
  - Map tasks scheduled so GFS input block replica are on same machine or same rack

- Effect
  - Thousands of machines read input at local disk speed
    - Without this, rack switches limit read rate

Refinement: Skipping Bad Records

- Map/Reduce functions sometimes fail for particular inputs
  - Best solution is to debug & fix
    - Not always possible ~ third-party source libraries
  - On segmentation fault:
    - Send UDP packet to master from signal handler
    - Include sequence number of record being processed
  - If master sees two failures for same record:
    - Next worker is told to skip the record

Other Refinements

- Sorting guarantees
  - within each reduce partition
- Compression of intermediate data
- Combiner
  - Useful for saving network bandwidth
- Local execution for debugging/testing
- User-defined counters

Performance

Tests run on cluster of 1800 machines:

- 4 GB of memory
- Dual-processor 2 GHz Xeons with Hyperthreading
- Dual 160 GB IDE disks
- Gigabit Ethernet per machine
- Bisection bandwidth approximately 100 Gbps

Two benchmarks:

MR_GrepScan 1010 100-byte records to extract records matching a rare pattern (92K matching records)

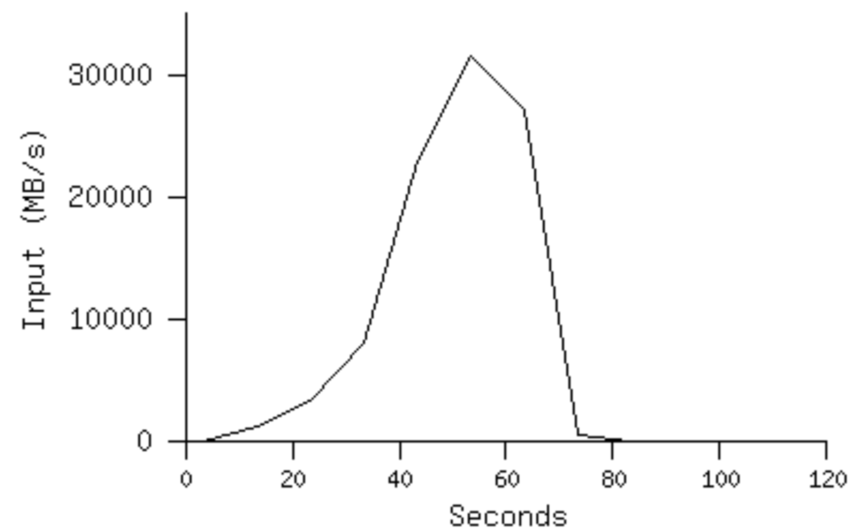
MR_SortSort 1010 100-byte records (modeled after TeraSort benchmark)

MR_Grep

Locality optimization helps:

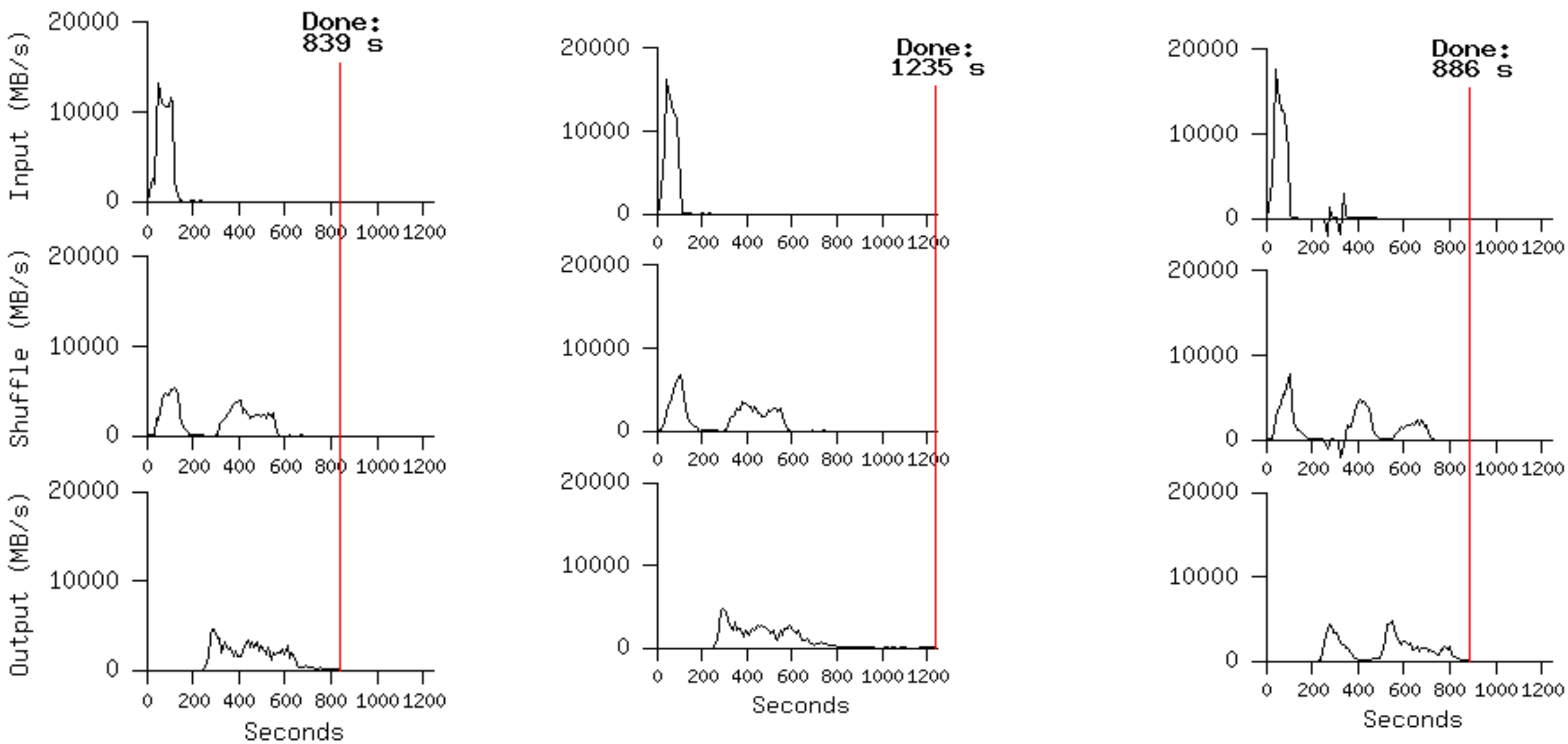
- 1800 machines read 1 TB at peak ~31 GB/s
- W/out this, rack switches would limit to 10 GB/s

Startup overhead is significant for short jobs



MR_Sort

Normal **No backup tasks** **200 processes killed**



- Backup tasks reduce job completion time a lot!
- System deals well with failures

2: Hadoop

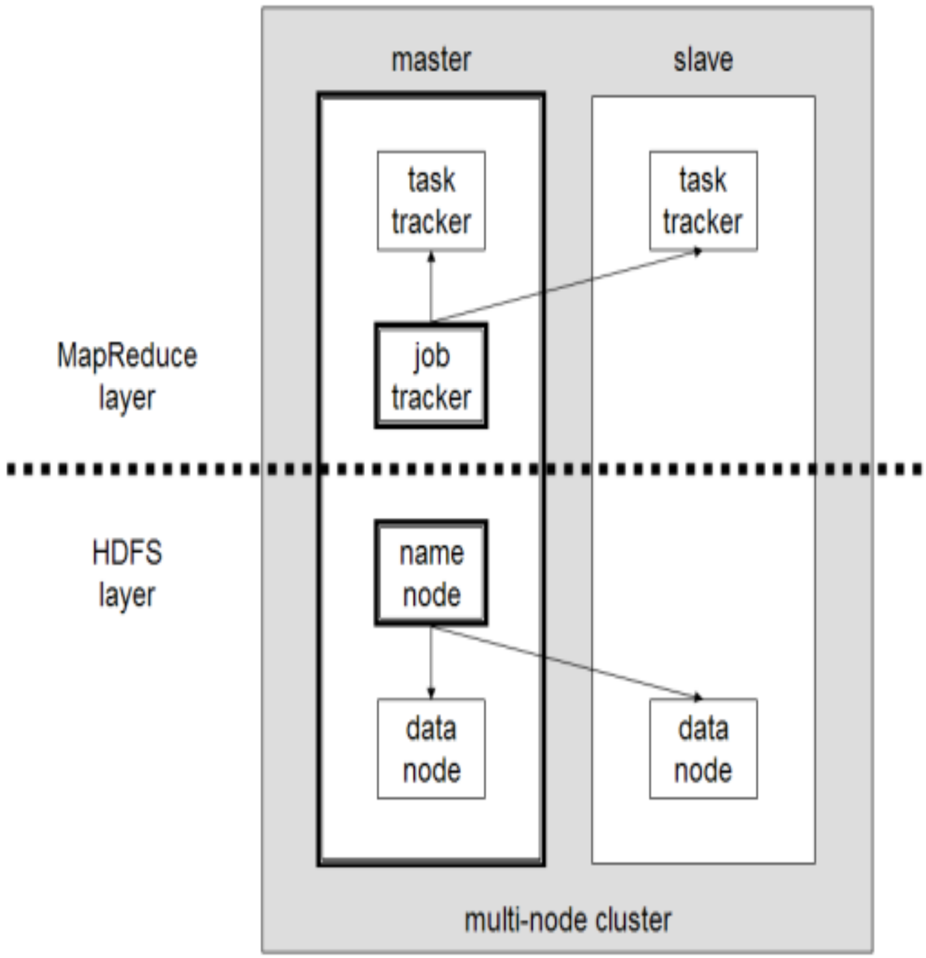
Apache Hadoop

- Apache Hadoop's MapReduce and HDFS components originally derived from
  - Google File System (GFS)[1] – 2003
  - Google's MapReduce[2] - 2004
- Data is broken in splits that are processed in different machines.
- Industry wide standard for processing Big Data.

Overview of Hadoop

- Basic components of Hadoop are:
  - **Map Reduce Layer**
    - **Job tracker** (master) -which coordinates the execution of jobs;
    - **Task trackers** (slaves)- which control the execution of map and reduce tasks in the machines that do the processing;
  - **HDFS Layer**- which stores files.
    - **Name Node** (master)- manages the file system, keeps metadata for all the files and directories in the tree
    - **Data Nodes** (slaves)- work horses of the file system. Store and retrieve blocks when they are told to (by clients or name node) and report back to name node periodically

Overview of Hadoop contd.



Job Tracker - coordinates the execution of jobs

Task Tracker – control the execution of map and reduce tasks in slave machines

Name Node – Manages the file system, keeps metadata

Data Node – Follow the instructions from name node
- stores, retrieves data

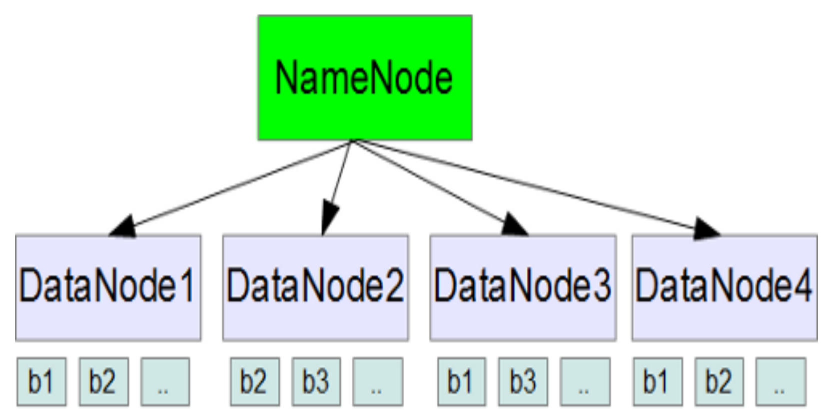
Hadoop Versions

Feature	1.x	0.22	2.x
Secure authentication	Yes	No	Yes
Old configuration names	Yes	Deprecated	Deprecated
New configuration names	No	Yes	Yes
Old MapReduce API	Yes	Yes	Yes
New MapReduce API	Yes (with somemissing libraries)	Yes	Yes
MapReduce 1 runtime (Classic)	Yes	Yes	No
MapReduce 2 runtime (YARN)	No	No	Yes
HDFS federation	No	No	Yes
HDFS high-availability	No	No	Yes

- MapReduce 2 runtime and HDFS HA was introduced in Hadoop 2.x

Fault Tolerance in HDFS layer

- Hardware failure is the norm rather than the exception
- **Detection of faults and quick, automatic recovery from them** is a core architectural goal of HDFS.
- Master Slave Architecture with NameNode (master) and DataNode (slave)
- Common types of failures
  - NameNode failures
  - DataNode failures



Handling Data Node Failure

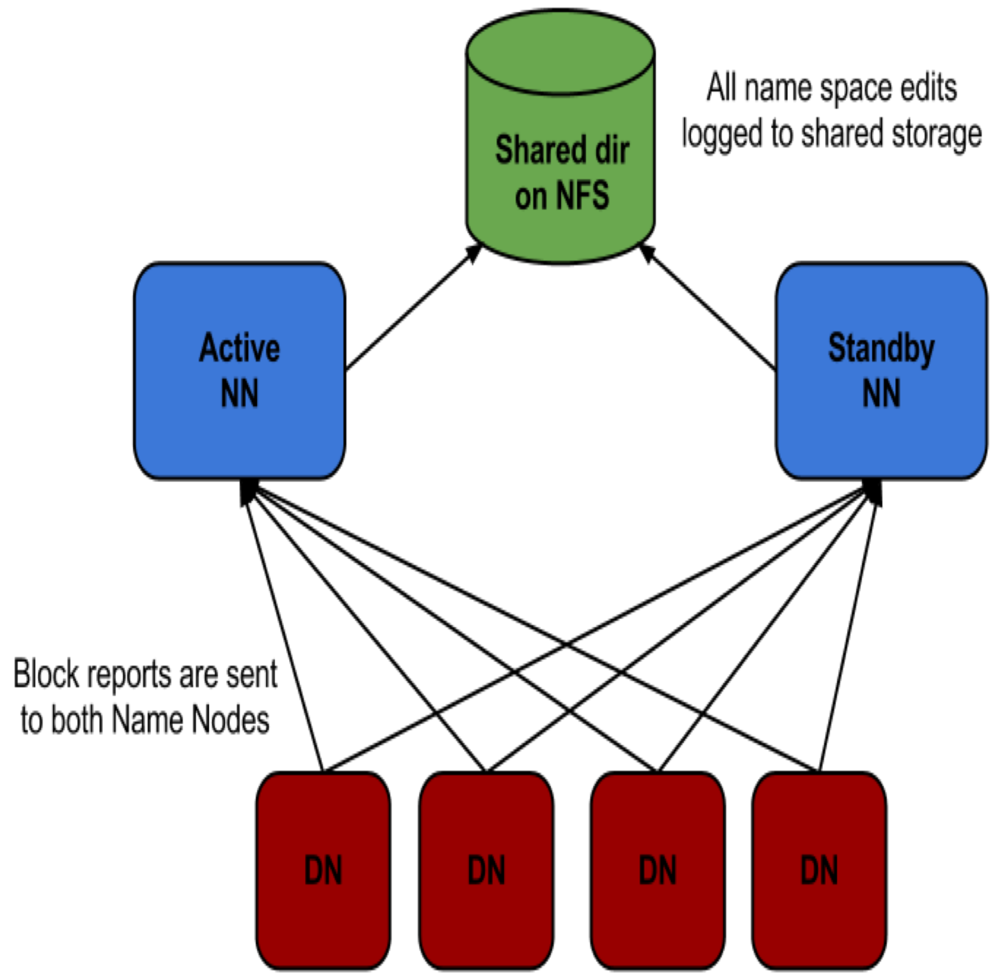
- Each DataNode sends a Heartbeat message to the NameNode periodically
- If the namenode does not receive a heartbeat from a particular data node for 10 minutes, then it considers that data node to be dead/out of service.
- Name Node initiates replication of blocks which were hosted on that data node to be hosted on some other data node.

Handling Name Node Failure

- Single Name Node per cluster.
- Prior to Hadoop 2.0.0, the NameNode was a single point of failure (SPOF) in an HDFS cluster.
- If NameNode becomes unavailable, the cluster as a whole would be unavailable
  - NameNode has to be restarted
  - Brought up on a separate machine.

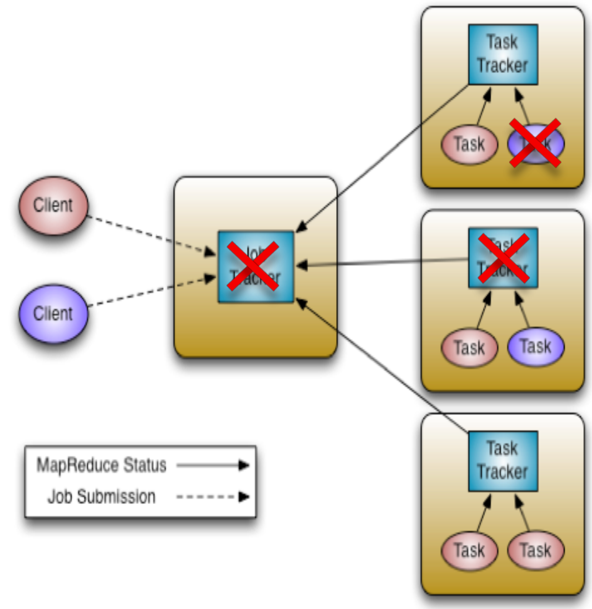
HDFS High Availability

- Provides an option of running two redundant NameNodes in the same cluster
- Active/Passive configuration with a hot standby.
- Fast failover to a new NameNode in the case that a machine crashes
- Graceful administrator-initiated failover for the purpose of planned maintenance.



Classic MapReduce (v1)

- Job Tracker
  - Manage Cluster Resources and Job Scheduling
- Task Tracker
  - Per-node agent
  - Manage Tasks
- Jobs can fail
  - While running the task (Task Failure)
  - Task Tracker failure
  - Job Tracker failure



Handling Task Failure

- User code bug in map/reduce
  - Throws a RunTimeException
  - Child JVM reports a failure back to the parent task tracker before it exits.
- Sudden exit of the child JVM
  - Bug that causes the JVM to exit for the conditions exposed by map/reduce code.
- Task tracker marks the task attempt as failed, makes room available to another task.

Task Tracker Failure

- Task tracker will stop sending the heartbeat to the Job Tracker
- Job Tracker notices this failure
  - Hasn't received a heart beat from 10 mins
  - Can be configured via mapred.tasktracker.expiry.interval property
- Job Tracker removes this task from the task pool
- Rerun the Job even if map task has ran completely
  - Intermediate output resides in the failed task trackers local file system which is not accessible by the reduce tasks.

Job Tracker Failure

- This is serious than the other two modes of failure.
  - Single point of failure.
  - In this case all jobs will fail.
- After restarting Job Tracker all the jobs running at the time of the failure needs to be resubmitted.

3. Spark

Slides by Matei Zaharia, UC Berkeley

Motivation

- Map reduce based tasks are slow
  - Sharing of data across jobs is stable storage
  - Replication of data and disk I/O
- Support iterative algorithms
- Support interactive data mining tools – search

Existing literature on large distributed algorithms on clusters

- General : Language-integrated "distributed dataset" API, but cannot share datasets efficiently across queries
  - Map Reduce
    - Map
    - Shuffle
    - Reduce
  - DyradLinq
  - Ciel
- Specific : Specialized models; can't run arbitrary / ad-hoc queries
  - Pregel – Google's graph based
  - Haloop – iterative Hadoop

- (Cont)
- Caching systems
  - Nectar – Automatic expression caching, but over distributed FS
  - Ciel – not explicit control over cached data
  - PacMan - Memory cache for HDFS, but writes still go to network/disk
- Lineage
  - To track dependency of task information across a DAG of tasks

Resilient Distributed Datasets (RDDs)

- Restricted form of distributed shared memory
  - Read only/ Immutable , partitioned collections of records
  - Deterministic
  - From coarse grained operations (map, filter, join, etc.)
  - From stable storage or other RDDs
  - User controlled persistence
  - User controlled partitioning

Representing RDDs

Operation	Meaning
partitions()	Return a list of Partition objects
preferredLocations(p)	List nodes where partition p can be accessed faster due to data locality
dependencies()	Return a list of dependencies
iterator(p, *parentIters*)	Compute the elements of partition p given iterators for its parent partitions
partitioner()	Return metadata specifying whether the RDD is hash/range partitioned

Table 3: Interface used to represent RDDs in Spark.

No need of check pointing

Checkpoint
ing

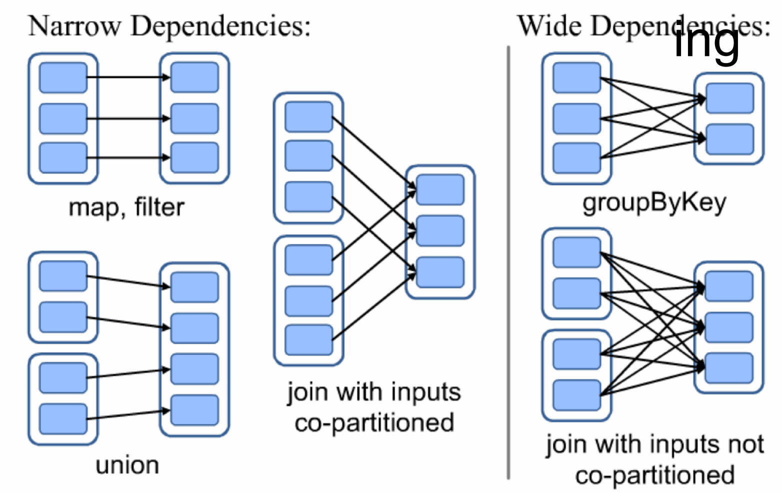


Figure 4: Examples of narrow and wide dependencies. Each box is an RDD, with partitions shown as shaded rectangles.

Spark programming interface

- Lazy operations
  - Transformations not done until action
- Operations on RDDs
  - Transformations - build new RDDs
  - Actions - compute and output results
- Partitioning – layout across nodes
- Persistence – storage in RAM / Disc

RDD on Spark

Transformations	$map(f : \mathrm{T} \Rightarrow \mathrm{U})$	:	RDD[T] ⇒ RDD[U]
	$filter(f : \mathrm{T} \Rightarrow \mathrm{Bool})$	:	RDD[T] ⇒ RDD[T]
	$flatMap(f : \mathrm{T} \Rightarrow \mathrm{Seq[U]})$	:	RDD[T] ⇒ RDD[U]
	$sample(fraction : \mathrm{Float})$	:	RDD[T] ⇒ RDD[T] (Deterministic sampling)
	$groupByKey()$	:	RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
	$reduceByKey(f : (\mathrm{V}, \mathrm{V}) \Rightarrow \mathrm{V})$	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	$union()$	:	(RDD[T], RDD[T]) ⇒ RDD[T]
	$join()$	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]
	$cogroup()$	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]
	$crossProduct()$	:	(RDD[T], RDD[U]) ⇒ RDD[(T, U)]
	$mapValues(f : \mathrm{V} \Rightarrow \mathrm{W})$	:	RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)
	$sort(c : \mathrm{Comparator[K]})$	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	$partitionBy(p : \mathrm{Partitioner[K]})$	:	RDD[(K, V)] ⇒ RDD[(K, V)]
Actions	$count()$	:	RDD[T] ⇒ Long
	$collect()$	:	RDD[T] ⇒ Seq[T]
	$reduce(f : (\mathrm{T}, \mathrm{T}) \Rightarrow \mathrm{T})$	:	RDD[T] ⇒ T
	$lookup(k : \mathrm{K})$	:	RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs)
	$save(path : \mathrm{String})$	:	Outputs RDD to a storage system, *e.g.,* HDFS

Table 2: Transformations and actions available on RDDs in Spark. Seq[T] denotes a sequence of elements of type T.

Example : Console Log mining

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
errors.persist()
```

```
errors.count()
```

```
// Count errors mentioning MySQL:
errors.filter(_.contains("MySQL")).count()

// Return the time fields of errors mentioning
// HDFS as an array (assuming time is field
// number 3 in a tab-separated format):
errors.filter(_.contains("HDFS"))
      .map(_.split('\t')(3))
      .collect()
```

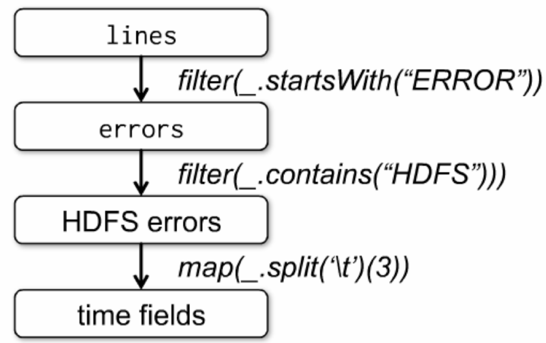


Figure 1: Lineage graph for the third query in our example. Boxes represent RDDs and arrows represent transformations.

Example : Logistic regression

- Classification problem that searches for hyper plane w

```
val points = spark.textFile(...)
                  .map(parsePoint).persist()
var w = // random initial vector
for (i <- 1 to ITERATIONS) {
  val gradient = points.map{ p =>
    p.x * (1/(1+exp(-p.y*(w dot p.x)))-1)*p.y
  }.reduce((a,b) => a+b)
  w -= gradient
}
```

Transforms text to point object

Repetitive map and reduce to compute gradient

Example : PageRank

- Start each page with rank 1/N.
- On each iteration update the page rank
  - $= \Sigma_{i \in neighbors} \; rank_i \, / \, |neighbors|$

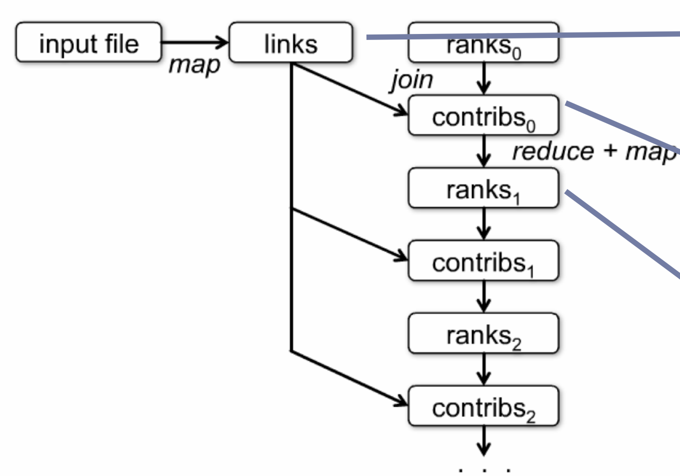
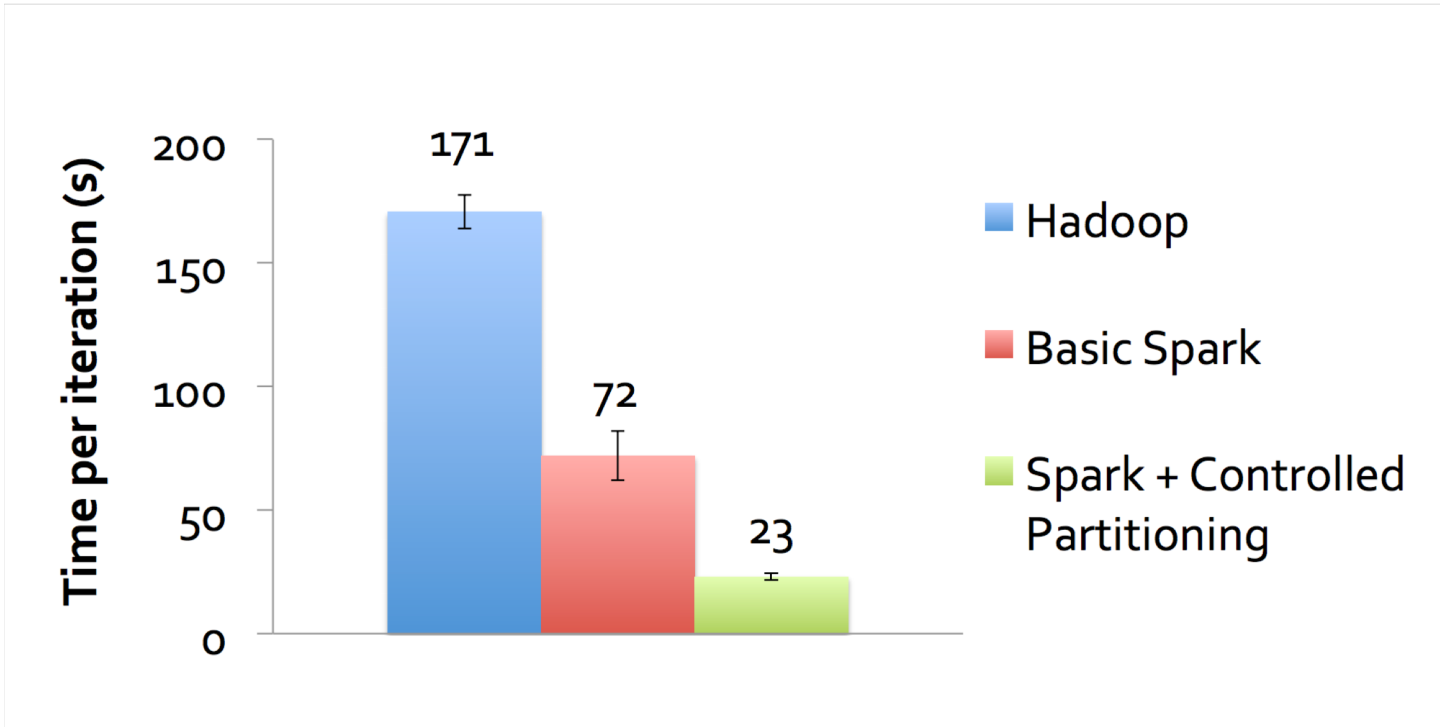


Figure 3: Lineage graph for datasets in PageRank.

```
val links = spark.textFile(...).map(...).persist()
var ranks = // RDD of (URL, rank) pairs
for (i <- 1 to ITERATIONS) {
  // Build an RDD of (targetURL, float) pairs
  // with the contributions sent by each page
  val contribs = links.join(ranks).flatMap {
    (url, (links, rank)) =>
      links.map(dest => (dest, rank/links.size))
  }
  // Sum contributions by URL and get new ranks
  ranks = contribs.reduceByKey((x,y) => x+y)
             .mapValues(sum => a/N + (1-a)*sum)
}
```

PageRank performance



RDDs versus DSMs

Lookup by key

Aspect	RDDs	Distr. Shared Mem.
Reads	Coarse- or fine-grained	Fine-grained
Writes	Coarse-grained	Fine-grained
Consistency	Trivial (immutable)	Up to app / runtime
Fault recovery	Fine-grained and low-overhead using lineage	Requires checkpoints and program rollback
Straggler mitigation	Possible using backup tasks	Difficult
Work placement	Automatic based on data locality	Up to app (runtimes aim for transparency)
Behavior if not enough RAM	Similar to existing data flow systems	Poor performance (swapping?)

Table 1: Comparison of RDDs with distributed shared memory.

RDDs unsuitable for applications that mak
fine- grained updates to shared state,
-storage system for a web application
-an incremental web crawler

Implementation in Spark

- Job scheduler
  - Data locality captured using delay scheduling
- Interpreter integration
  - Class shipping
  - Modified code generation
- Memory management
  - In memory and swap memory
  - LRU
- Support for checkpointing
  - Good for long lineage graphs

Evaluation

- Runs on Mesos to share clusters with Hadoop
- Can read from any Hadoop input source (HDFS or HBase)
- RDD implemented in Spark
  - Ability to be used over any other cluster systems as well

Iterative ML applications

scalability

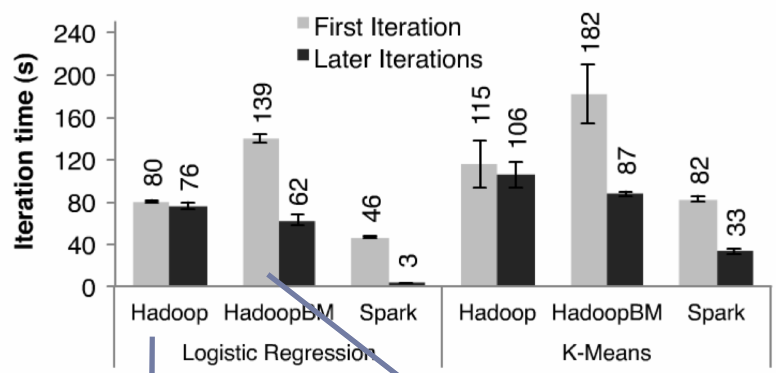
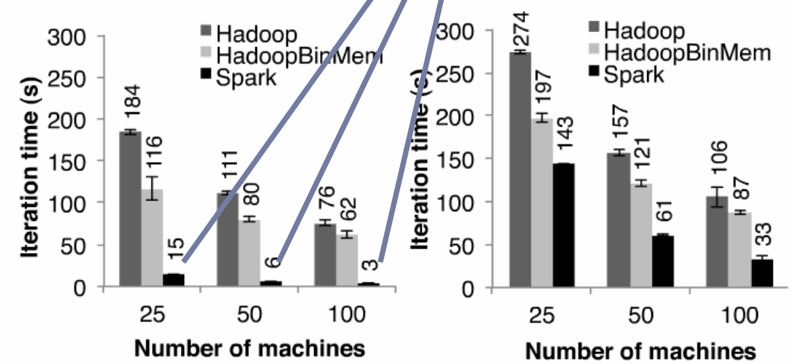


Figure 7: Duration of the first and later iterations in Hadoop, HadoopBinMem and Spark for logistic regression and k-means using 100 GB of data on a 100-node cluster.



(a) Logistic Regression (b) K-Means

Figure 8: Running times for iterations after the first in Hadoop, HadoopBinMem, and Spark. The jobs all processed 100 GB.

No improvement in successive iterations
Slow due to heartbeat signals

Initially slow due to conversion of text to binary in-Mem a

Understanding Speedup

Reading from HDFS costs 2 second

10 second difference

Text to binary parsing = 7 s

Conversion of binary recor

3sec

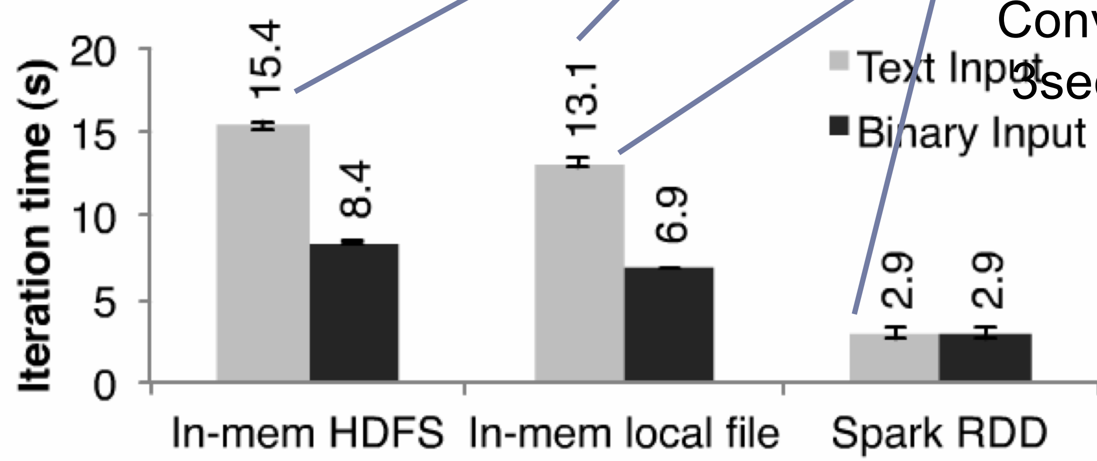


Figure 9: Iteration times for logistic regression using 256 MB data on a single machine for different sources of input.

Failure in RDD

RDDs track the graph of transformations that built them (their lineage) to rebuild lost data

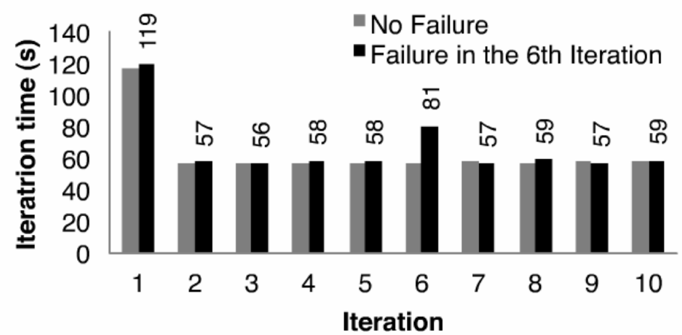


Figure 11: Iteration times for k-means in presence of a failure. One machine was killed at the start of the 6th iteration, resulting in partial reconstruction of an RDD using lineage.

In sufficient memory

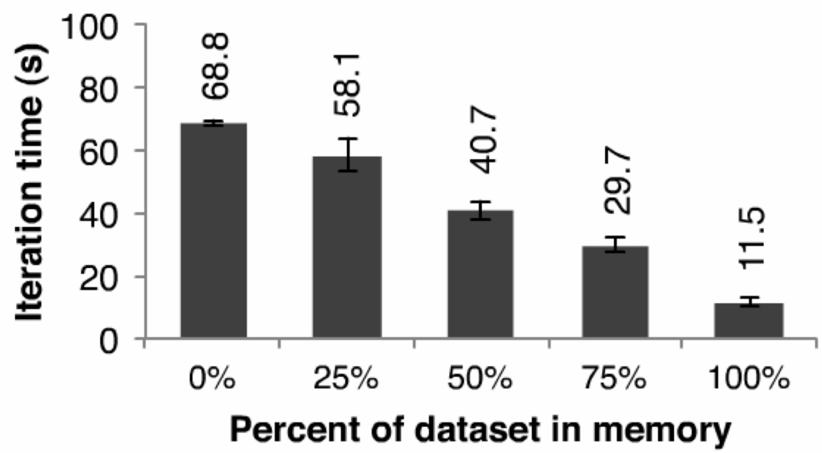
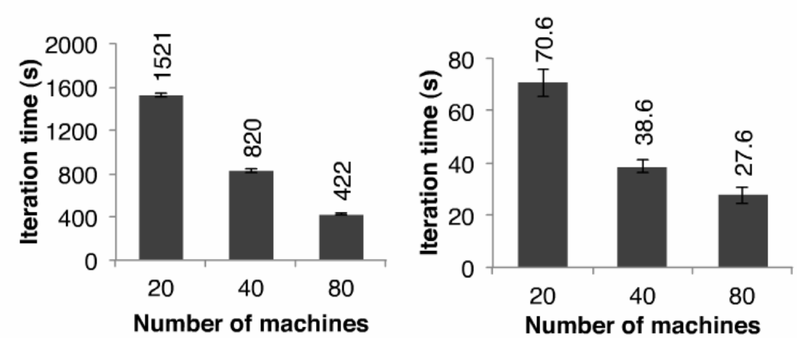


Figure 12: Performance of logistic regression using 100 GB data on 25 machines with varying amounts of data in memory.

User applications using Spark

- In memory analytics at Conviva : 40x speedup
- Traffic modeling (Traffic prediction via EM - Mobile Millennium)
- Twitter spam classification(Mo
- DNA sequence analysis (SN



(a) Traffic modeling (b) Spam classification

Figure 13: Per-iteration running time of two user applications implemented with Spark. Error bars show standard deviations.

RDDs

- Good
  - RDDs offer a simple and efficient programming model
  - Open source and scalable implementation at Spark
  - Improves the speed to the memory bandwidth limit – good for batch processes
- Improvements
  - Memory leak if too many RDDs loaded - garbage collection to be built in
    - Uses LRU – better memory replacement algorithms possible
  - Handling data locality using partition/hash and delay scheduling
  - Hybrid system for handling fine grained updates
  - Use for debugging

What is Spark?

- Fast, expressive cluster computing system compatible with Apache Hadoop
  - Works with any Hadoop-supported storage system (HDFS, S3, Avro, …)
- Improves **efficiency** through: → Up to 100 × faster
  - In-memory computing primitives
  - General computation graphs
- Improves **usability** through: → Often 2-10 × less code
  - Rich APIs in Java, Scala, Python
  - Interactive shell

Key Idea

- **Work with distributed collections as you would with local ones**

- Concept: resilient distributed datasets (RDDs)
  - Immutable collections of objects spread across a cluster
  - Built through parallel transformations (map, filter, etc)
  - Automatically rebuilt on failure
  - Controllable persistence (e.g. caching in RAM)

Operations

- Transformations (e.g. map, filter, groupBy, join)
  - Lazy operations to build RDDs from other RDDs
- Actions (e.g. count, collect, save)
  - Return a result or write it to storage

Example: Mining Console Logs

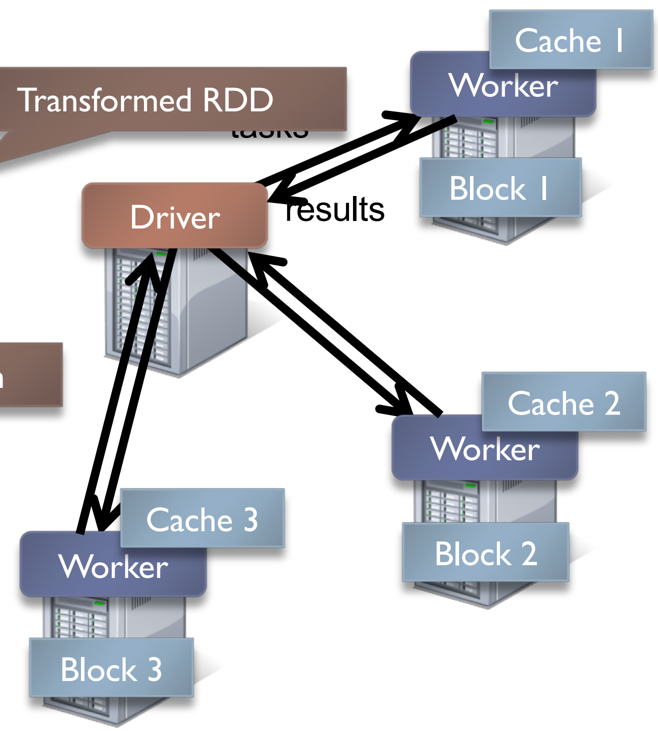
- Load error messages from a log into memory, then interactively search for patterns

Base RDD

Transformed RDD

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split('\t')[2])
messages.cache()

messages.filter(lambda s: "foo" in s).count()
messages.filter(lambda s: "bar" in s).count()
. . .
```

Action



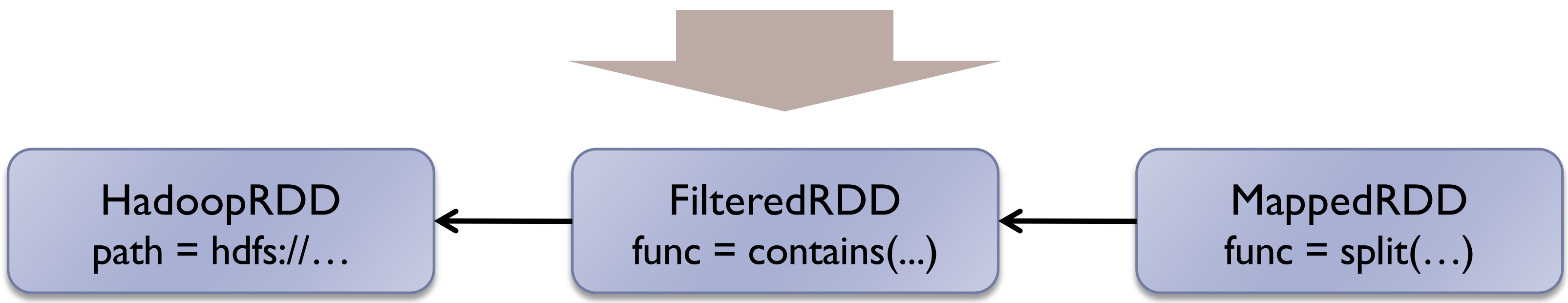
> **Result:** scaled to 1 TB data in 5-7 sec
> (vs 170 sec for on-disk data)

RDD Fault Tolerance

RDDs track the transformations used to build them (their *lineage*) to recompute lost data

E.g:

```
messages = textFile(...).filter(lambda s: s.contains("ERROR"))
                        .map(lambda s: s.split('\t')[2])
```



Spark in Java and Scala

Java API:

```
JavaRDD<String> lines = spark.textFile(…);

errors = lines.filter(
  new Function<String, Boolean>() {
    public Boolean call(String s) {
      return s.contains(“ERROR”);
    }
});

errors.count()
```

Scala API:

```
val lines = spark.textFile(…)

errors = lines.filter(s =>
s.contains(“ERROR”))
// can also write filter(_.contains(“ERROR”))

errors.count
```

Which Language Should I Use?

- Standalone programs can be written in any, but console is only Python & Scala
- **Python developers:** can stay with Python for both
- **Java developers:** consider using Scala for console (to learn the API)

- Performance: Java / Scala will be faster (statically typed), but Python can do well for numerical work with NumPy

Scala Cheat Sheet

Variables:

```
var x: Int = 7
var x = 7      // type inferred

val y = “hi”   // read-only
```

Functions:

```
def square(x: Int): Int = x*x

def square(x: Int): Int = {
  x*x   // last line returned
}
```

Collections and closures:

```
val nums = Array(1, 2, 3)

nums.map((x: Int) => x + 2) // => Array(3, 4, 5)

nums.map(x => x + 2)  // => same
nums.map(_ + 2)       // => same

nums.reduce((x, y) => x + y) // => 6
nums.reduce(_ + _)           // => 6
```

Java interop:

```
import java.net.URL

new URL(“http://cnn.com”).openStream()
```

More details:
scala-lang.org

Outline

- Introduction to Spark
- Tour of Spark operations
- Job execution
- Standalone programs
- Deployment options

Learning Spark

- Easiest way: Spark interpreter (`spark-shell` or `pyspark`)
  - Special Scala and Python consoles for cluster use
- Runs in local mode on 1 thread by default, but can control with MASTER environment var:

```
MASTER=local       ./spark-shell          # local, 1 thread
MASTER=local[2]    ./spark-shell          # local, 2 threads
MASTER=spark://host:port ./spark-shell    # Spark standalone cluster
```

First Stop: SparkContext

- Main entry point to Spark functionality
- Created for you in Spark shells as variable `sc`
- In standalone programs, you'd make your own (see later for details)

Creating RDDs

```
# Turn a local collection into an RDD
sc.parallelize([1, 2, 3])

# Load text file from local FS, HDFS, or S3
sc.textFile("file.txt")
sc.textFile("directory/*.txt")
sc.textFile("hdfs://namenode:9000/path/file")

# Use any existing Hadoop InputFormat
sc.hadoopFile(keyClass, valClass, inputFmt,
conf)
```

Basic Transformations

```
nums = sc.parallelize([1, 2, 3])

# Pass each element through a function
squares = nums.map(lambda x: x*x)   # => {1,
4, 9}

# Keep elements passing a predicate
even = squares.f                x % 2 == 0) #
=> {4}

# Map each element to zero or more others
nums.flatMap(lambda x: range(0, x))  # => {0,
```

Range object (sequence of numbers 0, 1, …, x-1)

Basic Actions

```
nums = sc.parallelize([1, 2, 3])

# Retrieve RDD contents as a local collection
nums.collect() # => [1, 2, 3]

# Return first K elements
nums.take(2)   # => [1, 2]

# Count number of elements
nums.count()   # => 3

# Merge elements with an associative function
nums.reduce(lambda x, y: x + y)  # => 6

# Write elements to a text file
nums.saveAsTextFile("hdfs://file.txt")
```

Working with Key-Value Pairs

- Spark's "distributed reduce" transformations act on RDDs of *key-value pairs*
- Python:

```python
pair = (a, b)
pair[0] # => a
pair[1] # => b
```

- Scala:

```scala
val pair = (a, b)
pair._1 // => a
pair._2 // => b
```

- Java:

```java
Tuple2 pair = new Tuple2(a, b);  // class scala.Tuple2
pair._1 // => a
pair._2 // => b
```

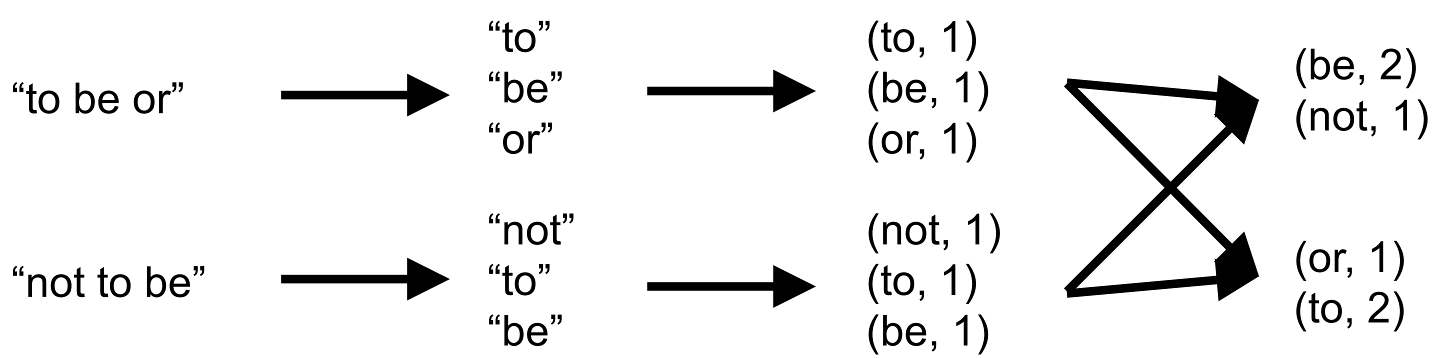
Some Key-Value Operations

```
pets = sc.parallelize([(“cat”, 1), (“dog”, 1), (“cat”, 2)])

pets.reduceByKey(lambda x, y: x + y)
# => {(cat, 3), (dog, 1)}

pets.groupByKey()
# => {(cat, Seq(1, 2)), (dog, Seq(1)}

pets.sortByKey()
# => {(cat, 1), (cat, 2), (dog, 1)}
```

reduceByKey also automatically implements combiners on the map side

Example: Word Count

```
lines = sc.textFile("hamlet.txt")
counts = lines.flatMap(lambda line: line.split(" ")) \
              .map(lambda word: (word, 1)) \
              .reduceByKey(lambda x, y: x + y)
```



Multiple Datasets

```
visits = sc.parallelize([(“index.html”, “1.2.3.4”),
                         (“about.html”, “3.4.5.6”),
                         (“index.html”, “1.3.3.1”)])

pageNames = sc.parallelize([(“index.html”, “Home”), (“about.html”, “About”)])

visits.join(pageNames)
# (“index.html”, (“1.2.3.4”, “Home”))
# (“index.html”, (“1.3.3.1”, “Home”))
# (“about.html”, (“3.4.5.6”, “About”))

visits.cogroup(pageNames)
# (“index.html”, (Seq(“1.2.3.4”, “1.3.3.1”), Seq(“Home”)))
# (“about.html”, (Seq(“3.4.5.6”), Seq(“About”)))
```

Controlling the Level of Parallelism

- All the pair RDD operations take an optional second parameter for number of tasks

```
words.reduceByKey(lambda x, y: x + y, 5)
words.groupByKey(5)
visits.join(pageViews, 5)
```

Using Local Variables

- External variables you use in a closure will automatically be shipped to the cluster:

```
query = raw_input("Enter a query:")
pages.filter(lambda x: x.startswith(query)).count()
```

- Some caveats:
  - Each task gets a new copy (updates aren't sent back)
  - Variable must be Serializable (Java/Scala) or Pickle-able (Python)
  - Don't use fields of an outer object (ships all of it!)

Closure Mishap Example

```
class MyCoolRddApp {
  val param = 3.14
  val log = new Log(...)
  ...

  def work(rdd: RDD[Int]) {
    rdd.map(x => x + param)
       .reduce(...)
  }
}
```

NotSerializableException: MyCoolRddApp (or Log)

How to get around it:

```
class MyCoolRddApp {
  ...

  def work(rdd: RDD[Int]) {
    val param_ = param
    rdd.map(x => x + param_)
       .reduce(...)
  }
}
```

References only local variable instead of `this.param`

More Details

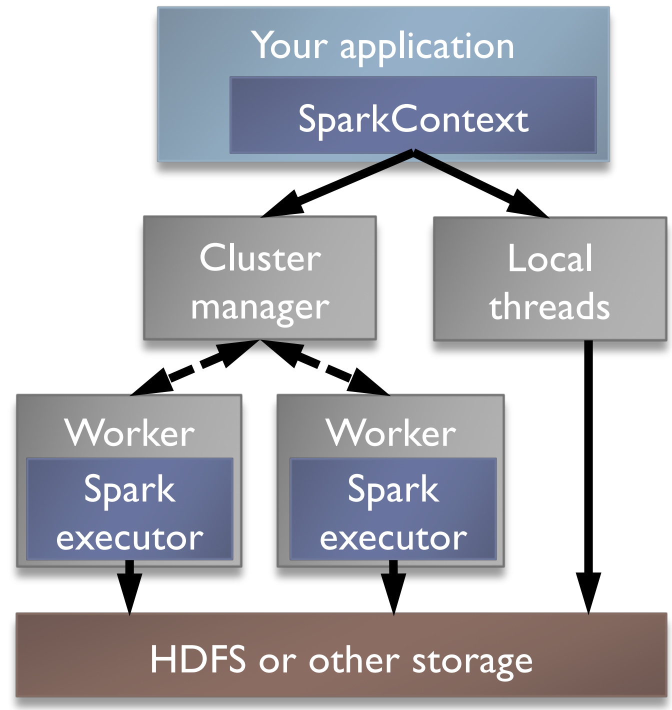
- Spark supports lots of other operations!
- Full programming guide: spark-project.org/documentation

Outline

- Introduction to Spark
- Tour of Spark operations
- Job execution
- Standalone programs
- Deployment options

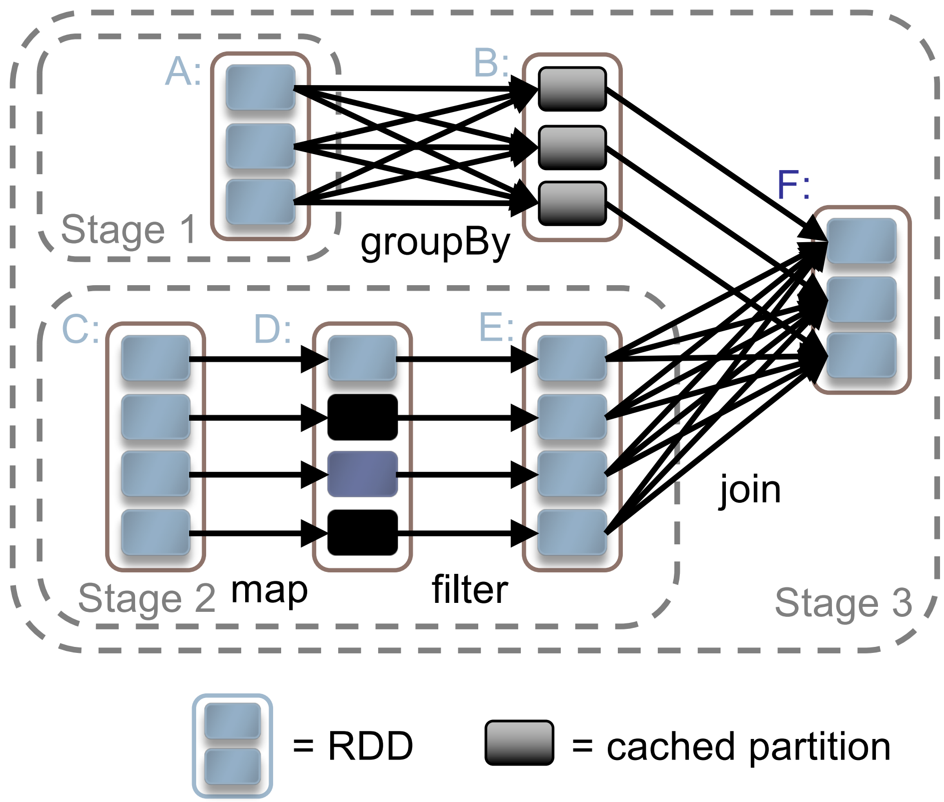
Software Components

- Spark runs as a library in your program
  (one instance per app)
- Runs tasks locally or on a cluster
  - Standalone deploy cluster, Mesos or YARN
- Accesses storage via Hadoop InputFormat API
  - Can use HBase, HDFS, S3, …



Task Scheduler

- Supports general task graphs
- Pipelines functions where possible
- Cache-aware data reuse & locality
- Partitioning-aware to avoid shuffles



Hadoop Compatibility

- Spark can read/write to any storage system / format that has a plugin for Hadoop!
  - Examples: HDFS, S3, HBase, Cassandra, Avro, SequenceFile
  - Reuses Hadoop's InputFormat and OutputFormat APIs
- APIs like `SparkContext.textFile` support filesystems, while `SparkContext.hadoopRDD` allows passing any Hadoop JobConf to configure an input source

3: Mesos

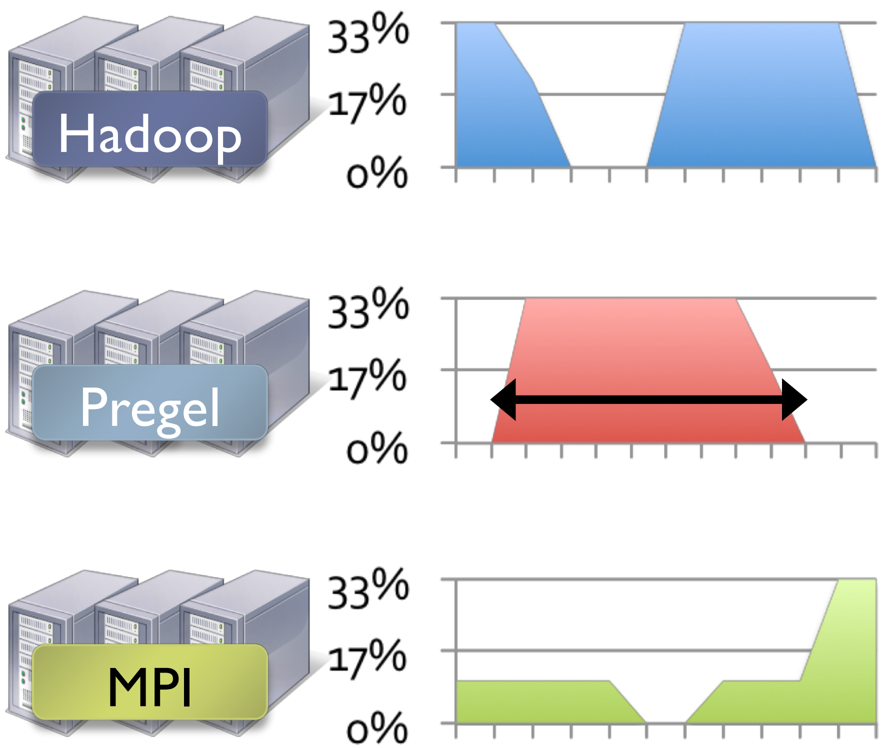
Slides by Matei Zaharia

Problem

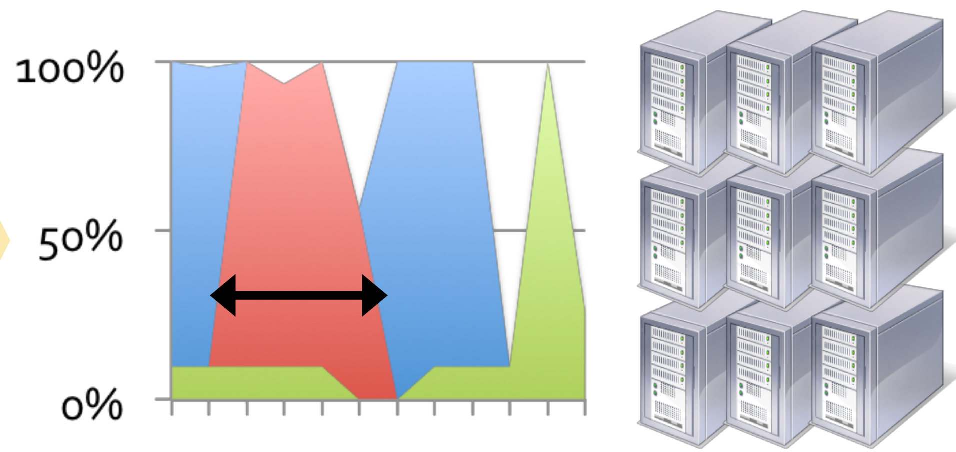
- Rapid innovation in cluster computing frameworks
- No single framework optimal for all applications
- Want to run multiple frameworks in a single cluster
  - …to maximize utilization
  - …to share data between frameworks

Where We Want to Go

Today: static partitioning



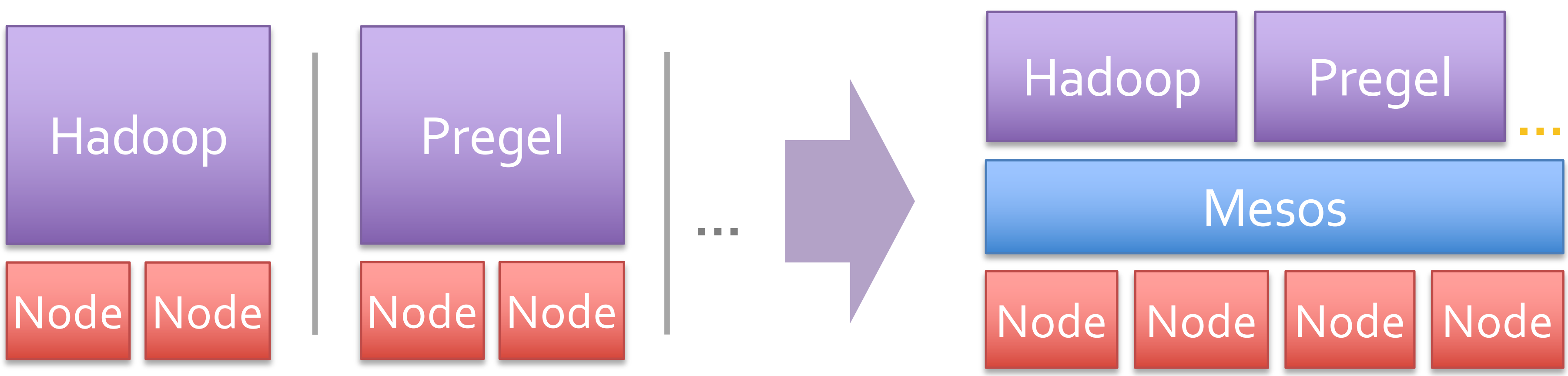
Mesos: dynamic sharing



Shared cluster

Solution

- Mesos is a common resource sharing layer over which diverse frameworks can run



Other Benefits of Mesos

- Run multiple instances of the same framework
  - Isolate production and experimental jobs
  - Run multiple versions of the framework concurrently
- Build specialized frameworks targeting particular problem domains
  - Better performance than general-purpose abstractions

Mesos Goals

- High utilization of resources
- Support diverse frameworks (current & future)
- Scalability to 10,000's of nodes
- Reliability in face of failures

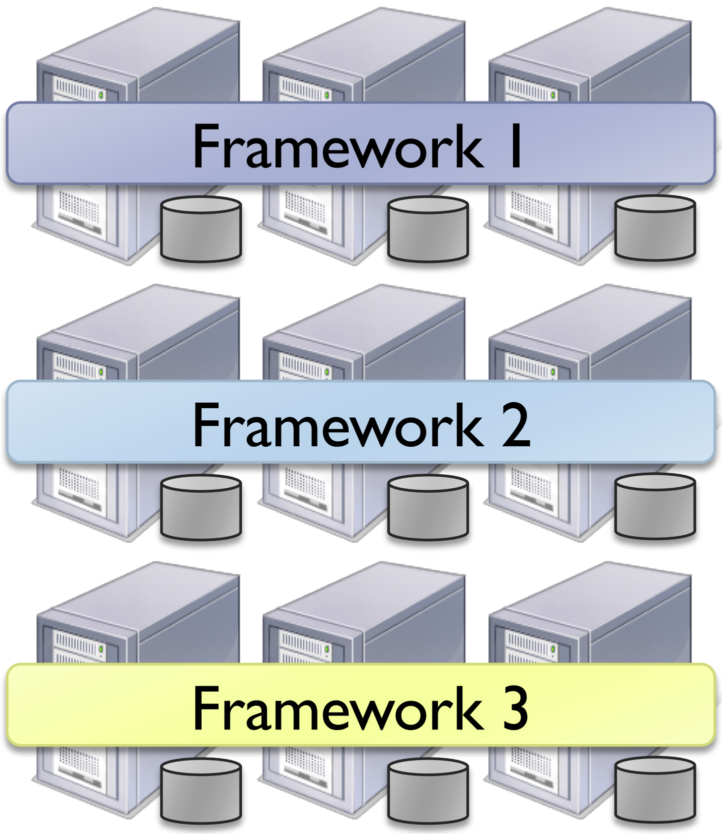
Resulting design: Small microkernel-like core that pushes scheduling logic to frameworks

Design Elements

- Fine-grained sharing:
  - Allocation at the level of tasks within a job
  - Improves utilization, latency, and data locality
- Resource offers:
  - Simple, scalable application-controlled scheduling mechanism

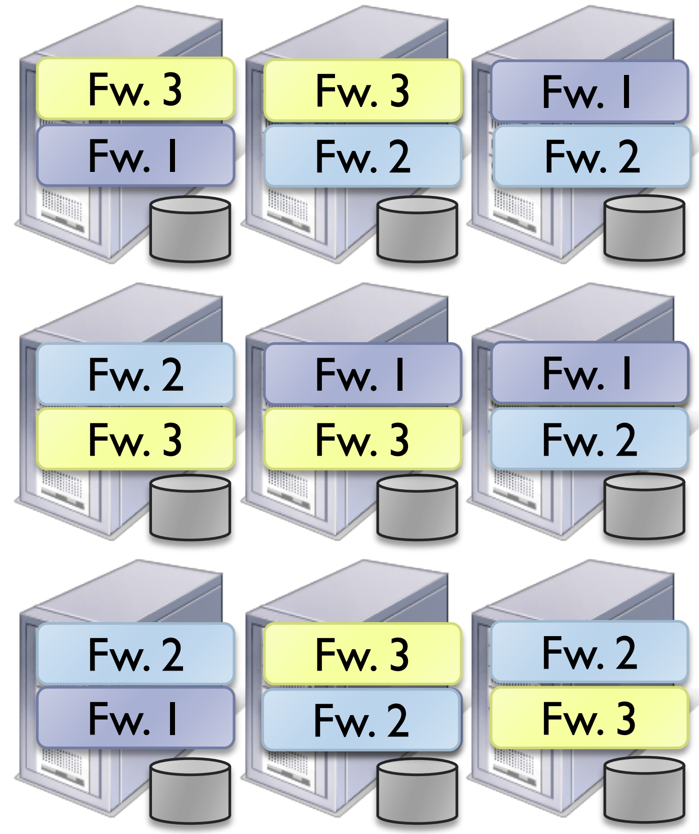
Element 1: Fine-Grained Sharing

Coarse-Grained Sharing (HPC):



Storage System (e.g. HDFS)

Fine-Grained Sharing (Mesos):



Storage System (e.g. HDFS)

+ Improved utilization, responsiveness, data locality

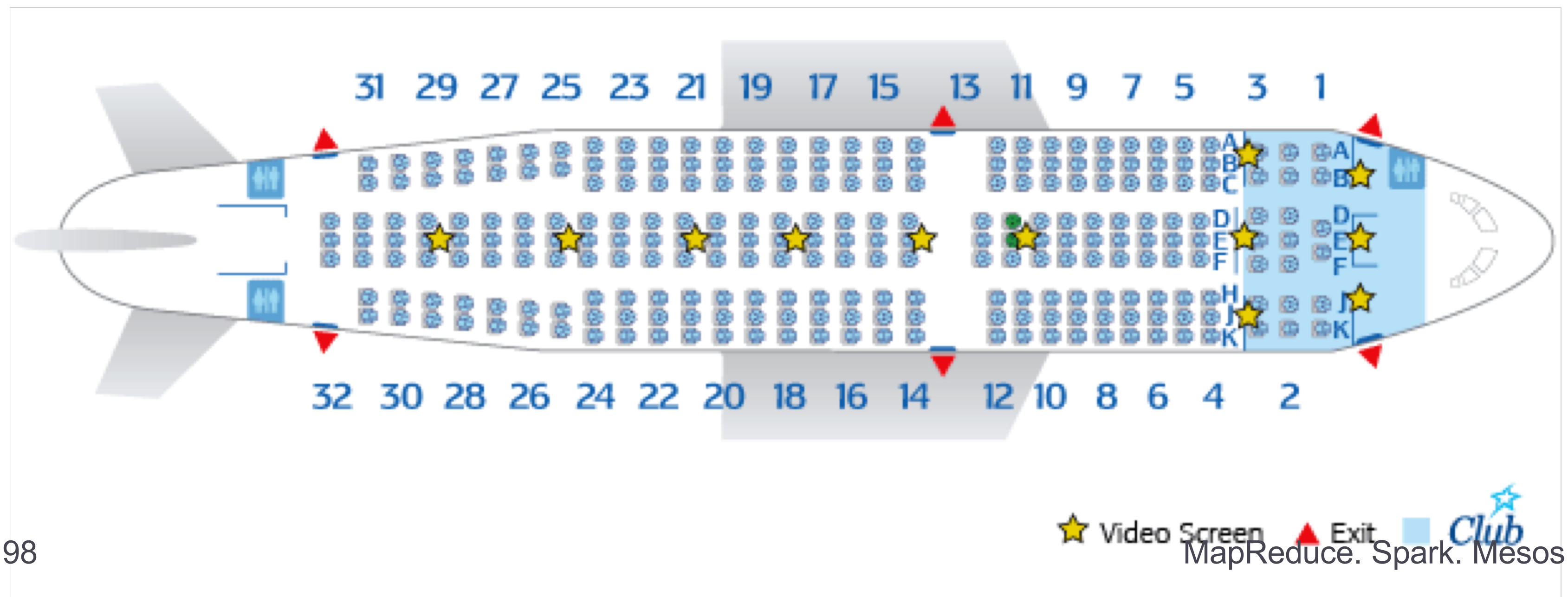
Element 2: Resource Offers

- Option: Global scheduler
  - Frameworks express needs in a specification language, global scheduler matches them to resources
    - + Can make optimal decisions
- – Complex: language must support all framework needs
    - – Difficult to scale and to make robust
    - – Future frameworks may have unanticipated needs

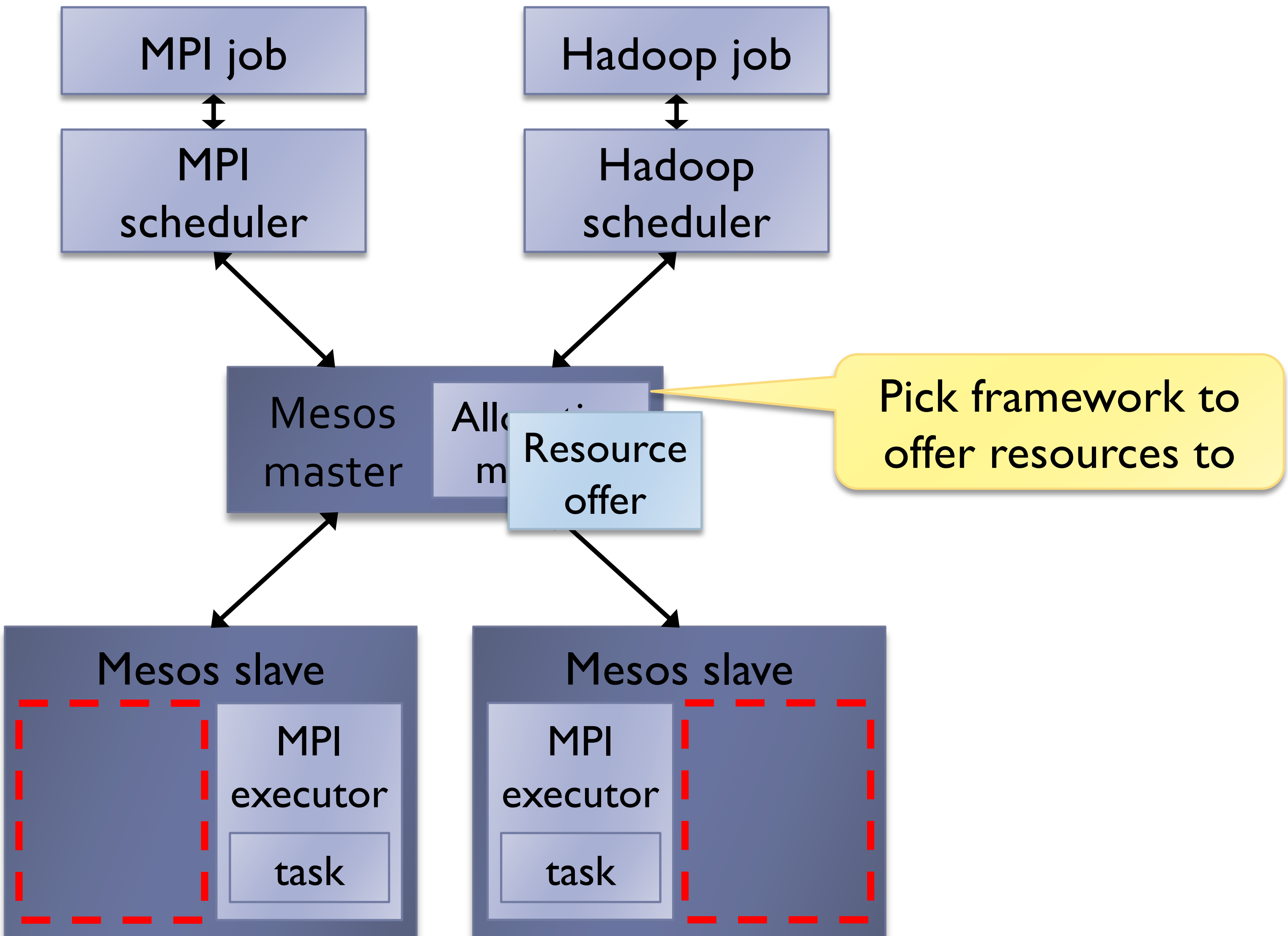
Element 2: Resource Offers

- Mesos: Resource offers
  - Offer available resources to frameworks, let them pick which resources to use and which tasks to launch

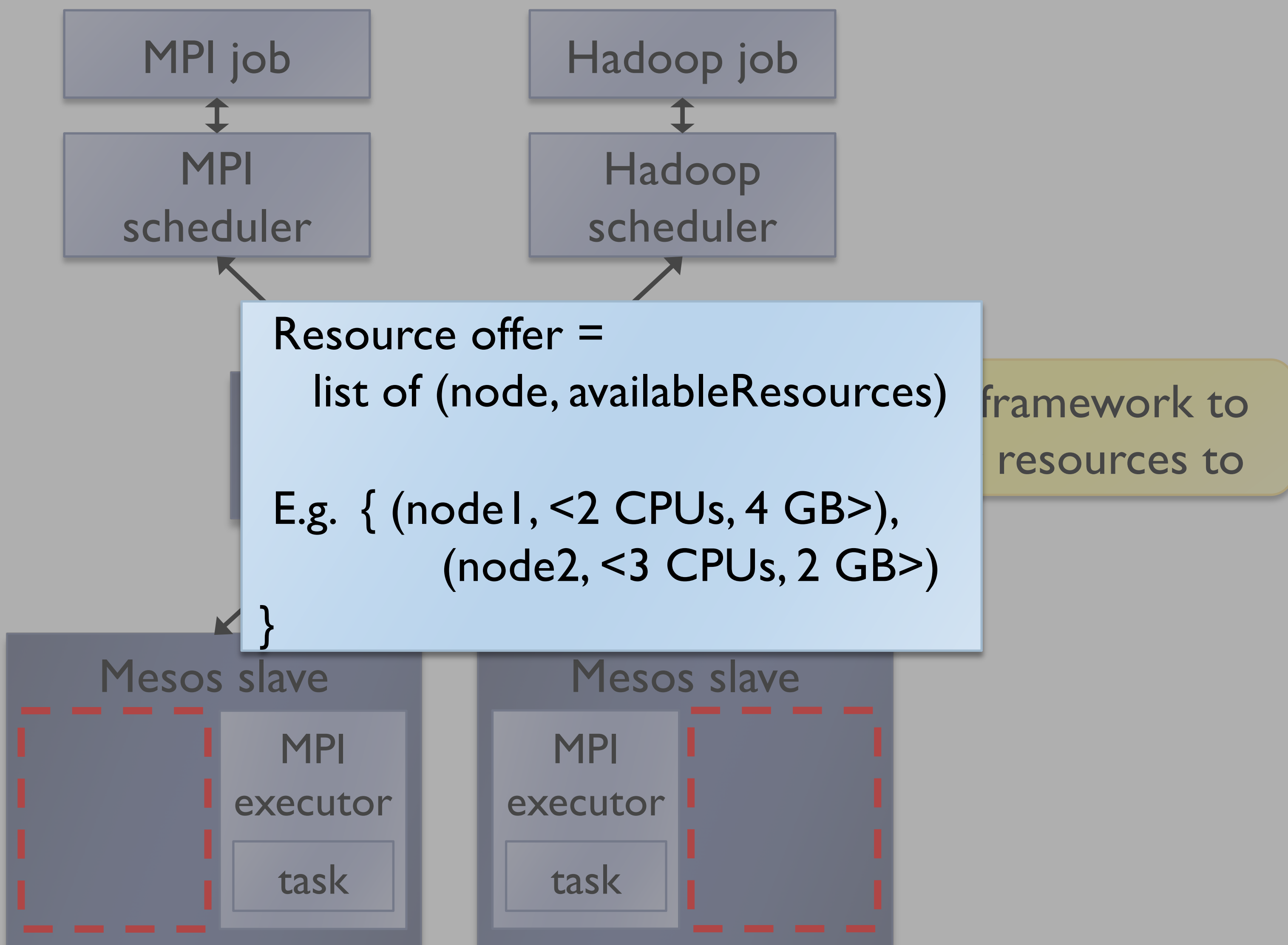
  - Keeps Mesos simple, lets it support future frameworks
  - Decentralized decisions might not be optimal



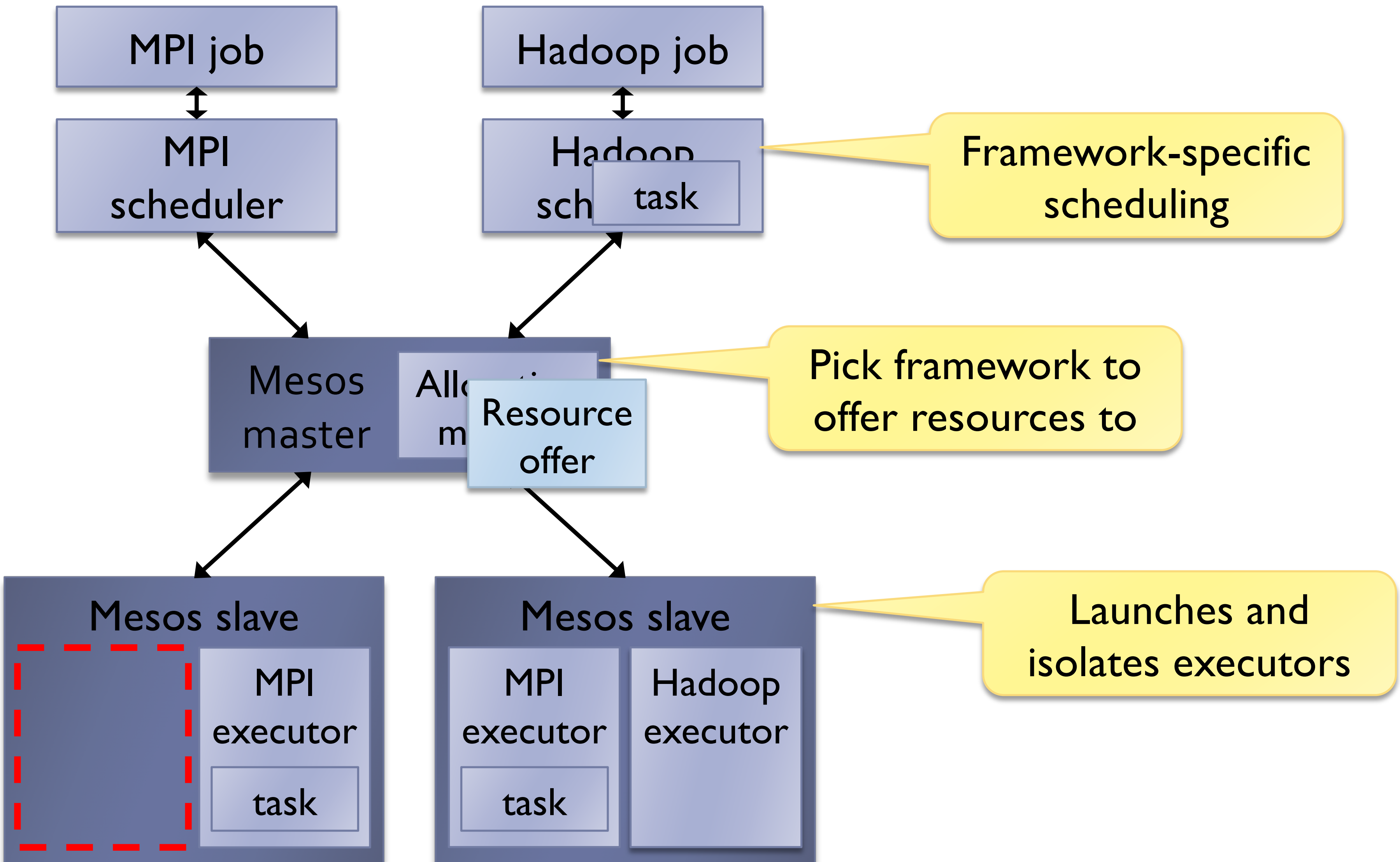
Mesos Architecture



Mesos Architecture



Mesos Architecture



Optimization: Filters

- Let frameworks short-circuit rejection by providing a predicate on resources to be offered
  - E.g. "nodes from list L" or "nodes with > 8 GB RAM"
  - Could generalize to other hints as well
- Ability to reject still ensures correctness when needs cannot be expressed using filters

Implementation Stats

- 20,000 lines of C++
- Master failover using ZooKeeper
- Frameworks ported: Hadoop, MPI, Torque
- New specialized framework: Spark, for iterative jobs (up to 20 $\times$ faster than Hadoop)

- Open source in Apache Incubator

Users

- Twitter uses Mesos on > 100 nodes to run ~12 production services (mostly stream processing)
- Berkeley machine learning researchers are running several algorithms at scale on Spark
- Conviva is using Spark for data analytics
- UCSF medical researchers are using Mesos to run Hadoop and eventually non-Hadoop apps

Framework Isolation

- Mesos uses OS isolation mechanisms, such as Linux containers and Solaris projects
- Containers currently support CPU, memory, IO and network bandwidth isolation
- Not perfect, but much better than no isolation

Analysis

- Resource offers work well when:
  - Frameworks can scale up and down elastically
  - Task durations are homogeneous
  - Frameworks have many preferred nodes
- These conditions hold in current data analytics frameworks (MapReduce, Dryad, …)
  - Work divided into short tasks to facilitate load balancing and fault recovery
  - Data replicated across multiple nodes

Revocation

- Mesos allocation modules can revoke (kill) tasks to meet organizational SLOs
- Framework given a grace period to clean up
- "Guaranteed share" API lets frameworks avoid revocation by staying below a certain share

Mesos API

Scheduler Callbacks	Scheduler Actions
resourceOffer(offerId, offers) offerRescinded(offerId) statusUpdate(taskId, status) slaveLost(slaveId)	replyToOffer(offerId, tasks) setNeedsOffers(bool) setFilters(filters) getGuaranteedShare() killTask(taskId)

Executor Callbacks	Executor Actions
launchTask(taskDescriptor) killTask(taskId)	sendStatus(taskId, status)

Results

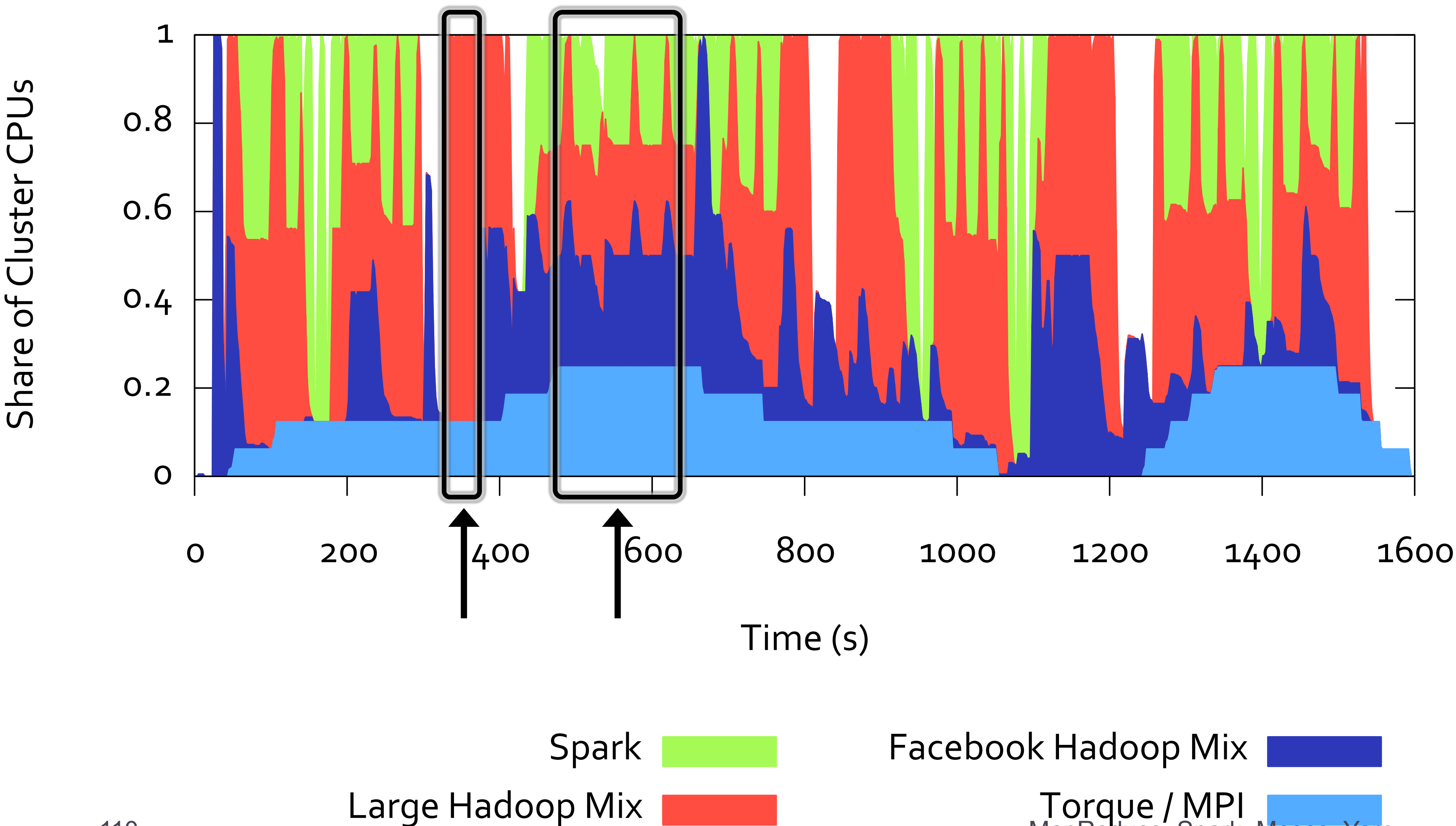
» Utilization and performance vs static partitioning

» Framework placement goals: data locality

» Scalability

» Fault recovery

Dynamic Resource Sharing



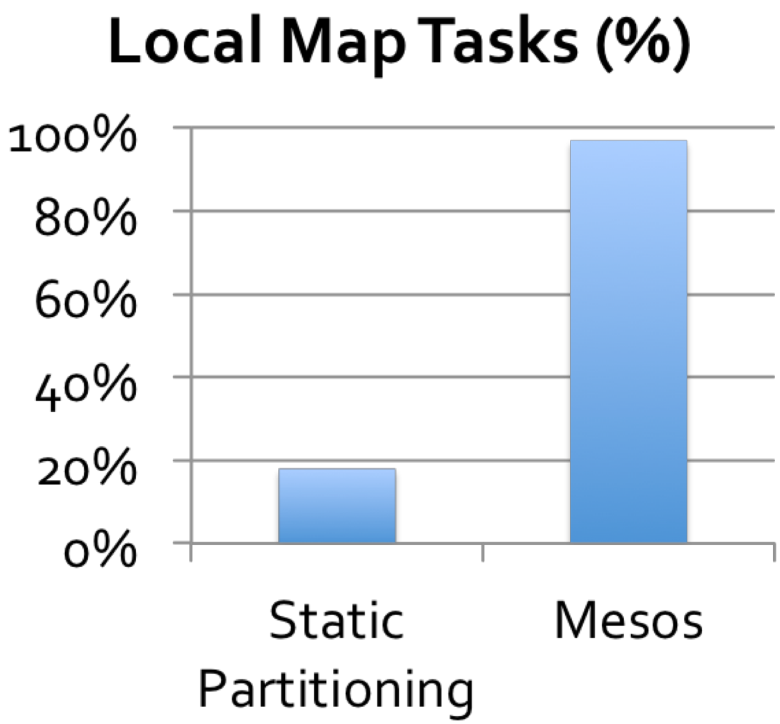
Mesos vs Static Partitioning

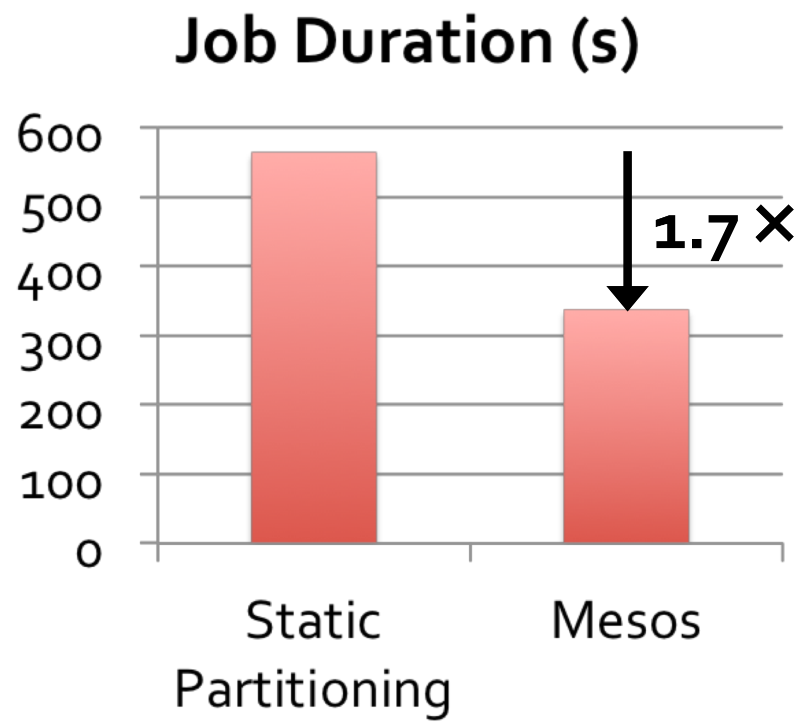
- Compared performance with statically partitioned cluster where each framework gets 25% of nodes

Framework	Speedup on Mesos
Facebook Hadoop Mix	1.14×
Large Hadoop Mix	2.10×
Spark	1.26×
Torque / MPI	0.96×

Data Locality with Resource Offers

- Ran 16 instances of Hadoop on a shared HDFS cluster
- Used delay scheduling [EuroSys '10] in Hadoop to get locality (wait a short time to acquire data-local nodes)



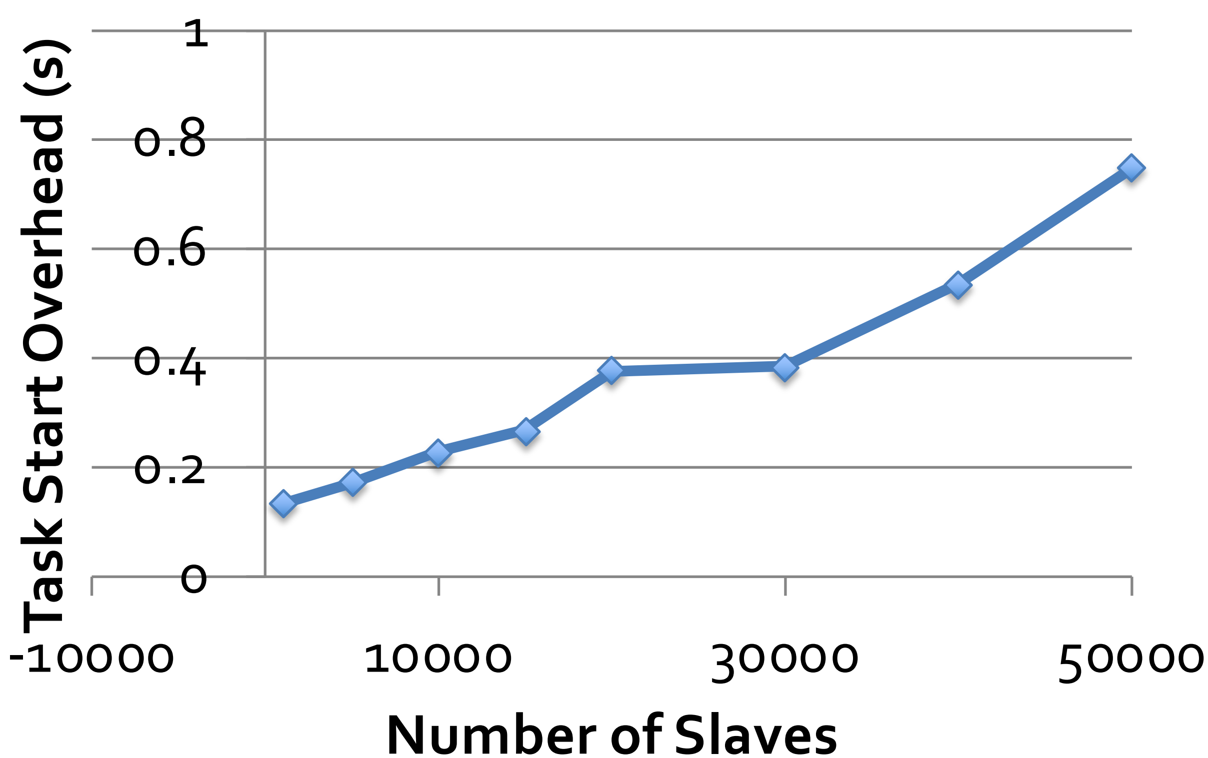


Scalability

- Mesos only performs inter-framework scheduling (e.g. fair sharing), which is easier than intra-framework scheduling

Result:

Scaled to 50,000 emulated slaves, 200 frameworks, 100K tasks (30s len)



Fault Tolerance

- Mesos master has only soft state: list of currently running frameworks and tasks
- Rebuild when frameworks and slaves re-register with new master after a failure
- Result: fault detection and recovery in ~10 sec

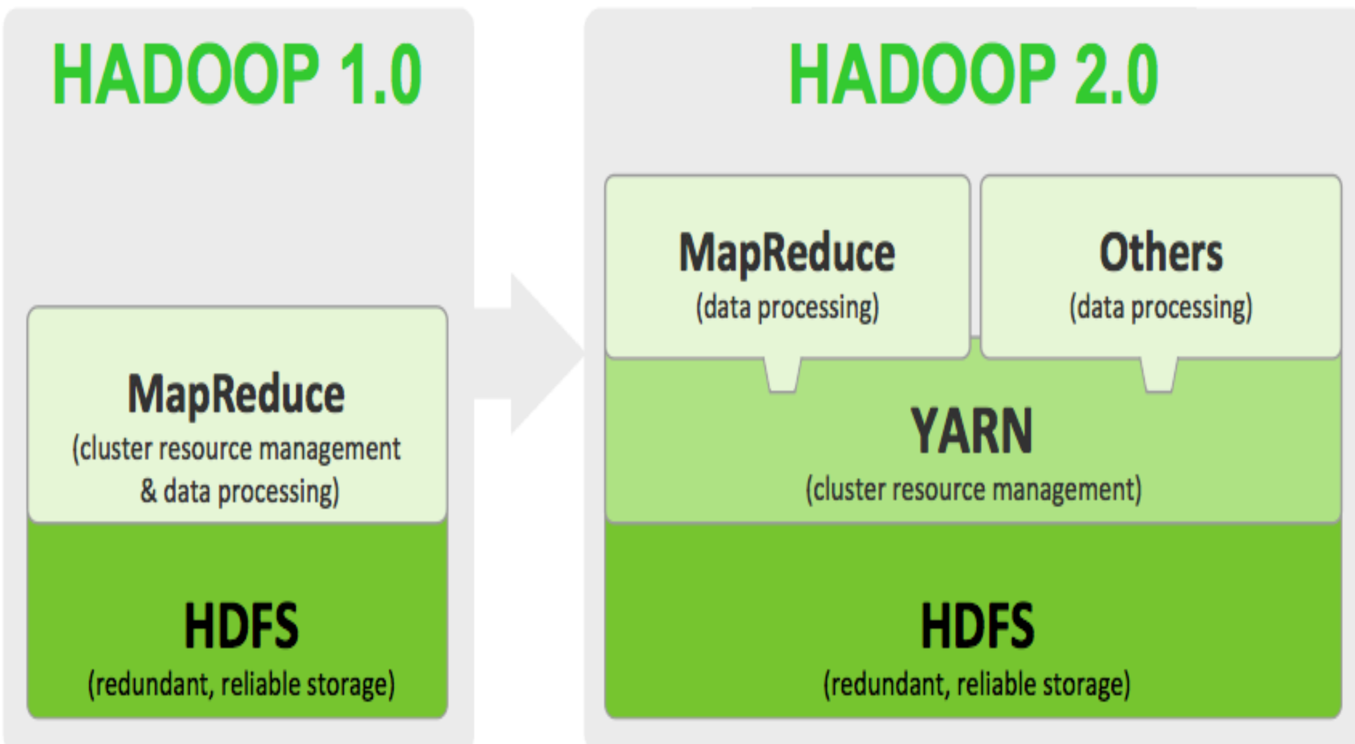
Conclusion

- Mesos shares clusters efficiently among diverse frameworks thanks to two design elements:
  - Fine-grained sharing at the level of tasks
  - Resource offers, a scalable mechanism for application-controlled scheduling
- Enables co-existence of current frameworks and development of new specialized ones
- In use at Twitter, UC Berkeley, Conviva and UCSF

4: Yarn

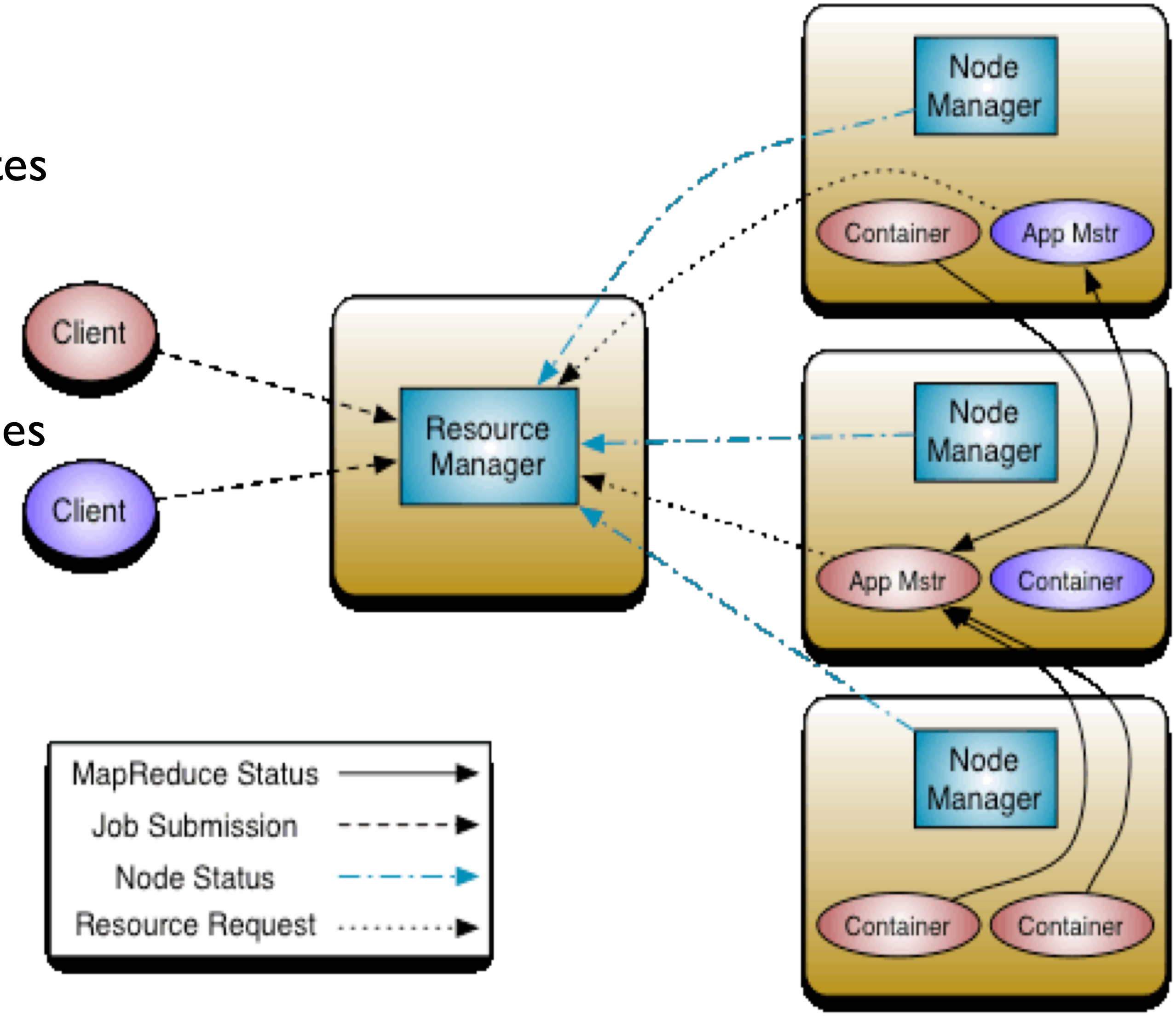
YARN - Yet Another Resource Negotiator

- Next version of MapReduce or MapReduce 2.0 (MRv2)
- In 2010 group at Yahoo! Began to design the next generation of MR



YARN architecture

- **Resource Manager**
  - Central Agent – Manages and allocates cluster resources
- **Node Manager**
  - Per-node agent – Manages and enforces node resource allocations
- **Application Master**
  - Per Application
  - Manages application life cycle and task scheduling



YARN – Resource Manager Failure

- After a crash a new Resource Manager instance needs to brought up (by an administrator)
- It recovers from saved state
- State consists of
  - node managers in the systems
  - running applications
- State to manage is much more manageable than that of Job Tracker.
  - Tasks are not part of Resource Managers state.
  - They are handled by the application master.