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 allows programmers to perform in-memory computations on large clusters while retaining the fault tolerance of data flow models like MapReduce. RDDs are motivated by two types of applications that current data flow systems handle inefficiently: iterative algorithms, which are common in graph applications and machine learning, 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 highly restricted form of shared memory: they are read-only datasets that can only be constructed through bulk operations on other RDDs. However, we show that RDDs are expressive enough to capture a wide class of computations, including MapReduce and specialized programming models for iterative jobs such as Pregel. Our implementation of RDDs can outperform Hadoop by $20\times$ for iterative jobs and can be used interactively to search a 1 TB dataset with latencies of 5–7 seconds.

1 Introduction

High-level cluster programming models like MapReduce [12] and Dryad [18] have been widely adopted to process the growing volumes of data in industry and the sciences [4]. These systems simplify distributed programming by automatically providing locality-aware scheduling, fault tolerance, and load balancing, enabling a wide range of users to analyze big datasets on commodity clusters.

Most current cluster computing systems are based on an acyclic data flow model, where records are loaded from stable storage (*e.g.,* a distributed file system), passed through a DAG of deterministic operators, and written back out to stable storage. Knowledge of the data flow graph allows the runtime to automatically schedule work and to recover from failures.

While acyclic data flow is a powerful abstraction, there are applications that cannot be expressed efficiently using only this construct. Our work focuses on one broad class of applications not well served by the acyclic model: those that reuse a *working set* of data in multiple parallel operations. This class includes iterative algorithms commonly used in machine learning and graph applications, which apply a similar function to the data on each step, and interactive data mining tools, where a user repeatedly queries a subset of the data. Because data flow based frameworks do not explicitly provide support for working sets, these applications have to output data to disk and reload it on each query with current systems, leading to significant overhead.

We propose a distributed memory abstraction called *resilient distributed datasets (RDDs)* that supports applications with working sets while retaining the attractive properties of data flow models: automatic fault tolerance, locality-aware scheduling, and scalability. RDDs allow users to explicitly cache working sets in memory across queries, leading to substantial speedups on future reuse.

RDDs provide a highly restricted form of shared memory: they are read-only, partitioned collections of records that can only be created through deterministic transformations (*e.g.,* map, join and group-by) on other RDDs. These restrictions, however, allow for low-overhead fault tolerance. In contrast to distributed shared memory systems [24], which require costly checkpointing and rollback, RDDs reconstruct lost partitions through *lineage*: an RDD has enough information about how it was derived from other RDDs to rebuild just the missing partition, without having to checkpoint any data.[1] Although RDDs are not a general shared memory abstraction, they represent a sweet-spot between expressivity, scalability and reliability, and we have found them well-suited for a wide range of data-parallel applications.

Our work is not the first to note the limitations of acyclic data flow. For example, Google's Pregel [21] is a specialized programming model for iterative graph algorithms, while Twister [13] and HaLoop [8] provide an iterative MapReduce model. However, these systems offer restricted communication patterns for specific classes of applications. In contrast, RDDs are a more general abstraction for applications with working sets. They allow users to explicitly name and materialize intermediate

[1] Checkpointing may be useful when a lineage chain grows large, however, and we discuss when and how to perform it in Section 5.3.

results, control their partitioning, and use them in operations of their choice (as opposed to giving the runtime a set of MapReduce steps to loop). We show that RDDs can be used to express both Pregel, iterative MapReduce, and applications that neither of these models capture well, such as interactive data mining tools (where a user loads a dataset into RAM and runs ad-hoc queries).

We have implemented RDDs in a system called Spark, which is being used to develop a variety of parallel applications at our organization. Spark provides a language-integrated programming interface similar to DryadLINQ [34] in the Scala programming language [5], making it easy for users to write parallel jobs. In addition, Spark can be used interactively to query big datasets from a modified version of the Scala interpreter. We believe that Spark is the first system to allow an efficient, general-purpose programming language to be used interactively to analyze large datasets on clusters.

We evaluate RDDs through both microbenchmarks and measurements of user applications. We show that Spark outperforms Hadoop by up to $20\times$ for iterative applications, improves the performance of a data analytics report by $40\times$, and can be used interactively to scan a 1 TB dataset with latencies of 5–7s. In addition, we have implemented the Pregel and HaLoop programming models on top of Spark, including the placement optimizations they employ, as relatively small libraries (100 and 200 lines of Scala respectively). Finally, we have taken advantage of the deterministic nature of RDDs to build `rddbg`, a debugging tool for Spark that lets users rebuild any RDD created during a job using its lineage and rerun tasks on it in a conventional debugger.

We begin this paper by introducing RDDs in Section 2. Section 3 then presents the API of Spark. Section 4 shows how RDDs can express several parallel applications, including Pregel and HaLoop. Section 5 discusses the representation of RDDs in Spark and our job scheduler. Section 6 describes our implementation and `rddbg`. We evaluate RDDs in Section 7. We survey related work in Section 8 and conclude in Section 9.

2 Resilient Distributed Datasets (RDDs)

This section describes RDDs and our programming model. We start by discussing our goals, from which we motivate our design (§2.1). We then define RDDs (§2.2), discuss their programming model in Spark (§2.3), and show an example (§2.4). We finish by comparing RDDs with distributed shared memory in §2.5.

2.1 Goals and Overview

Our goal is to provide an abstraction that supports applications with working sets (*i.e.,* applications that reuse an intermediate result in multiple parallel operations) while preserving the attractive properties of MapReduce and related models: automatic fault tolerance, locality-aware scheduling, and scalability. RDDs should be as easy to program against as data flow models, but capable of efficiently expressing computations with working sets.

Out of our desired properties, the most difficult one to support efficiently is fault tolerance. In general, there are two options to make a distributed dataset fault-tolerant: checkpointing the data or logging the updates made to it. In our target environment (large-scale data analytics), checkpointing the data is expensive: it would require replicating big datasets across machines over the datacenter network, which typically has much lower bandwidth than the memory bandwidth within a machine [16], and it would also consume additional storage (replicating data in RAM would reduce the total amount that can be cached, while logging it to disk would slow down applications). Consequently, we choose to log updates. However, logging updates is also expensive if there are many of them. Consequently, RDDs only support *coarse-grained* transformations, where we can log a single operation to be applied to many records. We then remember the series of transformations used to build an RDD (*i.e.,* its lineage) and use it to recover lost partitions.

While supporting only coarse-grained transformations restricts the programming model, we have found RDDs suitable for a wide range of applications. In particular, RDDs are well-suited for data-parallel batch analytics applications, including data mining, machine learning, and graph algorithms, because these programs naturally perform the same operation on many records. RDDs would be less suitable for applications that asynchronously update shared state, such as a parallel web crawler. However, our goal is to provide an efficient programming model for a large array of analytics applications, and leave other applications to specialized systems.

2.2 RDD Abstraction

Formally, an RDD is a read-only, partitioned collection of records. RDDs can be only created through *deterministic* operations on either (1) a dataset in stable storage or (2) other existing RDDs. We call these operations *transformations* to differentiate them from other operations that programmers may apply on RDDs. Examples of transformations include *map*, *filter*, *groupBy* and *join*.

RDDs do not need to be materialized at all times. Instead, an RDD has enough information about how it was derived from other datasets (*i.e.,* its *lineage*) to *compute* its partitions from data in stable storage.

2.3 Programming Model

In Spark, RDDs are represented by objects, and transformations are invoked using methods on these objects.

After defining one or more RDDs, programmers can use them in *actions*, which are operations that return a

value to the application or export data to a storage system. Examples of actions include *count* (which returns the number of elements in the RDD), *collect* (which returns the elements themselves), and *save* (which outputs the RDD to a storage system). In Spark, RDDs are only computed the first time they are used in an action (*i.e.,* they are lazily evaluated), allowing the runtime to pipeline several transformations when building an RDD.

Programmers can also control two other aspects of RDDs: *caching* and *partitioning*. A user may ask for an RDD to be cached, in which case the runtime will store partitions of the RDD that it has computed to speed up future reuse. Cached RDDs are typically stored in memory, but they spill to disk if there is not enough memory.[2]

Lastly, RDDs optionally allow users to specify a partitioning order based on a key associated with each record. We currently support hash and range partitioning. For example, an application may request that two RDDs be hash-partitioned in the same way (placing records with the same keys on the same machine) to speed up joins between them. Consistent partition placement across iterations is one of the main optimizations in Pregel and HaLoop, so we let users express this optimization.

2.4 Example: Console Log Mining

We illustrate RDDs through an example use case. Suppose that a large website is experiencing errors and an operator wants to search terabytes of logs in the Hadoop filesystem (HDFS) to find the cause. Using Spark, our implementation of RDDs, the operator can load just the error messages from the logs into RAM across a set of nodes and query them interactively. She would first type the following Scala code at the Spark interpreter:[3]

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

Line 1 defines an RDD backed by an HDFS file (as a collection of lines of text), while line 2 derives a filtered RDD from it. Line 3 asks for `errors` to be cached. Note that the argument to *filter* is Scala syntax for a closure.

At this point, no work has been performed on the cluster. However, the user can now use the RDD in actions, *e.g.,* to count the number of messages:

```
errors.count()
```

The user can also perform further transformations on the RDD and use their results, as in the following lines:

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

[2]Users can also request other behaviors, like caching *only* on disk.

[3]Spark's language-integrated API is similar to DryadLINQ [34].

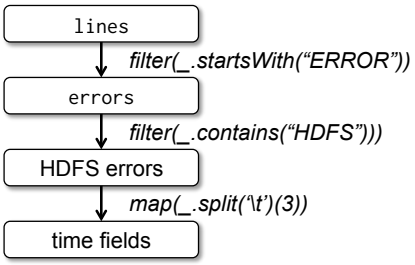


Figure 1: Lineage graph for the third query in our example. Boxes represent RDDs and arrows represent transformations.

```
// 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()
```

After the first action involving `errors` runs, Spark will cache the partitions of `errors` in memory, greatly speeding up subsequent computations on it. Note that the base RDD, `lines`, is never cached. This is desirable because the error messages might only be a small fraction of the data (small enough to fit into memory).

Finally, to illustrate how our model achieves fault tolerance, we show the lineage graph for the RDDs in our third query in Figure 1. In this query, we started with `errors`, the result of a filter on `lines`, and applied a further filter and map before running a *collect*. The Spark scheduler will pipeline the latter two transformations and send a set of tasks to compute them to the nodes holding the cached partitions of `errors`. In addition, if a partition of `errors` is lost, Spark rebuilds it by applying a filter on only the corresponding partition of `lines`.

2.5 RDDs vs. Distributed Shared Memory

To further understand the capabilities of RDDs as a distributed memory abstraction, we compare them against distributed shared memory (DSM) [24] in Table 1. In DSM systems, applications read and write to arbitrary locations in a global address space. (Note that under this definition, we include not only traditional shared memory systems, but also systems where an application might share data through a distributed hash table or filesystem, like Piccolo [28].) DSM is a very general abstraction, but this generality makes it harder to implement in an efficient and fault-tolerant manner on commodity clusters.

The main difference between RDDs and DSM is that RDDs can only be created ("written") through bulk transformations, while DSM allows reads and writes to each memory location. This restricts RDDs to applications that perform bulk writes, but allows for more efficient fault tolerance. In particular, RDDs do not need to incur the overhead of checkpointing, as they can be recovered

Aspect	RDDs	Distr. Shared Mem.
Reads	Bulk or fine-grained	Fine-grained
Writes	Bulk transformations	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.

using lineage.[4] Furthermore, only the lost partitions of an RDD need to be recomputed upon failure, and they can be recomputed in parallel on different nodes, without having to roll back the whole program.

One interesting observation is that RDDs also let a system tolerate slow nodes (stragglers) by running backup copies of tasks, as in MapReduce [12]. Backup tasks would be hard to implement with DSM as both copies of a task would read/write to the same memory addresses.

The RDD model also provides two other benefits over DSM. First, in bulk operations on RDDs, a runtime can schedule tasks based on data locality to improve performance. Second, cached RDDs degrade gracefully when there is not enough memory to store them, as long as they are only being used in scan-based operations. Partitions that do not fit in RAM can be stored on disk and will provide similar performance to current data flow systems.

One final point of comparison is the granularity of reads. Many of our actions on RDDs (*e.g., count* and *collect*) perform bulk reads that scan the whole dataset, which also lets us schedule them close to the data. However, RDDs can be used for fine-grained reads as well, through key lookups on hash or range partitioned RDDs.

3 Spark Programming Interface

Spark provides the RDD abstraction through a language-integrated API in Scala [5]. Scala is a statically typed functional and object-oriented language for the Java VM. We chose to use Scala due to its combination of conciseness (which is especially useful for interactive use) and efficiency (due to static typing). However, nothing about the RDD abstraction requires a functional language; it would also be possible to provide RDDs in other languages by using classes to represent the user's functions, as is done in Hadoop [2].

To use Spark, developers write a *driver program* that

[4]In some applications, it can still help to checkpoint RDDs with long lineage chains, as we discuss in Section 5.3.

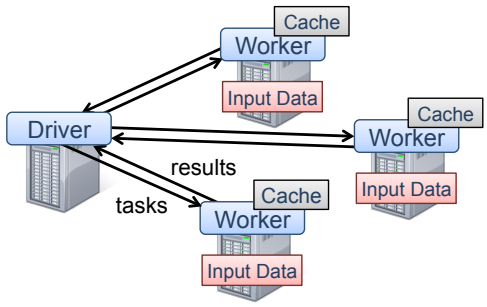


Figure 2: Spark runtime. A user's driver program launches multiple workers, which read data blocks from a distributed file system and can cache computed RDD partitions in memory.

connects to a cluster to run *workers*, as shown in Figure 2. The driver defines one or more RDDs and invokes actions on them. The workers are long-lived processes that can cache RDD partitions in RAM as Java objects.

As can be seen from the example in Section 2.4, users provide arguments to RDD operations like *map* by passing closures (function literals). Scala represents each closure as a Java object, and these objects can be serialized and loaded on another node to pass the closure across the network. Scala also saves any variables bound in the closure as fields in the Java object. For example, one can write code like `var x = 5; rdd.map(_ + x)` to add 5 to each element of an RDD.[5] Overall, Spark's language integration is similar to DryadLINQ [34].

RDDs themselves are statically typed objects parametrized by an element type. For example, RDD[Int] is an RDD of integers. However, most of our examples omit types since Scala supports type inference.

Although our method of exposing RDDs in Scala is conceptually simple, we had to work around issues with Scala's closure objects using reflection.[6] We also needed more work to make Spark usable from the Scala interpreter, as we shall discuss in Section 6. Nonetheless, we did *not* have to modify the Scala compiler.

3.1 RDD Operations in Spark

Table 2 lists the main RDD transformations and actions available in Spark. We give the signature of each operation, showing type parameters in square brackets. Recall that *transformations* are lazy operations that define a new RDD, while *actions* launch a computation to return a value to the program or write data to external storage.

Note that some operations, such as *join*, are only available on RDDs of key-value pairs. Also, our function names are chosen to match other APIs in Scala and other functional languages; for example, *map* is a one-to-one

[5]We save each closure at the time it is created, so that the *map* in this example will always add 5 even if `x` changes.

[6]Specifically, when a closure is nested inside another closure, it may indirectly reference variables from the outer closure that it does not need. We discuss the issue and our fix in an extended technical report.

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

Table 2: RDD transformations and actions available in Spark. Seq[T] denotes a sequence of elements of type T.

mapping, while *flatMap* maps each input value to one or more outputs (similar to the map in MapReduce).

In addition to these operators, users can ask for an RDD to be cached. Furthermore, users can get an RDD's partition order, which is represented by a Partitioner class, and partition another RDD according to it. Operations such as *groupByKey*, *reduceByKey* and *sort* automatically result in a hash or range partitioned RDD.

4 Example Applications

We now illustrate how RDDs can be used to express several data-parallel applications. We start by discussing iterative machine learning applications (§4.1), and then show how RDDs can also express several existing cluster programming models: MapReduce (§4.2), Pregel (§4.3), and HaLoop (§4.4). Finally, we discuss classes of applications that RDDs are *not* suitable for (§4.5).

4.1 Iterative Machine Learning

Many machine learning algorithms are iterative in nature because they run iterative optimization procedures, such as gradient descent, to optimize an objective function. These algorithms can be sped up substantially if their working set fits into RAM across a cluster. Furthermore, these algorithms often employ bulk operations like maps and sums, making them easy to express with RDDs.

As an example, the following program implements logistic regression [15], a common classification algorithm that searches for a hyperplane w that best separates two sets of points (*e.g.,* spam and non-spam emails). The algorithm uses gradient descent: it starts w at a random value, and on each iteration, it sums a function of w over the data to move w in a direction that improves it.

```
val points = spark.textFile(...)
                  .map(parsePoint).cache()
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
}
```

We start by defining a cached RDD called `points` as the result of a *map* transformation on a text file that parses each line of text into a Point object. We then repeatedly run *map* and *reduce* on `points` to compute the gradient at each step by summing a function of the current w. In Section 7.1, we show that caching `points` in memory in this manner yields significant speedups over loading the data file from disk and parsing it on each step.

Other examples of iterative machine learning algorithms that users have implemented with Spark include k-means, which runs one map/reduce pair per iteration like logistic regression; expectation maximization (EM), which alternates between two different map/reduce steps; and alternating least squares matrix factorization, a collaborative filtering algorithm. Chu et al. have shown that iterated MapReduce can also be used to implement other common learning algorithms [11].

4.2 MapReduce using RDDs

The MapReduce model [12] can readily be expressed using RDDs. Given an input dataset with elements of type T and functions *myMap* : $T \Rightarrow \text{List}[(K_i, V_i)]$ and *myReduce* : $(K_i, \text{List}[V_i]) \Rightarrow \text{List}[R]$, one can write:

```
data.flatMap(myMap)
    .groupByKey()
    .map((k, vs) => myReduce(k, vs))
```

Alternatively, if the job has a combiner, one can write:

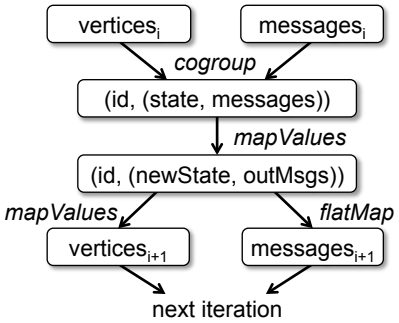


Figure 3: Data flow for one iteration of Pregel as implemented using RDDs. Boxes show RDDs, arrows show transformations.

```
data.flatMap(myMap)
    .reduceByKey(myCombiner)
    .map((k, v) => myReduce(k, v))
```

The *reduceByKey* operation performs partial aggregations on the mapper nodes, like MapReduce's combiners.

4.3 Pregel using RDDs

Pregel [21] is a programming model for graph applications based on the Bulk Synchronous Parallel paradigm [32]. Programs run as a series of coordinated iterations called *supersteps*. On each superstep, each vertex in the graph runs a user function that can update state associated with the vertex, mutate the graph topology, and send messages to other vertices for use in the *next* superstep. This model can express many graph algorithms, including shortest paths, bipartite matching, and PageRank.

As an example, we sketch a Pregel implementation of PageRank [7]. The program associates a current PageRank r as state with each vertex. On each superstep, each vertex sends a contribution of $\frac{r}{n}$ to each of its neighbors, where n is its number of neighbors. At the start of the next superstep, each vertex updates its rank to $\alpha/N + (1-\alpha)\sum c_i$, where the sum is over the contributions it received and N is the total number of vertices.

Pregel partitions the input graph across the workers and stores it in memory. At each superstep, the workers exchange messages through a MapReduce-like shuffle.

Pregel's communication pattern can be expressed using RDDs, as we show in Figure 3 and in the code below. The key idea is to store the vertex states and sent messages at each superstep as RDDs and to group them by vertex IDs to perform the shuffle communication pattern. We then apply the user's function on the state and messages for each vertex ID to produce a new RDD of (VertexID, (NewState, OutgoingMessages)) pairs, and map it to separate out the next iteration's states and messages.[7]

```
val vertices = // RDD of (ID, State) pairs
val messages = // RDD of (ID, Message) pairs
val grouped = vertices.cogroup(messages)
val newData = grouped.mapValues {
  (vert, msgs) => userFunc(vert, msgs)
       // returns (newState, outgoingMsgs)
}.cache()
val newVerts = newData.mapValues((v,ms) => v)
val newMsgs = newData.flatMap((id,(v,ms)) => ms)
```

An important aspect of this implementation is that the `grouped`, `newData`, and `newVerts` RDDs will be partitioned in the same way as the input RDD, `vertices`. As a result, vertex states will not leave the machines that they started on, much like in Pregel, reducing the job's communication cost. This partitioning happens automatically because *cogroup* and *mapValues* preserve the partitioning of their input RDD.[8]

The full Pregel programming model includes several other facilities, such as combiners. We discuss how to implement them in Appendix A. In the remainder of this section, we discuss fault tolerance in Pregel, and show how RDDs can provide the same fault recovery properties and can also reduce the amount of data to checkpoint.

We have implemented a Pregel-like API as a library on top of Spark in about 100 lines of Scala, and we evaluate it with PageRank in Section 7.2.

4.3.1 Fault Tolerance for Pregel

Pregel currently checkpoints vertex states and messages periodically for fault tolerance [21]. However, the authors also describe work on confined recovery to rebuild lost partitions individually by logging sent messages on other nodes. RDDs can support both approaches.

With just the implementation in Section 4.3, Spark can always rebuild the vertex and message RDDs using lineage, but recovery may be expensive due to the long lineage chain. Because each iteration's RDDs depend on the previous one's, losing a node may cause the loss of all iterations' versions of the state for some partition, requiring "cascaded re-execution" [20] to rebuild each lost partition in turn. To avoid this, users can call *save* on the vertices and messages RDDs periodically to checkpoint them to stable storage. Once this is done, Spark will automatically recompute only the lost partitions (rather than rolling back the whole program) on a failure.

Finally, we note that RDDs can also *reduce* the amount of checkpointing required by expressing a more efficient checkpointing scheme. In many Pregel jobs, vertex states include both immutable and mutable components. For example, in PageRank, a vertex's list of neighbors never changes, but its rank does. In such cases, we can place the immutable data in a separate RDD from the mutable data, with a short lineage chain, and checkpoint just the mutable state. We illustrate this approach in Figure 4.

[7] We assume that the State type includes both the vertex's edge list and its user-defined state, *e.g.,* its rank in PageRank.

[8] Users could use a custom partitioner to further reduce communication, *e.g.,* by placing pages from the same domain on the same node.

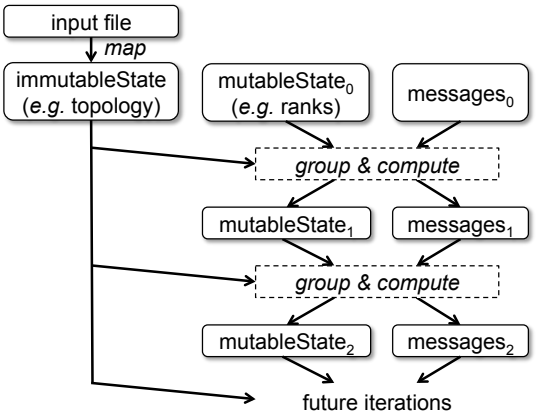


Figure 4: Data flow for an optimized variant of Pregel using RDDs. Only the mutable state RDDs must be checkpointed; immutable state can be rebuilt quickly by re-mapping the input.

In PageRank, the immutable state (a list of neighbors) is much larger than the mutable state (a floating-point value), so this approach can greatly reduce overhead.

4.4 HaLoop using RDDs

HaLoop [8] is an extended version of the Hadoop framework designed to improve the performance of iterative MapReduce programs. Applications expressed using HaLoop's programming model uses the output of the reduce phases of earlier iterations as the input to the map phase of the next iteration. Its loop-aware task scheduler tries to ensure that in every iteration, consecutive map and reduce tasks that process the same partition of data are scheduled on the same physical machine. Ensuring inter-iteration locality reduces data transfer between machines and allows the data to be cached in local disks for reuse in later iterations.

We have implemented a HaLoop-like API on Spark using RDDs to express HaLoop's optimizations. The consistent partitioning across iterations is ensured through *partitionBy*, and the input and output of every phase are cached for use in later iterations. This library is written in 200 lines of Scala code.

4.5 Applications Not Suitable for RDDs

As discussed in Section 2.1, RDDs are best suited for applications that perform bulk transformations that apply the same operation to all the elements of a dataset. In these cases, RDDs can efficiently remember each transformation as one step in a lineage graph and can recover lost partitions without having to log a large amount of data. RDDs would be less suitable for applications that make asynchronous fine-grained updates to shared state, such as a storage system for a web application or an incremental web crawler and indexer. For these applications, it is more efficient to use systems that perform traditional update logging and data checkpointing, such as databases, RAMCloud [26], Percolator [27] and Piccolo [28]. Our goal is to provide an efficient programming model for batch analytics applications and leave these asynchronous applications to specialized systems.

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: Internal interface of RDDs in Spark.

5 RDD Representation and Job Scheduling

In Spark, we wanted to support a wide range of composable transformations on RDDs without having to modify the scheduler for each new transformation, and we wanted to capture lineage information across all these transformations. We designed a small, common internal interface for RDDs that facilitates these goals.

In a nutshell, each RDD has a set of *partitions*, which are atomic pieces of the dataset; a set of *dependencies* on parent RDDs, which capture its lineage; a function for computing the RDD based on its parents; and metadata about its partitioning scheme and data placement. For example, an RDD representing an HDFS file has a partition for each block of the file and knows which nodes each block is on from HDFS. Meanwhile, the result of a *map* on this RDD has the same partitions, but applies the map function to the parent's data when computing its elements. We summarize the RDD interface in Table 3.

The most interesting question in designing this interface is how to represent dependencies between RDDs. We found it both sufficient and useful to classify dependencies into two types: *narrow* dependencies, where each partition of the child RDD depends on a constant number of partitions of the parent (not proportional to its size), and *wide* dependencies, where each partition of the child can depend on data from all partitions of the parent. For example, *map* leads to a narrow dependency, while *join* leads to to wide dependencies (unless the parents are hash-partitioned). Figure 5 shows other examples.[9]

This distinction is useful for two reasons. First, narrow dependencies allow for pipelined execution on one cluster node, which can compute all the parent partitions. For example, one can apply a *map* followed by a *filter* on an element-by-element basis. In contrast, wide dependencies require data from all parent partitions to be available

[9]While one can envision dependencies that are neither narrow nor wide, we found that these classes captured all the operations in §3.1.

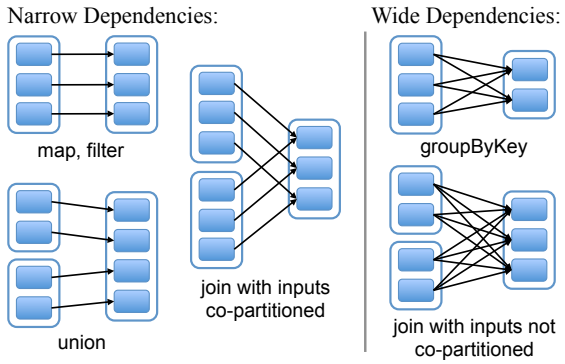


Figure 5: Examples of narrow and wide dependencies. Each box is an RDD, with partitions shown as shaded rectangles.

and to be shuffled across the nodes using a MapReduce-like operation. Second, recovery after a node failure is more efficient with a narrow dependency, as only the lost parent partitions need to be recomputed, and they can be recomputed in parallel on different nodes. In contrast, in a lineage graph with wide dependencies, a single failed node might cause the loss of some partition from all the ancestors of an RDD, requiring a complete re-execution.

Thanks to our choice of common interface for RDDs, we could implement most transformations in Spark in less than 20 lines of code. We sketch some examples in §5.1. We then discuss how we use the RDD interface for scheduling (§5.2). Lastly, we discuss when it makes sense to checkpoint data in RDD-based programs (§5.3).

5.1 Examples of RDD Implementations

HDFS files: The input RDDs in our examples are all files in HDFS. For these RDDs, *partitions* returns one partition for each block of the file (with the block's offset stored in each Partition object), *preferredLocations* gives the nodes the block is on, and *iterator* reads the block.

***map*:** Calling *map* on any RDD returns a MappedRDD object. This object has the same partitions and preferred locations as its parent, but applies the function passed to *map* to the parent's records in its *iterator* method.

***union*:** Calling *union* on two RDDs returns an RDD whose partitions are the union of those of the parents. Each child partition is computed through a narrow dependency on the corresponding parent.[10]

***sample*:** Sampling is similar to mapping, except that the RDD stores a random number generator seed for each partition to deterministically sample parent records.

***join*:** Joining two RDDs may lead to either two narrow dependencies (if they are both hash/range partitioned with the same partitioner), two wide dependencies, or a mix (if one parent has a partitioner and one does not).

[10]Note that our *union* operation does not drop duplicate values—it is equivalent to SQL's `UNION ALL`.

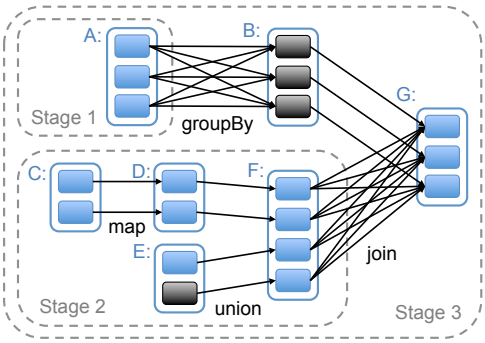


Figure 6: Example showing how Spark computes job stages. Boxes with solid outlines are RDDs. Partitions are shaded rectangles, in black if they are cached. To run an action on RDD G, the scheduler builds stages at wide dependencies and pipelines narrow transformations inside each stage. In this case, stage 1 does not need to run since B is cached, so we run 2 and then 3.

5.2 Spark Job Scheduler

Our scheduler uses the structure of RDDs to find an efficient execution plan for each action. The interface of the scheduler is a *runJob* function that takes an RDD to work on, a set of partitions of interest, and a function to run on the partitions. This interface is sufficient to express all the actions in Spark (*count*, *collect*, *save*, etc).

Overall, our scheduler is similar to Dryad's [18], but additionally takes into account which partitions of RDDs are available in caches. The scheduler examines the lineage graph of the target RDD to builds a DAG of *stages* to execute. Each stage contains as many pipelined transformations with narrow dependencies as possible. The boundaries of the stages are the shuffle operations required for wide dependencies, or any cached partitions that can short-circuit the computation of a parent RDD. Figure 6 shows an example. We launch tasks to compute missing partitions from each stage when its parents end.

The scheduler places tasks based on data locality to minimize communication. If a task needs to process a cached partition, we send it to a node that has that partition. Otherwise, if a task processes a partition for which the containing RDD provides preferred locations (*e.g.,* due to data locality in HDFS), we send it to those.

For wide dependencies (*i.e.,* shuffle dependencies), we currently materialize intermediate records on the nodes holding parent partitions to simplify fault recovery, much like MapReduce materializes map outputs.

If a task fails, we re-run it on another node as long as its stage's parents are still available. If some stages have become unavailable (*e.g.,* because an output from the "map side" of a shuffle was lost), we resubmit tasks to compute the missing partitions of those stages.

Finally, the *lookup* action, which lets the user fetch an element from a hash or range partitioned RDD by its key, poses an interesting design question. When *lookup* is

called by the driver program, we can build just the partition that the desired key falls in using the existing scheduler interface. However, we are also experimenting with allowing tasks on the cluster (*e.g.,* maps) to call *lookup*, to let users treat RDDs as large distributed hash tables. In this case, the dependency between the tasks and the RDD being looked up is not explicitly captured (since workers are calling *lookup*), but we have the task tell the scheduler to compute the RDD in question if it cannot find cached partitions for it registered on any nodes.

5.3 Checkpointing

Although the lineage information tracked by RDDs always allows a program to recover from failures, such recovery may be time-consuming for RDDs with long lineage chains. For example, in the Pregel jobs in Section 4.3, each iteration's vertex states and messages depends on the previous ones. Thus, it can be helpful to checkpoint RDDs with long lineage chains to stable storage.

In general, checkpointing is useful for RDDs with long lineage graphs composed of wide dependencies. In these cases, a node failure in the cluster may result in the loss of some slice of data from each parent RDD, requiring a full recomputation [20]. In contrast, for RDDs with narrow dependencies on data in stable storage, such as the points in our logistic regression example (§4.1) and the immutable vertex states in our optimized variant of Pregel (§4.3.1), checkpointing may never be worthwhile. Whenever a node fails, lost partitions from these RDDs can be recomputed in parallel on other nodes, at a fraction of the cost of replicating the whole RDD.

Spark currently provides an API for checkpointing but leaves the decision of which data to checkpoint to the user. In future work, we plan to implement automatic checkpointing using cost-benefit analysis to pick the best cut in the RDD lineage graph to checkpoint at.[11]

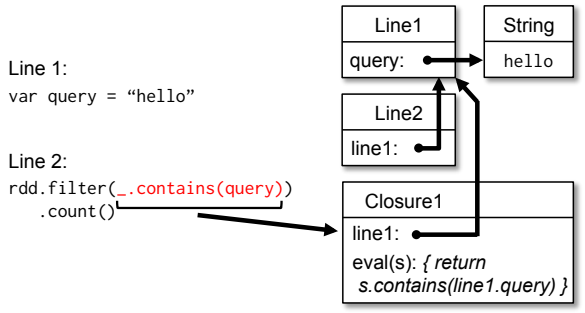
Note that because RDDs are read-only, they can be checkpointed in the background without incurring any overhead to maintain consistency (*e.g.,* copy-on-write schemes, distributed snapshots, or program pauses).

6 Implementation

We have implemented Spark in about 10,000 lines of Scala. The system can use any Hadoop data source (*e.g.,* HDFS, HBase) as input, making it easy to integrate into Hadoop environments. We did *not* need to modify the Scala compiler; Spark is implemented as a library.

We discuss three of the technically interesting parts of the implementation: our modified Scala interpreter that allows interactive use of Spark (§6.1), cache management (§6.2), and our debugging tool, `rddbg` (§6.3).

[11]The problem of checkpoint selection for is similar to optimal checkpointing for HPC systems [33].



a) Lines typed by user b) Resulting object graph

Figure 7: Example showing how the Spark interpreter translates two lines entered by the user into Java objects.

6.1 Interpreter Integration

Scala includes an interactive shell similar to those of Ruby and Python. Given the low latencies attained with in-memory data, we wanted to let users run Spark interactively from the interpreter as a large-scale "parallel calculator" for mining big datasets.

The Scala interpreter normally operates by compiling a class for each line typed by the user, loading it into the JVM, and invoking a function on it. This class includes a singleton object that contains the variables or functions on that line and runs the line's code in an initialize method. For example, if the user types `var x = 5` followed by `println(x)`, the interpreter defines a class called `Line1` containing `x` and causes the second line to compile to `println(Line1.getInstance().x)`.

We made two changes to the interpreter in Spark:

1. *Class shipping:* To let the worker nodes fetch the bytecode for the classes created on each line, we made the interpreter serve these classes over HTTP.
2. *Modified code generation:* Normally, the singleton object created for each line of code is accessed through a static method on its corresponding class. This means that when we serialize a closure referencing a variable defined on a previous line, such as `Line1.x` in the example above, Java will not trace through the object graph to ship the `Line1` instance wrapping around `x`. Therefore, the worker nodes will not receive `x`. We modified the code generation logic to reference the instance of each line object directly.

Figure 7 shows how the interpreter translates a set of lines typed by the user to Java objects after our changes.

We found the Spark interpreter to be useful in processing large traces obtained as part of our research and exploring datasets stored in HDFS. We also plan to use it as a basis for interactive tools providing higher-level data analytics languages, such as variants of SQL and Matlab.

6.2 Cache Management

Our worker nodes cache RDD partitions in memory as Java objects. We use an LRU replacement policy at the level of RDDs (*i.e.,* we do not evict partitions from an RDD in order to load other partitions from the same RDD) because most operations are scans. We found this simple policy to work well in all our user applications so far. Programmers that want more control can also set a retention priority for each RDD as an argument to *cache*.

6.3 `rddbg`: A Tool for Debugging RDD Programs

While we initially designed RDDs to be deterministically recomputable for fault tolerance, this property also facilitates debugging. We built a tool called `rddbg` that uses lineage information logged by a program to let the user (1) reconstruct any RDDs created by the program and query it interactively and (2) re-run any task in the job in a single-process Java debugger (*e.g.,* `jdb`) by feeding in the recomputed RDD partitions it depended on.

We stress that `rddbg` is *not* a full replay debugger: in particular, it does not replay nondeterministic code or behavior. However, it is useful for finding logic bugs as well as performance bugs where one task is consistently slow (*e.g.,* due to skewed data or unusual input).

For example, we used `rddbg` to fix a bug in a user's spam classification job (§7.5) where all iterations were producing only zero values. Re-executing a reduce task in the debugger quickly indicated that the input weight vector (stored in a custom sparse vector class) was unexpectedly empty. As reads from unset elements in the sparse vector return zero by design, no runtime exception was ever raised. Setting a breakpoint in the vector class and executing the task again quickly led into the deserialization code where we found the array of the vector class's field names being deserialized was also empty, which allowed us to diagnose the bug: the data field in the sparse vector class was mistakenly marked as transient (an out-of-date annotation used with a previous serialization library), preventing it from being serialized.

`rddbg` adds minimal overhead to a program's execution because the program already has to serialize and send all the closures associated with each RDD across the network. We simply log them to disk as well.

7 Evaluation

We evaluated Spark and RDDs through a series of experiments on Amazon EC2 [1], including comparisons with Hadoop and benchmarks of user applications. Overall, our results show the following:

- Spark outperforms Hadoop by up to $20\times$ in iterative machine learning applications. The speedup comes from storing data in memory and from avoiding deserialization cost by caching Java objects.
- Applications written by our users perform well. In particular, we used Spark to speed up an analytics report that was running on Hadoop by $40\times$.
- When nodes fail, Spark can recover quickly by rebuilding only the lost RDD partitions.
- Spark can be used to query a 1 TB dataset interactively with latencies of 5–7 seconds.

We start by presenting benchmarks for iterative machine learning applications (§7.1) and PageRank (§7.2) against Hadoop. We then evaluate fault recovery in Spark (§7.3) and behavior when the working set does not fit in cache (§7.4). Finally, we discuss results for user applications (§7.5) and interactive data mining (§7.6).

Unless otherwise specified, our experiments used `m1.xlarge` EC2 nodes with 4 cores and 15 GB of RAM. We used HDFS for persistent storage, with 256 MB blocks. Before each job, we cleared OS buffer caches across the cluster to measure disk read times accurately.

7.1 Iterative Machine Learning Applications

We implemented two iterative machine learning (ML) applications, logistic regression and k-means, to compare the performance of the following systems:

- *Hadoop:* The Hadoop 0.20.2 stable release.
- *HadoopBinMem:* A Hadoop deployment that preprocesses the input data into a low-overhead binary format in the first iteration to eliminate text parsing in later iterations and stores it in an in-memory HDFS.
- *Spark:* An RDD-enabled system that caches Java objects in the first iteration to eliminate parsing and deserialization overheads in later iterations.

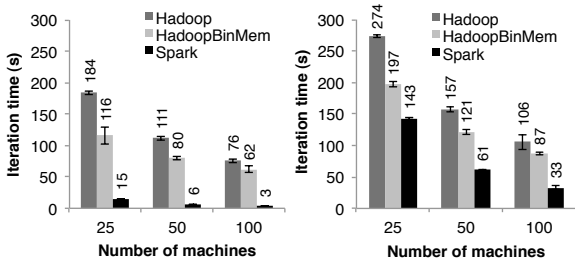
We ran both logistic regression and k-means for 10 iterations using 100 GB input datasets (Table 4) on 25–100 machines using 400 tasks (one for each block of 256 MB input). The key difference between the two jobs is the amount of computation required per byte per iteration. The iteration time of k-means is dominated by computation for updating cluster coordinates. Logistic regression is less compute-intensive and more sensitive to time spent in deserialization and parsing.

Since typical ML algorithms require tens of iterations to converge, we report times for the first iteration and subsequent iterations separately. We find that caching RDDs in memory greatly speeds up future iterations.

First Iterations All three systems read text input from HDFS in their first iterations. As shown in Figure 9 for the "first iteration" bars, Spark was faster than Hadoop across experiments mostly due to signaling overheads involved in Hadoop's heartbeat protocol between distributed components. HadoopBinMem was the slowest because it ran an extra MapReduce job to convert the data to binary.

Application	Data Description	Size
Logistic Regression	1 billion 9-dimensional points	100 GB
K-means	1 billion 10-dimensional points (k = 10 clusters)	100 GB
PageRank	Link graph from 4 million Wikipedia articles	49 GB
Interactive Data Mining	Wikipedia page view logs from October, 2008 to April, 2009	1 TB

Table 4: Applications used to benchmark Spark.



(a) Logistic Regression (b) K-Means

Figure 8: Running times for iterations after the first one in Hadoop, HadoopBinMem, and Spark. The jobs all processed 100 GB data on an increasing number of machines.

Subsequent Iterations Figure 9 also shows the average times for subsequent iterations, and Figure 8 shows those times for different cluster sizes. We found Spark to be up to $25.3\times$ and $20.7\times$ faster than Hadoop and HadoopBinMem respectively for logistic regression on 100 machines. From Figure 8(b), Spark only achieved a $1.9\times$ to $3.2\times$ speedup over Hadoop because k-means is dominated by computation overhead (using more machines helps increase the speedup factor).

In the subsequent iterations, Hadoop still read text input from HDFS, so Hadoop's iteration times did not improve noticeably from its first iterations. Using pre-converted SequenceFiles (Hadoop's binary storage format), HadoopBinMem saved parsing costs in subsequent iterations. However, HadoopBinMem still incurred other overheads, such as reading each file from the in-memory HDFS and converting bytes to Java objects. Because Spark operates directly on cached Java objects in RDDs, its iteration times decreased significantly, scaling linearly as the cluster size increased.

Understanding the Speedup We were surprised to find that Spark outperformed even Hadoop with in-memory storage of binary data (HadoopBinMem) by a $20\times$ margin. Hadoop ran slower due to several factors:

1. Minimum overhead of the Hadoop software stack,
2. Overhead of the HDFS stack while reading input, and
3. Deserialization cost to convert binary records to usable in-memory Java objects.

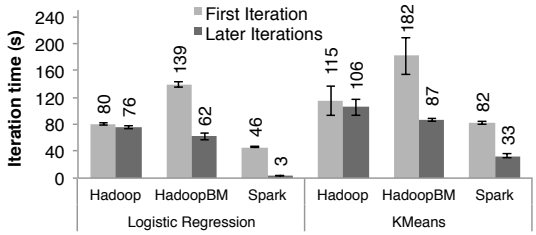


Figure 9: Length of the first and later iterations for Hadoop, HadoopBinMem, and Spark for logistic regression and k-means using 100 GB data on a 100-node cluster.

	In-memory HDFS File	In-memory Local File	Cached RDD
Text Input	15.38 (0.26)	13.13 (0.26)	2.93 (0.31)
Binary Input	8.38 (0.10)	6.86 (0.02)	

Table 5: Iteration times for logistic regression using 256 MB data on a single machine for different forms of input. Numbers in parentheses represent standard deviations.

To measure (1) we ran no-op Hadoop jobs, and these took at least 25 seconds just to complete the minimal requirements of job setup, starting tasks, and cleaning up. Regarding (2), we found that HDFS performed multiple memory copies and a checksum to serve each block.

To measure (3), we ran microbenchmarks on a single machine to compute logistic regression on 256 MB inputs, and show the results in Table 5. First, the differences between in-memory HDFS and local file show that reading through the HDFS interface introduced a 2-second overhead, even when data was in memory on the local machine. Second, the differences between the text and binary input indicate parsing overhead was 7 seconds. Finally, converting pre-parsed binary data into in-memory Java objects took 3 seconds. These overheads will accumulate as each machine processes more blocks. By caching RDDs as in-memory Java objects, Spark incurred only the 3 seconds of computing time.

7.2 PageRank

We compared the performance of our Pregel implementation using RDDs with Hadoop for the PageRank application using a 49 GB Wikipedia dump stored in HDFS. We ran 10 iterations of the PageRank algorithm to process a link graph of approximately 4 million articles. Figure 10 demonstrates that in-memory caching alone provided Spark with a $2\times$ speedup over Hadoop on 30 nodes. Specifying hash partitioning for the input vertices RDD improved the speedup to $2.6\times$, and introducing combiners further increased it to $3.6\times$. The results also scaled nearly linearly to 60 nodes.

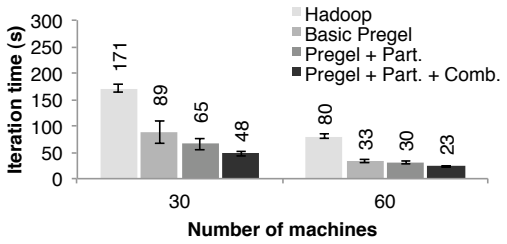


Figure 10: Comparative iteration times for Hadoop and Pregel implementations on Spark for the PageRank application on a 49 GB Wikipedia dump.

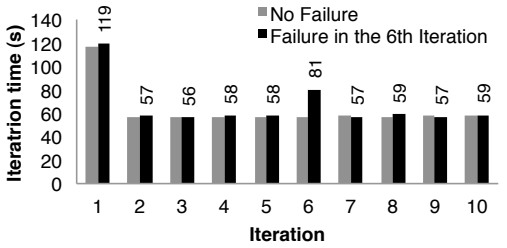


Figure 11: Iteration times during k-means computation in presence of single node failure. One machine was killed at the beginning of the 6th iteration, which resulted in partial RDD reconstruction using lineage information.

7.3 Fault Recovery

We evaluated the overhead of reconstructing RDD partitions using lineage information in case of a single-node failure for the k-means application. Figure 11 compares the iteration times for 10 iterations of the k-means application on a 75-node cluster in normal operating scenario with when the cluster lost one node at the beginning of the 6th iteration. Without any failure, each iteration consisted of 400 tasks working on 100 GB of data.

Until the end of the 5th iteration, the iteration times were about 58 seconds. In the 6th iteration, one of the machines was killed, and the tasks running in that machine died along with the partitions cached there. The Spark scheduler re-ran these tasks in parallel on other machines, where they re-read corresponding input data, reconstructed RDDs from lineage information, and re-cached them, which increased the iteration time to 80 seconds. Once the lost RDD partitions were reconstructed, the average iteration time went back down to 58 seconds, same as before the failure.

7.4 Behavior with Insufficient Memory

So far, we ensured that every machine in the cluster had enough memory to cache all the RDDs across iterations. A natural question is how Spark runs if there is not enough memory to store a job's working set. In this experiment, we configured Spark not to use more than a certain percentage of required memory to cache RDDs in each machine. We present results for various amounts of cache space for logistic regression in Figure 12. We see

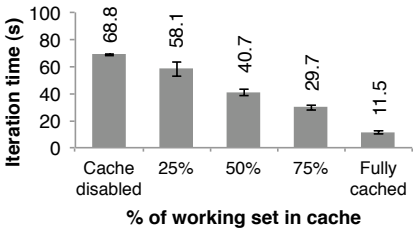


Figure 12: Spark performance for logistic regression using 100 GB data on 25 machines for varying size of RDD cache.

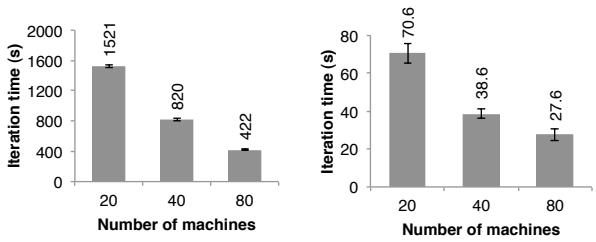
that performance degrades gracefully with less space.

7.5 User Applications Built on Spark

In-Memory Analytics Conviva Inc, a video distribution company, used Spark to greatly accelerate an analytics report that they compute for their customers, which previously ran using about 20 Hive [3] queries over Hadoop. These queries all worked on the same subset of the data (records matching some customer-provided predicates), but performed aggregations (sums, averages, percentiles, and `COUNT DISTINCT`) over different grouping fields, requiring separate MapReduce jobs. The company used Spark to load the data of interest into RAM once and then run the aggregations. A report on about 200 GB of compressed data that used to take 20 hours on a Hadoop cluster now runs in 30 minutes on only two Spark machines with 96 GB RAM. The $40\times$ speedup comes from not having to decompress and filter the data repeatedly on each job.

Traffic Modeling Researchers in the Mobile Millennium project at Berkeley [17] parallelized a learning algorithm for inferring road traffic congestion from sporadic automobile GPS measurements. The source data were a 10,000 link road network for a metropolitan area, as well as 600,000 samples of point-to-point trip times for GPS-equipped automobiles (travel times for each path may include multiple road links). Using a traffic model, the system can estimate and congestion by inferring the expected time it takes to travel across individual road links. The researchers trained this model using an Spark implementation of an iterative expectation maximization (EM) algorithm, which involved broadcasting road network data to the worker nodes, and performing *reduceByKey* operations between the E and M steps. The application scaled nearly linearly from 20 to 80 nodes (with 4 cores/node), as shown in Figure 13(a).

Social Network Spam Classification The Monarch project at Berkeley [31] used Spark to identify link spam in Twitter messages. They implemented a logistic regression classifier on top of Spark similar to the example in Section 7.1, but they used a distributed *reduceByKey* to sum the gradient vectors in parallel. In Figure 13(b) we



(a) Traffic modeling (b) Spam classification

Figure 13: Per-iteration run time of (b) traffic modeling application and (a) social network spam classification on Spark. Error bars represent one standard deviation.

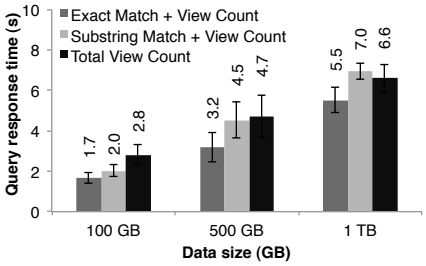


Figure 14: Response times for different interactive queries on increasingly larger input datasets.

show the scaling results for training a classifier over a 50 GB subset of the data — 250,000 URLs and at least 10^7 distinct features/dimensions related to the network, content and lexical properties of the pages associated with visiting a URL. The scaling is not as close to linear as for the traffic application due to a higher fixed communication cost per iteration.

7.6 Interactive Data Mining

To demonstrate Spark' ability to interactively process large data volumes, we used it to analyze 1TB of Wikipedia page view logs from Oct. 2008 to Apr. 2009. For this experiment, we used 100 `m2.4xlarge` EC2 instances each with 8 cores and 68 GB of RAM. We tried the following simple queries to find total views of (1) all pages, (2) pages with titles exactly matching a given word, and (3) pages with titles partially matching a word. Each of these queries scanned the entire input data.

Figure 14 presents the response times for the queries on the full dataset as well as on half and one-tenth of the dataset. Even at 1 TB of data, queries on Spark took 5–7 seconds. This was more than an order of magnitude faster than working with on-disk data; for example, querying the 1 TB dataset from disk took 170s. This illustrates that RDD caching makes Spark a powerful tool for interactive data mining.

8 Related Work

Distributed Shared Memory (DSM): RDDs can be viewed as a form of distributed shared memory, which is a well-studied abstraction [24]. As discussed in §2.5, RDDs provide a more restricted programming model than DSM, but one that lets datasets be rebuilt efficiently if nodes fail. While some DSM systems achieve fault tolerance through checkpointing [19], Spark reconstructs lost partitions of RDDs using lineage. These partitions can be recomputed in parallel on different nodes, without reverting the whole program to a checkpoint. RDDs also push computation to the data as in MapReduce [12], and can mitigate stragglers through speculative execution, which would be difficult to implement in a general shared memory system.

In-Memory Cluster Computing: Piccolo [28] is a cluster programming model based on mutable, in-memory distributed tables. Because Piccolo allows read-write access to table entries, it has similar recovery characteristics to distributed shared memory, requiring checkpoints and rollback and not supporting speculative execution. Piccolo also does not provide higher-level data flow operators like *groupBy* and *sort*, requiring the user to read and write table cells directly to implement these operators. As such, Piccolo's programming model is lower-level than Spark's, but higher-level than DSM.

RAMClouds [26] were proposed as a storage system for web applications. They likewise provide fine-grained reads and writes, so they must log data for fault tolerance.

Data Flow Systems: RDDs extend the popular "parallel collection" programming model used in DryadLINQ [34], Pig [25], and FlumeJava [9] to efficiently support applications with working sets, by allowing users to explicitly store datasets in memory as unserialized objects, control their partitioning, and perform random key lookups. As such, RDDs retain the high-level nature of programming in these data flow systems and will be familiar to existing developers, while supporting a substantially broader class of applications. While the extensions RDDs add are conceptually simple, to our knowledge, Spark is the first system to add these capabilities to a DryadLINQ-like programming model to efficiently support applications with working sets.[12]

Several specialized systems have been developed for applications with working sets, including Twister [13] and HaLoop [8], which implement an iterative MapReduce model, and Pregel [21], which provides a Bulk Synchronous Parallel programming model for graph applications. RDDs are a more general abstraction that can express both iterative MapReduce, Pregel, and applications that are not captured by these systems, such as interactive data mining. In particular, they allow programmers

[12] FlumeJava supports a "cached execution" mode where it will save intermediate results to a distributed file system for reuse in future runs, but it does not provide in-memory storage or avoid the cost of replication in the distributed file system, and does not let the programmer explicitly choose which datasets to cache.

to choose which operations to run on RDDs dynamically (e.g. look at the result of one query to decide what query to run next), rather than providing a fixed set of steps to iterate, and they support more types of transformations.

Finally, Dremel [22] is a low-latency query engine for large on-disk datasets that leverages a column-oriented format for nested record data. The same format could be used to store RDDs in Spark, but Spark also gives users the ability to load data into RAM for faster queries.

Lineage: Capturing lineage or provenance information for data has long been a research topic in scientific computing and databases, for applications such as explaining results, allowing them to be reproduced by others, and recomputing data if a bug is found in a workflow or if a dataset is lost. We refer the reader to [6] and [10] for surveys of this work. RDDs provide a restricted programming model where fine-grained lineage is inexpensive to capture, so that it can be used for fault recovery.

Caching Systems: Nectar [14] can reuse intermediate results across DryadLINQ jobs by identifying common subexpressions with program analysis. This capability would be very interesting to add to an RDD-based system. However, Nectar does not provide in-memory caching, nor does it let users explicitly control which datasets to cache and how to partition them. Ciel [23] can likewise memoize task results but does not provide in-memory caching and explicit control over caching.

Language Integration: Spark's language integration resembles that of DryadLINQ [34], which captures expression trees to run on a cluster using LINQ.[13] Unlike DryadLINQ, Spark lets users explicitly store RDDs in RAM across queries and control their partitioning to optimize communication. DryadLINQ also does not support interactive use like Spark.

Relational Databases: RDDs are conceptually similar to views in a database, and cached RDDs resemble materialized views [29]. However, like DSM systems, databases typically allow read-write access to all records, requiring logging of operations and data for fault tolerance and additional overhead to maintain consistency. These overheads are not required with the more restricted programming model of RDDs.

9 Conclusion

We have presented resilient distributed datasets (RDDs), a distributed memory abstraction for data-parallel applications on commodity clusters. RDDs support a broad range of applications with working sets, including iterative machine learning and graph algorithms and interactive data mining, while preserving the attractive properties of data flow models, such as automatic fault recovery, straggler mitigation, and locality-aware scheduling. This is accomplished by restricting the programming model to allow for efficient reconstruction of RDD partitions. Our implementation of RDDs outperforms Hadoop by up to $20\times$ in iterative jobs and can be used to interactively query hundreds of gigabytes of data.

[13] One advantage of LINQ over our closure-based approach, however, is that LINQ provides an abstract syntax tree for each expression, allowing for query optimization through operator reordering. We plan to leverage Scala equivalents of LINQ (*e.g.,* [30]) to achieve this.

Acknowledgements

We thank the first Spark users, including Timothy Hunter, Lester Mackey, Dilip Joseph, Jibin Zhan, and Teodor Moldovan, for trying out our system in their real applications, providing many helpful suggestions, and identifying a few new research challenges along the way. This research was supported in part by gifts from the Berkeley AMP Lab founding sponsors Google and SAP, AMP Lab sponsors Amazon Web Services, Cloudera, Huawei, IBM, Intel, Microsoft, NEC, NetApp, and VMWare, and by matching funds from the State of California's MICRO program (grants 06-152, 07-010), the National Science Foundation (grant CNS-0509559), the University of California Industry/University Cooperative Research Program (UC Discovery) grant COM07-10240, and the Natural Sciences and Engineering Research Council of Canada.

References

[1] Amazon EC2. `http://aws.amazon.com/ec2`.

[2] Apache Hadoop. `http://hadoop.apache.org`.

[3] Apache Hive. `http://hadoop.apache.org/hive`.

[4] Applications powered by Hadoop. `http://wiki.apache.org/hadoop/PoweredBy`.

[5] Scala. `http://www.scala-lang.org`.

[6] R. Bose and J. Frew. Lineage retrieval for scientific data processing: a survey. *ACM Computing Surveys*, 37:1–28, 2005.

[7] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *WWW*, 1998.

[8] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. HaLoop: efficient iterative data processing on large clusters. *Proc. VLDB Endow.*, 3:285–296, September 2010.

[9] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. Flumejava: easy, efficient data-parallel pipelines. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10. ACM, 2010.

[10] J. Cheney, L. Chiticariu, and W.-C. Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 1(4):379–474, 2009.

[11] C. T. Chu, S. K. Kim, Y. A. Lin, Y. Yu, G. R. Bradski, A. Y. Ng, and K. Olukotun. Map-reduce for machine learning on multicore. In *NIPS '06*, pages 281–288. MIT Press, 2006.

[12] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[13] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. Twister: a runtime for iterative mapreduce. In *HPDC '10*, 2010.

[14] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang. Nectar: automatic management of data and computation in datacenters. In *OSDI '10*, 2010.

[15] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer Publishing Company, New York, NY, 2009.

[16] U. Hoelzle and L. A. Barroso. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers, 1st edition, 2009.

[17] *Mobile Millennium* Project. `http://traffic.berkeley.edu`.

[18] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys 07*, 2007.

[19] A.-M. Kermarrec, G. Cabillic, A. Gefflaut, C. Morin, and I. Puaut. A recoverable distributed shared memory integrating coherence and recoverability. In *FTCS '95*, 1995.

[20] S. Y. Ko, I. Hoque, B. Cho, and I. Gupta. On availability of intermediate data in cloud computations. In *HotOS '09*, 2009.

[21] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, pages 135–146, 2010.

[22] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: interactive analysis of web-scale datasets. *Proc. VLDB Endow.*, 3:330–339, Sept 2010.

[23] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand. Ciel: a universal execution engine for distributed data-flow computing. In *NSDI*, 2011.

[24] B. Nitzberg and V. Lo. Distributed shared memory: a survey of issues and algorithms. *Computer*, 24(8):52 –60, aug 1991.

[25] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD '08*, pages 1099–1110.

[26] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The case for RAMClouds: scalable high-performance storage entirely in dram. *SIGOPS Oper. Syst. Rev.*, 43:92–105, Jan 2010.

[27] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *OSDI 2010*.

[28] R. Power and J. Li. Piccolo: Building fast, distributed programs with partitioned tables. In *Proc. OSDI 2010*, 2010.

[29] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, Inc., 3 edition, 2003.

[30] D. Spiewak and T. Zhao. ScalaQL: Language-integrated database queries for scala. In *SLE*, pages 154–163, 2009.

[31] K. Thomas, C. Grier, J. Ma, V. Paxson, and D. Song. Design and evaluation of a real-time URL spam filtering service. In *IEEE Symposium on Security and Privacy*, 2011.

[32] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33:103–111, August 1990.

[33] J. W. Young. A first order approximation to the optimum checkpoint interval. *Commun. ACM*, 17:530–531, Sept 1974.

[34] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI '08*, 2008.

A Pregel Details

Pregel provides several other facilities in addition to message passing. We sketch how to implement each of them using RDDs, starting with the code in Section 4.3.

Termination Votes: Vertices in Pregel vote to halt the program. This can be implemented in Spark by keeping a voted-to-halt flag in the State class and running a *reduce* over `newData` to AND together these flags.

Combiners: Messages in Pregel can be aggregated on source nodes with a combiner. This can be done using a *reduceByKey* on the `messages` RDD before the *cogroup*.

Aggregators: Pregel's aggregators merge values from all the nodes using an associative function on each iteration and make the result visible on the next one. We can merge aggregators using a *reduce* over vertex states.

Topology Mutations: Pregel lets users add and delete vertices or edges. This can be supported in Spark by sending a special message to the vertex ID whose state must be updated. In addition, to support adding and removing vertices, we need to change the *mapValues* call that creates the new vertices RDD to *flatMapValues*.