# Spark: Cluster Computing with Working Sets

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

## Abstract

MapReduce and its variants have been highly successful in implementing large-scale data-intensive applications on commodity clusters. However, most of these systems are built around an acyclic data flow model that is not suitable for other popular applications. This paper focuses 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 tools. We propose a new framework called Spark that supports these applications while retaining the scalability and fault tolerance of MapReduce. To achieve these goals, Spark introduces an 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 [11] pioneered this model, while systems like Dryad [17] and Map-Reduce-Merge [24] 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 includes two use cases where we have seen Hadoop users report that MapReduce is deficient:

- **Iterative jobs**: Many common machine learning algorithms apply a function repeatedly to the same dataset to optimize a parameter (e.g., through gradient descent). While each iteration can be expressed as a MapReduce/Dryad job, each job must reload the data from disk, incurring a significant performance penalty.
- **Interactive analytics**: Hadoop is often used to run ad-hoc exploratory queries on large datasets, through SQL interfaces such as Pig [21] 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, which supports applications with working sets while providing similar scalability and fault tolerance properties to 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 is implemented in Scala [5], a statically typed high-level programming language for the Java VM, and exposes a functional programming interface similar to DryadLINQ [25]. 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).

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 [10] 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 vari-

[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.

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

able" 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 supports type inference.

## 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.

```
// Read points from a text file and cache them
val points = spark.textFile(...)
                  .map(parsePoint).cache()
// Initialize w to random D-dimensional vector
var w = Vector.random(D)
// Run multiple iterations to update w
for (i <- 1 to ITERATIONS) {
  val grad = 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
    grad += s * p.x
  }
  w -= grad.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 [27] 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 Mesos [16, 15], 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 alongside existing cluster computing frameworks, such as Mesos ports of Hadoop and MPI, and share data with them. In addition, building on Mesos 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 [26]. Once launched on a worker, each task calls *getIterator* to start reading its partition.

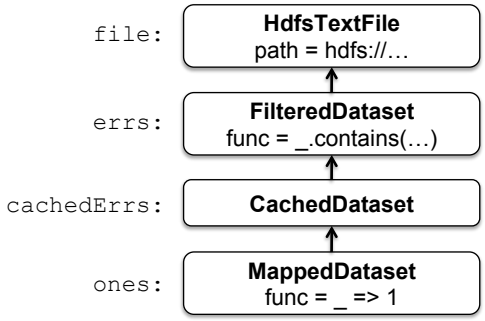



Figure 1: Lineage chain for the distributed dataset objects defined in the example in Section 4.

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

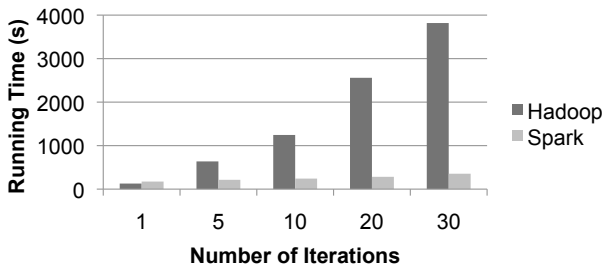



Figure 2: Logistic regression performance in Hadoop and Spark.

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 [20]. 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 cluster nodes fail. While some DSM systems achieve fault tolerance through checkpointing [18], 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 [11], 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 [8] lets programmers annotate variables with the access pattern they will have so as to choose an optimal consistency protocol for them. Linda [13] provides a tuple space programming model that may be implemented in a fault-tolerant fashion. Thor [19] provides an interface to persistent shared objects.

**Cluster Computing Frameworks:** Spark's parallel operations fit into the MapReduce model [11]. 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, 12], 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 [25], 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 [14], 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 [22] 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.

**Lineage:** Capturing lineage or provenance information for datasets has long been a research topic in the scientific computing an database fields, for applications such as explaining results, allowing them to be reproduced by others, and recomputing data if a bug is found in a workflow step or if a dataset is lost. We refer the reader to [7], [23] and [9] for surveys of this work. Spark provides a restricted parallel programming model where fine-grained lineage is inexpensive to capture, so that this information can be used to recompute lost dataset elements.

## 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 RDDs 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 a "shuffle" operation that repartitions an RDD by a given key. Such an operation would allow us to implement group-bys and joins.
4. Provide higher-level interactive interfaces on top of the Spark interpreter, such as SQL and R [4] shells.

## 8 Acknowledgements

We thank Ali Ghodsi for his feedback on this paper. 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!.

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

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

[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] 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.

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

[12] 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.

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

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

[15] 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. Technical Report UCB/EECS-2010-87, EECS Department, University of California, Berkeley, May 2010.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

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

[21] 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.

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

[23] Y. L. Simmhan, B. Plale, and D. Gannon. A survey of data provenance in e-science. *SIGMOD Rec.*, 34(3):31–36, 2005.

[24] 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.

[25] 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.

[26] 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*, April 2010.

[27] 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.

# Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters

Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, Ion Stoica
*University of California, Berkeley*

## Abstract

Many important "big data" applications need to process data arriving in real time. However, current programming models for distributed stream processing are relatively low-level, often leaving the user to worry about consistency of state across the system and fault recovery. Furthermore, the models that provide fault recovery do so in an expensive manner, requiring either hot replication or long recovery times. We propose a new programming model, *discretized streams (D-Streams)*, that offers a high-level functional programming API, strong consistency, and efficient fault recovery. D-Streams support a new recovery mechanism that improves efficiency over the traditional replication and upstream backup solutions in streaming databases: *parallel recovery* of lost state across the cluster. We have prototyped D-Streams in an extension to the Spark cluster computing framework called Spark Streaming, which lets users seamlessly intermix streaming, batch and interactive queries.

## 1 Introduction

Much of "big data" is received in real time, and is most valuable at its time of arrival. For example, a social network may want to identify trending conversation topics within minutes, an ad provider may want to train a model of which users click a new ad, and a service operator may want to mine log files to detect failures within seconds.

To handle the volumes of data and computation they involve, these applications need to be distributed over clusters. However, despite substantial work on cluster programming models for batch computation [6, 22], there are few similarly high-level tools for stream processing. Most current distributed stream processing systems, including Yahoo!'s S4 [19], Twitter's Storm [21], and streaming databases [2, 3, 4], are based on a *record-at-a-time* processing model, where nodes receive each record, update internal state, and send out new records in response. This model raises several challenges in a large-scale cloud environment:

- **Fault tolerance:** Record-at-a-time systems provide recovery through either *replication*, where there are two copies of each processing node, or *upstream backup*, where nodes buffer sent messages and replay them to a second copy of a failed downstream node. Neither approach is attractive in large clusters: replication needs $2\times$ the hardware and may not work if two nodes fail, while upstream backup takes a long time to recover, as the entire system must wait for the standby node to recover the failed node's state.
- **Consistency:** Depending on the system, it can be hard to reason about the global state, because different nodes may be processing data that arrived at different times. For example, suppose that a system counts page views from male users on one node and from females on another. If one of these nodes is backlogged, the ratio of their counters will be wrong.
- **Unification with batch processing:** Because the interface of streaming systems is event-driven, it is quite different from the APIs of batch systems, so users have to write two versions of each analytics task. In addition, it is difficult to *combine* streaming data with historical data, *e.g.,* join a stream of events against historical data to make a decision.

In this work, we present a new programming model, *discretized streams (D-Streams)*, that overcomes these challenges. The key idea behind D-Streams is to treat a streaming computation as a *series of deterministic batch computations on small time intervals*. For example, we might place the data received each second into a new interval, and run a MapReduce operation on each interval to compute a count. Similarly, we can perform a running count over several intervals by adding the new counts from each interval to the old result. Two immediate advantages of the D-Stream model are that consistency is well-defined (each record is processed atomically with the interval in which it arrives), and that the processing model is easy to unify with batch systems. In addition, as we shall show, we can use similar recovery mechanisms to batch systems, albeit at a much smaller timescale, to mitigate failures more efficiently than existing streaming systems, *i.e.,* recover data faster at a lower cost.

There are two key challenges in realizing the D-Stream model. The first is making the latency (interval granularity) low. Traditional batch systems like Hadoop and Dryad fall short here because they keep state on disk between jobs and take tens of seconds to run each

job. Instead, to meet a target latency of several seconds, we keep intermediate state in memory. However, simply putting the state into a general-purpose in-memory storage system, such as a key-value store [17], would be expensive due to the cost of data replication. Instead, we build on Resilient Distributed Datasets (RDDs) [23], a storage abstraction that can rebuild lost data *without* replication by tracking the operations needed to recompute it. Along with a fast execution engine (Spark [23]) that supports tasks as small as 100 ms, we show that we can achieve latencies as low as a second. We believe that this is sufficient for many real-world big data applications, where the timescale of events monitored (*e.g.,* trends in a social network) is much higher.

The second challenge is recovering quickly from failures. Here, we use the deterministic nature of the batch operations in each interval to provide a new recovery mechanism that has not been present in previous streaming systems: *parallel recovery* of a lost node's state. Each node in the cluster works to recompute part of the lost node's RDDs, resulting in faster recovery than upstream backup without the cost of replication. Parallel recovery was hard to implement in record-at-a-time systems due to the complex state maintenance protocols needed even for basic replication (*e.g.,* Flux [20]),[1] but is simple with the deterministic model of D-Streams.

We have prototyped D-Streams in Spark Streaming, an extension to the Spark cluster computing engine [23]. In addition to enabling low-latency stream processing, Spark Streaming interoperates cleanly with Spark's batch and interactive processing features, letting users run ad-hoc queries on arriving streams or mix streaming and historical data from the same high-level API.

## 2 Discretized Streams (D-Streams)

The key idea behind our model is to treat streaming computations as a series of deterministic batch computations on small time intervals. The input data received during each interval is stored reliably across the cluster to form an *input dataset* for that interval. Once the time interval completes, this dataset is processed via deterministic parallel operations, such as *map*, *reduce* and *groupBy*, to produce new datasets representing program outputs or intermediate state. We store these results in *resilient distributed datasets (RDDs)* [23], an efficient storage abstraction that avoids replication by using lineage for fault recovery, as we shall explain later.

A *discretized stream* or D-Stream groups together a *series* of RDDs and lets the user manipulate them to through various operators. D-Streams provide both *stateless* operators, such as *map*, which act independently on each time interval, and *stateful* operators, such as aggregation over a sliding window, which operate on multiple intervals and may produce intermediate RDDs as state.

[1] The one parallel recovery algorithm we are aware of, by Hwang *et al.* [11], only tolerates one node failure and cannot mitigate stragglers.

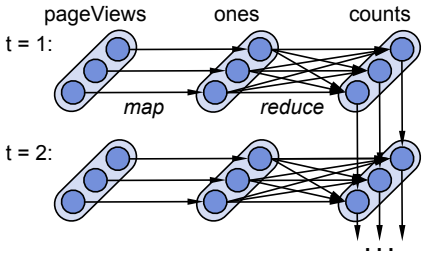



Figure 1: Lineage graph for the RDDs in the view count program. Each oblong shows an RDD, whose partitions are drawn as circles. Lineage is tracked at the granularity of partitions.

We illustrate the idea with a Spark Streaming program that computes a running count of page view events by URL. Spark Streaming provides a language-integrated API similar to DryadLINQ [22] in the Scala language. The code for the view count program is:

```
pageViews = readStream("http://...", "1s")
ones = pageViews.map(event => (event.url, 1))
counts = ones.runningReduce((a, b) => a + b)
```

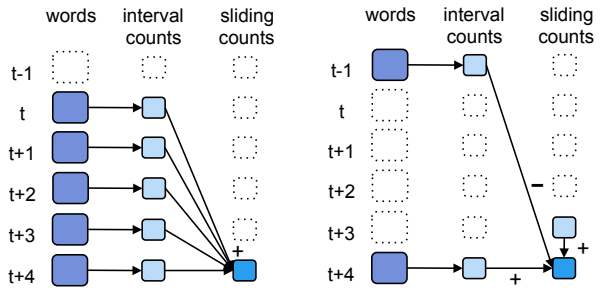
This code creates a D-Stream called `pageViews` by reading an event stream over HTTP, and groups it into 1-second intervals. It then transforms the event stream to get a D-Stream of (URL, 1) pairs called `ones`, and performs a running count of these using the *runningReduce* operator. The arguments to *map* and *runningReduce* are Scala syntax for a closure (function literal).

Finally, to recover from failures, both D-Streams and RDDs track their *lineage*, that is, the set of deterministic operations used to build them. We track this information as a dependency graph, similar to Figure 1. When a node fails, we recompute the RDD partitions that were on it by rerunning the *map*, *reduce*, etc. operations used to build them on the data still in the cluster. The system also periodically checkpoints state RDDs (*e.g.,* by replicating every fifth RDD) to prevent infinite recomputation, but this does not need to happen for all data, because recovery is often fast: the lost partitions can be recomputed *in parallel* on separate nodes, as we shall discuss in Section 3.

**D-Stream Operators** D-Streams provide two types of operators to let users build streaming programs:

- *Transformation* operators, which produce a new D-Stream from one or more parent streams. These can be either *stateless* (*i.e.,* act independently on each interval) or *stateful* (share data across intervals).
- *Output* operators, which let the program write data to external systems (*e.g.,* save each RDD to HDFS).

D-Streams support the same stateless transformations available in typical batch frameworks [6, 22], including *map*, *reduce*, *groupBy*, and *join*. We reused all of the op-




(a) Naïve evaluation (b) Incremental evaluation

Figure 2: Comparing the naïve variant of *reduceByWindow* (a) with the incremental variant for invertible functions (b). Both versions compute a per-interval count only once, but the second avoids re-summing each window. Boxes denote RDDs, while arrows show the operations used to compute window $[t, t+5)$.

erators in Spark [23]. For example, a program could run a canonical MapReduce word count on each time interval of a D-Stream of sentences using the following code:

```
words = sentences.flatMap(s => s.split(" "))
pairs = words.map(w => (w, 1))
counts = pairs.reduceByKey((a, b) => a + b)
```

In addition, D-Streams introduce new *stateful* operators that work over multiple intervals. These include:

- *Windowing:* The *window* operator groups all of the records from a range of past time intervals into a single RDD. For example, in our earlier code, calling `pairs.window("5s").reduceByKey(_+_)` yields a D-Stream of word counts on intervals $[0, 5)$, $[1, 6)$, $[2, 7)$, etc. *Window* is the most general stateful operator, but it is also often inefficient, as it repeats work.
- *Incremental aggregation:* For the common use case of computing an aggregate value, such as a count or sum, over a sliding window, D-Streams have several variants of a *reduceByWindow* operator. The simplest one only takes an associative "merge" operation for combining values. For example, one might write:

  ```
  pairs.reduceByWindow("5s", (a, b) => a + b)
  ```

  This computes a per-interval count for each time interval only once, but has to add the counts for the past five seconds repeatedly, as in Figure 2a. A more efficient version for *invertible* aggregation functions also takes a function for "subtracting" values and updates state incrementally (Figure 2b).
- *Time-skewed joins:* Users can join a stream against its own RDDs from some time in the past to compute trends—for example, how current page view counts compare to page views five minutes ago.

Finally, the user calls *output operators* to transfer results out of D-Streams into external systems (*e.g.,* for display on a dashboard). We provide two such operators: *save*, which writes each RDD in a D-Stream to a storage system,[2] and *foreach*, which runs a user code snippet (any Spark code) on each RDD in a stream. For example, a user can print the counts computed above with:

```
counts.foreach(rdd => println(rdd.collect()))
```

**Unification with Batch and Interactive Processing** Because D-Streams follow the same processing model as batch systems, the two can naturally be combined. Spark Streaming provides several powerful features to unify streaming and batch processing.

First, D-Streams can be combined with static RDDs computed, for example, by loading a file. For example, one might join a stream of incoming tweets against a precomputed spam filter, or compare it with historical data.

Second, users can run a D-Stream program as a batch job on previous historical data. This makes it easy compute a new streaming report on past data as well.

Third, users can attach a Scala console to a Spark Streaming program to run ad-hoc queries on D-Streams *interactively*, using Spark's existing fast interactive query capability [23]. For example, the user could query the most popular words in a time range by typing:

```
counts.slice("21:00", "21:05").topK(10)
```

The ability to quickly query any state in the system is invaluable for users troubleshooting issues in real time.

## 3 Fault Recovery

Classical streaming systems update mutable state on a per-record basis and use either replication or upstream backup for fault recovery [12].

The replication approach creates two or more copies of each process in the data flow graph [2, 20]. Supporting one node failure therefore doubles the hardware cost. Furthermore, if two nodes in the same replica fail, the system is not recoverable. For these reasons, replication is not cost-effective in our large-scale cloud setting.

In upstream backup [12], each upstream node buffers the data sent to downstream nodes locally until it gets an acknowledgement that all related computations are done. When a node fails, the upstream node retransmits all unacknowledged data to a standby node, which takes over the role of the failed node and reprocesses the data. The disadvantage of this approach is long recovery times, as the system must wait for the standby node to catch up.

To address these issues, D-Streams employ a new approach: *parallel recovery*. The system periodically checkpoints some of the state RDDs, by asynchronously replicating them to other nodes. For example, in a view count program computing hourly windows, the system

[2] We can use any storage system supported by Hadoop, *e.g.,* HDFS or HBase, by calling into Hadoop's I/O interfaces to these systems.

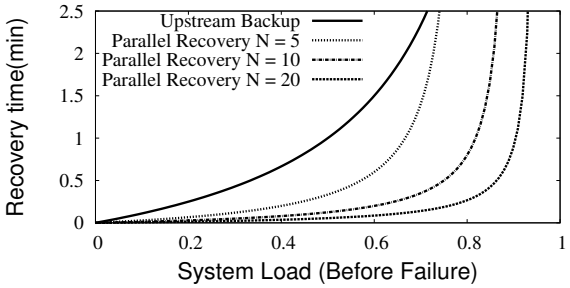



Figure 3: Recovery time for parallel recovery vs. upstream backup on $N$ nodes, as a function of the load before a failure.

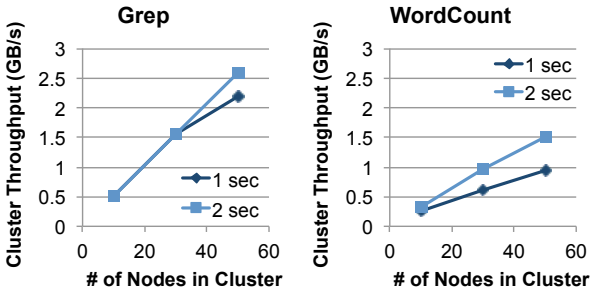



Figure 4: Performance of Grep and sliding WordCount on different cluster sizes. We show the maximum throughput attainable under an end-to-end latency below 1 second or 2 seconds.

could checkpoint results every minute. When a node fails, the system detects the missing RDD partitions and launches tasks to recover them from latest checkpoint. Many fine-grained tasks can be launched *at the same time* to compute different RDD partitions on different nodes. Thus, parallel recovery finishes faster than upstream backup, at a much lower cost than replication.

To show the benefit of this approach, we present results from a simple analytical model in Figure 3. The model assumes that the system is recovering from a minute-old checkpoint and that the bottleneck resource in the recovery process is CPU. In the upstream backup line, a single idle machine performs all of the recovery and then starts processing new records. It takes a long time to catch up at high system loads because new records for it continue to accumulate while it is rebuilding old state.[3] In the other lines, all of the machines partake in recovery, while also processing new records. With more nodes, parallel recovery catches up with the arriving stream much faster than upstream backup.[4]

One reason why parallel recovery was hard to perform in previous streaming systems is that they process data on a per-record basis, which requires complex and costly bookkeeping protocols (*e.g.,* Flux [20]) even for basic replication. In contrast, D-Streams apply deterministic transformations at the much coarser granularity of RDD partitions, which leads to far lighter bookkeeping and simple recovery similar to batch data flow systems [6].

Finally, beside node failures, another important concern in large clusters is stragglers [6]. Fortunately, D-Streams can also recover from stragglers in the same way as batch frameworks like MapReduce, by executing speculative backup copies of slow tasks. This type of speculation would again be difficult in a record-at-a-time system, but becomes simple with deterministic tasks.

[3]For example, when the load is 0.5, the standby node first has to spend 0.5 minutes to recompute the 1 minute of state lost since the checkpoint, then 0.25 minutes to process the data that arrived in those 0.5 minutes, then 0.125 minutes to process the data in this time, etc.

[4]Note that when the system's load before failure is high enough, it can never recover because the load exceeds the available resources.

## 4 Results

We implemented a prototype of Spark Streaming that extends the existing Spark runtime and can receive streams of data either from the network or from files periodically uploaded to HDFS. We briefly evaluated its scalability and fault recovery using experiments on Amazon EC2. We used nodes with 4 cores and 15 GB RAM.

**Scalability** We evaluated the scalability of the system through two applications: Grep, which counts input records matching a pattern, and WordCount, which performs a sliding window word count over 10 second windows. For both applications, we measured the maximum throughput achievable on different-sized clusters with an end-to-end latency target of either 1 or 2 seconds. By end-to-end latency, we mean the total time between when a record enters the system and when it appears in a result, including the time it waits for its batch to start. We used a batching interval of 0.5s and 100-byte records.

Figure 4 plots the results. We see that the system can process roughly 40 MB/second/node (400K records/s/node) for Grep and 20 MB/s/node (200K records/s/node) for WordCount at sub-second latency, as well as slightly more data if we allow 2s of latency. The system also scales nearly linearly to 50 nodes. The scaling is not perfect because there are more stragglers with more nodes.

**Parallel Recovery** We evaluated parallel fault recovery using two applications, both of which received 10 MB/s of data per node on 10 nodes, and used 2-second batching intervals. The first application, *MaxCount*, performed a word count in each 2-second interval, and computed the maximum count for each word over the past 30 seconds using a sliding window. Because max is not an invertible operation, we used the naïve `reduceByWindow` that recomputes every 2s. We ran this application both without any checkpointing (except for replication of the input data). Each interval took **1.66s** to process before the failure, whereas the average processing time of the interval during which a failure happened was **2.64s**

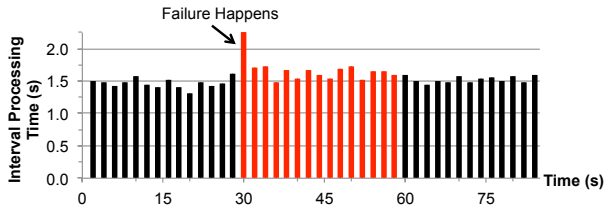



Figure 5: Processing time of intervals during one run of the *WordCount* job. After a failure, the jobs for the next 30s (shown in red) can take slightly longer than average because they may need to recompute the local counts from 30s ago.

(std.dev. 0.19s). Even though results dating back 30 seconds had to be recomputed, this was done in parallel, costing only one extra second of latency.

The second application performed a sliding word count with a 30s window using the the incremental `reduceByWindow` operator, and checkpointed data every 30s. Here, a failure-free interval took **1.47s**, while an interval with a failure took on average **2.31s** (std.dev. 0.43s). Recovery was faster than with *MaxCount* because each interval's output only depends on three previous RDDs (the total count for the previous interval, the local count for the current interval, and the local count for 30 seconds ago). However, one interesting effect was that *any* interval within the next 30s after a failure could exhibit a slowdown, because it might discover that part of the local counts for the interval 30s before it were lost. Figure 5 shows an example of this, where the interval at 30s, when the failure occurs, takes 2.26s to recover, but the intervals at times 32, 34, 46 and 48 also take slightly longer. We plan to eagerly recompute lost RDDs from the past to mitigate this.

## 5 Related Work

The seminal academic work on stream processing was in streaming databases such as Aurora, Borealis, Telegraph, and STREAM [4, 2, 3, 1]. These systems provide a SQL interface and achieve fault recovery through either replication (an active or passive standby for each node [2, 13]) or upstream backup [12]. We make two contributions over these systems. First, we provide a general programming interface, similar to DryadLINQ [22], instead of just SQL. Second, we provide a more efficient recovery mechanism: parallel recomputation of lost state. Parallel recovery is feasible due to the *deterministic* nature of D-Streams, which lets us recompute lost partitions on other nodes. In contrast, streaming DBs update mutable state for each incoming record, and thus require complex protocols for both replication (*e.g.,* Flux [20]) and upstream backup [12]. The only parallel recovery protocol we are aware of, by Hwang et al [11], only tolerates one node failure, and cannot handle stragglers.

In industry, most stream processing frameworks use a lower-level message passing interface, where users write stateful code to process records in a queue. Examples include S4, Storm, and Flume [19, 21, 7]. These systems generally guarantee at-least-once message delivery, but unlike D-Streams, they require the user to manually handle state recovery on failures (*e.g.,* by keeping all state in a replicated database) and consistency across nodes.

Several recent research systems have looked at online processing in clusters. MapReduce Online [5] is a streaming Hadoop runtime, but cannot compose multiple MapReduce steps into a query or recover stateful reduce tasks. iMR [15] is an in-situ MapReduce engine for log processing, but does not support more general computation graphs and can lose data on failure. CBP [14] and Comet [10] provide "bulk incremental processing" by running MapReduce jobs on new data every few minutes to update state in a distributed file system; however, they incur the high overhead of replicated on-disk storage. In contrast, D-Streams can keep state in memory without costly replication, and achieve order of magnitude lower latencies. Naiad [16] runs computations incrementally, but does not yet have a cluster implementation or a discussion of fault tolerance. Percolator [18] performs incremental computations using triggers, but does not offer consistency guarantees across nodes or high-level operators like *map* and *join*.

Finally, our parallel recovery mechanism is conceptually similar to recovery techniques in MapReduce, GFS, and RAMCloud [6, 9, 17], which all leverage repartitioning. Our contribution is to show that this mechanism can be applied on small enough timescales for stream processing. In addition, unlike GFS and RAMCloud, we *recompute* lost data instead of having to replicate all data, avoiding the network and storage cost of replication.

## 6 Conclusion

We have presented discretized streams (D-Streams), a stream programming model for large clusters that provides consistency, efficient fault recovery, and powerful integration with batch systems. The key idea is to treat streaming as a series of short batch jobs, and bring down the latency of these jobs as much as possible. This brings many of the benefits of batch processing models to stream processing, including clear consistency semantics and a new parallel recovery technique that we believe is the first truly cost-efficient recovery technique for stream processing in large clusters. Our implementation, Spark Streaming, lets users seamlessly intermix streaming, batch and interactive queries.

In future work, we plan to use Spark Streaming to build integrated systems that combine these types of processing [8], and to further explore the limits of D-Streams. In particular, we are interested in pushing the latency even lower (to about 100 ms) and in recovering from failures faster by providing approximate results.

## 7 Acknowledgements

This research is supported in part by an NSF CISE Expeditions award, gifts from Google, SAP, Amazon Web Services, Blue Goji, Cisco, Cloudera, Ericsson, General Electric, Hewlett Packard, Huawei, Intel, MarkLogic, Microsoft, NetApp, Oracle, Quanta, Splunk, and VMware, by DARPA (contract #FA8650-11-C-7136), and by a Google PhD Fellowship.

## References

[1] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein, and J. Widom. STREAM: The Stanford stream data management system. *SIGMOD*, 2003.

[2] M. Balazinska, H. Balakrishnan, S. R. Madden, and M. Stonebraker. Fault-tolerance in the Borealis distributed stream processing system. *ACM Trans. Database Syst.*, 2008.

[3] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[4] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. B. Zdonik. Scalable distributed stream processing. In *CIDR*, 2003.

[5] T. Condie, N. Conway, P. Alvaro, and J. M. Hellerstein. MapReduce online. *NSDI*, 2010.

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

[7] Apache Flume. http://incubator.apache.org/flume/.

[8] M. Franklin, S. Krishnamurthy, N. Conway, A. Li, A. Russakovsky, and N. Thombre. Continuous analytics: Rethinking query processing in a network-effect world. *CIDR*, 2009.

[9] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *Proceedings of SOSP '03*, 2003.

[10] 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*.

[11] J. Hwang, Y. Xing, and S. Zdonik. A cooperative, self-configuring high-availability solution for stream processing. In *ICDE*, 2007.

[12] J.-H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik. High-availability algorithms for distributed stream processing. In *ICDE*, 2005.

[13] S. Krishnamurthy, M. Franklin, J. Davis, D. Farina, P. Golovko, A. Li, and N. Thombre. Continuous analytics over discontinuous streams. In *SIGMOD*, 2010.

[14] D. Logothetis, C. Olston, B. Reed, K. C. Webb, and K. Yocum. Stateful bulk processing for incremental analytics. In *SoCC*, 2010.

[15] D. Logothetis, C. Trezzo, K. C. Webb, and K. Yocum. In-situ MapReduce for log processing. In *USENIX ATC*, 2011.

[16] F. McSherry, R. Isaacs, M. Isard, and D. G. Murray. Naiad: The animating spirit of rivers and streams. In *SOSP Poster Session*, 2011.

[17] D. Ongaro, S. M. Rumble, R. Stutsman, J. K. Ousterhout, and M. Rosenblum. Fast crash recovery in RAMCloud. In *SOSP*, 2011.

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

[19] Apache S4. http://incubator.apache.org/s4/.

[20] M. Shah, J. Hellerstein, and E. Brewer. Highly available, fault-tolerant, parallel dataflows. *SIGMOD*, 2004.

[21] Storm. https://github.com/nathanmarz/storm/wiki.

[22] 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.

[23] 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. In *NSDI*, 2012.

# 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.

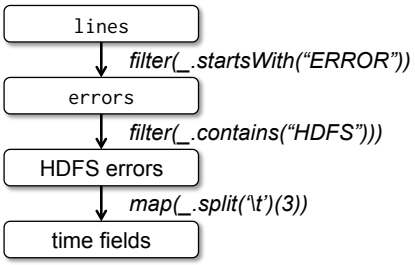



Figure 1: Lineage graph for the third query in our example. Boxes represent RDDs and arrows represent transformations.

```
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.

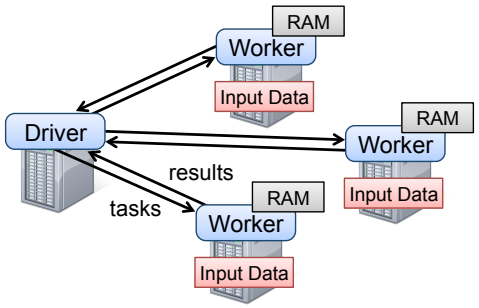



Figure 2: Spark runtime. The user's driver program launches multiple workers, which read data blocks from a distributed file system and can persist computed RDD partitions in memory.

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
```

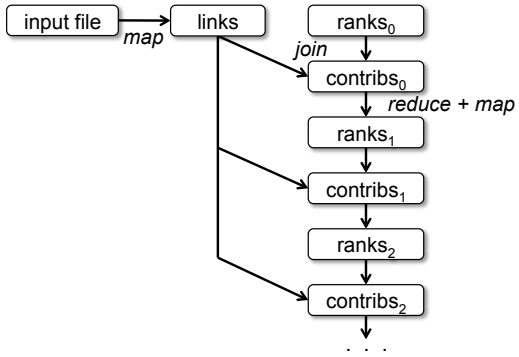



Figure 3: Lineage graph for datasets in PageRank.

```
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

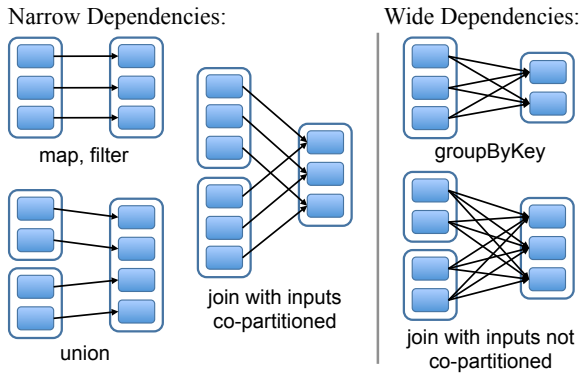



Figure 4: Examples of narrow and wide dependencies. Each box is an RDD, with partitions shown as shaded rectangles.

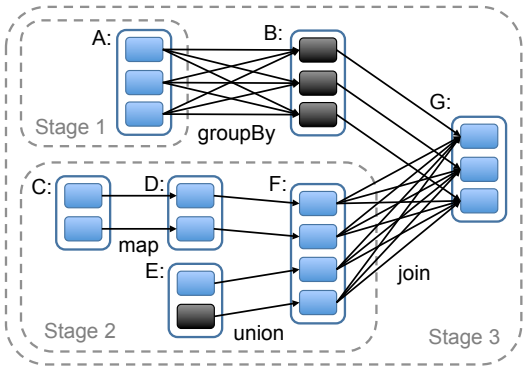



Figure 5: Example of how Spark computes job stages. Boxes with solid outlines are RDDs. Partitions are shaded rectangles, in black if they are already in memory. To run an action on RDD G, we build build stages at wide dependencies and pipeline narrow transformations inside each stage. In this case,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

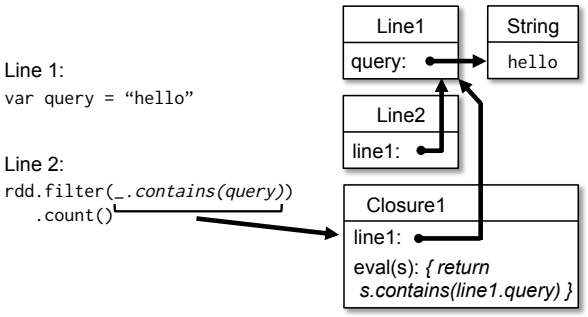
Our scheduler assigns tasks to machines based on data locality using delay scheduling [32]. If a task needs to process a partition that is available in memory on a node, we send it to that node. Otherwise, if a task processes a partition for which the containing RDD provides preferred locations (*e.g.,* 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.

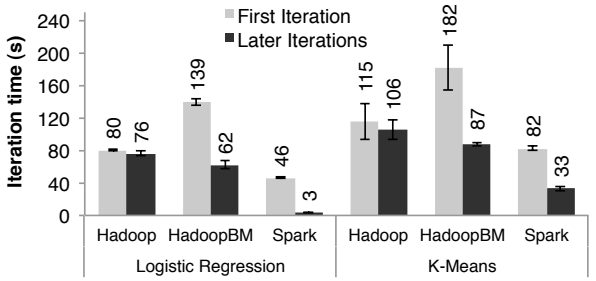
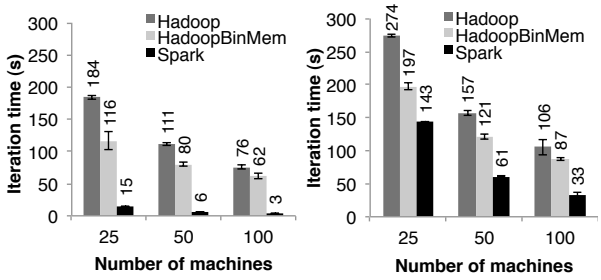



Figure 7: Duration of the first and later iterations in Hadoop, HadoopBinMem and Spark for logistic regression and k-means using 100 GB of data on a 100-node cluster.




(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

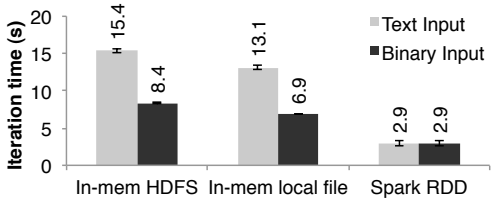



Figure 9: Iteration times for logistic regression using 256 MB data on a single machine for different sources of input.

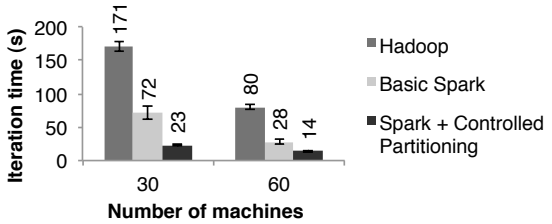



Figure 10: Performance of PageRank on Hadoop and Spark.

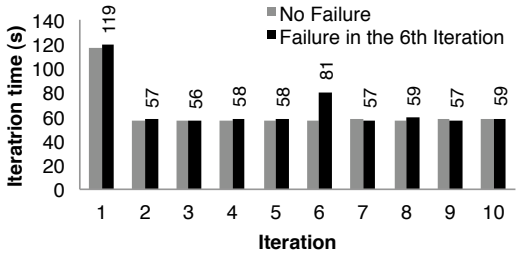



Figure 11: Iteration times for k-means in presence of a failure. One machine was killed at the start of the 6th iteration, resulting in partial reconstruction of an RDD using lineage.

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-

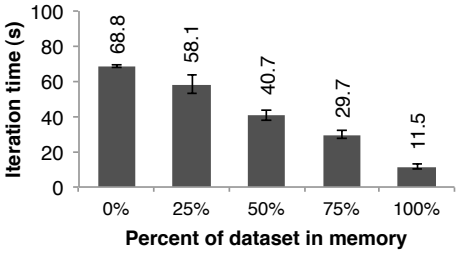
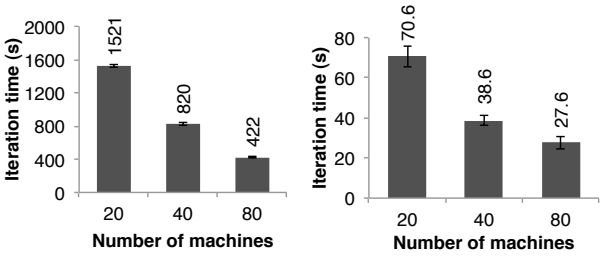



Figure 12: Performance of logistic regression using 100 GB data on 25 machines with varying amounts of data in memory.




(a) Traffic modeling (b) Spam classification

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

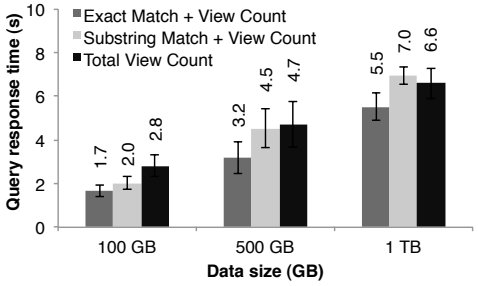



Figure 14: Response times for interactive queries on Spark, scanning increasingly larger input datasets on 100 machines.

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.

# Shark: Fast Data Analysis Using Coarse-grained Distributed Memory

Cliff Engle, Antonio Lupher, Reynold Xin, Matei Zaharia,
Michael J. Franklin, Scott Shenker, Ion Stoica

AMPLab, EECS, UC Berkeley
{cengle, alupher, rxin, matei, franklin, shenker, istoica}@cs.berkeley.edu

## ABSTRACT

Shark is a research data analysis system built on a novel coarse-grained distributed shared-memory abstraction. Shark marries query processing with deep data analysis, providing a unified system for easy data manipulation using SQL and pushing sophisticated analysis closer to data. It scales to thousands of nodes in a fault-tolerant manner. Shark can answer queries 40X faster than Apache Hive and run machine learning programs 25X faster than MapReduce programs in Apache Hadoop on large datasets.

## Categories and Subject Descriptors

H.2 [**Database Management**]: Systems

## General Terms

DESIGN, MANAGEMENT

## Keywords

Databases, Data Warehouse, Machine Learning, Resilient Distributed Dataset, Spark, Shark

## 1. INTRODUCTION

Modern data analysis employs statistical methods that go well beyond the roll-up and drill-down capabilities provided by traditional enterprise data warehouse (EDW) solutions. Data scientists appreciate the ability to use SQL for simple data manipulation but rely on other systems for machine learning on these data. What is needed is a system that consolidates both. For sophisticated data analysis at scale, it is important to exploit in-memory computation. This is particularly true with machine learning algorithms that are iterative in nature and exhibit strong temporal locality. Main-memory database systems use a *fine-grained* memory abstraction in which records can be updated individually. This fine-grained approach is difficult to scale to hundreds or thousands of nodes in a fault-tolerant manner for massive datasets. In contrast, a *coarse-grained abstraction*, in which transformations are performed on an entire collection of data, has been shown to scale more easily [1].


*SIGMOD '12,* May 20–24, 2012, Scottsdale, Arizona, USA.


### 1.1 Coarse-grained Distributed Memory

We have previously proposed a new distributed memory abstraction, Resilient Distributed Datasets (RDDs) [4], for in-memory computations on large clusters. RDDs provide a restricted form of shared memory, based on coarse-grained transformations on immutable collections of records rather than fine-grained updates to shared states. RDDs can be made fault-tolerant based on lineage information rather than replication. When the workload exhibits temporal locality, programs written using RDDs outperform systems such as MapReduce by orders of magnitude. Surprisingly, although restrictive, RDDs have been shown to be expressive enough to capture a wide class of computations, ranging from more general models like MapReduce to more specialized models such as Pregel.

It might seem counterintuitive to expect memory-based solutions to help when petabyte-scale data warehouses prevail. However, it is unlikely, for example, that an entire EDW fact table is needed to answer most queries. Queries usually focus on a particular subset or time window, *e.g.,* http logs from the previous month, touching only the (small) dimension tables and a small portion of the fact table. Thus, in many cases it is plausible to fit the working set into a cluster's memory. In fact, one study [1] analyzed the access patterns in the Hive warehouses at Facebook and discovered that for the vast majority (96%) of jobs, the entire inputs could fit into a fraction of the cluster's total memory.

### 1.2 Introducing Shark

The goal of the Shark (Hive on Spark) project is to design a data warehouse system capable of deep data analysis using the RDD memory abstraction. It unifies the SQL query processing engine with analytical algorithms based on this common abstraction, allowing the two to run in the same set of workers and share intermediate data.

Apart from the ability to run deep analysis, Shark is much more flexible and scalable compared with EDW solutions. Data need not be extracted, transformed, and loaded into the rigid relational form before analysis. Since RDDs are designed to scale horizontally, it is easy to add or remove nodes to accommodate more data or faster query processing. The

[1]MapReduce is an example of coarse-grained updates as the same map and reduce functions are executed on all records.

system scales out to thousands of nodes in a fault-tolerant manner. It can recover from node failures gracefully without terminating running queries and machine learning functions.

Compared with disk-oriented warehouse solutions and batch infrastructures such as Apache Hive [3], Shark excels at ad-hoc, exploratory queries by exploiting inter-query temporal locality and also leverages the intra-query locality inherent in iterative machine learning algorithms. By efficiently exploiting a cluster's memory using RDDs, queries previously taking minutes or hours to run can now return in seconds. This significant reduction in time is achieved by caching the working set of data in a cluster's memory, eliminating the need to repeatedly read from and write to disk.

In the remainder of this demonstration proposal, we sketch Shark's system design and give a brief overview of the system's performance. Finally, we describe in detail how we plan to demonstrate Shark at SIGMOD. Due to space constraints, we refer readers to [4] for more details on RDDs.

## 2. SYSTEM OVERVIEW

For ease of adoption, we have designed Shark to be entirely hot-swappable with Hive. Anyone can have Shark up and running in an existing Hive warehouse. Queries will return the same set of results in Shark, albeit much faster, without any modification to data or the queries themselves.

We have implemented Shark using Spark, a system that provides the RDD abstraction through a language-integrated API in Scala (a statically typed functional programming language for the Java VM). Each RDD dataset is represented as a Scala object, while the transformations are invoked using methods on those objects. A Shark cluster consists of masters and workers. A master's lifetime can span one or several queries. The workers are long-lived processes that can store dataset partitions in memory across operations. When the user runs a query, the client connects to a master, which defines RDDs for the workers and invokes operations on them.

Figure 1 shows the general architecture of Shark. Data are stored physically in the underlying distributed file system HDFS. The Hive metastore is used without modification in Shark and tracks the metadata and statistics about tables, much like the system catalog found in traditional RDBMS.

### 2.1 Query Processing

From a higher level, Shark's query execution consists of three steps similar to traditional RDBMS: query parsing, logical plan generation, and physical plan generation. Hive provides a SQL-like declarative query language, HiveQL, which gets compiled into lower level operators that are executed in a sequence of MapReduce programs. Shark uses Hive as a third-party Java library for query parsing and logical plan generation. The main difference is in the physical plan execution, as Shark has its own operators written specifically to exploit the benefits provided by RDDs.

Given a HiveQL query, the Hive query compiler is used to parse the query and generate an abstract syntax tree. The tree is then turned into the logical plan and basic logical optimization such as predicate pushdown is applied. Up to this point, Shark and Hive share an identical approach. In Hive, this operator tree would then be converted into a physical plan that consists of subtrees for separate map reduce tasks. In contrast, in Shark, this operator tree is converted into operators that perform transformations on

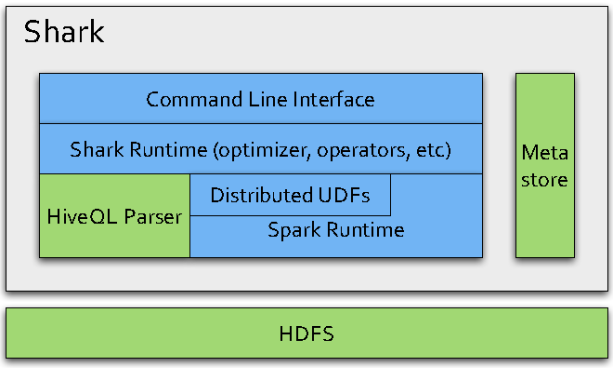



**Figure 1: Shark Architecture**

RDDs. An iterator traverses the operator tree and produces an immutable RDD for each operator on the tree. This RDD is not materialized until the execution engine returns the query results.

Much of the common structure of Hive operators is retained in order to ensure interoperability with HiveQL. We have currently implemented support for all of the essential Hive operators. We reuse all of Hive's data model and type system, in addition to supporting Hive's user-defined functions, user-defined aggregate functions, custom types, and custom serialization/deserialization methods.

A major advantage of Shark over Hive is the inter-query caching of data. The Spark framework provides a simple mechanism to cache RDDs in memory across clusters and recompute RDDs in the event of failures.

In the Shark configuration file, users can specify types of operators whose outputs will be cached automatically by Shark. Each output is cached as an in-memory RDD and Shark computes a signature for this RDD based on the subtree of the query plan. Signatures are computed for subsequent query subtrees and are compared with those of the in-memory RDDs, and in case of a match, the in-memory RDD is used to replace the subtree. For example, in order to force Shark to naively cache all input data, users can enable caching for the TableScan operator.

In addition, users can choose to explicitly cache certain data by using the CREATE TABLE AS SELECT statement. If the name of the table created using this statement ends with "_cached", the table is automatically cached as an in-memory RDD and can be used to answer subsequent queries. For example, the following clause creates an in-memory table:

```
CREATE TABLE top_employee_cached AS
SELECT employee_id, name, salary
FROM employee_data WHERE salary > 200000;
```

We have implemented an LRU cache replacement policy at RDD granularity, and are currently exploring more sophisticated algorithms that perform cost-based analysis for intermediate data. We are also investigating how to answer queries using in-memory RDDs by rewriting the queries given by users.

### 2.2 Deep Data Analysis

Shark provides a streamlined interface to marry deep data analysis with SQL query processing. It combines SQL's convenience in data manipulation with sophisticated analysis

using machine learning algorithms. The analytical algorithms run in the same set of workers as the query processing engine and can reuse intermediate data in the form of RDDs created by the engine.

Shark allows users to write user-defined functions (UDFs) in Spark to express their algorithms for distributed computation. Users can integrate these functions with SQL using a special kind of table-valued UDF. Shark provides a simple API for these UDFs, which accept a *Table* RDD as input and emit a *Table* RDD as output.

We have already implemented a number of basic machine learning algorithms, *e.g.,* linear regression, logistics regression, k-means. In most cases, the user only needs to implement a UDF that transforms the input *Table* RDD into the desired input data type for the selected algorithm and transforms the output from the algorithm into a separate *Table* RDD.

The following example illustrates a distributed UDF implementing k-means clustering in Shark. The `kmeans_core` function does the iterative k-means computation that partitions n points into k clusters represented by the centroids.

```
def kmeans_core(points: RDD[Point], k: Int) = {
  // Initialize the centroids.
  clusters = new HashMap[Int, Point]
  for (i <- 0 until k) centroids(i) = Point.random()

  for (i <- 1 until 10) {
    // Assign points to centroids and update centroids.
    clusters = points.groupBy(closestCentroid)
      .map{
         (id, points) => (id, points.sum / points.size)
      }.collectMap()
  }
}
```

Note that since the output of the UDFs is also an RDD, the system is in closed form and the UDF output can be further processed by other Shark operators or analysis algorithms.

# 3. PERFORMANCE

This section briefly discusses Shark's performance, including query processing and iterative machine learning execution. We also report experience from an alpha testing user. Note that the intent is to demonstrate the benefits in real applications, rather than to give an extensive, scientific performance study.

## 3.1 Query Processing

We used the teragen program from [2] to generate a 50GB dataset and experiment with a simple grep query on a 10-node Hive cluster and a 10-node Shark cluster. The 50GB of input data fits in the cluster's memory.

```
SELECT * FROM grep WHERE field LIKE '%XYZ%';
```

Figure 3.1 reports the execution time of running the query in Hive, in Shark when the data are not cached, and in Shark when input is cached. It is clear that Shark provides at least an order of magnitude speedup when we need to run several queries on the same working set of data.

## 3.2 Iterative Machine Learning

Many machine learning algorithms are iterative in nature because they run iterative optimization procedures, such as gradient descent, to optimize an objective function. These

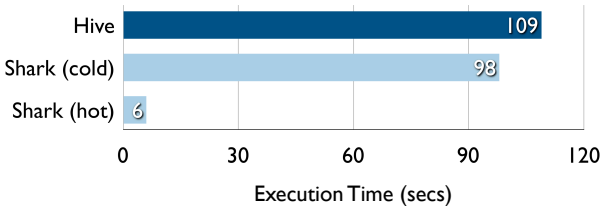



**Figure 2: Grep Query Performance**

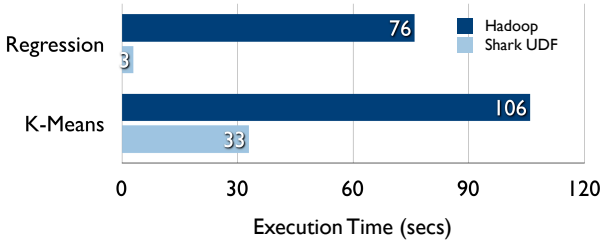



**Figure 3: Logistic Regression and K-Means Performance**

algorithms can be sped up substantially using Shark if their working set fits into RAM across a cluster. Furthermore, these algorithms often employ bulk operations like maps and sums, making them easy to express with RDDs in UDFs.

We implemented two iterative machine learning algorithms, logistic regression and k-means, in Hadoop MapReduce and in Shark distributed UDFs. We ran both algorithms for 10 iterations on 100 GB datasets using 100 machines. The key difference between the two algorithms 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.

Note that by the time the user runs the machine learning algorithms in Shark, the working set are likely to be already in memory through the SQL manipulations. Since typical learning algorithms need tens of iterations to converge, we report times for the subsequent iterations. Figure 3.1 shows that sharing data via RDDs greatly speeds up future iterations.

For logistic regression, Shark was 25.3× faster than Hadoop on 100 machines. For the more compute-intensive k-means application, Shark still achieved speedup of 3.2×. It is also worth noting that the implementation of these algorithms are much more concise as Shark UDFs than as Hadoop MapReduce programs.

## 3.3 Conviva Data Warehouse

Conviva Inc, a video distribution company, runs a 20 node Hive warehouse for data analytics. They have two types of queries: predefined reporting queries and ad-hoc debugging queries.

Their reporting queries mostly work on the same subset of the data (records matching a customer-provided predicate), but perform aggregations (averages, percentiles, and `COUNT DISTINCT`) over different grouping fields, requiring separate

MapReduce jobs. A typical reporting query takes 20 hours to run on 200 GB of compressed data. They experimented with an earlier prototype of Shark that required the developers to hand code the query plans. The query now runs in 30 mins using only two nodes with 96GB of RAM. This is a 40× improvement in query runtime, using only 10% of the hardware resources. The speedup comes from a combination of keeping the columns of interest in memory and avoiding repeated decompression and filtering of the same data files.

Conviva is now running approximately 30% of its reporting queries on the earlier prototype of Shark instead of Hive, but this requires manual porting of SQL queries. With the new version, Conviva doesn't need to rewrite their queries and will be able to achieve the same performance gains.

In addition, a number of users at Conviva use Hive interactively for debugging *e.g.,* finding commonalities between users who experienced low video quality to identify misconfigurations and software bugs. Like the reporting queries, these queries repeatedly access and refine the same dataset, so running them over Shark would greatly reduce the length of debug cycles.

# 4. DEMONSTRATION DETAILS

## 4.1 Demo Setup

We will present an end-to-end implementation of Shark and demonstrate the benefits it provides. The demonstration exhibits three main components:

**Data Set**: We will have a HDFS cluster on Amazon EC2 hosting one terabyte (uncompressed) of tweets. It contains roughly 400 million tweets, collected using the Twitter streaming API prior to the conference. These tweets are stored using a nested JSON format.

**Shark and Hive Clusters**: During the demo, we will have a 100-node Shark cluster and an equal-sized Apache Hive cluster running on EC2 for side-by-side comparison.

**Web Console**: Since it is hard to capture the internals of query processing given only the query and the output, we have implemented a web console that illustrates the query plan, the caching of RDDs for multi-query optimization, and status of each node. In addition, to demonstrate Shark's fault-tolerance feature, we will arbitrarily submit kill signals to Spark nodes from the web console.

## 4.2 Story Line

Imagine a situation in which a data journalist is exploring new topics for a story. At her disposal is a large collection of tweets.

She first generates a histogram of the number of tweets and realizes there is a spike in early October, 2011. Since Twitter activities usually correlate with events, she would like to gain insight into these events by understanding the causes of the spike. To do so, she drills down to focus on the particular two weeks, clustering tweets by their hash tags. She then realizes that there is a large cluster representing the Occupy Wall Street movement.

Suppose she then decides to write a piece about the movement and wants to investigate the public's sentiment toward this movement. Since it is unlikely for the world to have a unanimous view on this particular issue, she would like to compare the distribution of sentiments across different cities: *e.g.,* how are people in San Francisco similar to those in New York City? She achieves this by grouping the tweets in this window by geographical location and running the sentiment analysis algorithm. She then uses statistical distribution comparison algorithms to compare the sentiments.

SIGMOD attendees will be able to run queries outlined above as well as perform ad-hoc exploration of the dataset through the command-line interface.

# 5. TAKE-AWAY MESSAGE

This demonstration highlights the benefits of a coarse-grained distributed memory abstraction in allowing deep analysis and interactive querying of massive datasets. Our working prototype provides optimized execution of ad-hoc, exploratory queries that exploit inter-query temporal locality. It additionally provides efficient execution for iterative algorithms that exhibit intra-query temporal locality. We demonstrate Shark's scalability, fault-tolerance and high performance on a realistic analytical workload, while comparing it to Hive. Users will observe the ease of combining deep analysis with SQL, demonstrating how a unified system allows the reuse of intermediate data and significantly improves the performance of analytical queries on massive datasets.

# 6. ACKNOWLEDGMENTS

We would like to thank Peter Alvaro, Eric Yi Liu, Tim Kraska, Gene Pang, and Andrew Wang for feedback.

This research is supported in part by gifts from Google, SAP, Amazon Web Services, Blue Goji, Cloudera, Ericsson, General Electric, Hewlett Packard, Huawei, IBM, Intel, MarkLogic, Microsoft, NEC Labs, NetApp, Oracle, Quanta, Splunk, VMware and by DARPA (contract #FA8650-11-C-7136).

# 7. REFERENCES

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

[2] A. Pavlo, E. Paulson, A. Rasin, D. Abadi, D. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the 35th SIGMOD international conference on Management of data*, pages 165–178. ACM, 2009.

[3] A. Thusoo, J. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy. Hive-a petabyte scale data warehouse using hadoop. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 996–1005. IEEE, 2010.

[4] 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. In *NSDI 2012*.

# Shark: SQL and Rich Analytics at Scale

Reynold S. Xin, Josh Rosen, Matei Zaharia,
Michael J. Franklin, Scott Shenker, Ion Stoica

AMPLab, EECS, UC Berkeley
{rxin, joshrosen, matei, franklin, shenker, istoica}@cs.berkeley.edu

## ABSTRACT

Shark is a new data analysis system that marries query processing with complex analytics on large clusters. It leverages a novel distributed memory abstraction to provide a unified engine that can run SQL queries and sophisticated analytics functions (*e.g.,* iterative machine learning) at scale, and efficiently recovers from failures mid-query. This allows Shark to run SQL queries up to 100× faster than Apache Hive, and machine learning programs more than 100× faster than Hadoop. Unlike previous systems, Shark shows that it is possible to achieve these speedups while retaining a MapReduce-like execution engine, and the fine-grained fault tolerance properties that such engine provides. It extends such an engine in several ways, including column-oriented in-memory storage and dynamic mid-query replanning, to effectively execute SQL. The result is a system that matches the speedups reported for MPP analytic databases over MapReduce, while offering fault tolerance properties and complex analytics capabilities that they lack.

## Categories and Subject Descriptors

H.2 [**Database Management**]: Systems

## Keywords

Databases; Data Warehouse; Machine Learning; Spark; Shark; Hadoop

## 1 Introduction

Modern data analysis faces a confluence of growing challenges. First, data volumes are expanding dramatically, creating the need to scale out across clusters of hundreds of commodity machines. Second, such high scale increases the incidence of faults and stragglers (slow tasks), complicating parallel database design. Third, the *complexity* of data analysis has also grown: modern data analysis employs sophisticated statistical methods, such as machine learning algorithms, that go well beyond the roll-up and drill-down capabilities of traditional enterprise data warehouse systems. Finally, despite these increases in scale and complexity, users still expect to be able to query data at interactive speeds.

To tackle the "big data" problem, two major lines of systems have recently been explored. The first, consisting of MapReduce [17] and various generalizations [22, 13], offers a fine-grained fault tolerance model suitable for large clusters, where tasks on failed or slow nodes can be deterministically re-executed on other nodes. MapReduce is also fairly general: it has been shown to be able to express many statistical and learning algorithms [15]. It also easily supports unstructured data and "schema-on-read." However, MapReduce engines lack many of the features that make databases efficient, and thus exhibit high latencies of tens of seconds to hours. Even systems that have significantly optimized MapReduce for SQL queries, such as Google's Tenzing [13], or that combine it with a traditional database on each node, such as HadoopDB [4], report a minimum latency of 10 seconds. As such, MapReduce approaches have largely been dismissed for interactive-speed queries [31], and even Google is developing new engines for such workloads [29].

Instead, most MPP analytic databases (*e.g.,* Vertica, Greenplum, Teradata) and several of the new low-latency engines proposed for MapReduce environments (*e.g.,* Google Dremel [29], Cloudera Impala [1]) employ a coarser-grained recovery model, where an entire query has to be resubmitted if a machine fails.[1] This works well for short queries where a retry is inexpensive, but faces significant challenges for long queries as clusters scale up [4]. In addition, these systems often lack the rich analytics functions that are easy to implement in MapReduce, such as machine learning and graph algorithms. Furthermore, while it may be possible to implement some of these functions using UDFs, these algorithms are often expensive, exacerbating the need for fault and straggler recovery for long queries. Thus, most organizations tend to use other systems alongside MPP databases to perform complex analytics.

To provide an effective environment for big data analysis, we believe that processing systems will need to support *both* SQL and complex analytics efficiently, and to provide fine-grained fault recovery across both types of operations. This paper describes a new system that meets these goals, called Shark. Shark is open source and compatible with Apache Hive, and has already been used at web companies to speed up queries by 40–100×.

Shark builds on a recently-proposed distributed shared memory abstraction called Resilient Distributed Datasets (RDDs) [39] to perform most computations in memory while offering fine-grained fault tolerance. In-memory computing is increasingly important in large-scale analytics for two reasons. First, many complex analytics functions, such as machine learning and graph algorithms, are iterative, scanning the data multiple times; thus, the fastest systems deployed for these applications are in-memory [28, 27, 39]. Second, even traditional SQL warehouse workloads exhibit strong temporal and spatial locality, because more-recent fact table data

[1] Dremel provides fault tolerance within a query, but Dremel is limited to aggregation trees instead of the more complex communication patterns in joins.


*SIGMOD'13,* June 22–27, 2013, New York, New York, USA.

and small dimension tables are read disproportionately often. A study of Facebook's Hive warehouse and Microsoft's Bing analytics cluster showed that over 95% of queries in both systems could be served out of memory using just 64 GB/node as a cache, even though each system manages more than 100 PB of total data [6].

The main benefit of RDDs is an efficient mechanism for fault recovery. Traditional main-memory databases support fine-grained updates to tables and replicate writes across the network for fault tolerance, which is expensive on large commodity clusters. In contrast, RDDs restrict the programming interface to *coarse-grained* deterministic operators that affect multiple data items at once, such as *map*, *group-by* and *join*, and recover from failures by tracking the *lineage* of each dataset and recomputing lost data. This approach works well for data-parallel relational queries, and has also been shown to support machine learning and graph computation [39]. Thus, when a node fails, Shark can recover mid-query by rerunning the deterministic operations used to build lost data partitions on other nodes, similar to MapReduce. Indeed, it typically recovers within seconds by parallelizing this work across the cluster.

To run SQL efficiently, however, we also had to extend the RDD execution model, bringing in several concepts from traditional analytical databases and some new ones. We started with an existing implementation of RDDs called Spark [39], and added several features. First, to store and process relational data efficiently, we implemented in-memory columnar storage and columnar compression. This reduced both the data size and the processing time by as much as $5\times$ over naïvely storing the data in a Spark program in its original format. Second, to optimize SQL queries based on the data characteristics even in the presence of analytics functions and UDFs, we extended Spark with *Partial DAG Execution (PDE)*: Shark can reoptimize a running query after running the first few stages of its task DAG, choosing better join strategies or the right degree of parallelism based on observed statistics. Third, we leverage other properties of the Spark engine not present in traditional MapReduce systems, such as control over data partitioning.

Our implementation of Shark is compatible with Apache Hive [34], supporting all of Hive's SQL dialect and UDFs and allowing execution over unmodified Hive data warehouses. It augments SQL with complex analytics functions written in Spark, using Spark's Java, Scala or Python APIs. These functions can be combined with SQL in a single execution plan, providing in-memory data sharing and fast recovery across both types of processing.

Experiments show that using RDDs and the optimizations above, Shark can answer SQL queries up to $100\times$ faster than Hive, runs iterative machine learning algorithms more than $100\times$ faster than Hadoop, and can recover from failures mid-query within seconds. Shark's speed is comparable to that of MPP databases in benchmarks like Pavlo et al.'s comparison with MapReduce [31], but it offers fine-grained recovery and complex analytics features that these systems lack.

More fundamentally, our work shows that MapReduce-like execution models can be applied effectively to SQL, and offer a promising way to combine relational and complex analytics. In addition, we explore why current SQL engines implemented on top of MapReduce runtimes, such as Hive, are slow. We show how a combination of enhancements in Shark (*e.g.,* PDE), and engine properties that have not been optimized in MapReduce, such as the overhead of launching tasks, eliminate many of the bottlenecks in traditional MapReduce systems.

# 2 System Overview

As described in the previous section, Shark is a data analysis system that supports both SQL query processing and machine learning functions. Shark is compatible with Apache Hive, enabling users to run Hive queries much faster without any changes to either the queries or the data.

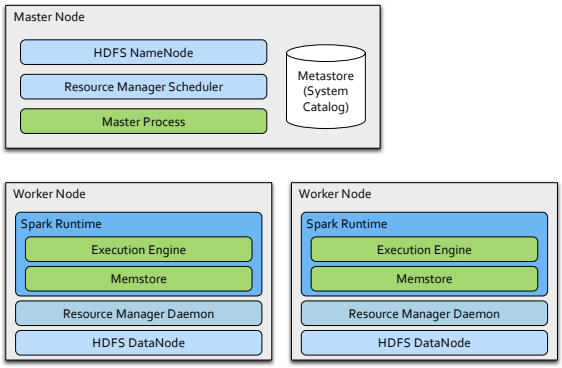



**Figure 1: Shark Architecture**

Thanks to its Hive compatibility, Shark can query data in any system that supports the Hadoop storage API, including HDFS and Amazon S3. It also supports a wide range of data formats such as text, binary sequence files, JSON, and XML. It inherits Hive's schema-on-read capability and nested data types [34].

In addition, users can choose to load high-value data into Shark's memory store for fast analytics, as illustrated below:

```
CREATE TABLE latest_logs
  TBLPROPERTIES ("shark.cache"=true)
AS SELECT * FROM logs WHERE date > now()-3600;
```

Figure 1 shows the architecture of a Shark cluster, consisting of a single master node and a number of worker nodes, with the warehouse metadata stored in an external transactional database. It is built on top of Spark, a modern MapReduce-like cluster computing engine. When a query is submitted to the master, Shark compiles the query into operator tree represented as RDDs, as we shall discuss in Section 2.4. These RDDs are then translated by Spark into a graph of tasks to execute on the worker nodes.

Cluster resources can optionally be allocated by a resource manager (*e.g.,* Hadoop YARN [2] or Apache Mesos [21]) that provides resource sharing and isolation between different computing frameworks, allowing Shark to coexist with engines like Hadoop.

In the remainder of this section, we cover the basics of Spark and the RDD programming model, and then we describe how Shark query plans are generated and executed.

## 2.1 Spark

Spark is the MapReduce-like cluster computing engine used by Shark. Spark has several features that differentiate it from traditional MapReduce engines [39]:

1. Like Dryad [22] and Hyracks [10], it supports general computation DAGs, not just the two-stage MapReduce topology.
2. It provides an in-memory storage abstraction called Resilient Distributed Datasets (RDDs) that lets applications keep data in memory across queries, and automatically reconstructs any data lost during failures [39].
3. The engine is optimized for low latency. It can efficiently manage tasks as short as 100 milliseconds on clusters of thousands of cores, while engines like Hadoop incur a latency of 5–10 seconds to launch each task.

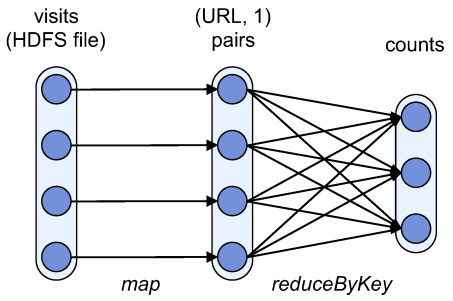



**Figure 2: Lineage graph for the RDDs in our Spark example. Oblongs represent RDDs, while circles show partitions within a dataset. Lineage is tracked at the granularity of partitions.**

RDDs are unique to Spark, and were essential to enabling mid-query fault tolerance. However, the other differences are important engineering elements that contribute to Shark's performance.

In addition to these features, we have also modified the Spark engine for Shark to support *partial DAG execution,* that is, modification of the query plan DAG after only some of the stages have finished, based on statistics collected from these stages. Similar to [25], we use this technique to optimize join algorithms and other aspects of the execution mid-query, as we shall discuss in Section 3.1.

### 2.2 Resilient Distributed Datasets (RDDs)

Spark's main abstraction is *resilient distributed datasets* (RDDs), which are immutable, partitioned collections that can be created through various data-parallel *operators* (*e.g., map*, *group-by*, *hash-join*). Each RDD is either a collection stored in an external storage system, such as a file in HDFS, or a derived dataset created by applying operators to other RDDs. For example, given an RDD of (visitID, URL) pairs for visits to a website, we might compute an RDD of (URL, count) pairs by applying a *map* operator to turn each event into an (URL, 1) pair, and then a *reduce* to add the counts by URL.

In Spark's native API, RDD operations are invoked through a functional interface similar to DryadLINQ [24] in Scala, Java or Python. For example, the Scala code for the query above is:

```
val visits = spark.hadoopFile("hdfs://...")
val counts = visits.map(v => (v.url, 1))
                   .reduceByKey((a, b) => a + b)
```

RDDs can contain arbitrary data types as elements (since Spark runs on the JVM, these elements are Java objects), and are automatically partitioned across the cluster, but they are immutable once created, and they can only be created through Spark's deterministic parallel operators. These two restrictions, however, enable highly efficient fault recovery. In particular, instead of replicating each RDD across nodes for fault-tolerance, Spark remembers the *lineage* of the RDD (the graph of operators used to build it), and recovers lost partitions by *recomputing* them from base data [39].[2] For example, Figure 2 shows the lineage graph for the RDDs computed above. If Spark loses one of the partitions in the (URL, 1) RDD, for example, it can recompute it by rerunning the *map* on just the corresponding partition of the input file.

The RDD model offers several key benefits in our large-scale in-memory computing setting. First, RDDs can be written at the speed of DRAM instead of the speed of the network, because there is no need to replicate each byte written to another machine for fault-tolerance. DRAM in a modern server is over $10\times$ faster than even a 10-Gigabit network. Second, Spark can keep just one copy of each RDD partition in memory, saving precious memory over a replicated system, since it can always recover lost data using lineage. Third, when a node fails, its lost RDD partitions can be rebuilt *in parallel* across the other nodes, allowing speedy recovery.[3] Fourth, even if a node is just slow (a "straggler"), we can recompute necessary partitions on other nodes because RDDs are immutable so there are no consistency concerns with having two copies of a partition. These benefits make RDDs attractive as the foundation for our relational processing in Shark.

### 2.3 Fault Tolerance Guarantees

To summarize the benefits of RDDs, Shark provides the following fault tolerance properties, which have been difficult to support in traditional MPP database designs:

1. Shark can tolerate the loss of *any set of worker nodes*. The execution engine will re-execute any lost tasks and recompute any lost RDD partitions using lineage.[4] This is true even within a query: Spark will rerun any failed tasks, or lost dependencies of new tasks, without aborting the query.
2. Recovery is parallelized across the cluster. If a failed node contained 100 RDD partitions, these can be rebuilt in parallel on 100 different nodes, quickly recovering the lost data.
3. The deterministic nature of RDDs also enables straggler mitigation: if a task is slow, the system can launch a speculative "backup copy" of it on another node, as in MapReduce [17].
4. Recovery works even for queries that *combine* SQL and machine learning UDFs (Section 4), as these operations all compile into a single RDD lineage graph.

### 2.4 Executing SQL over RDDs

Shark runs SQL queries over Spark using a three-step process similar to traditional RDBMSs: query parsing, logical plan generation, and physical plan generation.

Given a query, Shark uses the Hive query compiler to parse the query and generate an abstract syntax tree. The tree is then turned into a logical plan and basic logical optimization, such as predicate pushdown, is applied. Up to this point, Shark and Hive share an identical approach. Hive would then convert the operator into a physical plan consisting of multiple MapReduce stages. In the case of Shark, its optimizer applies additional rule-based optimizations, such as pushing `LIMIT` down to individual partitions, and creates a physical plan consisting of transformations on RDDs rather than MapReduce jobs. We use a variety of operators already present in Spark, such as *map* and *reduce*, as well as new operators we implemented for Shark, such as broadcast joins. Spark's master then executes this graph using standard MapReduce scheduling techniques, such as placing tasks close to their input data, rerunning lost tasks, and performing straggler mitigation [39].

While this basic approach makes it possible to run SQL over Spark, doing it *efficiently* is challenging. The prevalence of UDFs and complex analytic functions in Shark's workload makes it difficult to determine an optimal query plan at compile time, especially for new data that has not undergone ETL. In addition, even with

[2]We assume that external files for RDDs representing data do not change, or that we can take a snapshot of a file when we create an RDD from it.

[3]To provide fault tolerance across "shuffle" operations like a parallel reduce, the execution engine also saves the "map" side of the shuffle in memory on the source nodes, spilling to disk if necessary.

[4]Support for master recovery could also be added by reliabliy logging the RDD lineage graph and the submitted jobs, because this state is small, but we have not implemented this yet.

such a plan, naïvely executing it over Spark (or other MapReduce runtimes) can be inefficient. In the next section, we discuss several extensions we made to Spark to efficiently store relational data and run SQL, starting with a mechanism that allows for *dynamic*, statistics-driven re-optimization at run-time.

# 3 Engine Extensions

In this section, we describe our modifications to the Spark engine to enable efficient execution of SQL queries.

## 3.1 Partial DAG Execution (PDE)

Systems like Shark and Hive are frequently used to query fresh data that has not undergone a data loading process. This precludes the use of static query optimization techniques that rely on accurate a priori data statistics, such as statistics maintained by indices. The lack of statistics for fresh data, combined with the prevalent use of UDFs, requires dynamic approaches to query optimization.

To support dynamic query optimization in a distributed setting, we extended Spark to support *partial DAG execution* (PDE), a technique that allows dynamic alteration of query plans based on data statistics collected at run-time.

We currently apply partial DAG execution at blocking "shuffle" operator boundaries where data is exchanged and repartitioned, since these are typically the most expensive operations in Shark. By default, Spark materializes the output of each map task in memory before a shuffle, spilling it to disk as necessary. Later, reduce tasks fetch this output.

PDE modifies this mechanism in two ways. First, it gathers customizable statistics at global and per-partition granularities while materializing map outputs. Second, it allows the DAG to be altered based on these statistics, either by choosing different operators or altering their parameters (such as their degrees of parallelism).

These statistics are customizable using a simple, pluggable accumulator API. Some example statistics include:

1. Partition sizes and record counts, which can be used to detect skew.
2. Lists of "heavy hitters," *i.e.,* items that occur frequently in the dataset.
3. Approximate histograms, which can be used to estimate partitions' data distributions.

These statistics are sent by each worker to the master, where they are aggregated and presented to the optimizer. For efficiency, we use lossy compression to record the statistics, limiting their size to 1–2 KB per task. For instance, we encode partition sizes (in bytes) with logarithmic encoding, which can represent sizes of up to 32 GB using only one byte with at most 10% error. The master can then use these statistics to perform various run-time optimizations, as we shall discuss next.

Partial DAG execution complements existing adaptive query optimization techniques that typically run in a single-node system [7, 25, 36], as we can use existing techniques to dynamically optimize the local plan *within* each node, and use PDE to optimize the global structure of the plan at stage boundaries. This fine-grained statistics collection, and the optimizations that it enables, differentiates PDE from graph rewriting features in previous systems, such as DryadLINQ [24].

### 3.1.1 Join Optimization

Partial DAG execution can be used to perform several run-time optimizations for join queries.

Figure 3 illustrates two communication patterns for MapReduce-style joins. In *shuffle join*, both join tables are hash-partitioned by

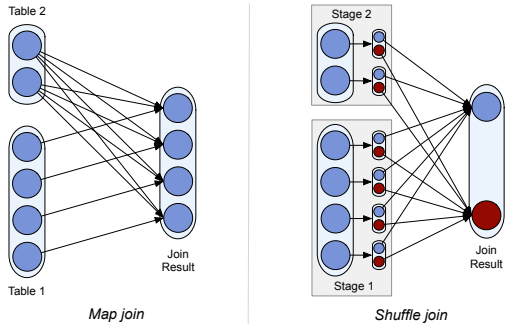



**Figure 3: Data flows for map join and shuffle join. Map join broadcasts the small table to all large table partitions, while shuffle join repartitions and shuffles both tables.**

the join key. Each reducer joins corresponding partitions using a local join algorithm, which is chosen by each reducer based on run-time statistics. If one of a reducer's input partitions is small, then it constructs a hash table over the small partition and probes it using the large partition. If both partitions are large, then a symmetric hash join is performed by constructing hash tables over both inputs.

In *map join*, also known as *broadcast join*, a small input table is broadcast to all nodes, where it is joined with each partition of a large table. This approach can result in significant cost savings by avoiding an expensive repartitioning and shuffling phase.

Map join is only worthwhile if some join inputs are small, so Shark uses partial DAG execution to select the join strategy at run-time based on its inputs' exact sizes. By using sizes of the join inputs gathered at run-time, this approach works well even with input tables that have no prior statistics, such as intermediate results.

Run-time statistics also inform the join tasks' scheduling policies. If the optimizer has a prior belief that a particular join input will be small, it will schedule that task before other join inputs and decide to perform a map-join if it observes that the task's output is small. This allows the query engine to avoid performing the pre-shuffle partitioning of a large table once the optimizer has decided to perform a map-join.

### 3.1.2 Skew-handling and Degree of Parallelism

Partial DAG execution can also be used to determine operators' degrees of parallelism and to mitigate skew.

The degree of parallelism for reduce tasks can have a large performance impact: launching too few reducers may overload reducers' network connections and exhaust their memories, while launching too many may prolong the job due to task scheduling overhead. Hive's performance is especially sensitive to the number of reduce tasks [8], due to Hadoop's large scheduling overhead.

Using partial DAG execution, Shark can use individual partitions' sizes to determine the number of reducers at run-time by coalescing many small, fine-grained partitions into fewer coarse partitions that are used by reduce tasks. To mitigate skew, fine-grained partitions are assigned to coalesced partitions using a greedy bin-packing heuristic that attempts to equalize coalesced partitions' sizes [19]. This offers performance benefits, especially when good bin-packings exist.

Somewhat surprisingly, we discovered that Shark can obtain similar performance improvement simply by running a larger number of reduce tasks. We attribute this to Spark's low scheduling and task-launching overhead.

## 3.2 Columnar Memory Store

In-memory computation is essential to low-latency query answering, given that memory's throughput is orders of magnitude higher

than that of disks. Naïvely using Spark's memory store, however, can lead to undesirable performance. For this reason, Shark implements a columnar memory store on top of Spark's native memory store.

In-memory data representation affects both space footprint and read throughput. A naïve approach is to simply cache the on-disk data in its native format, performing on-demand deserialization in the query processor. This deserialization becomes a major bottleneck: in our studies, we saw that modern commodity CPUs can deserialize at a rate of only 200MB per second per core.

The approach taken by Spark's default memory store is to store data partitions as collections of JVM objects. This avoids deserialization, since the query processor can directly use these objects, but leads to significant storage space overheads. Common JVM implementations add 12 to 16 bytes of overhead per object. For example, storing 270 MB of TPC-H lineitem table as JVM objects uses approximately 971 MB of memory, while a serialized representation requires only 289 MB, nearly three times less space. A more serious implication, however, is the effect on garbage collection (GC). With a 200 B record size, a 32 GB heap can contain 160 million objects. The JVM garbage collection time correlates linearly with the number of objects in the heap, so it could take minutes to perform a full GC on a large heap. These unpredictable, expensive garbage collections cause large variability in response times.

Shark stores all columns of primitive types as JVM primitive arrays. Complex data types supported by Hive, such as `map` and `array`, are serialized and concatenated into a single byte array. Each column creates only one JVM object, leading to fast GCs and a compact data representation. The space footprint of columnar data can be further reduced by cheap compression techniques at virtually no CPU cost. Similar to columnar database systems, *e.g.,* C-store [32], Shark implements CPU-efficient compression schemes such as dictionary encoding, run-length encoding, and bit packing.

Columnar data representation also leads to better cache behavior, especially for for analytical queries that frequently compute aggregations on certain columns.

### 3.3 Distributed Data Loading

In addition to query execution, Shark also uses Spark's execution engine for distributed data loading. During loading, a table is split into small partitions, each of which is loaded by a Spark task. The loading tasks use the data schema to extract individual fields from rows, marshal a partition of data into its columnar representation, and store those columns in memory.

Each data loading task tracks metadata to decide whether each column in a partition should be compressed. For example, the loading task will compress a column using dictionary encoding if its number of distinct values is below a threshold. This allows each task to choose the best compression scheme for each partition, rather than conforming to a global compression scheme that might not be optimal for local partitions. These local decisions do not require coordination among data loading tasks, allowing the load phase to achieve a maximum degree of parallelism, at the small cost of requiring each partition to maintain its own compression metadata. It is important to clarify that an RDD's lineage does not need to contain the compression scheme and metadata for each partition. The compression scheme and metadata are simply byproducts of the RDD computation, and can be deterministically recomputed along with the in-memory data in the case of failures.

As a result, Shark can load data into memory at the aggregated throughput of the CPUs processing incoming data.

Pavlo et al.[31] showed that Hadoop was able to perform data loading at 5 to 10 times the throughput of MPP databases. Tested using the same dataset used in [31], Shark provides the same throughput as Hadoop in loading data into HDFS. Shark is 5 times faster than Hadoop when loading data into its memory store.

### 3.4 Data Co-partitioning

In some warehouse workloads, two tables are frequently joined together. For example, the TPC-H benchmark frequently joins the lineitem and order tables. A technique commonly used by MPP databases is to co-partition the two tables based on their join key in the data loading process. In distributed file systems like HDFS, the storage system is schema-agnostic, which prevents data co-partitioning. Shark allows co-partitioning two tables on a common key for faster joins in subsequent queries. This can be accomplished with the `DISTRIBUTE BY` clause:

```
CREATE TABLE l_mem TBLPROPERTIES ("shark.cache"=true)
AS SELECT * FROM lineitem DISTRIBUTE BY L_ORDERKEY;

CREATE TABLE o_mem TBLPROPERTIES (
  "shark.cache"=true, "copartition"="l_mem")
AS SELECT * FROM order DISTRIBUTE BY O_ORDERKEY;
```

When joining two co-partitioned tables, Shark's optimizer constructs a DAG that avoids the expensive shuffle and instead uses map tasks to perform the join.

### 3.5 Partition Statistics and Map Pruning

Typically, data is stored using some logical clustering on one or more columns. For example, entries in a website's traffic log data might be grouped by users' physical locations, because logs are first stored in data centers that have the best geographical proximity to users. Within each data center, logs are append-only and are stored in roughly chronological order. As a less obvious case, a news site's logs might contain `news_id` and `timestamp` columns that are strongly correlated. For analytical queries, it is typical to apply filter predicates or aggregations over such columns. For example, a daily warehouse report might describe how different visitor segments interact with the website; this type of query naturally applies a predicate on timestamps and performs aggregations that are grouped by geographical location. This pattern is even more frequent for interactive data analysis, during which drill-down operations are frequently performed.

*Map pruning* is the process of pruning data partitions based on their natural clustering columns. Since Shark's memory store splits data into small partitions, each block contains only one or few logical groups on such columns, and Shark can avoid scanning certain blocks of data if their values fall out of the query's filter range.

To take advantage of these natural clusterings of columns, Shark's memory store on each worker piggybacks the data loading process to collect statistics. The information collected for each partition includes the range of each column and the distinct values if the number of distinct values is small (*i.e.,* enum columns). The collected statistics are sent back to the master program and kept in memory for pruning partitions during query execution.

When a query is issued, Shark evaluates the query's predicates against all partition statistics; partitions that do not satisfy the predicate are pruned and Shark does not launch tasks to scan them.

We collected a sample of queries from the Hive warehouse of a video analytics company, and out of the 3833 queries we obtained, at least 3277 of them contained predicates that Shark can use for map pruning. Section 6 provides more details on this workload.

## 4 Machine Learning Support

A key design goal of Shark is to provide a single system capable of efficient SQL query processing and sophisticated machine learning. Following the principle of pushing computation to data, Shark

```
def logRegress(points: RDD[Point]): Vector {
  var w = Vector(D, _ => 2 * rand.nextDouble - 1)
  for (i <- 1 to ITERATIONS) {
    val gradient = points.map { p =>
      val denom = 1 + exp(-p.y * (w dot p.x))
      (1 / denom - 1) * p.y * p.x
    }.reduce(_ + _)
    w -= gradient
  }
  w
}

val users = sql2rdd("SELECT * FROM user u
   JOIN comment c ON c.uid=u.uid")

val features = users.mapRows { row =>
  new Vector(extractFeature1(row.getInt("age")),
             extractFeature2(row.getStr("country")),
             ...)}
val trainedVector = logRegress(features.cache())
```

**Listing 1: Logistic Regression Example**

supports machine learning as a first-class citizen. This is enabled by the design decision to choose Spark as the execution engine and RDD as the main data structure for operators. In this section, we explain Shark's language and execution engine integration for SQL and machine learning.

Other research projects [16, 18] have demonstrated that it is possible to express certain machine learning algorithms in SQL and avoid moving data out of the database. The implementation of those projects, however, involves a combination of SQL, UDFs, and driver programs written in other languages. The systems become obscure and difficult to maintain; in addition, they may sacrifice performance by performing expensive parallel numerical computations on traditional database engines that were not designed for such workloads. Contrast this with the approach taken by Shark, which offers in-database analytics that push computation to data, but does so using a runtime that is optimized for such workloads and a programming model that is designed to express machine learning algorithms.

### 4.1 Language Integration

In addition to executing a SQL query and returning its results, Shark also allows queries to return the RDD representing the query plan. Callers to Shark can then invoke distributed computation over the query result using the returned RDD.

As an example of this integration, Listing 1 illustrates a data analysis pipeline that performs logistic regression over a user database. Logistic regression, a common classification algorithm, searches for a hyperplane $w$ that best separates two sets of points (e.g. spammers and non-spammers). The algorithm applies gradient descent optimization by starting with a randomized $w$ vector and iteratively updating it by moving along gradients towards an optimum value.

The program begins by using `sql2rdd` to issue a SQL query to retreive user information as a `TableRDD`. It then performs feature extraction on the query rows and runs logistic regression over the extracted feature matrix. Each iteration of `logRegress` applies a function of $w$ to all data points to produce a set of gradients, which are summed to produce a net gradient that is used to update $w$.

The highlighted `map`, `mapRows`, and `reduce` functions are automatically parallelized by Shark to execute across a cluster, and the master program simply collects the output of the `reduce` function to update $w$.

Note that this distributed logistic regression implementation in Shark looks remarkably similar to a program implemented for a single node in the Scala language. The user can conveniently mix the best parts of both SQL and MapReduce-style programming.

Currently, Shark provides native support for Scala, Java and Python. We have modified the Scala shell to enable interactive execution of both SQL and distributed machine learning algorithms. Because Shark is built on top of the JVM, it would be relatively straightforward to support other JVM languages, such as Clojure or JRuby.

We have implemented a number of basic machine learning algorithms, including linear regression, logistic regression, and k-means clustering. In most cases, the user only needs to supply a `mapRows` function to perform feature extraction and can invoke the provided algorithms.

The above example demonstrates how machine learning computations can be performed on query results. Using RDDs as the main data structure for query operators also enables one to use SQL to query the results of machine learning computations in a single execution plan.

### 4.2 Execution Engine Integration

In addition to language integration, another key benefit of using RDDs as the data structure for operators is the execution engine integration. This common abstraction allows machine learning computations and SQL queries to share workers and cached data without the overhead of data movement.

Because SQL query processing is implemented using RDDs, lineage is kept for the whole pipeline, which enables end-to-end fault tolerance for the entire workflow. If failures occur during the machine learning stage, partitions on faulty nodes will automatically be recomputed based on their lineage.

## 5 Implementation

While implementing Shark, we discovered that a number of engineering details had significant performance impacts. Overall, to improve the query processing speed, one should minimize the tail latency of tasks and the CPU cost of processing each row.

**Memory-based Shuffle:** Both Spark and Hadoop write map output files to disk, hoping that they will remain in the OS buffer cache when reduce tasks fetch them. In practice, we have found that the extra system calls and file system journaling adds significant overhead. In addition, the inability to control when buffer caches are flushed leads to variability in shuffle tasks. A query's response time is determined by the last task to finish, and thus the increasing variability leads to long-tail latency, which significantly hurts shuffle performance. We modified the shuffle phase to materialize map outputs in memory, with the option to spill them to disk.

**Temporary Object Creation:** It is easy to write a program that creates many temporary objects, which can burden the JVM's garbage collector. For a parallel job, a slow GC at one task may slow the entire job. Shark operators and RDD transformations are written in a way that minimizes temporary object creations.

**Bytecode Compilation of Expression Evaluators:** In its current implementation, Shark sends the expression evaluators generated by the Hive parser as part of the tasks to be executed on each row. By profiling Shark, we discovered that for certain queries, when data is served out of the memory store the majority of the CPU cycles are wasted in interpreting these evaluators. We are working on a compiler to transform these expression evaluators into JVM bytecode, which can further increase the execution engine's throughput.

**Specialized Data Structures:** Using specialized data structures is an optimization that we have yet to exploit. For example, Java's

hash table is built for generic objects. When the hash key is a primitive type, the use of specialized data structures can lead to more compact data representations, and thus better cache behavior.

# 6 Experiments

We evaluated Shark using four datasets:

1. Pavlo et al. Benchmark: 2.1 TB of data reproducing Pavlo et al.'s comparison of MapReduce vs. analytical DBMSs [31].
2. TPC-H Dataset: 100 GB and 1 TB datasets generated by the DBGEN program [35].
3. Real Hive Warehouse: 1.7 TB of sampled Hive warehouse data from an early industrial user of Shark.
4. Machine Learning Dataset: 100 GB synthetic dataset to measure the performance of machine learning algorithms.

Overall, our results show that Shark can perform more than $100\times$ faster than Hive and Hadoop, even though we have yet to implement some of the performance optimizations mentioned in the previous section. In particular, Shark provides comparable performance gains to those reported for MPP databases in Pavlo et al.'s comparison [31]. In some cases where data fits in memory, Shark exceeds the performance reported for MPP databases.

We emphasize that we are *not* claiming that Shark is fundamentally faster than MPP databases; there is no reason why MPP engines could not implement the same processing optimizations as Shark. Indeed, our implementation has several disadvantages relative to commercial engines, such as running on the JVM. Instead, we aim to show that it is possible to achieve comparable performance while retaining a MapReduce-like engine, and the fine-grained fault recovery features that such engines provide. In addition, Shark can leverage this engine to perform machine learning functions on the same data, which we believe will be essential for future analytics workloads.

## 6.1 Methodology and Cluster Setup

Unless otherwise specified, experiments were conducted on Amazon EC2 using 100 `m2.4xlarge` nodes. Each node had 8 virtual cores, 68 GB of memory, and 1.6 TB of local storage.

The cluster was running 64-bit Linux 3.2.28, Apache Hadoop 0.20.205, and Apache Hive 0.9. For Hadoop MapReduce, the number of map tasks and the number of reduce tasks per node were set to 8, matching the number of cores. For Hive, we enabled JVM reuse between tasks and avoided merging small output files, which would take an extra step after each query to perform the merge.

We executed each query six times, discarded the first run, and report the average of the remaining five runs. We discard the first run in order to allow the JVM's just-in-time compiler to optimize common code paths. We believe that this more closely mirrors real-world deployments where the JVM will be reused by many queries.

## 6.2 Pavlo et al. Benchmarks

Pavlo et al. compared Hadoop versus MPP databases and showed that Hadoop excelled at data ingress, but performed unfavorably in query execution [31]. We reused the dataset and queries from their benchmarks to compare Shark against Hive.

The benchmark used two tables: a 1 GB/node *rankings* table, and a 20 GB/node *uservisits* table. For our 100-node cluster, we recreated a 100 GB *rankings* table containing 1.8 billion rows and a 2 TB *uservisits* table containing 15.5 billion rows. We ran the four queries in their experiments comparing Shark with Hive and report the results in Figures 4 and 5. In this subsection, we hand-tuned Hive's number of reduce tasks to produce optimal results for

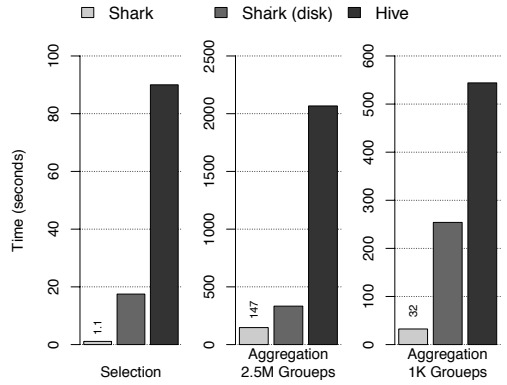



**Figure 4: Selection and aggregation query runtimes (seconds) from Pavlo et al. benchmark**

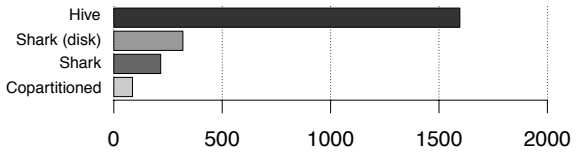



**Figure 5: Join query runtime (seconds) from Pavlo benchmark**

Hive. Despite this tuning, Shark outperformed Hive in all cases by a wide margin.

### 6.2.1 Selection Query

The first query was a simple selection on the *rankings* table:

```
SELECT pageURL, pageRank
FROM rankings WHERE pageRank > X;
```

In [31], Vertica outperformed Hadoop by a factor of 10 because a clustered index was created for Vertica. Even without a clustered index, Shark was able to execute this query $80\times$ faster than Hive for in-memory data, and $5\times$ on data read from HDFS.

### 6.2.2 Aggregation Queries

The Pavlo et al. benchmark ran two aggregation queries:

```
SELECT sourceIP, SUM(adRevenue)
FROM uservisits GROUP BY sourceIP;

SELECT SUBSTR(sourceIP, 1, 7), SUM(adRevenue)
FROM uservisits GROUP BY SUBSTR(sourceIP, 1, 7);
```

In our dataset, the first query had two million groups and the second had approximately one thousand groups. Shark and Hive both applied task-local aggregations and shuffled the data to parallelize the final merge aggregation. Again, Shark outperformed Hive by a wide margin. The benchmarked MPP databases perform local aggregations on each node, and then send all aggregates to a single query coordinator for the final merging; this performed very well when the number of groups was small, but performed worse with large number of groups. The MPP databases' chosen plan is similar to choosing a single reduce task for Shark and Hive.

### 6.2.3 Join Query

The final query from Pavlo et al. involved joining the 2 TB *uservisits* table with the 100 GB *rankings* table.

```
SELECT INTO Temp sourceIP, AVG(pageRank),
SUM(adRevenue) as totalRevenue
FROM rankings AS R, uservisits AS UV
WHERE R.pageURL = UV.destURL
AND UV.visitDate BETWEEN Date('2000-01-15')
AND Date('2000-01-22')
GROUP BY UV.sourceIP;
```

Again, Shark outperformed Hive in all cases. Figure 5 shows that for this query, serving data out of memory did not provide much benefit over disk. This is because the cost of the join step dominated the query processing. Co-partitioning the two tables, however, provided significant benefits as it avoided shuffling 2.1 TB of data during the join step.

#### 6.2.4 Data Loading

Hadoop was shown by [31] to excel at data loading, as its data loading throughput was five to ten times higher than that of MPP databases. As explained in Section 2, Shark can be used to query data in HDFS directly, which means its data ingress rate is at least as fast as Hadoop's.

After generating the 2 TB *uservisits* table, we measured the time to load it into HDFS and compared that with the time to load it into Shark's memory store. We found the rate of data ingress was $5\times$ higher in Shark's memory store than that of HDFS.

### 6.3 Micro-Benchmarks

To understand the factors affecting Shark's performance, we conducted a sequence of micro-benchmarks. We generated 100 GB and 1 TB of data using the DBGEN program provided by TPC-H [35]. We chose this dataset because it contains tables and columns of varying cardinality and can be used to create a myriad of micro-benchmarks for testing individual operators.

While performing experiments, we found that Hive and Hadoop MapReduce were very sensitive to the number of reducers set for a job. Hive's optimizer automatically sets the number of reducers based on the estimated data size. However, we found that Hive's optimizer frequently made the wrong decision, leading to incredibly long query execution times. We hand-tuned the number of reducers for Hive based on characteristics of the queries and through trial and error. We report Hive performance numbers for both optimizer-determined and hand-tuned numbers of reducers. Shark, on the other hand, was much less sensitive to the number of reducers and required minimal tuning.

#### 6.3.1 Aggregation Performance

We tested the performance of aggregations by running group-by queries on the TPC-H *lineitem* table. For the 100 GB dataset, *lineitem* table contained 600 million rows. For the 1 TB dataset, it contained 6 billion rows.

The queries were of the form:

```
SELECT [GROUP_BY_COLUMN], COUNT(*) FROM lineitem
GROUP BY [GROUP_BY_COLUMN]
```

We chose to run one query with no group-by column (*i.e.,* a simple count), and three queries with group-by aggregations: SHIPMODE (7 groups), RECEIPTDATE (2500 groups), and SHIPMODE (150 million groups in 100 GB, and 537 million groups in 1 TB).

For both Shark and Hive, aggregations were first performed on each partition, and then the intermediate aggregated results were partitioned and sent to reduce tasks to produce the final aggregation. As the number of groups becomes larger, more data needs to be shuffled across the network.

Figure 6 compares the performance of Shark and Hive, measuring Shark's performance on both in-memory data and data loaded from HDFS. As can be seen in the figure, Shark was $80\times$ faster than hand-tuned Hive for queries with small numbers of groups, and $20\times$ faster for queries with large numbers of groups, where the shuffle phase domniated the total execution cost.

We were somewhat surprised by the performance gain observed for on-disk data in Shark. After all, both Shark and Hive had to read data from HDFS and deserialize it for query processing. This difference, however, can be explained by Shark's very low task launching overhead, optimized shuffle operator, and other factors; see Section 7 for more details.

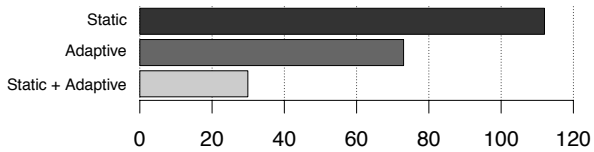



**Figure 7: Join strategies chosen by optimizers (seconds)**

#### 6.3.2 Join Selection at Run-time

In this experiment, we tested how partial DAG execution can improve query performance through run-time re-optimization of query plans. The query joined the *lineitem* and *supplier* tables from the 1 TB TPC-H dataset, using a UDF to select suppliers of interest based on their addresses. In this specific instance, the UDF selected 1000 out of 10 million suppliers. Figure 7 summarizes these results.

```
SELECT * from lineitem l join supplier s
ON l.L_SUPPKEY = s.S_SUPPKEY
WHERE SOME_UDF(s.S_ADDRESS)
```

Lacking good selectivity estimation on the UDF, a static optimizer would choose to perform a shuffle join on these two tables because the initial sizes of both tables are large. Leveraging partial DAG execution, after running the pre-shuffle map stages for both tables, Shark's dynamic optimizer realized that the filtered *supplier* table was small. It decided to perform a map-join, replicating the filtered *supplier* table to all nodes and performing the join using only map tasks on *lineitem*.

To further improve the execution, the optimizer can analyze the logical plan and infer that the probability of *supplier* table being small is much higher than that of *lineitem* (since *supplier* is smaller initially, and there is a filter predicate on *supplier*). The optimizer chose to pre-shuffle only the *supplier* table, and avoided launching two waves of tasks on *lineitem*. This combination of static query analysis and partial DAG execution led to a $3\times$ performance improvement over a naïve, statically chosen plan.

#### 6.3.3 Fault Tolerance

To measure Shark's performance in the presence of node failures, we simulated failures and measured query performance before, during, and after failure recovery. Figure 8 summarizes fives runs of our failure recovery experiment, which was performed on a 50-node `m2.4xlarge` EC2 cluster.

We used a group-by query on the 100 GB *lineitem* table to measure query performance in the presence of faults. After loading the *lineitem* data into Shark's memory store, we killed a worker machine and re-ran the query. Shark gracefully recovered from this failure and parallelized the reconstruction of lost partitions on the other 49 nodes. This recovery had a small performance impact, but it was significantly cheaper than the cost of re-loading the entire dataset and re-executing the query (14 vs 34 secs).

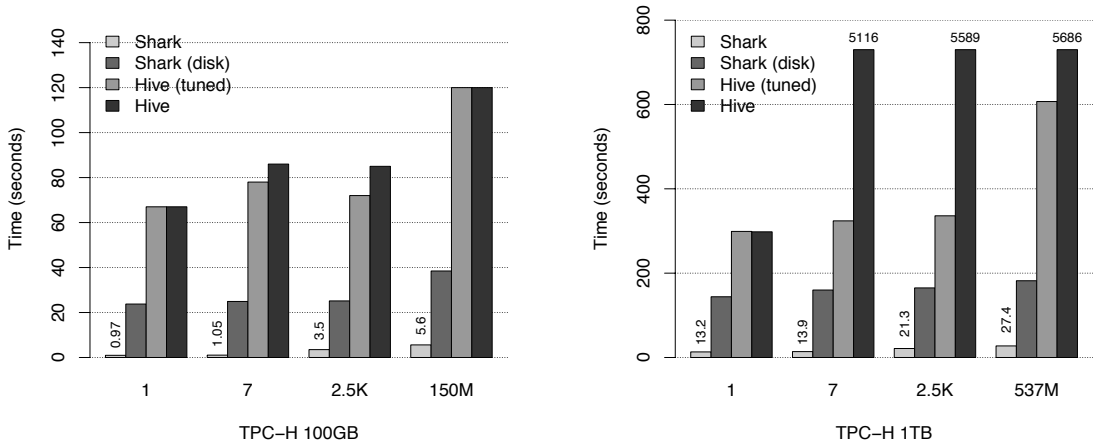



**Figure 6: Aggregation queries on *lineitem* table. X-axis indicates the number of groups for each aggregation query.**

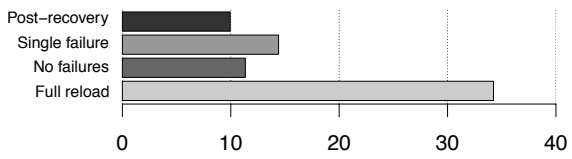



**Figure 8: Query time with failures (seconds)**

After this recovery, subsequent queries operated against the recovered dataset, albeit with fewer machines. In Figure 8, the post-recovery performance was marginally better than the pre-failure performance; we believe that this was a side-effect of the JVM's JIT compiler, as more of the scheduler's code might have become compiled by the time the post-recovery queries were run.

### 6.4 Real Hive Warehouse Queries

An early industrial user provided us with a sample of their Hive warehouse data and two years of query traces from their Hive system. A leading video analytics company for content providers and publishers, the user built most of their analytics stack based on Hadoop. The sample we obtained contained 30 days of video session data, occupying 1.7 TB of disk space when decompressed. It consists of a single fact table containing 103 columns, with heavy use of complex data types such as `array` and `struct`. The sampled query log contains 3833 analytical queries, sorted in order of frequency. We filtered out queries that invoked proprietary UDFs and picked four frequent queries that are prototypical of other queries in the complete trace. These queries compute aggregate video quality metrics over different audience segments:

1. Query 1 computes summary statistics in 12 dimensions for users of a specific customer on a specific day.
2. Query 2 counts the number of sessions and distinct customer/-client combination grouped by countries with filter predicates on eight columns.
3. Query 3 counts the number of sessions and distinct users for all but 2 countries.
4. Query 4 computes summary statistics in 7 dimensions grouping by a column, and showing the top groups sorted in descending order.

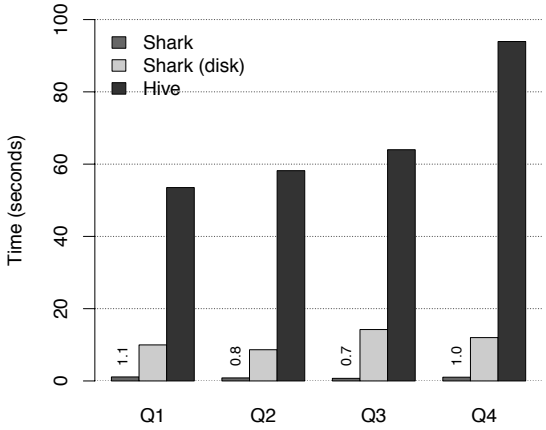



**Figure 9: Real Hive warehouse workloads**

Figure 9 compares the performance of Shark and Hive on these queries. The result is very promising as Shark was able to process these real life queries in sub-second latency in all but one cases, whereas it took Hive 50 to 100 times longer to execute them.

A closer look into these queries suggests that this data exhibits the natural clustering properties mentioned in Section 3.5. The map pruning technique, on average, reduced the amount of data scanned by a factor of 30.

### 6.5 Machine Learning

A key motivator of using SQL in a MapReduce environment is the ability to perform sophisticated machine learning on big data. We implemented two iterative machine learning algorithms, logistic regression and k-means, to compare the performance of Shark versus running the same workflow in Hive and Hadoop.

The dataset was synthetically generated and contained 1 billion rows and 10 columns, occupying 100 GB of space. Thus, the feature matrix contained 1 billion points, each with 10 dimensions. These machine learning experiments were performed on a 100-node `m1.xlarge` EC2 cluster.

Data was initially stored in relational form in Shark's memory store and HDFS. The workflow consisted of three steps: (1) selecting the data of interest from the warehouse using SQL, (2) extracting features, and (3) applying iterative algorithms. In step 3, both algorithms were run for 10 iterations.

Figures 10 and 11 show the time to execute a single iteration

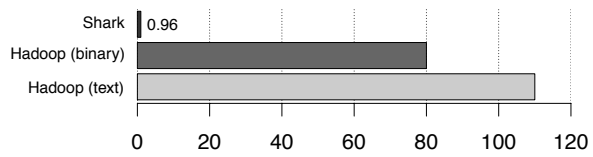



**Figure 10: Logistic regression, per-iteration runtime (seconds)**

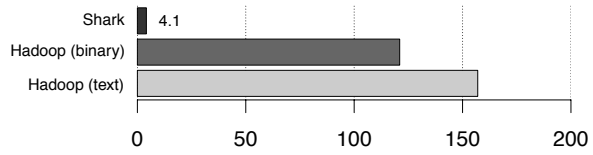



**Figure 11: K-means clustering, per-iteration runtime (seconds)**

of logistic regression and k-means, respectively. We implemented two versions of the algorithms for Hadoop, one storing input data as text in HDFS and the other using a serialized binary format. The binary representation was more compact and had lower CPU cost in record deserialization, leading to improved performance. Our results show that Shark is $100\times$ faster than Hive and Hadoop for logistic regression and $30\times$ faster for k-means. K-means experienced less speedup because it was computationally more expensive than logistic regression, thus making the workflow more CPU-bound.

In the case of Shark, if data initially resided in its memory store, step 1 and 2 were executed in roughly the same time it took to run one iteration of the machine learning algorithm. If data was not loaded into the memory store, the first iteration took 40 seconds for both algorithms. Subsequent iterations, however, reported numbers consistent with Figures 10 and 11. In the case of Hive and Hadoop, every iteration took the reported time because data was loaded from HDFS for every iteration.

# 7 Discussion

Shark shows that it is possible to run fast relational queries in a fault-tolerant manner using the fine-grained deterministic task model introduced by MapReduce. This design offers an effective way to scale query processing to ever-larger workloads, and to combine it with rich analytics. In this section, we consider two questions: first, why were previous MapReduce-based systems, such as Hive, slow, and what gave Shark its advantages? Second, are there other benefits to the fine-grained task model? We argue that fine-grained tasks also help with multitenancy and elasticity, as has been demonstrated in MapReduce systems.

## 7.1 Why are Previous MapReduce-Based Systems Slow?

Conventional wisdom is that MapReduce is slower than MPP databases for several reasons: expensive data materialization for fault tolerance, inferior data layout (*e.g.,* lack of indices), and costlier execution strategies [31, 33]. Our exploration of Hive confirms these reasons, but also shows that a combination of conceptually simple "engineering" changes to the engine (*e.g.,* in-memory storage) and more involved architectural changes (*e.g.,* partial DAG execution) can alleviate them. We also find that a somewhat surprising variable not considered in detail in MapReduce systems, the task scheduling overhead, actually has a dramatic effect on performance, and greatly improves load balancing if minimized.

**Intermediate Outputs:** MapReduce-based query engines, such as Hive, materialize intermediate data to disk in two situations. First, *within* a MapReduce job, the map tasks save their output in case a reduce task fails [17]. Second, many queries need to be compiled into *multiple* MapReduce steps, and engines rely on replicated file systems, such as HDFS, to store the output of each step.

For the first case, we note that map outputs were stored on disk primarily as a convenience to ensure there is sufficient space to hold them in large batch jobs. Map outputs are *not* replicated across nodes, so they will still be lost if the mapper node fails [17]. Thus, if the outputs fit in memory, it makes sense to store them in memory initially, and only spill them to disk if they are large. Shark's shuffle implementation does this by default, and sees far faster shuffle performance (and no seeks) when the outputs fit in RAM. This is often the case in aggregations and filtering queries that return a much smaller output than their input.[5] Another hardware trend that may improve performance, even for large shuffles, is SSDs, which would allow fast random access to a larger space than memory.

For the second case, engines that extend the MapReduce execution model to general task DAGs can run multi-stage jobs without materializing any outputs to HDFS. Many such engines have been proposed, including Dryad, Tenzing and Spark [22, 13, 39].

**Data Format and Layout:** While the naïve pure schema-on-read approach to MapReduce incurs considerable processing costs, many systems use more efficient storage formats within the MapReduce model to speed up queries. Hive itself supports "table partitions" (a basic index-like system where it knows that certain key ranges are contained in certain files, so it can avoid scanning a whole table), as well as column-oriented representation of on-disk data [34]. We go further in Shark by using fast in-memory columnar representations within Spark. Shark does this without modifying the Spark runtime by simply representing a block of tuples as a single Spark record (one Java object from Spark's perspective), and choosing its own representation for the tuples within this object.

Another feature of Spark that helps Shark, but was not present in previous MapReduce runtimes, is control over the data partitioning across nodes (Section 3.4). This lets us co-partition tables.

Finally, one capability of RDDs that we do not yet exploit is random reads. While RDDs only support coarse-grained operations for their *writes*, *read* operations on them can be fine-grained, accessing just one record [39]. This would allow RDDs to be used as indices. Tenzing can use such remote-lookup reads for joins [13].

**Execution Strategies:** Hive spends considerable time on sorting the data before each shuffle and writing the outputs of each MapReduce stage to HDFS, both limitations of the rigid, one-pass MapReduce model in Hadoop. More general runtime engines, such as Spark, alleviate some of these problems. For instance, Spark supports hash-based distributed aggregation and general task DAGs.

To truly optimize the execution of relational queries, however, we found it necessary to select execution plans based on data statistics. This becomes difficult in the presence of UDFs and complex analytics functions, which we seek to support as first-class citizens in Shark. To address this problem, we proposed partial DAG execution (PDE), which allows our modified version of Spark to *change* the downstream portion of an execution graph once each stage completes based on data statistics. PDE goes beyond the runtime graph rewriting features in previous systems, such as DryadLINQ [24], by collecting fine-grained statistics about ranges of keys and by allowing switches to a completely different join strategy, such as broadcast join, instead of just selecting the number of reduce tasks.

[5] Systems like Hadoop also benefit from the OS buffer cache in serving map outputs, but we found that the extra system calls and file system journalling from writing map outputs to files still adds overhead (Section 5).

**Task Scheduling Cost:** Perhaps the most surprising engine property that affected Shark, however, was a purely "engineering" concern: the overhead of launching tasks. Traditional MapReduce systems, such as Hadoop, were designed for multi-hour batch jobs consisting of tasks that were several minutes long. They launched each task in a separate OS process, and in some cases had a high latency to even submit a task. For instance, Hadoop uses periodic "heartbeats" from each worker every 3 seconds to assign tasks, and sees overall task startup delays of 5–10 seconds. This was sufficient for batch workloads, but clearly falls short for ad-hoc queries.

Spark avoids this problem by using a fast event-driven RPC library to launch tasks and by reusing its worker processes. It can launch thousands of tasks per second with only about 5 ms of overhead per task, making task lengths of 50–100 ms and MapReduce jobs of 500 ms viable. What surprised us is how much this affected query performance, even in large (multi-minute) queries.

Sub-second tasks allow the engine to balance work across nodes extremely well, even when some nodes incur unpredictable delays (*e.g.,* network delays or JVM garbage collection). They also help dramatically with skew. Consider, for example, a system that needs to run a hash aggregation on 100 cores. If the system launches 100 reduce tasks, the key range for each task needs to be carefully chosen, as any imbalance will slow down the entire job. If it could split the work among 1000 tasks, then the slowest task can be as much as $10\times$ slower than the average without affecting the job response time much! After implementing skew-aware partition selection in PDE, we were somewhat disappointed that it did not help compared to just having a higher number of reduce tasks in most workloads, because Spark could comfortably support thousands of such tasks. However, this property makes the engine highly robust to unexpected skew.

In this way, Spark stands in contrast to Hadoop/Hive, where using the wrong number of tasks was sometimes $10\times$ slower than an optimal plan, and there has been considerable work to automatically choose the number of reduce tasks [26, 19]. Figure 12 shows how job execution times vary as the number of reduce tasks launched by Hadoop and Spark in a simple aggregation query on a 100-node cluster. Since a Spark job can launch thousands of reduce tasks without incurring much overhead, partition data skew can be mitigated by always launching many tasks.

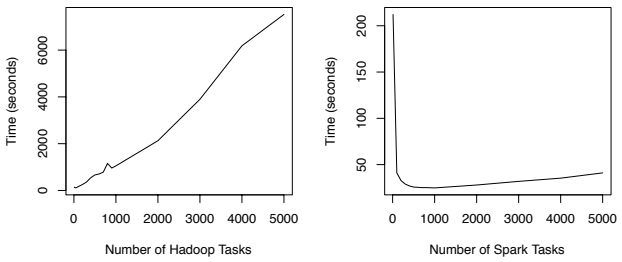



**Figure 12: Task launching overhead**

More fundamentally, there are few reasons why sub-second tasks should not be feasible even at higher scales than we have explored, such as tens of thousands of nodes. Systems like Dremel [29] routinely run sub-second, multi-thousand-node jobs. Indeed, even if a single master cannot keep up with the scheduling decisions, the scheduling could be delegated across "lieutenant" masters for subsets of the cluster. Fine-grained tasks also offer many advantages over coarser-grained execution graphs beyond load balancing, such as faster recovery (by spreading out lost tasks across more nodes) and query elasticity [30]; we discuss some of these next.

### 7.2 Other Benefits of the Fine-Grained Task Model

While this paper has focused primarily on the fault tolerance benefits of fine-grained deterministic tasks, the model also provides other attractive properties. We wish to point out two benefits that have been explored in MapReduce-based systems.

**Elasticity:** In traditional MPP databases, a distributed query plan is selected once, and the system needs to run at that level of parallelism for the whole duration of the query. In a fine-grained task system, however, nodes can appear or go away during a query, and pending work will automatically be spread onto them. This enables the database engine to naturally be elastic. If an administrator wishes to remove nodes from the engine (*e.g.,* in a virtualized corporate data center), the engine can simply treat those as failed, or (better yet) proactively replicate their data to other nodes if given a few minutes' warning. Similarly, a database engine running on a cloud could scale *up* by requesting new VMs if a query is expensive. Amazon's Elastic MapReduce [3] already supports resizing clusters at runtime.

**Multitenancy:** The same elasticity, mentioned above, enables dynamic resource sharing between users. In some traditional MPP databases, if an important query arrives while another large query is using most of the cluster, there are few options beyond canceling the earlier query. In systems based on fine-grained tasks, one can simply wait a few seconds for the current tasks from the first query to finish, and start giving the nodes tasks from the second query. For instance, Facebook and Microsoft have developed fair schedulers for Hadoop and Dryad that allow large historical queries, compute-intensive machine learning jobs, and short ad-hoc queries to safely coexist [38, 23].

## 8 Related Work

To the best of our knowledge, Shark is the only low-latency system that can efficiently combine SQL and machine learning workloads, while supporting fine-grained fault recovery.

We categorize large-scale data analytics systems into three classes. First, systems like ASTERIX [9], Tenzing [13], SCOPE [12], Cheetah [14], and Hive [34] compile declarative queries into MapReduce-style jobs. Although some of them modify the execution engine they are built on, it is hard for these systems to achieve interactive query response times for reasons discussed in Section 7.

Second, several projects aim to provide low-latency engines using architectures resembling shared-nothing parallel databases. Such projects include PowerDrill [20] and Impala [1]. These systems do not support fine-grained fault tolerance. In case of mid-query faults, the entire query needs to be re-executed. Google's Dremel [29] does rerun lost tasks, but it only supports an aggregation tree topology for query execution, and not the more complex shuffle DAGs required for large joins or distributed machine learning.

A third class of systems take a hybrid approach by combining a MapReduce-like engine with relational databases. HadoopDB [4] connects multiple single-node database systems using Hadoop as the communication layer. Queries can be parallelized using Hadoop MapReduce, but within each MapReduce task, data processing is pushed into the relational database system. Osprey [37] is a middleware layer that adds fault-tolerance properties to parallel databases. It does so by breaking a SQL query into multiple small queries and sending them to parallel databases for execution. Shark presents a much simpler single-system architecture that supports all of the properties of this third class of systems, as well as statistical learning capabilities that HadoopDB and Osprey lack.

The partial DAG execution (PDE) technique introduced by Shark resembles adaptive query optimization techniques proposed in [7,

36, 25]. It is, however, unclear how these single-node techniques would work in a distributed setting and scale out to hundreds of nodes. In fact, PDE actually complements some of these techniques, as Shark can use PDE to optimize how data gets shuffled *across* nodes, and use the traditional single-node techniques *within* a local task. DryadLINQ [24] optimizes its number of reduce tasks at run-time based on map output sizes, but does not collect richer statistics, such as histograms, or make broader execution plan changes, such as changing join algorithms, like PDE can. RoPE [5] proposes using historical query information to optimize query plans, but relies on repeatedly executed queries. PDE works on queries that are executing for the first time.

Finally, Shark builds on the distributed approaches for machine learning developed in systems like Graphlab [27], Haloop [11], and Spark [39]. However, Shark is unique in offering these capabilities in a SQL engine, allowing users to select data of interest using SQL and immediately run learning algorithms on it without time-consuming export to another system. Compared to Spark, Shark also provides far more efficient in-memory representation of relational data, and mid-query optimization using PDE.

# 9 Conclusion

We have presented Shark, a new data warehouse system that combines fast relational queries and complex analytics in a single, fault-tolerant runtime. Shark significantly enhances a MapReduce-like runtime to efficiently run SQL, by using existing database techniques (e.g., column-oriented storage) and a novel *partial DAG execution (PDE)* technique that leverages fine-grained data statistics to dynamically reoptimize queries at run-time. This design enables Shark to approach the speedups reported for MPP databases over MapReduce, while providing support for machine learning algorithms, as well as mid-query fault tolerance across both SQL queries and machine learning computations. Overall, the system can be up to $100\times$ faster than Hive for SQL, and more than $100\times$ faster than Hadoop for machine learning. More fundamentally, this research represents an important step towards a unified architecture for efficiently combining complex analytics and relational query processing.

We have open sourced Shark at `shark.cs.berkeley.edu`. The latest stable release implements most of the techniques discussed in this paper, and more advanced features such as PDE and data copartitioning will be incorporated soon. We have also worked with two Internet companies as early users. They report speedups of 40–$100\times$ on real queries, consistent with our results.

# 10 Acknowledgments

We thank Cliff Engle, Harvey Feng, Shivaram Venkataraman, Ram Sriharsha, Tim Tully, Denny Britz, Antonio Lupher, Patrick Wendell, Paul Ruan, Jason Dai, Shane Huang, and other colleagues in the AMPLab for their work on Shark. We also thank Andy Pavlo and his colleagues for making their benchmark dataset and queries available. This research is supported in part by NSF CISE Expeditions award CCF-1139158 and DARPA XData Award FA8750-12-2-0331, and gifts from Amazon Web Services, Google, SAP, Blue Goji, Cisco, Clearstory Data, Cloudera, Ericsson, Facebook, General Electric, Hortonworks, Huawei, Intel, Microsoft, NetApp, Oracle, Quanta, Samsung, Splunk, VMware and Yahoo!, and by a Google PhD Fellowship.

# 11 References

[1] https://github.com/cloudera/impala.
[2] http://hadoop.apache.org/.
[3] http://aws.amazon.com/elasticmapreduce/.
[4] A. Abouzeid et al. Hadoopdb: an architectural hybrid of mapreduce and dbms technologies for analytical workloads. *VLDB*, 2009.
[5] S. Agarwal et al. Re-optimizing data-parallel computing. In *NSDI'12*.
[6] G. Ananthanarayanan et al. Pacman: Coordinated memory caching for parallel jobs. In *NSDI*, 2012.
[7] R. Avnur and J. M. Hellerstein. Eddies: continuously adaptive query processing. In *SIGMOD*, 2000.
[8] S. Babu. Towards automatic optimization of mapreduce programs. In *SoCC'10*.
[9] A. Behm et al. Asterix: towards a scalable, semistructured data platform for evolving-world models. *Distributed and Parallel Databases*, 29(3):185–216, 2011.
[10] V. Borkar et al. Hyracks: A flexible and extensible foundation for data-intensive computing. In *ICDE'11*.
[11] Y. Bu et al. HaLoop: efficient iterative data processing on large clusters. *Proc. VLDB Endow.*, 2010.
[12] R. Chaiken et al. Scope: easy and efficient parallel processing of massive data sets. *VLDB*, 2008.
[13] B. Chattopadhyay, , et al. Tenzing a sql implementation on the mapreduce framework. *PVLDB*, 4(12):1318–1327, 2011.
[14] S. Chen. Cheetah: a high performance, custom data warehouse on top of mapreduce. *VLDB*, 2010.
[15] C. Chu et al. Map-reduce for machine learning on multicore. *Advances in neural information processing systems*, 19:281, 2007.
[16] J. Cohen, B. Dolan, M. Dunlap, J. Hellerstein, and C. Welton. Mad skills: new analysis practices for big data. *VLDB*, 2009.
[17] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.
[18] X. Feng et al. Towards a unified architecture for in-rdbms analytics. In *SIGMOD*, 2012.
[19] B. Guffler et al. Handling data skew in mapreduce. In *CLOSER'11*.
[20] A. Hall et al. Processing a trillion cells per mouse click. *VLDB*.
[21] B. Hindman et al. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI'11*.
[22] M. Isard et al. Dryad: distributed data-parallel programs from sequential building blocks. *SIGOPS*, 2007.
[23] M. Isard et al. Quincy: Fair scheduling for distributed computing clusters. In *SOSP '09*, 2009.
[24] M. Isard and Y. Yu. Distributed data-parallel computing using a high-level programming language. In *SIGMOD*, 2009.
[25] N. Kabra and D. J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *SIGMOD*, 1998.
[26] Y. Kwon et al. Skewtune: mitigating skew in mapreduce applications. In *SIGMOD '12*, 2012.
[27] Y. Low et al. Distributed graphlab: a framework for machine learning and data mining in the cloud. *VLDB*, 2012.
[28] G. Malewicz et al. Pregel: a system for large-scale graph processing. In *SIGMOD*, 2010.
[29] S. Melnik et al. Dremel: interactive analysis of web-scale datasets. *Proc. VLDB Endow.*, 3:330–339, Sept 2010.
[30] K. Ousterhout et al. The case for tiny tasks in compute clusters. In *HotOS'13*.
[31] A. Pavlo et al. A comparison of approaches to large-scale data analysis. In *SIGMOD*, 2009.
[32] M. Stonebraker et al. C-store: a column-oriented dbms. In *VLDB'05*.
[33] M. Stonebraker et al. Mapreduce and parallel dbmss: friends or foes? *Commun. ACM*.
[34] A. Thusoo et al. Hive-a petabyte scale data warehouse using hadoop. In *ICDE*, 2010.
[35] Transaction Processing Performance Council. *TPC BENCHMARK H*.
[36] T. Urhan, M. J. Franklin, and L. Amsaleg. Cost-based query scrambling for initial delays. In *SIGMOD*, 1998.
[37] C. Yang et al. Osprey: Implementing mapreduce-style fault tolerance in a shared-nothing distributed database. In *ICDE*, 2010.
[38] M. Zaharia et al. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys 10*, 2010.
[39] M. Zaharia et al. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. NSDI, 2012.

# Discretized Streams: Fault-Tolerant Streaming Computation at Scale

Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, Ion Stoica
*University of California, Berkeley*

## Abstract

Many "big data" applications must act on data in real time. Running these applications at ever-larger scales requires parallel platforms that automatically handle faults and stragglers. Unfortunately, current distributed stream processing models provide fault recovery in an expensive manner, requiring hot replication or long recovery times, and do not handle stragglers. We propose a new processing model, *discretized streams* (D-Streams), that overcomes these challenges. D-Streams enable a *parallel recovery* mechanism that improves efficiency over traditional replication and backup schemes, and tolerates stragglers. We show that they support a rich set of operators while attaining high per-node throughput similar to single-node systems, linear scaling to 100 nodes, sub-second latency, and sub-second fault recovery. Finally, D-Streams can easily be composed with batch and interactive query models like MapReduce, enabling rich applications that combine these modes. We implement D-Streams in a system called Spark Streaming.

## 1 Introduction

Much of "big data" is received in real time, and is most valuable at its time of arrival. For example, a social network may wish to detect trending conversation topics in minutes; a search site may wish to model which users visit a new page; and a service operator may wish to monitor program logs to detect failures in seconds. To enable these low-latency processing applications, there is a need for streaming computation models that scale transparently to large clusters, in the same way that batch models like MapReduce simplified offline processing.

Designing such models is challenging, however, because the scale needed for the largest applications (*e.g.,* realtime log processing or machine learning) can be hundreds of nodes. At this scale, two major problems are *faults* and *stragglers* (slow nodes). Both problems are inevitable in large clusters [12], so streaming applications must recover from them quickly. Fast recovery is even *more* important in streaming than it was in batch jobs: while a 30 second delay to recover from a fault or straggler is a nuisance in a batch setting, it can mean losing the chance to make a key decision in a streaming setting.

Unfortunately, existing streaming systems have limited fault and straggler tolerance. Most distributed streaming systems, including Storm [37], TimeStream [33], MapReduce Online [11], and streaming databases [5, 9, 10], are based on a *continuous operator* model, in which long-running, stateful operators receive each record, update internal state, and send new records. While this model is quite natural, it makes it difficult to handle faults and stragglers.

Specifically, given the continuous operator model, systems perform recovery through two approaches [20]: *replication*, where there are two copies of each node [5, 34], or *upstream backup*, where nodes buffer sent messages and replay them to a new copy of a failed node [33, 11, 37]. Neither approach is attractive in large clusters: replication costs $2\times$ the hardware, while upstream backup takes a long time to recover, as the whole system must wait for a new node to serially rebuild the failed node's state by rerunning data through an operator. In addition, neither approach handles stragglers: in upstream backup, a straggler must be treated as a failure, incurring a costly recovery step, while replicated systems use synchronization protocols like Flux [34] to coordinate replicas, so a straggler will slow down both replicas.

This paper presents a new stream processing model, *discretized streams* (D-Streams), that overcomes these challenges. Instead of managing long-lived operators, the idea in D-Streams is to structure a streaming computation as a series of *stateless, deterministic batch computations* on small time intervals. For example, we might place the data received every second (or every 100ms) into an interval, and run a MapReduce operation on each interval to compute a count. Similarly, we can run a rolling count over several intervals by adding the new count from each interval to the old result. By structuring computations this way, D-Streams make (1) the *state* at each timestep fully deterministic given the input data, forgoing the need for synchronization protocols, and (2) the *dependencies* between this state and older data visi-


*SOSP'13*, Nov. 3–6, 2013, Farmington, Pennsylvania, USA.
ACM 978-1-4503-2388-8/13/11.
http://dx.doi.org/10.1145/2517349.2522737

ble at a fine granularity. We show that this enables powerful recovery mechanisms, similar to those in batch systems, that outperform replication and upstream backup.

There are two challenges in realizing the D-Stream model. The first is making the latency (interval granularity) low. Traditional batch systems, such as Hadoop, fall short here because they keep state in replicated, on-disk storage systems between jobs. Instead, we use a data structure called Resilient Distributed Datasets (RDDs) [43], which keeps data in memory and can recover it without replication by tracking the *lineage graph* of operations that were used to build it. With RDDs, we show that we can attain sub-second end-to-end latencies. We believe that this is sufficient for many real-world big data applications, where the timescale of the events tracked (*e.g.,* trends in social media) is much higher.

The second challenge is recovering quickly from faults and stragglers. Here, we use the determinism of D-Streams to provide a new recovery mechanism that has not been present in previous streaming systems: *parallel recovery* of a lost node's state. When a node fails, each node in the cluster works to recompute part of the lost node's RDDs, resulting in significantly faster recovery than upstream backup without the cost of replication. Parallel recovery was hard to perform in continuous processing systems due to the complex state synchronization protocols needed even for basic replication (*e.g.,* Flux [34]),[1] but becomes simple with the fully deterministic D-Stream model. In a similar way, D-Streams can recover from stragglers using speculative execution [12], while previous streaming systems do not handle them.

We have implemented D-Streams in a system called Spark Streaming, based on the Spark engine [43]. The system can process over 60 million records/second on 100 nodes at sub-second latency, and can recover from faults and stragglers in sub-second time. Spark Streaming's per-node throughput is comparable to commercial streaming databases, while offering linear scalability to 100 nodes, and is 2–5× faster than the open source Storm and S4 systems, while offering fault recovery guarantees that they lack. Apart from its performance, we illustrate Spark Streaming's expressiveness through ports of two real applications: a video distribution monitoring system and an online machine learning system.

Finally, because D-Streams use the same processing model and data structures (RDDs) as batch jobs, a powerful advantage of our model is that streaming queries can seamlessly be *combined* with batch and interactive computation. We leverage this feature in Spark Streaming to let users run ad-hoc queries on streams using Spark, or join streams with historical data computed as an RDD. This is a powerful feature in practice, giving users a single API to combine previously disparate computations. We sketch how we have used it in our applications to blur the line between live and offline processing.

[1] The only parallel recovery algorithm we are aware of, by Hwang *et al.* [21], only tolerates one node failure and cannot handle stragglers.

## 2 Goals and Background

Many important applications process large streams of data arriving in real time. Our work targets applications that need to run on tens to hundreds of machines, and tolerate a latency of several seconds. Some examples are:

- **Site activity statistics:** Facebook built a distributed aggregation system called Puma that gives advertisers statistics about users clicking their pages within 10–30 seconds and processes $10^6$ events/s [35].
- **Cluster monitoring:** Datacenter operators often collect and mine program logs to detect problems, using systems like Flume [3] on hundreds of nodes [17].
- **Spam detection:** A social network such as Twitter may wish to identify new spam campaigns in real time using statistical learning algorithms [39].

For these applications, we believe that the 0.5–2 second latency of D-Streams is adequate, as it is well below the timescale of the trends monitored. We purposely do *not* target applications with latency needs below a few hundred milliseconds, such as high-frequency trading.

### 2.1 Goals

To run these applications at large scales, we seek a system design that meets four goals:

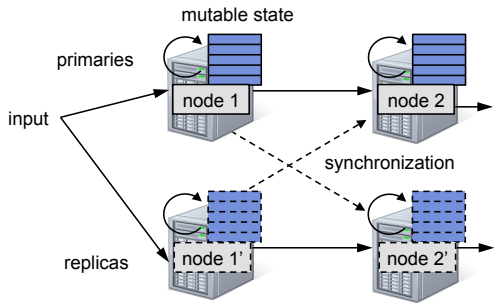
1. Scalability to hundreds of nodes.
2. Minimal cost beyond base processing—we do not wish to pay a 2× replication overhead, for example.
3. Second-scale latency.
4. Second-scale recovery from faults and stragglers.

To our knowledge, previous systems do not meet these goals: replicated systems have high overhead, while upstream backup based ones can take tens of seconds to recover lost state [33, 41], and neither tolerates stragglers.

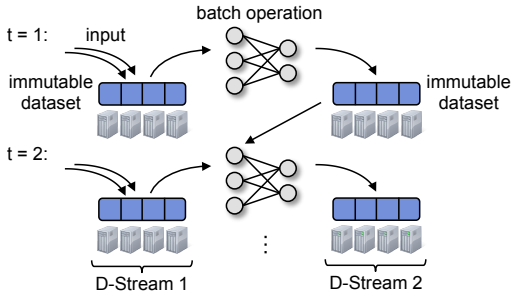
### 2.2 Previous Processing Models

Though there has been a wide set of work on distributed stream processing, most previous systems use the same *continuous operator* model. In this model, streaming computations are divided into a set of long-lived stateful operators, and each operator processes records as they arrive by updating internal state (*e.g.,* a table tracking page view counts over a window) and sending new records in response [10]. Figure 1(a) illustrates.

While continuous processing minimizes latency, the stateful nature of operators, combined with nondeterminism that arises from record interleaving on the network, makes it hard to provide fault tolerance efficiently. Specifically, the main recovery challenge is rebuilding




(a) Continuous operator processing model. Each node continuously receives records, updates internal state, and emits new records. Fault tolerance is typically achieved through replication, using a synchronization protocol like Flux or DPC [34, 5] to ensure that replicas of each node see records in the same order (*e.g.,* when they have multiple parent nodes).




(b) D-Stream processing model. In each time interval, the records that arrive are stored reliably across the cluster to form an immutable, partitioned dataset. This is then processed via deterministic parallel operations to compute other distributed datasets that represent program output or state to pass to the next interval. Each series of datasets forms one D-Stream.

**Figure 1: Comparison of traditional record-at-a-time stream processing (a) with discretized streams (b).**

the state of operators on a lost, or slow, node. Previous systems use one of two schemes, *replication* and *upstream backup* [20], which offer a sharp tradeoff between cost and recovery time.

In replication, which is common in database systems, there are two copies of the processing graph, and input records are sent to both. However, simply replicating the nodes is not enough; the system also needs to run a *synchronization protocol*, such as Flux [34] or Borealis's DPC [5], to ensure that the two copies of each operator see messages from upstream parents in the same order. For example, an operator that outputs the union of two parent streams (the sequence of all records received on either one) needs to see the parent streams in the same order to produce the same output stream, so the two copies of this operator need to coordinate. Replication is thus costly, though it recovers quickly from failures.

In upstream backup, each node retains a copy of the messages it sent since some checkpoint. When a node fails, a standby machine takes over its role, and the parents replay messages to this standby to rebuild its state. This approach thus incurs high recovery times, because a single node must recompute the lost state by running data through the serial stateful operator code. TimeStream [33] and MapReduce Online [11] use this model. Popular message queueing systems, like Storm [37], also use this approach, but typically only provide "at-least-once" delivery for *messages*, relying on the user's code to handle state recovery.[2]

More importantly, neither replication nor upstream backup handle stragglers. If a node runs slowly in the replication model, the whole system is affected because of the synchronization required to have the replicas receive messages in the same order. In upstream backup, the only way to mitigate a straggler is to treat it as a failure, which requires going through the slow state recovery process mentioned above, and is heavy-handed for a problem that may be transient.[3] Thus, while traditional streaming approaches work well at smaller scales, they face substantial problems in a large commodity cluster.

## 3 Discretized Streams (D-Streams)

D-Streams avoid the problems with traditional stream processing by structuring computations as a set of *short, stateless, deterministic tasks* instead of continuous, stateful operators. They then store the state in memory across tasks as fault-tolerant data structures (RDDs) that can be recomputed deterministically. Decomposing computations into short tasks exposes dependencies at a fine granularity and allows powerful recovery techniques like parallel recovery and speculation. Beyond fault tolerance, the D-Stream model gives other benefits, such as powerful unification with batch processing.

### 3.1 Computation Model

We treat a streaming computation as a series of deterministic batch computations on small time intervals. The data received in each interval is stored reliably across the cluster to form an *input dataset* for that interval. Once the time interval completes, this dataset is processed via deterministic parallel operations, such as *map*, *reduce* and *groupBy*, to produce new datasets representing either program outputs or intermediate state. In the former case, the results may be pushed to an external sys-

[2] Storm's Trident layer [26] automatically keeps state in a replicated database instead, writing updates in batches. This is expensive, as all updates must be replicated transactionally across the network.

[3] Note that a speculative execution approach as in batch systems would be challenging to apply here because the operator code assumes that it is fed inputs serially, so even a backup copy of an operator would need to spend a long time recovering from its last checkpoint.

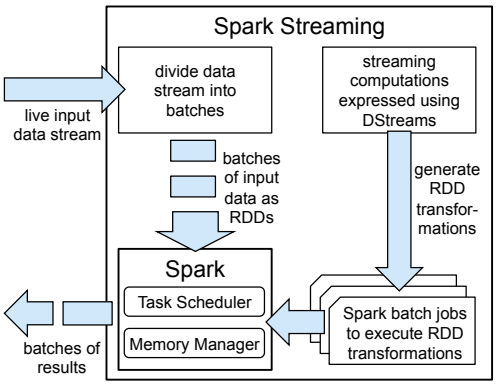



**Figure 2: High-level overview of the Spark Streaming system. Spark Streaming divides input data streams into batches and stores them in Spark's memory. It then executes a streaming application by generating Spark jobs to process the batches.**

tem in a distributed manner. In the latter case, the intermediate state is stored as *resilient distributed datasets (RDDs)* [43], a fast storage abstraction that avoids replication by using lineage for recovery, as we shall explain. This state dataset may then be processed along with the next batch of input data to produce a new dataset of updated intermediate states. Figure 1(b) shows our model.

We implemented our system, Spark Streaming, based on this model. We used Spark [43] as our batch processing engine for each batch of data. Figure 2 shows a high-level sketch of the computation model in the context of Spark Streaming. This is explained in more detail later.

In our API, users define programs by manipulating objects called *discretized streams (D-Streams)*. A D-Stream is a sequence of immutable, partitioned datasets (RDDs) that can be acted on by deterministic *transformations*. These transformations yield new D-Streams, and may create intermediate *state* in the form of RDDs.

We illustrate the idea with a Spark Streaming program that computes a running count of view events by URL. Spark Streaming exposes D-Streams through a functional API similar to LINQ [42, 2] in the Scala programming language.[4] The code for our program is:

```
pageViews = readStream("http://...", "1s")
ones = pageViews.map(event => (event.url, 1))
counts = ones.runningReduce((a, b) => a + b)
```

This code creates a D-Stream called `pageViews` by reading an event stream over HTTP, and groups these into 1-second intervals. It then transforms the event stream to get a new D-Stream of (URL, 1) pairs called `ones`, and performs a running count of these with a stateful *runningReduce* transformation. The arguments to *map* and *runningReduce* are Scala function literals.

[4]Other interfaces, such as streaming SQL, would also be possible.

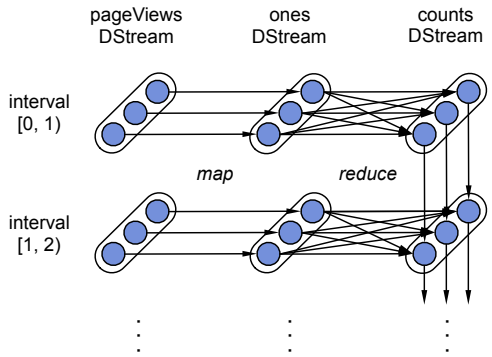



**Figure 3: Lineage graph for RDDs in the view count program. Each oval is an RDD, with partitions shown as circles. Each sequence of RDDs is a D-Stream.**

To execute this program, Spark Streaming will receive the data stream, divide it into one second batches and store them in Spark's memory as RDDs (see Figure 2). Additionally, it will invoke RDD transformations like *map* and *reduce* to process the RDDs. To execute these transformations, Spark will first launch *map* tasks to process the events and generate the url-one pairs. Then it will launch *reduce* tasks that take both the results of the maps and the results of the previous interval's reduces, stored in an RDD. These tasks will produce a new RDD with the updated counts. Each D-Stream in the program thus turns into a sequence of RDDs.

Finally, to recover from faults and stragglers, both D-Streams and RDDs track their *lineage*, that is, the graph of deterministic operations used to build them [43]. Spark tracks this information at the level of *partitions* within each distributed dataset, as shown in Figure 3. When a node fails, it recomputes the RDD partitions that were on it by re-running the tasks that built them from the original input data stored reliably in the cluster. The system also periodically checkpoints state RDDs (*e.g.,* by asynchronously replicating every tenth RDD)[5] to prevent infinite recomputation, but this does not need to happen for all data, because recovery is often fast: the lost partitions can be recomputed *in parallel* on separate nodes. In a similar way, if a node straggles, we can speculatively execute copies of its tasks on other nodes [12], because they will produce the same result.

We note that the parallelism usable for recovery in D-Streams is higher than in upstream backup, even if one ran multiple operators per node. D-Streams expose parallelism across both *partitions* of an operator and *time*:

1. Much like batch systems run multiple tasks per node, each timestep of a transformation may create multiple RDD partitions per node (*e.g.,* 1000 RDD partitions on a 100-core cluster). When the node fails, we can recompute its partitions in parallel on others.

[5]Since RDDs are immutable, checkpointing does not block the job.

2. The lineage graph often enables data from different timesteps to be rebuilt in parallel. For example, in Figure 3, if a node fails, we might lose some *map* outputs from each timestep; the maps from different timesteps can be rerun in parallel, which would not be possible in a continuous operator system that assumes serial execution of each operator.

Because of these properties, D-Streams can parallelize recovery over hundreds of cores and recover in 1–2 seconds even when checkpointing every 30s (§6.2).

In the rest of this section, we describe the guarantees and programming interface of D-Streams in more detail. We then return to our implementation in Section 4.

### 3.2 Timing Considerations

Note that D-Streams place records into input datasets based on the time when each record *arrives* at the system. This is necessary to ensure that the system can always start a new batch on time, and in applications where the records are generated in the same location as the streaming program, *e.g.,* by services in the same datacenter, it poses no problem for semantics. In other applications, however, developers may wish to group records based on an *external timestamp* of when an event happened, *e.g.,* when a user clicked a link, and records may arrive out of order. D-Streams provide two means to handle this case:

1. The system can *wait* for a limited "slack time" before starting to process each batch.
2. User programs can correct for late records at the *application level*. For example, suppose that an application wishes to count clicks on an ad between time $t$ and $t+1$. Using D-Streams with an interval size of one second, the application could provide a count for the clicks received between $t$ and $t+1$ as soon as time $t+1$ passes. Then, in future intervals, the application could collect any further events with external timestamps between $t$ and $t+1$ and compute an updated result. For example, it could output a *new* count for time interval $[t,t+1)$ at time $t+5$, based on the records for this interval received between $t$ and $t+5$. This computation can be performed with an efficient incremental *reduce* operation that adds the old counts computed at $t+1$ to the counts of new records since then, avoiding wasted work. This approach is similar to order-independent processing [23].

These timing concerns are inherent to stream processing, as any system must handle external delays. They have been studied in detail in databases [23, 36]. In general, any such technique can be implemented over D-Streams by "discretizing" its computation in small batches (running the same logic in batches). Thus, we do not explore these approaches further in this paper.

### 3.3 D-Stream API

Because D-Streams are primarily an execution strategy (describing how to break a computation into steps), they can be used to implement many of the standard operations in streaming systems, such as sliding windows and incremental processing [10, 4], by simply batching their execution into small timesteps. To illustrate, we describe the operations in Spark Streaming, though other interfaces (*e.g.,* SQL) could also be supported.

In Spark Streaming, users register one or more streams using a functional API. The program can define *input* streams to be read from outside, which receive data either by having nodes listen on a port or by loading it periodically from a storage system (*e.g.,* HDFS). It can then apply two types of operations to these streams:

- *Transformations*, which create a new D-Stream from one or more parent streams. These may be *stateless*, applying separately on the RDDs in each time interval, or they may produce state across intervals.
- *Output operations*, which let the program write data to external systems. For example, the *save* operation will output each RDD in a D-Stream to a database.

D-Streams support the same stateless transformations available in typical batch frameworks [12, 42], including *map*, *reduce*, *groupBy*, and *join*. We provide all the operations in Spark [43]. For example, a program could run a canonical MapReduce word count on each time interval of a D-Stream of words using the following code:

```
pairs = words.map(w => (w, 1))
counts = pairs.reduceByKey((a, b) => a + b)
```

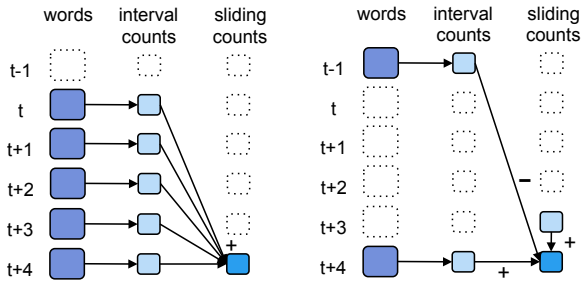
In addition, D-Streams provide several *stateful* transformations for computations spanning multiple intervals, based on standard stream processing techniques such as sliding windows [10, 4]. These include:

**Windowing:** The *window* operation groups all the records from a sliding window of past time intervals into one RDD. For example, calling `words.window("5s")` in the code above yields a D-Stream of RDDs containing the words in intervals $[0,5)$, $[1,6)$, $[2,7)$, etc.

**Incremental aggregation:** For the common use case of computing an aggregate, like a count or max, over a sliding window, D-Streams have several variants of an incremental *reduceByWindow* operation. The simplest one only takes an associative merge function for combining values. For instance, in the code above, one can write:

```
pairs.reduceByWindow("5s", (a, b) => a + b)
```

This computes a per-interval count for each time interval only once, but has to add the counts for the past five seconds repeatedly, as shown in Figure 4(a). If the aggregation function is also *invertible*, a more efficient version also takes a function for "subtracting" values and main-




(a) Associative only (b) Associative & invertible

**Figure 4: *reduceByWindow* execution for the associative-only and associative+invertible versions of the operator. Both versions compute a per-interval count only once, but the second avoids re-summing each window. Boxes denote RDDs, while arrows show the operations used to compute window $[t, t+5)$.**

tains the state incrementally (Figure 4(b)):

```
pairs.reduceByWindow("5s", (a,b) => a+b, (a,b) => a-b)
```

**State tracking:** Often, an application has to track *states* for various objects in response to a stream of events indicating state changes. For example, a program monitoring online video delivery may wish to track the number of active *sessions*, where a session starts when the system receives a "join" event for a new client and ends when it receives an "exit" event. It can then ask questions such as "how many sessions have a bitrate above $X$."
D-Streams provide a *track* operation that transforms streams of (Key, Event) records into streams of (Key, State) records based on three arguments:

- An *initialize* function for creating a State from the first Event for a new key.
- An *update* function for returning a new State given an old State and an Event for its key.
- A *timeout* for dropping old states.

For example, one could count the active sessions from a stream of (ClientID, Event) pairs called as follows:

```
sessions = events.track(
  (key, ev) => 1,          // initialize function
  (key, st, ev) =>         // update function
    ev == Exit ? null : 1,
  "30s")                   // timeout
counts = sessions.count()  // a stream of ints
```

This code sets each client's state to `1` if it is active and drops it by returning `null` from update when it leaves. Thus, `sessions` contains a (ClientID, 1) element for each active client, and `counts` counts the sessions.

These operators are all implemented using the batch operators in Spark, by applying them to RDDs from different times in parent streams. For example, Figure 5

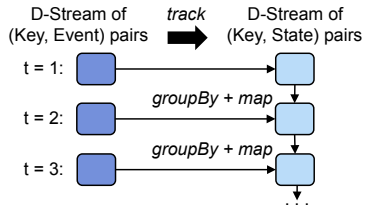



**Figure 5: RDDs created by the *track* operation.**

shows the RDDs built by *track*, which works by grouping the old states and the new events for each interval.

Finally, the user calls *output operators* to send results out of Spark Streaming into external systems (*e.g.,* for display on a dashboard). We offer two such operators: *save*, which writes each RDD in a D-Stream to a storage system (*e.g.,* HDFS or HBase), and *foreachRDD*, which runs a user code snippet (any Spark code) on each RDD. For example, a user can print the top K counts with `counts.foreachRDD(rdd => print(rdd.top(K)))`.

### 3.4 Consistency Semantics

One benefit of D-Streams is that they provide clean consistency semantics. Consistency of state across nodes can be a problem in streaming systems that process each record eagerly. For instance, consider a system that counts page views by country, where each page view event is sent to a different node responsible for aggregating statistics for its country. If the node responsible for England falls behind the node for France, *e.g.,* due to load, then a snapshot of their states would be inconsistent: the counts for England would reflect an older prefix of the stream than the counts for France, and would generally be lower, confusing inferences about the events. Some systems, like Borealis [5], synchronize nodes to avoid this problem, while others, like Storm, ignore it.

With D-Streams, the consistency semantics are clear, because time is naturally discretized into intervals, and each interval's output RDDs reflect *all* of the input received in that and previous intervals. This is true regardless of whether the output and state RDDs are distributed across the cluster—users do not need to worry about whether nodes have fallen behind each other. Specifically, the result in each output RDD, when computed, is the same as if all the batch jobs on previous intervals had run in lockstep and there were no stragglers and failures, simply due to the determinism of computations and the separate naming of datasets from different intervals. Thus, D-Streams provide consistent, "exactly-once" processing across the cluster.

### 3.5 Unification with Batch & Interactive Processing

Because D-Streams follow the same processing model, data structures (RDDs), and fault tolerance mechanisms as batch systems, the two can seamlessly be combined.

| Aspect | D-Streams | Continuous proc. systems |
|---|---|---|
| Latency | 0.5–2 s | 1–100 ms unless records are batched for consistency |
| Consistency | Records processed atomically with interval they arrive in | Some systems wait a short time to sync operators before proceeding [5, 33] |
| Late records | Slack time or app-level correction | Slack time, out of order processing [23, 36] |
| Fault recovery | Fast parallel recovery | Replication or serial recovery on one node |
| Straggler recovery | Possible via speculative execution | Typically not handled |
| Mixing w/ batch | Simple unification through RDD APIs | In some DBs [15]; not in message queueing systems |

**Table 1: Comparing D-Streams with record-at-a-time systems.**

Spark Streaming provides several powerful features to unify streaming and batch processing.

First, D-Streams can be combined with static RDDs computed using a standard Spark job. For instance, one can *join* a stream of message events against a precomputed spam filter, or compare them with historical data.

Second, users can run a D-Stream program on previous historical data using a "batch mode." This makes it easy compute a new streaming report on past data.

Third, users run ad-hoc queries on D-Streams *interactively* by attaching a Scala console to their Spark Streaming program and running arbitrary Spark operations on the RDDs there. For example, the user could query the most popular words in a time range by typing:

```
counts.slice("21:00", "21:05").topK(10)
```

Discussions with developers who have written both offline (Hadoop-based) and online processing applications show that these features have significant practical value. Simply having the data types and functions used for these programs in the same codebase saves substantial development time, as streaming and batch systems currently have separate APIs. The ability to also query state in the streaming system interactively is even more attractive: it makes it simple to debug a running computation, or to ask queries that were not anticipated when defining the aggregations in the streaming job, *e.g.,* to troubleshoot an issue with a website. Without this ability, users typically need to wait tens of minutes for the data to make it into a batch cluster, even though all the relevant state is in memory on stream processing nodes.

### 3.6 Summary

To end our overview of D-Streams, we compare them with continuous operator systems in Table 1. The main difference is that D-Streams divide work into small, deterministic tasks operating on batches. This raises their minimum latency, but lets them employ highly efficient recovery techniques. In fact, some continuous operator systems, like TimeStream and Borealis [33, 5], *also* delay records, in order to deterministically execute operators that have multiple upstream parents (by waiting for periodic "punctuations" in streams) and to provide consistency. This raises their latency past the millisecond scale and into the second scale of D-Streams.

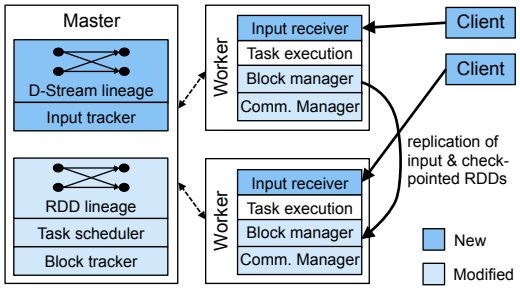



**Figure 6: Components of Spark Streaming, showing what we added and modified over Spark.**

## 4 System Architecture

We have implemented D-Streams in a system called Spark Streaming, based on a modified version of the Spark processing engine [43]. Spark Streaming consists of three components, shown in Figure 6:

- A *master* that tracks the D-Stream lineage graph and schedules *tasks* to compute new RDD partitions.
- *Worker nodes* that receive data, store the partitions of input and computed RDDs, and execute tasks.
- A *client library* used to send data into the system.

As shown in the figure, Spark Streaming reuses many components of Spark, but we also modified and added multiple components to enable streaming. We discuss those changes in Section 4.2.

From an architectural point of view, the main difference between Spark Streaming and traditional streaming systems is that Spark Streaming divides its computations into short, stateless, deterministic *tasks*, each of which may run on any node in the cluster, or even on multiple nodes. Unlike the rigid topologies in traditional systems, where moving part of the computation to another machine is a major undertaking, this approach makes it straightforward to balance load across the cluster, react to failures, or launch speculative copies of slow tasks. It matches the approach used in batch systems, such as MapReduce, for the same reasons. However, tasks in Spark Streaming are far shorter, usually just 50–200 ms, due to running on in-memory RDDs.

All state in Spark Streaming is stored in fault-tolerant data structures (RDDs), instead of being part of a long-running operator process as in previous systems. RDD partitions can reside on any node, and can even be com-

puted on multiple nodes, because they are computed deterministically. The system tries to place both state and tasks to maximize data locality, but this underlying flexibility makes speculation and parallel recovery possible.

These benefits come naturally from running on a batch platform (Spark), but we also had to make significant changes to support streaming. We discuss job execution in more detail before presenting these changes.

### 4.1 Application Execution

Spark Streaming applications start by defining one or more input streams. The system can load streams either by receiving records directly from clients, or by loading data periodically from an external storage system, such as HDFS, where it might be placed by a log collection system [3]. In the former case, we ensure that new data is replicated across two worker nodes before sending an acknowledgement to the client library, because D-Streams require input data to be stored reliably to recompute results. If a worker fails, the client library sends unacknowledged data to another worker.

All data is managed by a *block store* on each worker, with a tracker on the master to let nodes find the locations of blocks. Because both our input blocks and the RDD partitions we compute from them are immutable, keeping track of the block store is straightforward—each block is simply given a unique ID, and any node that has that ID can serve it (*e.g.,* if multiple nodes computed it). The block store keeps new blocks in memory but drops them in an LRU fashion, as described later.

To decide when to start processing a new interval, we assume that the nodes have their clocks synchronized via NTP, and have each node send the master a list of block IDs it received in each interval when it ends. The master then starts launching tasks to compute the output RDDs for the interval, *without* requiring any further kind of synchronization. Like other batch schedulers [22], it simply starts each task whenever its parents are finished.

Spark Streaming relies on Spark's existing batch scheduler within each timestep [43], and performs many of the optimizations in systems like DryadLINQ [42]:

- It pipelines operators that can be grouped into a single task, such as a *map* followed by another *map*.
- It places tasks based on data locality.
- It controls the *partitioning* of RDDs to avoid shuffling data across the network. For example, in a *reduceByWindow* operation, each interval's tasks need to "add" the new partial results from the current interval (*e.g.,* a click count for each page) and "subtract" the results from several intervals ago. The scheduler partitions the state RDDs for different intervals in the same way, so that data for each key (*e.g.,* a page) is consistently on the same node across timesteps. More details are given in [43].

### 4.2 Optimizations for Stream Processing

While Spark Streaming builds on Spark, we also had to make sigificant optimizations and changes to this batch engine to support streaming. These included:

**Network communication:** We rewrote Spark's data plane to use asynchronous I/O to let tasks with remote inputs, such as reduce tasks, fetch them faster.

**Timestep pipelining:** Because the tasks inside each timestep may not perfectly utilize the cluster (*e.g.,* at the end of the timestep, there might only be a few tasks left running), we modified Spark's scheduler to allow submitting tasks from the next timestep *before* the current one has finished. For example, consider our first *map* + *runningReduce* job in Figure 3. Because the maps at each step are independent, we can begin running the maps for timestep 2 before timestep 1's reduce finishes.

**Task Scheduling:** We made multiple optimizations to Spark's task scheduler, such as hand-tuning the size of control messages, to be able to launch parallel jobs of hundreds of tasks every few hundred milliseconds.

**Storage layer:** We rewrote Spark's storage layer to support asynchronous checkpointing of RDDs and to increase performance. Because RDDs are immutable, they can be checkpointed over the network without blocking computations on them and slowing jobs. The new storage layer also uses zero-copy I/O for this when possible.

**Lineage cutoff:** Because lineage graphs between RDDs in D-Streams can grow indefinitely, we modified the scheduler to forget lineage after an RDD has been checkpointed, so that its state does not grow arbitrarily. Similarly, other data structures in Spark that grew without bound were given a periodic cleanup process.

**Master recovery:** Because streaming applications need to run 24/7, we added support for recovering the Spark master's state if it fails (Section 5.3).

Interestingly, the optimizations for stream processing also improved Spark's performance in batch benchmarks by as much as $2\times$. This is a powerful benefit of using the same engine for stream and batch processing.

### 4.3 Memory Management

In our current implementation of Spark Streaming, each node's block store manages RDD partitions in an LRU fashion, dropping data to disk if there is not enough memory. In addition, the user can set a maximum history timeout, after which the system will simply forget old blocks without doing disk I/O (this timeout must be bigger than the checkpoint interval). We found that in many applications, the memory required by Spark Streaming is not onerous, because the state within a computation is typically much smaller than the input data (many appli-

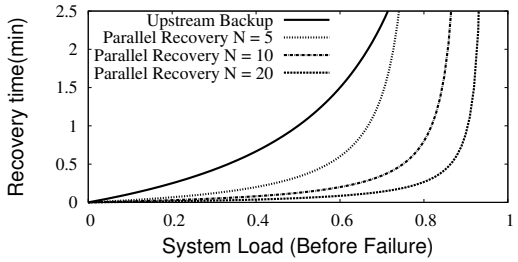



**Figure 7: Recovery time for single-node upstream backup vs. parallel recovery on $N$ nodes, as a function of the load before a failure. We assume the time since the last checkpoint is 1 min.**

cations compute aggregate statistics), and any reliable streaming system needs to replicate data received over the network to multiple nodes, as we do. However, we also plan to explore ways to prioritize memory use.

# 5 Fault and Straggler Recovery

The deterministic nature of D-Streams makes it possible to use two powerful recovery techniques for worker state that are hard to apply in traditional streaming systems: parallel recovery and speculative execution. In addition, it simplifies master recovery, as we shall also discuss.

## 5.1 Parallel Recovery

When a node fails, D-Streams allow the state RDD partitions that were on the node, and all tasks that it was currently running, to be recomputed in parallel on other nodes. The system periodically *checkpoints* some of the state RDDs, by asynchronously replicating them to other worker nodes.[6] For example, in a program computing a running count of page views, the system could choose to checkpoint the counts every minute. Then, when a node fails, the system detects all missing RDD partitions and launches tasks to recompute them from the last checkpoint. Many tasks can be launched *at the same time* to compute different RDD partitions, allowing the whole cluster to partake in recovery. As described in Section 3, D-Streams exploit parallelism both across *partitions* of the RDDs in each timestep and across *timesteps* for independent operations (*e.g.,* an initial *map*), as the lineage graph captures dependencies at a fine granularity.

To show the benefit of parallel recovery, Figure 7 compares it with single-node upstream backup using a simple analytical model. The model assumes that the system is recovering from a minute-old checkpoint.

In the upstream backup line, a single idle machine performs all of the recovery and then starts processing new records. It takes a long time to catch up at high loads because new records for it continue to arrive while it is rebuilding old state. Indeed, suppose that the load before failure was $\lambda$. Then during each minute of recovery, the backup node can do 1 min of work, but receives $\lambda$ minutes of new work. Thus, it fully recovers from the $\lambda$ units of work that the failed node did since the last checkpoint at a time $t_{\text{up}}$ such that $t_{\text{up}} \cdot 1 = \lambda + t_{\text{up}} \cdot \lambda$, which is

$$t_{\text{up}} = \frac{\lambda}{1-\lambda}.$$

In the other lines, all of the machines partake in recovery, while also processing new records. Supposing there where $N$ machines in the cluster before the failure, the remaining $N-1$ machines now each have to recover $\lambda/N$ work, but also receive new data at a rate of $\frac{N}{N-1}\lambda$. The time $t_{\text{par}}$ at which they catch up with the arriving stream satisfies $t_{\text{par}} \cdot 1 = \frac{\lambda}{N} + t_{\text{par}} \cdot \frac{N}{N-1}\lambda$, which gives

$$t_{\text{par}} = \frac{\lambda/N}{1-\frac{N}{N-1}\lambda} \approx \frac{\lambda}{N(1-\lambda)}.$$

Thus, with more nodes, parallel recovery catches up with the arriving stream much faster than upstream backup.

## 5.2 Straggler Mitigation

Besides failures, another concern in large clusters is stragglers [12]. Fortunately, D-Streams also let us mitigate stragglers like batch systems do, by running speculative backup copies of slow tasks. Such speculation would be difficult in a continuous operator system, as it would require launching a new copy of a node, populating its state, and overtaking the slow copy. Indeed, replication algorithms for stream processing, such as Flux and DPC [34, 5], focus on *synchronizing* two replicas.

In our implementation, we use a simple threshold to detect stragglers: whenever a task runs more than $1.4\times$ longer than the median task in its job stage, we mark it as slow. More refined algorithms could also be used, but we show that this method still works well enough to recover from stragglers within a second.

## 5.3 Master Recovery

A final requirement to run Spark Streaming 24/7 was to tolerate failures of Spark's master. We do this by (1) writing the state of the computation reliably when starting each timestep and (2) having workers connect to a new master and report their RDD partitions to it when the old master fails. A key aspect of D-Streams that simplifies recovery is that *there is no problem if a given RDD is computed twice*. Because operations are deterministic, such an outcome is similar to recovering from a failure.[7] This means that it is fine to lose some running tasks while the master reconnects, as they can be redone.

[6] Because RDDs are immutable, checkpointing does not block the current timestep's execution.

[7] One subtle issue here is output operators; we have designed operators like *save* to be idempotent, so that the operator outputs each timestep's worth of data to a known path, and does not overwrite previous data if that timestep was already computed.

Our current implementation stores D-Stream metadata in HDFS, writing (1) the graph of the user's D-Streams and Scala function objects representing user code, (2) the time of the last checkpoint, and (3) the IDs of RDDs since the checkpoint in an HDFS file that is updated through an atomic rename on each timestep. Upon recovery, the new master reads this file to find where it left off, and reconnects to the workers to determine which RDD partitions are in memory on each one. It then resumes processing each timestep missed. Although we have not yet optimized the recovery process, it is reasonably fast, with a 100-node cluster resuming work in 12 seconds.

# 6 Evaluation

We evaluated Spark Streaming using both several benchmark applications and by porting two real applications to it: a commercial video distribution monitoring system and a machine learning algorithm for estimating traffic conditions from automobile GPS data [19]. These latter applications also leverage D-Streams' unification with batch processing, as we shall discuss.

## 6.1 Performance

We tested the performance of the system using three applications of increasing complexity: Grep, which finds the number of input strings matching a pattern; WordCount, which performs a sliding window count over 30s; and TopKCount, which finds the $k$ most frequent words over the past 30s. The latter two applications used the incremental *reduceByWindow* operator. We first report the raw scaling performance of Spark Streaming, and then compare it against two widely used streaming systems, S4 from Yahoo! and Storm from Twitter [29, 37]. We ran these applications on "m1.xlarge" nodes on Amazon EC2, each with 4 cores and 15 GB RAM.

Figure 8 reports the maximum throughput that Spark Streaming can sustain while keeping the end-to-end latency below a given target. By "end-to-end latency," we mean the time from when records are sent to the system to when results incorporating them appear. Thus, the latency includes the time to wait for a new input batch to start. For a 1 second latency target, we use 500 ms input intervals, while for a 2 s target, we use 1 s intervals. In both cases, we used 100-byte input records.

We see that Spark Streaming scales nearly linearly to 100 nodes, and can process up to 6 GB/s (64M records/s) at sub-second latency on 100 nodes for Grep, or 2.3 GB/s (25M records/s) for the other, more CPU-intensive jobs.[8] Allowing a larger latency improves throughput slightly, but even the performance at sub-second latency is high.

[8] Grep was network-bound due to the cost to replicate the input data to multiple nodes—we could not get the EC2 network to send more than 68 MB/s per node. WordCount and TopK were more CPU-heavy, as they do more string processing (hashes & comparisons).

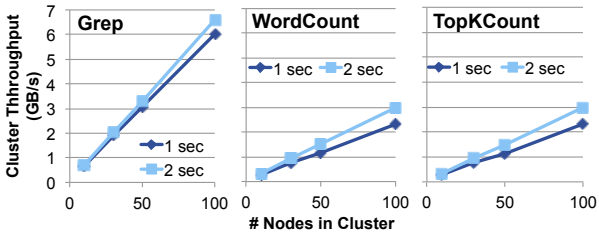



**Figure 8: Maximum throughput attainable under a given latency bound (1 s or 2 s) by Spark Streaming.**

**Comparison with Commercial Systems** Spark Streaming's per-node throughput of 640,000 records/s for Grep and 250,000 records/s for TopKCount on 4-core nodes is comparable to the speeds reported for commercial single-node streaming systems. For example, Oracle CEP reports a throughput of 1 million records/s on a 16-core machine [31], StreamBase reports 245,000 records/s on 8 cores [40], and Esper reports 500,000 records/s on 4 cores [13]. While there is no reason to expect D-Streams to be slower or faster per-node, the key advantage is that Spark Streaming scales nearly linearly to 100 nodes.

**Comparison with S4 and Storm** We also compared Spark Streaming against two open source distributed streaming systems, S4 and Storm. Both are continuous operators systems that do not offer consistency across nodes and have limited fault tolerance guarantees (S4 has none, while Storm guarantees at-least-once delivery of records). We implemented our three applications in both systems, but found that S4 was limited in the number of records/second it could process per node (at most 7500 records/s for Grep and 1000 for WordCount), which made it almost 10× slower than Spark and Storm. Because Storm was faster, we also tested it on a 30-node cluster, using both 100-byte and 1000-byte records.

We compare Storm with Spark Streaming in Figure 9, reporting the throughput Spark attains at sub-second latency. We see that Storm is still adversely affected by smaller record sizes, capping out at 115K records/s/node for Grep for 100-byte records, compared to 670K for Spark. This is despite taking several precautions in our Storm implementation to improve performance, including sending "batched" updates from Grep every 100 input records and having the "reduce" nodes in WordCount and TopK only send out new counts every second, instead of each time a count changes. Storm was faster with 1000-byte records, but still 2× slower than Spark.

## 6.2 Fault and Straggler Recovery

We evaluated fault recovery under various conditions using the WordCount and Grep applications. We used 1-second batches with input data residing in HDFS, and set the data rate to 20 MB/s/node for WordCount and

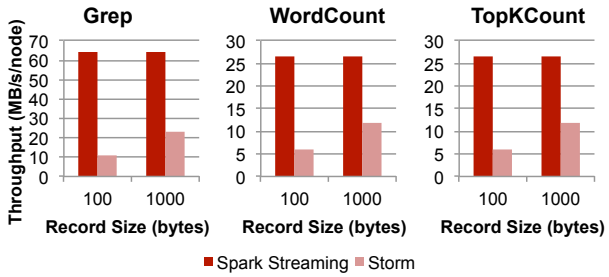



**Figure 9: Throughput vs Storm on 30 nodes.**

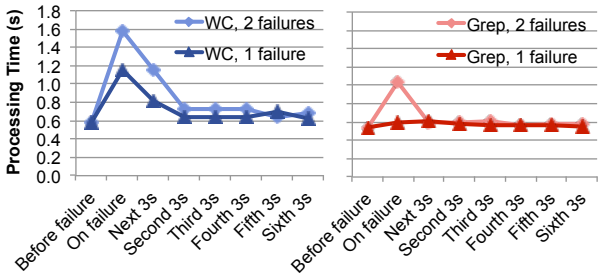



**Figure 10: Interval processing times for WordCount (WC) and Grep under failures. We show the average time to process each 1s batch of data before a failure, during the interval of the failure, and during 3-second periods after. Results are over 5 runs.**

80 MB/s/node for Grep, which led to a roughly equal per-interval processing time of 0.58s for WordCount and 0.54s for Grep. Because the WordCount job performs an incremental *reduceByKey*, its lineage graph grows indefinitely (since each interval subtracts data from 30 seconds in the past), so we gave it a checkpoint interval of 10 seconds. We ran the tests on 20 four-core nodes, using 150 map tasks and 10 reduce tasks per job.

We first report recovery times under these these base conditions, in Figure 10. The plot shows the average processing time of 1-second data intervals before the failure, during the interval of failure, and during 3-second periods thereafter, for either 1 or 2 concurrent failures. (The processing for these later periods is delayed while recovering data for the interval of failure, so we show how the system restabilizes.) We see that recovery is fast, with delays of at most 1 second even for two failures and a 10s checkpoint interval. WordCount's recovery takes longer because it has to recompute data going far back, whereas Grep just loses four tasks on each failed node.

**Varying the Checkpoint Interval** Figure 11 shows the effect of changing WordCount's checkpoint interval. Even when checkpointing every 30s, results are delayed at most 3.5s. With 2s checkpoints, the system recovers in just 0.15s, while still paying less than full replication.

**Varying the Number of Nodes** To see the effect of parallelism, we also tried the WordCount application on

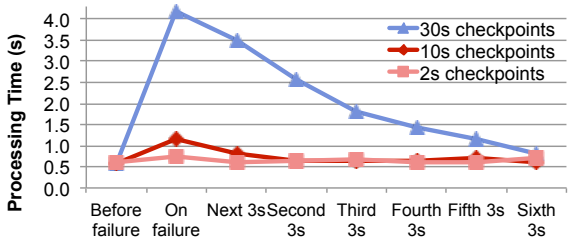



**Figure 11: Effect of checkpoint time in WordCount.**

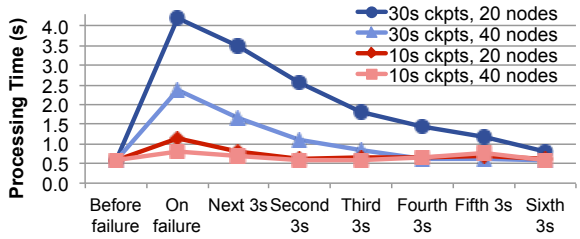



**Figure 12: Recovery of WordCount on 20 & 40 nodes.**

40 nodes. As Figure 12 shows, doubling the nodes reduces the recovery time in half. While it may seem surprising that there is so much parallelism given the linear dependency chain of the sliding *reduceByWindow* operator in WordCount, the parallelism comes because the *local* aggregations on each timestep can be done in parallel (see Figure 4), and these are the bulk of the work.

**Straggler Mitigation** Finally, we tried slowing down one of the nodes instead of killing it, by launching a 60-thread process that overloaded the CPU. Figure 13 shows the per-interval processing times without the straggler, with the straggler but with speculative execution (backup tasks) disabled, and with the straggler and speculation enabled. Speculation improves the response time significantly. Note that our current implementation does *not* attempt to remember straggler nodes across time, so these improvements occur despite repeatedly launching new tasks on the slow node. This shows that even unexpected stragglers can be handled quickly. A full implementation would blacklist slow nodes.

### 6.3 Real Applications

We evaluated the expressiveness of D-Streams by porting two real applications. Both applications are significantly more complex than the test programs shown so far, and both took advantage of D-Streams to perform batch or interactive processing in addition to streaming.

#### 6.3.1 Video Distribution Monitoring

Conviva provides a commercial management platform for video distribution over the Internet. One feature of this platform is the ability to track the performance across different geographic regions, CDNs, client devices, and ISPs, which allows the broadcasters to quickly

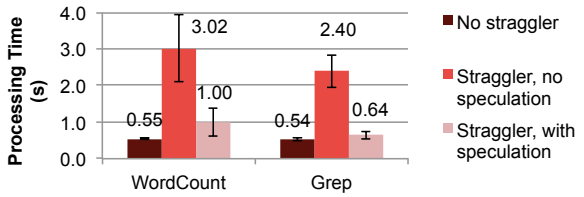
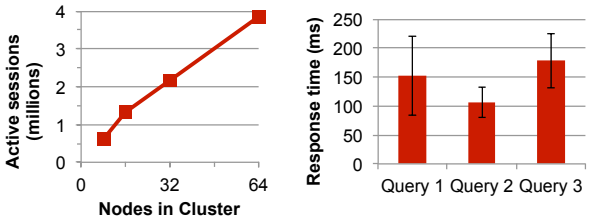



**Figure 13: Processing time of intervals in Grep and WordCount in normal operation, as well as in the presence of a straggler, with and without speculation.**




(a) Scalability (b) Ad-hoc queries

**Figure 14: Results for the video application. (a) shows the number of client sessions supported vs. cluster size. (b) shows the performance of three ad-hoc queries from the Spark shell, which count (1) all active sessions, (2) sessions for a specific customer, and (3) sessions that have experienced a failure.**

idenify and respond to delivery problems. The system receives events from video players and uses them to compute more than fifty metrics, including complex metrics such as unique viewers and session-level metrics such as buffering ratio, over different grouping categories.

The current application is implemented in two systems: a custom-built distributed streaming system for live data, and a Hadoop/Hive implementation for historical data and ad-hoc queries. Having both live and historical data is crucial because customers often want to go back in time to debug an issue, but implementing the application on these two separate systems creates significant challenges. First, the two implementations have to be kept in sync to ensure that they compute metrics in the same way. Second, there is a lag of several minutes minutes before data makes it through a sequence of Hadoop import jobs into a form ready for ad-hoc queries.

We ported the application to D-Streams by wrapping the *map* and *reduce* implementations in the Hadoop version. Using a 500-line Spark Streaming program and an additional 700-line wrapper that executed Hadoop jobs within Spark, we were able to compute all the metrics (a 2-stage MapReduce job) in batches as small as 2 seconds. Our code uses the *track* operator described in Section 3.3 to build a session state object for each client ID and update it as events arrive, followed by a sliding *reduceByKey* to aggregate the metrics over sessions.

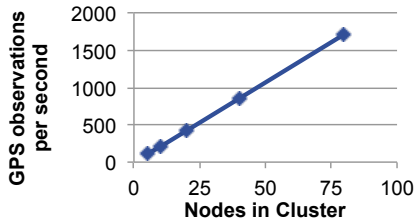



**Figure 15: Scalability of the *Mobile Millennium* job.**

We measured the scaling performance of the application and found that on 64 quad-core EC2 nodes, it could process enough events to support 3.8 million concurrent viewers, which exceeds the peak load experienced at Conviva so far. Figure 14(a) shows the scaling.

In addition, we used D-Streams to add a *new* feature not present in the original application: ad-hoc queries on the live stream state. As shown in Figure 14(b), Spark Streaming can run ad-hoc queries from a Scala shell in less than a second on the RDDs representing session state. Our cluster could easily keep ten minutes of data in RAM, closing the gap between historical and live processing, and allowing a single codebase to do both.

### 6.3.2 Crowdsourced Traffic Estimation

We applied the D-Streams to the *Mobile Millennium* traffic information system [19], a machine learning based project to estimate automobile traffic conditions in cities. While measuring traffic for highways is straightforward due to dedicated sensors, *arterial roads* (the roads in a city) lack such infrastructure. *Mobile Millennium* attacks this problem by using crowdsourced GPS data from fleets of GPS-equipped cars (*e.g.,* taxi cabs) and cellphones running a mobile application.

Traffic estimation from GPS data is challenging, because the data is noisy (due to GPS inaccuracy near tall buildings) and sparse (the system only receives one measurement from each car per minute). *Mobile Millennium* uses a highly compute-intensive expectation maximization (EM) algorithm to infer the conditions, using Markov Chain Monte Carlo and a traffic model to estimate a travel time distribution for each road link. The previous implementation [19] was an iterative batch job in Spark that ran over 30-minute windows of data.

We ported this application to Spark Streaming using an online version of the EM algorithm that merges in new data every 5 seconds. The implementation was 260 lines of Spark Streaming code, and wrapped the existing *map* and *reduce* functions in the offline program. In addition, we found that only using the real-time data could cause overfitting, because the data received in five seconds is so sparse. We took advantage of D-Streams to also combine this data with *historical* data from the same time during the past ten days to resolve this problem.

Figure 15 shows the performance of the algorithm on

up to 80 quad-core EC2 nodes. The algorithm scales nearly perfectly because it is CPU-bound, and provides answers more than 10× faster than the batch version.[9]

## 7 Discussion

We have presented discretized streams (D-Streams), a new stream processing model for clusters. By breaking computations into short, deterministic tasks and storing state in lineage-based data structures (RDDs), D-Streams can use powerful recovery mechanisms, similar to those in batch systems, to handle faults and stragglers.

Perhaps the main limitation of D-Streams is that they have a fixed minimum latency due to batching data. However, we have shown that total delay can still be as low as 1–2 seconds, which is enough for many real-world use cases. Interestingly, even some continuous operator systems, such as Borealis and TimeStream [5, 33], add delays to ensure determinism: Borealis's SUnion operator and TimeStream's HashPartition wait to batch data at "heartbeat" boundaries so that operators with multiple parents see input in a deterministic order. Thus, D-Streams' latency is in a similar range to these systems, while offering significantly more efficient recovery.

Beyond their recovery benefits, we believe that the most important aspect of D-Streams is that they show that streaming, batch and interactive computations can be unified in the same platform. As "big" data becomes the *only* size of data at which certain applications can operate (*e.g.,* spam detection on large websites), organizations will need the tools to write both lower-latency applications and more interactive ones that use this data, not just the periodic batch jobs used so far. D-Streams integrate these modes of computation at a deep level, in that they follow not only a similar API but also the same data structures and fault tolerance model as batch jobs. This enables rich features like combining streams with offline data or running ad-hoc queries on stream state.

Finally, while we presented a basic implementation of D-Streams, there are several areas for future work:

**Expressiveness:** In general, as the D-Stream abstraction is primarily an *execution* strategy, it should be possible to run most streaming algorithms within them, by simply "batching" the execution of the algorithm into steps and emitting state across them. It would be interesting to port languages like streaming SQL [4] or Complex Event Processing models [14] over them.

**Setting the batch interval:** Given any application, setting an appropriate batch interval is very important as it directly determines the trade-off between the end-to-end latency and the throughput of the streaming workload. Currently, a developer has to explore this trade-off and determine the batch interval manually. It may be possible for the system to tune it automatically.

**Memory usage:** Our model of stateful stream processing generates new a RDD to store each operator's state after each batch of data is processed. In our current implementation, this will incur a higher memory usage than continuous operators with mutable state. Storing different versions of the state RDDs is essential for the system perform lineage-based fault recovery. However, it may be possible to reduce the memory usage by storing only the deltas between these state RDDs.

**Approximate results:** In addition to recomputing lost work, another way to handle a failure is to return approximate partial results. D-Streams provide the opportunity to compute partial results by simply launching a task *before* its parents are all done, and offer lineage data to know *which* parents were missing.

## 8 Related Work

**Streaming Databases** Streaming databases such as Aurora, Telegraph, Borealis, and STREAM [7, 9, 5, 4] were the earliest academic systems to study streaming, and pioneered concepts such as windows and incremental operators. However, *distributed* streaming databases, such as Borealis, used replication or upstream backup for recovery [20]. We make two contributions over them.

First, D-Streams provide a more efficient recovery mechanism, parallel recovery, that runs faster than upstream backup without the cost of replication. Parallel recovery is feasible because D-Streams discretize computations into stateless, deterministic tasks. In contrast, streaming databases use a stateful continous operator model, and require complex protocols for both replication (*e.g.,* Borealis's DPC [5] or Flux [34]) and upstream backup [20]. The only parallel recovery protocol we are aware of, by Hwang et al [21], only tolerates one node failure, and cannot handle stragglers.

Second, D-Streams also tolerate stragglers, using speculative execution [12]. Straggler mitigation is difficult in continuous operator models because each node has mutable state that cannot be rebuilt on another node without a costly serial replay process.

**Large-scale Streaming** While several recent systems enable streaming computation with high-level APIs similar to D-Streams, they also lack the fault and straggler recovery benefits of the discretized stream model.

TimeStream [33] runs the continuous, stateful operators in Microsoft StreamInsight [2] on a cluster. It uses a recovery mechanism similar to upstream backup that tracks which upstream data each operator depends on and replays it serially through a new copy of the operator. Recovery thus happens on a single node for each op-

[9] Note that the raw rate of records/second for this algorithm is lower than in our other programs because it performs far more work for each record, drawing 300 Markov Chain Monte Carlo samples per record.

erator, and takes time proportional to that operator's processing window (*e.g.,* 30 seconds for a 30-second sliding window) [33]. In contrast, D-Streams use stateless transformations and explicitly put state in data structures (RDDs) that can (1) be checkpointed asynchronously to bound recovery time and (2) be rebuilt in parallel, exploiting parallelism across data partitions and timesteps to recover in sub-second time. D-Streams can also handle stragglers, while TimeStream does not.

Naiad [27, 28] automatically incrementalizes data flow computations written in LINQ and is unique in also being able to incrementalize *iterative* computations. However, it uses traditional synchronous checkpointing for fault tolerance, and cannot respond to stragglers.

MillWheel [1] runs stateful computations using an event-driven API but handles reliability by writing all state to a replicated storage system like BigTable.

MapReduce Online [11] is a streaming Hadoop runtime that pushes records between maps and reduces and uses upstream backup for reliability. However, it cannot recover reduce tasks with long-lived state (the user must manually checkpoint such state into an external system), and does not handle stragglers. Meteor Shower [41] also uses upstream backup, and can take tens of seconds to recover state. iMR [25] offers a MapReduce API for log processing, but can lose data on failure. Percolator [32] runs incremental computations using triggers, but does not offer high-level operators like *map* and *join*.

Finally, to our knowledge, almost none of these systems support *combining* streaming with batch and ad-hoc queries, as D-Streams do. Some streaming databases have supported combining tables and streams [15].

**Message Queueing Systems** Systems like Storm, S4, and Flume [37, 29, 3] offer a message passing model where users write stateful code to process records, but they generally have limited fault tolerance guarantees. For example, Storm ensures "at-least-once" delivery of *messages* using upstream backup at the source, but requires the user to manually handle the recovery of *state*, *e.g.,* by keeping all state in a replicated database [38]. Trident [26] provides a functional API similar to LINQ on top of Storm that manages state automatically. However, Trident does this by storing all state in a replicated database to provide fault tolerance, which is expensive.

**Incremental Processing** CBP [24] and Comet [18] provide "bulk incremental processing" on traditional MapReduce platforms by running MapReduce jobs on new data every few minutes. While these systems benefit from the scalability and fault/straggler tolerance of MapReduce *within* each timestep, they store all state in a replicated, on-disk filesystem *across* timesteps, incurring high overheads and latencies of tens of seconds to minutes. In contrast, D-Streams can keep state unreplicated in memory using RDDs and can recover it across timesteps using lineage, yielding order-of-magnitude lower latencies. Incoop [6] modifies Hadoop to support incremental recomputation of job outputs when an input file changes, and also includes a mechanism for straggler recovery, but it still uses replicated on-disk storage between timesteps, and does not offer an explicit streaming interface with concepts like windows.

**Parallel Recovery** One recent system that adds parallel recovery to streaming operators is SEEP [8], which allows continuous operators to expose and split up their state through a standard API. However, SEEP requires invasive rewriting of each operator against this API, and does not extend to stragglers.

Our parallel recovery mechanism is also similar to techniques in MapReduce, GFS, and RAMCloud [12, 16, 30], all of which partition recovery work on failure. Our contribution is to show how to structure a streaming computation to allow the use of this mechanism across data partitions and time, and to show that it can be implemented at a small enough timescale for streaming.

## 9 Conclusion

We have proposed D-Streams, a new model for distributed streaming computation that enables fast (often sub-second) recovery from both faults and stragglers without the overhead of replication. D-Streams forgo conventional streaming wisdom by *batching* data into small timesteps. This enables powerful recovery mechanisms that exploit parallelism across data partitions and time. We showed that D-Streams can support a wide range of operators and can attain high per-node throughput, linear scaling to 100 nodes, sub-second latency, and sub-second fault recovery. Finally, because D-Streams use the same execution model as batch platforms, they compose seamlessly with batch and interactive queries. We used this capability in Spark Streaming to let users combine these models in powerful ways, and showed how it can add rich features to two real applications.

Spark Streaming is open source, and is now included in Spark at http://spark-project.org.

## 10 Acknowledgements

We thank the SOSP reviewers and our shepherd for their detailed feedback. This research was supported in part by NSF CISE Expeditions award CCF-1139158 and DARPA XData Award FA8750-12-2-0331, a Google PhD Fellowship, and gifts from Amazon Web Services, Google, SAP, Cisco, Clearstory Data, Cloudera, Ericsson, Facebook, FitWave, General Electric, Hortonworks, Huawei, Intel, Microsoft, NetApp, Oracle, Samsung, Splunk, VMware, WANdisco and Yahoo!.

## References


[1] T. Akidau, A. Balikov, K. Bekiroglu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. MillWheel: Fault-tolerant stream processing at internet scale. In *VLDB*, 2013.

[2] M. H. Ali, C. Gerea, B. S. Raman, B. Sezgin, T. Tarnavski, T. Verona, P. Wang, P. Zabback, A. Ananthanarayan, A. Kirilov, M. Lu, A. Raizman, R. Krishnan, R. Schindlauer, T. Grabs, S. Bjeletich, B. Chandramouli, J. Goldstein, S. Bhat, Y. Li, V. Di Nicola, X. Wang, D. Maier, S. Grell, O. Nano, and I. Santos. Microsoft CEP server and online behavioral targeting. *Proc. VLDB Endow.*, 2(2):1558, Aug. 2009.

[3] Apache Flume. http://incubator.apache.org/flume/.

[4] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein, and J. Widom. STREAM: The Stanford stream data management system. *SIGMOD 2003*.

[5] M. Balazinska, H. Balakrishnan, S. R. Madden, and M. Stonebraker. Fault-tolerance in the Borealis distributed stream processing system. *ACM Trans. Database Syst.*, 2008.

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

[7] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams: a new class of data management applications. In *VLDB '02*, 2002.

[8] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *SIGMOD*, 2013.

[9] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[10] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. B. Zdonik. Scalable distributed stream processing. In *CIDR*, 2003.

[11] T. Condie, N. Conway, P. Alvaro, and J. M. Hellerstein. MapReduce online. *NSDI*, 2010.

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

[13] EsperTech. Performance-related information. http://esper.codehaus.org/esper/performance/performance.html, Retrieved March 2013.

[14] EsperTech. Tutorial. http://esper.codehaus.org/tutorials/tutorial/tutorial.html, Retrieved March 2013.

[15] M. Franklin, S. Krishnamurthy, N. Conway, A. Li, A. Russakovsky, and N. Thombre. Continuous analytics: Rethinking query processing in a network-effect world. *CIDR*, 2009.

[16] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *Proceedings of SOSP '03*, 2003.

[17] J. Hammerbacher. Who is using flume in production? http://www.quora.com/Flume/Who-is-using-Flume-in-production/answer/Jeff-Hammerbacher.

[18] 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*, 2010.

[19] 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.

[20] J.-H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik. High-availability algorithms for distributed stream processing. In *ICDE*, 2005.

[21] J. hyon Hwang, Y. Xing, and S. Zdonik. A cooperative, self-configuring high-availability solution for stream processing. In *ICDE*, 2007.

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

[23] S. Krishnamurthy, M. Franklin, J. Davis, D. Farina, P. Golovko, A. Li, and N. Thombre. Continuous analytics over discontinuous streams. In *SIGMOD*, 2010.

[24] D. Logothetis, C. Olston, B. Reed, K. C. Webb, and K. Yocum. Stateful bulk processing for incremental analytics. SoCC, 2010.

[25] D. Logothetis, C. Trezzo, K. C. Webb, and K. Yocum. In-situ MapReduce for log processing. In *USENIX ATC*, 2011.

[26] N. Marz. Trident: a high-level abstraction for realtime computation. http://engineering.twitter.com/2012/08/trident-high-level-abstraction-for.html.

[27] F. McSherry, D. G. Murray, R. Isaacs, and M. Isard. Differential dataflow. In *Conference on Innovative Data Systems Research (CIDR)*, 2013.

[28] D. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. In *SOSP '13*, 2013.

[29] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *Intl. Workshop on Knowledge Discovery Using Cloud and Distributed Computing Platforms (KDCloud)*,

2010.

[30] D. Ongaro, S. M. Rumble, R. Stutsman, J. K. Ousterhout, and M. Rosenblum. Fast crash recovery in RAMCloud. In *SOSP*, 2011.

[31] Oracle. Oracle complex event processing performance. http://www.oracle.com/technetwork/middleware/complex-event-processing/overview/cepperformancewhitepaper-128060.pdf, 2008.

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

[33] Z. Qian, Y. He, C. Su, Z. Wu, H. Zhu, T. Zhang, L. Zhou, Y. Yu, and Z. Zhang. Timestream: Reliable stream computation in the cloud. In *EuroSys '13*, 2013.

[34] M. Shah, J. Hellerstein, and E. Brewer. Highly available, fault-tolerant, parallel dataflows. *SIGMOD*, 2004.

[35] Z. Shao. Real-time analytics at Facebook. XLDB 2011, http://www-conf.slac.stanford.edu/xldb2011/talks/xldb2011_tue_0940_facebookrealtimeanalytics.pdf.

[36] U. Srivastava and J. Widom. Flexible time management in data stream systems. In *PODS*, 2004.

[37] Storm. https://github.com/nathanmarz/storm/wiki.

[38] Guaranteed message processing (Storm wiki). https://github.com/nathanmarz/storm/wiki/Guaranteeing-message-processing.

[39] 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.

[40] R. Tibbetts. Streambase performance & scalability characterization. http://www.streambase.com/wp-content/uploads/downloads/StreamBase_White_Paper_Performance_and_Scalability_Characterization.pdf, 2009.

[41] H. Wang, L.-S. Peh, E. Koukoumidis, S. Tao, and M. C. Chan. Meteor shower: A reliable stream processing system for commodity data centers. In *IPDPS '12*, 2012.

[42] 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.

[43] 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. In *NSDI*, 2012.

# GraphX: Unifying Data-Parallel and Graph-Parallel Analytics

Reynold S. Xin
Joseph E. Gonzalez
Daniel Crankshaw
Michael J. Franklin
Ankur Dave
Ion Stoica

UC Berkeley AMPLab
{rxin, crankshaw, ankurd, jegonzal, franklin, istoica}@cs.berkeley.edu

## ABSTRACT

From social networks to language modeling, the growing scale and importance of graph data has driven the development of numerous new graph-parallel systems (e.g., Pregel, GraphLab). By restricting the computation that can be expressed and introducing new techniques to partition and distribute the graph, these systems can efficiently execute iterative graph algorithms orders of magnitude faster than more general data-parallel systems. However, the same restrictions that enable the performance gains also make it difficult to express many of the important stages in a typical graph-analytics pipeline: constructing the graph, modifying its structure, or expressing computation that spans multiple graphs. As a consequence, existing graph analytics pipelines compose graph-parallel and data-parallel systems using external storage systems, leading to extensive data movement and complicated programming model.

To address these challenges we introduce GraphX, a distributed graph computation framework that unifies graph-parallel and data-parallel computation. GraphX provides a small, core set of graph-parallel operators expressive enough to implement the Pregel and PowerGraph abstractions, yet simple enough to be cast in relational algebra. GraphX uses a collection of query optimization techniques such as automatic join rewrites to efficiently implement these graph-parallel operators. We evaluate GraphX on real-world graphs and workloads and demonstrate that GraphX achieves comparable performance as specialized graph computation systems, while outperforming them in end-to-end graph pipelines. Moreover, GraphX achieves a balance between expressiveness, performance, and ease of use.

## 1. INTRODUCTION

From social networks to language modeling, graphs capture the structure in data and play a central role in the recent advances in machine learning and data mining. The growing scale and importance of graph data has driven the development of numerous specialized systems for graph analytics (*e.g.,* Pregel [14], PowerGraph [10], and others [7, 5, 21]). Each system presents a new *restricted* programming abstraction to compactly express iterative graph algorithms (*e.g.,* PageRank and connected components). By leveraging the restricted abstraction in conjunction with the static graph structure, these systems are able to optimize the data layout and distribute the execution of complex iterative algorithms on graphs with tens of billions of vertices and edges.

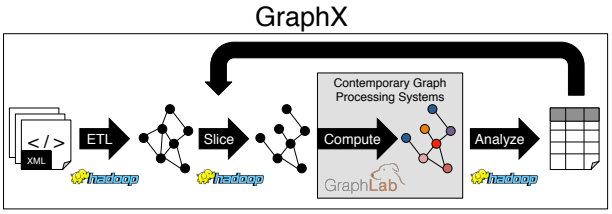



Figure 1: **Graph Analytics Pipeline:** Graph analytics is the process of going from raw data, to a graph, to the relevant subgraph, applying graph algorithms, analyzing the result, and then potentially repeating the process with a different subgraph. Currently, these pipelines compose data-parallel and graph-parallel systems through a distributed file interface. The goal of the GraphX system is to unify the data-parallel and graph-parallel views of computation into a single system and to accelerate the entire pipeline.

By restricting the types of computation they express to iterative vertex-centric algorithms on a single static graph, these *graph-parallel* systems are able to achieve orders-of-magnitude performance gains over contemporary data-parallel systems such as Hadoop MapReduce. However, these same restrictions make it difficult to express many of the operations found in a typical graph analytics pipeline (*e.g.,* Figure 1). These operations include constructing the graph from external sources, modifying the graph structure (*e.g.,* collapsing groups of vertices), and expressing computation that spans multiple graphs (*e.g.,* merging two graphs). For example, while the PowerGraph system can compactly express and execute algorithms like PageRank several orders of magnitude faster than contemporary data-parallel systems, it is not well suited for extracting graphs from a collection of databases, collapsing vertices within the same domain (*i.e.,* constructing a domain graph), or comparing the PageRank across several web graphs. Fundamentally, operations that move information outside of the graph topology or require a more global view are not well suited for graph-parallel systems.

In contrast, *data-parallel* systems like MapReduce [8] and Spark [23] are well suited for these tasks as they place minimal constraints on data movement and operate at a more global view. By exploiting data-parallelism, these systems are highly scalable; more recent systems like Spark even enable interactive data processing.

However, directly implementing iterative graph algorithms in these data-parallel abstractions can be challenging and typically leads to complex joins and excessive data movement due to the failure to exploit the graph structure or take advantage of any of the recent developments [5, 6, 10] in distributed graph partitioning and representation.

As a consequence, existing graph analytics pipelines (*e.g.,* GraphBuilder [11]) resort to composing graph-parallel graph analytics and data-parallel systems for graph loading through external storage systems such as HDFS. The resulting APIs are tailored to specific tasks and do not enable users to easily and efficiently compose graph-parallel and data-parallel operations on their data.

To address these challenges we introduce GraphX, a distributed graph computation framework which unifies graph-parallel and data-parallel computation in a single system. GraphX presents a unified abstraction which allows the same data to be viewed both as a graph and as tables without data movement or duplication. In addition to the standard data-parallel operators (*e.g.,* map, reduce, filter, join, etc.), GraphX introduces a small set of graph-parallel operators including subgraph and mrTriplets, which transform graphs through a highly parallel *edge*-centric API. We demonstrate that these operators are expressive enough to implement the Pregel and PowerGraph abstractions but also simple enough to be cast in relational algebra.

The GraphX system is inspired by the realization that (i) graphs can be encoded efficiently as tables of edges and vertices with some simple auxiliary indexing data structures, and (ii) graph computations can be cast as a sequence of relational operators including joins and aggregations on these tables. The contributions of this paper are:

1. a data model that unifies graphs and collections as composable first-class objects and enables both data-parallel and graph-parallel operations.
2. identifying a "narrow-waist" for graph computation, consisting of a small, core set of graph-operators cast in classic relational algebra; we believe these operators can express all graph computations in previous graph parallel systems, including Pregel and GraphLab.
3. an efficient distributed graph representation embedded in horizontally partitioned collections and indices, and a collection of execution strategies that achieve efficient graph computations by exploiting properties of graph computations.

# 2. GRAPH PROCESSING SYSTEMS

In contrast to general data processing systems (*e.g.,* MapReduce, Dryad, and Spark) which compose data-parallel operators to transform collections and are capable of expressing a wide range of computation, graph processing systems apply vertex-centric logic to transform data on a graph and exploit the graph structure to achieve more efficient distributed execution. In this section we introduce the key ideas behind graph-parallel systems and how they enable substantial performance gains. We then describe how the same restrictions that enable substantial performance gains limit the applicability of these systems to many important tasks in graph analytics.

## 2.1 Property Graphs

Graph data comes in many forms. The graph can be explicit (*e.g.,* social networks, web graphs, and financial transaction networks) or imposed through modeling assumptions (*e.g.,* collaborative filtering, language modeling, deep learning, and computer vision). We denote the structure of a graph $G = (V, E)$ by a set of vertices[1] $V = \{1, \ldots, n\}$ and a set of $m$ directed edges $E$. The directed edge $(i, j) \in E$ connects the source vertex $i \in V$ with the target vertex $j \in V$. The resulting graphs can have tens of billions of vertices and edges and are often highly sparse with complex, irregular, and often power-law structure.

In most cases attributes (properties) are associated with each vertex and edge. The properties can be both observed (*e.g.,* user profiles, time stamps, and weights) as well as model parameters and algorithm state (*e.g.,* PageRank, latent factors, and messages). We denote the vertex properties as $P_V(i)$ for vertex $i \in V$, the edge properties as $P_E(i, j)$ for edge $(i, j) \in E$, and the collection of all properties as $P = (P_V, P_E)$. Note that properties can consist of arbitrary data (*e.g.,* images, text, and objects).

The combination of graph structure and properties forms a **property graph** [19] $G(P) = (V, E, P)$ which is the basic representation of graph data and a core part of the GraphX data model. The property graph is a flexible model of graph data in that it imposes no constraints on the properties and allows the composition of different property collections with the same graph structure. For example, in parsing raw graph data we might begin with $G(P)$ and then transform the properties $f(P) \rightarrow P'$, yielding the new property graph $G(P')$ which retains the original structure. This separation of structure and properties is an important part of the GraphX system.

## 2.2 Graph-Parallel Computation

The recursive nature of graph data (*e.g.,* my interests are a function of my profile and the interests of my friends) necessitates the ability to calculate recursive properties on a graph. Algorithms ranging from PageRank and connected components to label propagation and collaborative filtering recursively define transformations on vertex and edge properties in terms of functions on the properties of adjacent vertices and edges. For example, the PageRank of each vertex may be computed by iteratively recomputing the PageRank of each vertex as a function of the PageRank of its neighboring vertices. The corresponding algorithms *iteratively* propagate information along the graph structure by transforming intermediate vertex and edge properties and solving for the fixed-point assignments. This common pattern of iterative local updates forms the basis of graph-parallel computation.

Graph-parallel computation is the analogue of data-parallel computation applied to graph data (*i.e.,* property graphs). Just as data-parallel computation adopts a record-centric view of collections, graph-parallel computation adopts a vertex-centric view of graphs. In contrast to **data-parallel** computation which derives parallelism by processing independent data on separate resources, **graph-parallel** computation derives parallelism by partitioning the graph (dependent) data across processing resources and then resolving dependencies (along edges) through iterative computation and communication. More precisely, graph-parallel computation recursively defines the transformations of properties in terms of functions on *neighboring* properties and achieves parallelism by executing those transformations in parallel.

## 2.3 Graph-Parallel Systems

The increasing scale and importance of graph-structured data has led to the emergence of a range of graph-parallel systems [13, 14, 12, 10, 5, 7, 21]. Each system is built around a variation of the graph-parallel abstraction [10], which consists of an property graph $G = (V, E, P)$ and a vertex-program $Q$ that is instantiated concurrently as $Q(v)$ for each vertex $v \in V$ and can interact with adjacent vertex-programs through messages (*e.g.,* Pregel [14]) or shared state

[1] In practice we do not constrain vertex identifiers to the consecutive integers $\{1, \ldots, n\}$.

```
def PageRank(v: Id, msgs: List[Double]) {
  // Compute the message sum
  var msgSum = 0
  for (m <- msgs) { msgSum = msgSum + m }
  // Update the PageRank (PR)
  A(v).PR = 0.15 + 0.85 * msgSum
  // Broadcast messages with new PR
  for (j <- OutNbrs(v)) {
    msg = A(v).PR / A(v).NumLinks
    send_msg(to=j, msg)
  }
  // Check for termination
  if (converged(A(v).PR)) voteToHalt(v)
}
```

Listing 1: **PageRank in Pregel**

(*e.g.,* GraphLab [12] and PowerGraph [10]). The instantiation of the vertex-program $Q(v)$ can read and modify the vertex property $P(v)$ as well as the properties on adjacent edges $P(v, j)$ for $\{v, j\} \in E$ and in some cases [12, 10] even the properties on adjacent vertices $P(j)$.

The extent to which vertex-programs run concurrently differs across systems. Most systems (*e.g.,* [14, 5, 10]) adopt the bulk synchronous execution model, in which all vertex-programs run concurrently in a sequence of super-steps operating on the adjacent vertex-program state or on messages from the previous super-step. Others (*e.g.,* [13, 12, 21, 10]) adopt an asynchronous execution model in which vertex-programs run as resources become available and impose constraints on whether neighboring vertex-programs can run concurrently. While [13] demonstrated significant gains from prioritized asynchronous scheduling, these gains are often offset by the additional complexity of highly asynchronous systems. The GraphX system adopts the bulk-synchronous model of computation because it ensures deterministic execution, simplifies debugging, and enables fault tolerance.

We will use the PageRank algorithm as a concrete running example to illustrate graph-parallel computation. In Listing 1 we express the PageRank algorithm as a simple Pregel vertex-program. The vertex-program for the vertex $v$ begins by receiving the messages (weighted PageRank of neighboring vertices) from the previous iteration and computing the sum. The PageRank is then recomputed using the message sum (with reset probability $0.15$). Then the vertex-program broadcasts its new PageRank value (weighted by the number of links on that page) to its neighbors. Finally, the vertex-program assesses whether it has converged (locally) and then votes to halt. If all vertex-programs vote to halt on the same iteration the program terminates. Notice that vertex-programs communicate with neighboring vertex-programs by passing messages along edges and that the vertex program iterates over its neighboring vertices.

More recently, Gonzalez et al. [10] observed that many vertex-programs factor along edges both when receiving messages and when computing messages to neighboring vertices. As a consequence they proposed the gather-apply-scatter (GAS) decomposition that breaks the vertex-program into purely edge-parallel and vertex-parallel stages, eliminating the ability to directly iterate over the neighborhood of a vertex. In Listing 2 we decompose the vertex-program in Listing 1 into Gather, Apply, and Scatter functions. The commutative associative gather function is responsible for accumulating the inbound messages, the apply function operates only on the vertex, and the scatter function computes the message for each edge and can be safely executed in parallel. The GAS decomposition enables vertices to be split across machines, increasing parallelism and addressing the challenge of the high-degree vertices common

```
def Gather(a: Double, b: Double) = a + b
def Apply(v, msgSum) {
  A(v).PR = 0.15 + 0.85 * msgSum
  if (converged(A(v).PR)) voteToHalt(v)
}
def Scatter(v, j) = A(v).PR / A(v).NumLinks
```

Listing 2: **PageRank in PowerGraph**

to many real-world graphs. The GraphX system adopts this more edge-centric perspective, enabling high-degree vertices to be split across machines.

The graph-parallel abstraction is sufficiently expressive to support a wide range of algorithms and at the same time sufficiently restrictive to enable the corresponding systems to efficiently execute these algorithms in parallel on large clusters. The static graph structure constrains data movement (communication) to the static topology of the graph, enabling the system to optimize the distributed execution. By leveraging advances in graph partitioning and representation, these systems are able to reduce communication and storage overhead. For example, [10] uses a range of vertex-based partitioning heuristics to efficiently split large power-law graphs across the cluster and vertex-replication and pre-aggregation to reduce communication. Given the result of the previous iteration, vertex-programs are *independent* and can be executed in any order, providing opportunities for better cache efficiency [20] and on-disk computation. As graph algorithms proceed, vertex-programs converge at different rates, leading to active sets (the collection of active vertex-programs) that shrink quickly. For example, when computing PageRank, vertices with no in-links will converge in the first iteration. Recent systems [14, 9, 12, 10] track active vertices and eliminate data movement and additional computation for vertices that have converged. Through GraphX we demonstrate that many of these same optimizations can be integrated into a data-parallel platform to support scalable graph computation.

## 2.4 Limitations of Graph-Parallel Systems

The same restrictions that enable graph-parallel systems to outperform contemporary data-parallel systems when applied to graph computation also limit their applicability to many of the operations found in a typical graph analytics pipeline (*e.g.,* Figure 1). For example, while graph-parallel systems can efficiently compute PageRank or label diffusion, they are not well suited to building graphs from multiple data sources, coarsening the graph (*e.g.,* building a domain graph), or comparing properties across multiple graphs. More precisely, the narrow view of computation provided by the graph-parallel abstraction is unable to express operations that build and transform the graph structure or span multiple independent graphs. Instead, these operations require data movement beyond the topology of the graph and a view of computation at the level of graphs rather than individual vertices and edges. For example, we might want to take an existing graph (*e.g.,* customer relationships) and *merge* external data (*e.g.,* sales information) prior to applying a graph-parallel diffusion algorithm (*e.g.,* for ad targeting). Furthermore, we might want to restrict our analysis to several subgraphs based on (*e.g.,* user demographics or time) and compare the results requiring both structural modifications as well as the ability to define computation spanning multiple graphs (*e.g.,* changes in predicted interests). In this example, the graph-parallel system is well suited for applying the computationally expensive diffusion algorithm but not the remaining operations which are fundamental to real-world analytics tasks.

To address the lack of support for these essential operations, exist-

ing graph-parallel systems either rely on additional graph ETL support tools (*e.g.,* GraphBuilder [11]) or have special internal functions for specific ETL tasks (*e.g.,* parsing a text file into a property graph). These solutions are limited in the range of operations they support and use external storage systems for sharing data across framework boundaries, incurring extensive data copying and movement. Finally, these systems do not address the challenge of computation that spans multiple graphs.

# 3. THE GraphX LOGICAL ABSTRACTION

The GraphX abstraction unifies the data-parallel and graph-parallel computation through a data model that presents graphs and collections as first class objects and a set of primitive operators that enables their composition. By unifying graphs and collections as first class composable objects, the GraphX data model is capable of spanning the entire graph analytics pipeline. By presenting a set of data-parallel and graph-parallel operators that can be composed in any order, GraphX allows users to leverage the programming model best suited for the current task without having to sacrifice performance or flexibility of future operations. We now describe the its data model and operators and demonstrate their composability and expressiveness through example applications.

## 3.1 The GraphX Data Model

The GraphX data model consists of *immutable* collections and property graphs. The *immutability* constraint simplifies the abstraction and enables data reuse and fault tolerance. Collections in GraphX consist of unordered tuples (*i.e.,* key-value pairs) and represent unstructured data. The key can be null and does not need to be unique, and the value can be an arbitrary object. The unordered collection view of data is essential for processing raw input, evaluating the results of graph computation, and certain graph transformations. For example, when loading data from a file we might begin with a collection of strings (with null keys) and then apply relational operators to obtain a collection of edge properties (keyed by edge), construct a graph and run PageRank, and finally view the resulting PageRank values (keyed by vertex identifier) as a collection for additional analytics.

The property graph $G(P) = (V, E, P)$ combines structural information, $V$ and $E$, with properties $P = (P_V, P_E)$ describing the vertices and edges. The vertex identifiers $i \in V$ can be arbitrary, but the GraphX system currently uses 64-bit integers (without consecutive ordering constraints). These identifiers may be derived externally (*e.g.,* user ids) or by applying a hash function to a vertex property (*e.g.,* page URL). Logically the property graph combines the vertex and edge property collections consisting of key-value pairs $(i, P_V(i))$ and $((i, j), P_E(i, j))$ respectively. We introduce the property graph as a first class object in the data model to enable graph specific optimizations which span the vertex and edge property collections and to present a more natural graph-oriented API.

## 3.2 The Operators

Computation in the GraphX abstraction is achieved by composing graph-parallel and data-parallel operators that take graphs and collections as input and produce new graphs and collections. In selecting the core set of operators we try to balance the desire for parsimony with the ability to exploit specific lower-level optimizations. As a consequence these operators form a narrow interface to the underlying system, enabling the GraphX abstraction to be expressive and yet feasible to implement and execute efficiently on a wide range of *data-parallel* systems. To simplify graph analytics, GraphX exposes a rich API of more complex graph operators (*e.g.,* coarsening, neighborhood aggregation) and even other abstractions (*e.g.,* Pregel) by composing the basic set of primitive operators.

```
class Col[K,V] {
  def filter(pred: (K,V) => Boolean): Col[K,V]
  def map(f: (K,V) => (K2,V2)): Col[K2,V2]
  def reduceByKey(reduce: (V, V) => V): Col[K,V]
  def leftJoin(a: Col[K, U]): Col[K, (T, U)]
  ...
}
```

Listing 3: **Collection operators**. The map function takes a collection of key-value paris of type *(K,V)* and a UDF which maps to a new key-value pair of type *(K2,V2)*. Collections are special case of relational tables, and each collection operator has its relational counterpart (map vs project, reduceByKey vs aggregates, etc).

```
class Graph[V,E] {
  def Graph(v: Col[(Id,V)], e: Col[(Id,Id,E)],
            mergeV: (V, V) => V,
            defaultV: V): Graph[V,E]

  def vertices: Col[Id, V]
  def edges: Col[(Id, Id), E]
  def triplets: Col[(Id, Id), (V, E, V)]

  def mapV(m: (Id, V) => V2): Graph[V2,E]
  def mapE(m: Triplet[V,E] => E2): Graph[V,E2]

  def leftJoin(t: Col[Id, U]): Graph[(V,U), E]

  def subgraph(vPred: (Id, V) => Boolean,
               ePred: Triplet[V,E] => Boolean):
      Graph[V, E]

  def mrTriplets(m: Trplt[V,E] => (M, M),
                 r: (M, M) => M,
                 skipStale: Direction = None):
      Col[Id, M]
}
```

Listing 4: **Graph operators:** The *mapE* operator takes a Graph over vertex and edge properties of type *V* and *E* and a *map* UDF from triplets to a new edge property and returns the graph with the new edge properties.

The GraphX system exposes the standard data-parallel operators (Listing 3) found in contemporary data-flow systems. The unary operators *filter*, *map*, and *reduceByKey* each takes a single collection and produces a new collection with the records removed, transformed, or aggregated. The binary operator *leftJoin* performs a standard left outer equi-join by key. Both the *map* and *filter* operators are entirely data-parallel without requiring any data movement or communication. On the other hand, the *reduceByKey* and *leftJoin* operators may require substantial data movement depending on how the data is partitioned.

In Listing 4 we describe the set of graph-parallel operators that produce new graphs and collections. These operators join vertex and edge collections, apply transformations on the properties and structure, and move data along edges in the graph. In all cases, these operators express local transformations on the graph (*i.e.,* UDFs have access to at most a single triplet at a time).

The *Graph* operator constructs a property graph from vertex and edge collections. In many applications the vertex collection may contain duplicate vertex properties or may not contain properties for vertices in the edge collection. For example when working with web data, web-links may point to missing pages or pages may have been crawled multiple times. By applying the *merge* UDF to duplicate

vertices and substituting the default property for missing vertices, the *Graph* operator ensures that the resulting graph is *consistent*: without missing or duplicate vertices.

While the *Graph* operator produces a graph-oriented view of collections, the *vertices*, *edges*, and *triplets* produce collection-oriented views of a graph. The *vertices* and *edges* operators deconstruct the property graph into the corresponding vertex and edge collections. The collection views are used when computing aggregates, analyzing the results of graph computation, or when saving graphs to external data stores. The *triplets* operator is logically a three-way join to form a new collection consisting of key-value pairs of the form $((i, j), (P_v(i), P_E(i, j), P_V(j)))$. This essential graph operator can be concisely cast in terms of relational operators:

```
SELECT s.Id, t.Id, s.P, e.P, t.P
FROM edges AS e
JOIN vertices AS s, vertices AS t
ON e.srcId = s.Id AND e.dstId = d.Id
```

By joining properties along edges, the *triplets* operator enables a wide range of graph computation. For example, the composition of the *triplets* and data-parallel *filter* operators can be used to extract edges that span two domains or connect users with different interests. Furthermore, the *triplets* operator is used to construct the other graph-parallel operators (*e.g., subgraph* and *mrTriplets*).

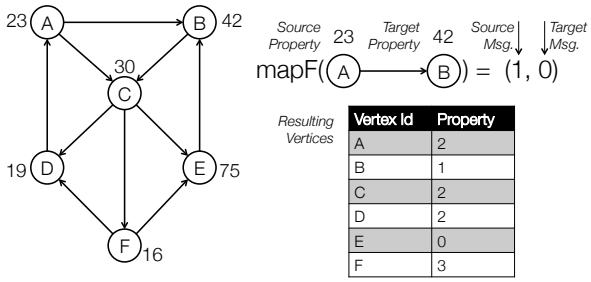
The *mapV* and *mapE* operators transform the vertex and edge properties respectively and return the transformed graph. The *map* UDF provided to *mapV* and *mapE* can only return a new attribute value and cannot modify the structure (*i.e.,* change the vertex identifiers for the vertex or edge). As a consequence, the resulting graph is guaranteed to be consistent and can reuse the underlying structural representation.

In many cases it is necessary to merge external vertex properties (*e.g.,* merging user profile data with a social network) stored in a vertex property collection with an existing graph. This can be accomplished in GraphX using the *leftJoin* graph operator. *leftJoin* takes a collection of vertex properties and returns a new graph that incorporates the properties into all matching vertices in the graph. The *leftJoin* preserves the original graph structure and is logically equivalent to a left outer equi-join of the vertices with the input vertex property collection.

Comparing the results of graph computation (*e.g.,* PageRank) on different slices (*i.e.,* subgraphs) of a graph based on vertex and edge properties (*e.g.,* time) often reveals trends in the data. To support this type of analysis we need to be able to efficiently construct subgraphs and compare properties and structural changes across subgraphs. The *subgraph* operator restricts the graph to the vertices and edges that satisfy the respective predicates. To ensure that the graph is consistent, all retained edges must satisfy both the source and target vertex predicate as well as the edge predicate.

The *mrTriplets* (*i.e.,* Map Reduce Triplets) operator is logically the composition of the *triplets* graph-parallel operator with the *map* and *reduceByKey* data-parallel operators. More precisely, the *mrTriplets* operator applies the map UDF to each triplet in the output of the *triplets* operator. The map UDF optionally constructs *"messages"* (of arbitrary type) to send to the source and target vertices (or both). All messages destined for the same vertex are aggregated using the commutative associative reduce UDF and the resulting aggregates are returned as a collection keyed by the destination vertex. This can be expressed in the following SQL query:

```
SELECT t.dstId, r(m(t)) AS sum
FROM triplets AS t GROUPBY t.dstId
WHERE sum IS NOT null
```




```
val graph: Graph[User, Double]
def mapF(t: Triplet[User, Double])
  : Iterator[Vid, Int] = {
  if (t.src.age > t.dst.age) (t.dstId, 1)
  else (t.src.age < t.dst.age) (t.srcId, 1)
  else Iterator.empty
}
def reduceUDF(a: Int, b: Int): Int = a + b
val seniors = graph.mrTriplets(mapUDF, reduceUDF)
```

Figure 2: **Example use of mrTriplets**: The *mrTriplets* operator is used to compute the number of more senior neighbors of each vertex. Note that vertex $E$ does not have a more senior neighbor and therefore does not appear in the collection.

The constraint that the map UDF only emits messages to the source or target vertex ensures that data movement remains along edges in the graph, preserving the graph dependency semantics. By expressing message computation as an *edge-parallel* map operation followed by a commutative associative aggregation, we eliminate the effect of high degree vertices on communication and parallel scalability. The *mrTriplets* operator is the primary building block of graph-parallel algorithms. For example, in Figure 2 we use the *mrTriplets* operator to compute the number of more senior neighbors for each user in a social network. In the next section we show how to compose these basic operators to express more complex tasks like graph coarsening and even implement existing graph-parallel abstractions.

When solving recursive properties on a graph, vertices typically only communicate when their values change. As a consequence, executing the *mrTriplets* function on all edges can be wasteful especially only a few vertices have changed. While it is possible to implement such logic within message calculation, the system must still invoke the message calculation on all edges. Therefore, *mrTriplets* has an optional argument *skipStale* which by default is disabled. However, if the *skipStale* flag is set to Out, for example, then edges originating from vertices that haven't changed since *mrTriplets* was last invoked are automatically skipped. In Section 4 we will see how this flag in conjunction with internal change tracking can efficiently skip a large fraction of the edges.

## 3.3 Composing Operators

The GraphX operators can express efficient versions of some of the most widely adopted graph-parallel abstractions. We have currently implemented enhanced versions of Pregel and the PowerGraph abstractions. In Listing 5 we construct an enhanced version of Pregel built around the more efficient GAS decomposition. The Pregel abstraction iteratively computes the messages on the active subgraph using the *mrTriplets* operator and then applies the *mapV* operator to execute the vertex program UDF. In this example we use change tracking option in *mrTriplets* to restrict execution to out-edges of vertices that changed in the previous round. In Section 4 we show that allowing *mrTriplets* to track changes enables several

```
def pregel(g: Graph[V,E],
           vprog: (V, M) => V,
           sendMsg: Triplet[V, E] => M,
           gather: (M, M) => M):
          Graph[V, E] = {
  def send(t: Triplet[V, E]) = {
    Iterator(t.dstId, sendMsg(t))
  }
  var live = g.vertices.count
  // Loop until convergence
  while (live > 0) {
    // Compute the messages
    val msgs = g.mrTriplets(send, gather, Out)
    // Receive the messages and run vertex program
    g = g.leftJoin(msgs).mapV(vprog)
    // Count the vertices that don't want to halt
    live = g.vertices.filter(v=>!v.halt).count
  }
  return g
}
```

Listing 5: **Enhanced Pregel:** We implemented a version of Pregel built around the GAS decomposition to enable degree independence and at the same allow message computation to read the remote vertex attributes.

```
def ConnectedComp(g: Graph[V,E]): Graph[Id, E] = {
  // Initialize the vertex properties
  g = g.mapV(v => v.id)
  def vProg(v: Id, m: Id): Id = {
    if (v == m) voteToHalt(v)
    return min(v, m)
  }
  def sendMsg(e: Triplet): Id =
    if(e.src.cc > e.dst.cc) (e.dst.cc, None)
    else if(e.src.cc < e.dst.cc) (None, e.src.cc)
    else (None, None)
  def gatherMsg(a: Id, b: Id): Id = min(a, b)
  return Pregel(g, vProg, sendMsg, gatherMsg)
}
```

Listing 6: **Connected Components:** We implement the connected components algorithm using the enhance version of Pregel.

important system optimizations. Unlike the original formulation of Pregel, our version exposes both the source and target vertex properties during message calculation. In Section 4.5.2 we demonstrate how through UDF bytecode inspection in the *mrTriplets* operator we can eliminate extra data movement if only one of the source or target attribute is accessed when computing the message (*e.g.,* PageRank).

In Listing 6 we used our version of Pregel to implement connected components. The connected components algorithm computes for each vertex its lowest reachable vertex id. We first initialize the vertex properties using the *vMap* operator and then define the three functions required to use the GAS version of Pregel. The *sendMsg* function leverages the triplet view of the edge to only send a message to neighboring vertices when their component id is larger.

Often groups of connected vertices are better modeled as a single vertex. In these cases it is desirable to coarsen the graph by aggregating *connected* vertices that share a common characteristic (*e.g.,* web domain) to derive a new graph (*e.g.,* the domain graph). We use the GraphX abstraction to implement a coarsening in Listing 7. The coarsening operation takes an edge predicate and a vertex aggregation function and collapses all edges that satisfy the predicate, merging their respective vertices. The edge predicate is used to first construct the subgraph of edges that are to be collapsed. Then the graph-parallel connected components algorithm is run on the subgraph. Each connected component corresponds to a super-vertex

```
def coarsen(g: Graph[V, E],
            pred: Triplet[V, E] => Boolean,
            reduce: (V,V) => V): Graph[V,E] = {
  // Restrict graph to contractable edges
  val subG = g.subgraph(v => True, pred)
  // Compute connected component id for all V
  val cc: Col[Id, Id] = ConnectedComp(subG).vertices
  // Merge all vertices in same component
  val superVerts = g.vertices.leftJoin(cc).map {
      (vId, (vProp, cc)) => (cc, vProp))
    }.reduceByKey(reduce)
  // Link remaining edges between components
  val invG = g.subgraph(v=>True, !pred)
  val remainingEdges =
    invG.leftJoin(cc).triplets.map {
      e => ((e.src.cc, e.dst.cc), e.attr)
    }
  // Return the final graph
  Graph(superVerts, remainingEdges)
}
```

Listing 7: **Coarsen:** The *coarsening* operator merges vertices connected by edges that satisfy an edge predicate UDF.

in the new coarsened graph with the component id being the lowest vertex id in the component. The super-vertices are constructed by aggregating all the vertices with the same component id. Finally, we update the edges to link together super-vertices and generate the new graph. The *coarsen* operator demonstrates the power of a unified abstraction by combining both data-parallel and graph-parallel operators in a single graph-analytics task.

# 4. THE GraphX SYSTEM

The scalability and performance of GraphX is derived from the design decisions and optimizations made in the physical execution layer. The design of the physical representation and execution model is heavily influenced by three characteristics of graph computation. First, in Section 3 we demonstrated that graph computation can be modeled as a series of joins and aggregations. Maintaining the proper indexes can substantially speed up local join and aggregation performance. Second, as outlined in [10], we can minimize communication in real-world graphs by using vertex-cut partitioning, in which edges are partitioned evenly across a cluster and vertices are *replicated* to machines with adjacent edges. Finally, graph computations are typically iterative and therefore we can afford to construct indexes. Furthermore, as computation proceeds, the *active* set of vertices – those changing between iterations – often decreases.

In the remainder of this section, we introduce Apache Spark, the open source data-parallel engine on which GraphX was built. We then describe the physical representation of data and the execution strategies adopted by GraphX. Along the way, we quantify the effectiveness of each optimization technique. Readers are referred to Section 5 for details on datasets and experimental setup.

## 4.1 Apache Spark

GraphX is implemented on top of Spark [23], a widely used data-parallel engine. Similar to Hadoop MapReduce, a Spark cluster consists of a single driver node and multiple worker nodes. The driver node is responsible for task scheduling and dispatching, while the worker nodes are responsible for the actual computation and physical data storage. However, Spark also has several features that differentiate it from traditional MapReduce engines and are important to the design of GraphX.

**In-Memory Caching:** Spark provides the *Resilient Distributed Dataset* (RDD) in-memory storage abstraction. RDDs are collec-

tions of objects that are partitioned across a cluster. GraphX uses RDDs as the foundation for distributed collections and graphs.

**Computation DAGs:** In contrast to the two-stage MapReduce topology, Spark supports general computation DAGs by composing multiple data-parallel operators on RDDs, making it more suitable for expressing complex data flows. GraphX uses and extends Spark operators to achieve the unified programming abstraction.

**Lineage-Based Fault Tolerance:** RDDs and the data-parallel computations on RDDs are fault-tolerant. Spark can automatically reconstruct any data or execute tasks lost during failures.

**Programmable Partitioning:** RDDs can be co-partitioned and co-located. When joining two RDDs that are co-partitioned and co-located, GraphX can exploit this property to avoid any network communication.

**Interactive Shell:** Spark allows users to interactively execute Spark commands in a Scala or Python shell. We have extended the Spark shell to support interactive graph analytics.

## 4.2 Distributed Graph Representation

GraphX represents graphs internally using two Spark distributed collections (RDDs) – an edge collection and a vertex collection. By default, the *edges* are partitioned according to their configuration in the input collection (*e.g.,* original placement on HDFS). However, they can be repartitioned by their source and target vertex ids using a user-defined partition function. GraphX provides a range of built-in partitioning functions, including a 2D hash partitioner that provides an upper bound on communication for the *mrTriplets* operator that is $O\left(n\sqrt{p}\right)$ for $p$ partitions and $n$ vertices. For efficient lookup of edges by their source and target vertices, the edges within a partition are clustered by source vertex id, and there is an unclustered index on target vertex id. The clustered index on source vertex id is a *compressed sparse row* (CSR) representation that maps a vertex id to the block of its out-edges. Section 4.6 discusses how these indexes are used to accelerate iterative computation.

The *vertices* are hash partitioned by their vertex ids, and on each partition, they are stored in a hash index (*i.e.,* clustered by the hash index). Each vertex partition also contains a *bitmask* and *routing table*. The bitmask enables the set intersection and filtering required by the *subgraph* and *join* operators. Vertices hidden by the bitmask do not participate in the graph operations. The *routing table* contains the join sites for each vertex in the partition and is used when broadcasting vertices to construct triplets. The routing table is logically a map from a vertex id to the set of edge partitions that contain adjacent edges and is derived from the edge table by collecting the source and target vertex ids for each edge partitions and aggregating the result by vertex id. The routing table is stored as a compressed bitmap (*i.e.,* for each edge partition, which vertices are present).

## 4.3 Structural Index Reuse

Because the collections and graphs are *immutable* they can share the structural indexes associated within each vertex and edge partition to both reduce memory overhead and accelerate *local* graph operations. For example, within a vertex partition, we can use the hash index to perform fast aggregations and the resulting aggregates would *share* the same index as the vertices. This shared index enables very efficient joining of the original vertices and the aggregates by converting the join into coordinated sequential scans (similar to a merge join). In our benchmarks, index reuse brings down the runtime of PageRank on the Twitter graph from 27 seconds per iteration to 16 seconds per iteration. Index reuse has the added benefit of reducing memory allocation, because the indexes are reused in memory from one collection and graph to the next, and only the properties are changed.

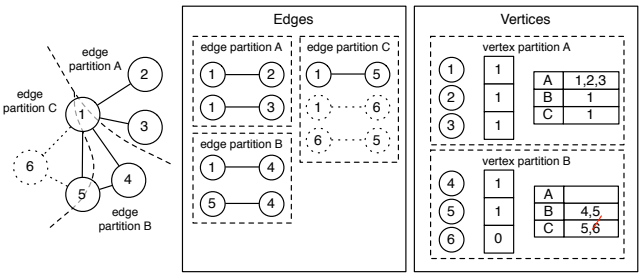



Figure 3: **Distributed representation of a graph**: The graph on the left is represented using distributed collections. It is partitioned into three edge partitions. The vertices are partitioned by id. Within each vertex partition, the routing table stores for each edge partition the set of vertices present. Vertex 6 and adjacent edges (shown with dotted lines) have been restricted from the graph, so they are removed from the edges and the routing table. Vertex 6 remains in the vertex partitions, but it is hidden by the bitmask.

Most of the GraphX operators preserve the structural indexes to maximize index reuse. Operators that do not modify the graph structure (*e.g., mapV*, *mapE*, *leftJoin*, and *mrTriplets*) directly preserve the indexes. To reuse indexes for operations that *restrict* the graph structure (*e.g., subgraph* and *innerJoin*), GraphX relies on the bitmask to construct the restricted graph view. Some of the collections operations (*e.g., g.vertices.map*) enable more general transformations (*e.g.,* renumbering vertices) that destroy the index but have more restrictive analogues that preserve the index (*e.g., g.mapV*). Finally, in some cases extensive index reuse could lead to decreased efficiency, such as for graphs that are highly filtered. GraphX therefore provides a *reindex* operator for graphs which rebuilds the index over the visible vertices.

## 4.4 Graph Operator Execution Strategies

The GraphX abstraction consists of both data-parallel and graph-parallel operators. For the data-parallel operators we adopt the standard well-established execution strategies, using indexes when available. Therefore, in this section we focus on execution strategies for the graph-parallel operators outlined in Section 3.

The graph-parallel operators defined in Listing 4 are implemented by transforming the vertex and edge RDDs using the Spark API. The execution strategies for each operator are as follows:

*vertices, edges:* Directly extract the vertex and edge RDDs.

*mapV, mapE:* Transform the internal vertex and edge collections, preserving the indexes.

*leftJoin:* Co-partition the input with the vertex attributes, join the vertex attributes with the co-partitioned input using the internal indexes, and produce a new set of vertex attributes. As a consequence only the input is shuffled across the network.

*triplets:* Logically requires a multiway distributed join between the vertex and edge RDDs. However using the routing map, we move the execution site of the multiway join to the edges, allowing the system to shuffle only the vertex data and avoid moving the edges, which are often orders of magnitude larger than the vertices. The triplets are assembled at the edges by placing the vertices in a local hash map and then scanning the edge table.

*subgraph:* (1) Generate the graph's triplets, (2) filter the triplets using the conjunction of the edge triplet predicate and the vertex predicate on both source and target vertices to produce a restricted edge set, and (3) filter the vertices using the vertex predicate. To

avoid allocation and provide fast joins between the subgraph and the original graph, the vertex filter is performed using the bitmask in the internal vertex collection, as described in Section 4.3.

*innerJoin:* (1) Perform an inner join between the input and the internal vertex collection to produce the new vertex properties, and (2) ensure consistency by joining the ids in the input collection with the internal edge collection and dropping invalidated edges.

The distributed join in step 2 is only performed separately when the user requests the edges of the result. It is redundant for operations on the triplet view of the graph, such as *triplets* and *mrTriplets*, because the joins in these operations implicitly filter out edges with no corresponding vertex attributes.

Vertices eliminated by the inner join in step 1 can be removed using the bitmask in a similar fashion as for *subgraph*, enabling fast joins between the resulting vertex set and the original graph. We exploit this in our Enhanced Pregel implementation, as described in Section 4.5.1.

*mrTriplets:* Apply the map UDF to each triplet and aggregate the resulting messages by target vertex id using the reduce UDF. Implementing the *skipStale* argument requires the Incremental View Maintenance optimization in section 4.5.1, so its implementation is described there.

## 4.5 Distributed Join Optimizations

The logical query plan for the *mrTriplets* operator consists of a three-way join to bring the source vertex attributes and the target vertex attributes to the edges and generate the triplets view of the graph, followed by an aggregation step to apply the map and reduce UDFs. We use the routing table to ship vertices to edges and set the edge partition as the join sites, which is equivalent to the idea of vertex-cut partitioning in PowerGraph. In addition, we describe two techniques we have developed that further minimize the communication in the join step. The first applies the concept of incremental view maintenance to communicate only vertices that change values after a graph operation, and the second uses bytecode analysis to automatically rewrite the physical join plan. These techniques enable GraphX to present a simple logical view of triplets with the capability to optimize the communication patterns in the physical execution plan.

### *4.5.1 Incremental View Maintenance*

We observe that the number of vertices that change in iterative graph computations usually decreases as the computation converges to a fixed-point. This presents an opportunity to further optimize the join in *mrTriplets* using techniques in incremental view maintenance. Recall that in order to compute the join, GraphX uses the routing table to route vertices to the appropriate join sites in the internal edge RDD. After each graph operation, we update a bit mask to track which vertex properties have changed. When GraphX needs to ship the vertices to materialize (in-memory) the replicated vertex view, it creates the view by shipping only vertices that have changed, and uses values from the previously materialized replicated vertex view for vertices that have not changed.

Internally, GraphX maintains a bitmask alongside the replicated vertex view to record which vertices have changed. The *mrTriplets* operator uses this bitmask to support *skipStale*, which determines for each edge whether to skip running the map UDF based on whether the source and/or target vertex of the edge has changed.

Figure 4 illustrates the impact of incrementally maintaining the replicated vertex view in both PageRank and connected components on the Twitter graph.

### *4.5.2 Automatic Join Elimination*

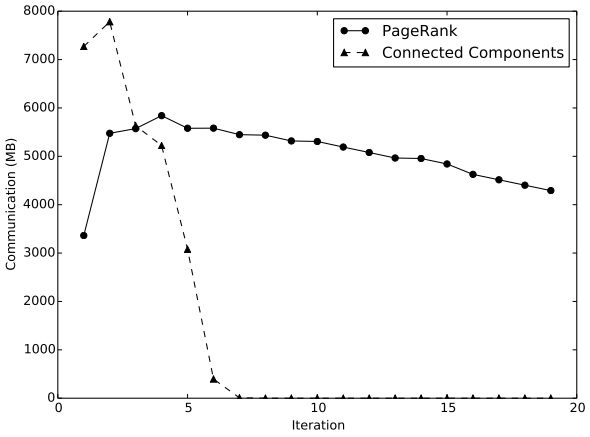



Figure 4: **Impact of incrementally maintaining the replicated vertex view**: For both PageRank and connected components, as vertices converge, communication decreases due to incremental view maintenance. We suspect the initial steep rise in communication is due to compression; many early rank update messages are the same and can be run-length encoded.

The map UDF in the *mrTriplets* operator may only access one of the vertices, or none at all, in many algorithms. For example, when *mrTriplets* is used to count the degree of each vertex, the map UDF does not access any vertex attributes.[2] In the case of PageRank, only the source vertex attributes are referenced.

GraphX implements a JVM bytecode analyzer that inspects the bytecode of the map UDF at runtime for a given *mrTriplets* query plan and determines whether the source or target vertex attributes are referenced. If only the source attributes are referenced, GraphX automatically rewrites the query plan from a three-way join to a two-way join. If none of the vertex attributes are referenced, GraphX eliminates the join entirely. Figure 5 demonstrates the impact of this physical execution plan rewrite on communication and runtime.

## 4.6 Sequential Scan vs Index Scan

Recall that in most operators, GraphX uses the structural indexes and relies on bitmasks to track whether a particular vertex is still visible. While this reduces the cost of computing index structures in iterative computations, it also prevents the physical data set from shrinking in size. For example, in the last iteration of connected components on the Twitter graph, only a few of the vertices are still active. However, to execute the *mrTriplets* on the triplet view we still need to sequentially scan 1.5 billion edges and verify for each edge whether its vertices are still visible using the bitmask.

To mitigate this problem, we implement an index scan access method on the bitmask and switch from sequential scan on edges to bitmap index scan on vertices when the fraction of active vertices is less than 0.8. This bitmap index scan on vertices exploits the property that edges are clustered by their source vertex id to efficiently join vertices and edges together. Figure 6 illustrates the performance of sequential scan versus index scan in both PageRank and connected components.

When *skipStale* is passed to the *mrTriplets* operator, the index scan can be further optimized by checking the bitmask for each vertex id and filtering the index as specified by *skipStale* rather than performing the filter on the output of the index scan.

## 4.7 Additional Engineering Techniques

[2]The map UDF does access vertex IDs, but they are part of the edge structure and do not require shipping the vertices.

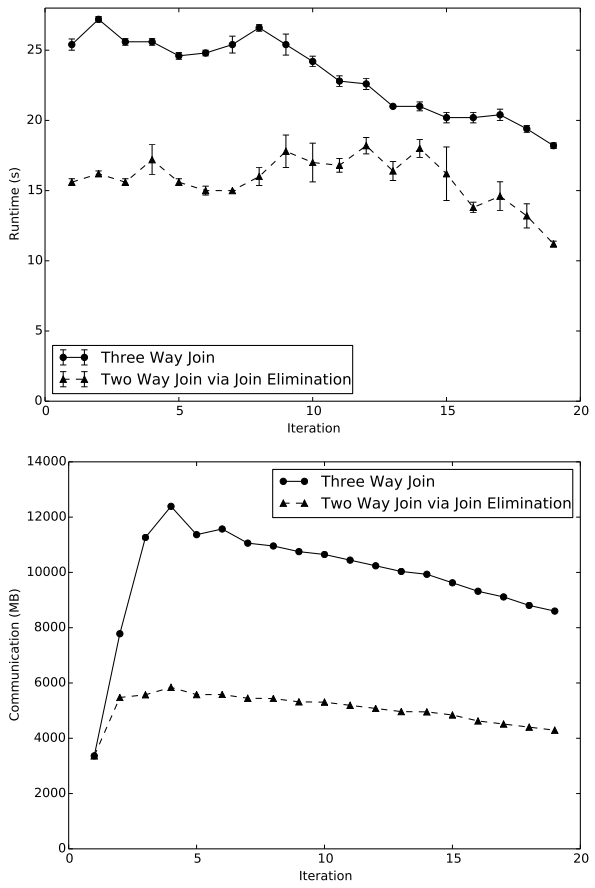



Figure 5: **Impact of automatic join elimination on communication and runtime**: We ran PageRank for 20 iterations on the Twitter dataset with join elimination turned on and off. We observe that automatic join elimination reduces the amount of communication by almost half and substantially decreases the total execution time as well.

While implementing GraphX, we discovered that a number of low level engineering details had significant performance impact. We sketch some of them here.

**Memory-based Shuffle:** GraphX relies on Spark's shuffle mechanism for join and aggregation communication. Spark's default implementation materializes the shuffle data to disk, hoping that it will remain in the OS buffer cache when the data is fetched by remote nodes. In practice, we have found that the extra system calls and file system journaling adds significant overhead, and the inability to control when buffer caches are flushed leads to variability in communication-intensive workloads like graph algorithms. We modified the shuffle phase to materialize map outputs in memory and remove this temporary data using a timer.

**Batching and Columnar Structure:** In our join code path, rather than shuffling the vertices one by one, we batch a block of vertices routed to the same target join site and convert the block from row-oriented format to column-oriented format. We then apply the LZF compression algorithm on these blocks to send them. Batching has a negligible impact on CPU time while improving the compression ratio of LZF by 10–40% in our benchmarks.

**Variable Integer Encoding:** Though we use 64-bit integer identifiers for vertices and edges, in most cases the ids are much smaller

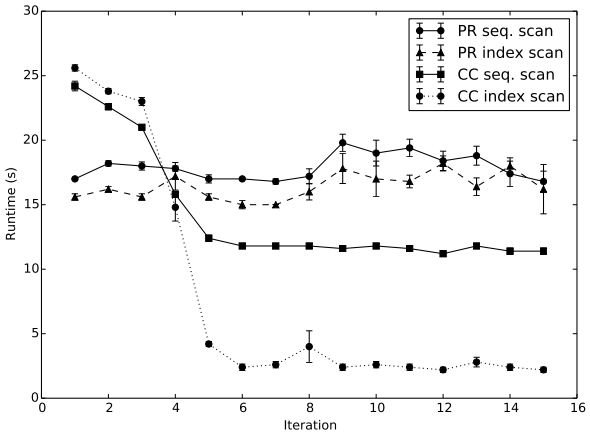



Figure 6: **Sequential scan vs index scan**: Connected components (CC) on Twitter graph benefits greatly from switching to index scan after the 4th iteration, while PageRank (PR) benefits only slightly because the set of active vertices is large even at the 15th iteration.

than $2^{64}$. To exploit this fact, during shuffling, we encode integers using a variable-encoding scheme where for each byte, we use only the first 7 bits to encode the value, and use the highest order bit to indicate whether we need another byte to encode the value. In this case, smaller integers are encoded with fewer bytes. In the worst case, integers greater than $2^{56}$ require 5 bytes to encode. This technique reduces our communication in PageRank by 20%.

# 5. SYSTEM EVALUATION

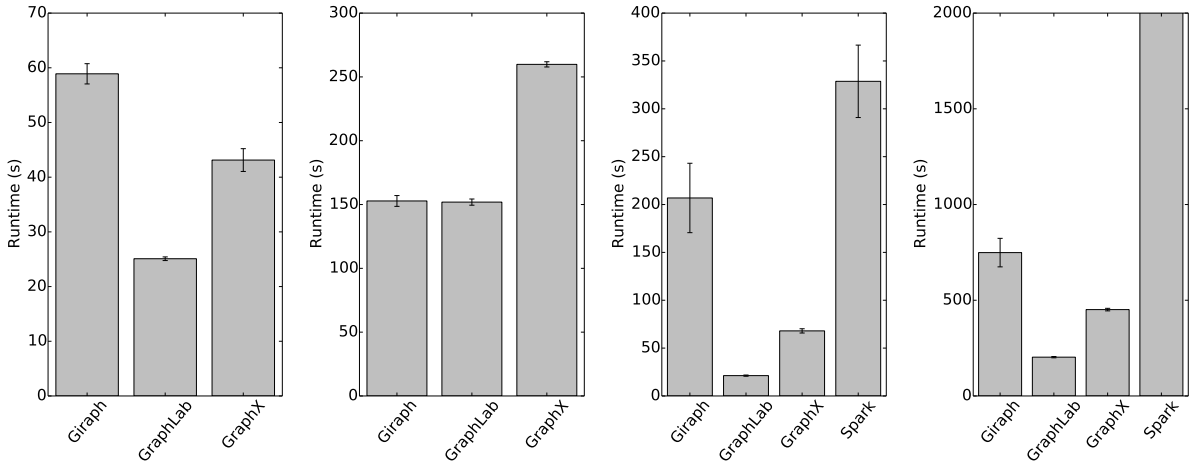
We evaluate the performance of GraphX on specific graph-parallel computation tasks as well as end-to-end graph analytic pipelines, comparing to the following systems:

1. Apache Spark 0.8.1: the data-parallel cluster compute engine GraphX builds on. We use Spark to demonstrate the performance of graph algorithms implemented naively on data-parallel systems. We chose Spark over Hadoop MapReduce because of Spark's support for distributed joins and its reported superior performance [23, 22].
2. Apache Giraph 1.0: an open source implementation of Google's Pregel. It is a popular graph computation engine in the Hadoop ecosystem initially open-sourced by Yahoo!.
3. GraphLab 2.2 (PowerGraph): another open source graph computation engine commonly believed to be one of the fastest available. Note that GraphLab is implemented in C++, while both other systems and GraphX run on the JVM. It is expected that even if all four systems implement identical optimizations, GraphLab would have an "unfair" advantage due to its native runtime.

For graph-parallel algorithms, we demonstrate that GraphX is more than an order of magnitude faster than idiomatic Spark and performs comparably to the specialized systems, while outperforming them in end-to-end pipelines.

All experiments were conducted on Amazon EC2 using 16 m2.4xlarge worker nodes in November and December 2013. Each node had 8 virtual cores, 68 GB of memory, and two hard disks. The cluster was running 64-bit Linux 3.2.28. We plot the mean and standard deviation for 10 trials of each experiment.

## 5.1 Graph-Parallel Performance




(a) Conn. Comp. LiveJournal (b) Conn. Comp. Twitter (c) PageRank LiveJournal (d) PageRank Twitter[a]

[a] Spark PageRank on Twitter encountered memory constraints and took over 5000 s, so we have truncated its bar to ease comparison between the graph systems.

Figure 7: **Graph-parallel performance comparison**

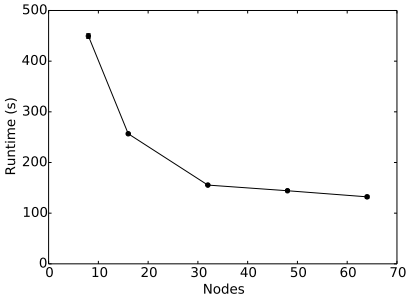



Figure 8: **GraphX Strong Scaling for PageRank on Twitter**

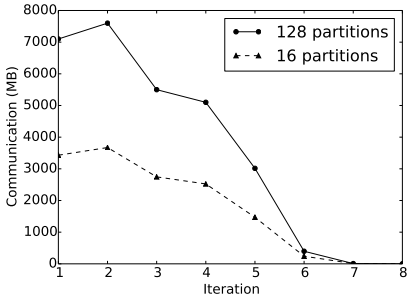



Figure 9: **Effect of partitioning on communication**

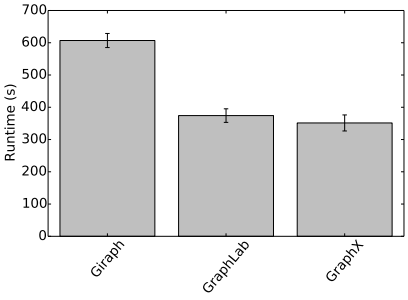



Figure 10: **End-to-end pipeline performance comparison**

| Dataset | Edges | Vertices |
|---|---|---|
| LiveJournal | 68,993,773 | 4,847,571 |
| Wikipedia | 116,841,365 | 6,556,598 |
| Twitter [3, 2] | 1,468,365,182 | 41,652,230 |

Table 1: **Graph datasets**

We evaluated the performance of GraphX on PageRank and Connected Components, two well-understood graph algorithms that are simple enough to serve as an effective measure of the system's performance rather than the performance of the user-defined functions.

For each system, we ran both algorithms on the Twitter and LiveJournal social network graphs (see Table 1). We used the implementations of these algorithms included in the Giraph and PowerGraph distributions, and we additionally implemented PageRank using idiomatic Spark dataflow operators.

Figures 7a and 7b show the total runtimes for the connected components algorithm running until convergence. On the Twitter graph, Giraph outperforms GraphX and is as fast as GraphLab despite the latter's highly optimized C++ implementation. We conjecture that this is due to the difference in partitioning strategies: GraphLab and GraphX use vertex cuts while Giraph uses edge cuts. Vertex cuts split high-degree vertices across partitions, but incur some overhead due to the joins and aggregation needed to coordinate vertex properties across partitions containing adjacent edges. The connected components algorithm does very little communication per iteration (see Figure 4), negating the benefit of vertex cuts but still incurring the overhead. In the case of LiveJournal, Giraph is slower because it uses Hadoop MapReduce for resource scheduling and the overhead of that (approximately 10 seconds) is quite substantial when the graph is small.

Figures 7c and 7d show the total runtimes for PageRank for 20 iterations on each system, including the idiomatic Spark dataflow implementation of PageRank. PageRank on GraphX is much faster than PageRank on Spark, and since GraphX is built on Spark, the difference can be isolated to the fact that GraphX exploits the graph structure using vertex cuts, structural indices, and the other optimizations described in Section 4. The specialized systems also outperform the Spark dataflow implementation for similar reasons.

In Figure 8 we plot the strong scaling performance of GraphX running PageRank on the Twitter follower graph. As we move from 8 to 32 machines (a factor of 4) we see a 3x speedup. However as we move to 64 machines (a factor of 8) we only see a 3.5x speedup. While this is hardly linear scaling, it is actually slightly better than the 3.2x speedup reported by PowerGraph [10]. The poor scaling performance of PageRank has been attributed by [10] to high communication overhead relative to computation.

The fact that GraphX is able to scale slightly better than PowerGraph is relatively surprising given that the Spark shared-nothing worker model eliminates the potential for shared memory parallelism and forces the graph to be partitioned across processors and not machines. However, Figure 9 shows the communication of GraphX as a function of the number of partitions. Going from 16 to 128 partitions (a factor of 8) yields only around a 2-fold increase in communication. Returning to the analysis conducted by [10], we find that the vertex-cut partitioning adopted by GraphX mitigates the 8-fold increase in communication due to Spark.

## 5.2 End-to-End Pipeline Performance

Specialized graph-parallel systems are much faster than data-parallel systems such as Hadoop MapReduce and Apache Spark for iterative graph algorithms, but they are not well suited for many of the operations found in a typical graph analytics pipeline. To illustrate the unification of graph-parallel and data-parallel analytics in GraphX, we evaluate the end-to-end performance of each system in performing a multi-step pipeline that determines the 20 most important articles in the English Wikipedia by PageRank.

This analytics pipeline contains three stages: (1) parsing an XML file containing a snapshot of all English Wikipedia articles and extracting the link graph, (2) computing PageRank on the link graph, and (3) joining the 20 highest-ranked articles with their full text. Existing graph processing systems focus only on stage 2, and we demonstrate that GraphX's unified approach provides better end-to-end performance than specialized graph-parallel systems even for simple pipelines.

Because Giraph and GraphLab do not support general data-parallel operations such as XML parsing, joins, or top-K, we implemented these operations in their pipelines by transferring data to and from a data-parallel system using files. We used Spark and HDFS for this purpose. The GraphX unified model was capable of expressing the entire pipeline.

Figure 10 shows the performance of each system's pipeline. Despite GraphLab's superior performance on the graph-parallel portion of the pipeline, GraphX outperforms it in end-to-end runtime by avoiding the overhead of serialization, replication, and disk I/O at the stage boundaries. The GraphX pipeline was also simpler and easier to write due to the unified programming model.

# 6. RELATED WORK

We have already discussed related work on graph-parallel engines extensively in Section 2. This section focuses on related work in RDF and data-parallel systems.

The Semantic Web movement led to several areas of related work. The *Resource Description Framework* (RDF) graph [15] is a flexible representation of data as a graph consisting of *subject-predicate-object* triplets (*e.g., NYC-isA-city*) viewed as directed edges (*e.g.,* NYC $\xrightarrow{\text{isA}}$ city). The property graph data model adopted by GraphX contains the RDF graph as a special case [19]. The property graph corresponding to an RDF graph contains the predicates as edge properties and the subjects and objects as vertex properties. In the RDF model the subject and predicate must be a Universal Resource Identifier (URI) and the value can either be a URI or a string literal. As a consequence complex vertex properties (*e.g.,* name, age, and interests) must actually be expressed as a subgraphs connected to a URI corresponding to a person. In this sense, the RDF may be thought of a normalized property graph. As a consequence the RDF graph does not closely model the original graph structure or exploit the inherent grouping of fields (*e.g.,* information about a user), which must therefore be materialized through repeated self joins. Nonetheless, we adopt some of the core ideas from the RDF work including the triples view of graphs.

Numerous systems [4, 17, 1] have been proposed for storing and executing queries against RDF graphs using query languages such as SPARQL [18]. These systems as well as the SPARQL query language target subgraph queries and aggregation for OLTP workloads where the focus is on low-latency rather than throughput and the query is over small subgraphs (*e.g.,* short paths). Furthermore, this work is geared towards the RDF graph data models. In contrast, graph computation systems generally operate on the entire graph by transforming properties rather than returning subsets of vertices with a focus on throughput. Nonetheless, we believe that some of the ideas developed for GraphX (*e.g.,* distributed graph representations) may be beneficial in the design of low-latency distributed graph query processing systems.

There has been recent work applying incremental iterative data-parallel systems to graph computation. Both Ewen et al. [9] and Murray et al. [16] proposed systems geared towards incremental iterative data-parallel computation and demonstrated performance gains for specialized implementations of PageRank. While this work demonstrates the importance of incremental updates in graph computation, neither proposed a graph oriented view of the data or graph specific optimizations beyond incremental data-flows.

# 7. DISCUSSION

In this work, we revisit the concept of **Physical Data Independence** in the context of graphs and collections. We posit that collections and graphs are not only logical data models presented to programmers but in fact can be efficiently implemented using the same physical representation of the underlying data. Through the GraphX abstraction we proposed a common substrate that allows these data to be viewed as both collections and graphs and supports efficient data-parallel and graph-parallel computation using a combination of in-memory indexing, data storage formats, and various join optimization techniques. Our experiments show that this common substrate can match the performance of specialized graph computation systems and support the composition of graphs and tables in a single data model. In this section, we discuss the impact of our discoveries.

**Domain Specific Views:** Historically, physical independence focused on the flexibility to implement different physical storage, indexing, and access strategies without changing the applications. We argue that physical independence also enables the presentation of *multiple logical views* with different semantics and corresponding constraints on the same physical data. Because each view individually restricts the computation, the underlying system in turn can exploit those restrictions to optimize its physical execution strategies. However, by allowing the composition of multiple views, the system is able to maintain a high degree of generality. Furthermore, the semantic properties of each view enable the design of domain specific operators which can be further specialized by the system. We believe there is opportunity for further research into the composition of specialized views (*e.g.,* queues and sets) and operators and their corresponding optimizations.

**Graph Computation as Joins:** The design of the GraphX system revealed a strong connection between distributed graph computation and distributed join optimizations. When viewed through the lens of relational operators, graph computation reduces to joining vertices with edges (*i.e.,* triplets) and then applying aggregations. These two stages correspond to the Scatter and Gather phases of the GAS abstraction [10]. Likewise, the optimizations used to distribute and accelerate the GAS abstraction correspond to horizontal partitioning and indexing strategies. In particular, the construction of the triplets relies heavily on distributed routing tables that resemble the

join site selection optimization in distributed databases. Exploiting the iterative nature in graph computation, GraphX reuses many of the intermediate data structures built for joins across iterations, and employs techniques in incremental view maintenance to optimize the joins. We hope this connection will inspire further research into distributed join optimizations for graph computation.

**The Narrow Waist:** In the design of the GraphX abstraction we sought to develop a thin extension on top of relational operators with the goal of identifying the essential data model and core operations needed to support graph computation and achieve a portable framework that can be embedded in a range of data-parallel platforms. We restricted our attention to the small set of primitive operators needed to express existing graph-parallel frameworks such as Pregel and PowerGraph. In doing so, we identified the property graph and its tabular analog the unordered type collection as the essential data-model, as well as a small set of basic operators which can be cast in relational operators. It is our hope that, as a consequence, the GraphX design can be adopted by other data-parallel systems, including MPP databases, to efficiently support a wide range of graph computations.

**Simplified Analytics Pipeline:** Some key benefits of GraphX are difficult to quantify. The ability to stay within a single framework throughout the analytics process means it is no longer necessary to learn and support multiple systems or develop the data interchange formats and plumbing needed to move betweens systems. As a consequence, it is substantially easier to iteratively slice, transform, and compute on large graphs and share code that spans a much larger part of the pipeline. The gains in performance and scalability for graph computation translate to a tighter analytics feedback loop and therefore a more efficient work flow. Finally, GraphX creates the opportunity for rich libraries of graph operators tailored to specific analytics tasks.

# 8. CONCLUSION

The growing scale and importance of graph data has driven the development of specialized graph computation engines capable of inferring complex recursive properties of graph-structured data. However, these systems are unable to express many of the inherently data-parallel stages in a typical graph-analytics pipeline. As a consequence, existing graph analytics pipelines [11] resort to multiple stages of data-parallel and graph-parallel systems composed through external storage systems. This approach to graph analytics is inefficient, difficult to adopt to new workloads, and difficult to maintain.

In this work we introduced GraphX, a distributed graph processing framework that unifies graph-parallel and data-parallel computation in a single system and is capable of succinctly expressing and efficiently executing the entire graph analytics pipeline. The GraphX abstraction unifies the data-parallel and graph-parallel abstractions through a data model that presents graphs and collections as first-class objects with a set of primitive operators enabling their composition. We demonstrated that these operators are expressive enough to implement the Pregel and PowerGraph abstractions but also simple enough to be cast in relational algebra.

GraphX encodes graphs as collections of edges and vertices along with simple auxiliary index structures, and represents graph computations as a sequence of relational joins and aggregations. It incorporates techniques such as incremental view maintenance and index scans in databases and adapts these techniques to exploit common characteristics of graph computation workloads. The result is a system that achieves performance comparable to contemporary graph-parallel systems in graph computation while retaining the expressiveness of contemporary data-parallel systems.

We have open sourced GraphX at www.anon-sys.com. Though it has not been officially released, a brave industry user has successfully deployed GraphX and achieved a speedup of two orders of magnitude over their pre-existing graph analytics pipelines.

# 9. REFERENCES


[1] D. J. Abadi et al. Sw-store: A vertically partitioned dbms for semantic web data management. *VLDB'09*. 6

[2] P. Boldi et al. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *WWW'11*. **??**

[3] P. Boldi and S. Vigna. The WebGraph framework I: Compression techniques. In *WWW'04*. **??**

[4] J. Broekstra et al. Sesame: A generic architecture for storing and querying rdf and rdf schema. In *ISWC 2002*. 6

[5] A. Buluç and J. R. Gilbert. The combinatorial blas: design, implementation, and applications. *IJHPCA*, 25(4):496–509, 2011. 1, 2.3

[6] U. V. Çatalyürek, C. Aykanat, and B. Uçar. On two-dimensional sparse matrix partitioning: Models, methods, and a recipe. *SIAM J. Sci. Comput.*, 32(2):656–683, 2010. 1

[7] R. Cheng et al. Kineograph: taking the pulse of a fast-changing and connected world. In *EuroSys*, 2012. 1, 2.3

[8] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI*, 2004. 1

[9] S. Ewen et al. Spinning fast iterative data flows. *VLDB'12*. 2.3, 6

[10] J. E. Gonzalez et al. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI '12*. 1, 2.3, 2.3, 2.3, 4, 5.1, 7

[11] N. Jain et al. Graphbuilder: Scalable graph etl framework. In *GRADES '13*. 1, 2.4, 8

[12] Y. Low et al. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *VLDB'2012*. 2.3, 2.3

[13] Y. Low et al. Graphlab: A new parallel framework for machine learning. In *UAI*, pages 340–349, 2010. 2.3

[14] G. Malewicz et al. Pregel: a system for large-scale graph processing. In *SIGMOD'10*. 1, 2.3, 2.3

[15] F. Manola and E. Miller. RDF primer. *W3C Recommendation*, 10:1–107, 2004. 6

[16] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. In *SOSP '13*. 6

[17] T. Neumann and G. Weikum. Rdf-3x: A risc-style engine for rdf. *VLDB'08*. 6

[18] E. Prud'hommeaux and A. Seaborne. Sparql query language for rdf. Latest version available as `http://www.w3.org/TR/rdf-sparql-query/`, January 2008. 6

[19] I. Robinson, J. Webber, and E. Eifrem. *Graph Databases*. O'Reilly Media, Incorporated, 2013. 2.1, 6

[20] A. Roy et al. X-stream: Edge-centric graph processing using streaming partitions. In *SOSP '13*. 2.3

[21] P. Stutz, A. Bernstein, and W. Cohen. Signal/collect: graph algorithms for the (semantic) web. In *ISWC*, 2010. 1, 2.3

[22] R. S. Xin et al. Shark: SQL and Rich Analytics at Scale. In *SIGMOD'13*. 1

[23] M. Zaharia et al. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. NSDI, 2012. 1, 4.1, 1

# Spark SQL: Relational Data Processing in Spark

Michael Armbrust[†], Reynold S. Xin[†], Cheng Lian[†], Yin Huai[†], Davies Liu[†], Joseph K. Bradley[†], Xiangrui Meng[†], Tomer Kaftan[‡], Michael J. Franklin[†‡], Ali Ghodsi[†], Matei Zaharia[†*]

[†]Databricks Inc. [*]MIT CSAIL [‡]AMPLab, UC Berkeley

## ABSTRACT

Spark SQL is a new module in Apache Spark that integrates relational processing with Spark's functional programming API. Built on our experience with Shark, Spark SQL lets Spark programmers leverage the benefits of relational processing (*e.g.,* declarative queries and optimized storage), and lets SQL users call complex analytics libraries in Spark (*e.g.,* machine learning). Compared to previous systems, Spark SQL makes two main additions. First, it offers much tighter integration between relational and procedural processing, through a declarative DataFrame API that integrates with procedural Spark code. Second, it includes a highly extensible optimizer, Catalyst, built using features of the Scala programming language, that makes it easy to add composable rules, control code generation, and define extension points. Using Catalyst, we have built a variety of features (*e.g.,* schema inference for JSON, machine learning types, and query federation to external databases) tailored for the complex needs of modern data analysis. We see Spark SQL as an evolution of both SQL-on-Spark and of Spark itself, offering richer APIs and optimizations while keeping the benefits of the Spark programming model.

## Categories and Subject Descriptors

H.2 [**Database Management**]: Systems

## Keywords

Databases; Data Warehouse; Machine Learning; Spark; Hadoop

## 1 Introduction

Big data applications require a mix of processing techniques, data sources and storage formats. The earliest systems designed for these workloads, such as MapReduce, gave users a powerful, but low-level, procedural programming interface. Programming such systems was onerous and required manual optimization by the user to achieve high performance. As a result, multiple new systems sought to provide a more productive user experience by offering relational interfaces to big data. Systems like Pig, Hive, Dremel and Shark [29, 36, 25, 38] all take advantage of declarative queries to provide richer automatic optimizations.


*SIGMOD'15,* May 31–June 4, 2015, Melbourne, Victoria, Australia.

ACM 978-1-4503-2758-9/15/05 ...$15.00.
http://dx.doi.org/10.1145/2723372.2742797.

While the popularity of relational systems shows that users often prefer writing declarative queries, the relational approach is insufficient for many big data applications. First, users want to perform ETL to and from various data sources that might be semi- or unstructured, requiring custom code. Second, users want to perform advanced analytics, such as machine learning and graph processing, that are challenging to express in relational systems. In practice, we have observed that most data pipelines would ideally be expressed with a combination of both relational queries and complex procedural algorithms. Unfortunately, these two classes of systems—relational and procedural—have until now remained largely disjoint, forcing users to choose one paradigm or the other.

This paper describes our effort to combine both models in Spark SQL, a major new component in Apache Spark [39]. Spark SQL builds on our earlier SQL-on-Spark effort, called Shark. Rather than forcing users to pick between a relational or a procedural API, however, Spark SQL lets users seamlessly intermix the two.

Spark SQL bridges the gap between the two models through two contributions. First, Spark SQL provides a *DataFrame API* that can perform relational operations on both external data sources and Spark's built-in distributed collections. This API is similar to the widely used data frame concept in R [32], but evaluates operations lazily so that it can perform relational optimizations. Second, to support the wide range of data sources and algorithms in big data, Spark SQL introduces a novel extensible optimizer called *Catalyst*. Catalyst makes it easy to add data sources, optimization rules, and data types for domains such as machine learning.

The DataFrame API offers rich relational/procedural integration within Spark programs. DataFrames are collections of structured records that can be manipulated using Spark's procedural API, or using new relational APIs that allow richer optimizations. They can be created directly from Spark's built-in distributed collections of Java/Python objects, enabling relational processing in existing Spark programs. Other Spark components, such as the machine learning library, take and produce DataFrames as well. DataFrames are more convenient and more efficient than Spark's procedural API in many common situations. For example, they make it easy to compute multiple aggregates in one pass using a SQL statement, something that is difficult to express in traditional functional APIs. They also automatically store data in a columnar format that is significantly more compact than Java/Python objects. Finally, unlike existing data frame APIs in R and Python, DataFrame operations in Spark SQL go through a relational optimizer, Catalyst.

To support a wide variety of data sources and analytics workloads in Spark SQL, we designed an extensible query optimizer called Catalyst. Catalyst uses features of the Scala programming language, such as pattern-matching, to express composable rules in a Turing-complete language. It offers a general framework for transforming

trees, which we use to perform analysis, planning, and runtime code generation. Through this framework, Catalyst can also be extended with new data sources, including semi-structured data such as JSON and "smart" data stores to which one can push filters (*e.g.,* HBase); with user-defined functions; and with user-defined types for domains such as machine learning. Functional languages are known to be well-suited for building compilers [37], so it is perhaps no surprise that they made it easy to build an extensible optimizer. We indeed have found Catalyst effective in enabling us to quickly add capabilities to Spark SQL, and since its release we have seen external contributors easily add them as well.

Spark SQL was released in May 2014, and is now one of the most actively developed components in Spark. As of this writing, Apache Spark is the most active open source project for big data processing, with over 400 contributors in the past year. Spark SQL has already been deployed in very large scale environments. For example, a large Internet company uses Spark SQL to build data pipelines and run queries on an 8000-node cluster with over 100 PB of data. Each individual query regularly operates on tens of terabytes. In addition, many users adopt Spark SQL not just for SQL queries, but in programs that combine it with procedural processing. For example, 2/3 of customers of Databricks Cloud, a hosted service running Spark, use Spark SQL within other programming languages. Performance-wise, we find that Spark SQL is competitive with SQL-only systems on Hadoop for relational queries. It is also up to $10\times$ faster and more memory-efficient than naive Spark code in computations expressible in SQL.

More generally, we see Spark SQL as an important evolution of the core Spark API. While Spark's original functional programming API was quite general, it offered only limited opportunities for automatic optimization. Spark SQL simultaneously makes Spark accessible to more users and improves optimizations for existing ones. Within Spark, the community is now incorporating Spark SQL into more APIs: DataFrames are the standard data representation in a new "ML pipeline" API for machine learning, and we hope to expand this to other components, such as GraphX and streaming.

We start this paper with a background on Spark and the goals of Spark SQL (§2). We then describe the DataFrame API (§3), the Catalyst optimizer (§4), and advanced features we have built on Catalyst (§5). We evaluate Spark SQL in §6. We describe external research built on Catalyst in §7. Finally, §8 covers related work.

# 2 Background and Goals

## 2.1 Spark Overview

Apache Spark is a general-purpose cluster computing engine with APIs in Scala, Java and Python and libraries for streaming, graph processing and machine learning [6]. Released in 2010, it is to our knowledge one of the most widely-used systems with a "language-integrated" API similar to DryadLINQ [20], and the most active open source project for big data processing. Spark had over 400 contributors in 2014, and is packaged by multiple vendors.

Spark offers a functional programming API similar to other recent systems [20, 11], where users manipulate distributed collections called Resilient Distributed Datasets (RDDs) [39]. Each RDD is a collection of Java or Python objects partitioned across a cluster. RDDs can be manipulated through operations like `map`, `filter`, and `reduce`, which take functions in the programming language and ship them to nodes on the cluster. For example, the Scala code below counts lines starting with "ERROR" in a text file:

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(s => s.contains("ERROR"))
println(errors.count())
```

This code creates an RDD of strings called `lines` by reading an HDFS file, then transforms it using `filter` to obtain another RDD, `errors`. It then performs a `count` on this data.

RDDs are fault-tolerant, in that the system can recover lost data using the lineage graph of the RDDs (by rerunning operations such as the `filter` above to rebuild missing partitions). They can also explicitly be cached in memory or on disk to support iteration [39].

One final note about the API is that RDDs are evaluated *lazily*. Each RDD represents a "logical plan" to compute a dataset, but Spark waits until certain output operations, such as `count`, to launch a computation. This allows the engine to do some simple query optimization, such as pipelining operations. For instance, in the example above, Spark will pipeline reading lines from the HDFS file with applying the filter and computing a running count, so that it never needs to materialize the intermediate `lines` and `errors` results. While such optimization is extremely useful, it is also limited because the engine does not understand the structure of the data in RDDs (which is arbitrary Java/Python objects) or the semantics of user functions (which contain arbitrary code).

## 2.2 Previous Relational Systems on Spark

Our first effort to build a relational interface on Spark was Shark [38], which modified the Apache Hive system to run on Spark and implemented traditional RDBMS optimizations, such as columnar processing, over the Spark engine. While Shark showed good performance and good opportunities for integration with Spark programs, it had three important challenges. First, Shark could only be used to query external data stored in the Hive catalog, and was thus not useful for relational queries on data *inside* a Spark program (*e.g.,* on the `errors` RDD created manually above). Second, the only way to call Shark from Spark programs was to put together a SQL string, which is inconvenient and error-prone to work with in a modular program. Finally, the Hive optimizer was tailored for MapReduce and difficult to extend, making it hard to build new features such as data types for machine learning or support for new data sources.

## 2.3 Goals for Spark SQL

With the experience from Shark, we wanted to extend relational processing to cover native RDDs in Spark and a much wider range of data sources. We set the following goals for Spark SQL:

1. Support relational processing both within Spark programs (on native RDDs) and on external data sources using a programmer-friendly API.
2. Provide high performance using established DBMS techniques.
3. Easily support new data sources, including semi-structured data and external databases amenable to query federation.
4. Enable extension with advanced analytics algorithms such as graph processing and machine learning.

# 3 Programming Interface

Spark SQL runs as a library on top of Spark, as shown in Figure 1. It exposes SQL interfaces, which can be accessed through JDBC/ODBC or through a command-line console, as well as the DataFrame API integrated into Spark's supported programming languages. We start by covering the DataFrame API, which lets users intermix procedural and relational code. However, advanced functions can also be exposed in SQL through UDFs, allowing them to be invoked, for example, by business intelligence tools. We discuss UDFs in Section 3.7.

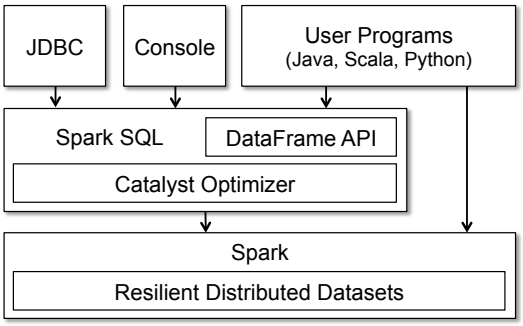



**Figure 1: Interfaces to Spark SQL, and interaction with Spark.**

### 3.1 DataFrame API

The main abstraction in Spark SQL's API is a DataFrame, a distributed collection of rows with the same schema. A DataFrame is equivalent to a table in a relational database, and can also be manipulated in similar ways to the "native" distributed collections in Spark (RDDs).[1] Unlike RDDs, DataFrames keep track of their schema and support various relational operations that lead to more optimized execution.

DataFrames can be constructed from tables in a system catalog (based on external data sources) or from existing RDDs of native Java/Python objects (Section 3.5). Once constructed, they can be manipulated with various relational operators, such as `where` and `groupBy`, which take expressions in a domain-specific language (DSL) similar to data frames in R and Python [32, 30]. Each DataFrame can also be viewed as an RDD of Row objects, allowing users to call procedural Spark APIs such as `map`.[2]

Finally, unlike traditional data frame APIs, Spark DataFrames are lazy, in that each DataFrame object represents a *logical plan* to compute a dataset, but no execution occurs until the user calls a special "output operation" such as `save`. This enables rich optimization across all operations that were used to build the DataFrame.

To illustrate, the Scala code below defines a DataFrame from a table in Hive, derives another based on it, and prints a result:

```
ctx = new HiveContext()
users = ctx.table("users")
young = users.where(users("age") < 21)
println(young.count())
```

In this code, `users` and `young` are DataFrames. The snippet `users("age") < 21` is an *expression* in the data frame DSL, which is captured as an abstract syntax tree rather than representing a Scala function as in the traditional Spark API. Finally, each DataFrame simply represents a logical plan (*i.e.,* read the users table and filter for age < 21). When the user calls `count`, which is an output operation, Spark SQL builds a physical plan to compute the final result. This might include optimizations such as only scanning the "age" column of the data if its storage format is columnar, or even using an index in the data source to count the matching rows.

We next cover the details of the DataFrame API.

### 3.2 Data Model

Spark SQL uses a nested data model based on Hive [19] for tables and DataFrames. It supports all major SQL data types, including boolean, integer, double, decimal, string, date, and timestamp, as well as complex (i.e., non-atomic) data types: structs, arrays, maps and unions. Complex data types can also be nested together to create more powerful types. Unlike many traditional DBMSes, Spark SQL provides first-class support for complex data types in the query language and the API. In addition, Spark SQL also supports user-defined types, as described in Section 4.4.2.

Using this type system, we have been able to accurately model data from a variety of sources and formats, including Hive, relational databases, JSON, and native objects in Java/Scala/Python.

### 3.3 DataFrame Operations

Users can perform relational operations on DataFrames using a domain-specific language (DSL) similar to R data frames [32] and Python Pandas [30]. DataFrames support all common relational operators, including projection (`select`), filter (`where`), `join`, and aggregations (`groupBy`). These operators all take *expression* objects in a limited DSL that lets Spark capture the structure of the expression. For example, the following code computes the number of female employees in each department.

```
employees
  .join(dept, employees("deptId") === dept("id"))
  .where(employees("gender") === "female")
  .groupBy(dept("id"), dept("name"))
  .agg(count("name"))
```

Here, `employees` is a DataFrame, and `employees("deptId")` is an expression representing the `deptId` column. Expression objects have many operators that return new expressions, including the usual comparison operators (*e.g.,* === for equality test, > for greater than) and arithmetic ones (+, -, etc). They also support aggregates, such as `count("name")`. All of these operators build up an abstract syntax tree (AST) of the expression, which is then passed to Catalyst for optimization. This is unlike the native Spark API that takes functions containing arbitrary Scala/Java/Python code, which are then opaque to the runtime engine. For a detailed listing of the API, we refer readers to Spark's official documentation [6].

Apart from the relational DSL, DataFrames can be registered as temporary tables in the system catalog and queried using SQL. The code below shows an example:

```
users.where(users("age") < 21)
     .registerTempTable("young")
ctx.sql("SELECT count(*), avg(age) FROM young")
```

SQL is sometimes convenient for computing multiple aggregates concisely, and also allows programs to expose datasets through JDBC/ODBC. The DataFrames registered in the catalog are still unmaterialized views, so that optimizations can happen *across* SQL and the original DataFrame expressions. However, DataFrames can also be materialized, as we discuss in Section 3.6.

### 3.4 DataFrames versus Relational Query Languages

While on the surface, DataFrames provide the same operations as relational query languages like SQL and Pig [29], we found that they can be significantly easier for users to work with thanks to their integration in a full programming language. For example, users can break up their code into Scala, Java or Python functions that pass DataFrames between them to build a logical plan, and will still benefit from optimizations across the *whole* plan when they run an output operation. Likewise, developers can use control structures like if statements and loops to structure their work. One user said that the DataFrame API is "concise and declarative like SQL, except I can name intermediate results," referring to how it is easier to structure computations and debug intermediate steps.

To simplify programming in DataFrames, we also made API analyze logical plans *eagerly* (*i.e.,* to identify whether the column

[1] We chose the name DataFrame because it is similar to structured data libraries in R and Python, and designed our API to resemble those.

[2] These Row objects are constructed on the fly and do not necessarily represent the internal storage format of the data, which is typically columnar.

names used in expressions exist in the underlying tables, and whether their data types are appropriate), even though query results are computed lazily. Thus, Spark SQL reports an error as soon as user types an invalid line of code instead of waiting until execution. This is again easier to work with than a large SQL statement.

### 3.5 Querying Native Datasets

Real-world pipelines often extract data from heterogeneous sources and run a wide variety of algorithms from different programming libraries. To interoperate with procedural Spark code, Spark SQL allows users to construct DataFrames directly against RDDs of objects native to the programming language. Spark SQL can automatically infer the schema of these objects using reflection. In Scala and Java, the type information is extracted from the language's type system (from JavaBeans and Scala case classes). In Python, Spark SQL samples the dataset to perform schema inference due to the dynamic type system.

For example, the Scala code below defines a DataFrame from an RDD of `User` objects. Spark SQL automatically detects the names ("name" and "age") and data types (string and int) of the columns.

```
case class User(name: String, age: Int)

// Create an RDD of User objects
usersRDD = spark.parallelize(
  List(User("Alice", 22), User("Bob", 19)))

// View the RDD as a DataFrame
usersDF = usersRDD.toDF
```

Internally, Spark SQL creates a logical data scan operator that points to the RDD. This is compiled into a physical operator that accesses fields of the native objects. It is important to note that this is very different from traditional object-relational mapping (ORM). ORMs often incur expensive conversions that translate an entire object into a different format. In contrast, Spark SQL accesses the native objects in-place, extracting only the fields used in each query.

The ability to query native datasets lets users run optimized relational operations within existing Spark programs. In addition, it makes it simple to combine RDDs with external structured data. For example, we could join the `users` RDD with a table in Hive:

```
views = ctx.table("pageviews")
usersDF.join(views, usersDF("name") === views("user"))
```

### 3.6 In-Memory Caching

Like Shark before it, Spark SQL can materialize (often referred to as "cache") hot data in memory using columnar storage. Compared with Spark's native cache, which simply stores data as JVM objects, the columnar cache can reduce memory footprint by an order of magnitude because it applies columnar compression schemes such as dictionary encoding and run-length encoding. Caching is particularly useful for interactive queries and for the iterative algorithms common in machine learning. It can be invoked by calling `cache()` on a DataFrame.

### 3.7 User-Defined Functions

User-defined functions (UDFs) have been an important extension point for database systems. For example, MySQL relies on UDFs to provide basic support for JSON data. A more advanced example is MADLib's use of UDFs to implement machine learning algorithms for Postgres and other database systems [12]. However, database systems often require UDFs to be defined in a separate programming environment that is different from the primary query interfaces. Spark SQL's DataFrame API supports inline definition of UDFs, without the complicated packaging and registration process found in other database systems. This feature has proven crucial for the adoption of the API.

In Spark SQL, UDFs can be registered inline by passing Scala, Java or Python functions, which may use the full Spark API internally. For example, given a `model` object for a machine learning model, we could register its prediction function as a UDF:

```
val model: LogisticRegressionModel = ...

ctx.udf.register("predict",
  (x: Float, y: Float) => model.predict(Vector(x, y)))

ctx.sql("SELECT predict(age, weight) FROM users")
```

Once registered, the UDF can also be used via the JDBC/ODBC interface by business intelligence tools. In addition to UDFs that operate on scalar values like the one here, one can define UDFs that operate on an entire table by taking its name, as in MADLib [12], and use the distributed Spark API within them, thus exposing advanced analytics functions to SQL users. Finally, because UDF definitions and query execution are expressed using the same general-purpose language (*e.g.,* Scala or Python), users can debug or profile the entire program using standard tools.

The example above demonstrates a common use case in many pipelines, *i.e.,* one that employs both relational operators and advanced analytics methods that are cumbersome to express in SQL. The DataFrame API lets developers seamlessly mix these methods.

## 4 Catalyst Optimizer

To implement Spark SQL, we designed a new extensible optimizer, Catalyst, based on functional programming constructs in Scala. Catalyst's extensible design had two purposes. First, we wanted to make it easy to add new optimization techniques and features to Spark SQL, especially to tackle various problems we were seeing specifically with "big data" (*e.g.,* semistructured data and advanced analytics). Second, we wanted to enable external developers to extend the optimizer—for example, by adding data source specific rules that can push filtering or aggregation into external storage systems, or support for new data types. Catalyst supports both rule-based and cost-based optimization.

While extensible optimizers have been proposed in the past, they have typically required a complex domain specific language to specify rules, and an "optimizer compiler" to translate the rules into executable code [17, 16]. This leads to a significant learning curve and maintenance burden. In contrast, Catalyst uses standard features of the Scala programming language, such as pattern-matching [14], to let developers use the full programming language while still making rules easy to specify. Functional languages were designed in part to build compilers, so we found Scala well-suited to this task. Nonetheless, Catalyst is, to our knowledge, the first production-quality query optimizer built on such a language.

At its core, Catalyst contains a general library for representing *trees* and applying *rules* to manipulate them.[3] On top of this framework, we have built libraries specific to relational query processing (*e.g.,* expressions, logical query plans), and several sets of rules that handle different phases of query execution: analysis, logical optimization, physical planning, and code generation to compile parts of queries to Java bytecode. For the latter, we use another Scala feature, quasiquotes [34], that makes it easy to generate code at runtime from composable expressions. Finally, Catalyst offers several public extension points, including external data sources and user-defined types.

[3]Cost-based optimization is performed by generating multiple plans using rules, and then computing their costs.

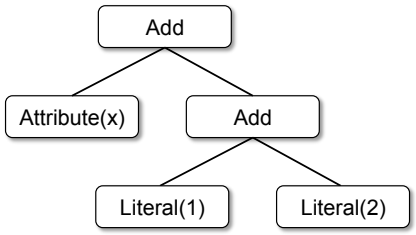



**Figure 2: Catalyst tree for the expression `x+(1+2)`.**

### 4.1 Trees

The main data type in Catalyst is a *tree* composed of *node* objects. Each node has a node type and zero or more children. New node types are defined in Scala as subclasses of the `TreeNode` class. These objects are immutable and can be manipulated using functional transformations, as discussed in the next subsection.

As a simple example, suppose we have the following three node classes for a very simple expression language:[4]

- `Literal(value: Int)`: a constant value
- `Attribute(name: String)`: an attribute from an input row, *e.g.,* "x"
- `Add(left: TreeNode, right: TreeNode)`: sum of two expressions.

These classes can be used to build up trees; for example, the tree for the expression `x+(1+2)`, shown in Figure 2, would be represented in Scala code as follows:

```
Add(Attribute(x), Add(Literal(1), Literal(2)))
```

### 4.2 Rules

Trees can be manipulated using *rules*, which are functions from a tree to another tree. While a rule can run arbitrary code on its input tree (given that this tree is just a Scala object), the most common approach is to use a set of *pattern matching* functions that find and replace subtrees with a specific structure.

Pattern matching is a feature of many functional languages that allows extracting values from potentially nested structures of algebraic data types. In Catalyst, trees offer a `transform` method that applies a pattern matching function recursively on all nodes of the tree, transforming the ones that match each pattern to a result. For example, we could implement a rule that folds Add operations between constants as follows:

```
tree.transform {
  case Add(Literal(c1), Literal(c2)) => Literal(c1+c2)
}
```

Applying this to the tree for `x+(1+2)`, in Figure 2, would yield the new tree `x+3`. The `case` keyword here is Scala's standard pattern matching syntax [14], and can be used to match on the type of an object as well as give names to extracted values (`c1` and `c2` here).

The pattern matching expression that is passed to `transform` is a *partial function*, meaning that it only needs to match to a subset of all possible input trees. Catalyst will tests which parts of a tree a given rule applies to, automatically skipping over and descending into subtrees that do not match. This ability means that rules only need to reason about the trees where a given optimization applies and not those that do not match. Thus, rules do not need to be modified as new types of operators are added to the system.

[4]We use Scala syntax for classes here, where each class's fields are defined in parentheses, with their types given using a colon.

Rules (and Scala pattern matching in general) can match multiple patterns in the same `transform` call, making it very concise to implement multiple transformations at once:

```
tree.transform {
  case Add(Literal(c1), Literal(c2)) => Literal(c1+c2)
  case Add(left, Literal(0)) => left
  case Add(Literal(0), right) => right
}
```

In practice, rules may need to execute multiple times to fully transform a tree. Catalyst groups rules into *batches*, and executes each batch until it reaches a *fixed point*, that is, until the tree stops changing after applying its rules. Running rules to fixed point means that each rule can be simple and self-contained, and yet still eventually have larger global effects on a tree. In the example above, repeated application would constant-fold larger trees, such as `(x+0)+(3+3)`. As another example, a first batch might analyze an expression to assign types to all of the attributes, while a second batch might use these types to do constant folding. After each batch, developers can also run sanity checks on the new tree (*e.g.,* to see that all attributes were assigned types), often also written via recursive matching.

Finally, rule conditions and their bodies can contain arbitrary Scala code. This gives Catalyst more power than domain specific languages for optimizers, while keeping it concise for simple rules.

In our experience, functional transformations on immutable trees make the whole optimizer very easy to reason about and debug. They also enable parallelization in the optimizer, although we do not yet exploit this.

### 4.3 Using Catalyst in Spark SQL

We use Catalyst's general tree transformation framework in four phases, shown in Figure 3: (1) analyzing a logical plan to resolve references, (2) logical plan optimization, (3) physical planning, and (4) code generation to compile parts of the query to Java bytecode. In the physical planning phase, Catalyst may generate multiple plans and compare them based on cost. All other phases are purely rule-based. Each phase uses different types of tree nodes; Catalyst includes libraries of nodes for expressions, data types, and logical and physical operators. We now describe each of these phases.

#### 4.3.1 Analysis

Spark SQL begins with a relation to be computed, either from an abstract syntax tree (AST) returned by a SQL parser, or from a DataFrame object constructed using the API. In both cases, the relation may contain unresolved attribute references or relations: for example, in the SQL query `SELECT col FROM sales`, the type of `col`, or even whether it is a valid column name, is not known until we look up the table `sales`. An attribute is called unresolved if we do not know its type or have not matched it to an input table (or an alias). Spark SQL uses Catalyst rules and a Catalog object that tracks the tables in all data sources to resolve these attributes. It starts by building an "unresolved logical plan" tree with unbound attributes and data types, then applies rules that do the following:

- Looking up relations by name from the catalog.
- Mapping named attributes, such as `col`, to the input provided given operator's children.
- Determining which attributes refer to the same value to give them a unique ID (which later allows optimization of expressions such as `col = col`).
- Propagating and coercing types through expressions: for example, we cannot know the type of `1 + col` until we have resolved `col` and possibly cast its subexpressions to compatible types.

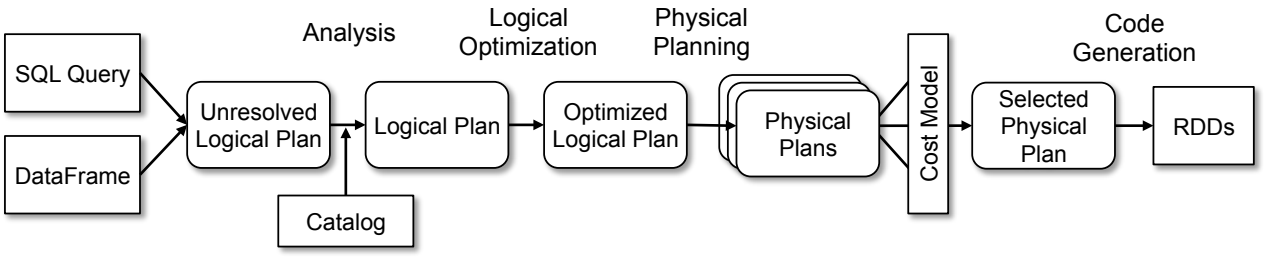



**Figure 3: Phases of query planning in Spark SQL. Rounded rectangles represent Catalyst trees.**

In total, the rules for the analyzer are about 1000 lines of code.

#### 4.3.2 Logical Optimization

The logical optimization phase applies standard rule-based optimizations to the logical plan. These include constant folding, predicate pushdown, projection pruning, null propagation, Boolean expression simplification, and other rules. In general, we have found it extremely simple to add rules for a wide variety of situations. For example, when we added the fixed-precision `DECIMAL` type to Spark SQL, we wanted to optimize aggregations such as sums and averages on `DECIMAL`s with small precisions; it took 12 lines of code to write a rule that finds such decimals in `SUM` and `AVG` expressions, and casts them to unscaled 64-bit `LONG`s, does the aggregation on that, then converts the result back. A simplified version of this rule that only optimizes SUM expressions is reproduced below:

```
object DecimalAggregates extends Rule[LogicalPlan] {
  /** Maximum number of decimal digits in a Long */
  val MAX_LONG_DIGITS = 18

  def apply(plan: LogicalPlan): LogicalPlan = {
    plan transformAllExpressions {
      case Sum(e @ DecimalType.Expression(prec, scale))
             if prec + 10 <= MAX_LONG_DIGITS =>
        MakeDecimal(Sum(LongValue(e)), prec + 10, scale)
    }
}
```

As another example, a 12-line rule optimizes `LIKE` expressions with simple regular expressions into `String.startsWith` or `String.contains` calls. The freedom to use arbitrary Scala code in rules made these kinds of optimizations, which go beyond pattern-matching the structure of a subtree, easy to express. In total, the logical optimization rules are 800 lines of code.

#### 4.3.3 Physical Planning

In the physical planning phase, Spark SQL takes a logical plan and generates one or more physical plans, using physical operators that match the Spark execution engine. It then selects a plan using a cost model. At the moment, cost-based optimization is only used to select join algorithms: for relations that are known to be small, Spark SQL uses a broadcast join, using a peer-to-peer broadcast facility available in Spark.[5] The framework supports broader use of cost-based optimization, however, as costs can be estimated recursively for a whole tree using a rule. We thus intend to implement richer cost-based optimization in the future.

The physical planner also performs rule-based physical optimizations, such as pipelining projections or filters into one Spark `map` operation. In addition, it can push operations from the logical plan into data sources that support predicate or projection pushdown. We will describe the API for these data sources in Section 4.4.1.

In total, the physical planning rules are about 500 lines of code.

[5]Table sizes are estimated if the table is cached in memory or comes from an external file, or if it is the result of a subquery with a `LIMIT`.

#### 4.3.4 Code Generation

The final phase of query optimization involves generating Java bytecode to run on each machine. Because Spark SQL often operates on in-memory datasets, where processing is CPU-bound, we wanted to support code generation to speed up execution. Nonetheless, code generation engines are often complicated to build, amounting essentially to a compiler. Catalyst relies on a special feature of the Scala language, quasiquotes [34], to make code generation simpler. Quasiquotes allow the programmatic construction of abstract syntax trees (ASTs) in the Scala language, which can then be fed to the Scala compiler at runtime to generate bytecode. We use Catalyst to transform a tree representing an expression in SQL to an AST for Scala code to evaluate that expression, and then compile and run the generated code.

As a simple example, consider the Add, Attribute and Literal tree nodes introduced in Section 4.2, which allowed us to write expressions such as `(x+y)+1`. Without code generation, such expressions would have to be interpreted for each row of data, by walking down a tree of Add, Attribute and Literal nodes. This introduces large amounts of branches and virtual function calls that slow down execution. With code generation, we can write a function to translate a specific expression tree to a Scala AST as follows:

```
def compile(node: Node): AST = node match {
  case Literal(value) => q"$value"
  case Attribute(name) => q"row.get($name)"
  case Add(left, right) =>
    q"${compile(left)} + ${compile(right)}"
}
```

The strings beginning with `q` are quasiquotes, meaning that although they look like strings, they are parsed by the Scala compiler at compile time and represent ASTs for the code within. Quasiquotes can have variables or other ASTs spliced into them, indicated using `$` notation. For example, `Literal(1)` would become the Scala AST for `1`, while `Attribute("x")` becomes `row.get("x")`. In the end, a tree like `Add(Literal(1), Attribute("x"))` becomes an AST for a Scala expression like `1+row.get("x")`.

Quasiquotes are type-checked at compile time to ensure that only appropriate ASTs or literals are substituted in, making them significantly more useable than string concatenation, and they result directly in a Scala AST instead of running the Scala parser at runtime. Moreover, they are highly composable, as the code generation rule for each node does not need to know how the trees returned by its children were built. Finally, the resulting code is further optimized by the Scala compiler in case there are expression-level optimizations that Catalyst missed. Figure 4 shows that quasiquotes let us generate code with performance similar to hand-tuned programs.

We have found quasiquotes very straightforward to use for code generation, and we observed that even new contributors to Spark SQL could quickly add rules for new types of expressions. Quasiquotes also work well with our goal of running on native Java

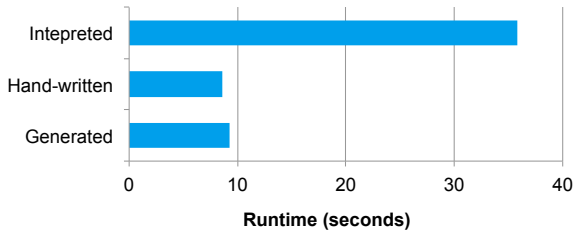



**Figure 4: A comparision of the performance evaluating the expresion `x+x+x`, where `x` is an integer, 1 billion times.**

objects: when accessing fields from these objects, we can code-generate a direct access to the required field, instead of having to copy the object into a Spark SQL Row and use the Row's accessor methods. Finally, it was straightforward to combine code-generated evaluation with interpreted evaluation for expressions we do not yet generate code for, since the Scala code we compile can directly call into our expression interpreter.

In total, Catalyst's code generator is about 700 lines of code.

### 4.4 Extension Points

Catalyst's design around composable rules makes it easy for users and third-party libraries to extend. Developers can add batches of rules to each phase of query optimization at runtime, as long as they adhere to the contract of each phase (*e.g.,* ensuring that analysis resolves all attributes). However, to make it even simpler to add some types of extensions without understanding Catalyst rules, we have also defined two narrower public extension points: data sources and user-defined types. These still rely on facilities in the core engine to interact with the rest of the rest of the optimizer.

#### 4.4.1 Data Sources

Developers can define a new data source for Spark SQL using several APIs, which expose varying degrees of possible optimization. All data sources must implement a `createRelation` function that takes a set of key-value parameters and returns a `BaseRelation` object for that relation, if one can be successfully loaded. Each `BaseRelation` contains a schema and an optional estimated size in bytes.[6] For instance, a data source representing MySQL may take a table name as a parameter, and ask MySQL for an estimate of the table size.

To let Spark SQL read the data, a `BaseRelation` can implement one of several interfaces that let them expose varying degrees of sophistication. The simplest, `TableScan`, requires the relation to return an RDD of `Row` objects for all of the data in the table. A more advanced `PrunedScan` takes an array of column names to read, and should return Rows containing only those columns. A third interface, `PrunedFilteredScan`, takes both desired column names and an array of Filter objects, which are a subset of Catalyst's expression syntax, allowing predicate pushdown.[7] The filters are advisory, *i.e.,* the data source should attempt to return only rows passing each filter, but it is allowed to return false positives in the case of filters that it cannot evaluate. Finally, a CatalystScan interface is given a complete sequence of Catalyst expression trees to use in predicate pushdown, though they are again advisory.

These interfaces allow data sources to implement various degrees of optimization, while still making it easy for developers to add simple data sources of virtually any type. We and others have used the interface to implement the following data sources:

- CSV files, which simply scan the whole file, but allow users to specify a schema.
- Avro [4], a self-describing binary format for nested data.
- Parquet [5], a columnar file format for which we support column pruning as well as filters.
- A JDBC data source that scans ranges of a table from an RDBMS in parallel and pushes filters into the RDBMS to minimize communication.

To use these data sources, programmers specify their package names in SQL statements, passing key-value pairs for configuration options. For example, the Avro data source takes a path to the file:

```
CREATE TEMPORARY TABLE messages
USING com.databricks.spark.avro
OPTIONS (path "messages.avro")
```

All data sources can also expose network locality information, *i.e.,* which machines each partition of the data is most efficient to read from. This is exposed through the RDD objects they return, as RDDs have a built-in API for data locality [39].

Finally, similar interfaces exist for writing data to an existing or new table. These are simpler because Spark SQL just provides an RDD of Row objects to be written.

#### 4.4.2 User-Defined Types (UDTs)

One feature we wanted to allow advanced analytics in Spark SQL was user-defined types. For example, machine learning applications may need a vector type, and graph algorithms may need types for representing a graph, which is possible over relational tables [15]. Adding new types can be challenging, however, as data types pervade all aspects of the execution engine. For example, in Spark SQL, the built-in data types are stored in a columnar, compressed format for in-memory caching (Section 3.6), and in the data source API from the previous section, we need to expose all possible data types to data source authors.

In Catalyst, we solve this issue by mapping user-defined types to structures composed of Catalyst's built-in types, described in Section 3.2. To register a Scala type as a UDT, users provide a mapping from an object of their class to a Catalyst Row of built-in types, and an inverse mapping back. In user code, they can now use the Scala type in objects that they query with Spark SQL, and it will be converted to built-in types under the hood. Likewise, they can register UDFs (see Section 3.7) that operate directly on their type.

As a short example, suppose we want to register two-dimensional points $(x, y)$ as a UDT. We can represent such vectors as two `DOUBLE` values. To register the UDT, we write the following:

```
class PointUDT extends UserDefinedType[Point] {
  def dataType = StructType(Seq( // Our native structure
    StructField("x", DoubleType),
    StructField("y", DoubleType)
  ))
  def serialize(p: Point) = Row(p.x, p.y)
  def deserialize(r: Row) =
    Point(r.getDouble(0), r.getDouble(1))
}
```

After registering this type, Points will be recognized within native objects that Spark SQL is asked to convert to DataFrames, and will be passed to UDFs defined on Points. In addition, Spark SQL will store Points in a columnar format when caching data (compressing `x` and `y` as separate columns), and Points will be writable to all of Spark SQL's data sources, which will see them as pairs of `DOUBLE`s. We use this capability in Spark's machine learning library, as we describe in Section 5.2.

[6] Unstructured data sources can also take a desired schema as a parameter; for example, there is a CSV file data source that lets users specify column names and types.

[7] At the moment, Filters include equality, comparisons against a constant, and `IN` clauses, each on one attribute.

```
{
  "text": "This is a tweet about #Spark",
  "tags": ["#Spark"],
  "loc": {"lat": 45.1, "long": 90}
}

{
  "text": "This is another tweet",
  "tags": [],
  "loc": {"lat": 39, "long": 88.5}
}

{
  "text": "A #tweet without #location",
  "tags": ["#tweet", "#location"]
}
```

**Figure 5: A sample set of JSON records, representing tweets.**

```
text STRING NOT NULL,
tags ARRAY<STRING NOT NULL> NOT NULL,
loc STRUCT<lat FLOAT NOT NULL, long FLOAT NOT NULL>
```

**Figure 6: Schema inferred for the tweets in Figure 5.**

# 5 Advanced Analytics Features

In this section, we describe three features we added to Spark SQL specifically to handle challenges in "big data" environments. First, in these environments, data is often unstructured or semistructured. While parsing such data procedurally is possible, it leads to lengthy boilerplate code. To let users query the data right away, Spark SQL includes a schema inference algorithm for JSON and other semistructured data. Second, large-scale processing often goes beyond aggregation and joins to machine learning on the data. We describe how Spark SQL is being incorporated into a new high-level API for Spark's machine learning library [26]. Last, data pipelines often combine data from disparate storage systems. Building on the data sources API in Section 4.4.1, Spark SQL supports query federation, allowing a single program to efficiently query disparate sources. These features all build on the Catalyst framework.

## 5.1 Schema Inference for Semistructured Data

Semistructured data is common in large-scale environments because it is easy to produce and to add fields to over time. Among Spark users, we have seen very high usage of JSON for input data. Unfortunately, JSON is cumbersome to work with in a procedural environment like Spark or MapReduce: most users resorted to ORM-like libraries (*e.g.,* Jackson [21]) to map JSON structures to Java objects, or some tried parsing each input record directly with lower-level libraries.

In Spark SQL, we added a JSON data source that automatically infers a schema from a set of records. For example, given the JSON objects in Figure 5, the library infers the schema shown in Figure 6. Users can simply register a JSON file as a table and query it with syntax that accesses fields by their path, such as:

```
SELECT loc.lat, loc.long FROM tweets
WHERE text LIKE '%Spark%' AND tags IS NOT NULL
```

Our schema inference algorithm works in one pass over the data, and can also be run on a sample of the data if desired. It is related to prior work on schema inference for XML and object databases [9, 18, 27], but simpler because it only infers a static tree structure, without allowing recursive nesting of elements at arbitrary depths.

Specifically, the algorithm attempts to infer a tree of `STRUCT` types, each of which may contain atoms, arrays, or other `STRUCT`s. For each field defined by a distinct path from the root JSON object (*e.g.,* `tweet.loc.latitude`), the algorithm finds the most specific Spark SQL data type that matches observed instances of the field. For example, if all occurrences of that field are integers that fit into 32 bits, it will infer `INT`; if they are larger, it will use `LONG` (64-bit) or `DECIMAL` (arbitrary precision); if there are also fractional values, it will use `FLOAT`. For fields that display multiple types, Spark SQL uses `STRING` as the most generic type, preserving the original JSON representation. And for fields that contain arrays, it uses the same "most specific supertype" logic to determine an element type from all the observed elements. We implement this algorithm using a single `reduce` operation over the data, which starts with schemata (*i.e.,* trees of types) from each individual record and merges them using an associative "most specific supertype" function that generalizes the types of each field. This makes the algorithm both single-pass and communication-efficient, as a high degree of reduction happens locally on each node.

As a short example, note how in Figures 5 and 6, the algorithm generalized the types of `loc.lat` and `loc.long`. Each field appears as an integer in one record and a floating-point number in another, so the algorithm returns `FLOAT`. Note also how for the `tags` field, the algorithm inferred an array of strings that cannot be null.

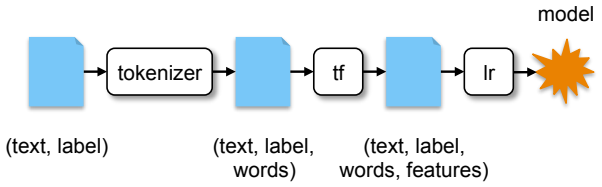
In practice, we have found this algorithm to work well with real-world JSON datasets. For example, it correctly identifies a usable schema for JSON tweets from Twitter's firehose, which contain around 100 distinct fields and a high degree of nesting. Multiple Databricks customers have also successfully applied it to their internal JSON formats.

In Spark SQL, we also use the same algorithm for inferring schemas of RDDs of Python objects (see Section 3), as Python is not statically typed so an RDD can contain multiple object types. In the future, we plan to add similar inference for CSV files and XML. Developers have found the ability to view these types of datasets as tables and immediately query them or join them with other data extremely valuable for their productivity.

## 5.2 Integration with Spark's Machine Learning Library

As an example of Spark SQL's utility in other Spark modules, MLlib, Spark's machine learning library, introduced a new high-level API that uses DataFrames [26]. This new API is based on the concept of machine learning *pipelines*, an abstraction in other high-level ML libraries like SciKit-Learn [33]. A pipeline is a graph of transformations on data, such as feature extraction, normalization, dimensionality reduction, and model training, each of which exchange *datasets*. Pipelines are a useful abstraction because ML workflows have many steps; representing these steps as composable elements makes it easy to change parts of the pipeline or to search for tuning parameters at the level of the whole workflow.

To exchange data between pipeline stages, MLlib's developers needed a format that was compact (as datasets can be large) yet flexible, allowing multiple types of fields to be stored for each record. For example, a user may start with records that contain text fields as well as numeric ones, then run a featurization algorithm such as TF-IDF on the text to turn it into a vector, normalize one of the other fields, perform dimensionality reduction on the whole set of features, etc. To represent datasets, the new API uses DataFrames, where each column represents a feature of the data. All algorithms that can be called in pipelines take a name for the input column(s) and output column(s), and can thus be called on any subset of the fields and produce new ones. This makes it easy for developers to build complex pipelines while retaining the original data for each record. To illustrate the API, Figure 7 shows a short pipeline and the schemas of DataFrames created.




```
data = <DataFrame of (text, label) records>

tokenizer = Tokenizer()
  .setInputCol("text").setOutputCol("words")
tf = HashingTF()
  .setInputCol("words").setOutputCol("features")
lr = LogisticRegression()
  .setInputCol("features")

pipeline = Pipeline().setStages([tokenizer, tf, lr])
model = pipeline.fit(data)
```

**Figure 7: A short MLlib pipeline and the Python code to run it. We start with a DataFrame of (text, label) records, tokenize the text into words, run a term frequency featurizer (`HashingTF`) to get a feature vector, then train logistic regression.**

The main piece of work MLlib had to do to use Spark SQL was to create a user-defined type for vectors. This vector UDT can store both sparse and dense vectors, and represents them as four primitive fields: a boolean for the type (dense or sparse), a size for the vector, an array of indices (for sparse coordinates), and an array of double values (either the non-zero coordinates for sparse vectors or all coordinates otherwise). Apart from DataFrames' utility for tracking and manipulating columns, we also found them useful for another reason: they made it much easier to expose MLlib's new API in all of Spark's supported programming languages. Previously, each algorithm in MLlib took objects for domain-specific concepts (*e.g.,* a labeled point for classification, or a (user, product) rating for recommendation), and each of these classes had to be implemented in the various languages (*e.g.,* copied from Scala to Python). Using DataFrames everywhere made it much simpler to expose all algorithms in all languages, as we only need data conversions in Spark SQL, where they already exist. This is especially important as Spark adds bindings for new programming languages.

Finally, using DataFrames for storage in MLlib also makes it very easy to expose all its algorithms in SQL. We can simply define a MADlib-style UDF, as described in Section 3.7, which will internally call the algorithm on a table. We are also exploring APIs to expose pipeline construction in SQL.

### 5.3 Query Federation to External Databases

Data pipelines often combine data from heterogeneous sources. For example, a recommendation pipeline might combine traffic logs with a user profile database and users' social media streams. As these data sources often reside in different machines or geographic locations, naively querying them can be prohibitively expensive. Spark SQL data sources leverage Catalyst to push predicates down into the data sources whenever possible.

For example, the following uses the JDBC data source and the JSON data source to join two tables together to find the traffic log for the most recently registered users. Conveniently, both data sources can automatically infer the schema without users having to define it. The JDBC data source will also push the filter predicate down into MySQL to reduce the amount of data transferred.

```
CREATE TEMPORARY TABLE users USING jdbc
OPTIONS(driver "mysql" url "jdbc:mysql://userDB/users")

CREATE TEMPORARY TABLE logs
USING json OPTIONS (path "logs.json")

SELECT users.id, users.name, logs.message
FROM users JOIN logs WHERE users.id = logs.userId
AND users.registrationDate > "2015-01-01"
```

Under the hood, the JDBC data source uses the PrunedFilteredScan interface in Section 4.4.1, which gives it both the names of the columns requested and simple predicates (equality, comparison and `IN` clauses) on these columns. In this case, the JDBC data source will run the following query on MySQL:[8]

```
SELECT users.id, users.name FROM users
WHERE users.registrationDate > "2015-01-01"
```

In future Spark SQL releases, we are also looking to add predicate pushdown for key-value stores such as HBase and Cassandra, which support limited forms of filtering.

## 6 Evaluation

We evaluate the performance of Spark SQL on two dimensions: SQL query processing performance and Spark program performance. In particular, we demonstrate that Spark SQL's extensible architecture not only enables a richer set of functionalities, but brings substantial performance improvements over previous Spark-based SQL engines. In addition, for Spark application developers, the DataFrame API can bring substantial speedups over the native Spark API while making Spark programs more concise and easier to understand. Finally, applications that combine relational and procedural queries run faster on the integrated Spark SQL engine than by running SQL and procedural code as separate parallel jobs.

### 6.1 SQL Performance

We compared the performance of Spark SQL against Shark and Impala [23] using the AMPLab big data benchmark [3], which uses a web analytics workload developed by Pavlo et al. [31]. The benchmark contains four types of queries with different parameters performing scans, aggregation, joins and a UDF-based MapReduce job. We used a cluster of six EC2 `i2.xlarge` machines (one master, five workers) each with 4 cores, 30 GB memory and an 800 GB SSD, running HDFS 2.4, Spark 1.3, Shark 0.9.1 and Impala 2.1.1. The dataset was 110 GB of data after compression using the columnar Parquet format [5].

Figure 8 shows the results for each query, grouping by the query type. Queries 1–3 have different parameters varying their selectivity, with 1a, 2a, etc being the most selective and 1c, 2c, etc being the least selective and processing more data. Query 4 uses a Python-based Hive UDF that was not directly supported in Impala, but was largely bound by the CPU cost of the UDF.

We see that in all queries, Spark SQL is substantially faster than Shark and generally competitive with Impala. The main reason for the difference with Shark is code generation in Catalyst (Section 4.3.4), which reduces CPU overhead. This feature makes Spark SQL competitive with the C++ and LLVM based Impala engine in many of these queries. The largest gap from Impala is in query 3a where Impala chooses a better join plan because the selectivity of the queries makes one of the tables very small.

### 6.2 DataFrames vs. Native Spark Code

In addition to running SQL queries, Spark SQL can also help non-SQL developers write simpler and more efficient Spark code through the DataFrame API. Catalyst can perform optimizations on

[8]The JDBC data source also supports "sharding" a source table by a particular column and reading different ranges of it in parallel.

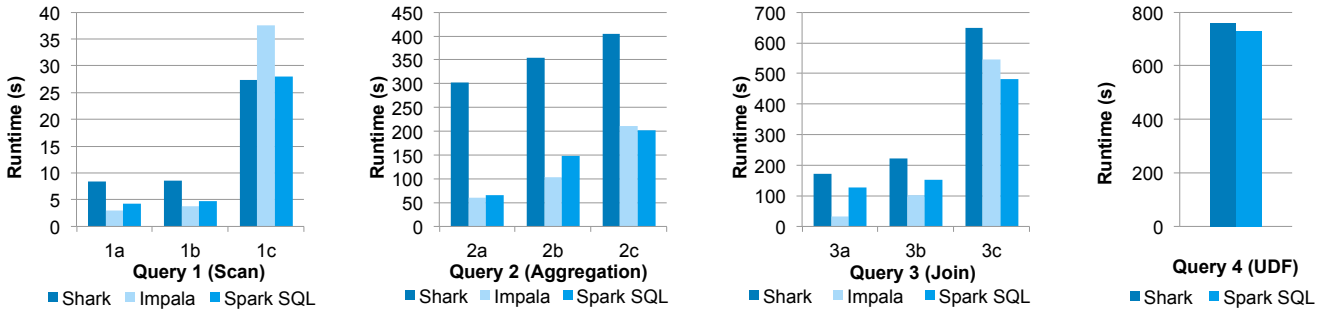



**Figure 8: Performance of Shark, Impala and Spark SQL on the big data benchmark queries [31].**

DataFrame operations that are hard to do with hand written code, such as predicate pushdown, pipelining, and automatic join selection. Even without these optimizations, the DataFrame API can result in more efficient execution due to code generation. This is especially true for Python applications, as Python is typically slower than the JVM.

For this evaluation, we compared two implementations of a Spark program that does a distributed aggregation. The dataset consists of 1 billion integer pairs, `(a, b)` with 100,000 distinct values of `a`, on the same five-worker `i2.xlarge` cluster as in the previous section. We measure the time taken to compute the average of `b` for each value of `a`. First, we look at a version that computes the average using the `map` and `reduce` functions in the Python API for Spark:

```
sum_and_count = \
  data.map(lambda x: (x.a, (x.b, 1))) \
      .reduceByKey(lambda x, y: (x[0]+y[0], x[1]+y[1])) \
      .collect()
[(x[0], x[1][0] / x[1][1]) for x in sum_and_count]
```

In contrast, the same program can written as a simple manipulation using the DataFrame API:

```
df.groupBy("a").avg("b")
```

Figure 9, shows that the DataFrame version of the code outperforms the hand written Python version by 12×, in addition to being much more concise. This is because in the DataFrame API, only the logical plan is constructed in Python, and all physical execution is compiled down into native Spark code as JVM bytecode, resulting in more efficient execution. In fact, the DataFrame version also outperforms a Scala version of the Spark code above by 2×. This is mainly due to code generation: the code in the DataFrame version avoids expensive allocation of key-value pairs that occurs in hand-written Scala code.

### 6.3 Pipeline Performance

The DataFrame API can also improve performance in applications that combine relational and procedural processing, by letting developers write all operations in a single program and pipelining computation across relational and procedural code. As a simple example, we consider a two-stage pipeline that selects a subset of text messages from a corpus and computes the most frequent words. Although very simple, this can model some real-world pipelines, *e.g.,* computing the most popular words used in tweets by a specific demographic.

In this experiment, we generated a synthetic dataset of 10 billion messages in HDFS. Each message contained on average 10 words drawn from an English dictionary. The first stage of the pipeline uses a relational filter to select roughly 90% of the messages. The second stage computes the word count.

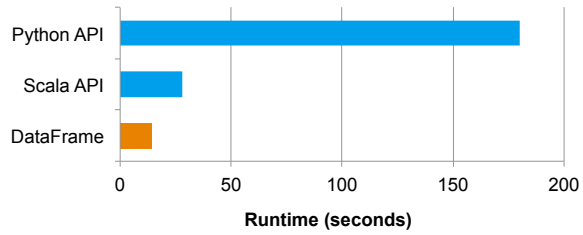



**Figure 9: Performance of an aggregation written using the native Spark Python and Scala APIs versus the DataFrame API.**

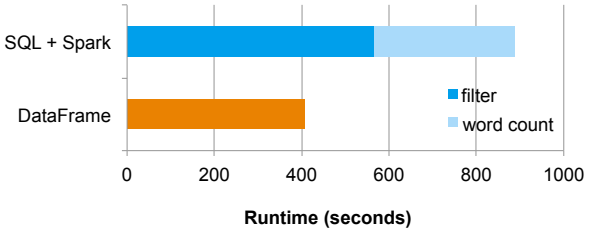



**Figure 10: Performance of a two-stage pipeline written as a separate Spark SQL query and Spark job (above) and an integrated DataFrame job (below).**

First, we implemented the pipeline using a separate SQL query followed by a Scala-based Spark job, as might occur in environments that run separate relational and procedural engines (*e.g.,* Hive and Spark). We then implemented a combined pipeline using the DataFrame API, *i.e.,* using DataFrame's relational operators to perform the filter, and using the RDD API to perform a word count on the result. Compared with the first pipeline, the second pipeline avoids the cost of saving the whole result of the SQL query to an HDFS file as an intermediate dataset before passing it into the Spark job, because SparkSQL pipelines the `map` for the word count with the relational operators for the filtering. Figure 10 compares the runtime performance of the two approaches. In addition to being easier to understand and operate, the DataFrame-based pipeline also improves performance by 2×.

## 7 Research Applications

In addition to the immediately practical production use cases of Spark SQL, we have also seen significant interest from researchers working on more experimental projects. We outline two research

projects that leverage the extensibility of Catalyst: one in approximate query processing and one in genomics.

### 7.1 Generalized Online Aggregation

Zeng et al. have used Catalyst in their work on improving the generality of online aggregation [40]. This work generalizes the execution of online aggregation to support arbitrarily nested aggregate queries. It allows users to view the progress of executing queries by seeing results computed over a fraction of the total data. These partial results also include accuracy measures, letting the user stop the query when sufficient accuracy has been reached.

In order to implement this system inside of Spark SQL, the authors add a new operator to represent a relation that has been broken up into sampled batches. During query planning a call to `transform` is used to replace the original full query with several queries, each of which operates on a successive sample of the data.

However, simply replacing the full dataset with samples is not sufficient to compute the correct answer in an online fashion. Operations such as standard aggregation must be replaced with stateful counterparts that take into account both the current sample and the results of previous batches. Furthermore, operations that might filter out tuples based on approximate answers must be replaced with versions that can take into account the current estimated errors.

Each of these transformations can be expressed as Catalyst rules that modify the operator tree until it produces correct online answers. Tree fragments that are not based on sampled data are ignored by these rules and can execute using the standard code path. By using Spark SQL as a basis, the authors were able to implement a fairly complete prototype in approximately 2000 lines of code.

### 7.2 Computational Genomics

A common operation in computational genomics involves inspecting overlapping regions based on a numerical offsets. This problem can be represented as a join with inequality predicates. Consider two datasets, `a` and `b`, with a schema of `(start LONG, end LONG)`. The range join operation can be expressed in SQL as follows:

```
SELECT * FROM a JOIN b
WHERE a.start < a.end
  AND b.start < b.end
  AND a.start < b.start
  AND b.start < a.end
```

Without special optimization, the preceding query would be executed by many systems using an inefficient algorithm such as a nested loop join. In contrast, a specialized system could compute the answer to this join using an interval tree. Researchers in the ADAM project [28] were able to build a special planning rule into a version of Spark SQL to perform such computations efficiently, allowing them to leverage the standard data manipulation abilities alongside specialized processing code. The changes required were approximately 100 lines of code.

## 8 Related Work

**Programming Model** Several systems have sought to combine relational processing with the procedural processing engines initially used for large clusters. Of these, Shark [38] is the closest to Spark SQL, running on the same engine and offering the same combination of relational queries and advanced analytics. Spark SQL improves on Shark through a richer and more programmer-friendly API, DataFrames, where queries can be combined in a modular way using constructs in the host programming language (see Section 3.4). It also allows running relational queries directly on native RDDs, and supports a wide range of data sources beyond Hive.

One system that inspired Spark SQL's design was DryadLINQ [20], which compiles language-integrated queries in C# to a distributed DAG execution engine. LINQ queries are also relational but can operate directly on C# objects. Spark SQL goes beyond DryadLINQ by also providing a DataFrame interface similar to common data science libraries [32, 30], an API for data sources and types, and support for iterative algorithms through execution on Spark.

Other systems use only a relational data model internally and relegate procedural code to UDFs. For example, Hive and Pig [36, 29] offer relational query languages but have widely used UDF interfaces. ASTERIX [8] has a semi-structured data model internally. Stratosphere [2] also has a semi-structured model, but offers APIs in Scala and Java that let users easily call UDFs. PIQL [7] likewise provides a Scala DSL. Compared to these systems, Spark SQL integrates more closely with native Spark applications by being able to directly query data in user-defined classes (native Java/Python objects), and lets developers mix procedural and relational APIs in the same language. In addition, through the Catalyst optimizer, Spark SQL implements both optimizations (*e.g.,* code generation) and other functionality (*e.g.,* schema inference for JSON and machine learning data types) that are not present in most large-scale computing frameworks. We believe that these features are essential to offering an integrated, easy-to-use environment for big data.

Finally, data frame APIs have been built both for single machines [32, 30] and clusters [13, 10]. Unlike previous APIs, Spark SQL optimizes DataFrame computations with a relational optimizer.

**Extensible Optimizers** The Catalyst optimizer shares similar goals with extensible optimizer frameworks such as EXODUS [17] and Cascades [16]. Traditionally, however, optimizer frameworks have required a domain-specific language to write rules in, as well as an "optimizer compiler" to translate them to runnable code. Our major improvement here is to build our optimizer using standard features of a functional programming language, which provide the same (and often greater) expressivity while decreasing the maintenance burden and learning curve. Advanced language features helped with many areas of Catalyst—for example, our approach to code generation using quasiquotes (Section 4.3.4) is one of the simplest and most composable approaches to this task that we know. While extensibility is hard to measure quantitatively, one promising indication is that Spark SQL had over 50 external contributors in the first 8 months after its release.

For code generation, LegoBase [22] recently proposed an approach using generative programming in Scala, which would be possible to use instead of quasiquotes in Catalyst.

**Advanced Analytics** Spark SQL builds on recent work to run advanced analytics algorithms on large clusters, including platforms for iterative algorithms [39] and graph analytics [15, 24]. The desire to expose analytics functions is also shared with MADlib [12], though the approach there is different, as MADlib had to use the limited interface of Postgres UDFs, while Spark SQL's UDFs can be full-fledged Spark programs. Finally, techniques including Sinew and Invisible Loading [35, 1] have sought to provide and optimize queries over semi-structured data such as JSON. We hope to apply some of these techniques in our JSON data source.

## 9 Conclusion

We have presented Spark SQL, a new module in Apache Spark providing rich integration with relational processing. Spark SQL extends Spark with a declarative DataFrame API to allow relational processing, offering benefits such as automatic optimization, and letting users write complex pipelines that mix relational and complex analytics. It supports a wide range of features tailored to large-scale

data analysis, including semi-structured data, query federation, and data types for machine learning. To enable these features, Spark SQL is based on an extensible optimizer called Catalyst that makes it easy to add optimization rules, data sources and data types by embedding into the Scala programming language. User feedback and benchmarks show that Spark SQL makes it significantly simpler and more efficient to write data pipelines that mix relational and procedural processing, while offering substantial speedups over previous SQL-on-Spark engines.

Spark SQL is open source at `http://spark.apache.org`.

## 10 Acknowledgments

We would like to thank Cheng Hao, Tayuka Ueshin, Tor Myklebust, Daoyuan Wang, and the rest of the Spark SQL contributors so far. We would also like to thank John Cieslewicz and the other members of the F1 team at Google for early discussions on the Catalyst optimizer. The work of authors Franklin and Kaftan was supported in part by: NSF CISE Expeditions Award CCF-1139158, LBNL Award 7076018, and DARPA XData Award FA8750-12-2-0331, and gifts from Amazon Web Services, Google, SAP, The Thomas and Stacey Siebel Foundation, Adatao, Adobe, Apple, Inc., Blue Goji, Bosch, C3Energy, Cisco, Cray, Cloudera, EMC2, Ericsson, Facebook, Guavus, Huawei, Informatica, Intel, Microsoft, NetApp, Pivotal, Samsung, Schlumberger, Splunk, Virdata and VMware.

## 11 References

[1] A. Abouzied, D. J. Abadi, and A. Silberschatz. Invisible loading: Access-driven data transfer from raw files into database systems. In *EDBT*, 2013.

[2] A. Alexandrov et al. The Stratosphere platform for big data analytics. *The VLDB Journal*, 23(6):939–964, Dec. 2014.

[3] AMPLab big data benchmark. https://amplab.cs.berkeley.edu/benchmark.

[4] Apache Avro project. http://avro.apache.org.

[5] Apache Parquet project. http://parquet.incubator.apache.org.

[6] Apache Spark project. http://spark.apache.org.

[7] M. Armbrust, N. Lanham, S. Tu, A. Fox, M. J. Franklin, and D. A. Patterson. The case for PIQL: a performance insightful query language. In *SOCC*, 2010.

[8] A. Behm et al. Asterix: towards a scalable, semistructured data platform for evolving-world models. *Distributed and Parallel Databases*, 29(3):185–216, 2011.

[9] G. J. Bex, F. Neven, and S. Vansummeren. Inferring XML schema definitions from XML data. In *VLDB*, 2007.

[10] BigDF project. https://github.com/AyasdiOpenSource/bigdf.

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

[12] J. Cohen, B. Dolan, M. Dunlap, J. Hellerstein, and C. Welton. MAD skills: new analysis practices for big data. *VLDB*, 2009.

[13] DDF project. http://ddf.io.

[14] B. Emir, M. Odersky, and J. Williams. Matching objects with patterns. In *ECOOP 2007 – Object-Oriented Programming, volume 4609 of LNCS*, pages 273–298. Springer, 2007.

[15] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. GraphX: Graph processing in a distributed dataflow framework. In *OSDI*, 2014.

[16] G. Graefe. The Cascades framework for query optimization. *IEEE Data Engineering Bulletin*, 18(3), 1995.

[17] G. Graefe and D. DeWitt. The EXODUS optimizer generator. In *SIGMOD*, 1987.

[18] J. Hegewald, F. Naumann, and M. Weis. XStruct: efficient schema extraction from multiple and large XML documents. In *ICDE Workshops*, 2006.

[19] Hive data definition language. https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DDL.

[20] M. Isard and Y. Yu. Distributed data-parallel computing using a high-level programming language. In *SIGMOD*, 2009.

[21] Jackson JSON processor. http://jackson.codehaus.org.

[22] Y. Klonatos, C. Koch, T. Rompf, and H. Chafi. Building efficient query engines in a high-level language. *PVLDB*, 7(10):853–864, 2014.

[23] M. Kornacker et al. Impala: A modern, open-source SQL engine for Hadoop. In *CIDR*, 2015.

[24] Y. Low et al. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *VLDB*, 2012.

[25] S. Melnik et al. Dremel: interactive analysis of web-scale datasets. *Proc. VLDB Endow.*, 3:330–339, Sept 2010.

[26] X. Meng, J. Bradley, E. Sparks, and S. Venkataraman. ML pipelines: a new high-level API for MLlib. https://databricks.com/blog/2015/01/07/ml-pipelines-a-new-high-level-api-for-mllib.html.

[27] S. Nestorov, S. Abiteboul, and R. Motwani. Extracting schema from semistructured data. In *ICDM*, 1998.

[28] F. A. Nothaft, M. Massie, T. Danford, Z. Zhang, U. Laserson, C. Yeksigian, J. Kottalam, A. Ahuja, J. Hammerbacher, M. Linderman, M. J. Franklin, A. D. Joseph, and D. A. Patterson. Rethinking data-intensive science using scalable analytics systems. In *SIGMOD*, 2015.

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

[30] `pandas` Python data analysis library. http://pandas.pydata.org.

[31] A. Pavlo et al. A comparison of approaches to large-scale data analysis. In *SIGMOD*, 2009.

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

[33] scikit-learn: machine learning in Python. http://scikit-learn.org.

[34] D. Shabalin, E. Burmako, and M. Odersky. Quasiquotes for Scala, a technical report. Technical Report 185242, École Polytechnique Fédérale de Lausanne, 2013.

[35] D. Tahara, T. Diamond, and D. J. Abadi. Sinew: A SQL system for multi-structured data. In *SIGMOD*, 2014.

[36] A. Thusoo et al. Hive–a petabyte scale data warehouse using Hadoop. In *ICDE*, 2010.

[37] P. Wadler. Monads for functional programming. In *Advanced Functional Programming*, pages 24–52. Springer, 1995.

[38] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: SQL and rich analytics at scale. In *SIGMOD*, 2013.

[39] M. Zaharia et al. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.

[40] K. Zeng et al. G-OLA: Generalized online aggregation for interactive analysis on big data. In *SIGMOD*, 2015.

# SparkR: Scaling R Programs with Spark

Shivaram Venkataraman[1], Zongheng Yang[1], Davies Liu[2], Eric Liang[2], Hossein Falaki[2]
Xiangrui Meng[2], Reynold Xin[2], Ali Ghodsi[2], Michael Franklin[1], Ion Stoica[1,2], Matei Zaharia[2,3]
[1]AMPLab UC Berkeley, [2] Databricks Inc., [3] MIT CSAIL

## ABSTRACT

R is a popular statistical programming language with a number of extensions that support data processing and machine learning tasks. However, interactive data analysis in R is usually limited as the R runtime is single threaded and can only process data sets that fit in a single machine's memory. We present SparkR, an R package that provides a frontend to Apache Spark and uses Spark's distributed computation engine to enable large scale data analysis from the R shell. We describe the main design goals of SparkR, discuss how the high-level DataFrame API enables scalable computation and present some of the key details of our implementation.

## 1. INTRODUCTION

Recent trends in big data analytics indicate the growing need for interactive analysis of large datasets. In response to this trend, a number of academic [12, 32, 8] and commercial systems [18] have been developed to support such use cases. However, data science surveys [1] show that in addition to relational query processing, data scientists often use tools like R to perform more advanced analysis on data. R is particularly popular as it provides support for structured data processing using data frames and includes a number of packages for statistical analysis and visualization.

However, data analysis using R is limited by the amount of memory available on a single machine and further as R is single threaded it is often impractical to use R on large datasets. Prior research has addressed some of these limitations through better I/O support [35], integration with Hadoop [13, 19] and by designing distributed R runtimes [28] that can be integrated with DBMS engines [25].

In this paper, we look at how we can scale R programs while making it easy to use and deploy across a number of workloads. We present SparkR: an R frontend for Apache Spark, a widely deployed [2] cluster computing engine. There are a number of benefits to designing an R frontend that is tightly integrated with Spark.
**Library Support**: The Spark project contains libraries for running SQL queries [10], distributed machine learning [23], graph analytics [16] and SparkR can reuse well-tested, distributed implementations for these domains.
**Data Sources**: Further, Spark SQL's data sources API provides support for reading input from a variety of systems including HDFS, HBase, Cassandra and a number of formats like JSON, Parquet, etc. Integrating with the data source API enables R users to directly process data sets from any of these data sources.
**Performance Improvements**: As opposed to a new distributed engine, SparkR can inherit all of the optimizations made to the Spark computation engine in terms of task scheduling, code generation, memory management [3], etc.

SparkR is built as an R package and requires no changes to R. The central component of SparkR is a distributed data frame that enables structured data processing with a syntax familiar to R users [31](Figure 1). To improve performance over large datasets, SparkR performs lazy evaluation on data frame operations and uses Spark's relational query optimizer [10] to optimize execution.

SparkR was initially developed at the AMPLab, UC Berkeley and has been a part of the Apache Spark project for the past eight months. SparkR is an active project with over 40 contributors and growing adoption [6, 7]. We next outline the design goals of SparkR and key details of our implementation. Following that we outline some of the efforts in progress to improve SparkR.

## 2. BACKGROUND

In this section we first provide a brief overview of Spark and R, the two main systems that are used to develop SparkR. We then discuss common application patterns used by R programmers for large scale data processing.

### 2.1 Apache Spark

Apache Spark [2] is a general purpose engine for large scale data processing. The Spark project first introduced Resilient Distributed Datasets (RDD) [34], an API for fault tolerant computation in a cluster computing environment. More recently a number of higher level APIs have been developed in Spark. These include MLlib [23], a library for large scale machine learning, GraphX [16], a library for processing large graphs and SparkSQL [10] a SQL API for analytical queries. Since the above libraries are closely integrated with the core API, Spark enables complex workflows where say SQL queries can be used to pre-process data and the results can then be analyzed using advanced machine learning algorithms. SparkSQL also includes Catalyst [10], a distributed query optimizer that improves performance by generating the optimal physical plan for a given query. More recent efforts [9] have looked at developing a high level distributed DataFrame API for structured data processing. As queries on DataFrames are executed using the SparkSQL query optimizer, DataFrames provide both better usability and performance compared to using RDDs [4]. We next discuss some of the important characteristics of data frames in the context of the R programming language.


*SIGMOD '16, June 26–July 1, 2016, San Francisco, CA, USA.*

DOI: http://dx.doi.org/XXXX.XXXX

## 2.2 R Project for Statistical Computing

The R project [26] consists of a programming language, an interactive development environment and a set of statistical computing libraries. R is an interpreted language and provides support for common constructs such as conditional execution (`if`) and loops (`for`, `while`, `repeat`) etc. R also includes extensive support for numerical computing, with data types for vectors, matrices, arrays and libraries for performing numerical operations.

**Data frames in R**: In addition to numerical computing, R provides support for structured data processing through data frames. Data frames are tabular data structures where each column consists of elements of a particular type (e.g., numerical or categorical). Data frames provide an easy syntax for filtering, summarizing data and packages like `dplyr` [31] have further simplified expressing complex data manipulation tasks on data frames. Specifically, `dplyr` provides a small number of *verbs* for data manipulation and these include relational operations like selection, projection, aggregations and joins. Given its popularity among users, the concept of data frames has been adopted by other languages like Pandas [21] for Python etc. Next, we look at some of the common workflows of data scientists who use R as their primary programming language and motivate our design for SparkR based on these workflows.

## 2.3 Application Patterns

**Big Data, Small Learning**: In this pattern, users typically start with a large dataset that is stored as a JSON or CSV file. Data analysis begins by joining the required datasets and users then perform data cleaning operations to remove invalid rows or columns. Following this users typically aggregate or sample their data and this step reduces the size of the dataset. The pre-processed data is then used for building models or performing other statistical tasks.

**Partition Aggregate**: Partition aggregate workflows are useful for a number of statistical applications like ensemble learning, parameter tuning or bootstrap aggregation. In these cases the user typically has a particular function that needs to be executed in parallel across different partitions of the input dataset and the results from each partition are then combined using a aggregation function. Additionally in some cases the input data could be small, but the same data is evaluated with a large number of parameter values.

**Large Scale Machine Learning**: Finally for some applications users run machine learning algorithms on large datasets. In such scenarios, the data is typically pre-processed to generate features and then the training features, labels are given as input to a machine learning algorithm to fit a model. The model being fit is usually much smaller in size compared to the input data and the model is then used to serve predictions.

We next present SparkR DataFrames and discuss how they can be used to address the above use cases.

# 3. DESIGN

In this section we present some of the design choices involved in building SparkR. We first present details about the SparkR DataFrames API and then present an overview of SparkR's architecture.

## 3.1 SparkR DataFrames API

The central component of SparkR is a distributed data frame implemented on top of Spark. SparkR DataFrames have an API similar to dplyr or local R data frames, but scale to large datasets using Spark's execution engine and relational query optimizer [10].

**DataFrame Operators**: SparkR's DataFrame supports a number of methods to read input and perform structured data analysis. As shown in Figure 1, SparkR's `read.df` method integrates with

```
1  # Load the flights CSV file using `read.df`
2  df <- read.df(sqlContext, "./nycflights13.csv",
3                "com.databricks.spark.csv")
4
5  # Select flights from JFK.
6  jfk_flights <- filter(df, df$origin == "JFK")
7
8  # Group and aggregate flights to each destination.
9  dest_flights <- agg(
10     groupBy(jfk_flights, jfk_flights$dest),
11     count = n(jfk_flights$dest))
12
13 # Running SQL Queries
14 registerTempTable(df, "table")
15 training <- sql(sqlContext,
16   "SELECT distance, depDelay, arrDelay FROM table")
```

**Figure 1: Example of the SparkR DataFrame API**

```
1  dest_flights <- filter(df, df$origin == "JFK") %>%
2      groupBy(df$dest) %>%
3      summarize(count = n(df$dest))
```

**Figure 2: Chaining DataFrame operators in SparkR**

Spark's data source API and this enables users to load data from systems like HBase, Cassandra etc. Having loaded the data, users are then able to use a familiar syntax for performing relational operations like selections, projections, aggregations and joins (lines 6–11). Further, SparkR supports more than 100 pre-defined functions on DataFrames including string manipulation methods, statistical functions and date-time operations. Users can also execute SQL queries directly on SparkR DataFrames using the `sql` command (lines 15–16). SparkR also makes it easy for users to chain commands using existing R libraries [11] as shown in Figure 2. Finally, SparkR DataFrames can be converted to a local R data frame using the `collect` operator and this is useful for the big data, small learning scenarios described earlier.

**Optimizations**: One of the main advantages of the high-level DataFrame API is that we can tightly integrate the R API with the optimized SQL execution engine in Spark. This means that even though users write their code in R, we do not incur overheads of running interpreted R code and can instead achieve the same performance as using Scala or SQL. For example, Figure 4 compares the performance of running group-by aggregation on 10 million integer pairs on a single machine using Spark with R, Python and Scala. From the figure we can see that SparkR's performance is similar to that of Scala / Python and this shows the benefits of separating the logical specification in R from the physical execution.

## 3.2 Architecture

SparkR's architecture consists of two main components as shown in Figure 3: an R to JVM binding on the driver that allows R programs to submit jobs to a Spark cluster and support for running R on the Spark executors. We discuss both these components below.

### *3.2.1 Bridging R and JVM*

One of the key challenges in implementing SparkR is having support for invoking Spark functions on a JVM from R. The main requirements we need to satisfy here include (a) a flexible approach where the JVM driver process could be launched independently by say a cluster manager like YARN (b) cross-platform support on Windows, Linux, etc. (c) a lightweight solution that does not make it cumbersome to install SparkR. While there are some existing packages which support starting an in-process JVM [27] we found

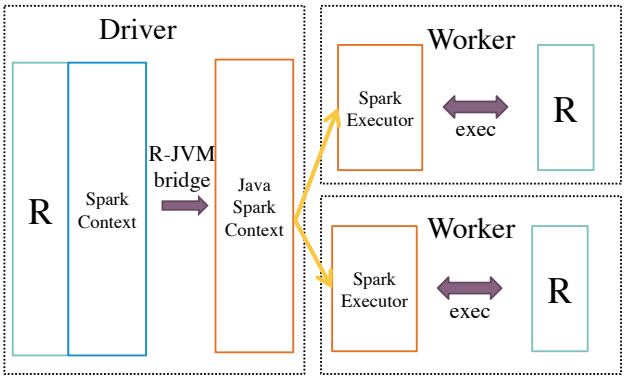



**Figure 3: SparkR Architecture**

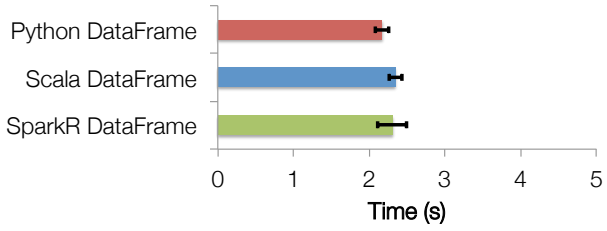



**Figure 4: SparkR Performance Comparison with Python, Scala APIs**

that these methods do not meet all our requirements.

Thus we developed a new socket-based SparkR internal API that can be used to invoke functions on the JVM from R. Our high level design is inspired by existing RPC or RMI-based systems [29] and we introduce a new SparkR JVM backend that listens on a Netty-based socket server. Our main reason for using sockets is that they are supported across platforms (in both Java and R) and are available without using any external libraries in both languages. As most of the messages being passed are control messages, the cost of using sockets as compared other in-process communication methods is not very high.

There are two kinds of RPCs we support in the SparkR JVM backend: method invocation and creating new objects. Method invocations are called using a reference to an existing Java object (or class name for static methods) and a list of arguments to be passed on to the method. The arguments are serialized using our custom wire format which is then deserialized on the JVM side. We then use Java reflection to invoke the appropriate method. In order to create objects, we use a special method name `init` and then similarly invoke the appropriate constructor based on the provided arguments. Finally, we use a new R class 'jobj' that refers to a Java object existing in the backend. These references are tracked on the Java side and are automatically garbage collected when they go out of scope on the R side.

### 3.2.2 *Spawning R workers*

The second part of SparkR's design consists of support to launch R processes on Spark executor machines. Our initial approach here was to fork an R process each time we need to run an R function. This is expensive because there are fixed overheads in launching the R process and in transferring the necessary inputs such as the Spark broadcast variables, input data, etc. We made two optimizations which reduce this overhead significantly. First, we implemented

```
# Query 1
# Top-5 destinations for flights departing from JFK.
jfk_flights <- filter(flights, flights$Origin == "JFK")
head(agg(group_by(jfk_flights, jfk_flights$Dest),
         count = n(jfk_flights$Dest)), 5L)

# Query 2
# Calculate the average delay across all flights.
collect(summarize(flights,
                  avg\_delay = mean(flights$DepDelay)))

# Query 3
# Count the number of distinct flight numbers.
count(distinct(select(flights, flights$TailNum)))
```

**Figure 7: Queries used for evaluation with the flights dataset**

support for coalescing R operations which lets us combine a number of R functions that need to be run. This is similar to operator pipelining used in database execution engines. Second, we added support for having a daemon R process that lives throughout the lifetime of a Spark job and manages the worker R processes using the `mcfork` feature in `parallel` package [26]. These optimizations both reduce the fixed overhead and the number of times we invoke an R process and help lower the end-to-end latency.

## 4. EVALUATION

In this section we evaluate some of our design choices described in the previous sections and also study how SparkR scales as we use more machines. The dataset we use in this section is the airline on-time performance dataset[1] that is used to evaluate existing R packages like dplyr [30]. This dataset contains arrival data for flights in USA and includes information such as departure and arrival delays, origin and destination airports etc. We use data across six years (2009-2014) and overall our input has 37.27M rows and 110 columns. The queries we use to evaluate SparkR are listed in Figure 7. The queries make use of filtering, aggregation and sorting and are representative of interactive queries used by R users. We use a cluster of 32 `r3.xlarge` machines on Amazon EC2 for our evaluation. Each machine consists of 2 physical cores, 30GB of memory and 80GB of SSD storage. All experiments were also run using Apache Spark 1.6.0 and we used the `spark-csv`[2] package for reading our input.

### 4.1 Strong Scaling

We first study the scaling behavior of SparkR by executing the three queries in Figure 7 and varying the number of cores used. In this experiment, the input data is directly processed from HDFS and not cached in memory. The time taken for each query as we vary the number of cores from 8 to 64 is shown in Figure 5. From the figure we can see that SparkR achieves near-linear scaling with the time taken reducing from around 115 seconds with 8 cores to around 20 seconds with 64 cores. However waiting for 20 seconds is often sub-optimal for interactive queries and we next see how caching data in memory can improve performance.

### 4.2 Importance of Caching

For studying the benefits of caching the input table in memory we fix the number of cores used as 64 and measure the time taken by each query when the input data is cached. Results from this experiment are shown in Figure 6. We see that caching the data can improve performance by $10x$ to $30x$ for this workload. These

[1]http://www.transtats.bts.gov/Tables.asp?DB_ID=120
[2]http://github.com/databricks/spark-csv

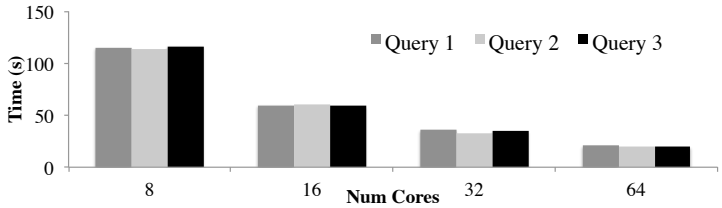



**Figure 5: Query performance as we scale the number of cores used for three queries from Figure 7**

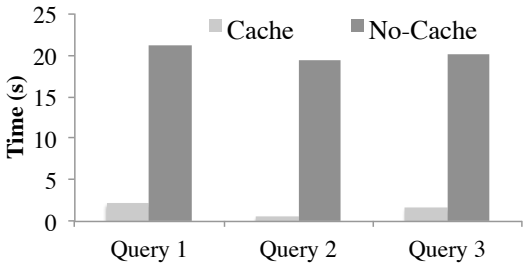



**Figure 6: Effect of caching input data**

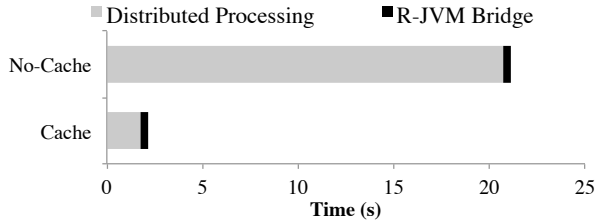



**Figure 8: Breakdown of time taken R-to-JVM bridge and distributed processing Spark for Query 1 from Figure 7**

```
# Train a GLM model
model <- glm(arrDelay ~ depDelay + distance,
             family = "gaussian", data = training)

# Print model summary
summary(model)

# Compute predictions using model
preds <- predict(model, training)
```

**Figure 9: Building Generalized Linear Models in SparkR**

results are in line with previous studies [34, 10] that measured the importance of caching in Spark. We would like to note that the benefits here come not only from using faster storage media, but also from avoiding CPU time in decompressing data and parsing CSV files. Finally, we can see that caching helps us achieve low latencies (less than 3 seconds) that make SparkR suitable for interactive query processing from the R shell.

## 4.3 Overhead of R-JVM binding

We next evaluate the overhead of using our socket-based R to JVM bridge discussed in Section 3.2.1. To do this we use query 1 from Figure 7 and run the query with both caching enabled and disabled on 64 cores. Using the performance metrics exported by Spark, we compute the time taken to run distributed computation and the time spent in the R-JVM bridge. In Figure 8, we see that the R-JVM bridge adds a constant overhead around 300 milliseconds irrespective of whether the data is cached or not. This overhead includes the time spent in serializing the query and in deserializing the results after it has been computed. For interactive query processing we find having an overhead of a few hundred milliseconds does not affect user experience. However, as the amount of data shipped between R and JVM increases we find that the overheads become more significant and we are exploring better serialization techniques in the R-JVM bridge to improve this.

# 5. ONGOING WORK

We are continuing work on SparkR in many areas to improve performance and enable more use cases. The two main areas we discuss here relate to large scale machine learning by integration with MLlib [23] and supporting partition aggregate workflows using user-defined functions (UDFs).

## 5.1 Large Scale Machine Learning

R includes support for a number of machine learning algorithms through the default `stats` package and other optional packages like `glmnet` [14], `cluster` [20] etc. The machine learning algorithms typically operate directly on data frames and use C or Fortran linkages for efficient implementations. One of the most widely used machine learning functions in R is the `glm` method that fits Generalized Linear Models. The `glm` method in R lets users specify the modeling of a response variable in a compact symbolic form using *formulas*. For example, the formula $y \sim a + b$ indicates the response $y$ is modeled linearly by variables $a$ and $b$. `glm` also lets users specify the loss function to use and can thus be to used to implement linear regression, logistic regression etc. The `glm` method returns a model trained using the input data and users typically use the `summary` function in R to print a number of statistics computed about the model.

To support large scale distributed machine learning in SparkR, we are working on integrating Spark's MLlib API with SparkR DataFrames. Our first focus is `glm` and to provide an intuitive interface for R users, we extend R's native methods for fitting and evaluating models as shown in Figure 9. We support a subset of the R formula operators in SparkR. These include the $+$ (inclusion), $-$ (exclusion), : (interactions) and intercept operators. SparkR implements the interpretation of R model formulas as an MLlib [23] feature transformer and this integrates with the ML Pipelines API [22]. This design also enables the same RFormula transformer to be used from Python, Scala and thereby enables an R-like succinct syntax for GLMs across different Spark APIs.

We are also working on implementing support for model summaries in SparkR to compute (a) minimum and maximum deviance residuals of the estimation (b) the coefficient values for the estimation (c) the estimated standard errors, $t$-values and $p$-values. Currently we have implemented these metrics for Gaussian GLMs trained using weighted least squares and we are working towards extending support for such metrics across different different families (Poisson, Gamma etc.) and link functions (logit, probit etc.) using iteratively re-weighted least squares (IRWLS).

## 5.2 User Defined Functions

To support the partition aggregate usage pattern discussed before, we are working on providing support for running user-defined functions (UDFs) in parallel. Spark supports UDFs written in Scala, Python and these APIs allow UDFs to run on each row of the input DataFrame. However, a number of R packages operate on local R data frames and it would be more user-friendly to support UDFs where R users can directly make use of these packages. In SparkR we plan to support UDFs that operate on each partition

of the distributed DataFrame and these functions will in turn return local R columnar data frames that will be then converted into the corresponding format in the JVM.

In addition to the above UDF-based API, we find that for some use cases like parameter tuning, the input dataset is small but there are a number of parameter values that need to be evaluated in parallel. To support such workflows we are working on a parallel execution API, where we take in a local list, a function to be executed and run the function for each element of the local list in one core in the cluster. Finally one of the main features that we aim to support with UDFs is closure capture or support for users to refer to external global variables inside their UDFs. We plan to implement this using R's support for reflection and one of the challenges here is to ensure that we only capture the necessary variables to avoid performance overheads.

### 5.3 Efficient Data Sharing

One of the main overheads when executing UDFs in SparkR is the time spent serializing input for the UDF from the JVM and then deserialzing it in R. This process is also repeated for the data output from the UDF and thus adds significant overhead to the execution time. Recent memory management improvements [3] have introduced support for off heap storage in Spark and we plan to investigate techniques to use off heap storage for sharing data efficiently between the JVM and R. One of the key challenges here is to develop a storage format that can be parsed easily in both languages. In addition to the serialization benefits, off heap data sharing can help us lower the memory overhead by reducing the number of data copies required.

## 6. RELATED WORK

A number of academic (Ricardo [13], RHIPE [17], RABID [19]) and commercial (RHadoop [5], BigR [33]) projects have looked at integrating R with Apache Hadoop. SparkR follows a similar approach but inherits the functionality [23] and performance [3] benefits of using Spark as the execution engine. The high level DataFrame API in SparkR is inspired by data frames in R [26], dplyr [31] and pandas [21]. Further, SparkR's data sources integration is similar to pluggable backends supported by dplyr. Unlike other data frame implementations, SparkR uses lazy evaluation and Spark's relational optimizer to improve performance for distributed computations. Finally, a number of projects like DistributedR [25], SciDB [24], SystemML [15] have looked at scaling array or matrix-based computations in R. In SparkR, we propose a high-level DataFrame API for structured data processing and integrate this with a distributed machine learning library to provide support for advanced analytics.

## 7. CONCLUSION

In summary, SparkR provides an R frontend to Apache Spark and allows users to run large scale data analysis using Spark's distributed computation engine. SparkR has been a part of the Apache Spark project since the 1.4.0 release and all of the functionality described in this work is open source. SparkR can be downloaded from `http://spark.apache.org`.

**Acknowledgments**: We would like to thank the anonymous reviewers and our shepherd Tyson Condie for their feedback. We would also like to thank Sun Rui, Yu Ishkawa, Chris Freeman, Dan Putler, Felix Cheung, Hao Lin, Antonio Piccolboni, Yanbo Liang, and all other contributors to the open source SparkR project. This research is supported in part by NSF CISE Expeditions Award CCF-1139158, DOE Award SN10040 DE-SC0012463, and DARPA XData Award FA8750-12-2-0331, and gifts from Amazon Web Services, Google, IBM, SAP, The Thomas and Stacey Siebel Foundation, Adatao, Adobe, Apple, Inc., Blue Goji, Bosch, Cisco, Cray, Cloudera, EMC, Ericsson, Facebook, Guavus, HP, Huawei, Informatica, Intel, Microsoft, NetApp, Pivotal, Samsung, Schlumberger, Splunk, Virdata and VMware.

## 8. REFERENCES

[1] 2015 data science salary survey. https://www.oreilly.com/ideas/2015-data-science-salary-survey.

[2] Apache Spark Project. http://spark.apache.org.

[3] Project Tungsten: Bringing Spark Closer to Bare Metal. https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html.

[4] Recent performance improvements in Apache Spark: SQL, Python, DataFrames, and More. https://goo.gl/RQS3ld.

[5] Rhadoop. http://projects.revolutionanalytics.com/rhadoop.

[6] Spark survey 2015. http://go.databricks.com/2015-spark-survey.

[7] Visual Analytics for Apache Spark and SparkR. http://goo.gl/zPje2i.

[8] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, et al. The Stratosphere platform for big data analytics. *VLDB Journal*, 23(6):939–964, 2014.

[9] M. Armbrust, T. Das, A. Davidson, A. Ghodsi, A. Or, J. Rosen, I. Stoica, P. Wendell, R. Xin, and M. Zaharia. Scaling spark in the real world: performance and usability. *Proceedings of the VLDB Endowment*, 8(12):1840–1843, 2015.

[10] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, et al. Spark SQL: Relational data processing in Spark. In *SIGMOD*, pages 1383–1394, 2015.

[11] S. M. Bache and H. Wickham. *magrittr: A Forward-Pipe Operator for R*, 2014. R package version 1.5.

[12] M. Barnett, B. Chandramouli, R. DeLine, S. Drucker, D. Fisher, J. Goldstein, P. Morrison, and J. Platt. Stat!: An interactive analytics environment for big data. In *SIGMOD 2013*, pages 1013–1016.

[13] S. Das, Y. Sismanis, K. S. Beyer, R. Gemulla, P. J. Haas, and J. McPherson. Ricardo: integrating R and Hadoop. In *SIGMOD 2010*, pages 987–998. ACM, 2010.

[14] J. Friedman, T. Hastie, and R. Tibshirani. Regularization paths for generalized linear models via coordinate descent. *Journal of Statistical Software*, 33(1):1–22, 2010.

[15] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, et al. SystemML: Declarative machine learning on MapReduce. In *ICDE*, pages 231–242. IEEE, 2011.

[16] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. Graphx: Graph processing in a distributed dataflow framework. In *OSDI 2014*, pages 599–613.

[17] S. Guha, R. Hafen, J. Rounds, J. Xia, J. Li, B. Xi, and W. S. Cleveland. Large complex data: Divide and Recombine (d&r) with RHIPE. *Stat*, 1(1):53–67, 2012.

[18] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi, J. Erickson, M. Grund, D. Hecht, M. Jacobs, et al. Impala: A modern, open-source SQL engine for Hadoop. In *CIDR 2015*.

[19] H. Lin, S. Yang, and S. Midkiff. RABID: A General

Distributed R Processing Framework Targeting Large Data-Set Problems. In *IEEE Big Data 2013*, pages 423–424, June 2013.

[20] M. Maechler, P. Rousseeuw, A. Struyf, M. Hubert, and K. Hornik. *cluster: Cluster Analysis Basics and Extensions*, 2015.

[21] W. McKinney. Data Structures for Statistical Computing in Python . In S. van der Walt and J. Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 51 – 56, 2010.

[22] X. Meng, J. Bradley, E. Sparks, and S. Venkataraman. ML Pipelines: A New High-Level API for MLlib. https://goo.gl/pluhq0, 2015.

[23] X. Meng, J. K. Bradley, B. Yavuz, E. R. Sparks, et al. MLlib: Machine Learning in Apache Spark. *CoRR*, abs/1505.06807, 2015.

[24] Paradigm4 and B. W. Lewis. *scidb: An R Interface to SciDB*, 2015. R package version 1.2-0.

[25] S. Prasad, A. Fard, V. Gupta, J. Martinez, J. LeFevre, V. Xu, M. Hsu, and I. Roy. Large-scale predictive analytics in vertica: Fast data transfer, distributed model creation, and in-database prediction. In *SIGMOD 2015*.

[26] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2015.

[27] S. Urbanek. *rJava: Low-Level R to Java Interface*, 2015. R package version 0.9-7.

[28] S. Venkataraman, E. Bodzsar, I. Roy, A. AuYoung, and R. S. Schreiber. Presto: Distributed Machine Learning and Graph Processing with Sparse Matrices. In *Eurosys 2013*, pages 197–210.

[29] J. Waldo. Remote Procedure Calls and Java Remote Method Invocation. *IEEE Concurrency*, 6(3):5–7, 1998.

[30] H. Wickham. *nycflights13: Data about flights departing NYC in 2013.*, 2014. R package version 0.1.

[31] H. Wickham and R. Francois. *dplyr: A Grammar of Data Manipulation*, 2015. R package version 0.4.3.

[32] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: SQL and rich analytics at scale. In *SIGMOD 2013*.

[33] L. Yejas, D. Oscar, W. Zhuang, and A. Pannu. Big R: Large-Scale Analytics on Hadoop Using R. In *IEEE Big Data 2014*, pages 570–577.

[34] M. Zaharia, M. Chowdhury, T. Das, A. Dave, et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. *NSDI*, 2012.

[35] Y. Zhang, W. Zhang, and J. Yang. I/O-efficient statistical computing with RIOT. In *ICDE 2010*, pages 1157–1160.

Journal of Machine Learning Research 17 (2016) 1-7 Submitted 5/15; Published 4/16

# MLlib: Machine Learning in Apache Spark

**Xiangrui Meng**† MENG@DATABRICKS.COM
*Databricks, 160 Spear Street, 13th Floor, San Francisco, CA 94105*

**Joseph Bradley** JOSEPH@DATABRICKS.COM
*Databricks, 160 Spear Street, 13th Floor, San Francisco, CA 94105*

**Burak Yavuz** BURAK@DATABRICKS.COM
*Databricks, 160 Spear Street, 13th Floor, San Francisco, CA 94105*

**Evan Sparks** SPARKS@CS.BERKELEY.EDU
*UC Berkeley, 465 Soda Hall, Berkeley, CA 94720*

**Shivaram Venkataraman** SHIVARAM@EECS.BERKELEY.EDU
*UC Berkeley, 465 Soda Hall, Berkeley, CA 94720*

**Davies Liu** DAVIES@DATABRICKS.COM
*Databricks, 160 Spear Street, 13th Floor, San Francisco, CA 94105*

**Jeremy Freeman** FREEMANJ11@JANELIA.HHMI.ORG
*HHMI Janelia Research Campus, 19805 Helix Dr, Ashburn, VA 20147*

**DB Tsai** DBT@NETFLIX.COM
*Netflix, 970 University Ave, Los Gatos, CA 95032*

**Manish Amde** MANISH@ORIGAMILOGIC.COM
*Origami Logic, 1134 Crane Street, Menlo Park, CA 94025*

**Sean Owen** SOWEN@CLOUDERA.COM
*Cloudera UK, 33 Creechurch Lane, London EC3A 5EB United Kingdom*

**Doris Xin** DORX0@ILLINOIS.EDU
*UIUC, 201 N Goodwin Ave, Urbana, IL 61801*

**Reynold Xin** RXIN@DATABRICKS.COM
*Databricks, 160 Spear Street, 13th Floor, San Francisco, CA 94105*

**Michael J. Franklin** FRANKLIN@CS.BERKELEY.EDU
*UC Berkeley, 465 Soda Hall, Berkeley, CA 94720*

**Reza Zadeh** REZAB@STANFORD.EDU
*Stanford and Databricks, 475 Via Ortega, Stanford, CA 94305*

**Matei Zaharia** MATEI@MIT.EDU
*MIT and Databricks, 160 Spear Street, 13th Floor, San Francisco, CA 94105*

**Ameet Talwalkar**† AMEET@CS.UCLA.EDU
*UCLA and Databricks, 4732 Boelter Hall, Los Angeles, CA 90095*

**Editor:** Mark Reid
† Corresponding authors.

## Abstract

Apache Spark is a popular open-source platform for large-scale data processing that is well-suited for iterative machine learning tasks. In this paper we present MLlib, Spark's open-source distributed machine learning library. MLlib provides efficient functionality for a wide range of learning settings and includes several underlying statistical, optimization, and linear algebra primitives. Shipped with Spark, MLlib supports several languages and provides a high-level API that leverages Spark's rich ecosystem to simplify the development of end-to-end machine learning pipelines. MLlib has experienced a rapid growth due to its vibrant open-source community of over 140 contributors, and includes extensive documentation to support further growth and to let users quickly get up to speed.

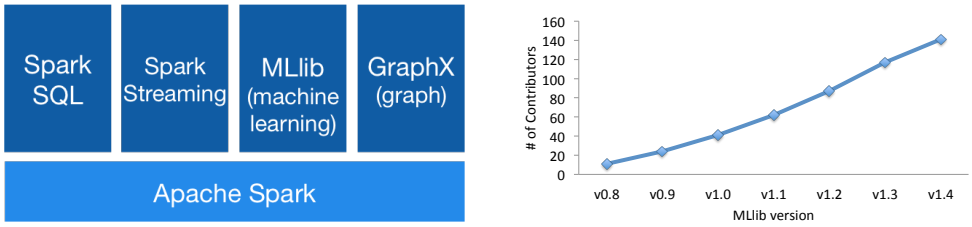
**Keywords:** scalable machine learning, distributed algorithms, apache spark

## 1. Introduction

Modern datasets are rapidly growing in size and complexity, and there is a pressing need to develop solutions to harness this wealth of data using statistical methods. Several 'next generation' data flow engines that generalize MapReduce (Dean and Ghemawat, 2004) have been developed for large-scale data processing, and building machine learning functionality on these engines is a problem of great interest. In particular, Apache Spark (Zaharia et al., 2012) has emerged as a widely used open-source engine. Spark is a fault-tolerant and general-purpose cluster computing system providing APIs in Java, Scala, Python, and R, along with an optimized engine that supports general execution graphs. Moreover, Spark is efficient at iterative computations and is thus well-suited for the development of large-scale machine learning applications.

In this work we present MLlib, Spark's distributed machine learning library, and the largest such library. The library targets large-scale learning settings that benefit from data-parallelism or model-parallelism to store and operate on data or models. MLlib consists of fast and scalable implementations of standard learning algorithms for common learning settings including classification, regression, collaborative filtering, clustering, and dimensionality reduction. It also provides a variety of underlying statistics, linear algebra, and optimization primitives. Written in Scala and using native (`C++` based) linear algebra libraries on each node, MLlib includes Java, Scala, and Python APIs, and is released as part of the Spark project under the Apache 2.0 license.

MLlib's tight integration with Spark results in several benefits. First, since Spark is designed with iterative computation in mind, it enables the development of efficient implementations of large-scale machine learning algorithms since they are typically iterative in nature. Improvements in low-level components of Spark often translate into performance gains in MLlib, without any direct changes to the library itself. Second, Spark's vibrant open-source community has led to rapid growth and adoption of MLlib, including contributions from over 140 people. Third, MLlib is one of several high-level libraries built on top of Spark, as shown in Figure 1(a). As part of Spark's rich ecosystem, and in part due to MLlib's `spark.ml` API for pipeline development, MLlib provides developers with a wide range of tools to simplify the development of machine learning pipelines in practice.




(a) (b)

Figure 1: (a) Apache Spark ecosytem. (b). Growth in MLlib contributors.

## 2. History and Growth

Spark was started in the UC Berkeley AMPLab and open-sourced in 2010. Spark is designed for efficient iterative computation and starting with early releases has been packaged with example machine learning algorithms. However, it lacked a suite of robust and scalable learning algorithms until the creation of MLlib. Development of MLlib began in 2012 as part of the MLbase project (Kraska et al., 2013), and MLlib was open-sourced in September 2013. From its inception, MLlib has been packaged with Spark, with the initial release of MLlib included in the Spark 0.8 release. As an Apache project, Spark (and consequently MLlib) is open-sourced under the Apache 2.0 license. Moreover, as of Spark version 1.0, Spark and MLlib are on a 3-month release cycle.

The original version of MLlib was developed at UC Berkeley by 11 contributors, and provided a limited set of standard machine learning methods. Since this original release, MLlib has experienced dramatic growth in terms of contributors. Less than two years later, as of the Spark 1.4 release, MLlib has over 140 contributors from over 50 organizations. Figure 1(b) demonstrates the growth in MLlib's open source community as a function of release version. The strength of this open-source community has spurred the development of a wide range of additional functionality.

## 3. Core Features

In this section we highlight the core features of MLlib; we refer the reader to the MLlib user guide for additional details (MLlib, 2015).

*Supported Methods and Utilities.* MLlib provides fast, distributed implementations of common learning algorithms, including (but not limited to): various linear models, naive Bayes, and ensembles of decision trees for classification and regression problems; alternating least squares with explicit and implicit feedback for collaborative filtering; and $k$-means clustering and principal component analysis for clustering and dimensionality reduction. The library also provides a number of low-level primitives and basic utilities for convex optimization, distributed linear algebra, statistical analysis, and feature extraction, and supports various I/O formats, including native support for LIBSVM format, data integration via Spark SQL (Armbrust et al., 2015), as well as PMML (Guazzelli et al., 2009) and MLlib's internal format for model export.

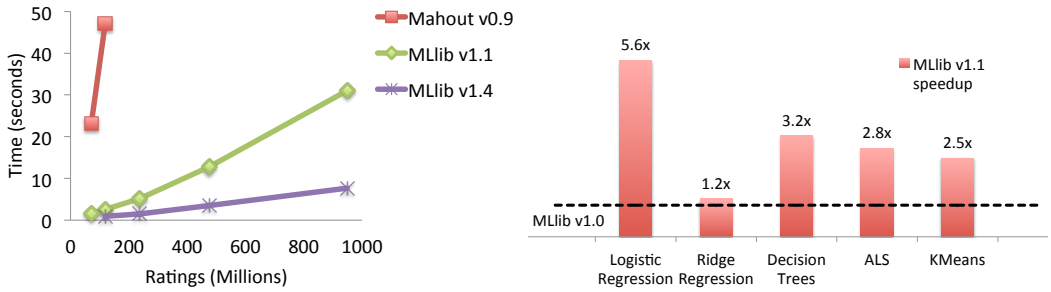
*Algorithmic Optimizations.* MLlib includes many optimizations to support efficient distributed learning and prediction. We highlight a few cases here. The ALS algorithm for

recommendation makes careful use of blocking to reduce JVM garbage collection overhead and to leverage higher-level linear algebra operations. Decision trees use many ideas from the PLANET project (Panda et al., 2009), such as data-dependent feature discretization to reduce communication costs, and tree ensembles parallelize learning both within trees and across trees. Generalized linear models are learned via optimization algorithms which parallelize gradient computation, using fast `C++`-based linear algebra libraries for worker computations. Many algorithms benefit from efficient communication primitives; in particular tree-structured aggregation prevents the driver from being a bottleneck, and Spark broadcast quickly distributes large models to workers.

*Pipeline API.* Practical machine learning pipelines often involve a sequence of data preprocessing, feature extraction, model fitting, and validation stages. Most machine learning libraries do not provide native support for the diverse set of functionality required for pipeline construction. Especially when dealing with large-scale datasets, the process of cobbling together an end-to-end pipeline is both labor-intensive and expensive in terms of network overhead. Leveraging Spark's rich ecosystem and inspired by previous work (Pedregosa et al., 2011; Buitinck et al., 2013; Sparks et al., 2013, 2015), MLlib includes a package aimed to address these concerns. This package, called `spark.ml`, simplifies the development and tuning of multi-stage learning pipelines by providing a uniform set of high-level APIs (Meng et al., 2015), including APIs that enable users to swap out a standard learning approach in place of their own specialized algorithms.

*Spark Integration.* MLlib benefits from the various components within the Spark ecosystem. At the lowest level, Spark core provides a general execution engine with over 80 operators for transforming data, e.g., for data cleaning and featurization. MLlib also leverages the other high-level libraries packaged with Spark. Spark SQL provides data integration functionality, SQL and structured data processing which can simplify data cleaning and preprocessing, and also supports the DataFrame abstraction which is fundamental to the `spark.ml` package. GraphX (Gonzalez et al., 2014) supports large-scale graph processing and provides a powerful API for implementing learning algorithms that can naturally be viewed as large, sparse graph problems, e.g., LDA (Blei et al., 2003; Bradley, 2015). Additionally, Spark Streaming (Zaharia et al., 2013) allows users to process live data streams and thus enables the development of online learning algorithms, as in Freeman (2015). Moreover, performance improvements in Spark core and these high-level libraries lead to corresponding improvements in MLlib.

*Documentation, Community, and Dependencies.* The MLlib user guide provides extensive documentation; it describes all supported methods and utilities and includes several code examples along with API docs for all supported languages (MLlib, 2015). The user guide also lists MLlib's code dependencies, which as of version 1.4 are the following open-source libraries: Breeze, netlib-java, and (in Python) NumPy (Breeze, 2015; Halliday, 2015; Braun, 2015; NumPy, 2015). Moreover, as part of the Spark ecosystem, MLlib has active community mailing lists, frequent meetup events, and JIRA issue tracking to facilitate open-source contributions (Community, 2015). To further encourage community contributions, Spark Packages (Packages, 2015) provides a community package index to track the growing number of open source packages and libraries that work with Spark. To date, several of the contributed packages consist of machine learning functionality that builds on MLlib.




(a) (b)

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

Finally, a massive open online course has been created to describe the core algorithmic concepts used to develop the distributed implementations within MLlib (Talwalkar, 2015).

## 4. Performance and Scalability

In this section we briefly demonstrate the speed, scalability, and continued improvements in MLlib over time. We first look at scalability by considering ALS, a commonly used collaborative filtering approach. For this benchmark, we worked with scaled copies of the Amazon Reviews dataset (McAuley and Leskovec, 2013), where we duplicated user information as necessary to increase the size of the data. We ran 5 iterations of MLlib's ALS for various scaled copies of the dataset, running on a 16 node EC2 cluster with m3.2xlarge instances using MLlib versions 1.1 and 1.4. For comparison purposes, we ran the same experiment using Apache Mahout version 0.9 (Mahout, 2014), which runs on Hadoop MapReduce. Benchmarking results, presented in Figure 2(a), demonstrate that MapReduce's scheduling overhead and lack of support for iterative computation substantially slow down its performance on moderately sized datasets. In contrast, MLlib exhibits excellent performance and scalability, and in fact can scale to much larger problems.

Next, we compare MLlib versions 1.0 and 1.1 to evaluate improvement over time. We measure the performance of common machine learning methods in MLlib, with all experiments performed on EC2 using m3.2xlarge instances with 16 worker nodes and synthetic datasets from the spark-perf package (`https://github.com/databricks/spark-perf`). The results are presented in Figure 2(b), and show a 3× speedup on average across all algorithms. These results are due to specific algorithmic improvements (as in the case of ALS and decision trees) as well as to general improvements to communication protocols in Spark and MLlib in version 1.1 (Yavuz and Meng, 2014).

## 5. Conclusion

MLlib is in active development, and the following link provides details on how to contribute: `https://cwiki.apache.org/confluence/display/SPARK/Contributing+to+Spark`. Moreover, we would like to acknowledge all MLlib contributors. The list of Spark contributors can be found at `https://github.com/apache/spark`, and the `git log` command can be used to identify MLlib contributors.

## References

Michael Armbrust, Reynold Xin, Cheng Lian, Yin Yuai, Davies Liu, Joseph Bradley, Xiangrui Meng, Tomer Kaftan, Michael Franklin, Ali Ghodsi, and Matei Zaharia. Spark SQL: Relational data processing in spark. In *ACM Special Interest Group on Management of Data*, 2015.

David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent dirichlet allocation. *Journal of Machine Learning Research*, 3, 2003.

Joseph Bradley. Topic modeling with LDA: MLlib meets GraphX. `https://databricks.com/?p=3135`, 2015.

Mikio Braun. jblas. `http://jblas.org/`, 2015.

Breeze. Breeze. `https://github.com/scalanlp/breeze/wiki`, 2015.

Lars Buitinck et al. API design for machine learning software: experiences from the scikit-learn project. *arXiv:1309.0238*, 2013.

Spark Community. Spark community. `https://spark.apache.org/community.html`, 2015.

Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *USENIX Symposium on Operating Systems Design and Implementation*, 2004.

Jeremy Freeman. Introducing streaming k-means in spark 1.2. `https://databricks.com/?p=2382`, 2015.

Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. In *Conference on Operating Systems Design and Implementation*, 2014.

Alex Guazzelli, Michael Zeller, Wen-Ching Lin, and Graham Williams. PMML: An open standard for sharing models. *The R Journal*, 1(1), 2009.

Sam Halliday. netlib-java. `https://github.com/fommil/netlib-java`, 2015.

Tim Kraska, Ameet Talwalkar, John Duchi, Rean Griffith, Michael Franklin, and Michael Jordan. MLbase: A Distributed Machine-learning System. In *Conference on Innovative Data Systems Research*, 2013.

Mahout. Apache mahout. `http://mahout.apache.org/`, 2014.

Julian McAuley and Jure Leskovec. Hidden factors and hidden topics: Understanding rating dimensions with review text. In *ACM Recommender Systems Conference*, 2013.

Xiangrui Meng, Joseph Bradley, Evan Sparks, and Shivaram Venkataraman. Ml pipelines: A new high-level api for MLlib. `https://databricks.com/?p=2473`, 2015.

MLlib. MLlib user guide. `https://spark.apache.org/docs/latest/mllib-guide.html`, 2015.

NumPy. Numpy. http://www.numpy.org/, 2015.

Spark Packages. Spark packages. https://spark-packages.org, 2015.

Biswanath Panda, Joshua S. Herbach, Sugato Basu, and Roberto J. Bayardo. Planet: Massively parallel learning of tree ensembles with mapreduce. In *International Conference on Very Large Databases*, 2009.

Fabian Pedregosa, Gael Varoquaux, Alexandre Gramfort, et al. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 2011.

Evan R. Sparks, Ameet Talwalkar, Virginia Smith, Jey Kottalam, Xinghao Pan, Joseph E. Gonzalez, Michael J. Franklin, Michael I. Jordan, and Tim Kraska. MLI: An API for Distributed Machine Learning. In *International Conference on Data Mining*, 2013.

Evan R Sparks, Ameet Talwalkar, Daniel Haas, Michael J. Franklin, Michael I. Jordan, and Tim Kraska. Automating model search for large scale machine learning. *Symposium on Cloud Computing*, 2015.

Ameet Talwalkar. BerkeleyX CS190-1x: Scalable machine learning. https://www.edx.org/course/scalable-machine-learning-uc-berkeleyx-cs190-1x, 2015.

Burak Yavuz and Xiangrui Meng. Spark 1.1: MLlib performance improvements. https://databricks.com/?p=1393, 2014.

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *USENIX Symposium on Networked Systems Design and Implementation*, 2012.

Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Symposium on Operating Systems Principles*, 2013.