Utah Distributed Systems Meetup and Reading Group - Map Reduce and Spark

JT Olds

Space Monkey
Vivint R&D

January 19 2016

Outline

Outline

Map Reduce

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

Our implementation of MapReduce runs on a large cluster of commodity machines and is highly scalable:

given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a *map* operation to each logical "record" in our input in order to compute a set of intermediate key/value pairs, and then applying a *reduce* operation to all the values that shared the same key, in order to combine the derived data appropriately.

Map Reduce

Google Map Reduce context

- Lots of conceptually simple tasks
- On an internet's worth of data
- Spread across thousands of commodity servers
- That are constantly failing

Google Map Reduce context

- Lots of conceptually simple tasks
- On an internet's worth of data
- Spread across thousands of commodity servers
- That are constantly failing

Google Map Reduce context

- Lots of conceptually simple tasks
- On an internet's worth of data
- Spread across thousands of commodity servers
- That are constantly failing

Google Map Reduce context

- Lots of conceptually simple tasks
- On an internet's worth of data
- Spread across thousands of commodity servers
- That are constantly failing

Google Map Reduce context: abstraction?

- Parallelize the computation
- Distribute the data
- Handle failures
- With simple code

Google Map Reduce context: abstraction?

- Parallelize the computation
- Distribute the data
- Handle failures
- With simple code

Google Map Reduce context: abstraction?

- Parallelize the computation
- Distribute the data
- Handle failures
- With simple code

Google Map Reduce context: abstraction?

- Parallelize the computation
- Distribute the data
- Handle failures
- With simple code

Map Reduce

Google Map Reduce: Map

$$\texttt{map}\,(n_1, d_1) \rightarrow \;[(k_1, v_1)\,,(k_2, v_2)\,,...]$$

Google Map Reduce: Map

$$\texttt{map}\,(n_1, d_1) \rightarrow [(k_1, v_1), (k_2, v_2), \ldots]$$

$$\texttt{map}\,(n_2, d_2) \rightarrow [(k_3, v_3), (k_1, v_4), \ldots]$$

Google Map Reduce: Reduce

$$\texttt{reduce}\,(k_1, [v_1, v_4, \ldots]) \rightarrow r_1$$

Google Map Reduce: Reduce

$$\texttt{reduce}\,(k_1, [v_1, v_4, \ldots]) \rightarrow r_1$$

$$\texttt{reduce}\,(k_2, [v_2, \ldots]) \rightarrow r_2$$

Map Reduce

Example: Word Count

```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, "1");
```

Example: Word Count

```
reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
```

Aside: Combiner

```
combine(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  EmitIntermediate(key, AsString(result));
```

Example: Reverse Web Link Graph

```
map(String key, String value):
  // key: source url
  // value: document contents
  for each link target in value:
    EmitIntermediate(target, key);
```

Example: Reverse Web Link Graph

```
reduce(String key, Iterator values):
  // key: target url
  // values: a list of source urls
  Emit(values.serialize());
```

Example: Inverted index

```
map(String key, String value):
  // key: document id
  // value: document contents
  for each unique word in value:
    EmitIntermediate(word, key);
```

Example: Inverted index

```
reduce(String key, Iterator values):
  // key: word
  // values: list of document ids
  Emit(values.serialize());
```

Example: Grep

```
map(String key, String value):
  // key: document name
  // value: document contents
  for each line lineno, line in value:
    if PATTERN matches line:
      EmitIntermediate(
          "%s:%d" % (key, lineno), line);
```

Example: Grep

```
reduce(String key, Iterator values):
  // key: document/lineno pair
  // values: line contents
  Emit(values.first);
```

Example: Distributed sort

```
map(String key, String value):
  // key: key
  // value: record
  EmitIntermediate(key, value);
```

Example: Distributed sort

```
reduce(String key, Iterator values):
  // key: key
  // values: list of records
  for each record in records:
    Emit(record);
```

Map Reduce

1. Map Reduce
   - Context
   - Overall idea
   - Examples
   - Architecture
   - Challenges

Architecture

- Master with workers
- Master pings all workers to track liveness
- Master keeps track of map and reduce task state, restarting failed ones
- Map task output gets written to local disk. So, even completed tasks on failed workers need to be restarted.

Architecture

- Master with workers
- Master pings all workers to track liveness
- Master keeps track of map and reduce task state, restarting failed ones
- Map task output gets written to local disk. So, even completed tasks on failed workers need to be restarted.

Architecture

- Master with workers
- Master pings all workers to track liveness
- Master keeps track of map and reduce task state, restarting failed ones
- Map task output gets written to local disk. So, even completed tasks on failed workers need to be restarted.

Architecture

- Master with workers
- Master pings all workers to track liveness
- Master keeps track of map and reduce task state, restarting failed ones
- Map task output gets written to local disk. So, even completed tasks on failed workers need to be restarted.

Architecture

- Input comes from GFS (triplicate), master tries to schedule map worker on or near server with input replicate.
- User configures reduce partitioning, within a partition all keys are processed in sorted order
- Output often goes back to GFS (output is often the input to another job)
- Stragglers a real problem - some jobs are started multiple times.

Architecture

- Input comes from GFS (triplicate), master tries to schedule map worker on or near server with input replicate.
- User configures reduce partitioning, within a partition all keys are processed in sorted order
- Output often goes back to GFS (output is often the input to another job)
- Stragglers a real problem - some jobs are started multiple times.

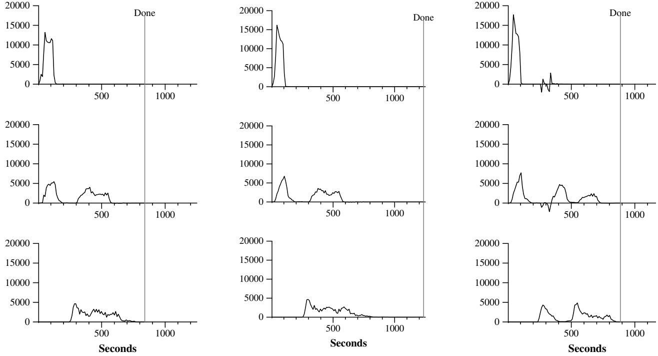
Architecture

- Input comes from GFS (triplicate), master tries to schedule map worker on or near server with input replicate.
- User configures reduce partitioning, within a partition all keys are processed in sorted order
- Output often goes back to GFS (output is often the input to another job)
- Stragglers a real problem - some jobs are started multiple times.

Architecture

- Input comes from GFS (triplicate), master tries to schedule map worker on or near server with input replicate.
- User configures reduce partitioning, within a partition all keys are processed in sorted order
- Output often goes back to GFS (output is often the input to another job)
- Stragglers a real problem - some jobs are started multiple times.

Stragglers



(a) Normal execution (b) No backup tasks (c) 200 tasks killed

Map Reduce

Challenges

- Must fit into map/reduce framework.
- Everything is written to disk, often to GFS, which means triplicate. Lots of disk I/O and network I/O.
- I/O exacerbated when output is the input to another job.

Challenges

- Must fit into map/reduce framework.
- Everything is written to disk, often to GFS, which means triplicate. Lots of disk I/O and network I/O.
- I/O exacerbated when output is the input to another job.

Challenges

- Must fit into map/reduce framework.
- Everything is written to disk, often to GFS, which means triplicate. Lots of disk I/O and network I/O.
- I/O exacerbated when output is the input to another job.

Outline

Spark

Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica
University of California, Berkeley

Abstract

We present Resilient Distributed Datasets (RDDs), a distributed memory abstraction that lets programmers perform in-memory computations on large clusters in a fault-tolerant manner. RDDs are motivated by two types of applications that current computing frameworks handle inefficiently: iterative algorithms and interactive data mining tools. In both cases, keeping data in memory can improve performance by an order of magnitude. To achieve fault tolerance efficiently, RDDs provide a restricted form of shared memory, based on coarse-grained transformations rather than fine-grained updates to shared state. However, we show that RDDs are expressive enough to capture a wide class of computations, including recent specialized programming models for iterative jobs, such as Pregel, and new applications that these models do not capture. We have implemented RDDs in a system called Spark, which we evaluate through a variety of user applications and benchmarks.

1 Introduction

Cluster computing frameworks like MapReduce [10] and Dryad [19] have been widely adopted for large-scale data analytics. These systems let users write parallel computations using a set of high-level operators, without having

tion, which can dominate application execution times.

Recognizing this problem, researchers have developed specialized frameworks for some applications that require data reuse. For example, Pregel [22] is a system for iterative graph computations that keeps intermediate data in memory, while HaLoop [7] offers an iterative MapReduce interface. However, these frameworks only support specific computation patterns (*e.g.,* looping a series of MapReduce steps), and perform data sharing implicitly for these patterns. They do not provide abstractions for more general reuse, *e.g.,* to let a user load several datasets into memory and run ad-hoc queries across them.

In this paper, we propose a new abstraction called *resilient distributed datasets (RDDs)* that enables efficient data reuse in a broad range of applications. RDDs are fault-tolerant, parallel data structures that let users explicitly persist intermediate results in memory, control their partitioning to optimize data placement, and manipulate them using a rich set of operators.

The main challenge in designing RDDs is defining a programming interface that can provide fault tolerance *efficiently*. Existing abstractions for in-memory storage on clusters, such as distributed shared memory [24], key-value stores [25], databases, and Piccolo [27], offer an interface based on fine-grained updates to mutable state (*e.g.,* cells in a table). With this interface, the only ways

Spark

Resilient Distributed Datasets

Existing frameworks are a poor fit for:

- Iterative algorithms
- Interactive data mining

Resilient Distributed Datasets

Existing frameworks are a poor fit for:

- Iterative algorithms
- Interactive data mining

Resilient Distributed Datasets

Both things can be sped up orders of magnitude by keeping stuff in memory!

Spark

Spark: Overall idea

- RDDs are implemented as lightweight objects inside of a Scala shell, where each object represents a sequence of deterministic transformations on some data.
- RDDs can only be constructed in the Scala shell by referencing some files on disk (usually HDFS), or by operations on other RDDs.

Spark: Overall idea

- RDDs are implemented as lightweight objects inside of a Scala shell, where each object represents a sequence of deterministic transformations on some data.
- RDDs can only be constructed in the Scala shell by referencing some files on disk (usually HDFS), or by operations on other RDDs.

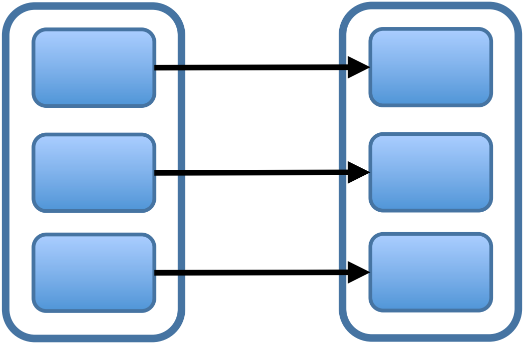
Spark: Overall idea (Example)

```
val lines = sc.textFile("hdfs://path/to/log")
val errors = lines.filter(_.startsWith("ERROR"))
val timestamps = (errors
      .filter(_.contains("HDFS"))
      .map(_.split("\t")(3)))
```

Spark: Overall idea

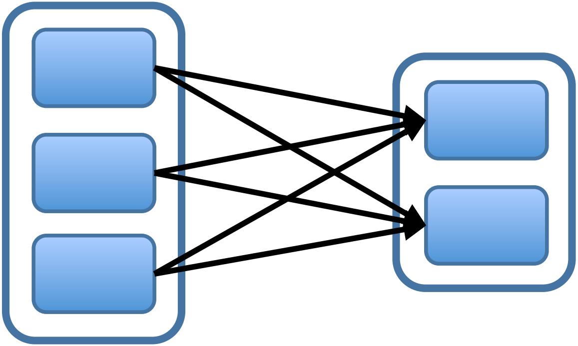
- An RDD is a read-only, partitioned collection of records.

Spark: Overall idea (Map)



Map (`flatMap` in Spark)

Spark: Overall idea (Reduce)



Reduce (`reduceByKey` in Spark)

Spark: Overall idea (Transformations)

- RDD operations are coarse-grained and high level, like `map`, `sample`, `filter`, `reduceByKey`, etc.
- RDDs are evaluated lazily. Data is processed as late as possible. The RDD simply records the high level operation order, dependencies, and data partitions.

Spark: Overall idea (Transformations)

- RDD operations are coarse-grained and high level, like `map`, `sample`, `filter`, `reduceByKey`, etc.
- RDDs are evaluated lazily. Data is processed as late as possible. The RDD simply records the high level operation order, dependencies, and data partitions.

Spark: Overall idea (Transformations)

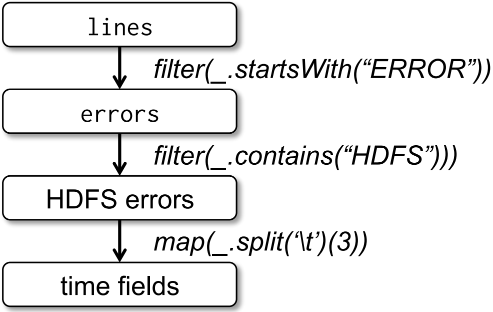


Figure 1: Lineage graph for the third query in our example. Boxes represent RDDs and arrows represent transformations.

Spark: Overall idea

Transformations	$map(f : \mathrm{T} \Rightarrow \mathrm{U})$	:	RDD[T] $\Rightarrow$ RDD[U]
	$filter(f : \mathrm{T} \Rightarrow \mathrm{Bool})$	:	RDD[T] $\Rightarrow$ RDD[T]
	$flatMap(f : \mathrm{T} \Rightarrow \mathrm{Seq[U]})$	:	RDD[T] $\Rightarrow$ RDD[U]
	$sample(fraction : \mathrm{Float})$	:	RDD[T] $\Rightarrow$ RDD[T] (Deterministic sampling)
	$groupByKey()$	:	RDD[(K, V)] $\Rightarrow$ RDD[(K, Seq[V])]
	$reduceByKey(f : (\mathrm{V}, \mathrm{V}) \Rightarrow \mathrm{V})$	:	RDD[(K, V)] $\Rightarrow$ RDD[(K, V)]
	$union()$	:	(RDD[T], RDD[T]) $\Rightarrow$ RDD[T]
	$join()$	:	(RDD[(K, V)], RDD[(K, W)]) $\Rightarrow$ RDD[(K, (V, W))]
	$cogroup()$	:	(RDD[(K, V)], RDD[(K, W)]) $\Rightarrow$ RDD[(K, (Seq[V], Seq[W]))]
	$crossProduct()$	:	(RDD[T], RDD[U]) $\Rightarrow$ RDD[(T, U)]
	$mapValues(f : \mathrm{V} \Rightarrow \mathrm{W})$	:	RDD[(K, V)] $\Rightarrow$ RDD[(K, W)] (Preserves partitioning)
	$sort(c : \mathrm{Comparator[K]})$	:	RDD[(K, V)] $\Rightarrow$ RDD[(K, V)]
	$partitionBy(p : \mathrm{Partitioner[K]})$	:	RDD[(K, V)] $\Rightarrow$ RDD[(K, V)]
Actions	$count()$	:	RDD[T] $\Rightarrow$ Long
	$collect()$	:	RDD[T] $\Rightarrow$ Seq[T]
	$reduce(f : (\mathrm{T}, \mathrm{T}) \Rightarrow \mathrm{T})$	:	RDD[T] $\Rightarrow$ T
	$lookup(k : \mathrm{K})$	:	RDD[(K, V)] $\Rightarrow$ 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.

Spark: Overall idea (Tuning)

- A user can indicate which RDDs may get reused and should be persisted (usually in-memory, but can be spilled to disk via priority) (`persist` or `cache`).
- A user can also configure the partitioning of the data, and the algorithm through which nodes are chosen by key (helps join efficiency, etc).

Spark: Overall idea (Tuning)

- A user can indicate which RDDs may get reused and should be persisted (usually in-memory, but can be spilled to disk via priority) (`persist` or `cache`).
- A user can also configure the partitioning of the data, and the algorithm through which nodes are chosen by key (helps join efficiency, etc).

Spark: Overall idea (Actions)

- Once you have constructed an RDD with apporiate transformations, the user can perform an action.
- Actions materialize an RDD, fire up the job scheduler, and cause work to be performed.
- Actions include `count`, `collect`, `reduce`, `save`.

Spark: Overall idea (Actions)

- Once you have constructed an RDD with apporiate transformations, the user can perform an action.
- Actions materialize an RDD, fire up the job scheduler, and cause work to be performed.
- Actions include `count`, `collect`, `reduce`, `save`.

Spark: Overall idea (Actions)

- Once you have constructed an RDD with apporiate transformations, the user can perform an action.
- Actions materialize an RDD, fire up the job scheduler, and cause work to be performed.
- Actions include `count`, `collect`, `reduce`, `save`.

Spark: Overall idea

Transformations	$map(f : \mathrm{T} \Rightarrow \mathrm{U})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[U]}$
	$filter(f : \mathrm{T} \Rightarrow \mathrm{Bool})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[T]}$
	$flatMap(f : \mathrm{T} \Rightarrow \mathrm{Seq[U]})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[U]}$
	$sample(fraction : \mathrm{Float})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[T]}$ (Deterministic sampling)
	$groupByKey()$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, Seq[V])]}$
	$reduceByKey(f : (\mathrm{V},\mathrm{V}) \Rightarrow \mathrm{V})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, V)]}$
	$union()$	:	$(\mathrm{RDD[T]},\mathrm{RDD[T]}) \Rightarrow \mathrm{RDD[T]}$
	$join()$	:	$(\mathrm{RDD[(K, V)]},\mathrm{RDD[(K, W)]}) \Rightarrow \mathrm{RDD[(K, (V, W))]}$
	$cogroup()$	:	$(\mathrm{RDD[(K, V)]},\mathrm{RDD[(K, W)]}) \Rightarrow \mathrm{RDD[(K, (Seq[V], Seq[W]))]}$
	$crossProduct()$	:	$(\mathrm{RDD[T]},\mathrm{RDD[U]}) \Rightarrow \mathrm{RDD[(T, U)]}$
	$mapValues(f : \mathrm{V} \Rightarrow \mathrm{W})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, W)]}$ (Preserves partitioning)
	$sort(c : \mathrm{Comparator[K]})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, V)]}$
	$partitionBy(p : \mathrm{Partitioner[K]})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, V)]}$
Actions	$count()$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{Long}$
	$collect()$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{Seq[T]}$
	$reduce(f : (\mathrm{T},\mathrm{T}) \Rightarrow \mathrm{T})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{T}$
	$lookup(k : \mathrm{K})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{Seq[V]}$ (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.

Spark: Overall idea

- RDDs are immutable and deterministic.
- Easy to launch backup workers for stragglers (clear win from Map Reduce)
- Easy to relaunch computations from failed nodes.
- Computations are computed lazily, which allows for rich optimizations for locality/partitioning by a job scheduler and query planner.

Spark: Overall idea

- RDDs are immutable and deterministic.
- Easy to launch backup workers for stragglers (clear win from Map Reduce)
- Easy to relaunch computations from failed nodes.
- Computations are computed lazily, which allows for rich optimizations for locality/partitioning by a job scheduler and query planner.

Spark: Overall idea

- RDDs are immutable and deterministic.
- Easy to launch backup workers for stragglers (clear win from Map Reduce)
- Easy to relaunch computations from failed nodes.
- Computations are computed lazily, which allows for rich optimizations for locality/partitioning by a job scheduler and query planner.

Spark: Overall idea

- RDDs are immutable and deterministic.
- Easy to launch backup workers for stragglers (clear win from Map Reduce)
- Easy to relaunch computations from failed nodes.
- Computations are computed lazily, which allows for rich optimizations for locality/partitioning by a job scheduler and query planner.

Spark

Example: Log querying

```
val lines = sc.textFile("hdfs://path/to/log")
val errors = lines.filter(_.startsWith("ERROR"))
errors.cache()
```

Example: Log querying

```
errors.count()
```

Example: Log querying

```
errors.filter(_.contains("MySQL")).count()
```

Example: Log querying

```
(errors.filter(_.contains("HDFS"))
        .map(_.split("\t")(3))
        .collect())
```

Example: Log querying

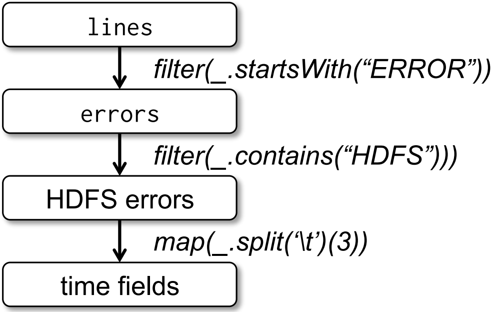


Figure 1: Lineage graph for the third query in our example. Boxes represent RDDs and arrows represent transformations.

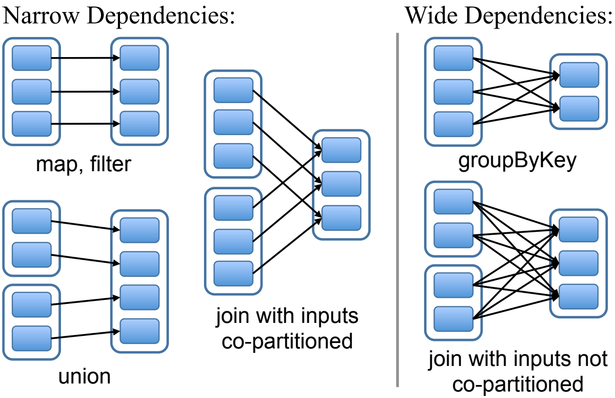
Spark

RDD Representation

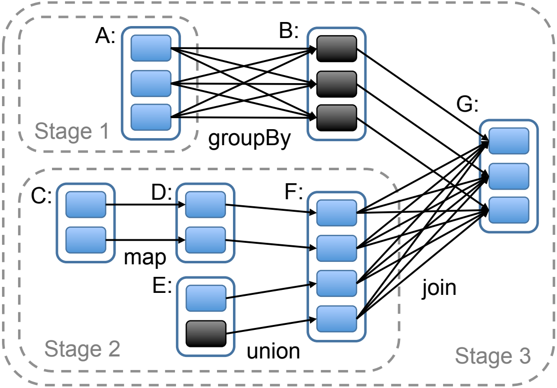
Operation	Meaning
partitions()	Return a list of Partition objects
preferredLocations(*p*)	List nodes where partition *p* can be accessed faster due to data locality
dependencies()	Return a list of dependencies
iterator(*p*, *parentIters*)	Compute the elements of partition *p* given iterators for its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

Table 3: Interface used to represent RDDs in Spark.

Narrow vs Wide Dependencies



Job Scheduling



Spark

Example: Word Count

```
val m = documents.flatMap(
    _._2.split("\\s+")
        .map(word => (word, "1")))
val r = m.reduceByKey(
    (a, b) => (a.toInt + b.toInt).toString)

// do we need a combine step?
```

Example: Reverse Index

```
val m = documents.flatMap(
    doc =>
        (doc._2.split("\\s+")
               .distinct
               .map(word => (word, doc._1))))
val r = m.reduceByKey(
    (a, b) => a + "\n" + b)
```

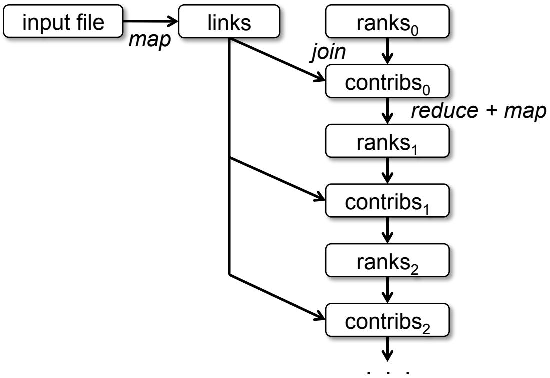
Example: Grep

```
def search(keyword: String) = {
  (documents
    .filter(_._2.contains(keyword))
    .map(_._1)
    .reduce((a, b) => (a + "\n" + b)))
}
```

Example: Page Rank

```
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))
}
```

Example: Page Rank



Outline

Conclusion?

3 Conclusion?

Conclusion?

- Can anything really be dead?
- Is Map Reduce dead?
- Is Spark the last word?
- Questions?

Conclusion?

- Can anything really be dead?
- Is Map Reduce dead?
- Is Spark the last word?
- Questions?

Conclusion?

- Can anything really be dead?
- Is Map Reduce dead?
- Is Spark the last word?
- Questions?

Conclusion?

- Can anything really be dead?
- Is Map Reduce dead?
- Is Spark the last word?
- Questions?

Space Monkey!

- Large scale storage
- Distributed system design and implementation
- Security and cryptography engineering
- Erasure codes
- Monitoring and sooo much data

Space Monkey!

Come work with us!