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 to worry about work distribution and fault tolerance.

Although current frameworks provide numerous abstractions for accessing a cluster's computational resources, they lack abstractions for leveraging distributed memory. This makes them inefficient for an important class of emerging applications: those that *reuse* intermediate results across multiple computations. Data reuse is common in many *iterative* machine learning and graph algorithms, including PageRank, K-means clustering, and logistic regression. Another compelling use case is *interactive* data mining, where a user runs multiple ad-hoc queries on the same subset of the data. Unfortunately, in most current frameworks, the only way to reuse data between computations (*e.g.,* between two MapReduce jobs) is to write it to an external stable storage system, *e.g.,* a distributed file system. This incurs substantial overheads due to data replication, disk I/O, and serialization, 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 to provide fault tolerance are to replicate the data across machines or to log updates across machines. Both approaches are expensive for data-intensive workloads, as they require copying large amounts of data over the cluster network, whose bandwidth is far lower than that of RAM, and they incur substantial storage overhead.

In contrast to these systems, RDDs provide an interface based on *coarse-grained* transformations (*e.g.,* map, filter and join) that apply the same operation to many data items. This allows them to efficiently provide fault tolerance by logging the transformations used to build a dataset (its *lineage*) rather than the actual data.[1] If a partition of an RDD is lost, the RDD has enough information about how it was derived from other RDDs to recompute

[1] Checkpointing the data in some RDDs may be useful when a lineage chain grows large, however, and we discuss how to do it in §5.4.

just that partition. Thus, lost data can be recovered, often quite quickly, without requiring costly replication.

Although an interface based on coarse-grained transformations may at first seem limited, RDDs are a good fit for many parallel applications, because *these applications naturally apply the same operation to multiple data items*. Indeed, we show that RDDs can efficiently express many cluster programming models that have so far been proposed as separate systems, including MapReduce, DryadLINQ, SQL, Pregel and HaLoop, as well as new applications that these systems do not capture, like interactive data mining. The ability of RDDs to accommodate computing needs that were previously met only by introducing new frameworks is, we believe, the most credible evidence of the power of the RDD abstraction.

We have implemented RDDs in a system called Spark, which is being used for research and production applications at UC Berkeley and several companies. Spark provides a convenient language-integrated programming interface similar to DryadLINQ [31] in the Scala programming language [2]. In addition, Spark can be used interactively to query big datasets from the Scala interpreter. We believe that Spark is the first system that allows a general-purpose programming language to be used at interactive speeds for in-memory data mining on clusters.

We evaluate RDDs and Spark through both microbenchmarks and measurements of user applications. We show that Spark is up to 20× faster than Hadoop for iterative applications, speeds up a real-world data analytics report by 40×, and can be used interactively to scan a 1 TB dataset with 5–7s latency. More fundamentally, to illustrate the generality of RDDs, we have implemented the Pregel and HaLoop programming models on top of Spark, including the placement optimizations they employ, as relatively small libraries (200 lines of code each).

This paper begins with an overview of RDDs (§2) and Spark (§3). We then discuss the internal representation of RDDs (§4), our implementation (§5), and experimental results (§6). Finally, we discuss how RDDs capture several existing cluster programming models (§7), survey related work (§8), and conclude.

2 Resilient Distributed Datasets (RDDs)

This section provides an overview of RDDs. We first define RDDs (§2.1) and introduce their programming interface in Spark (§2.2). We then compare RDDs with finer-grained shared memory abstractions (§2.3). Finally, we discuss limitations of the RDD model (§2.4).

2.1 RDD Abstraction

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

RDDs do not need to be materialized at all times. Instead, an RDD has enough information about how it was derived from other datasets (its *lineage*) to *compute* its partitions from data in stable storage. This is a powerful property: in essence, a program cannot reference an RDD that it cannot reconstruct after a failure.

Finally, users can control two other aspects of RDDs: *persistence* and *partitioning*. Users can indicate which RDDs they will reuse and choose a storage strategy for them (*e.g.,* in-memory storage). They can also ask that an RDD's elements be partitioned across machines based on a key in each record. This is useful for placement optimizations, such as ensuring that two datasets that will be joined together are hash-partitioned in the same way.

2.2 Spark Programming Interface

Spark exposes RDDs through a language-integrated API similar to DryadLINQ [31] and FlumeJava [8], where each dataset is represented as an object and transformations are invoked using methods on these objects.

Programmers start by defining one or more RDDs through transformations on data in stable storage (*e.g., map* and *filter*). They can then use these RDDs 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 dataset), *collect* (which returns the elements themselves), and *save* (which outputs the dataset to a storage system). Like DryadLINQ, Spark computes RDDs lazily the first time they are used in an action, so that it can pipeline transformations.

In addition, programmers can call a *persist* method to indicate which RDDs they want to reuse in future operations. Spark keeps persistent RDDs in memory by default, but it can spill them to disk if there is not enough RAM. Users can also request other persistence strategies, such as storing the RDD only on disk or replicating it across machines, through flags to *persist*. Finally, users can set a persistence priority on each RDD to specify which in-memory data should spill to disk first.

2.2.1 Example: Console Log Mining

Suppose that a web service is experiencing errors and an operator wants to search terabytes of logs in the Hadoop filesystem (HDFS) to find the cause. Using Spark, 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:

[2] Although individual RDDs are immutable, it is possible to implement mutable state by having multiple RDDs to represent multiple versions of a dataset. We made RDDs immutable to make it easier to describe lineage graphs, but it would have been equivalent to have our abstraction be versioned datasets and track versions in lineage graphs.

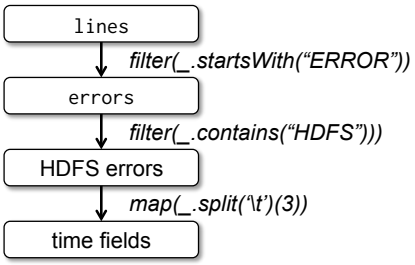


Figure 1: Lineage graph for the third query in our example. Boxes represent RDDs and arrows represent transformations.

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

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 then asks for `errors` to persist in memory so that it can be shared across queries. 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()

// 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 store the partitions of `errors` in memory, greatly speeding up subsequent computations on it. Note that the base RDD, `lines`, is *not* loaded into RAM. 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`.

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

Table 1: Comparison of RDDs with distributed shared memory.

2.3 Advantages of the RDD Model

To understand the benefits of RDDs as a distributed memory abstraction, we compare them against distributed shared memory (DSM) 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 [24], but also other systems where applications make fine-grained writes to shared state, including Piccolo [27], which provides a shared DHT, and distributed databases. 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 coarse-grained transformations, while DSM allows reads and writes to each memory location.[3] 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 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.

A second benefit of RDDs is that their immutable nature lets a system mitigate slow nodes (stragglers) by running backup copies of slow tasks as in MapReduce [10]. Backup tasks would be hard to implement with DSM, as the two copies of a task would access the same memory locations and interfere with each other's updates.

Finally, RDDs provide two other benefits over DSM. First, in bulk operations on RDDs, a runtime can sched-

[3]Note that *reads* on RDDs can still be fine-grained. For example, an application can treat an RDD as a large read-only lookup table.

[4]In some applications, it can still help to checkpoint RDDs with long lineage chains, as we discuss in Section 5.4. However, this can be done in the background because RDDs are immutable, and there is no need to take a snapshot of the *whole* application as in DSM.

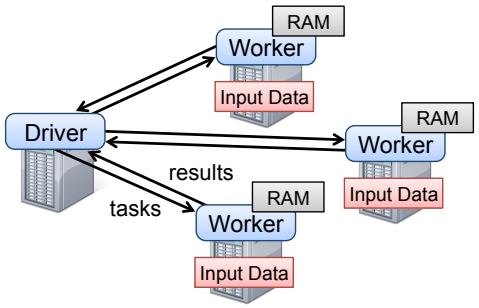


Figure 2: Spark runtime. The user's driver program launches multiple workers, which read data blocks from a distributed file system and can persist computed RDD partitions in memory.

ule tasks based on data locality to improve performance. Second, 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-parallel systems.

2.4 Applications Not Suitable for RDDs

As discussed in the Introduction, RDDs are best suited for batch applications that apply the same operation to all 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 large amounts 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. For these applications, it is more efficient to use systems that perform traditional update logging and data checkpointing, such as databases, RAMCloud [25], Percolator [26] and Piccolo [27]. Our goal is to provide an efficient programming model for batch analytics and leave these asynchronous applications to specialized systems.

3 Spark Programming Interface

Spark provides the RDD abstraction through a language-integrated API similar to DryadLINQ [31] in Scala [2], a statically typed functional programming language for the Java VM. We chose Scala due to its combination of conciseness (which is convenient for interactive use) and efficiency (due to static typing). However, nothing about the RDD abstraction requires a functional language.

To use Spark, developers write a *driver program* that connects to a cluster of *workers*, as shown in Figure 2. The driver defines one or more RDDs and invokes actions on them. Spark code on the driver also tracks the RDDs' lineage. The workers are long-lived processes that can store RDD partitions in RAM across operations.

As we showed in the log mining example in Section 2.2.1, 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]

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 [33]. We also needed more work to make Spark usable from the Scala interpreter, as we shall discuss in Section 5.2. 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 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 persist. Furthermore, users can get an RDD's partition order, which is represented by a Partitioner class, and partition another dataset according to it. Operations such as *groupByKey*, *reduceByKey* and *sort* automatically result in a hash or range partitioned RDD.

3.2 Example Applications

We complement the data mining example in Section 2.2.1 with two iterative applications: logistic regression and PageRank. The latter also showcases how control of RDDs' partitioning can improve performance.

3.2.1 Logistic Regression

Many machine learning algorithms are iterative in nature because they run iterative optimization procedures, such as gradient descent, to maximize a function. They can thus run much faster by keeping their data in memory.

As an example, the following program implements logistic regression [14], a common classification algorithm

[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.

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

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

We start by defining a persistent 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. Keeping `points` in memory across iterations can yield a 20× speedup, as we show in Section 6.1.

3.2.2 PageRank

A more complex pattern of data sharing occurs in PageRank [6]. The algorithm iteratively updates a *rank* for each document by adding up contributions from documents that link to it. On each iteration, each document sends a contribution of $\frac{r}{n}$ to its neighbors, where r is its rank and n is its number of neighbors. It then 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 documents. We can write PageRank in Spark as follows:

```
// Load graph as an RDD of (URL, outlinks) pairs
```

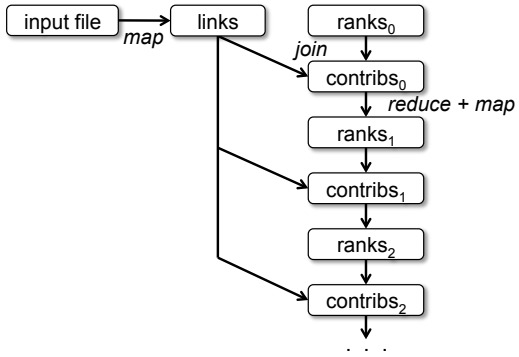


Figure 3: Lineage graph for datasets in PageRank.

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

This program leads to the RDD lineage graph in Figure 3. On each iteration, we create a new `ranks` dataset based on the `contribs` and `ranks` from the previous iteration and the static `links` dataset.[6] One interesting feature of this graph is that it grows longer with the number

[6]Note that although RDDs are immutable, the variables `ranks` and `contribs` in the program point to different RDDs on each iteration.

of iterations. Thus, in a job with many iterations, it may be necessary to reliably replicate some of the versions of `ranks` to reduce fault recovery times [20]. The user can call *persist* with a `RELIABLE` flag to do this. However, note that the `links` dataset does *not* need to be replicated, because partitions of it can be rebuilt efficiently by rerunning a *map* on blocks of the input file. This dataset will typically be much larger than `ranks`, because each document has many links but only one number as its rank, so recovering it using lineage saves time over systems that checkpoint a program's entire in-memory state.

Finally, we can optimize communication in PageRank by controlling the *partitioning* of the RDDs. If we specify a partitioning for `links` (*e.g.,* hash-partition the link lists by URL across nodes), we can partition `ranks` in the same way and ensure that the *join* operation between `links` and `ranks` requires no communication (as each URL's rank will be on the same machine as its link list). We can also write a custom Partitioner class to group pages that link to each other together (*e.g.,* partition the URLs by domain name). Both optimizations can be expressed by calling `partitionBy` when we define `links`:

```
links = spark.textFile(...).map(...)
             .partitionBy(myPartFunc).persist()
```

After this initial call, the *join* operation between `links` and `ranks` will automatically aggregate the contributions for each URL to the machine that its link lists is on, calculate its new rank there, and join it with its links. This type of consistent partitioning across iterations is one of the main optimizations in specialized frameworks like Pregel. RDDs let the user express this goal directly.

4 Representing RDDs

One of the challenges in providing RDDs as an abstraction is choosing a representation for them that can track lineage across a wide range of transformations. Ideally, a system implementing RDDs should provide as rich a set of transformation operators as possible (*e.g.,* the ones in Table 2), and let users compose them in arbitrary ways. We propose a simple graph-based representation for RDDs that facilitates these goals. We have used this representation in Spark to support a wide range of transformations without adding special logic to the scheduler for each one, which greatly simplified the system design.

In a nutshell, we propose representing each RDD through a common interface that exposes five pieces of information: a set of *partitions*, which are atomic pieces of the dataset; a set of *dependencies* on parent RDDs; a function for computing the dataset 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 machines each block is on. 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 this interface in Table 3.

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.

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 parent RDD is used by at most one partition of the child RDD, *wide* dependencies, where multiple child partitions may depend on it. For example, *map* leads to a narrow dependency, while *join* leads to to wide dependencies (unless the parents are hash-partitioned). Figure 4 shows other examples.

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 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.

This common interface for RDDs made it possible to implement most transformations in Spark in less than 20 lines of code. Indeed, even new Spark users have implemented new transformations (*e.g.,* sampling and various types of joins) without knowing the details of the scheduler. We sketch some RDD implementations below.

HDFS files: The input RDDs in our samples have been 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

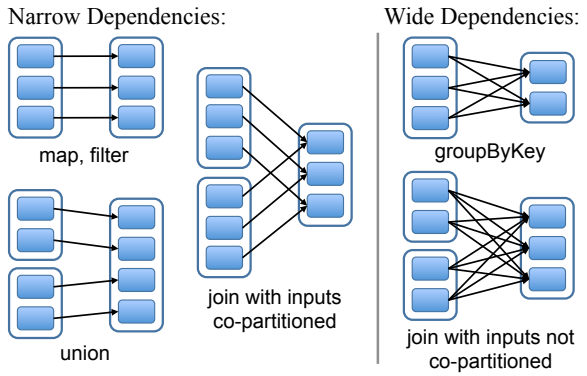


Figure 4: Examples of narrow and wide dependencies. Each box is an RDD, with partitions shown as shaded rectangles.

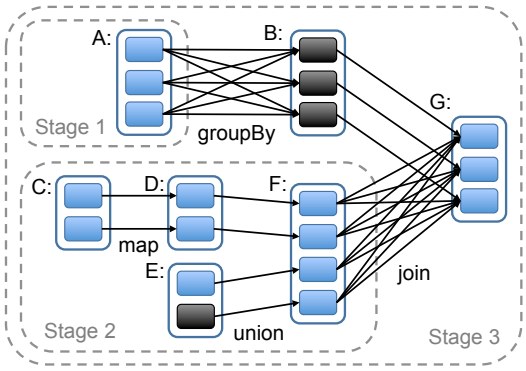


Figure 5: Example of how Spark computes job stages. Boxes with solid outlines are RDDs. Partitions are shaded rectangles, in black if they are already in memory. To run an action on RDD G, we build build stages at wide dependencies and pipeline narrow transformations inside each stage. In this case,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

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.[7]

***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). In either case, the output RDD has a partitioner (either one inherited from the parents or a default hash partitioner).

5 Implementation

We have implemented Spark in about 14,000 lines of Scala. The system runs over the Mesos cluster manager [17], allowing it to share resources with Hadoop, MPI and other applications. Each Spark program runs as a separate Mesos application, with its own driver (master) and workers, and resource sharing between these applications is handled by Mesos.

Spark can read data from any Hadoop input source (*e.g.,* HDFS or HBase) using Hadoop's existing input plugin APIs, and runs on an unmodified version of Scala.

We now sketch several of the technically interesting parts of the system: our job scheduler (§5.1), our Spark interpreter allowing interactive use (§5.2), memory management (§5.3), and support for checkpointing (§5.4).

5.1 Job Scheduling

Spark's scheduler uses our representation of RDDs, described in Section 4.

Overall, our scheduler is similar to Dryad's [19], but it additionally takes into account which partitions of persistent RDDs are available in memory. Whenever a user runs an action (*e.g., count* or *save*) on an RDD, the scheduler examines that RDD's lineage graph to build a DAG of *stages* to execute, as illustrated in Figure 5. 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 already computed partitions that can short-circuit the computation of a parent RDD. The scheduler then launches tasks to compute missing partitions from each stage until it has computed the target RDD.

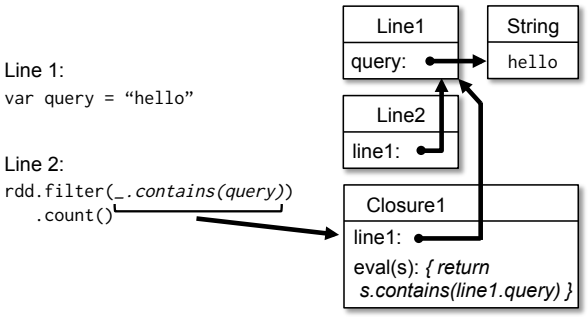
Our scheduler assigns tasks to machines based on data locality using delay scheduling [32]. If a task needs to process a partition that is available in memory on a node, we send it to that node. Otherwise, if a task processes a partition for which the containing RDD provides preferred locations (*e.g.,* an HDFS file), 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 in parallel. We do not yet tolerate scheduler failures, though replicating the RDD lineage graph would be straightforward.

Finally, although all computations in Spark currently run in response to actions called in the driver program, we are also experimenting with letting tasks on the cluster (*e.g.,* maps) call the *lookup* operation, which provides random access to elements of hash-partitioned RDDs by key. In this case, tasks would need to tell the scheduler to compute the required partition if it is missing.

[7] Note that our *union* operation does not drop duplicate values.



a) Lines typed by user b) Resulting object graph

Figure 6: Example showing how the Spark interpreter translates two lines entered by the user into Java objects.

5.2 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 to query 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 6 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 to run higher-level query languages interactively, *e.g.,* SQL.

5.3 Memory Management

Spark provides three options for storage of persistent RDDs: in-memory storage as deserialized Java objects, in-memory storage as serialized data, and on-disk storage. The first option provides the fastest performance, because the Java VM can access each RDD element natively. The second option lets users choose a more memory-efficient representation than Java object graphs when space is limited, at the cost of lower performance.[8] The third option is useful for RDDs that are too large to keep in RAM but costly to recompute on each use.

To manage the limited memory available, we use an LRU eviction policy at the level of RDDs. When a new RDD partition is computed but there is not enough space to store it, we evict a partition from the least recently accessed RDD, unless this is the same RDD as the one with the new partition. In that case, we keep the old partition in memory to prevent cycling partitions from the same RDD in and out. This is important because most operations will run tasks over an entire RDD, so it is quite likely that the partition already in memory will be needed in the future. We found this default policy to work well in all our applications so far, but we also give users further control via a "persistence priority" for each RDD.

Finally, each instance of Spark on a cluster currently has its own separate memory space. In future work, we plan to investigate sharing RDDs across instances of Spark through a unified memory manager.

5.4 Support for Checkpointing

Although lineage can always be used to recover RDDs after a failure, such recovery may be time-consuming for RDDs with long lineage chains. Thus, it can be helpful to checkpoint some RDDs to stable storage.

In general, checkpointing is useful for RDDs with long lineage graphs containing wide dependencies, such as the rank datasets in our PageRank example (§3.2.2). 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 (§3.2.1) and the link lists in PageRank, checkpointing may never be worthwhile. If 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 (a `REPLICATE` flag to *persist*), but leaves the decision of which data to checkpoint to the user. However, we are also investigating how to perform automatic checkpointing. Because our scheduler knows the size of each dataset as well as the time it took to first compute it, it should be able to select an optimal set of RDDs to checkpoint to minimize system recovery time [30].

Finally, note that the read-only nature of RDDs makes

[8] The cost depends on how much computation the application does per byte of data, but can be up to 2× for lightweight processing.

them simpler to checkpoint than general shared memory. Because consistency is not a concern, RDDs can be written out in the background without requiring program pauses or distributed snapshot schemes.

6 Evaluation

We evaluated Spark and RDDs through a series of experiments on Amazon EC2, as well as benchmarks of user applications. Overall, our results show the following:

- Spark outperforms Hadoop by up to 20× in iterative machine learning and graph applications. The speedup comes from avoiding I/O and deserialization costs by storing data in memory as Java objects.
- Applications written by our users perform and scale well. In particular, we used Spark to speed up an analytics report that was running on Hadoop by 40×.
- 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 (§6.1) and PageRank (§6.2) against Hadoop. We then evaluate fault recovery in Spark (§6.3) and behavior when a dataset does not fit in memory (§6.4). Finally, we discuss results for user applications (§6.5) and interactive data mining (§6.6).

Unless otherwise noted, our tests used `m1.xlarge` EC2 nodes with 4 cores and 15 GB of RAM. We used HDFS for storage, with 256 MB blocks. Before each test, we cleared OS buffer caches to measure IO costs accurately.

6.1 Iterative Machine Learning Applications

We implemented two iterative machine learning 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 converts the input data into a low-overhead binary format in the first iteration to eliminate text parsing in later ones, and stores it in an in-memory HDFS instance.
- *Spark:* Our implementation of RDDs.

We ran both algorithms for 10 iterations on 100 GB datasets using 25–100 machines. The key difference between the two applications is the amount of computation they perform per byte of data. The iteration time of k-means is dominated by computation, while logistic regression is less compute-intensive and thus more sensitive to time spent in deserialization and I/O.

Since typical learning algorithms need tens of iterations to converge, we report times for the first iteration and subsequent iterations separately. We find that sharing data via RDDs greatly speeds up future iterations.

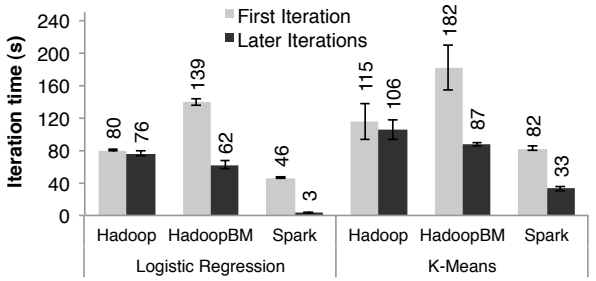
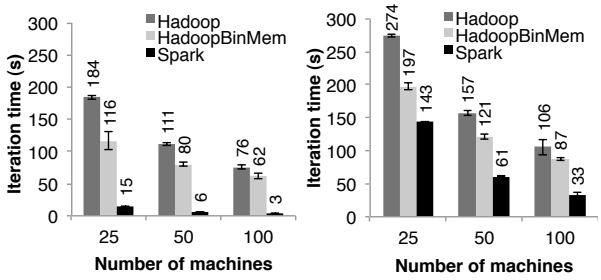


Figure 7: Duration of the first and later iterations in Hadoop, HadoopBinMem and Spark for logistic regression and k-means using 100 GB of data on a 100-node cluster.



(a) Logistic Regression (b) K-Means

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

First Iterations All three systems read text input from HDFS in their first iterations. As shown in the light bars in Figure 7, Spark was moderately faster than Hadoop across experiments. This difference was due to signaling overheads in Hadoop's heartbeat protocol between its master and workers. HadoopBinMem was the slowest because it ran an extra MapReduce job to convert the data to binary, it and had to write this data across the network to a replicated in-memory HDFS instance.

Subsequent Iterations Figure 7 also shows the average running times for subsequent iterations, while Figure 8 shows how these scaled with cluster size. For logistic regression, Spark 25.3× and 20.7× faster than Hadoop and HadoopBinMem respectively on 100 machines. For the more compute-intensive k-means application, Spark still achieved speedup of 1.9× to 3.2×.

Understanding the Speedup We were surprised to find that Spark outperformed even Hadoop with in-memory storage of binary data (HadoopBinMem) by a 20× margin. In HadoopBinMem, we had used Hadoop's standard binary format (SequenceFile) and a large block size of 256 MB, and we had forced HDFS's data directory to be on an in-memory file system. However, Hadoop still ran slower due to several factors:

1. Minimum overhead of the Hadoop software stack,
2. Overhead of HDFS while serving data, and

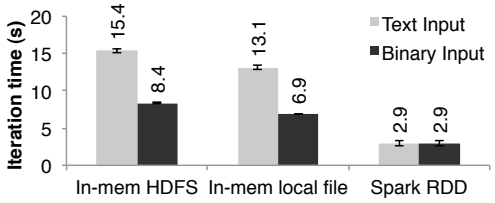


Figure 9: Iteration times for logistic regression using 256 MB data on a single machine for different sources of input.

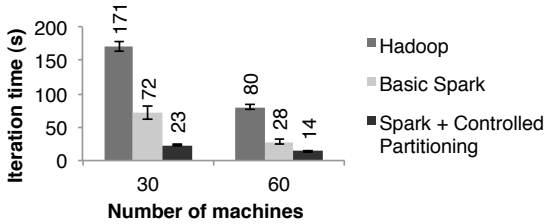


Figure 10: Performance of PageRank on Hadoop and Spark.

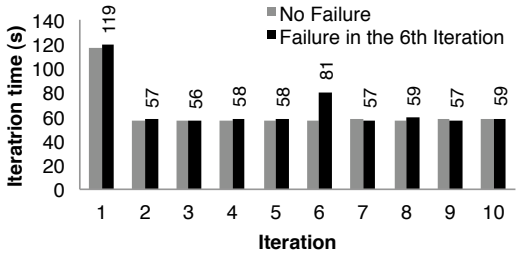


Figure 11: Iteration times for k-means in presence of a failure. One machine was killed at the start of the 6th iteration, resulting in partial reconstruction of an RDD using lineage.

3. Deserialization cost to convert binary records to usable in-memory Java objects.

We investigated each of these factors in turn. To measure (1), we ran no-op Hadoop jobs, and saw that these at incurred least 25s of overhead 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.

Finally, to measure (3), we ran microbenchmarks on a single machine to run the logistic regression computation on 256 MB inputs in various formats. In particular, we compared the time to process text and binary inputs from both HDFS (where overheads in the HDFS stack will manifest) and an in-memory local file (where the kernel can very efficiently pass data to the program).

We show the results of these tests in Figure 9. The differences between in-memory HDFS and local file show that reading through HDFS introduced a 2-second overhead, even when data was in memory on the local machine. The differences between the text and binary input indicate the parsing overhead was 7 seconds. Finally, even when reading from an in-memory file, converting the pre-parsed binary data into Java objects took 3 seconds, which is still almost as expensive as the logistic regression itself. By storing RDD elements directly as Java objects in memory, Spark avoids all these overheads.

6.2 PageRank

We compared the performance of Spark with Hadoop for PageRank using a 54 GB Wikipedia dump. We ran 10 iterations of the PageRank algorithm to process a link graph of approximately 4 million articles. Figure 10 demonstrates that in-memory storage alone provided Spark with a 2.4× speedup over Hadoop on 30 nodes. In addition, controlling the partitioning of the RDDs to make it consistent across iterations, as discussed in Section 3.2.2, improved the speedup to 7.4×. The results also scaled nearly linearly to 60 nodes.

We also evaluated a version of PageRank written using our implementation of Pregel over Spark, which we describe in Section 7.1. The iteration times were similar to the ones in Figure 10, but longer by about 4 seconds because Pregel runs an extra operation on each iteration to let the vertices "vote" whether to finish the job.

6.3 Fault Recovery

We evaluated the cost of reconstructing RDD partitions using lineage after a node failure in the k-means application. Figure 11 compares the running times for 10 iterations of k-means on a 75-node cluster in normal operating scenario, with one where a node fails at the start 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, resulting in the loss of the tasks running on that machine and the RDD partitions stored there. Spark re-ran these tasks in parallel on other machines, where they re-read corresponding input data and reconstructed RDDs via lineage, which increased the iteration time to 80s. Once the lost RDD partitions were reconstructed, the iteration time went back down to 58s.

Note that with a checkpoint-based fault recovery mechanism, recovery would likely require rerunning at least several iterations, depending on the frequency of checkpoints. Furthermore, the system would need to replicate the application's 100 GB working set (the text input data converted into binary) across the network, and would either consume twice the memory of Spark to replicate it in RAM, or would have to wait to write 100 GB to disk. In contrast, the lineage graphs for the RDDs in our examples were all less than 10 KB in size.

6.4 Behavior with Insufficient Memory

So far, we ensured that every machine in the cluster had enough memory to store all the RDDs across itera-

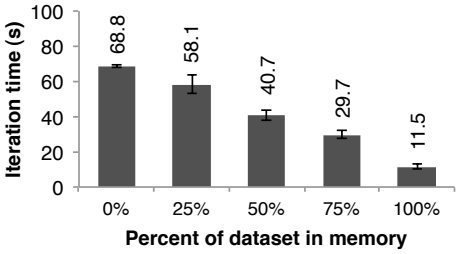
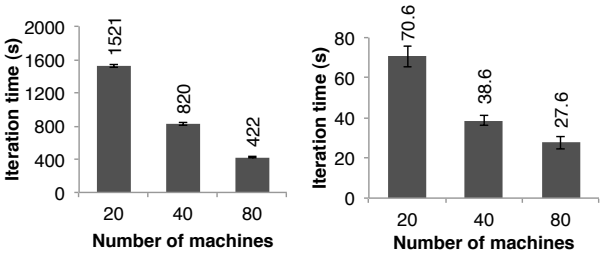


Figure 12: Performance of logistic regression using 100 GB data on 25 machines with varying amounts of data in memory.



(a) Traffic modeling (b) Spam classification

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

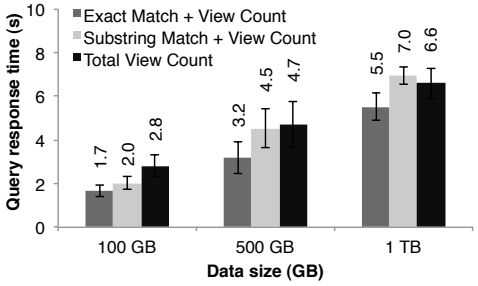


Figure 14: Response times for interactive queries on Spark, scanning increasingly larger input datasets on 100 machines.

tions. A natural question is how Spark runs if there is not enough memory to store a job's data. In this experiment, we configured Spark not to use more than a certain percentage of memory to store RDDs on each machine. We present results for various amounts of storage space for logistic regression in Figure 12. We see that performance degrades gracefully with less space.

6.5 User Applications Built with Spark

In-Memory Analytics Conviva Inc, a video distribution company, used Spark to accelerate a number of data analytics reports that previously ran over Hadoop. For example, one report ran as a series of Hive [1] queries that computed various statistics for a customer. These queries all worked on the same subset of the data (records matching a customer-provided filter), but performed aggregations (averages, percentiles, and `COUNT DISTINCT`) over different grouping fields, requiring separate MapReduce jobs. By implementing the queries in Spark and loading the subset of data shared across them once into an RDD, the company was able to speed up the report by 40×. A report on 200 GB of compressed data that took 20 hours on a Hadoop cluster now runs in 30 minutes using only two Spark machines. Furthermore, the Spark program only required 96 GB of RAM, because it only stored the rows and columns matching the customer's filter in an RDD, not the whole decompressed file.

Traffic Modeling Researchers in the Mobile Millennium project at Berkeley [18] 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 the time it takes to travel across individual road links. The researchers trained this model using an expectation maximization (EM) algorithm that repeats two *map* and *reduceByKey* steps iteratively. The application scales nearly linearly from 20 to 80 nodes with 4 cores each, as shown in Figure 13(a).

Twitter Spam Classification The Monarch project at Berkeley [29] 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 6.1, but they used a distributed *reduceByKey* to sum the gradient vectors in parallel. In Figure 13(b) we show the scaling results for training a classifier over a 50 GB subset of the data: 250,000 URLs and 10^7 features/dimensions related to the network and content properties of the pages at each URL. The scaling is not as close to linear due to a higher fixed communication cost per iteration.

6.6 Interactive Data Mining

To demonstrate Spark' ability to interactively query big datasets, we used it to analyze 1TB of Wikipedia page view logs (2 years of data). For this experiment, we used 100 `m2.4xlarge` EC2 instances with 8 cores and 68 GB of RAM each. We ran 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 query scanned the entire input data.

Figure 14 shows the response times of the queries on the full dataset and half and one-tenth of the data. 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 file from disk took 170s. This illustrates that RDDs make Spark a powerful tool for interactive data mining.

7 Discussion

Although RDDs seem to offer a limited programming interface due to their immutable nature and coarse-grained transformations, we have found them suitable for a wide class of applications. In particular, RDDs can express a surprising number of cluster programming models that have so far been proposed as separate frameworks, allowing users to *compose* these models in one program (*e.g.,* run a MapReduce operation to build a graph, then run Pregel on it) and share data between them. In this section, we discuss which programming models RDDs can express and why they are so widely applicable (§7.1). In addition, we discuss another benefit of the lineage information in RDDs that we are pursuing, which is to facilitate debugging across these models (§7.2).

7.1 Expressing Existing Programming Models

RDDs can *efficiently* express a number of cluster programming models that have so far been proposed independently. By "efficiently," we mean that not only can RDDs be used to produce the same output as programs written in these models, but that RDDs can also capture the *optimizations* that these frameworks perform, such as keeping specific data in memory, partitioning it to minimize communication, and recovering from failures efficiently. The models expressible using RDDs include:

MapReduce: This model can be expressed using the *flatMap* and *groupByKey* operations in Spark, or *reduceByKey* if there is a combiner.

DryadLINQ: The DryadLINQ system provides a wider range of operators than MapReduce over the more general Dryad runtime, but these are all bulk operators that correspond directly to RDD transformations available in Spark (*map*, *groupByKey*, *join*, etc).

SQL: Like DryadLINQ expressions, SQL queries perform data-parallel operations on sets of records.

Pregel: Google's Pregel [22] is a specialized model for iterative graph applications that at first looks quite different from the set-oriented programming models in other systems. In Pregel, a program runs as a series of coordinated "supersteps." On each superstep, each vertex in the graph runs a user function that can update state associated with the vertex, change 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.

The key observation that lets us implement this model with RDDs is that Pregel applies the *same* user function to all the vertices on each iteration. Thus, we can store the vertex states for each iteration in an RDD and perform a bulk transformation (*flatMap*) to apply this function and generate an RDD of messages. We can then join this RDD with the vertex states to perform the message exchange. Equally importantly, RDDs allow us to keep vertex states in memory like Pregel does, to minimize communication by controlling their partitioning, and to support partial recovery on failures. We have implemented Pregel as a 200-line library on top of Spark and refer the reader to [33] for more details.

Iterative MapReduce: Several recently proposed systems, including HaLoop [7] and Twister [11], provide an iterative MapReduce model where the user gives the system a series of MapReduce jobs to loop. The systems keep data partitioned consistently across iterations, and Twister can also keep it in memory. Both optimizations are simple to express with RDDs, and we were able to implement HaLoop as a 200-line library using Spark.

Batched Stream Processing: Researchers have recently proposed several incremental processing systems for applications that periodically update a result with new data [21, 15, 4]. For example, an application updating statistics about ad clicks every 15 minutes should be able to combine intermediate state from the previous 15-minute window with data from new logs. These systems perform bulk operations similar to Dryad, but store application state in distributed filesystems. Placing the intermediate state in RDDs would speed up their processing.

Explaining the Expressivity of RDDs Why are RDDs able to express these diverse programming models? The reason is that the restrictions on RDDs have little impact in many parallel applications. In particular, although RDDs can only be created through bulk transformations, many parallel programs naturally *apply the same operation to many records*, making them easy to express. Similarly, the immutability of RDDs is not an obstacle because one can create multiple RDDs to represent versions of the same dataset. Indeed, many of today's MapReduce applications run over filesystems that do not allow updates to files, such as HDFS.

One final question is why previous frameworks have not offered the same level of generality. We believe that this is because these systems explored specific problems that MapReduce and Dryad do not handle well, such as iteration, without observing that the *common cause* of these problems was a lack of data sharing abstractions.

7.2 Leveraging RDDs for Debugging

While we initially designed RDDs to be deterministically recomputable for fault tolerance, this property also facilitates debugging. In particular, by logging the lineage of RDDs created during a job, one can (1) reconstruct these RDDs later and let the user query them interactively and (2) re-run any task from the job in a single-process debugger, by recomputing the RDD partitions it depends on. Unlike traditional replay debuggers for general dis-

tributed systems [13], which must capture or infer the order of events across multiple nodes, this approach adds virtually zero recording overhead because only the RDD lineage graph needs to be logged.[9] We are currently developing a Spark debugger based on these ideas [33].

8 Related Work

Cluster Programming Models: Related work in cluster programming models falls into several classes. First, data flow models such as MapReduce [10], Dryad [19] and Ciel [23] support a rich set of operators for processing data but share it through stable storage systems. RDDs represent a more *efficient* data sharing abstraction than stable storage because they avoid the cost of data replication, I/O and serialization.[10]

Second, several high-level programming interfaces for data flow systems, including DryadLINQ [31] and FlumeJava [8], provide language-integrated APIs where the user manipulates "parallel collections" through operators like *map* and *join*. However, in these systems, the parallel collections represent either files on disk or ephemeral datasets used to express a query plan. Although the systems will pipeline data across operators in the same query (*e.g.,* a *map* followed by another *map*), they cannot share data efficiently *across* queries. We based Spark's API on the parallel collection model due to its convenience, and do not claim novelty for the language-integrated interface, but by providing RDDs as the storage abstraction behind this interface, we allow it to support a far broader class of applications.

A third class of systems provide high-level interfaces for *specific* classes of applications requiring data sharing. For example, Pregel [22] supports iterative graph applications, while Twister [11] and HaLoop [7] are iterative MapReduce runtimes. However, these frameworks perform data sharing implicitly for the pattern of computation they support, and do not provide a general abstraction that the user can employ to share data of her choice among operations of her choice. For example, a user cannot use Pregel or Twister to load a dataset into memory and *then* decide what query to run on it. RDDs provide a distributed storage abstraction explicitly and can thus support applications that these specialized systems do not capture, such as interactive data mining.

Finally, some systems expose shared mutable state to allow the user to perform in-memory computation. For example, Piccolo [27] lets users run parallel functions that read and update cells in a distributed hash table. Distributed shared memory (DSM) systems [24] and key-value stores like RAMCloud [25] offer a similar model. RDDs differ from these systems in two ways. First, RDDs provide a higher-level programming interface based on operators such as *map*, *sort* and *join*, whereas the interface in Piccolo and DSM is just reads and updates to table cells. Second, Piccolo and DSM systems implement recovery through checkpoints and rollback, which is more expensive than the lineage-based strategy of RDDs in many applications. Finally, as discussed in Section 2.3, RDDs also provide other advantages over DSM, such as straggler mitigation.

Caching Systems: Nectar [12] can reuse intermediate results across DryadLINQ jobs by identifying common subexpressions with program analysis [16]. This capability would be compelling to add to an RDD-based system. However, Nectar does not provide in-memory caching (it places the data in a distributed file system), nor does it let users explicitly control which datasets to persist and how to partition them. Ciel [23] and FlumeJava [8] can likewise cache task results but do not provide in-memory caching or explicit control over which data is cached.

Ananthanarayanan et al. have proposed adding an in-memory cache to distributed file systems to exploit the temporal and spatial locality of data access [3]. While this solution provides faster access to data that is already in the file system, it is not as efficient a means of sharing *intermediate* results within an application as RDDs, because it would still require applications to write these results to the file system between stages.

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 [5] and [9] for surveys of this work. RDDs provide a parallel programming model where fine-grained lineage is inexpensive to capture, so that it can be used for failure recovery.

Our lineage-based recovery mechanism is also similar to the recovery mechanism used *within* a computation (job) in MapReduce and Dryad, which track dependencies among a DAG of tasks. However, in these systems, the lineage information is lost after a job ends, requiring the use of a replicated storage system to share data *across* computations. In contrast, RDDs apply lineage to persist in-memory data efficiently across computations, without the cost of replication and disk I/O.

Relational Databases: RDDs are conceptually similar to views in a database, and persistent RDDs resemble materialized views [28]. However, like DSM systems, databases typically allow fine-grained read-write access to all records, requiring logging of operations and data for fault tolerance and additional overhead to maintain

[9] Unlike these systems, an RDD-based debugger will not replay nondeterministic behavior in the user's functions (*e.g.,* a nondeterministic *map*), but it can at least report it by checksumming data.

[10] Note that running MapReduce/Dryad over an in-memory data store like RAMCloud [25] would still require data replication and serialization, which can be costly for some applications, as shown in §6.1.

consistency. These overheads are not required with the coarse-grained transformation model of RDDs.

9 Conclusion

We have presented resilient distributed datasets (RDDs), an efficient, general-purpose and fault-tolerant abstraction for sharing data in cluster applications. RDDs can express a wide range of parallel applications, including many specialized programming models that have been proposed for iterative computation, and new applications that these models do not capture. Unlike existing storage abstractions for clusters, which require data replication for fault tolerance, RDDs offer an API based on coarse-grained transformations that lets them recover data efficiently using lineage. We have implemented RDDs in a system called Spark that outperforms Hadoop by up to 20× in iterative applications and can be used interactively to query hundreds of gigabytes of data.

We have open sourced Spark at `spark-project.org` as a vehicle for scalable data analysis and systems research.

Acknowledgements

We thank the first Spark users, including Tim Hunter, Lester Mackey, Dilip Joseph, and Jibin Zhan, for trying out our system in their real applications, providing many good suggestions, and identifying a few research challenges along the way. We also thank our shepherd, Ed Nightingale, and our reviewers for their feedback. This research was supported in part by Berkeley AMP Lab sponsors Google, SAP, Amazon Web Services, Cloudera, Huawei, IBM, Intel, Microsoft, NEC, NetApp and VMWare, by DARPA (contract #FA8650-11-C-7136), by a Google PhD Fellowship, and by the Natural Sciences and Engineering Research Council of Canada.

References

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

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

[3] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Disk-locality in datacenter computing considered irrelevant. In *HotOS '11*, 2011.

[4] P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquin. Incoop: MapReduce for incremental computations. In *ACM SOCC '11*, 2011.

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

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

[7] 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.

[8] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. FlumeJava: easy, efficient data-parallel pipelines. In *PLDI '10*. ACM, 2010.

[9] 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.

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

[11] 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.

[12] 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.

[13] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang. R2: an application-level kernel for record and replay. OSDI'08, 2008.

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

[15] B. He, M. Yang, Z. Guo, R. Chen, B. Su, W. Lin, and L. Zhou. Comet: batched stream processing for data intensive distributed computing. In *SoCC '10*.

[16] A. Heydon, R. Levin, and Y. Yu. Caching function calls using precise dependencies. In *ACM SIGPLAN Notices*, pages 311–320, 2000.

[17] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI '11*.

[18] T. Hunter, T. Moldovan, M. Zaharia, S. Merzgui, J. Ma, M. J. Franklin, P. Abbeel, and A. M. Bayen. Scaling the Mobile Millennium system in the cloud. In *SOCC '11*, 2011.

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

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

[21] D. Logothetis, C. Olston, B. Reed, K. C. Webb, and K. Yocum. Stateful bulk processing for incremental analytics. SoCC '10.

[22] 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*, 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] 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 Op. Sys. Rev.*, 43:92–105, Jan 2010.

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

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

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

[29] 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.

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

[31] 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.

[32] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys '10*, 2010.

[33] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. Technical Report UCB/EECS-2011-82, EECS Department, UC Berkeley, 2011.