Spark: Cluster Computing with Working Sets

Matei Zaharia
N. M. Mosharaf Chowdhury
Michael Franklin
Scott Shenker
Ion Stoica

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2010-53
http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-53.html

May 7, 2010

Acknowledgement

We thank Ali Ghodsi for feedback on this report. This research was supported by California MICRO, California Discovery, the Natural Sciences and Engineering Research Council of Canada, as well as the following Berkeley RAD Lab sponsors: Sun Microsystems, Google, Microsoft, Amazon, Cisco, Cloudera, eBay, Facebook, Fujitsu, HP, Intel, NetApp, SAP, VMware, and Yahoo!.

Spark: Cluster Computing with Working Sets

Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, Ion Stoica
University of California, Berkeley

MapReduce and its variants have been highly successful in implementing large-scale data intensive applications on clusters of unreliable machines. However, most of these systems are built around an acyclic data flow programming model that is not suitable for other popular applications. In this paper, we focus on one such class of applications: those that reuse a working set of data across multiple parallel operations. This includes many iterative machine learning algorithms, as well as interactive data analysis environments. We propose a new framework called Spark that supports these applications while maintaining the scalability and fault-tolerance properties of MapReduce. To achieve these goals, Spark introduces a data abstraction called resilient distributed datasets (RDDs). An RDD is a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost. Spark can outperform Hadoop by 10x in iterative machine learning jobs, and can be used to interactively query a 39 GB dataset with sub-second response time.

1 Introduction

A new model of cluster computing has become widely popular, in which data-parallel computations are executed on clusters of unreliable machines by systems that automatically provide locality-aware scheduling, fault tolerance, and load balancing. MapReduce [9] pioneered this model, while systems like Dryad [14] and Map-Reduce-Merge [20] generalized the types of data flows supported. These systems achieve their scalability and fault tolerance by providing a programming model where the user creates acyclic data flow graphs to pass input data through a set of operators. This allows the underlying system to manage scheduling and to react to faults without user intervention.

While this data flow programming model is useful for a large class of applications, there are applications that cannot be expressed efficiently as acyclic data flows. In this paper, we focus on one such class of applications: those that reuse a *working set* of data across multiple parallel operations. This class of applications includes two use cases where we have seen Hadoop users in academia and industry report that MapReduce by itself is inadequate:

- **Iterative jobs**: Many common machine learning algorithms apply a function repeatedly to the same dataset to optimize a parameter (e.g., through gradient descent). While each iteration can be expressed as a MapReduce/Dryad job, each job must reload the data from disk, incurring a significant performance penalty.
- **Interactive analysis**: Hadoop is often used to perform ad-hoc exploratory queries on big datasets, through SQL interfaces such as Pig [18] and Hive [1]. Ideally, a user would be able to load a dataset of interest into memory across a number of machines and query it repeatedly. However, with Hadoop, each query incurs significant latency (tens of seconds) because it runs as a separate MapReduce job and reads data from disk.

This paper presents a new cluster computing framework called Spark that supports applications with working sets while providing the same scalability and fault tolerance properties as MapReduce.

The main abstraction in Spark is that of a *resilient distributed dataset* (RDD), which represents a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost. Users can explicitly cache an RDD in memory across machines and reuse it in multiple MapReduce-like *parallel operations*. RDDs achieve fault tolerance through a notion of *lineage*: if a partition of an RDD is lost, the RDD has enough information about how it was derived from other RDDs to be able to rebuild just that partition. Although RDDs are not a general shared memory abstraction, they represent a sweet-spot between expressivity on the one hand and scalability and reliability on the other hand, and we have found them well-suited for a variety of applications.

Spark integrates into Scala [5], a statically typed high-level programming language for the Java VM, and exposes a functional programming interface similar to DryadLINQ [21]. In addition, Spark can be used interactively from a modified version of the Scala interpreter, which allows the user to define RDDs, functions, variables and classes and use them in parallel operations on a cluster. We believe that Spark is the first system to allow an efficient, general-purpose programming language to be used interactively to process large datasets on a cluster.

Although our implementation of Spark is still a prototype, early experience with the system is encouraging. We show that Spark can outperform Hadoop by 10x in iterative machine learning workloads and can be used interactively to scan a 39 GB dataset with sub-second latency.

This paper is organized as follows. Section 2 describes Spark's programming model and RDDs. Section 3 shows some example jobs. Section 4 describes our implementation, including our integration into Scala and its interpreter. Section 5 presents early results. We survey related work in Section 6 and end with a discussion in Section 7.

2 Programming Model

To use Spark, developers write a *driver program* that implements the high-level control flow of their application and launches various operations in parallel. Spark provides two main abstractions for parallel programming: *resilient distributed datasets* and *parallel operations* on these datasets (invoked by passing a function to apply on a dataset). In addition, Spark supports two restricted types of *shared variables* that can be used in functions running on the cluster, which we shall explain later.

2.1 Resilient Distributed Datasets (RDDs)

A resilient distributed dataset (RDD) is a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost. The elements of an RDD need not exist in physical storage; instead, a handle to an RDD contains enough information to *compute* the RDD starting from data in reliable storage. This means that RDDs can always be reconstructed if nodes fail.

In Spark, each RDD is represented by a Scala object. Spark lets programmers construct RDDs in four ways:

- From a *file* in a shared file system, such as the Hadoop Distributed File System (HDFS).
- By *"parallelizing"* a Scala collection (e.g., an array) in the driver program, which means dividing it into a number of slices that will be sent to multiple nodes.
- By *transforming* an existing RDD. A dataset with elements of type A can be transformed into a dataset with elements of type B using an operation called *flatMap*, which passes each element through a user-provided function of type $A \Rightarrow List[B]$.[1] Other transformations can be expressed using *flatMap*, including *map* (pass elements through a function of type $A \Rightarrow B$) and *filter* (pick elements matching a predicate).
- By changing the *persistence* of an existing RDD. By default, RDDs are lazy and ephemeral. That is, partitions of a dataset are materialized on demand when they are used in a parallel operation (e.g., by passing a block of a file through a map function), and are discarded from memory after use.[2] However, a user can alter the persistence of an RDD through two actions:
  - The *cache* action leaves the dataset lazy, but hints that it should be kept in memory after the first time it is computed, because it will be reused.
  - The *save* action evaluates the dataset and writes it to a distributed filesystem such as HDFS. The saved version is used in future operations on it.

We note that our *cache* action is only a hint: if there is not enough memory in the cluster to cache all partitions of a dataset, Spark will recompute them when they are used. We chose this design so that Spark programs keep working (at reduced performance) if nodes fail or if a dataset is too big. This idea is loosely analogous to virtual memory.

We also plan to extend Spark to support other levels of persistence (e.g., in-memory replication across multiple nodes). Our goal is to let users trade off between the cost of storing an RDD, the speed of accessing it, the probability of losing part of it, and the cost of recomputing it.

2.2 Parallel Operations

Several parallel operations can be performed on RDDs:

- *reduce*: Combines dataset elements using an associative function to produce a result at the driver program.
- *collect*: Sends all elements of the dataset to the driver program. For example, an easy way to update an array in parallel is to parallelize, map and collect the array.
- *foreach*: Passes each element through a user provided function. This is only done for the side effects of the function (which might be to copy data to another system or to update a shared variable as explained below).

[1] *flatMap* has the same semantics as the *map* in MapReduce, but *map* is usually used to refer to a one-to-one function of type $A \Rightarrow B$ in Scala.
[2] This is how "distributed collections" function in DryadLINQ.

We note that Spark does not currently support a grouped reduce operation as in MapReduce; reduce results are only collected at one process (the driver).[3] We plan to support grouped reductions in the future using a "shuffle" transformation on distributed datasets, as described in Section 7. However, even using a single reducer is enough to express a variety of useful algorithms. For example, a recent paper on MapReduce for machine learning on multicore systems [8] implemented ten learning algorithms without supporting parallel reduction.

2.3 Shared Variables

Programmers invoke operations like *map*, *filter* and *reduce* by passing closures (functions) to Spark. As is typical in functional programming, these closures can refer to variables in the scope where they are created. Normally, when Spark runs a closure on a worker node, these variables are copied to the worker. However, Spark also lets programmers create two restricted types of *shared variables* to support two simple but common usage patterns:

- *Broadcast variables*: If a large read-only piece of data (e.g., a lookup table) is used in multiple parallel operations, it is preferable to distribute it to the workers only once instead of packaging it with every closure. Spark lets the programmer create a "broadcast variable" object that wraps the value and ensures that it is only copied to each worker once.
- *Accumulators*: These are variables that workers can only "add" to using an associative operation, and that only the driver can read. They can be used to implement counters as in MapReduce and to provide a more imperative syntax for parallel sums. Accumulators can be defined for any type that has an "add" operation and a "zero" value. Due to their "add-only" semantics, they are easy to make fault-tolerant.

3 Examples

We now show some sample Spark programs. Note that we omit variable types because Scala uses type inference, but Scala is statically typed and performs comparably to Java.

3.1 Text Search

Suppose that we wish to count the lines containing errors in a large log file stored in HDFS. This can be implemented by starting with a file dataset object as follows:

```
val file = spark.textFile("hdfs://...")
val errs = file.filter(_.contains("ERROR"))
val ones = errs.map(_ => 1)
val count = ones.reduce(_+_)
```

We first create a distributed dataset called `file` that represents the HDFS file as a collection of lines. We transform this dataset to create the set of lines containing "ERROR" (`errs`), and then map each line to a 1 and add up these ones using *reduce*. The arguments to *filter*, *map* and *reduce* are Scala syntax for function literals.

Note that `errs` and `ones` are lazy RDDs that are never materialized. Instead, when *reduce* is called, each worker node scans input blocks in a streaming manner to evaluate `ones`, adds these to perform a local reduce, and sends its local count to the driver. When used with lazy datasets in this manner, Spark closely emulates MapReduce.

Where Spark differs from other frameworks is that it can make some of the intermediate datasets persist across operations. For example, if wanted to reuse the `errs` dataset, we could create a cached RDD from it as follows:

```
val cachedErrs = errs.cache()
```

We would now be able to invoke parallel operations on `cachedErrs` or on datasets derived from it as usual, but nodes would cache partitions of `cachedErrs` in memory after the first time they compute them, greatly speeding up subsequent operations on it.

3.2 Logistic Regression

The following program implements logistic regression [3], an iterative classification algorithm that attempts to find a hyperplane w that best separates two sets of points. The algorithm performs 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. It thus benefits greatly from caching the data in memory across iterations. We do not explain logistic regression in detail, but we use it to show a few new Spark features.

[3] Local reductions are first performed at each node, however.

```
// Read points from a text file and cache them
val points = spark.textFile(...)
                  .map(parsePoint).cache()
// Initialize w to a random D-dimensional vector
var w = Vector.random(D)
// Run multiple iterations to update w
for (i <- 1 to ITERATIONS) {
  val gradient = spark.accumulator(new Vector(D))
  for (p <- points) { // Runs in parallel
    val s = (1/(1+exp(-p.y*(w dot p.x)))-1)*p.y
    gradient += s * p.x
  }
  w -= gradient.value
}
```

First, although we create an RDD called `points`, we process it by running a `for` loop over it. The `for` keyword in Scala is syntactic sugar for invoking the `foreach` method of a collection with the loop body as a closure. That is, the code `for(p <- points){body}` is equivalent to `points.foreach(p => {body})`. Therefore, we are invoking Spark's parallel `foreach` operation.

Second, to sum up the gradient, we use an accumulator variable called `gradient` (with a value of type $Vector$). Note that the loop adds to `gradient` using an overloaded `+=` operator. The combination of accumulators and `for` syntax allows Spark programs to look much like imperative serial programs. Indeed, this example differs from a serial version of logistic regression in only three lines.

3.3 Alternating Least Squares

Our final example is an algorithm called alternating least squares (ALS). ALS is used for collaborative filtering problems, such as predicting users' ratings for movies that they have not seen based on their movie rating history (as in the Netflix Challenge). Unlike our previous examples, ALS is CPU-intensive rather than data-intensive.

We briefly sketch ALS and refer the reader to [23] for details. Suppose that we wanted to predict the ratings of u users for m movies, and that we had a partially filled matrix R containing the known ratings for some user-movie pairs. ALS models R as the product of two matrices M and U of dimensions $m \times k$ and $k \times u$ respectively; that is, each user and each movie has a k-dimensional "feature vector" describing its characteristics, and a user's rating for a movie is the dot product of its feature vector and the movie's. ALS solves for M and U using the known ratings and then computes $M \times U$ to predict the unknown ones. This is done using the following iterative process:

1. Initialize M to a random value.
2. Optimize U given M to minimize error on R.
3. Optimize M given U to minimize error on R.
4. Repeat steps 2 and 3 until convergence.

ALS can be parallelized by updating different users / movies on each node in steps 2 and 3. However, because all of the steps use R, it is helpful to make R a broadcast variable so that it does not get re-sent to each node on each step. A Spark implementation of ALS that does is shown below. Note that we *parallelize* the collection `0 until u` (a Scala range object) and *collect* it to update each array:

```
val Rb = spark.broadcast(R)
for (i <- 1 to ITERATIONS) {
  U = spark.parallelize(0 until u)
           .map(j => updateUser(j, Rb, M))
           .collect()
  M = spark.parallelize(0 until m)
           .map(j => updateUser(j, Rb, U))
           .collect()
}
```

4 Implementation

Spark is built on top of Nexus [13], a "cluster operating system" that lets multiple parallel applications share a cluster in a fine-grained manner and provides an API for applications to launch tasks on a cluster. This allows Spark to run

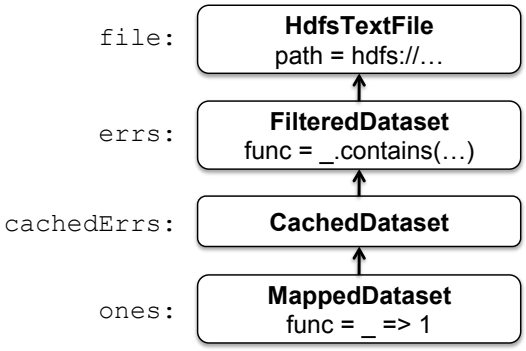

Figure 1: Lineage chain for the distributed dataset objects defined in the example in Section 4.

alongside existing cluster computing frameworks, such as Nexus ports of Hadoop and MPI, and share data with them. In addition, building on Nexus greatly reduced the programming effort that had to go into Spark.

The core of Spark is the implementation of resilient distributed datasets. As an example, suppose that we define a cached dataset called `cachedErrs` representing error messages in a log file, and that we count its elements using *map* and *reduce*, as in Section 3.1:

```
val file = spark.textFile("hdfs://...")
val errs = file.filter(_.contains("ERROR"))
val cachedErrs = errs.cache()
val ones = cachedErrs.map(_ => 1)
val count = ones.reduce(_+_)
```

These datasets will be stored as a chain of objects capturing the *lineage* of each RDD, shown in Figure 1. Each dataset object contains a pointer to its parent and information about how the parent was transformed.

Internally, each RDD object implements the same simple interface, which consists of three operations:

- *getPartitions*, which returns a list of partition IDs.
- *getIterator(partition)*, which iterates over a partition.
- *getPreferredLocations(partition)*, which is used for task scheduling to achieve data locality.

When a parallel operation is invoked on a dataset, Spark creates a *task* to process each partition of the dataset and sends these tasks to worker nodes. We try to send each task to one of its preferred locations using a technique called delay scheduling [22]. Once launched on a worker, each task calls *getIterator* to start reading its partition.

The different types of RDDs differ only in how they implement the RDD interface. For example, for a *HdfsTextFile*, the partitions are block IDs in HDFS, their preferred locations are the block locations, and *getIterator* opens a stream to read a block. In a *MappedDataset*, the partitions and preferred locations are the same as for the parent, but the iterator applies the map function to elements of the parent. Finally, in a *CachedDataset*, the *getIterator* method looks for a locally cached copy of a transformed partition, and each partition's preferred locations start out equal to the parent's preferred locations, but get updated after the partition is cached on some node to prefer reusing that node. This design makes faults easy to handle: if a node fails, its partitions are re-read from their parent datasets and eventually cached on other nodes.

Finally, shipping tasks to workers requires shipping closures to them—both the closures used to define a distributed dataset, and closures passed to operations such as *reduce*. To achieve this, we rely on the fact that Scala closures are Java objects and can be serialized using Java serialization; this is a feature of Scala that makes it relatively straightforward to send a computation to another machine. Scala's built-in closure implementation is not ideal, however, because we have found cases where a closure object references variables in the closure's outer scope that are not actually used in its body. We have filed a bug report about this, but in the meantime, we have solved the issue by performing a static analysis of closure classes' bytecode to detect these unused variables and set the corresponding fields in the closure object to `null`. We omit the details of this analysis due to lack of space.

Shared Variables: The two types of shared variables in Spark, broadcast variables and accumulators, are implemented using classes with custom serialization formats. When one creates a broadcast variable b with a value v, v is saved to a file in a shared file system. The serialized form of b is a path to this file. When b's value is queried on a

worker node, Spark first checks whether v is in a local cache, and reads it from the file system if it isn't. We initially used HDFS to broadcast variables, but we are developing a more efficient streaming broadcast system.

Accumulators are implemented using a different "serialization trick." Each accumulator is given a unique ID when it is created. When the accumulator is saved, its serialized form contains its ID and the "zero" value for its type. On the workers, a separate copy of the accumulator is created for each thread that runs a task using thread-local variables, and is reset to zero when a task begins. After each task runs, the worker sends a message to the driver program containing the updates it made to various accumulators. The driver applies updates from each partition of each operation only once to prevent double-counting when tasks are re-executed due to failures.

Interpreter Integration: Due to lack of space, we only sketch how we have integrated Spark into the Scala interpreter. The Scala interpreter normally operates by compiling a class for each line typed by the user. This class includes a singleton object that contains the variables or functions on that line and runs the line's code in its constructor. For example, if the user types `var x = 5` followed by `println(x)`, the interpreter defines a class (say `Line1`) containing `x` and causes the second line to compile to `println(Line1.getInstance().x)`. These classes are loaded into the JVM to run each line. To make the interpreter work with Spark, we made two changes:

1. We made the interpreter output the classes it defines to a shared filesystem, from which they can be loaded by the workers using a custom Java class loader.
2. We changed the generated code so that the singleton object for each line references the singleton objects for previous lines directly, rather than going through the static `getInstance` methods. This allows closures to capture the current state of the singletons they reference whenever they are serialized to be sent to a worker. If we had not done this, then updates to the singleton objects (e.g., a line setting `x = 7` in the example above) would not propagate to the workers.

5 Results

Although our implementation of Spark is still at an early stage, we relate the results of three experiments that show its promise as a cluster computing framework.

Logistic Regression: We compared the performance of the logistic regression job in Section 3.2 to an implementation of logistic regression for Hadoop, using a 29 GB dataset on 20 "m1.xlarge" EC2 nodes with 4 cores each. The results are shown in Figure 2. With Hadoop, each iteration takes 127s, because it runs as an independent MapReduce job. With Spark, the first iteration takes 174s (likely due to using Scala instead of Java), but subsequent iterations take only 6s, each because they reuse cached data. This allows the job to run up to 10x faster.

We have also tried crashing a node while the job was running. In the 10-iteration case, this slows the job down by 50s (21%) on average. The data partitions on the lost node are recomputed and cached in parallel on other nodes, but the recovery time was rather high in the current experiment because we used a high HDFS block size (128 MB), so there were only 12 blocks per node and the recovery process could not utilize all cores in the cluster. Smaller block sizes would yield faster recovery times.

Alternating Least Squares: We have implemented the alternating least squares job in Section 3.3 to measure the benefit of broadcast variables for iterative jobs that copy a shared dataset to multiple nodes. We found that without using broadcast variables, the time to resend the ratings matrix R on each iteration dominated the job's running time. Furthermore, with a naïve implementation of broadcast (using HDFS or NFS), the broadcast time grew linearly with the number of nodes, limiting the scalability of the job. We implemented an application-level multicast system to mitigate this. However, even with fast broadcast, resending R on each iteration is costly. Caching R in memory on the workers using a broadcast variable improved performance by 2.8x in an experiment with 5000 movies and 15000 users on a 30-node EC2 cluster.

Interactive Spark: We used the Spark interpreter to load a 39 GB dump of Wikipedia in memory across 15 "m1.xlarge" EC2 machines and query it interactively. The first time the dataset is queried, it takes roughly 35 seconds, comparable to running a Hadoop job on it. However, subsequent queries take only 0.5 to 1 seconds, even if they scan all the data. This provides a qualitatively different experience, comparable to working with local data.

6 Related Work

Distributed Shared Memory: Spark's resilient distributed datasets can be viewed as an abstraction for distributed shared memory (DSM), which has been studied extensively [17]. RDDs differ from DSM interfaces in two ways. First, RDDs provide a much more restricted programming model, but one that lets datasets be rebuilt efficiently if

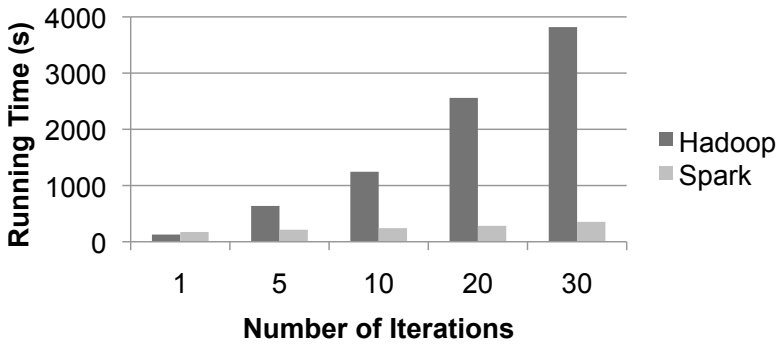

Figure 2: Logistic regression performance in Hadoop and Spark.

cluster nodes fail. While some DSM systems achieve fault tolerance through checkpointing [15], Spark reconstructs lost partitions of RDDs using lineage information captured in the RDD objects. This means that only the lost partitions need to be recomputed, and that they can be recomputed in parallel on different nodes, without requiring the program to revert to a checkpoint. In addition, there is no overhead if no nodes fail. Second, RDDs push computation to the data as in MapReduce [9], rather than letting arbitrary nodes access a global address space.

Other systems have also restricted the DSM programming model to improve performance, reliability and programmability. Munin [7] lets programmers annotate variables with the access pattern they will have so as to choose an optimal consistency protocol for them. Linda [11] provides a tuple space programming model that may be implemented in a fault-tolerant fashion. Thor [16] provides an interface to persistent shared objects.

Cluster Computing Frameworks: Spark's parallel operations fit into the MapReduce model [9]. However, they operate on RDDs that can persist *across* operations.

The need to extend MapReduce to support iterative jobs was also recognized by Twister [6, 10], a MapReduce framework that allows long-lived map tasks to keep static data in memory between jobs. However, Twister does not currently implement fault tolerance. Spark's abstraction of resilient distributed datasets is both fault-tolerant and more general than iterative MapReduce. A Spark program can define multiple RDDs and alternate between running operations on them, whereas a Twister program has only one map function and one reduce function. This also makes Spark useful for interactive data analysis, where a user can define several datasets and then query them.

Spark's broadcast variables provide a similar facility to Hadoop's distributed cache [2], which can disseminate a file to all nodes running a particular job. However, broadcast variables can be reused *across* parallel operations.

Language Integration: Spark's language integration is similar to that of DryadLINQ [21], which uses .NET's support for language integrated queries to capture an expression tree defining a query and run it on a cluster. Unlike DryadLINQ, Spark allows RDDs to persist in memory across parallel operations. In addition, Spark enriches the language integration model by supporting shared variables (broadcast variables and accumulators), implemented using classes with custom serialized forms.

We were inspired to use Scala for language integration by SMR [12], a Scala interface for Hadoop that uses closures to define map and reduce tasks. Our contributions over SMR are shared variables and a more robust implementation of closure serialization (described in Section 4).

Finally, IPython [19] is a Python interpreter for scientists that lets users launch computations on a cluster using a fault-tolerant task queue interface or low-level message passing interface. Spark provides a similar interactive interface, but focuses on data-intensive computations and uses a more efficient programming language (Scala).

7 Discussion and Future Work

Spark provides three simple data abstractions for programming clusters: resilient distributed datasets (RDDs), and two restricted types of shared variables: broadcast variables and accumulators. While these abstractions are limited, we have found that they are powerful enough to express several applications that pose challenges for existing cluster computing frameworks, including iterative and interactive computations. Furthermore, we believe that the core idea behind RDDs, of a dataset handle that has enough information to (re)construct the dataset from data available in reliable storage, may prove useful in developing other abstractions for programming clusters.

In future work, we plan to focus on four areas:

1. Formally characterize the properties of RDD and Spark's other abstractions, and their suitability for various classes of applications and workloads.
2. Enhance the RDD abstraction to allow programmers to trade between storage cost and re-construction cost.
3. Define new operations to transform RDDs, including "shuffle," which repartitions an RDD by a given key. Such an operation would allow us to implement group-by and join functionalities.
4. Provide higher-level interactive interfaces on top of the Spark interpreter, such as SQL and R [4] shells.

References

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

[2] Hadoop Map/Reduce tutorial. http://hadoop.apache.org/ common/docs/r0.20.0/mapred_tutorial.html.

[3] Logistic regression – Wikipedia. http://en.wikipedia.org/wiki/Logistic_regression.

[4] The R project for statistical computing. http://www.r-project.org.

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

[6] Twister: Iterative MapReduce. http://iterativemapreduce.org.

[7] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *SOSP '91*. ACM, 1991.

[8] 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.

[9] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[10] J. Ekanayake, S. Pallickara, and G. Fox. MapReduce for data intensive scientific analyses. In *ESCIENCE '08*, pages 277–284, Washington, DC, USA, 2008. IEEE Computer Society.

[11] D. Gelernter. Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985.

[12] D. Hall. A scalable language, and a scalable framework. http://www.scala-blogs.org/2008/09/scalable-language-and-scalable.html.

[13] B. Hindman, A. Konwinski, M. Zaharia, and I. Stoica. A common substrate for cluster computing. In *Workshop on Hot Topics in Cloud Computing (HotCloud) 2009*, 2009.

[14] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys 2007*, pages 59–72, 2007.

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

[16] B. Liskov, A. Adya, M. Castro, S. Ghemawat, R. Gruber, U. Maheshwari, A. C. Myers, M. Day, and L. Shrira. Safe and efficient sharing of persistent objects in thor. In *SIGMOD '96*, pages 318–329. ACM, 1996.

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

[18] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD '08*. ACM, 2008.

[19] F. Pérez and B. E. Granger. IPython: a system for interactive scientific computing. *Comput. Sci. Eng.*, 9(3):21–29, May 2007.

[20] H.-c. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *SIGMOD '07*, pages 1029–1040. ACM, 2007.

[21] 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*, San Diego, CA, 2008.

[22] 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 2010 (PDF at http://tinyurl.com/ydh9ysm)*, April 2010.

[23] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan. Large-scale parallel collaborative filtering for the Netflix prize. In *AAIM '08*, pages 337–348, Berlin, Heidelberg, 2008. Springer-Verlag.