Hadoop and MapReduce

Adapted from:

T. White, Hadoop: The Definitive Guide, O'Reilly 2012-2018

Limitations of HPC for Big Data

- An increasing number of HPC application requires the processing of large input data
- Processing data with HPC tools is hard: For example, to process a large file full of DNA-sequencing reads in parallel, we must manually split it up into smaller files and submit a job for each of those files to the cluster scheduler.
- Tools written for HPC environments often fail to decouple the in-memory data models from the lower-level storage models
- Many tools only know how to read data from a POSIX filesystem in a single stream, making it difficult to make tools naturally parallelize
- Storage backends, like databases, make it difficult to scale compute
- Recent systems in the Hadoop ecosystem provide abstractions that allow users to treat a cluster of computers more like a single computer—to automatically split up files and distribute storage over many machines, divide work into smaller tasks and execute them in a distributed manner, and recover from failures.
- The Hadoop ecosystem can automate a lot of the hassle of working with large data sets, and is far cheaper than HPC.

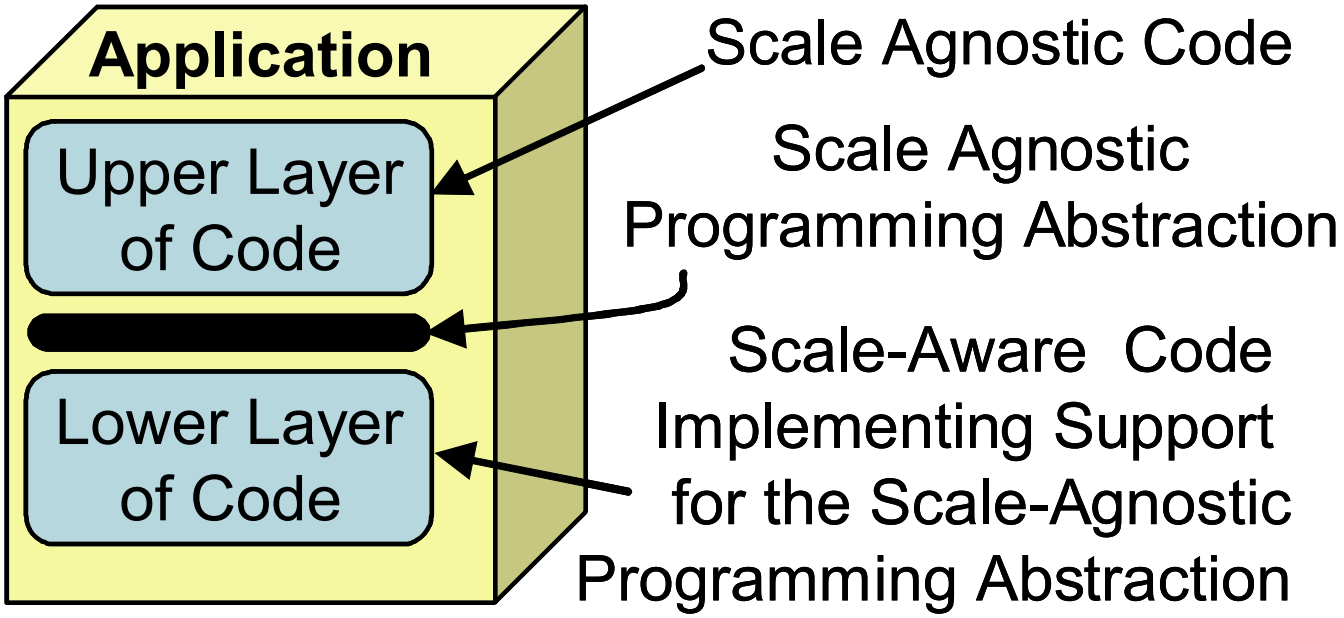
Uri Laserson; Josh Wills; Sean Owen; Sandy Ryza, Advanced Analytics with Spark, O'Reilly, 2017.

- **Distributed Applications:** Distributed applications are those that (a) need multiple resources, or (b) would benefit from the use of multiple resources; examples of how they could benefit include, increased peak time to solution or reliability.
- Developing distributed applications is hard!
  - Heterogeneous middleware, point solutions, usability, …
  - Occupied by plumbing; insufficient reasoning about when/how to distribute
- Distributed computing at scale needs performance, flexibility, and extensibility

- **Abstraction:** An abstraction is any process, mechanism or infrastructure to support a commonly occurring usage (e.g., computation, communication, and/or composition). Jha et. al., Distributed Programming Abstractions

- Abstractions help to mitigate the complexities of distributed systems.

- What are scalable distributed programming abstractions?


Application
Upper Layer of Code
Lower Layer of Code
Scale Agnostic Code
Scale Agnostic Programming Abstraction
Scale-Aware Code Implementing Support for the Scale-Agnostic Programming Abstraction

Hadoo and MapReduce in a Nutshell

- **MapReduce** is a programming abstraction designed for exploring large data sets:
  - **First phase:** map a user supplied function onto data distributed across multiple computers and generate a set of intermediate key/value pairs
  - **Second phase:** reduces the returned values from these map instances into a single result by merging all intermediate values associated with the same intermediate key
- **Hadoop** is a open source software infrastructure for support the **MapReduce** programming model.

- **Hadoop Distributed File System (HDFS):** distributed filesystem
- **YARN:** Yet Another Resource Negotiator
- **MapReduce:** Application Framework

Google Filesystem

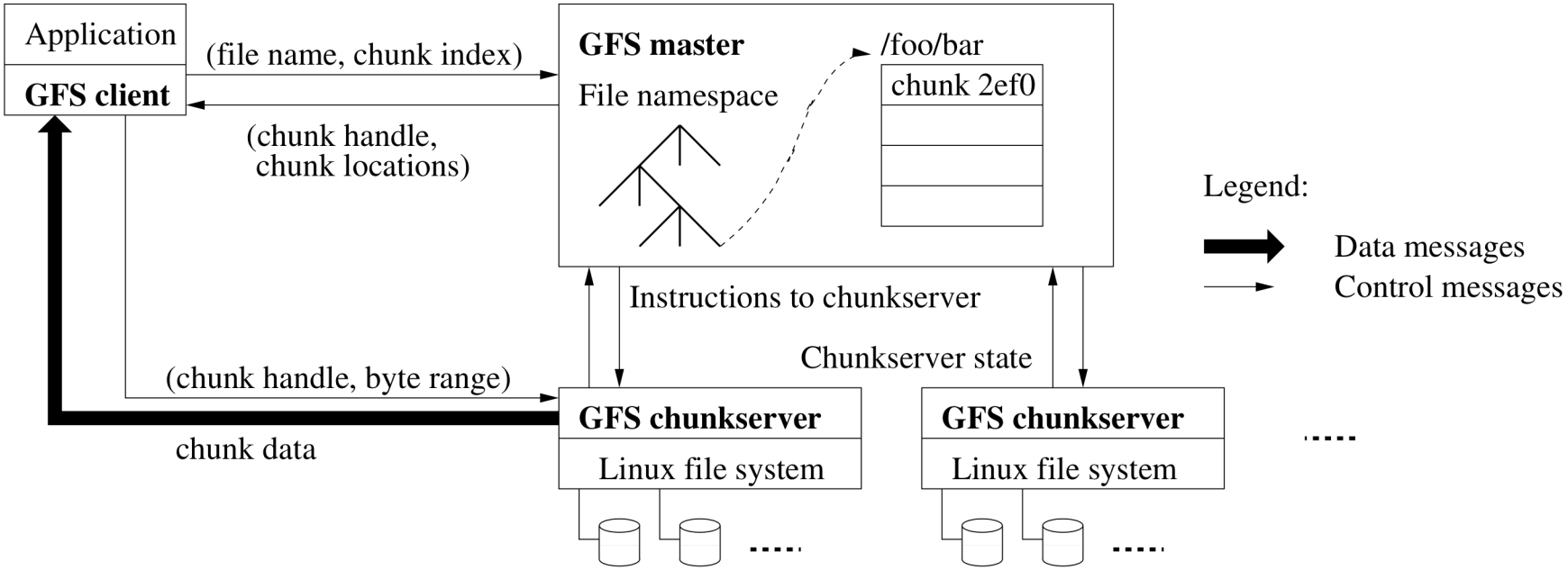
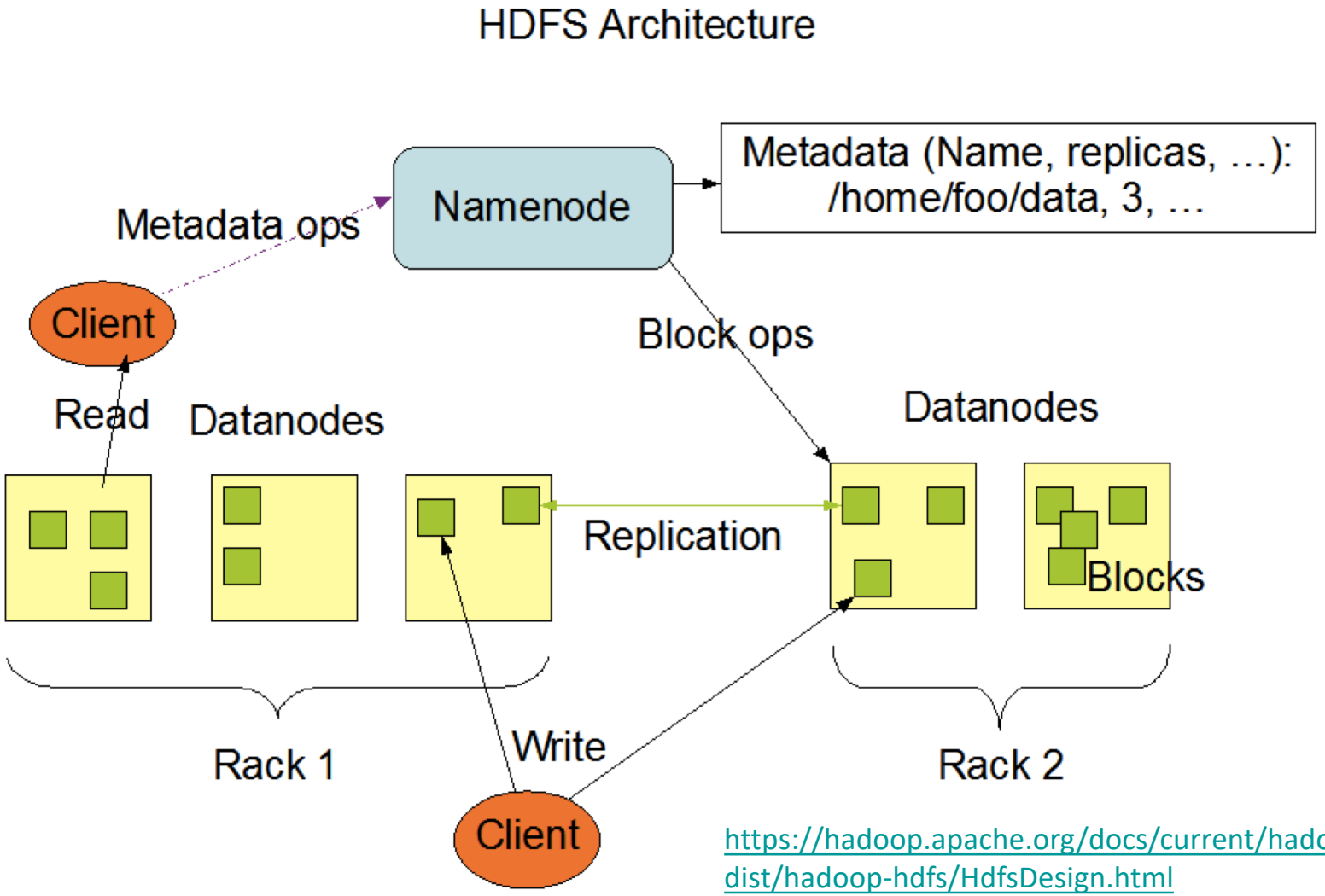


Figure 1: GFS Architecture

Hadoop Filesystem



https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html

- Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In OSDI'04: Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation, pages 137-150, Berkeley, CA, USA, 2004. USENIX Association.
- Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. SIGOPS Oper. Syst. Rev., 37(5):29, 2003.
- Konstantin Shvachko, Hairong Kuang, Sanjay Radia, Robert Chansler, The Hadoop Filesystem, MSST'2010, https://dl.acm.org/citation.cfm?id=1914427
- Hadoop File System. http://hadoop.apache.org/common/docs/current/hdfs_design.html

Unix Philosophy:

- Write programs that do one thing and do it well
- Write programs to work together
- Write programs that handle text streams, because

that is a universal interface

The Unix Pipe | represents a universal abstraction for connecting programs

Unix: MapReduce

Read a HTTP access log file an determine the most frequent HTTP return codes:

```
ubuntu@ip-10-186-164-81:~$ head /data/NASA_access_log_Jul95
199.72.81.55 - - [01/Jul/1995:00:00:01 -0400] "GET /history/apollo/ HTTP/1.0" 200 6245
unicomp6.unicomp.net - - [01/Jul/1995:00:00:06 -0400] "GET /shuttle/countdown/ HTTP/1.0" 200 3985
199.120.110.21 - - [01/Jul/1995:00:00:09 -0400] "GET /shuttle/missions/sts-73/mission-sts-73.html HTTP/1.0" 200 4085
burger.letters.com - - [01/Jul/1995:00:00:11 -0400] "GET /shuttle/countdown/liftoff.html HTTP/1.0" 304 0
```

```
cat /data/NASA_access_log_Jul95 |
```

Unix: MapReduce

Read a HTTP access log file an determine the most frequent HTTP return codes:

```
ubuntu@ip-10-186-164-81:~$ head /data/NASA_access_log_Jul95
199.72.81.55 - - [01/Jul/1995:00:00:01 -0400] "GET /history/apollo/ HTTP/1.0" 200 6245
unicomp6.unicomp.net - - [01/Jul/1995:00:00:06 -0400] "GET /shuttle/countdown/ HTTP/1.0" 200 3985
199.120.110.21 - - [01/Jul/1995:00:00:09 -0400] "GET /shuttle/missions/sts-73/mission-sts-73.html HTTP/1.0" 200 4085
burger.letters.com - - [01/Jul/1995:00:00:11 -0400] "GET /shuttle/countdown/liftoff.html HTTP/1.0" 304 0
```

```
cat /data/NASA_access_log_Jul95 |
      awk  '{print $(NF-1)}'|
      sort |
      uniq -c
```

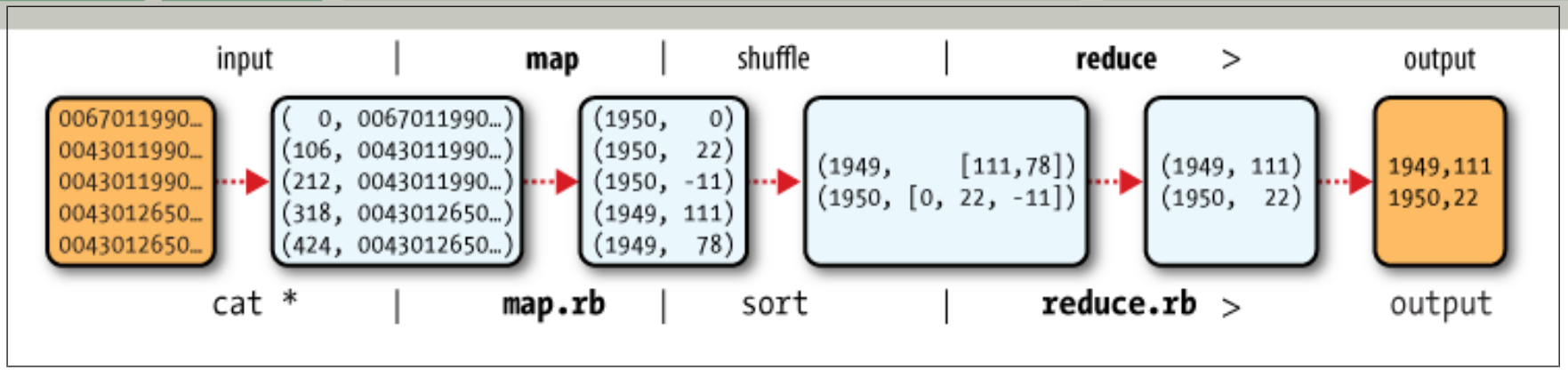
- **MapReduce** is a key/value-based API originally designed for processing text streams.
- Different file formats are supported via the **`InputFormat`** API. The `InputFormat` defines how input formats are partitioned and distributed to Mappers.
- The mapper tasks input is an **`InputSplit.`**

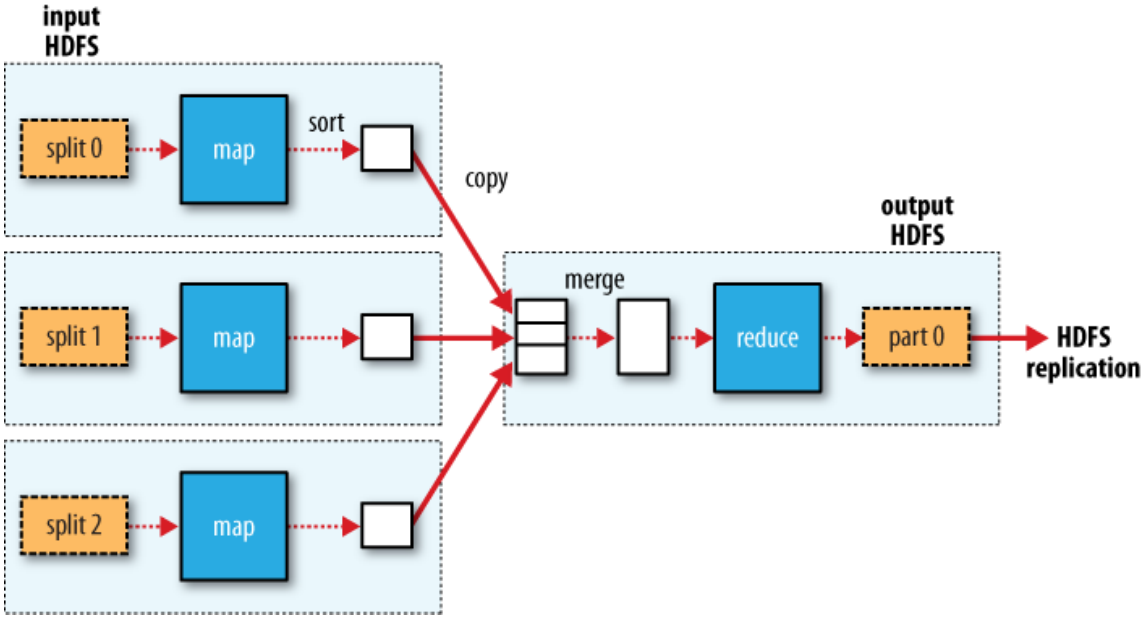
```
ubuntu@ip-10-186-164-81:~$ head /data/NASA_access_log_Jul95
199.72.81.55 - - [01/Jul/1995:00:00:01 -0400] "GET /history/apollo/ HTTP/1.0" 200 6245
unicomp6.unicomp.net - - [01/Jul/1995:00:00:06 -0400] "GET /shuttle/countdown/ HTTP/1.0" 200 3985
199.120.110.21 - - [01/Jul/1995:00:00:09 -0400] "GET /shuttle/missions/sts-73/mission-sts-73.html HTTP/1.0" 200 4085
burger.letters.com - - [01/Jul/1995:00:00:11 -0400] "GET /shuttle/countdown/liftoff.html HTTP/1.0" 304 0
199.120.110.21 - - [01/Jul/1995:00:00:11 -0400] "GET /shuttle/missions/sts-73/sts-73-patch-small.gif HTTP/1.0" 200 4179
burger.letters.com - - [01/Jul/1995:00:00:12 -0400] "GET /images/NASA-logosmall.gif HTTP/1.0" 304 0
burger.letters.com - - [01/Jul/1995:00:00:12 -0400] "GET /shuttle/countdown/video/livevideo.gif HTTP/1.0" 200 0
205.212.115.106 - - [01/Jul/1995:00:00:12 -0400] "GET /shuttle/countdown/countdown.html HTTP/1.0" 200 3985
d104.aa.net - - [01/Jul/1995:00:00:13 -0400] "GET /shuttle/countdown/ HTTP/1.0" 200 3985
129.94.144.152 - - [01/Jul/1995:00:00:13 -0400] "GET / HTTP/1.0" 200 7074
```

Input Split 1

Input Split 2

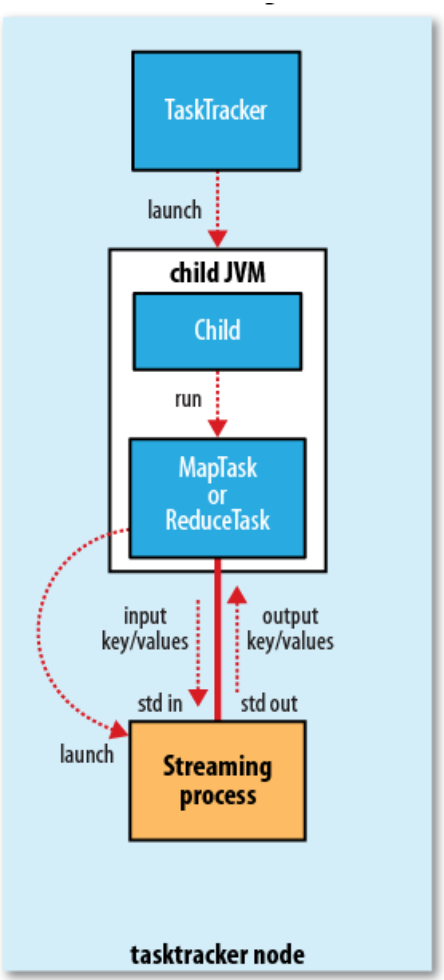
MapReduce in Detail





Hadoop Streaming API

- Programming language agnostic way to express MapReduce application
- Use stdin/stdout of programming framework to communicate with MapReduce framework
- Python example for MapReduce Log File Analysis



Source: T. White, Hadoop: The Definitive Guide, 2012

Hadoop Streaming API


```
def emit(key, value):
    """
    Emits a key->value pair.  Key and value should be strings.
    """
    try:
        print "\t".join( (key, value) )
    except:
        pass

def run_map():
    """Calls map() for each input value."""
    for line in sys.stdin:
        line = line.rstrip()
        map(line)

def run_reduce():
    """Gathers reduce() data in memory, and calls reduce()."""
    prev_key = None
    values = []
    for line in sys.stdin:
        line = line.rstrip()
        key, value = re.split("\t", line, 1)
        if prev_key == key:
            values.append(value)
        else:
            if prev_key is not None:
                reduce(prev_key, values)
            prev_key = key
```

Output of map function is written to **`stdout`**.

`InputSplits` are passed as **`stdin`** into application. Utility function for reading lines and passing it to map function.

Hadoop Streaming API


```
def map(line):
    try:
        words = line.split()
        http_response_code = words[-2]
        emit(http_response_code, str(1))
    except:
        pass

def reduce(key, values):
    emit(key, str(sum(__builtin__.map(int,values))))
```

How to count the HTTP Response codes with Map Reduce?

Hadoop Streaming API

- **Test local**:

```
cat /data/NASA_access_log_Jul95|
        python map_reduce.py map |
        sort |
        python map_reduce.py reduce
```

- **Test cluster:**

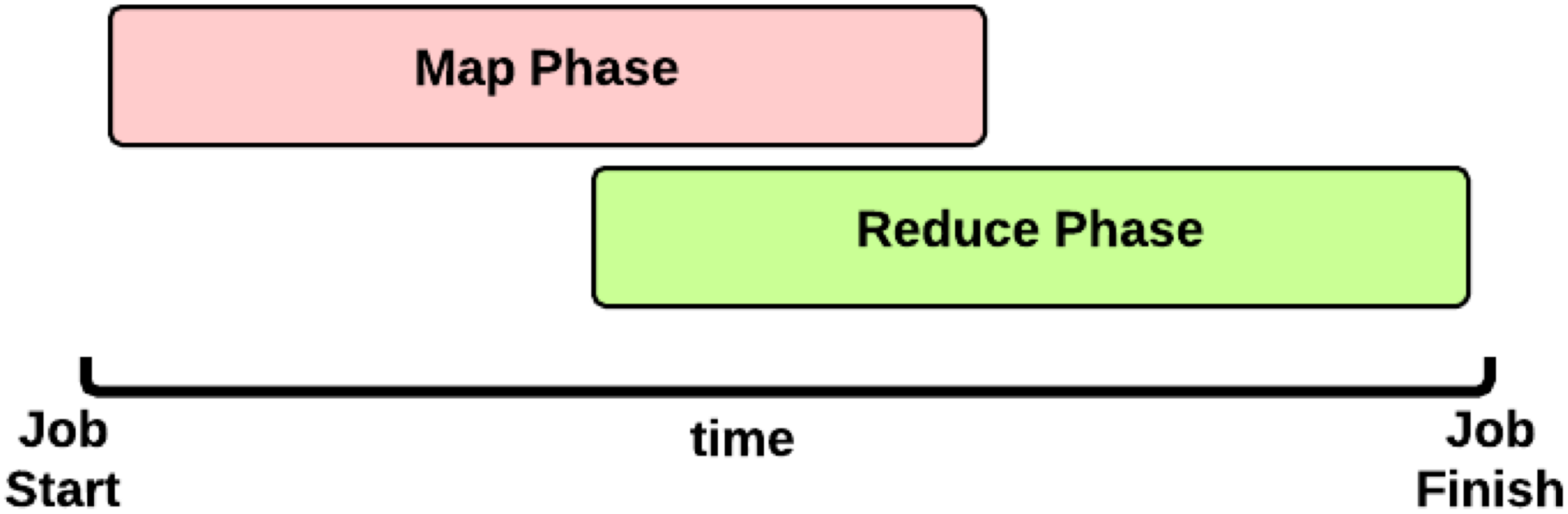
```
hadoop fs -put map_reduce.py
hadoop jar hadoop-streaming.jar
        -files map_reduce.py
        -input input-nasa/
        -output output-nasa/
        -mapper 'map_reduce.py map'
        -reducer 'map_reduce.py reduce'
```

Hadoop MapReduce Components (1)

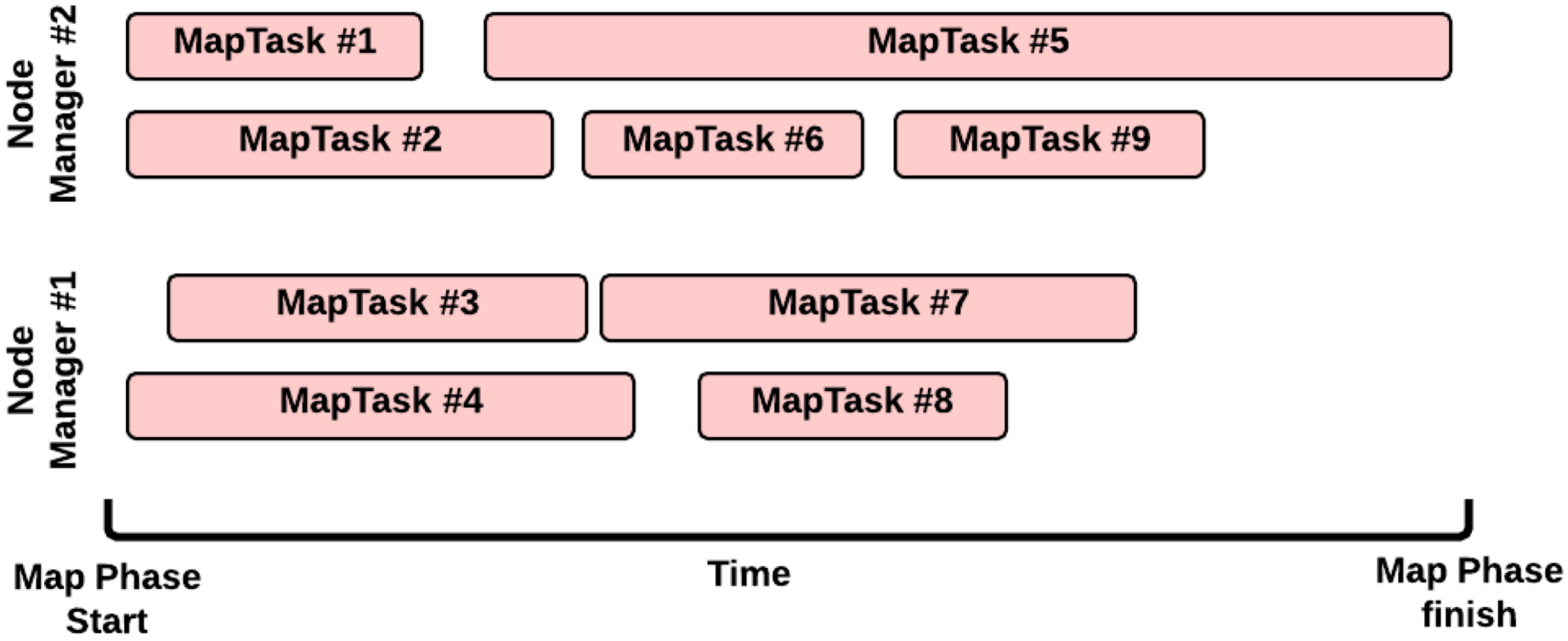
- **Record Reader:** *T*he record reader translates an input split generated by input format into records.
- **Map:** In the mapper, user-provided code is executed on each key/value pair from the record reader to produce zero or more new key/value pairs, called the intermediate pairs.
- **Combiner:** The combiner, an optional localized reducer, can group data in the map phase. It takes the intermediate keys from the mapper and applies a user-provided method to aggregate values in the small scope of that one mapper.
- **Partitioner:** The partitioner takes the intermediate key/value pairs from the mapper (or combiner if it is being used) and splits them up into shards, one shard per reducer.

- **Shuffle and Sort:** The reduce task starts with the *shuffle and sort* step. This step takes the output files written by all of the partitioners and downloads them to the local machine in which the reducer is running.
- **Reduce:** The reducer takes the grouped data as input and runs a function once per key grouping. The function is passed the key and an iterator over all of the values associated with that key.
- **Output Format:** The output format translates the final key/value pair from the function and writes it out to a file by a record writer.

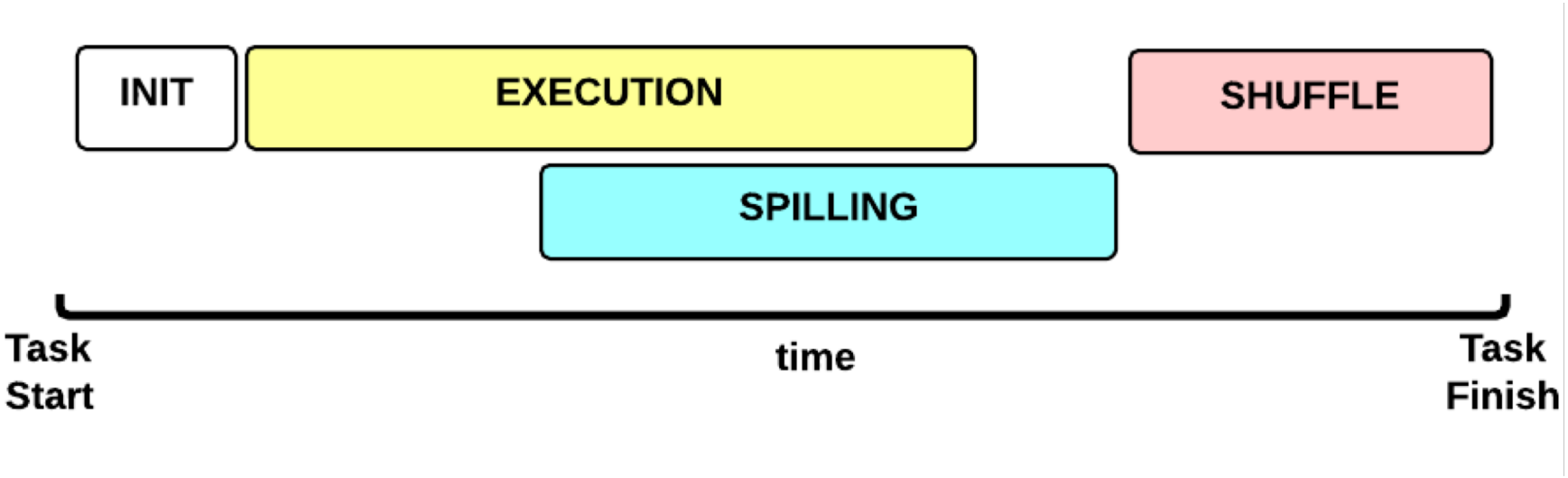
Anatomy of a MapReduce Job



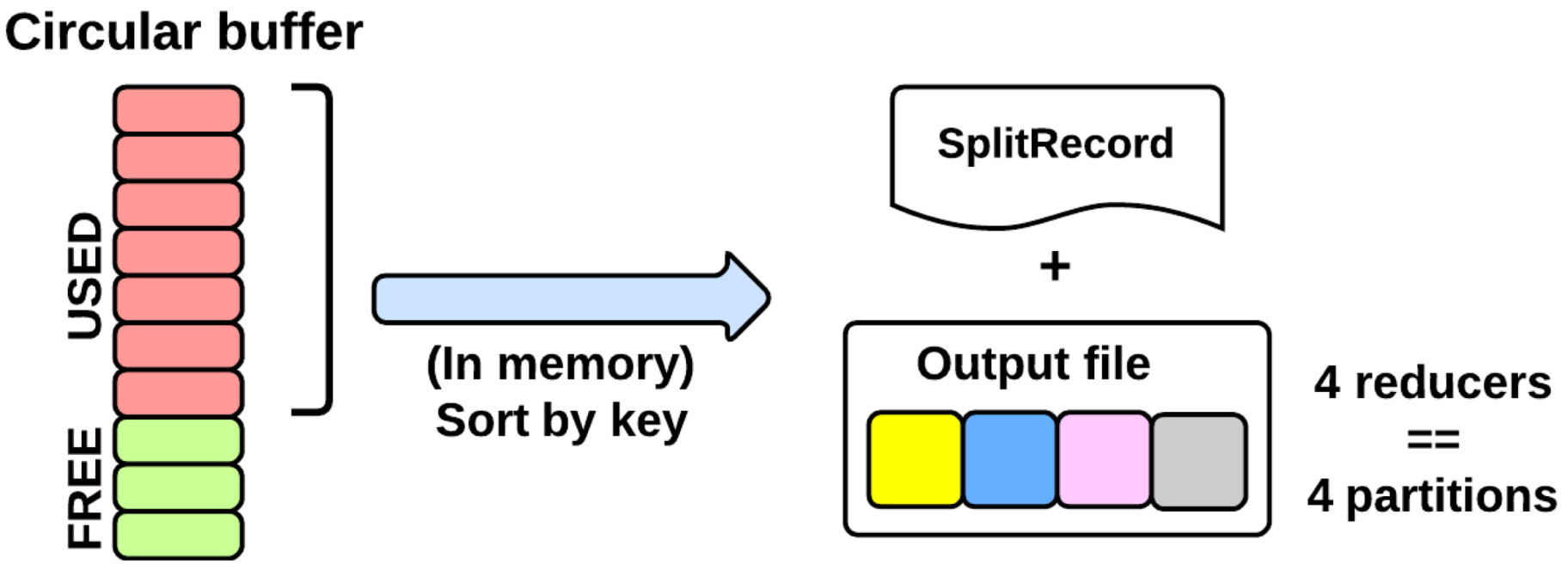
Anatomy of a MapReduce Job



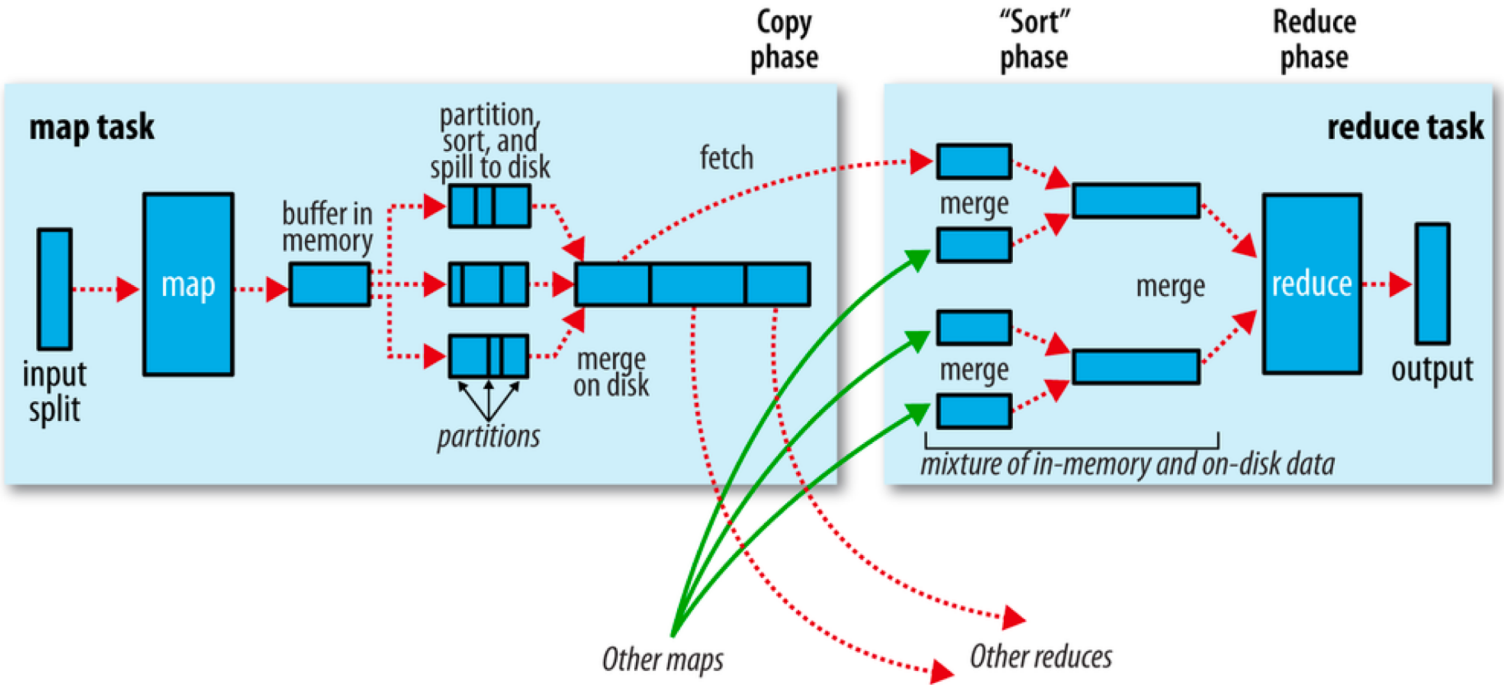
Anatomy of a MapReduce Job



Anatomy of a MapReduce Job

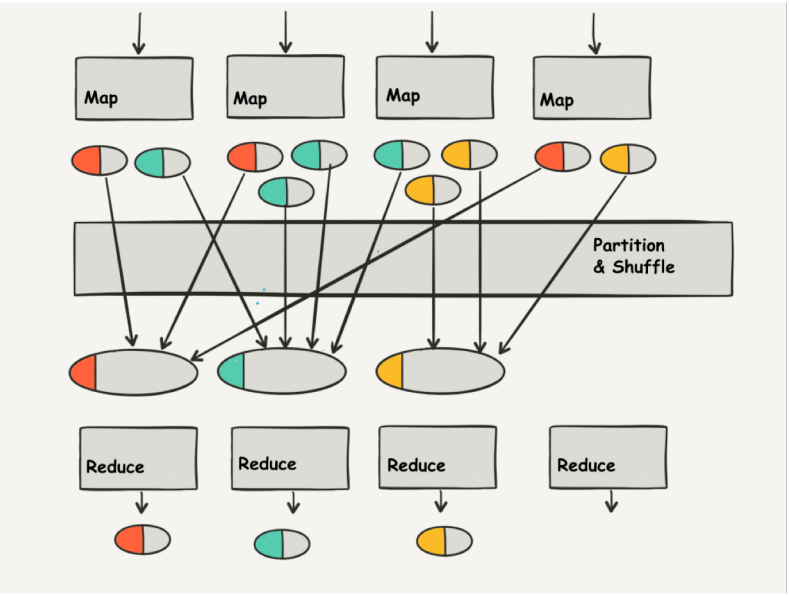


Shuffling



T. White, Hadoop: The Definitive Guide, 4th Edition, O'Reilly, 2017.

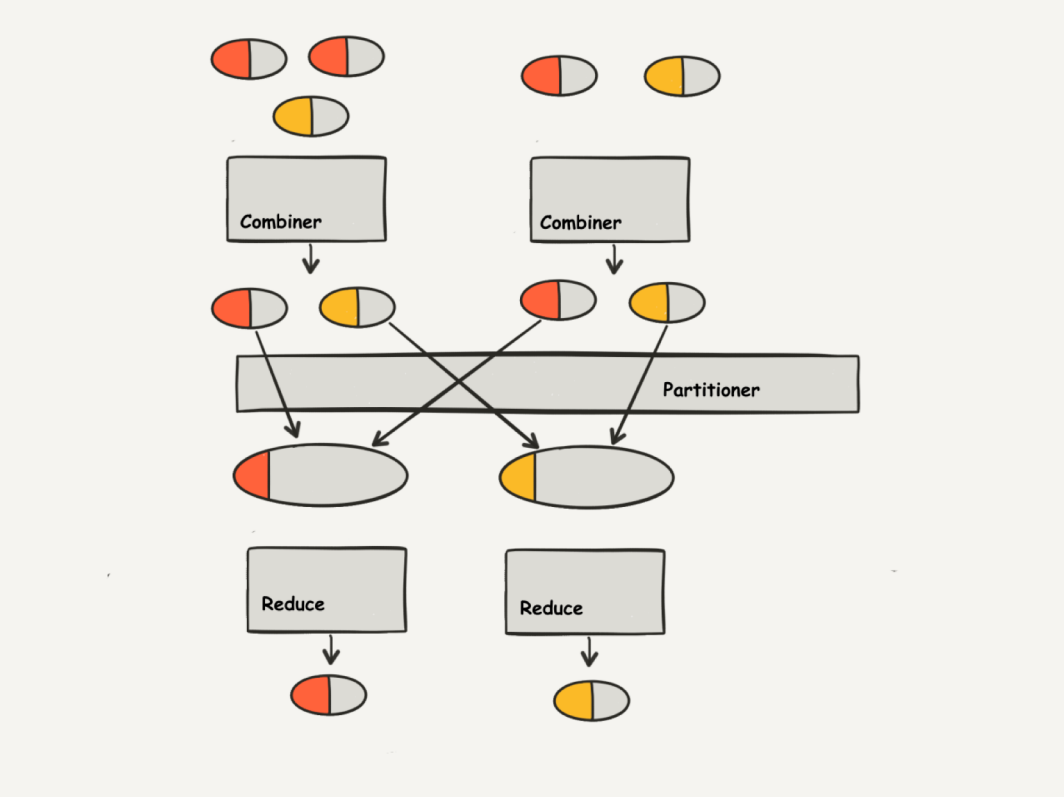
Shuffling



Maas, Garillot, Learning Spark Streaming, O'Reilly, 2017.

- Shuffling is the most "expensive" part of a MapReduce Job
- High demands on resources: CPU, RAM, disk and network IO
- Recommendations:
  - Avoid shuffle if possible
  - Minimize amount of data that gets shuffled
  - Use Compression
- A common benchmark for evaluating the shuffle performance of different frameworks is TeraSort.

Optimizing Shuffling: Combiners



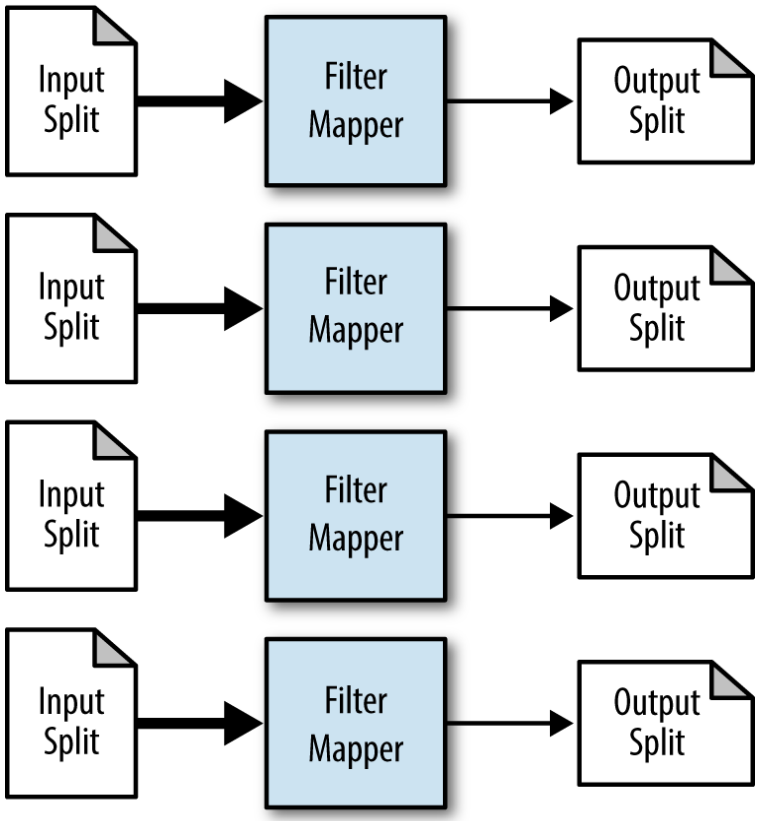
Maas, Garillot, Learning Spark Streaming, O'Reilly, 2017.

- **Combiner:** The combiner, an optional localized reducer, can group data in the map phase. It takes the intermediate keys from the mapper and applies a user-provided method to aggregate values in the small scope of that one mapper.
- Combiners allow the summarization of intermediate results as early as possible, before paying the price of sending data across the network.

Common MapReduce Patterns

- Framework for solving your data computation issues, without being specific to the problem domain.
- See Miner, Shook, MapReduce Design Patterns, O'Reilly, 2012.

MapReduce Patterns: Filtering

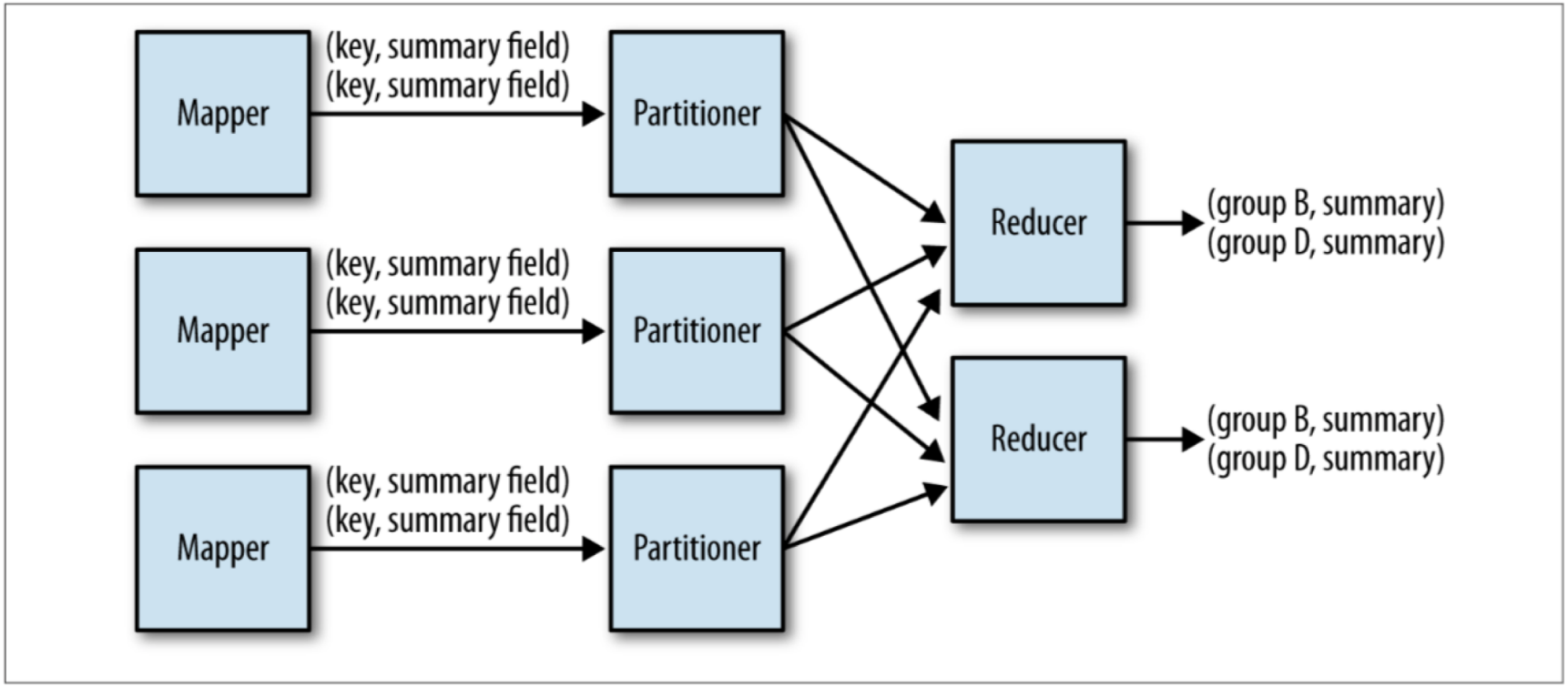



```
map(key, record):
   if we want to keep record then
      emit key,value
```

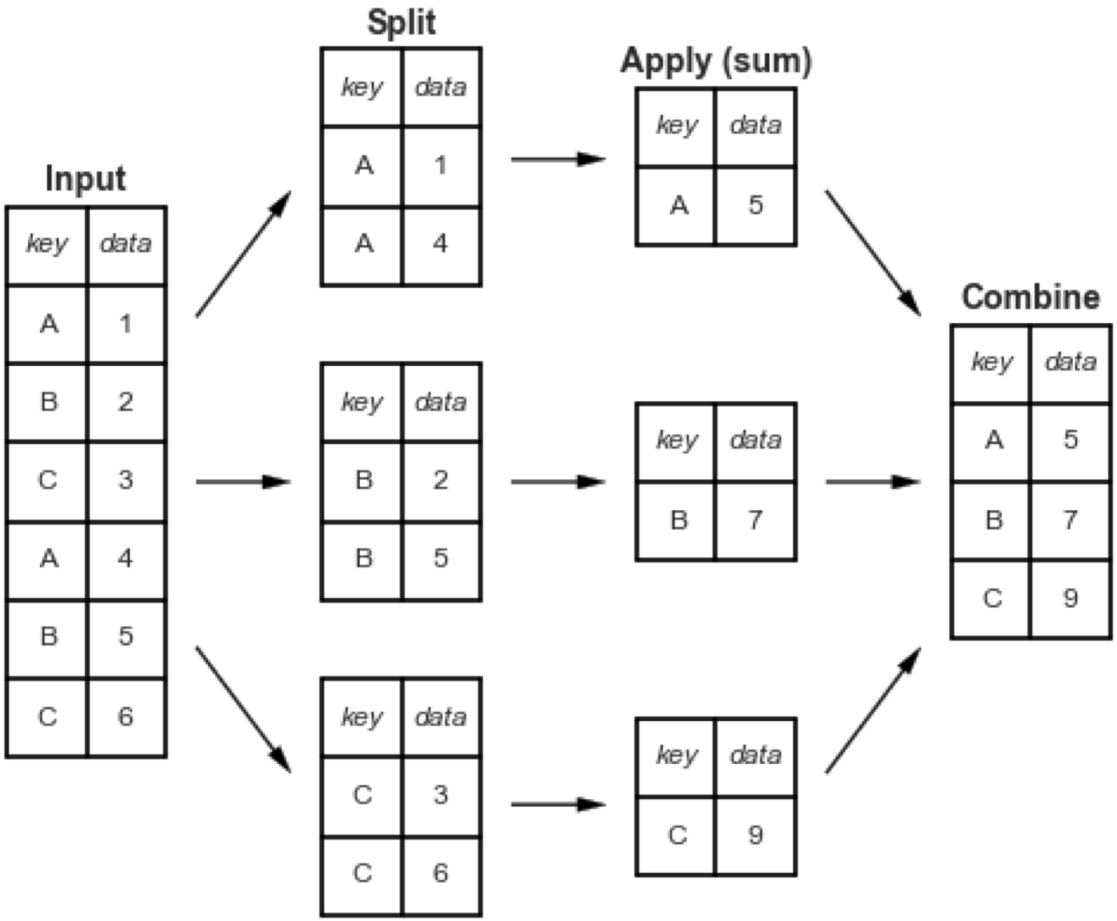
- Filter out records that are not of interest and keep ones that are.
- Examples:
  - Distributed Grep
  - Data cleaning/preparation
  - Sampling
  - Removing low-scoring data
- Embarrassingly parallel: no reducer and shuffle needed!

MapReduce Patterns: Summarization

Intent: Group records together by a key field and calculate a numerical aggregate per group to get a top-level view of the larger data set.
Example: Word Count, Average/Median/Standard deviation



MapReduce Pattern: Summarization



Split-Apply-Combine-Pattern

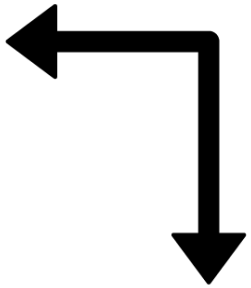
https://jakevdp.github.io/PythonDataScienceHandbook/03.08-aggregation-and-grouping.html

MapReduce Patterns: Summarization Min/Max/Count

Map Output / Combiner Input

	Input Key	Input Value		
	User	Minimum	Maximum	Count
Group 1	12345	10	10	1
	12345	8	8	1
	12345	21	21	1
Group 2	54321	1	1	1
	54321	47	47	1
	99999	7	7	1
	99999	12	12	1

Combiner executes over Group 1 and 2.
Does not execute over last two rows.



Combiner Output / Reducer Input

Output Key	Output Value		
	Minimum	Maximum	Count
12345	8	21	3
54321	1	47	2
99999	7	7	1
99999	12	12	1

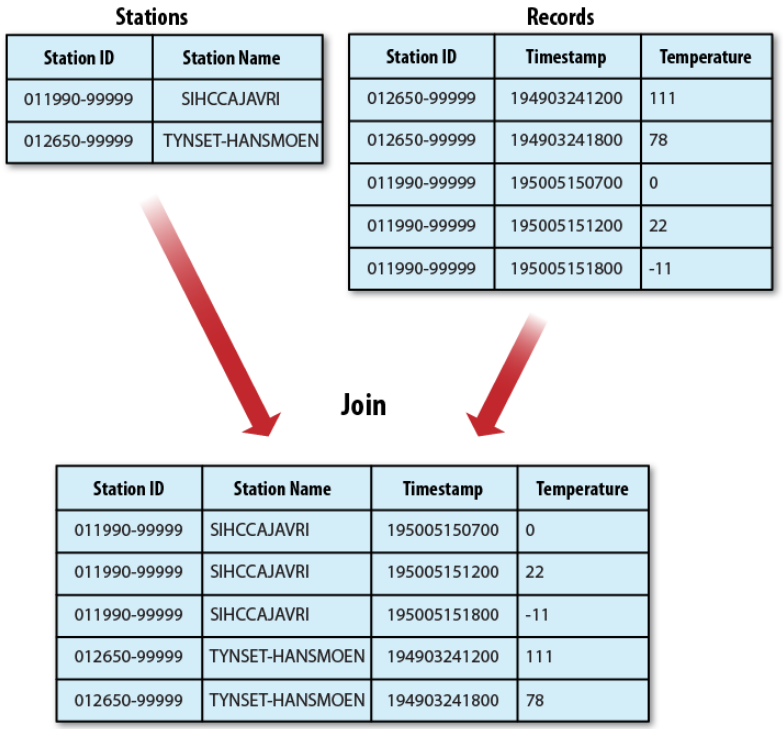
Donald Miner, Adam Shook, MapReduce Design Patterns, O'Reilly, 2012 (page 25).

MapReduce Patterns: Summarization Average

- An average is not an associative operation!
- The combiner cannot be used compute average directly
- Implement combiner by keeping track of the count and average (or sum).
- These two values can be multiplied to preserve the sum for the final reduce phase.

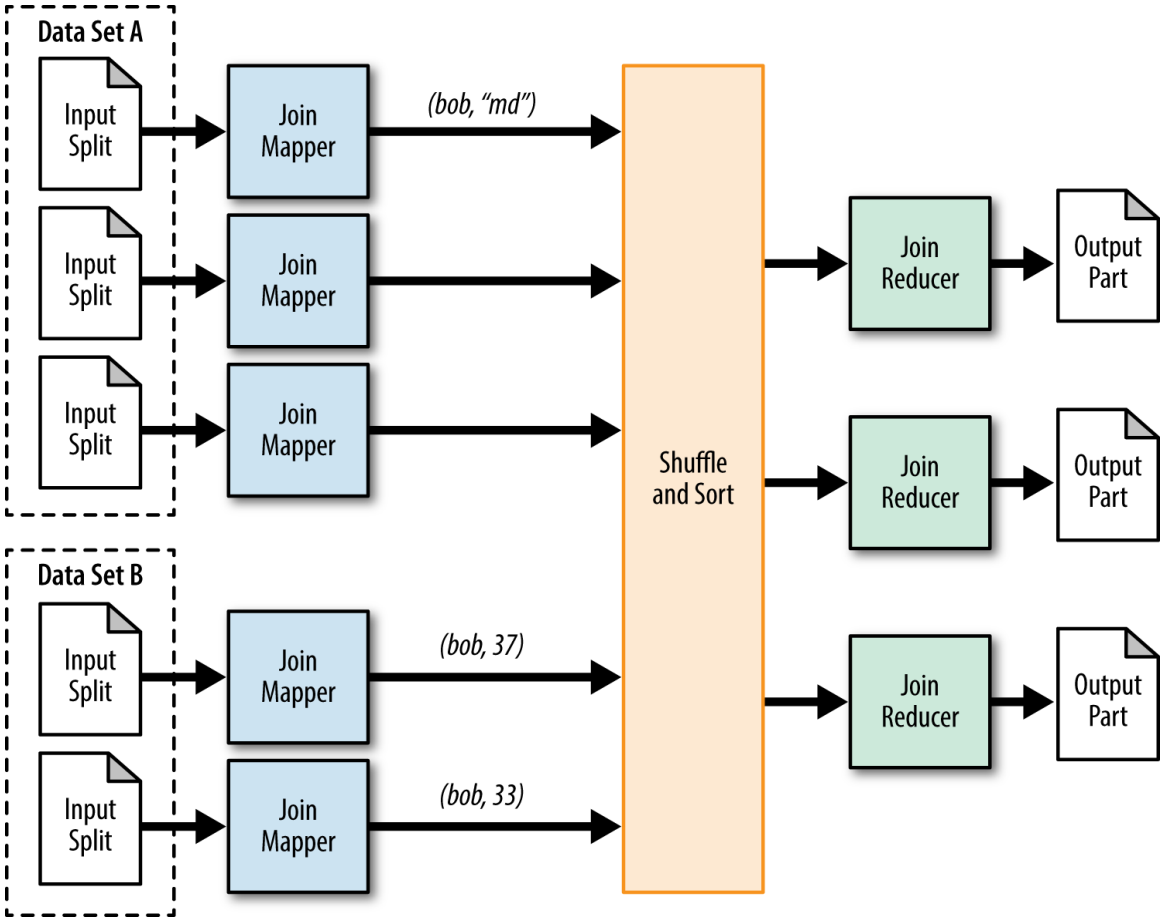
MapReduce Patterns: Joins

- **Reduce-Side Join:** more general, but both datasets have to go through shuffle phase (expensive!)
- **Map-Side Join:**
  - **Replicated:** one dataset is replicated to all Map tasks. Only feasible for small datasets.
  - **Composite:** Each input dataset must be divided into the same number of partitions and sort with the same key



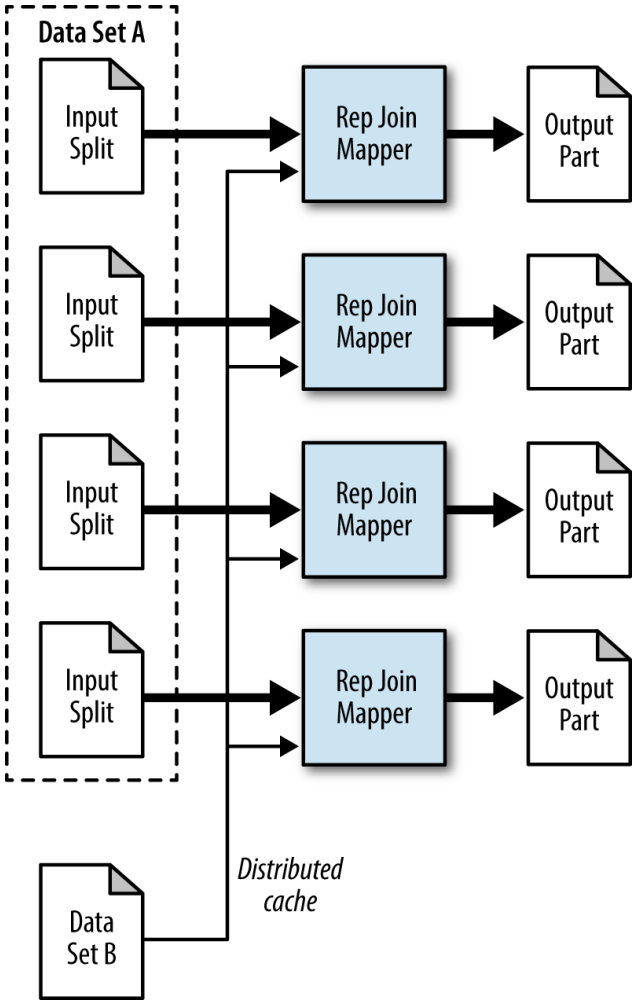
Source: T. White, Hadoop: The Definitive Guide, 2012

Reduce-Side Join

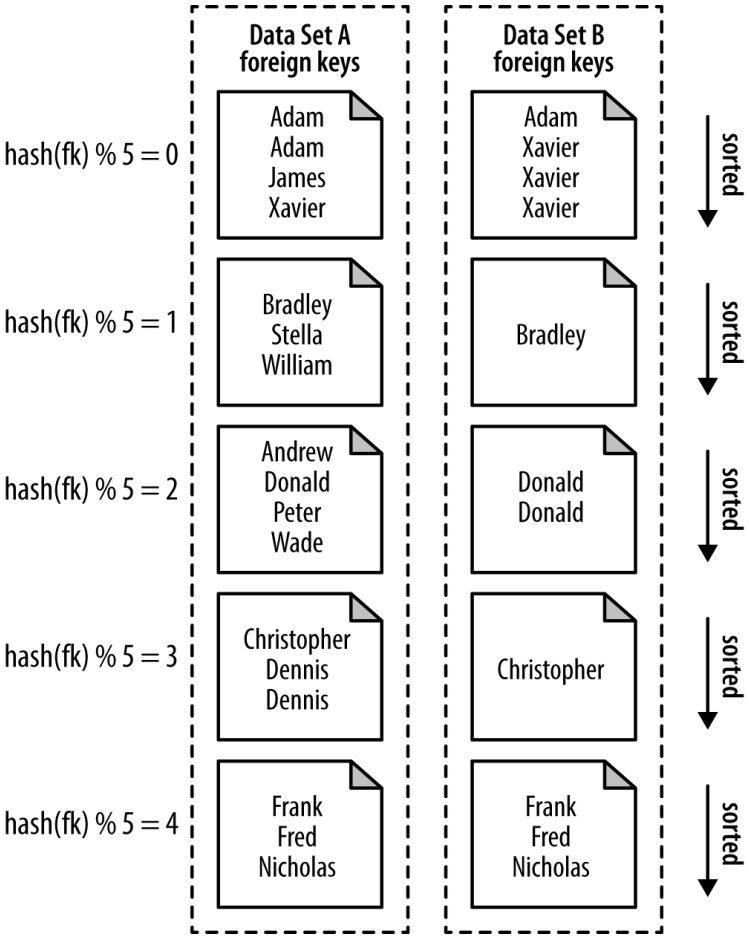


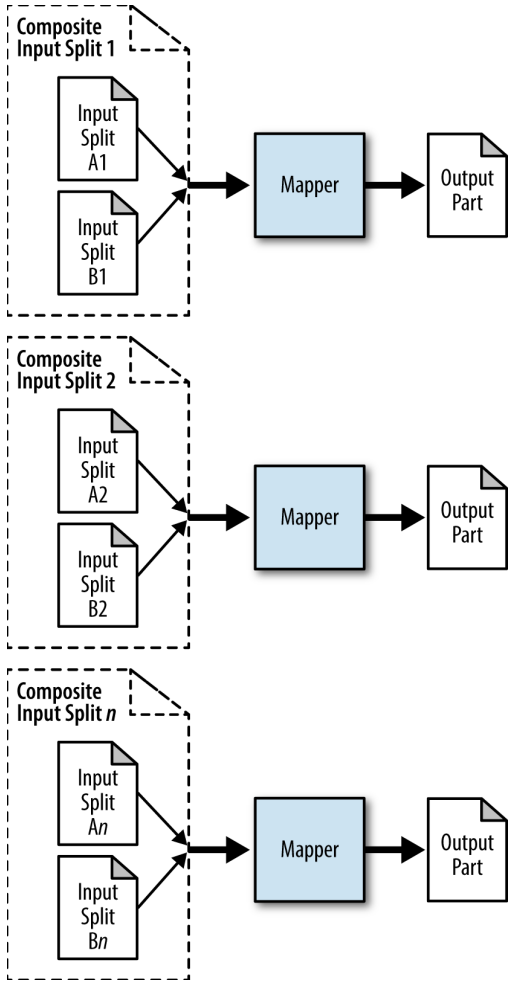
Donald Miner, Adam Shook, MapReduce Design Patterns, O'Reilly, 2012 (page 25).

Map-Side Join (Replicated)



Map-Side Join (Composite)





Donald Miner, Adam Shook, MapReduce Design Patterns, O'Reilly, 2012 (page 25).

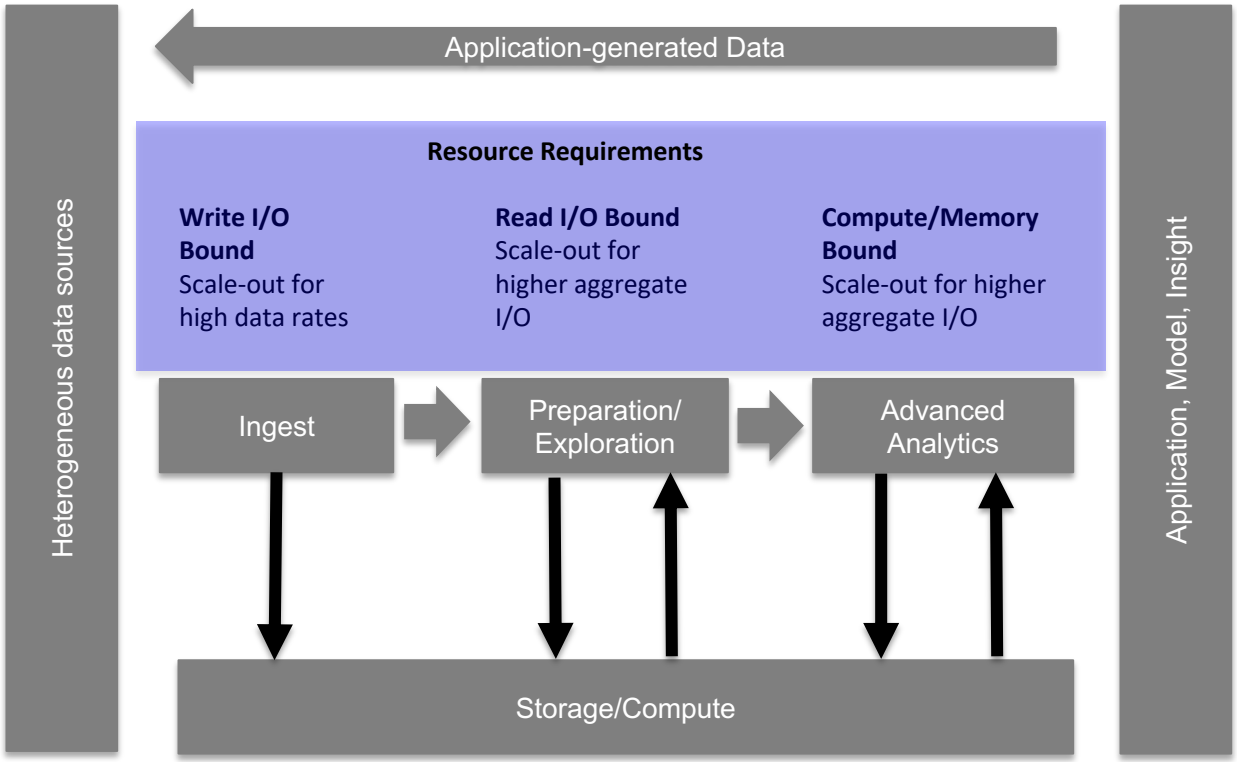
- **Machine Learning dataflow:** Iterative computations and keep track of state
  - There is both model and data, but only communicate the model
  - Collective communication operations, such as AllReduce AllGather, important
- We will revisit MapReduce patterns for Machine Learning in Scalable Machine Learning part of lecture.

Limitations of MapReduce

- **Iterative jobs:** Many common machine learning algorithms apply a function repeatedly to the same dataset to optimize a parameter (e.g., through gradient descent). While each iteration can be expressed as a MapReduce/Dryad job, each job must reload the data from disk, incurring a significant performance penalty.
- **Interactive analytics:** Hadoop is often used to run ad-hoc exploratory queries on large datasets, through SQL interfaces such as Pig and Hive. Ideally, a user would be able to load a dataset of interest into memory across a number of machines and query it repeatedly. However, with Hadoop, each query incurs significant latency (tens of seconds) because it runs as a separate MapReduce job and reads data from disk.
- **Caching layer needed for iterative jobs and interactive analytics**
- **Other criticism:**
  - **Rigid programming model**
  - **Not all algorithms suited for MapReduce**

http://static.usenix.org/legacy/events/hotcloud10/tech/full_papers/Zaharia.pdf

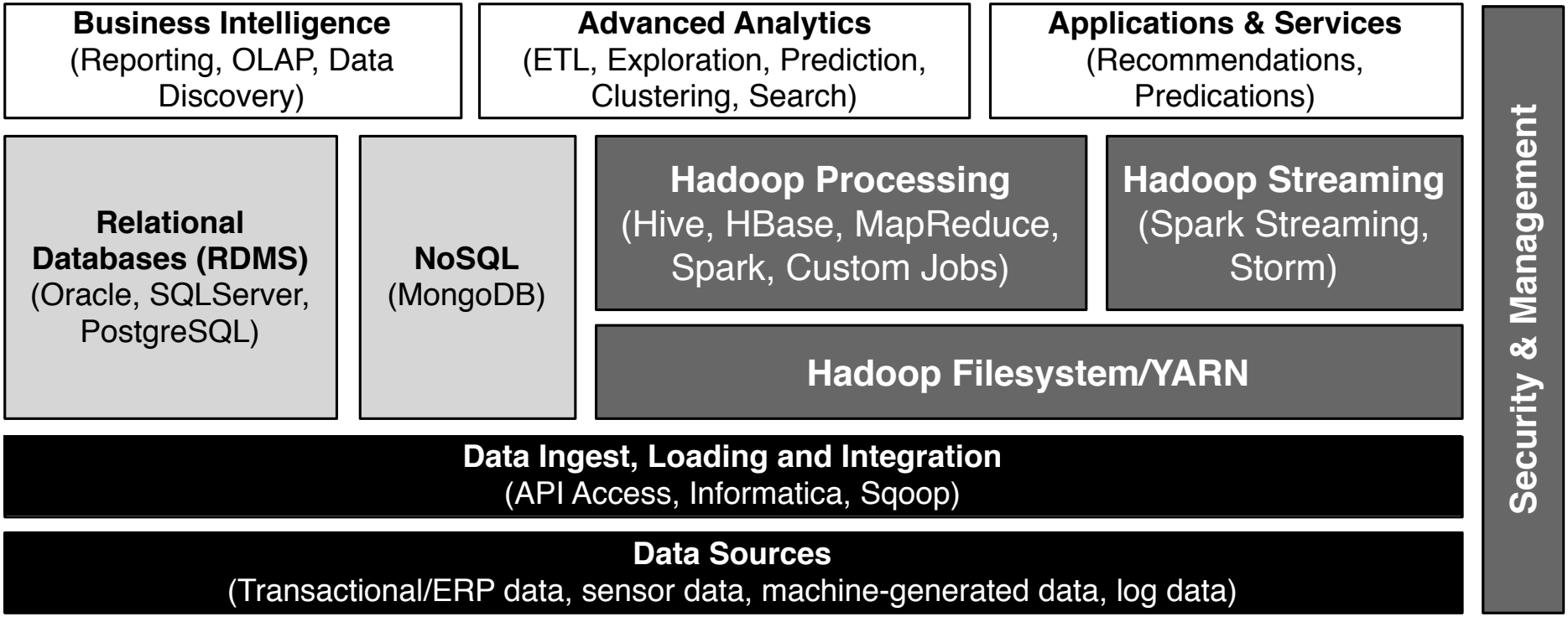
Data Lifecycle and Challenges



- **Data Lake** refers to the ability to retain large volumes of raw data in its original form in a Hadoop infrastructure and utilizing Hadoop-based tools to process, refine, combine and analyze the data.

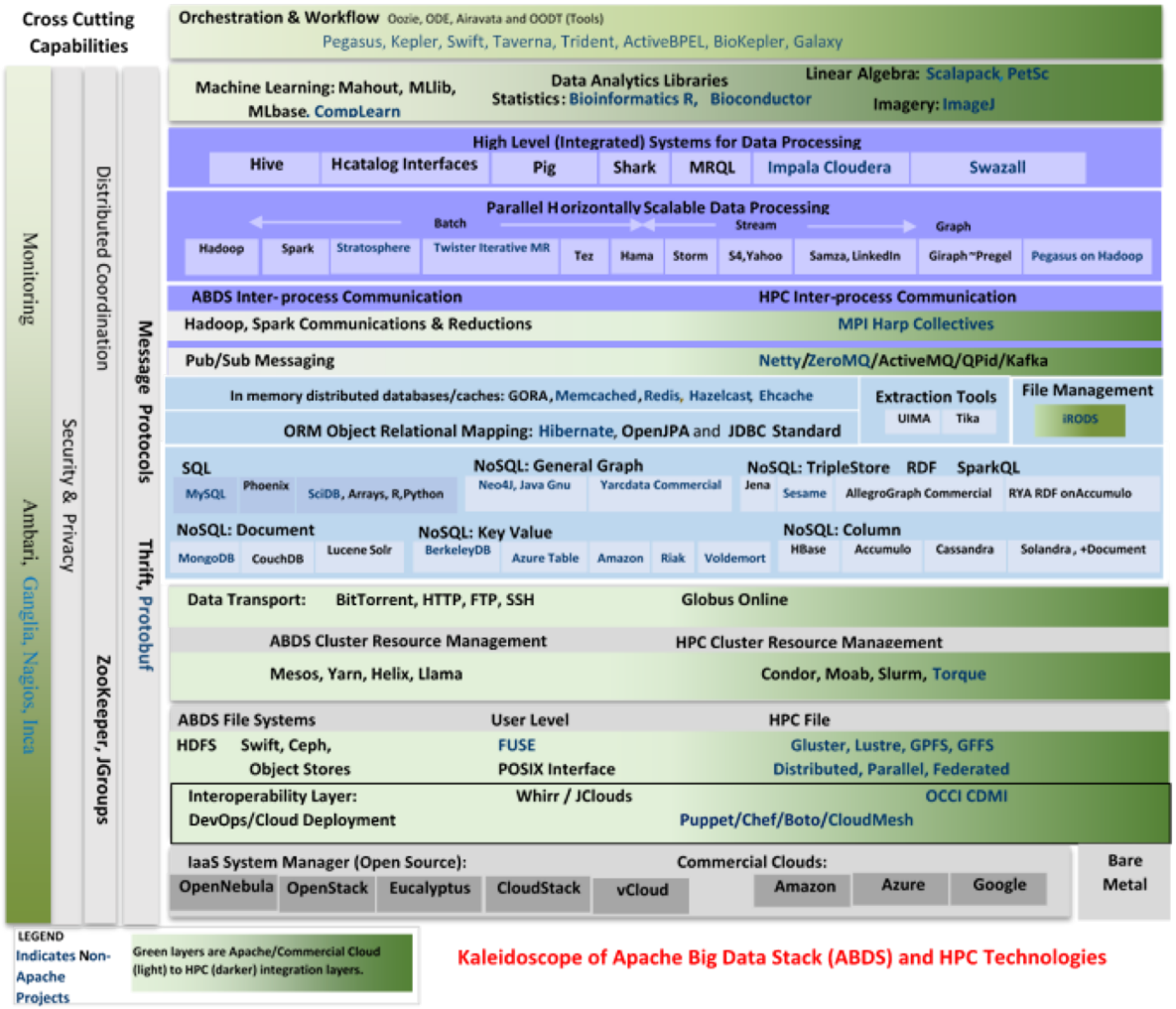
- **Properties:**
  - Economically storage and processing based on commodity hardware and open source
  - Flexible use: Different open data format (Text, ORC, Parquet) accessed using schema-on-read can be combined with different forms of processing (MapReduce, Spark/RDD, SQL, Machine Learning)

Apache Hadoop Stack



Apache Hadoop & HPC Integrated

On-going research
Indiana University,
Rutgers



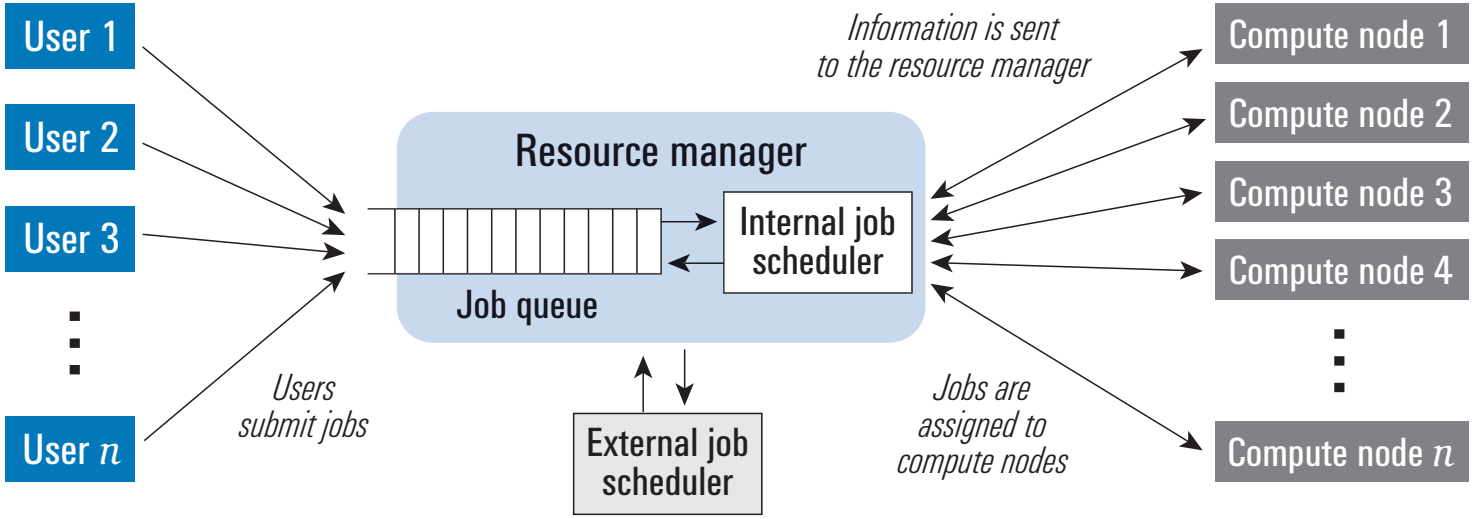
Kaleidoscope of Apache Big Data Stack (ABDS) and HPC Technologies

G. Fox, Big Data at the Intersection of Clouds and HPC, http://grids.ucs.indiana.edu/ptliupages/publications/

Resource Management

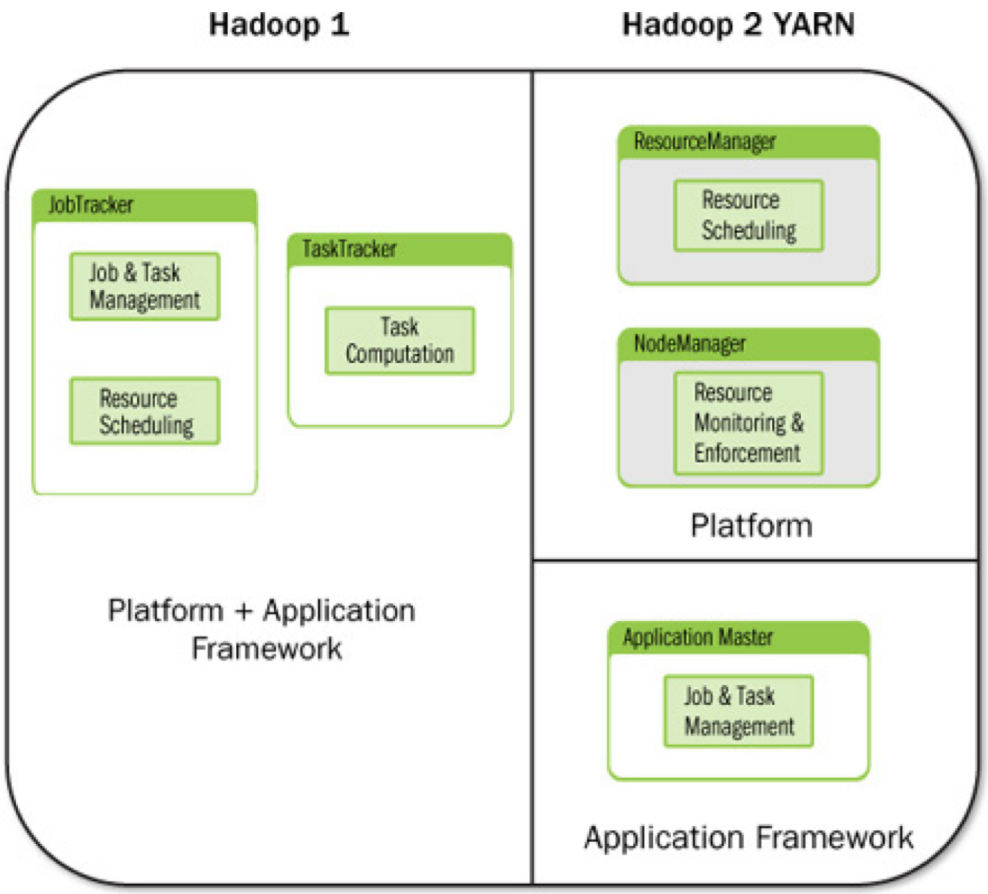
Cluster Scheduling

- The Resource Management System handles the computational resources of the cluster (CPU, Memory).
- Introduced in the context cluster computing:
  - Examples: SLURM, PBS, Torque, LSF. Palmetto uses PBS (http://citi.clemson.edu/palmetto/pages/userguide.html)



Iqbal et al., Job Scheduling in HPC Clusters, 2005

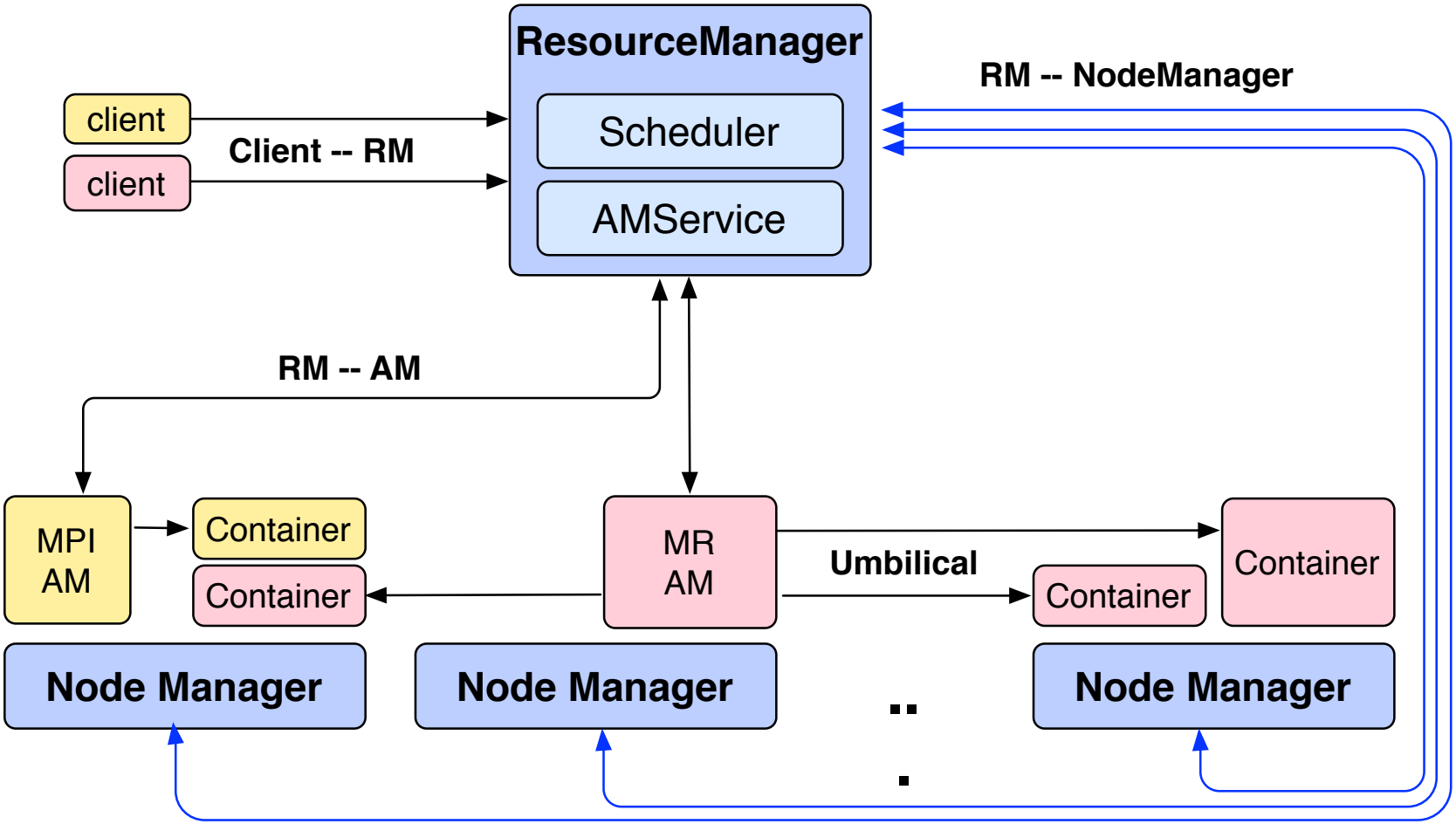
Hadoop Resource Management



- YARN decouples resource management and application framework
- YARN was specially designed for data-intensive applications
- Main differences to HPC:
  - Short-running, fine-grained data parallelism – **many small tasks in contrast to few large parallel jobs**
  - Negotiable resource requirements
  - Dynamic resource requirements that change during application lifecycle

Murphy et al., Apache Hadoop YARN: Moving beyond MapReduce and Batch Processing with Apache Hadoop 2, Addison-Wesley, 2014

Hadoop Resource Management: YARN



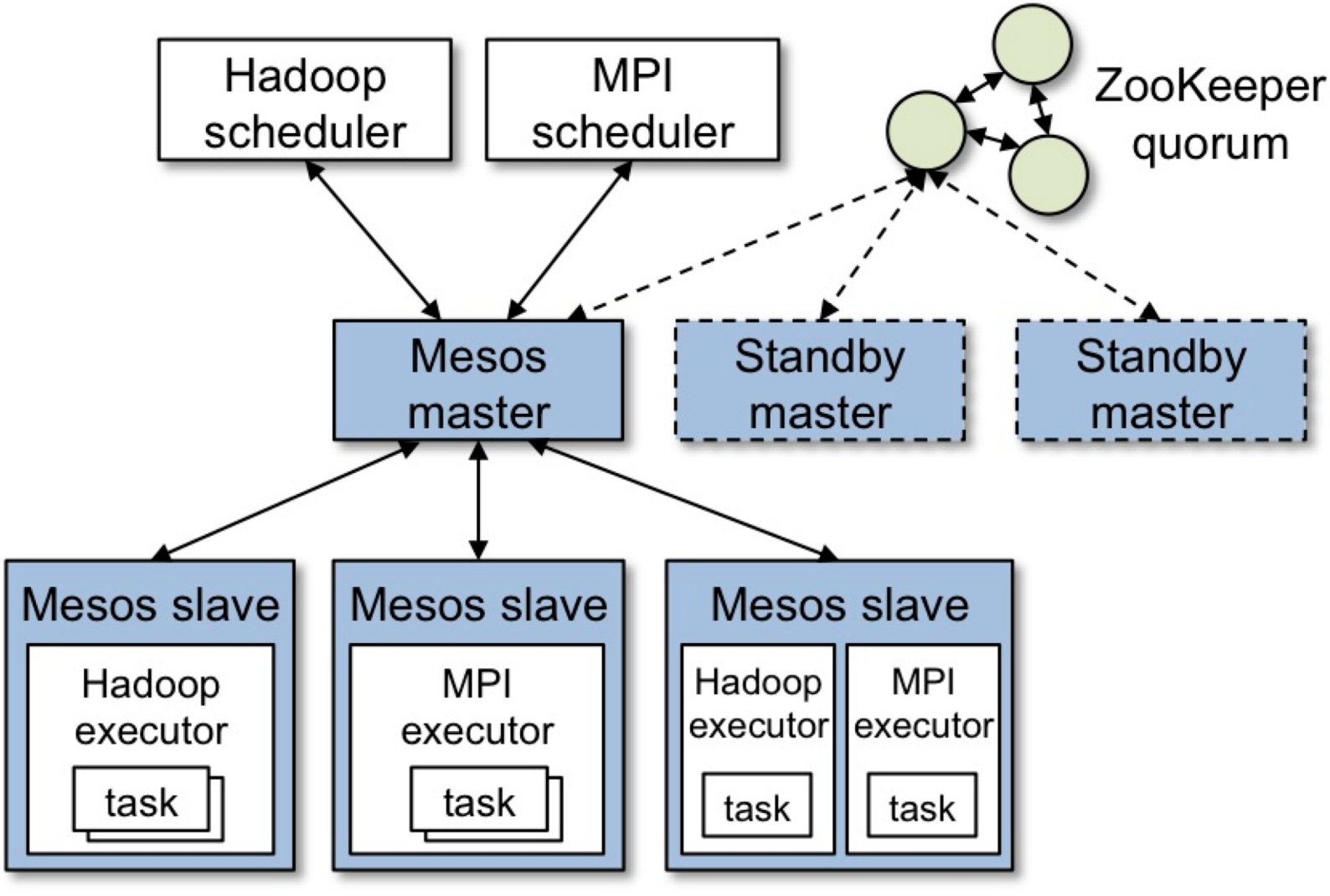
YARN Properties

- Enable fine-grained resource sharing - assigning complete nodes hurts utilization and should be avoided.
- More dynamic than traditional HPC model
- When the container starts, all data files, executables, and necessary dependencies are copied to local storage on the node
- Unlike some resource schedulers in which clients request hard limits, YARN allows applications to adapt (if possible) to the current cluster environment

YARN Properties

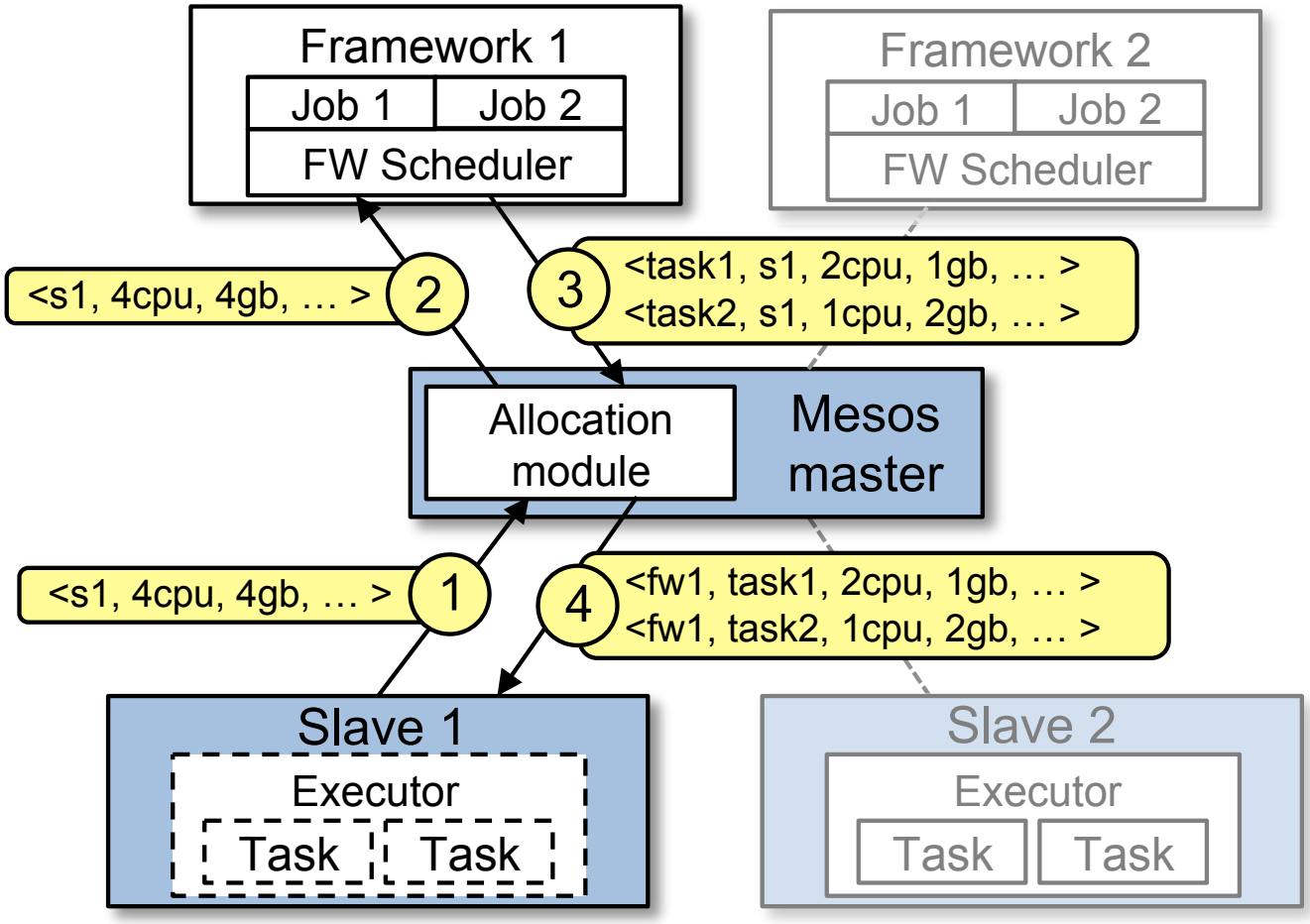
- Location-Awareness: Furthermore, the Capacity scheduler itself is locality aware, and is very good at automatically allocating resources on not only preferred nodes/racks, but also nodes/racks that are close to the preferred ones.
- Pluggable Scheduler:
  - Capacity Scheduler
  - Fair Scheduler
- Container Executor

Mesos Architecture



Hindmann, Mesos: A Platform For Fine-Grained Resource Sharing in the Data Center, https://people.eecs.berkeley.edu/~alig/papers/mesos.pdf

Mesos Architecture



Hindmann, Mesos: A Platform For Fine-Grained Resource Sharing in the Data Center, https://people.eecs.berkeley.edu/~alig/papers/mesos.pdf

High Performance Computing vs. Big Data Computing

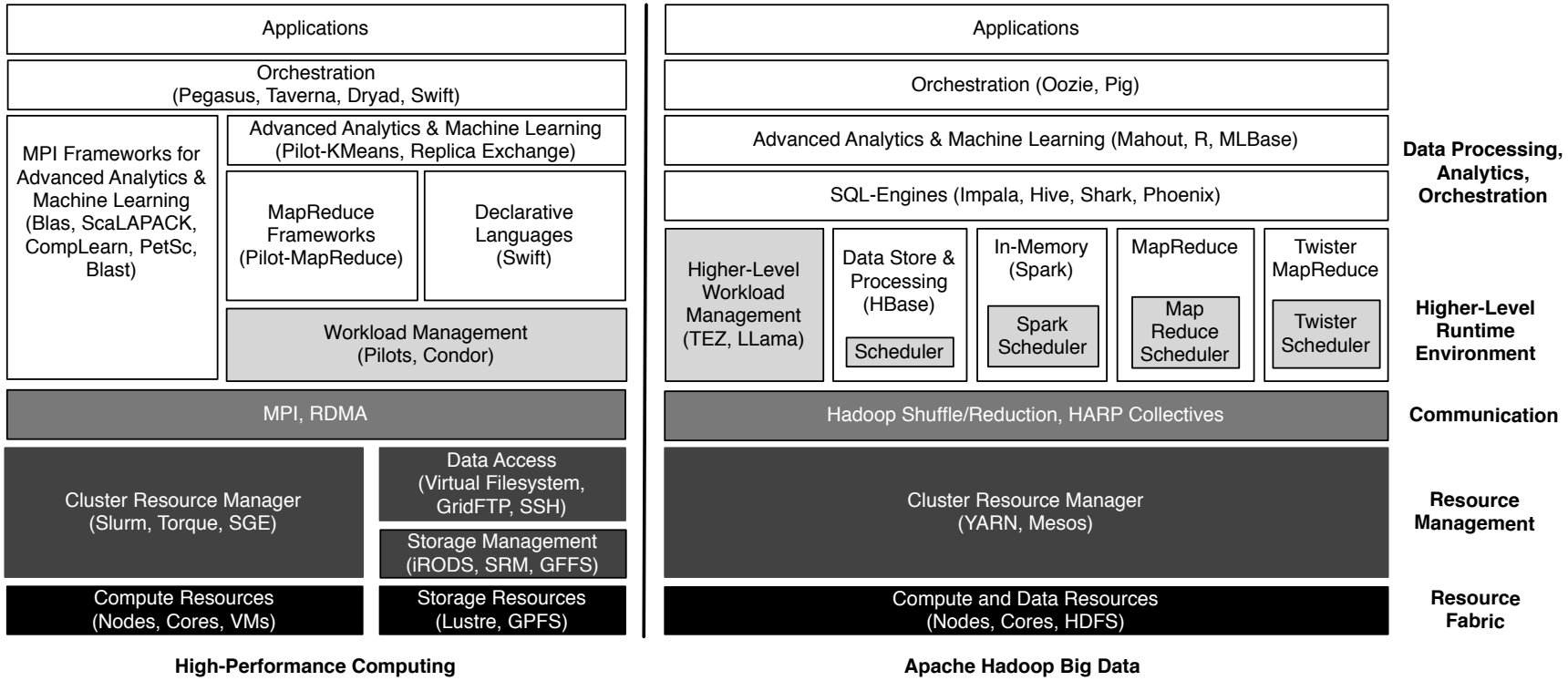


Fig. 1. **HPC and ABDS architecture and abstractions:** The HPC approach historically separated data and compute; ABDS co-locates compute and data. The YARN resource manager heavily utilizes multi-level, data-aware scheduling and supports a vibrant Hadoop-based ecosystem of data processing, analytics and machine learning frameworks. Each approach has a rich, but hitherto distinct resource management and communication capabilities.

Quelle: Shantenu Jha, Judy Qiu, Andre Luckow, Pradeep Mantha, Geoffrey C.Fox: *A Tale of Two Data-Intensive Paradigms: Applications, Abstractions, and Architectures*, https://arxiv.org/pdf/1403.1528.pdf

- Eric Baldeschwieler et al., Apache Hadoop YARN: Yet Another Resource Negotiator, https://www.cs.cmu.edu/~garth/15719/papers/yarn.pdf
- Hindemann, Meso: A Platform for Fine-Grained Resource Sharing in the Data Center, Technical Report, University of California, Berkley, 2010
- Verma et al., Large-scale cluster management at Google with Borg, Proceedings of the European Conference on Computer Systems (EuroSys), ACM, 2015
- Kubernetes, https://kubernetes.io/>
- Burns et al., Borg, Omega, and Kubernetes, ACM Queue, 2016

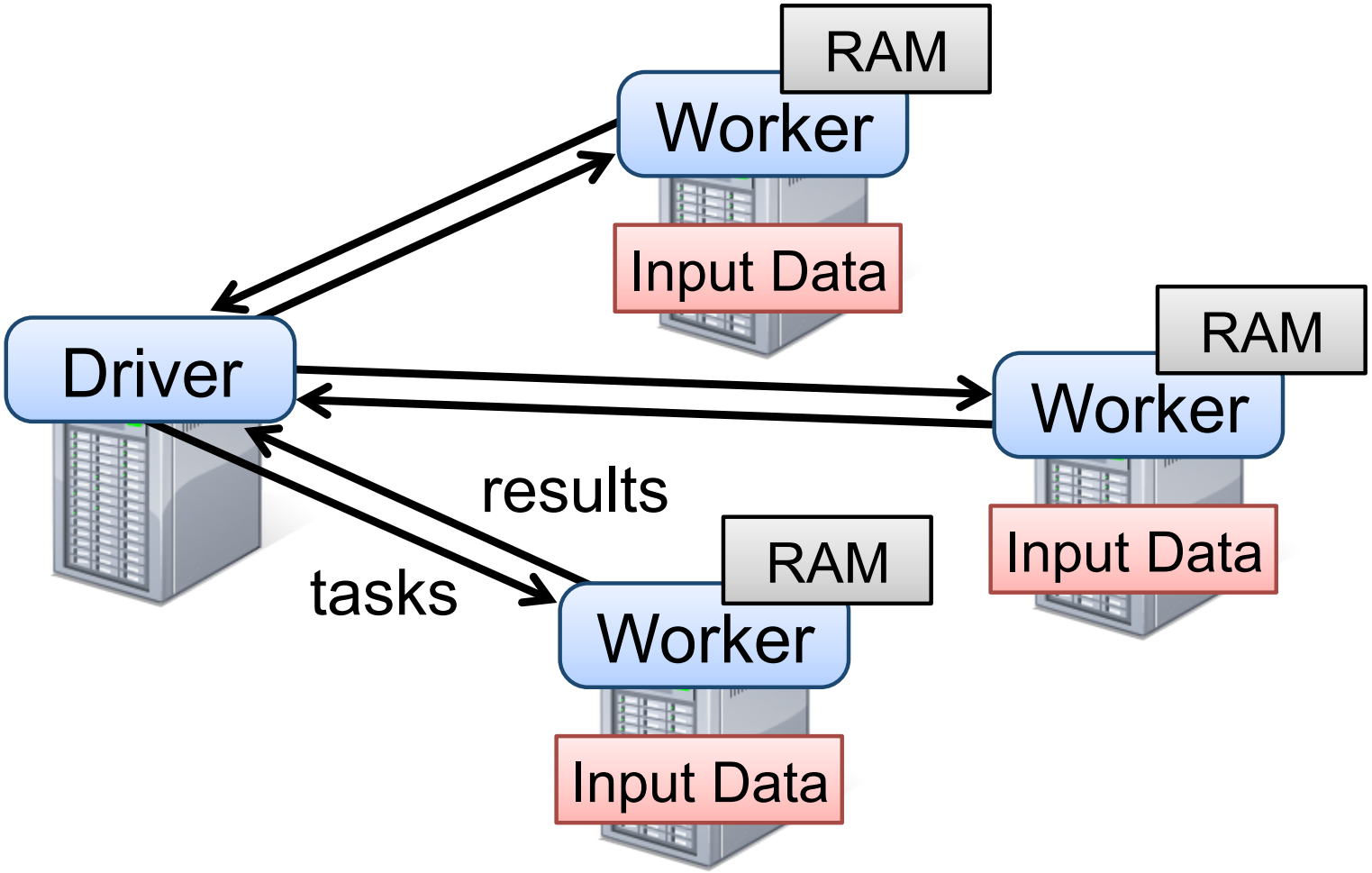
Apache Spark

Adapted from:

Zaharia et al., Resilient Distributed Datasets: A Fault-Tolerant Abstraction forIn-Memory Cluster Computing, https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf, NSDI, 2012

- Fast, expressive in-memory cluster computing system compatible with Apache Hadoop
  - Works with any Hadoop-supported storage system (HDFS, S3, Avro, …)
- **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)

Review: Apache Spark



Source: Stoica, Zaharia et al., An Introduction to the Berkley Data Analytics Stack: http://strataconf.com/strata2013/public/schedule/detail/27431
https://www.cs.berkeley.edu/~matei/papers/2012/nsdi_spark.pdf

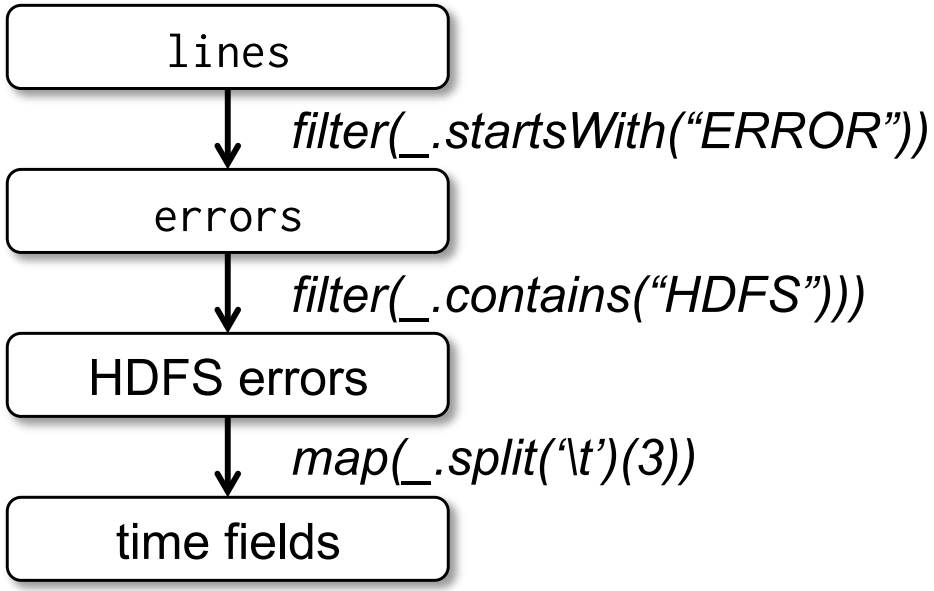
RDD vs. Distributed Shared Memory

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.

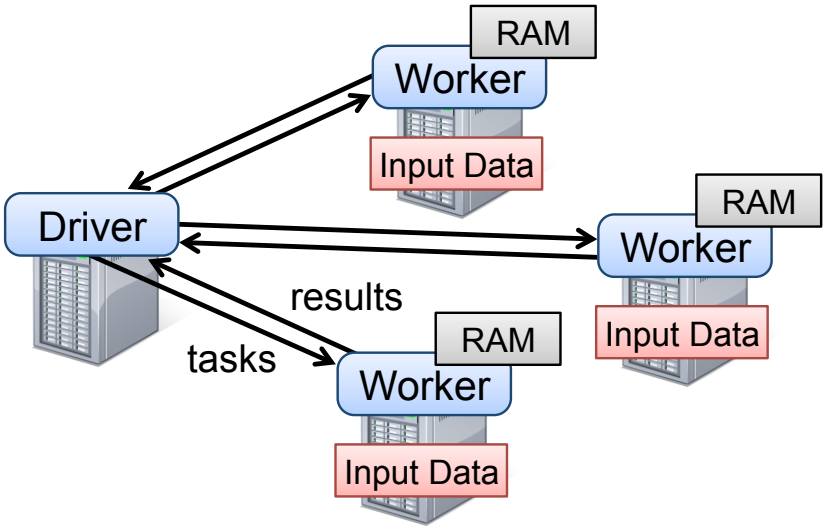
Apache Spark: Operations

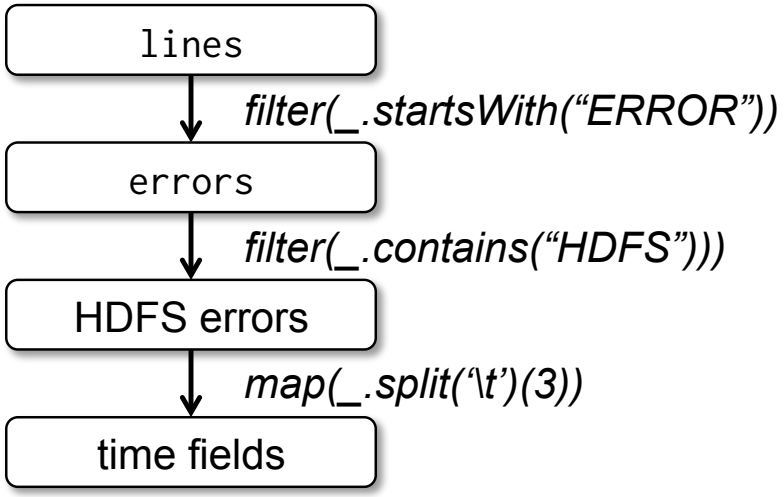
- Transformations (e.g. map, filter, groupBy, join)
  - Lazy operations to build RDDs from other RDDs
- Actions (e.g. collect, save)
  - Return a result or write it to storage
- Analytics Actions (e.g. count, sum, min, max, mean, top)



Source: Stoica, Zaharia et al., An Introduction to the Berkley Data Analytics Stack: http://strataconf.com/strata2013/public/schedule/detail/27438, https://www.cs.berkeley.edu/~matei/papers/2012/nsdi_spark.pdf

Distributed Processing Engines: Apache Spark





Source: Stoica, Zaharia et al., An Introduction to the Berkley Data Analytics Stack:
http://strataconf.com/strata2013/public/schedule/detail/27438
https://www.cs.berkeley.edu/~matei/papers/2012/nsdi_spark.pdf

Spark Transformation and Actions

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.

Apache Spark: Mining Console Logs

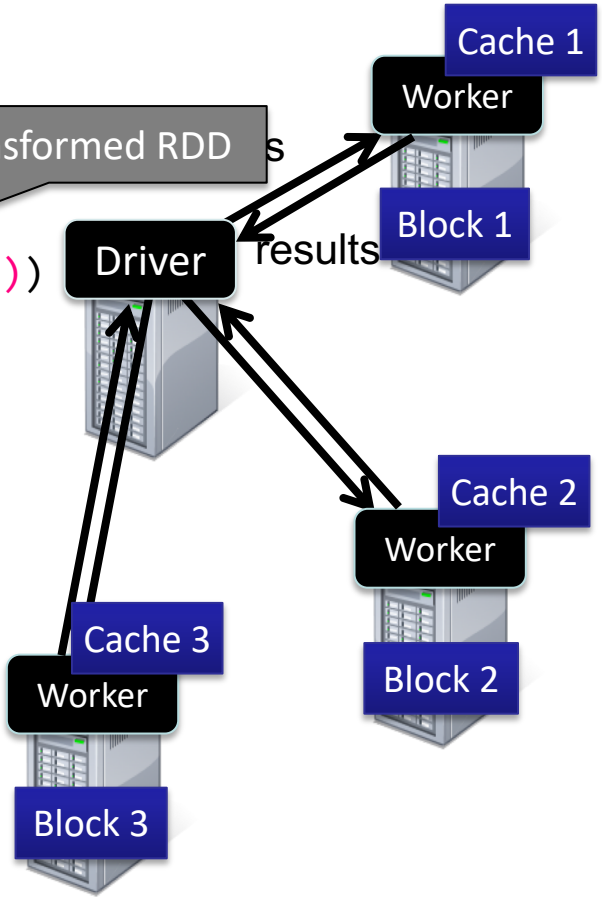
- Load error messages from a log into memory, then interactively search for patterns

```
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()
. . .
```

Base RDD

Transformed RDD

Action

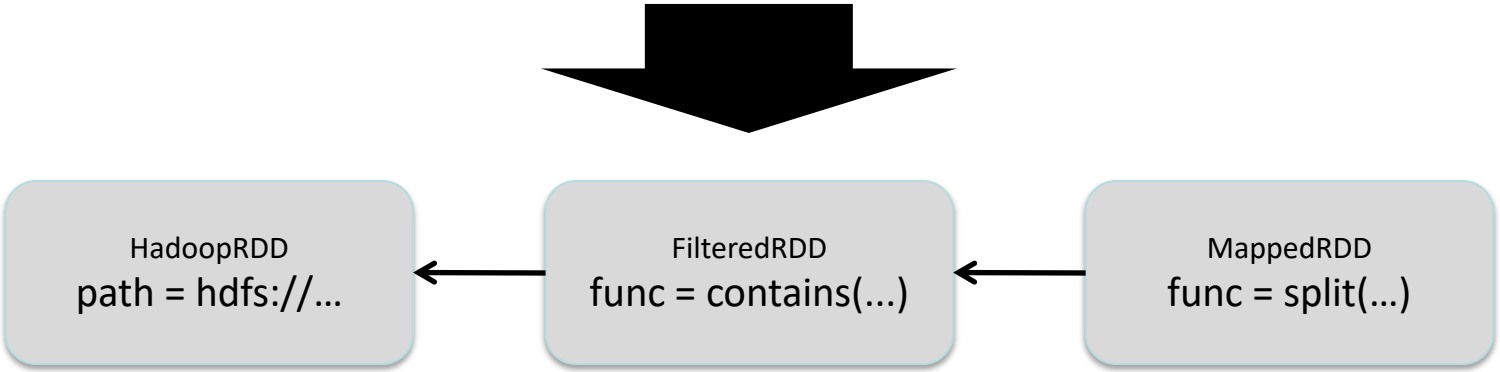


Source: Stoica, Zaharia et al., An Introduction to the Berkley Data Analytics Stack: http://strataconf.com/strata2013/public/schedule/detail/27438
https://www.cs.berkeley.edu/~matei/papers/2012/nsdi_spark.pdf

RDDs track the transformations used to build them (their *lineage*) to re-compute lost data

E.g:

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



Source: Stoica, Zaharia et al., An Introduction to the Berkley Data Analytics Stack: http://strataconf.com/strata2013/public/schedule/detail/27438 https://www.cs.berkeley.edu/~matei/papers/2012/nsdi_spark.pdf

Apache Spark: Spark Shell

- 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
```

- Spark Context:
  - Main entry point to Spark functionality
  - Created for you in Spark shells as variable `sc`

Source: Stoica, Zaharia et al., An Introduction to the Berkley Data Analytics Stack:
http://strataconf.com/strata2013/public/schedule/detail/27438
https://www.cs.berkeley.edu/~matei/papers/2012/nsdi_spark.pdf

Apache Spark: 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://ip-10-186-164-81:8022/path/file”)

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

Source: Stoica, Zaharia et al., An Introduction to the Berkley Data Analytics Stack: http://strataconf.com/strata2013/public/schedule/detail/27438, https://www.cs.berkeley.edu/~matei/papers/2012/nsdi_spark.pdf

Apache Spark: 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.filter(lambda x: x % 2 == 0) # => {4}

# Map each element to zero or more others
nums.flatMap(lambda x: range(0, x))  # => {0, 0, 1, 0, 1,
2}
```

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

Source: Stoica, Zaharia et al., An Introduction to the Berkley Data Analytics Stack: http://strataconf.com/strata2013/public/schedule/detail/27438, https://www.cs.berkeley.edu/~matei/papers/2012/nsdi_spark.pdf

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")
```

Source: Stoica, Zaharia et al., An Introduction to the Berkley Data Analytics Stack: http://strataconf.com/strata2013/public/schedule/detail/27438 https://www.cs.berkeley.edu/~matei/papers/2012/nsdi_spark.pdf

Apache Spark: Key-Value Pairs

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

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

- Scala:

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

- Java: `scala.Tuple2`

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

Source: Stoica, Zaharia et al., An Introduction to the Berkley Data Analytics Stack:
http://strataconf.com/strata2013/public/schedule/detail/27438
https://www.cs.berkeley.edu/~matei/papers/2012/nsdi_spark.pdf

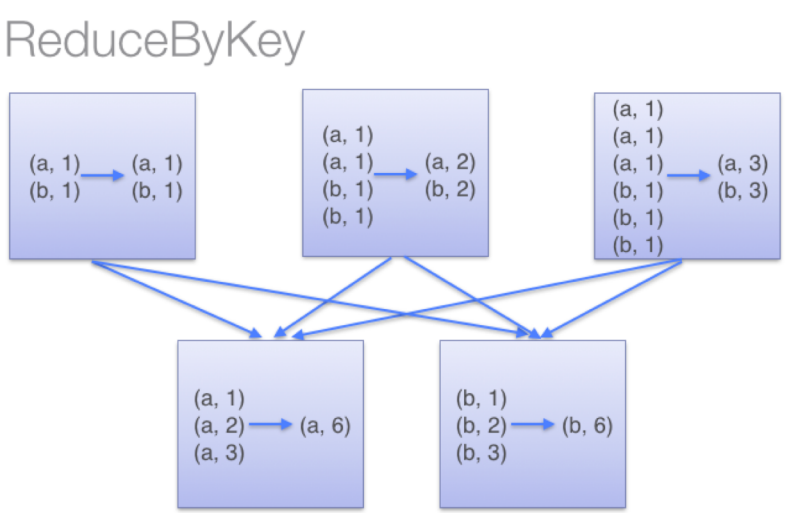
Apache Spark: Key-Value Operations

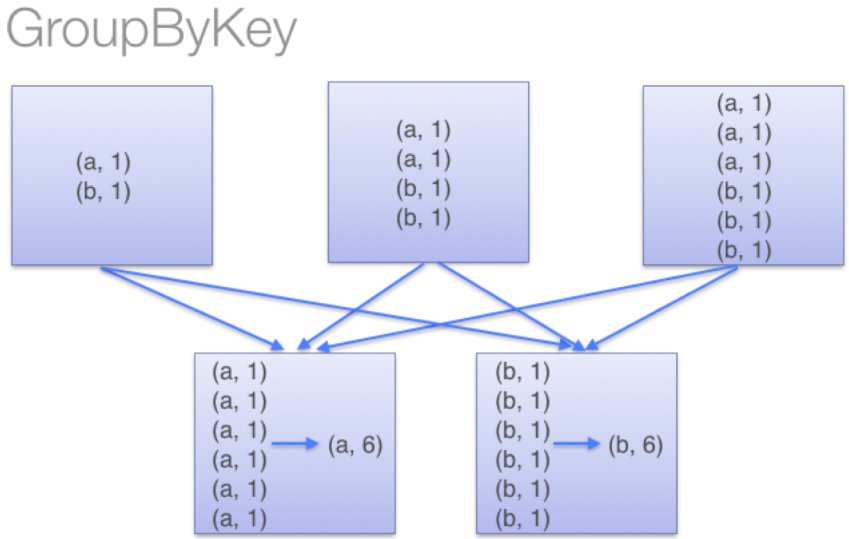

```
pets = sc.parallelize([(“cat”, 1), (“dog”, 1), (“cat”, 2)])

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

Source: Stoica, Zaharia et al., An Introduction to the Berkley Data Analytics Stack:
http://strataconf.com/strata2013/public/schedule/detail/27438
https://www.cs.berkeley.edu/~matei/papers/2012/nsdi_spark.pdf

ReduceByKey versus GroupByKey

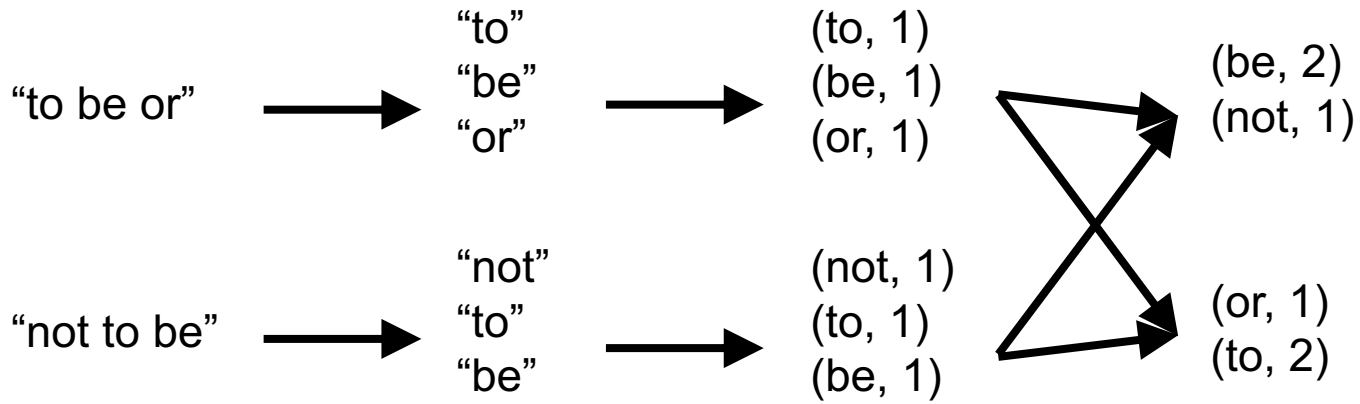




https://databricks.gitbooks.io/databricks-spark-knowledge-base/content/best_practices/prefer_reducebykey_over_groupbykey.html

Apache Spark: 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)
```



Source: Stoica, Zaharia et al., An Introduction to the Berkley Data Analytics Stack:
http://strataconf.com/strata2013/public/schedule/detail/27438
https://www.cs.berkeley.edu/~matei/papers/2012/nsdi_spark.pdf

Apache Spark: 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")))
```

Source: Stoica, Zaharia et al., An Introduction to the Berkley Data Analytics Stack: http://strataconf.com/strata2013/public/schedule/detail/27438, https://www.cs.berkeley.edu/~matei/papers/2012/nsdi_spark.pdf

Apache Spark: 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)
```

Source: Stoica, Zaharia et al., An Introduction to the Berkley Data Analytics Stack:
http://strataconf.com/strata2013/public/schedule/detail/27438
https://www.cs.berkeley.edu/~matei/papers/2012/nsdi_spark.pdf

Spark Job Stages and Linage

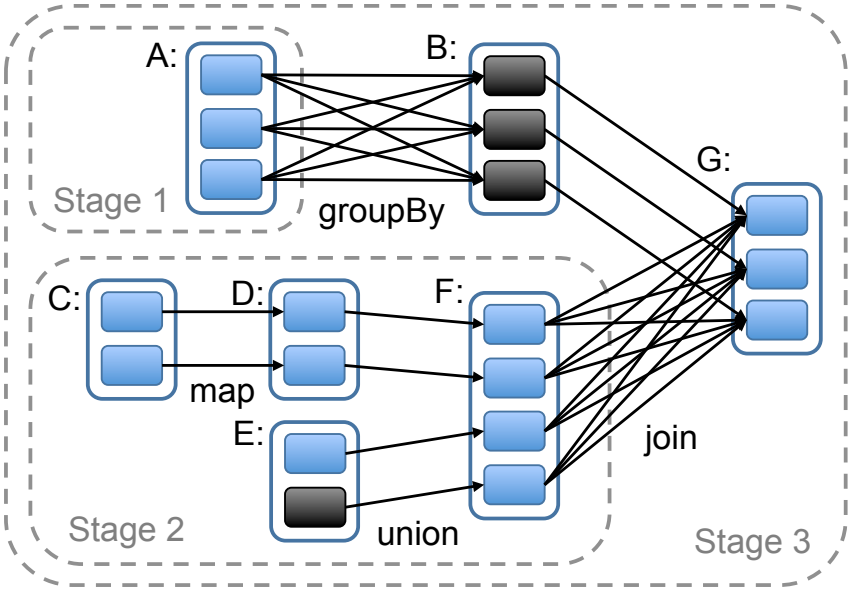
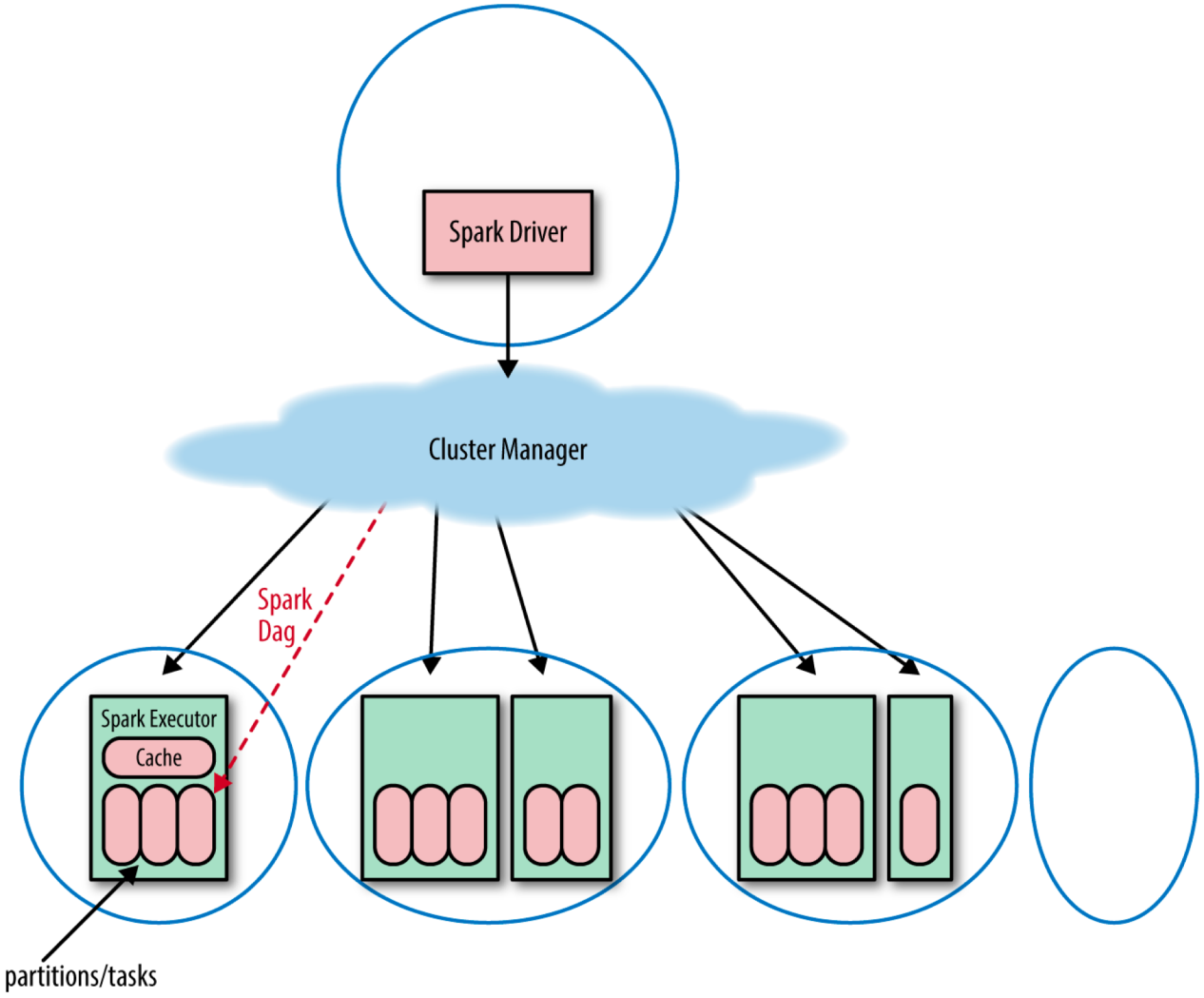


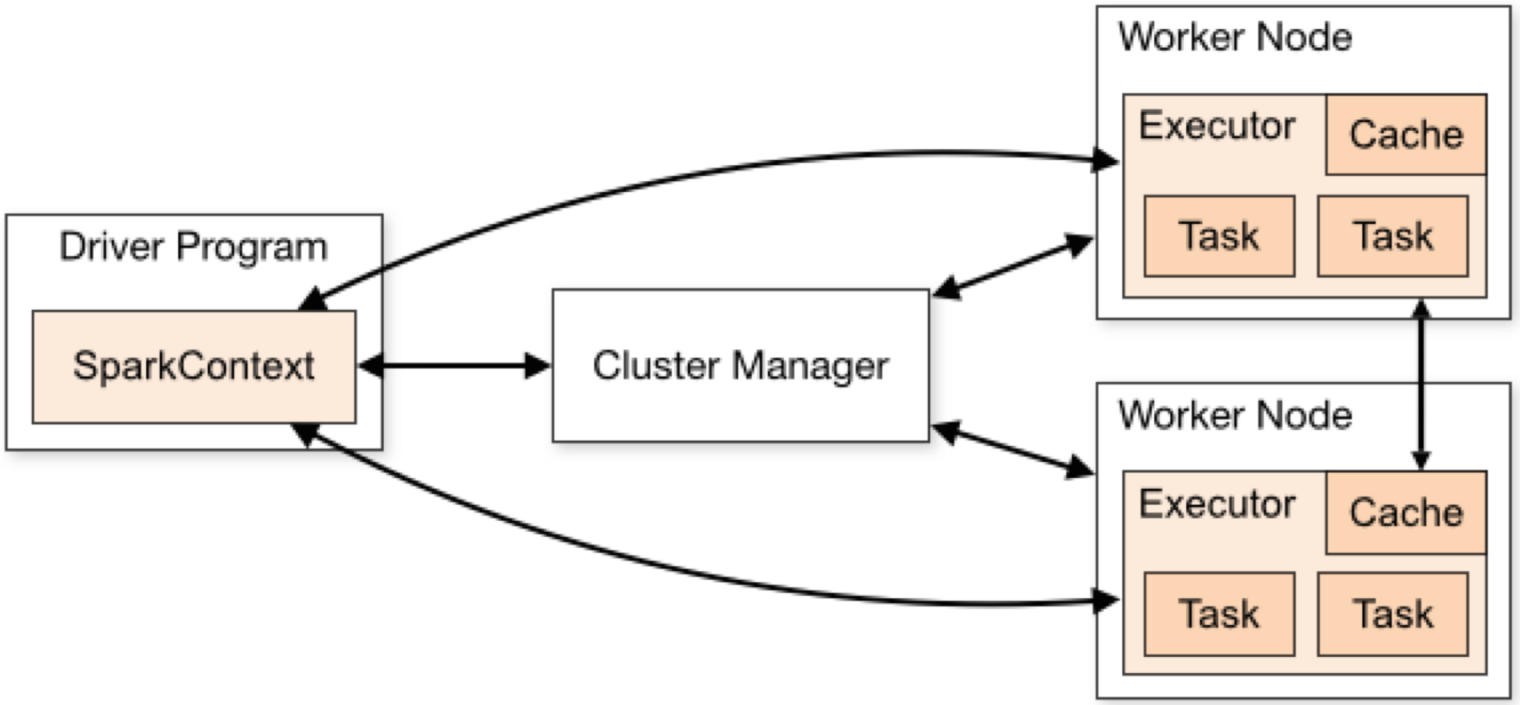
Figure 5: Example of how Spark computes job stages. Boxes with solid outlines are RDDs. Partitions are shaded rectangles, in black if they are already in memory. To run an action on RDD G, we build build stages at wide dependencies and pipeline narrow transformations inside each stage. In this case, stage 1's output RDD is already in RAM, so we run stage 2 and then 3.

Spark Application (1)



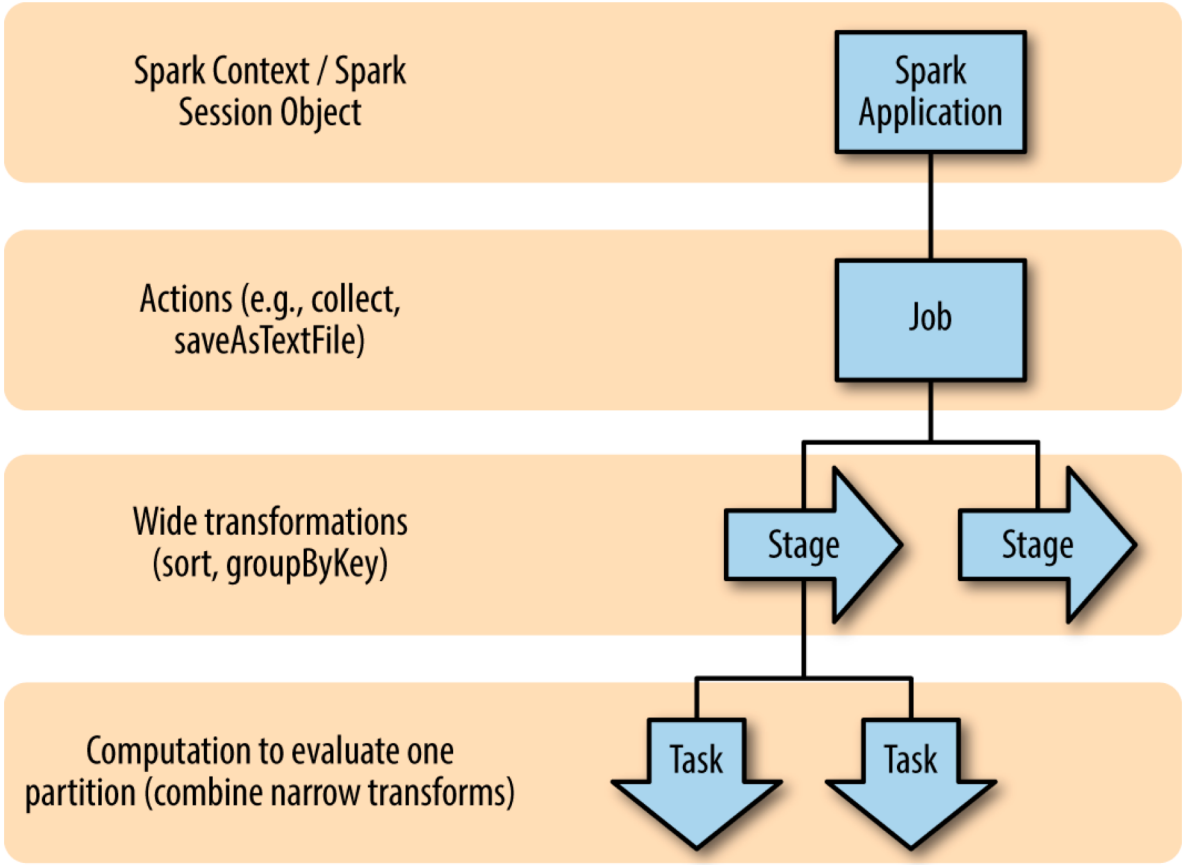
Holden Karau, Rachel Warren, High Performance Spark, O'Reilly, 2017.

Spark Application (3)



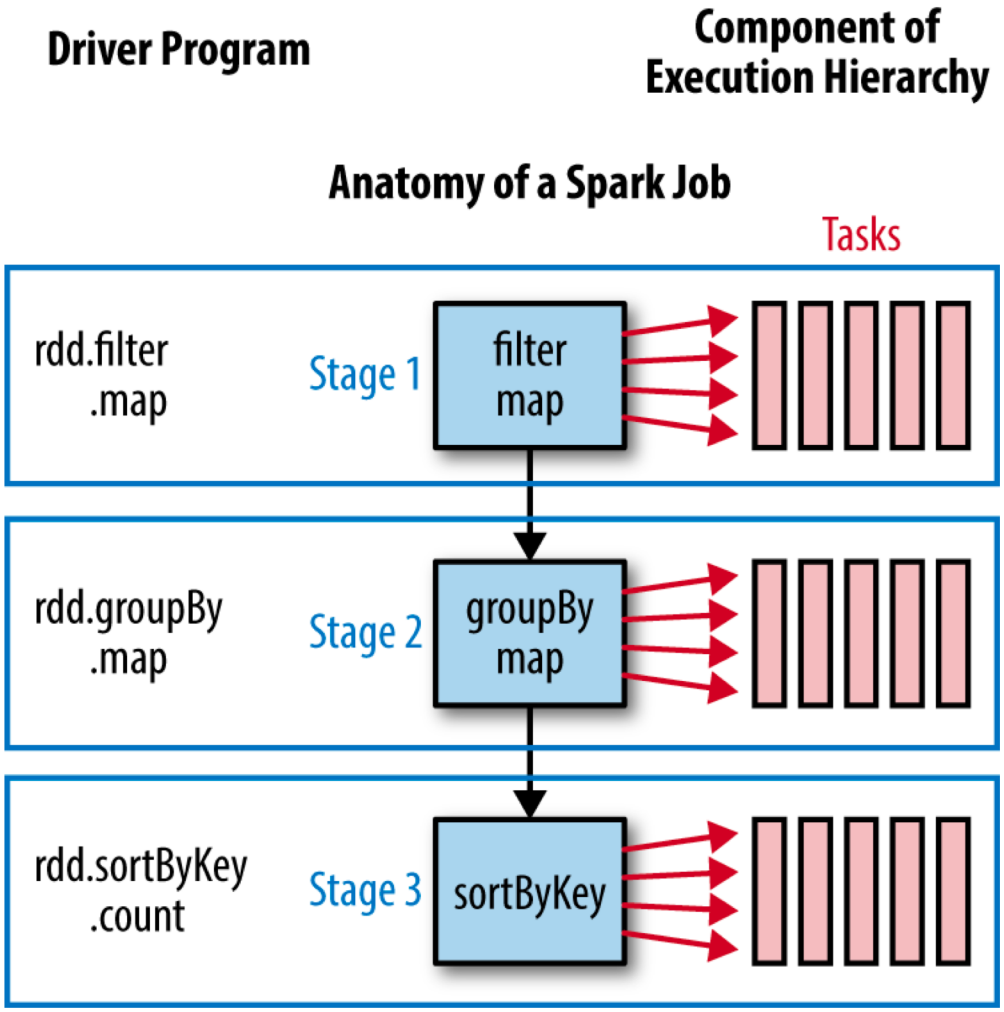
https://databricks.com/blog/2016/08/15/how-to-use-sparksession-in-apache-spark-2-0.html

Spark Application (3)



Holden Karau, Rachel Warren, High Performance Spark, O'Reilly, 2017.

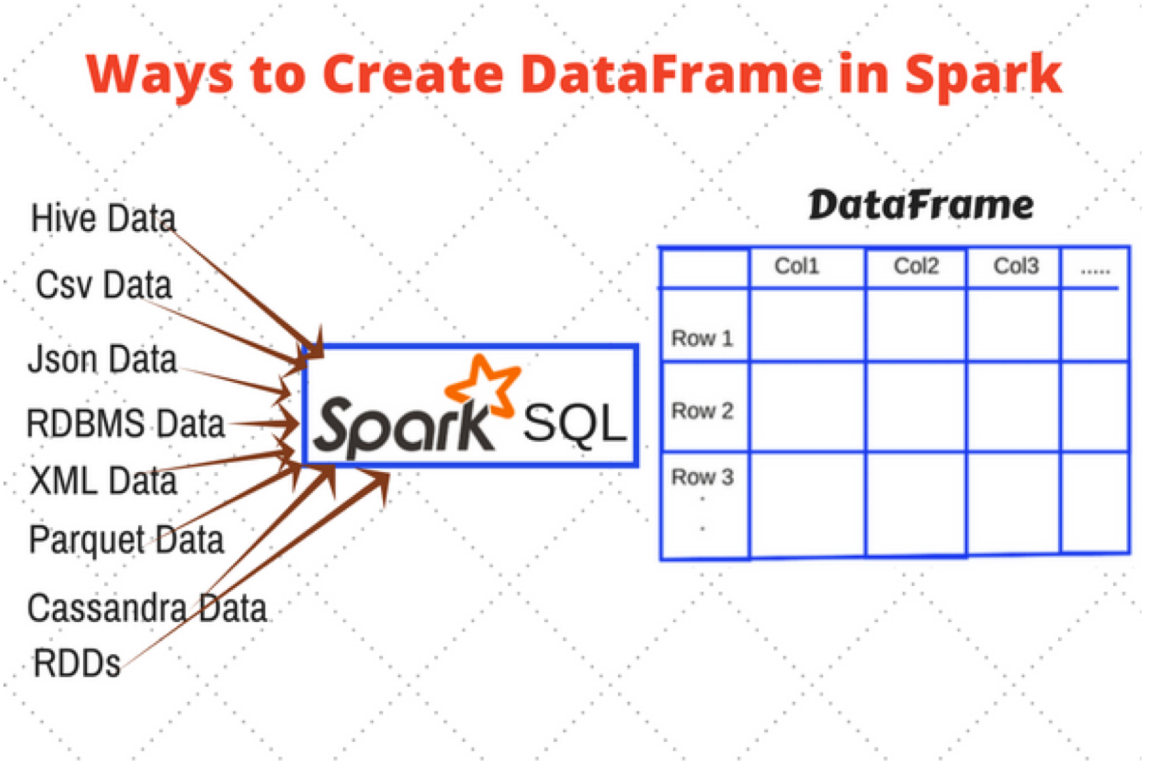
Spark Application (4)



Holden Karau, Rachel Warren, High Performance Spark, O'Reilly, 2017.

Spark Dataframes

- Table abstraction similar to relational database table or an Excel sheet with column headers.
- A DataFrame is a distributed collection of rows under named columns.
- Shares characteristics with RDDs:
  - Immutable
  - Lazy Evaluations
  - Distributed



More in SQL Section!

https://github.com/awantik/spark/wiki/DataFrame-on-PySpark

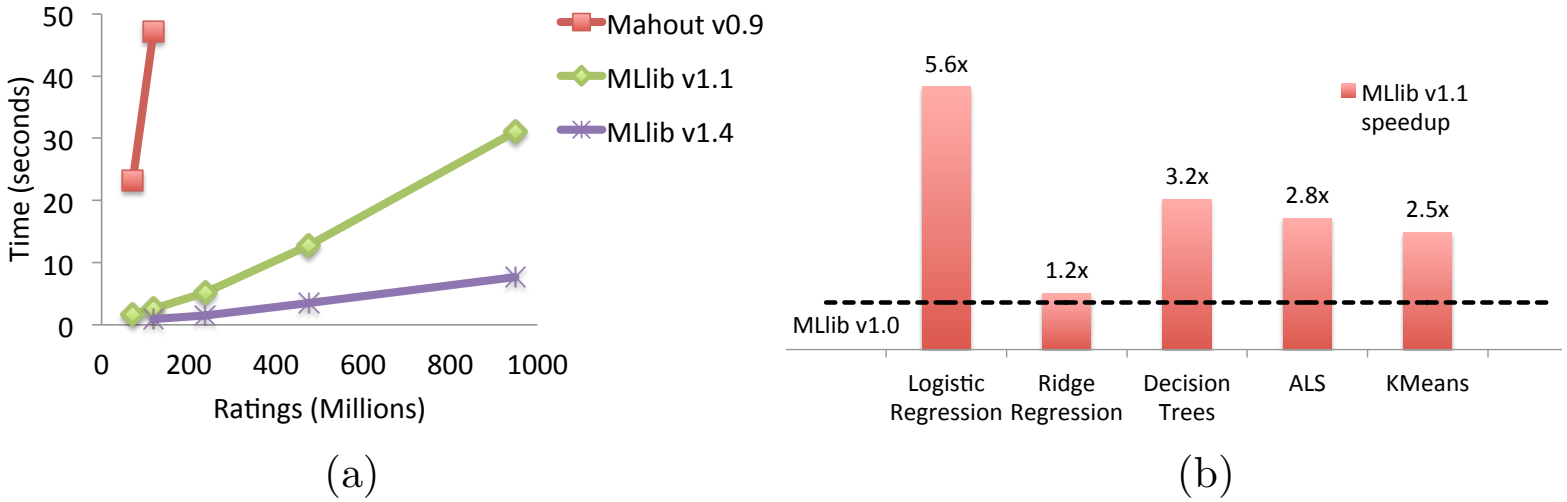


Figure 2: (a) Benchmarking results for ALS. (b) MLlib speedup between versions.

Further Frameworks: Flink

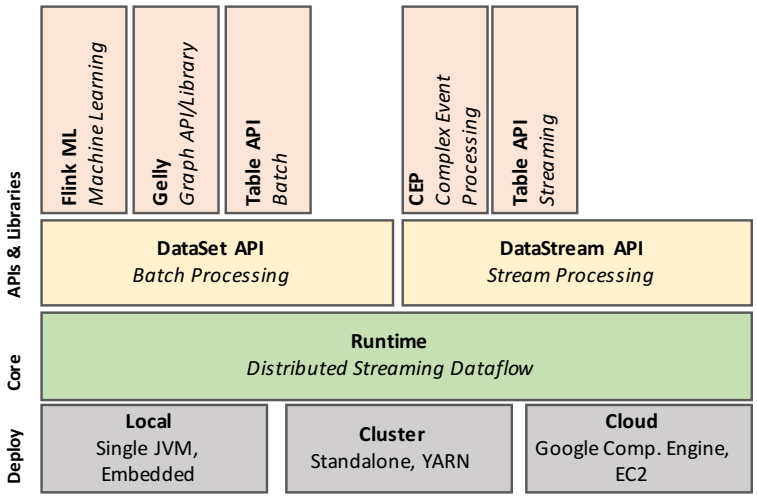


Figure 1: The Flink software stack.

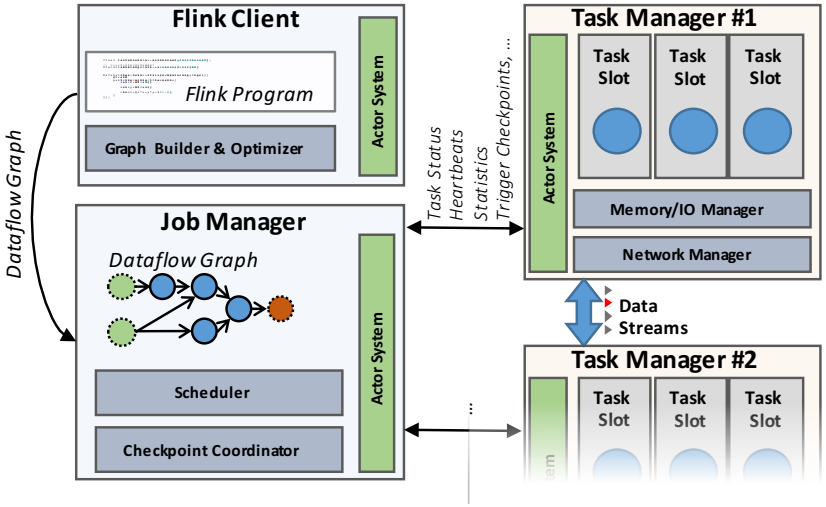
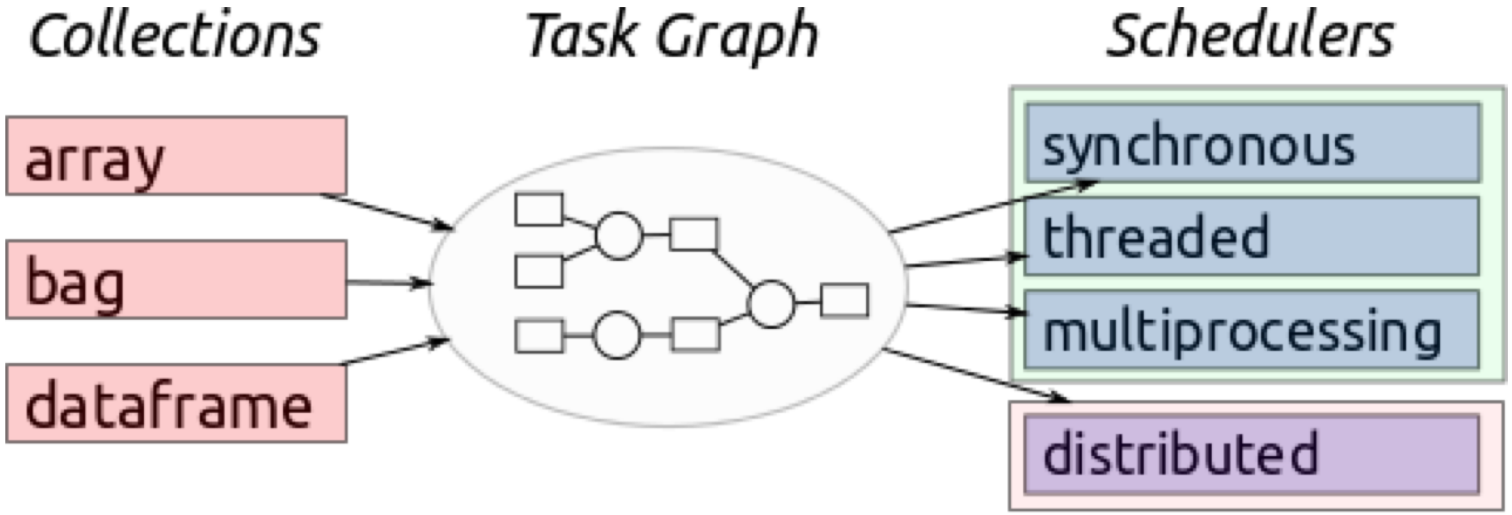


Figure 2: The Flink process model.

Flink: Stateful computations over streams real-time and historic fast, scalable, fault tolerant, in-memory, event time, large state, exactly-once

Further Frameworks: Dask



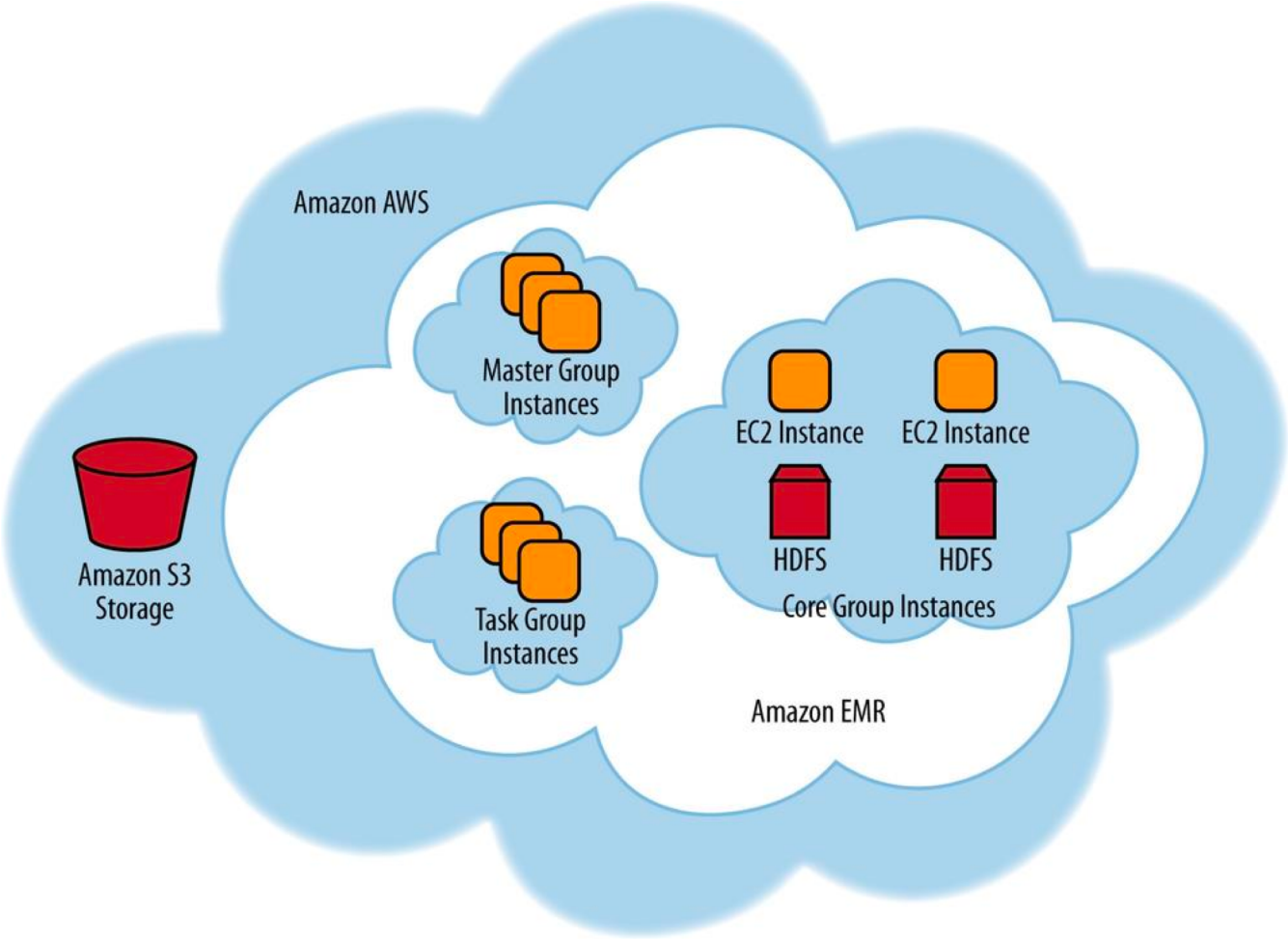
1.Dynamic task scheduling optimized for computation. This is similar to *Airflow, Luigi, Celery, or Make*, but optimized for interactive computational workloads.
2.“Big Data” collections like parallel arrays, dataframes, and lists that extend common interfaces like *NumPy, Pandas, or Python iterators* to larger-than-memory or distributed environments. These parallel collections run on top of the dynamic task schedulers.

https://dask.pydata.org/en/latest/

- Dean et al., MapReduce: Simplified Data Processing in Large Clusters, 2004
- Zaharia et al., Spark: Cluster Computing with Working Sets, Hotcloud 2010.
- Zaharia et al., Resilient Distributed Datasets: A Fault-Tolerant Abstraction forIn-Memory Cluster Computing, https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf, NSDI, 2012
- Carbone et al., Apache Flink: Stream and Batch Processing in a Single Engine, IEEE Engineering Bulletins, 2015
- Rocklin, Dask: Parallel Computation with Blocked algorithms and Task Scheduling, Proceedings of the 14th Python in Science Conference, 2015

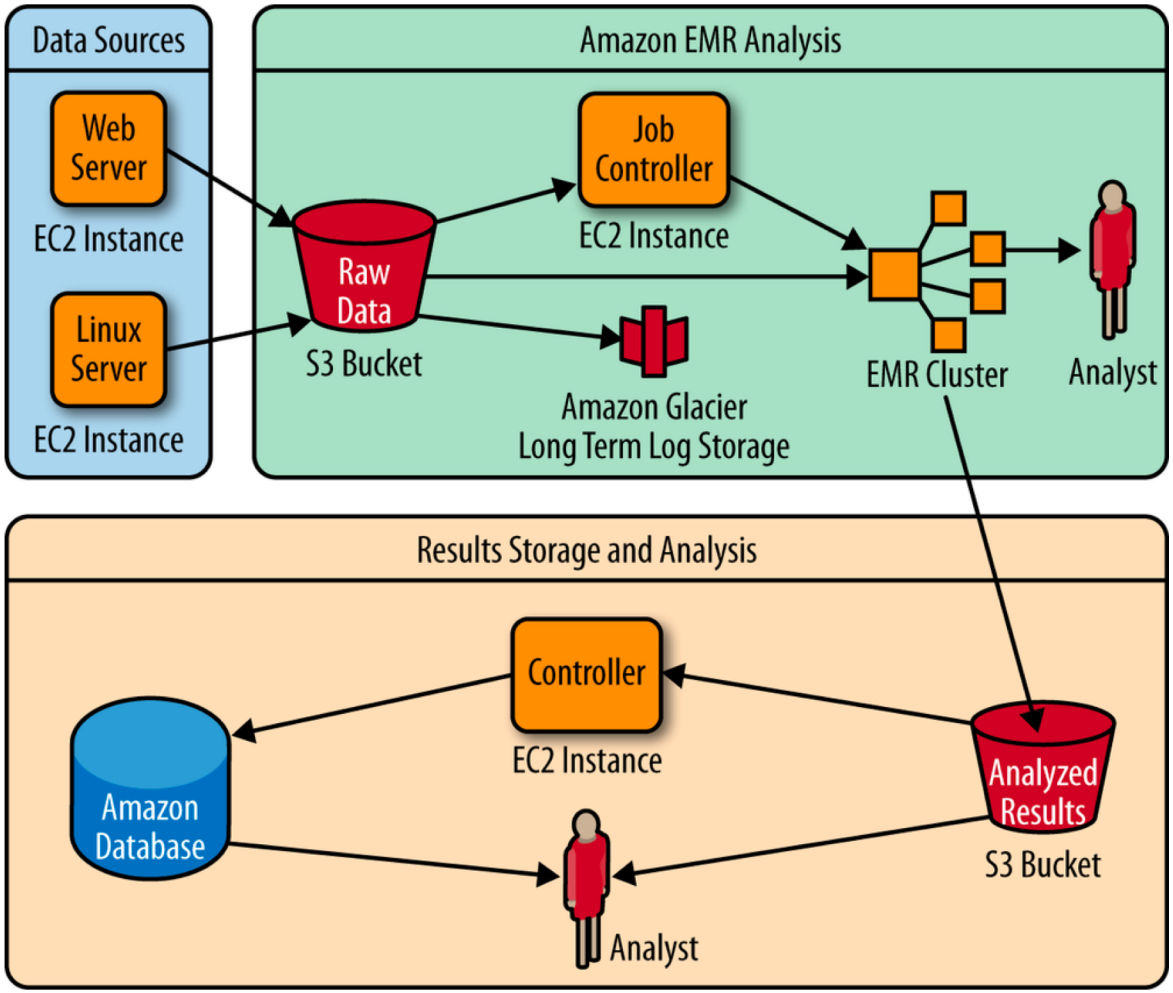
- Hadoop can be installed on most cloud services on their cores infrastructure services:
  - AWS EC2
  - Azure VMs
  - Google Compute Engines
- Hadoop Filesystem Adaptors for various object stores exists:
  - Simple Storage Service (S3)
  - Azure Blob Storage
  - Google Cloud Storage
- All three cloud vendors provided managed Haddop clusters:
  - Amazon Elastic MapReduce
  - Azure HDInsights
  - Google Cloud DataProc

Hadoop in the Cloud: Amazon Elastic MapReduce



Christopher Philips, Kevin Schmidt, Programming Elastic MapReduce, O'Reilly, 2013.

Hadoop in the Cloud: Amazon Elastic MapReduce



Christopher Philips, Kevin Schmidt, Programming Elastic MapReduce, O'Reilly, 2013.

Big Data on Different Infrastructure.

	On-Premise	HPC	Private	Public
Architecture	Dedicated Hardware	Dedicated Hardware managed by HPC scheduler (SLURM, PBS)	Manually setup VM clusters	Fully-managed VM Cluster (manual setup possible)
Maturity	High	Medium	Medium	Medium
Flexibility/ Extensibility/ Elasticity	Low	Low	Low	High (cluster can be dynamically expanded/ shrunk)
Performance	Optimal	Optimal	Good	Good
Security	High	High	High	Medium
Management Tools	Ambari, Cloudera Manager	saga-hadoop, jumpp	Apache Whirr, Cloudbreak	AWS Elastic MapReduce, Azure HDInsights