Title
An Architecture for and Fast and General Data Processing on Large Clusters

Author
Zaharia, Matei Alexandru

Publication Date
2013-01-01

Peer reviewed|Thesis/dissertation

An Architecture for Fast and General Data Processing on Large Clusters

by

Matei Alexandru Zaharia

A dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy
in

Computer Science

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:

Professor Scott Shenker, Chair
Professor Ion Stoica
Professor Alexandre Bayen
Professor Joshua Bloom

Fall 2013

An Architecture for Fast and General Data Processing on Large Clusters

Matei Alexandru Zaharia

Abstract

An Architecture for Fast and General Data Processing on Large Clusters

by

Matei Alexandru Zaharia

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Scott Shenker, Chair

The past few years have seen a major change in computing systems, as growing data volumes and stalling processor speeds require more and more applications to scale out to distributed systems. Today, a myriad data sources, from the Internet to business operations to scientific instruments, produce large and valuable data streams. However, the processing capabilities of single machines have not kept up with the size of data, making it harder and harder to put to use. As a result, a growing number of organizations—not just web companies, but traditional enterprises and research labs—need to scale out their most important computations to clusters of hundreds of machines.

At the same time, the speed and sophistication required of data processing have grown. In addition to simple queries, complex algorithms like machine learning and graph analysis are becoming common in many domains. And in addition to batch processing, streaming analysis of new real-time data sources is required to let organizations take timely action. Future computing platforms will need to not only scale out traditional workloads, but support these new applications as well.

This dissertation proposes an architecture for cluster computing systems that can tackle emerging data processing workloads while coping with larger and larger scales. Whereas early cluster computing systems, like MapReduce, handled batch processing, our architecture also enables streaming and interactive queries, while keeping the scalability and fault tolerance of previous systems. And whereas most deployed systems only support simple one-pass computations (*e.g.,* aggregation or SQL queries), ours also extends to the multi-pass algorithms required for more complex analytics (*e.g.,* iterative algorithms for machine learning). Finally, unlike the specialized systems proposed for some of these workloads, our architecture allows these computations to be *combined*, enabling rich new applications that intermix, for example, streaming and batch processing, or SQL and complex analytics.

We achieve these results through a simple extension to MapReduce that adds primitives for data sharing, called Resilient Distributed Datasets (RDDs). We show that this is enough to efficiently capture a wide range of workloads. We implement RDDs in the open source Spark system, which we evaluate using both synthetic

benchmarks and real user applications. Spark matches or exceeds the performance of specialized systems in many application domains, while offering stronger fault tolerance guarantees and allowing these workloads to be combined. We explore the generality of RDDs from both a theoretical modeling perspective and a practical perspective to see why this extension can capture a wide range of previously disparate workloads.

To my family

Contents

List of Figures

List of Tables

Acknowledgements

I am thankful to my advisors, Scott Shenker and Ion Stoica, who guided me tirelessly throughout my PhD. They are both brilliant researchers who were always able to push ideas one level further, provide the input needed to finish a task, and share their experience about the research process. I was especially fortunate to work with both of them and benefit from both their perspectives.

The work in this dissertation is the result of collaboration with many other people. Chapter 2 was joint work with Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Mike Franklin, Scott Shenker and Ion Stoica [118]. Chapter 3 highlights parts of the Shark project developed with Reynold Xin, Josh Rosen, Mike Franklin, Scott Shenker and Ion Stoica [113]. Chapter 4 was joint work with Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker and Ion Stoica [119]. More broadly, numerous people in the AMPLab and in Spark-related projects like GraphX [112] and MLI [98] contributed to the development and refinement of the ideas in this dissertation.

Beyond direct collaborators on the projects here, many other people contributed to my graduate work and made Berkeley an unforgettable experience. Over numerous cups of chai and Ceylon gold, Ali Ghodsi provided fantastic advice and ideas on both the research and open source sides. Ben Hindman, Andy Konwinski and Kurtis Heimerl were great fun to hang out with and great collaborators on some neat ideas. Taylor Sittler got me, and then much of the AMPLab, very excited about biology, which led to one of the most fun groups I've participated in under AMP-X, with Bill Bolosky, Ravi Pandya, Kristal Curtis, Dave Patterson and others. In other projects I was also fortunate to work with Vern Paxson, Dawn Song, Anthony Joseph, Randy Katz and Armando Fox, and to learn from their insights. Finally, the AMPLab and RAD Lab were amazing communities, both among the members at Berkeley and the industrial contacts who gave us constant feedback.

I was also very fortunate to work early on with the open source big data community. Dhruba Borthakur and Joydeep Sen Sarma got me started contributing to Hadoop at Facebook, while Eric Baldeschwieler, Owen O'Malley, Arun Murthy, Sanjay Radia and Eli Collins participated in many discussions to make our research ideas real. Once we started the Spark project, I was and continue to be overwhelmed by the talent and enthusiasm of the individuals who contribute to it. The Spark and Shark contributors, of which there are now over 130, deserve as much credit as anyone for making these projects real. The users, of course, have also contributed tremendously, by continuing to give great feedback and relentlessly pushing these systems in new directions that influence the core design. Of these, I am especially thankful to the project's earliest users, including Lester Mackey, Tim Hunter, Dilip Joseph, Jibin Zhan, Erich Nachbar, and Karthik Thiyagarajan.

Last but not least, I want to thank my family and friends for their unwavering support throughout my PhD.

Chapter 1

Introduction

The past few years have seen a major change in computing systems, as growing data volumes require more and more applications to scale out to large clusters. In both commercial and scientific fields, new data sources and instruments (*e.g.*, gene sequencers, RFID, and the web) are producing rapidly increasing amounts of information. Unfortunately, the processing and I/O capabilities of single machines have not kept up with this growth. As a result, more and more organizations have to scale out their computations across clusters.

The cluster environment comes with several challenges for programmability. The first one is parallelism: this setting requires rewriting applications in a parallel fashion, with programming models that can capture a wide range of computations. However, unlike in other parallel platforms, a second challenge in clusters is faults: both outright node failures and *stragglers* (slow nodes) become common at scale, and can greatly impact the performance of an application. Finally, clusters are typically shared between multiple users, requiring runtimes that can dynamically scale computations up and down and exacerbating the possibility of interference.

As a result, a wide range of new programming models have been designed for clusters. At first, Google's MapReduce [36] presented a simple and general model for batch processing that automatically handles faults. However, MapReduce was found poorly suited for other types of workloads, leading to a wide range of *specialized* models that differed significantly from MapReduce. For example, at Google, Pregel [72] offers a bulk-synchronous parallel (BSP) model for iterative graph algorithms; F1 [95] runs fast, but non-fault-tolerant, SQL queries; and MillWheel [2] supports continuous stream processing. Outside Google, systems like Storm [14], Impala [60], Piccolo [86] and GraphLab [71] offer similar models. With new models continuing to be implemented every year, it seems that cluster computing is bound to require an array of point solutions for different tasks.

This dissertation argues that, instead, we can design a *unified* programming abstraction that not only captures these disparate workloads, but enables new applications as well. In particular, we show that a simple extension to MapReduce called Resilient Distributed Datasets (RDDs), which just adds efficient data sharing

primitives, greatly increases its generality. The resulting architecture has several key advantages over current systems:

1. It supports batch, interactive, iterative and streaming computations in the same runtime, enabling rich applications that *combine* these modes, and offering significantly higher performance for these combined applications than disparate systems.

2. It provides fault and straggler tolerance across these computation modes at a very low cost. Indeed, in several areas (streaming and SQL), our RDD-based approach yields new system designs with significantly stronger fault tolerance properties than the status quo.

3. It achieves performance that is often 100× better than MapReduce, and comparable with specialized systems in individual application domains.

4. It is highly amenable to multitenancy, allowing applications to scale up and down elastically and share resources in a responsive fashion.

We implement the RDD architecture in a stack of open source systems including Apache Spark as the common foundation; Shark for SQL processing; and Spark Streaming for distributed stream processing (Figure 1.1). We evaluate these systems using both real user applications and traditional benchmarks. Our implementations provide state-of-the-art performance for both traditional and new data analysis workloads, as well as the first platform to let users compose these workloads.

From a longer-term perspective, we also discuss general techniques for implementing various data processing tasks on RDDs, as well as *why* the RDD model turns out to be so general. As cluster applications grow in complexity, we believe that the kind of unified processing architecture provided by RDDs will be increasingly important for both performance and ease of use.

Thesis Statement: *A common execution model based on resilient distributed datasets can efficiently support diverse distributed computations.*

In the remainder of this chapter, we survey some of the motivation for the RDD architecture, then highlight our key results.

1.1 Problems with Specialized Systems

Today's cluster computing systems are increasingly specialized for particular application domains. While models like MapReduce and Dryad [36, 61] aimed to capture fairly general computations, researchers and practitioners have since developed more and more specialized systems for new application domains. Recent

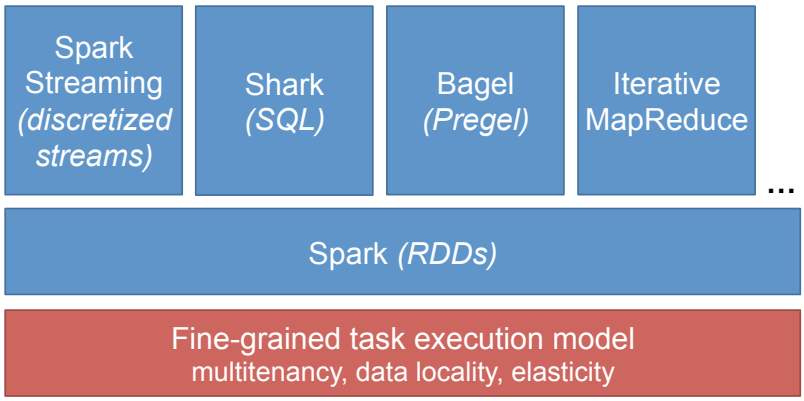


Figure 1.1. Computing stack implemented in this dissertation. Using the Spark implementation of RDDs, we build other processing models, such as streaming, SQL and graph computations, all of which can be intermixed within Spark programs. RDDs themselves execute applications as a series of fine-grained tasks, enabling efficient resource sharing.

examples include Dremel and Impala for interactive SQL queries [75, 60], Pregel for graph processing [72], GraphLab for machine learning [71], and others.

Although specialized systems seem like a natural way to scope down the challenging problems in the distributed environment, they also come with several drawbacks:

1. **Work duplication:** Many specialized systems still need to solve the same underlying problems, such as work distribution and fault tolerance. For example, a distributed SQL engine or machine learning engine both need to perform parallel aggregations. With separate systems, these problems need to be addressed anew for each domain.

2. **Composition:** It is both expensive and unwieldy to compose computations in different systems. For "big data" applications in particular, intermediate datasets are large and expensive to move. Current environments require exporting data to a replicated, stable storage system to share it between computing engines, which is often many times more expensive than the actual computations. Therefore, pipelines composed of multiple systems are often highly inefficient compared to a unified system.

3. **Limited scope:** If an application does not fit the programming model of a specialized system, the user must either modify it to make it fit within current systems, or else write a new runtime system for it.

4. **Resource sharing:** Dynamically sharing resources between computing engines is difficult because most engines assume they own the same set of machines for the duration of the application.

5. **Management and administration:** Separate systems require significantly more work to manage and deploy than a single one. Even for users, they require learning multiple APIs and execution models.

Because of these limitations, a unified abstraction for cluster computing would have significant benefits in not only usability but also performance, especially for complex applications and multi-user settings.

1.2 Resilient Distributed Datasets (RDDs)

To address this problem, we introduce a new abstraction, *resilient distributed datasets* (RDDs), that forms a simple extension to the MapReduce model. The insight behind RDDs is that although the workloads that MapReduce was unsuited for (*e.g.,* iterative, interactive and streaming queries) seem at first very different, they all require a common feature that MapReduce lacks: efficient *data sharing* across parallel computation stages. With an efficient data sharing abstraction and MapReduce-like operators, all of these workloads can be expressed efficiently, capturing the key optimizations in current specialized systems. RDDs offer such an abstraction for a broad set of parallel computations, in a manner that is both efficient and fault-tolerant.

In particular, previous fault-tolerant processing models for clusters, such as MapReduce and Dryad, structured computations as a directed acyclic graph (DAG) of tasks. This allowed them to efficiently replay just part of the DAG for fault recovery. Between separate computations, however (*e.g.,* between steps of an iterative algorithm), these models provided no storage abstraction other than replicated file systems, which add significant costs due to data replication across the network. RDDs are a fault-tolerant distributed memory abstraction that *avoids* replication. Instead, each RDD remembers the graph of operations used to build it, similarly to batch computing models, and can efficiently *recompute* data lost on failure. As long as the operations that create RDDs are relatively coarse-grained, i.e., a single operation applies to many data elements, this technique is much more efficient than replicating the data over the network. RDDs work well for a wide range of today's data-parallel algorithms and programming models, all of which apply each operation to many items.

While it may seem surprising that just adding data sharing greatly increases the generality of MapReduce, we explore from several perspectives why this is so. First, from an expressiveness perspective, we show that RDDs can emulate *any* distributed system, and will do so efficiently as long as the system tolerates some network latency. This is because, once augmented with fast data sharing, MapReduce can emulate the Bulk Synchronous Parallel (BSP) [108] model of parallel computing, with the main drawback being the latency of each MapReduce step. Empirically, in our Spark system, this can be as low as 50–100 ms. Second, from a systems perspective, RDDs, unlike plain MapReduce, give applications enough control

to optimize the bottleneck resources in most cluster computations (specifically, network and storage I/O). Because these resources often dominate execution time, just controlling these (*e.g.*, through control over data placement) is often enough to match the performance of specialized systems, which are bound by the same resources.

This exploration aside, we also show empirically that we can implement many of the specialized models in use today, as well as new programming models, using RDDs. Our implementations match the performance of specialized systems while offering rich fault tolerance properties and enabling composition.

1.3 Models Implemented on RDDs

We used RDDs to implement both several existing cluster programming models and new applications not supported by previous models. In some of these models, RDDs only match the performance of previous systems, but in others, they also add new properties, such as fault tolerance, straggler tolerance and elasticity, that current systems lack. We discuss four classes of models.

Iterative Algorithms One of the most common workloads for which specialized systems have been developed recently is iterative algorithms, such as those used in graph processing, numerical optimization, and machine learning. RDDs can capture a wide variety of such models, including Pregel [72], iterative MapReduce models like HaLoop and Twister [22, 37], and a deterministic version of the GraphLab and PowerGraph models [71, 48].

Relational Queries One of the first demands on MapReduce clusters was to run SQL queries, both as long-running, many-hour batch jobs and as interactive queries. This spurred the development of many systems that apply parallel database designs in commodity clusters [95, 60, 75]. While it was thought that MapReduce had major inherent disadvantages over parallel databases for interactive queries [84], for instance, due to its fault tolerance model, we showed that state-of-the-art performance can be achieved within the RDD model, by implementing many common database engine features (*e.g.*, column-oriented processing) within RDD operations. The resulting system, Shark [113], offers full fault tolerance, allowing it to scale to both short and long queries, and the ability to call into complex analytics functions (*e.g.*, machine learning) built on RDDs as well.

MapReduce By offering a superset of MapReduce, RDDs can also efficiently run MapReduce applications, as well as more general DAG-based applications such as DryadLINQ [115].

Stream Processing As our largest departure from specialized systems, we also use RDDs to implement stream processing. Stream processing has been studied in both the database and systems communities before, but implementing it at scale is challenging. Current models do not handle the problem of stragglers, which will frequently occur in large clusters, and offer limited means for fault recovery, requiring either expensive replication or long recovery times. In particular, current systems are based on a *continuous operator* model, where long-lived stateful operators process each record as it arrives. To recover a lost node, they need to either keep two copies of each operator, or replay upstream data through a costly serial process.

We propose a new model, *discretized streams* (D-Streams), that overcomes this problem. Instead of using long-lived stateful processes, D-streams execute streaming computations as a sequence of short, deterministic batch computations, storing state in RDDs in-between. The D-Stream model allows fast fault recovery *without* replication by parallelizing the recovery along the dependency graph of the RDDs involved. In addition, it supports straggler mitigation through speculative execution [36], i.e., running speculative backup copies of slow tasks. While D-streams do add some delay by running work as discrete jobs, we show that they can be implemented with sub-second latency while reaching similar per-node performance to previous systems and scaling linearly to 100 nodes. Their strong recovery properties make them one of the first stream processing models to handle the idiosyncrasies of large clusters, and their basis in RDDs also lets applications combine them with batch and interactive queries in powerful ways.

Combining these Models By putting these models together, RDDs also enable new applications that are hard to express with current systems. For example, many streaming applications also need to join information with historical data; with RDDs, one can combine batch and stream processing in the same program, obtaining data sharing and fault recovery across both models. Similarly, operators of streaming applications often need to run ad-hoc queries on stream state; the RDDs in D-Streams can be queried just like static data. We illustrate these use cases with real applications for online machine learning (Section 4.6.3) and video analytics (Section 4.6.3). More generally, even batch applications often need to combine processing types: for example, an application may extract a dataset using SQL, train a machine learning model on it, and then query the model. These combined workflows are inefficient with current systems because the distributed filesystem I/O required to share data between systems often dominates the computation. With an RDD-based system, the computations can run back-to-back in the same engine without external I/O.

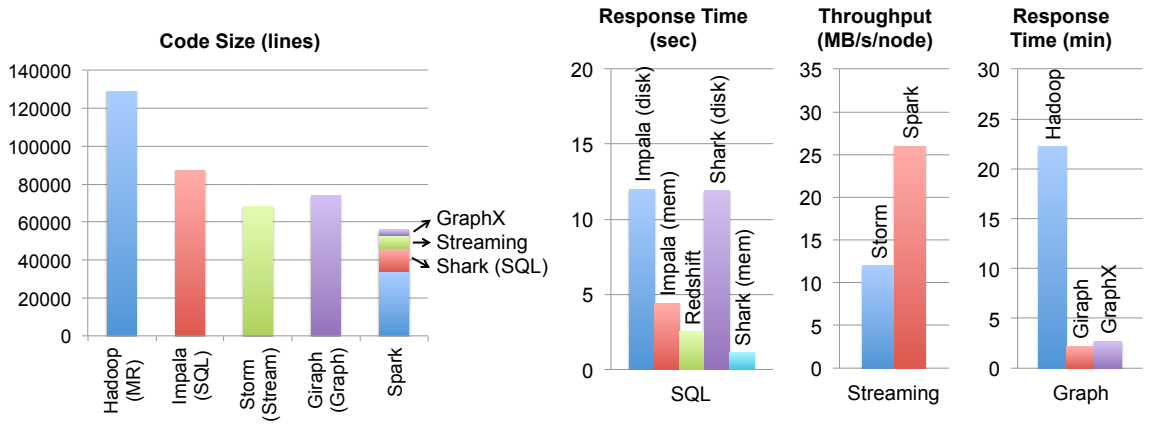


Figure 1.2. Comparing code size and performance between the Spark stack and specialized systems. Spark's code size is similar to specialized engines, while implementations of these models on Spark are significantly smaller. Nonetheless, performance in selected applications is comparable.

1.4 Summary of Results

We have implemented RDDs in an open source system called Spark, which is now hosted at the Apache Incubator and used in several commercial deployments [13]. Despite the generality of RDDs, Spark is relatively small: 34,000 lines of code in the (admittedly high-level) Scala language, putting it in the same range as specialized cluster computing systems. More importantly, specialized models built over Spark are much smaller than their standalone counterparts: we implemented Pregel and iterative MapReduce in a few hundred lines of code, discretized streams in 8000, and Shark, a SQL system that runs queries from the Apache Hive frontend on Spark, in 12,000. These systems are order of magnitude smaller than standalone implementations and support rich means to intermix models, but still generally match specialized systems in performance. As a short summary of results, Figure 1.2 compares the size and performance of Spark and three systems built on it (Shark, Spark Streaming, and GraphX [113, 119, 112]) against popular specialized systems (Impala and the Amazon Redshift DBMS for SQL; Storm for streaming; and Giraph for graph processing [60, 5, 14, 10]).

Beyond these practical results, we also cover general techniques for implementing sophisticated processing functions with RDDs, and discuss *why* the RDD model was so general. In particular, as sketched in Section 1.2, we show that the RDD model can emulate *any* distributed system and does so in a more efficient manner than MapReduce; and from a practical point of view, the RDD interface gives applications enough control over the bottleneck resources in a cluster to match specialized systems, while still enabling automatic fault recovery and efficient composition.

1.5 Dissertation Plan

This dissertation is organized as follows. Chapter 2 introduces the RDD abstraction and its application to capture several simple programming models. Chapter 3 covers techniques for implementing more advanced storage and processing models on RDDs, focusing on the Shark SQL system. Chapter 4 applies RDDs to develop discretized streams, a new model for stream processing. Chapter 5 then discusses why the RDD model was so general across these applications, as well as limitations and extensions. Finally, we conclude and discuss possible areas for future work in Chapter 6.

Chapter 2

Resilient Distributed Datasets

2.1 Introduction

In this chapter, we present the Resilient Distributed Dataset (RDD) abstraction, on which the rest of the rest of the dissertation builds a general-purpose cluster computing stack. RDDs extend the data flow programming model introduced by MapReduce [36] and Dryad [61], which is the most widely used model for large-scale data analysis today. Data flow systems were successful because they let users write computations using high-level operators, without worrying about work distribution and fault tolerance. As cluster workloads diversified, however, data flow systems were found inefficient for many imporant applications, including iterative algorithms, interactive queries, and stream processing. This led to the development of a wide range of specialized frameworks for these applications [72, 22, 71, 95, 60, 14, 2].

Our work starts from the observation that many of the applications that data flow models were not suited for have a characteristic in common: they all require efficient *data sharing* across computations. For example, iterative algorithms, such as PageRank, K-means clustering, or logistic regression, need to make multiple passes over the same dataset; interactive data mining often requires running multiple ad-hoc queries on the same subset of the data; and streaming applications need to maintain and share state across time. Unfortunately, although data flow frameworks offer numerous computational operators, they lack efficient primitives for data sharing. In these frameworks, the only way to share 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.

Indeed, examining the specialized frameworks built for these new applications, we see that many of them optimize data sharing. For example, Pregel [72] is a system for iterative graph computations that keeps intermediate state in memory, while HaLoop [22] is an iterative MapReduce system that can keep data partitioned in an efficient way across steps. Unfortunately, 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.

Instead, we propose a new abstraction called resilient distributed datasets (RDDs) that gives users direct control of data sharing. RDDs are fault-tolerant, parallel data structures that let users explicitly store data on disk or in memory, control its partitioning, and manipulate it using a rich set of operators. They offer a simple and efficient programming interface that can capture both current specialized models and new applications.

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 [79], key-value stores [81], databases, and Piccolo [86], 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 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 [115] in the Scala programming language [92]. In addition, Spark can be used interactively to query big datasets from the Scala interpreter. We believe

[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 Section 2.5.5.

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 80× 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. Finally, 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 chapter begins with an overview of RDDs (Section 2.2) and Spark (Section 2.3). We then discuss the internal representation of RDDs (Section 2.4), our implementation (Section 2.5), and experimental results (Section 2.6). Finally, we discuss how RDDs capture several existing programming models (Section 2.7), survey related work (Section 2.8), and conclude.

2.2 RDD Abstraction

This section provides an overview of RDDs. We first define RDDs (§2.2.1) and introduce their programming interface in Spark (§2.2.2). We then compare RDDs with finer-grained shared memory abstractions (§2.2.3). Finally, we discuss limitations of the RDD model (§2.2.4).

2.2.1 Definition

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

[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.

placement optimizations, such as ensuring that two datasets that will be joined together are hash-partitioned in the same way.

2.2.2 Spark Programming Interface

Spark exposes RDDs through a language-integrated API similar to DryadLINQ [115] and FlumeJava [25], 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.

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 [11]) 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:

```
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()
```

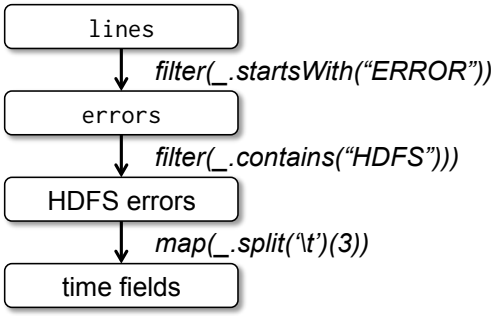


Figure 2.1. Lineage graph for the third query in our example. Boxes represent RDDs and arrows represent transformations.

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 2.1. In this query, we started with `errors`, the result of a filter on `lines`, and applied a further filter and map before running a *collect*. The Spark scheduler will pipeline the latter two transformations and send a set of tasks to compute them to the nodes holding the cached partitions of `errors`. In addition, if a partition of `errors` is lost, Spark rebuilds it by applying a filter on only the corresponding partition of `lines`.

2.2.3 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 2.1. 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 [79], but also other systems where applications make fine-grained writes to shared state, including Piccolo [86], which provides a shared DHT, and distributed

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 2.1. Comparison of RDDs with distributed shared memory.

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 [36]. 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 schedule 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.

[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 2.5.5. 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.

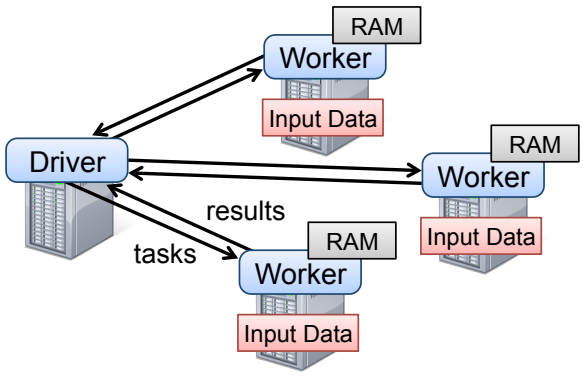


Figure 2.2. Spark runtime. The user's driver program launches multiple workers, which read data blocks from a distributed file system and can persist computed RDD partitions in memory.

2.2.4 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 [81], Percolator [85] and Piccolo [86]. Our goal is to provide an efficient programming model for batch analytics and leave these asynchronous applications to specialized systems. Nonetheless, Chapter 5 covers a few possible approaches for integrating these types of applications with the RDD model, such as batching updates.

2.3 Spark Programming Interface

Spark offers the RDD abstraction through a language-integrated API similar to DryadLINQ [115] in Scala [92], 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 2.2. 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 2.2.2, users provide argu-

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

Table 2.2. Some transformations and actions available on RDDs in Spark. Seq[T] denotes a sequence of elements of type T.

ments 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 – *e.g.*, RDD[Int] is an RDD of integers. However, most of our examples omit types due to Scala's 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 [118]. We also needed more work to make Spark usable from the Scala interpreter, as we shall discuss in Section 2.5.3. Nonetheless, we did *not* have to modify the Scala compiler.

2.3.1 RDD Operations in Spark

Table 2.2 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 without immediately computing it, 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.

2.3.2 Example Applications

We complement the data mining example in §2.2.2 with two iterative applications: logistic regression and PageRank. The latter also shows how control of RDDs partitioning can help performance.

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.

[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.

As an example, the following program implements logistic regression [53], a common classification algorithm that searches for a hyperplane w that best separates two sets of points (*e.g.*, spam and non-spam emails). The algorithm uses gradient descent: it starts w at a random value, and on each iteration, it sums a function of w over the data to move w in a direction that improves it.

```
val points = spark.textFile(...).map(parsePoint).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 2.6.1.

PageRank

A more complex pattern of data sharing occurs in PageRank [21]. 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, N is the total number of documents, and α is a tuning parameter. We can write PageRank in Spark as follows:

```
// Load graph as an RDD of (URL, outlinks) pairs
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 contributions sent by each page
  val contribs = links.join(ranks).flatMap {
    case (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)
}
```

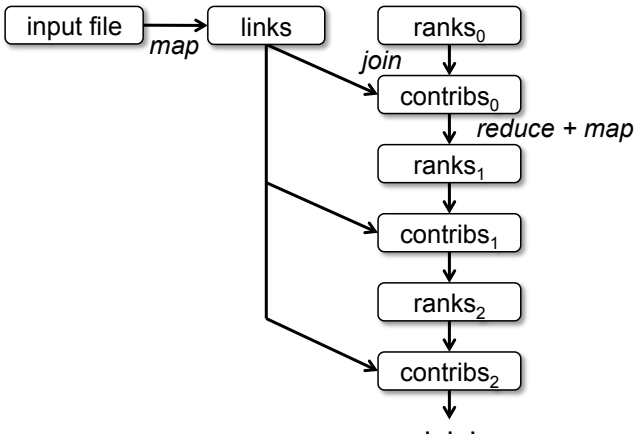


Figure 2.3. Lineage graph for datasets in PageRank.

This program leads to the RDD lineage graph in Figure 2.3. 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 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 [66]. 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 usually 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 full 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.

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

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 2.3. Interface used to represent RDDs in Spark.

2.4 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 2.2), 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 2.3.

The most interesting question in designing this interface is how to represent dependencies between RDDs. We found it both sufficient and useful to classify dependencies into two types: *narrow* dependencies, where each partition of the 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 2.4 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 *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 one 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).

2.5 Implementation

We have implemented Spark in about 34,000 lines of Scala. The system runs over diverse cluster managers including Apache Mesos [56], Hadoop YARN [109], Amazon EC2 [4], and its own built-in cluster manager. Each Spark program runs as

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

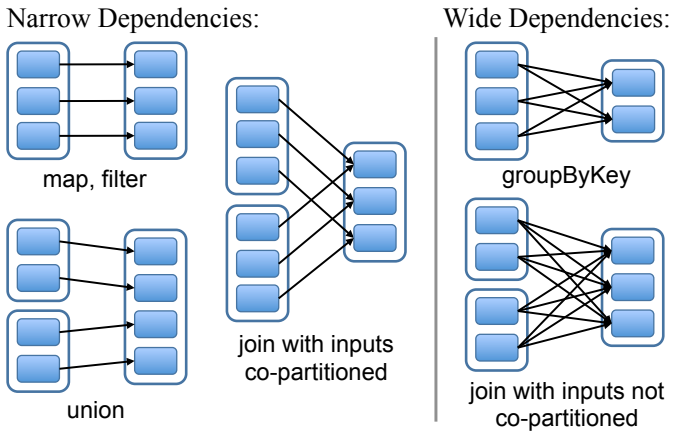


Figure 2.4. Examples of narrow and wide dependencies. Each box is an RDD, with partitions shown as shaded rectangles.

a separate application on the cluster, with its own driver (master) and workers, and resource sharing between these applications is handled by the cluster manager.

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 (§2.5.1), multi-user support (§2.5.2), our Spark interpreter allowing interactive use (§2.5.3), memory management (§2.5.4), and support for checkpointing (§2.5.5).

2.5.1 Job Scheduling

Spark's scheduler uses our representation of RDDs, described in Section 2.4.

Overall, our scheduler is similar to Dryad's [61], 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 2.5. 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 [117]. 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 in-

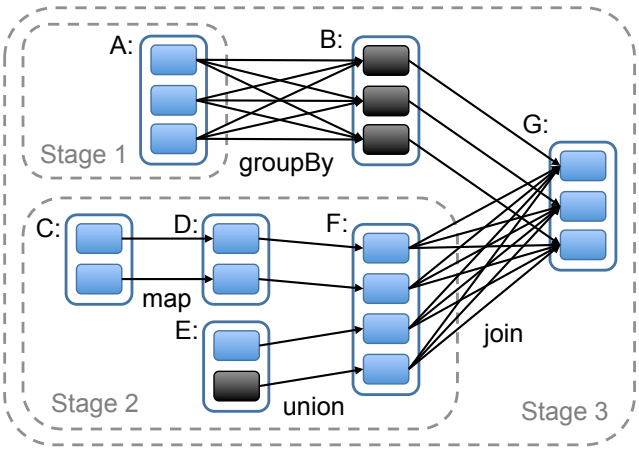


Figure 2.5. Example of how Spark computes job stages. Boxes with solid outlines are RDDs. Partitions are shaded rectangles, in black if they are already in memory. To run an action on RDD G, we build build stages at wide dependencies and pipeline narrow transformations inside each stage. In this case, stage 1's output RDD is already in RAM, so we run stage 2 and then 3.

termediate 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.

If a task runs slowly (*i.e.,* is a straggler), we also launch a speculative backup copy on another node, similar to MapReduce [36], and take the output of whichever copy finishes first.

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.

2.5.2 Multitenancy

Because the RDD model breaks computations into independent, fine-grained tasks, it allows for a variety of resource sharing algorithms in multi-user clusters. In particular, each RDD application can scale up and down dynamically during its execution, and applications can take turns accessing each machine, or can quicky be preempted by higher-priority ones. Most tasks in Spark applications are between 50 ms and a few seconds, allowing for highly responsive sharing.

While we do not focus on multi-user sharing algorithms in this dissertation, we list the available ones to give reader a sense of the resource sharing options possible.

- Within each application, Spark allows multiple threads to submit jobs concurrently, and allocates resources between these using a hierarchical fair scheduler similar to the Hadoop Fair Scheduler [117]. This feature is mostly used to build multi-user applications over the same in-memory data—for example, the Shark SQL engine has a server mode where multiple users can run queries concurrently. Fair sharing ensures that jobs are isolated from each other and short jobs retrun quickly even if long ones are otherwise filling the resources of the cluster.

- Spark's fair scheduler also uses delay scheduling [117] to give jobs high data locality while retaining fairness, by letting them take turns to access data on each machine. In almost all of the experiments in this chapter, memory locality was 100%. Spark supports multiple levels of locality, including memory, disk, and rack, to capture the varying costs of data access throughout a cluster.

- Because tasks are independent, the scheduler also supports job cancellation to make room for high-priority jobs [62].

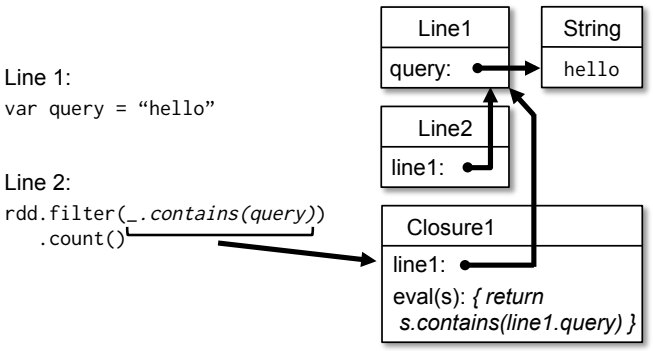
- Across Spark applications, Spark still supports fine-grained sharing using the resource offer concept in Mesos [56], which lets disparate applications launch fine-grained tasks on a cluster through a common API. This allows Spark applications to share resources dynamically both with each other and with other cluster computing frameworks, like Hadoop. Delay scheduling still works within the resource offer model to provide data locality.

- Finally, Spark has been extended to perform *distributed scheduling* using the Sparrow system [83]. This system allows multiple Spark applications to queue work on the same cluster in a decentralized manner while still providing data locality, low latency, and fairness. Distributed scheduling can greatly improve scalability when multiple applications submit jobs concurrently by avoiding a centralized master.

Because the vast majority of clusters are multi-user and workloads on them are becoming increasingly interactive, these features can give Spark significant performance benefits over more traditional static partitioning of clusters.

2.5.3 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.



a) Lines typed by user b) Resulting object graph

Figure 2.6. Example of how the Spark interpreter translates lines entered by the user to Java objects.

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 2.6 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.

2.5.4 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 share RDDs across Spark instances through a unified memory manager. The Tachyon project at Berkeley [68] is ongoing work in this direction.

2.5.5 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 (§2.3.2). 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 [66]. In contrast, for RDDs with narrow dependencies on data in stable storage, such as the points in our logistic regression example (§2.3.2) 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 [114].

Finally, note that the read-only nature of RDDs makes them simpler to check-

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

point 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.

2.6 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 80× 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 (§2.6.1) and PageRank (§2.6.2) against Hadoop. We then evaluate fault recovery in Spark (§2.6.3) and behavior when a dataset does not fit in memory (§2.6.4). Finally, we discuss results for interactive data mining (§2.6.5) and several real user applications (§2.6.6).

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 I/O costs accurately.

2.6.1 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.

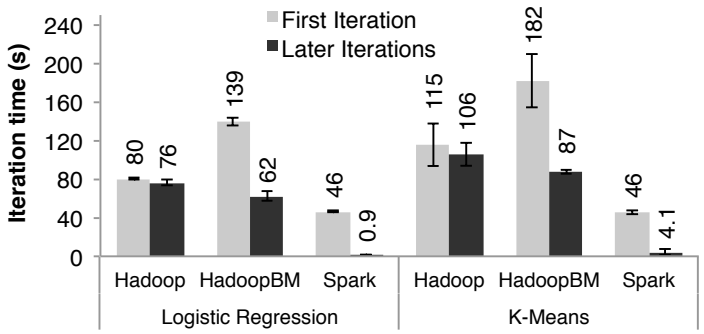


Figure 2.7. Duration of the first and later iterations in Hadoop, HadoopBinMem and Spark for logistic regression and k-means using 100 GB of data on a 100-node cluster.

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.

First Iterations All three systems read text input from HDFS in their first iterations. As shown in the light bars in Figure 2.7, 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 2.7 also shows the average running times for subsequent iterations. For logistic regression, Spark 85× and 70× faster than Hadoop and HadoopBinMem respectively on 100 machines. For the more compute-intensive k-means application, Spark still achieved speedup of 26× to 21×. Note that in all cases, the programs computed the same result via the same algorithm.

Understanding the Speedup We were surprised to find that Spark outperformed even Hadoop with in-memory storage of binary data (HadoopBinMem) by a 85× 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,

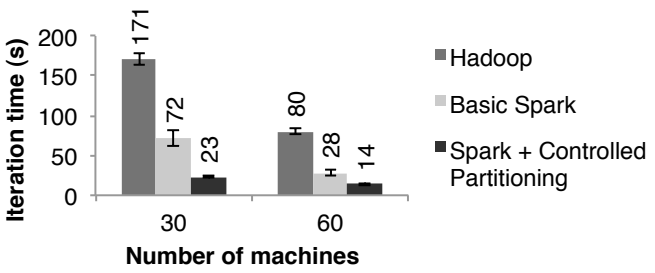


Figure 2.8. Performance of PageRank on Hadoop and Spark.

2. Overhead of HDFS while serving data, and
3. Deserialization cost to convert binary records to usable in-memory Java objects.

We confirmed these factors through separate microbenchmarks. For example, to measure Hadoop launch overhead, 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 HDFS overhead, we found that HDFS performed multiple memory copies and a checksum to serve each block. Finally, we found that even for in-memory binary data, the deserialization step to read it through Hadoop's SequenceFileInputFormat was more expensive than the logistic regression computation, explaining the slowdown of HadoopBinMem.

2.6.2 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 2.8 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 2.3.2, 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 2.7.1. The iteration times were similar to the ones in Figure 2.8, 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.

2.6.3 Fault Recovery

We evaluated the cost of reconstructing RDD partitions using lineage after a node failure in the k-means application. Figure 2.9 compares the running times for

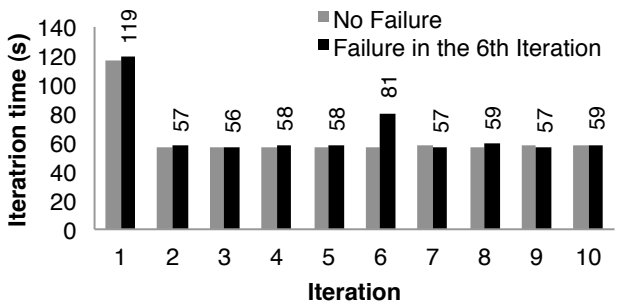


Figure 2.9. Iteration times for k-means in presence of a failure. One machine was killed at the start of the 6th iteration, resulting in partial reconstruction of an RDD using lineage.

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 rebuilt, 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.

2.6.4 Behavior with Insufficient Memory

So far, we ensured that every machine in the cluster had enough memory to store all the RDDs across iterations. 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 2.10. We see that performance degrades gracefully with less space.

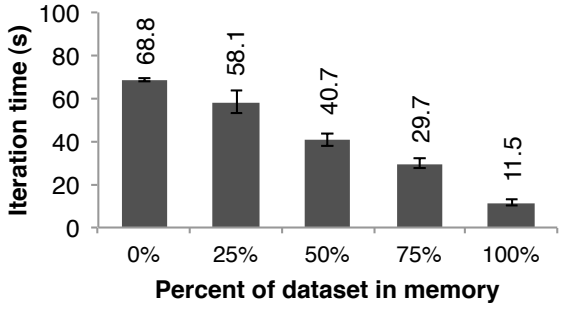


Figure 2.10. Performance of logistic regression using 100 GB data on 25 machines with varying amounts of data in memory.

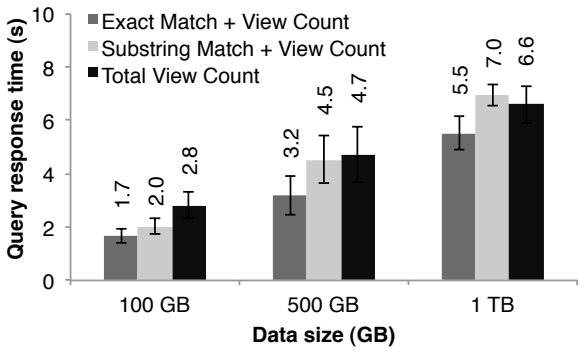


Figure 2.11. Response times for interactive queries on Spark, scanning increasingly larger input datasets on 100 machines.

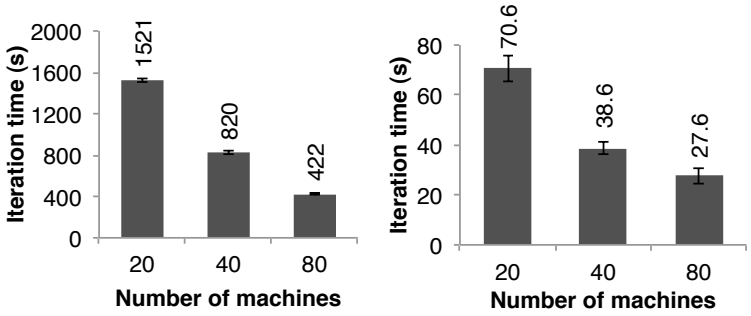
2.6.5 Interactive Data Mining

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 2.11 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.

2.6.6 Real Applications

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



(a) Traffic modeling (b) Spam classification

Figure 2.12. Per-iteration running time of two user applications implemented with Spark. Error bars show standard deviations.

example, one report ran as a series of Hive [104] 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 [57] 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 2.12(a).

Twitter Spam Classification The Monarch project at Berkeley [102] 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 2.6.1, but they used a distributed *reduceByKey* to sum the gradient vectors in parallel. In Figure 2.12(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.

2.7 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 (§2.7.1). In addition, we discuss another benefit of the lineage information in RDDs that we are pursuing, which is to facilitate debugging across these models (§2.7.3).

2.7.1 Expressing Existing Programming Models

RDDs can *efficiently* express a number of cluster programming models that have so far been proposed as independent systems. 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 datasets, so they can be implemented using RDD transformations. In Chapter 3, we describe an efficient implementation of SQL over RDDs called Shark.

Pregel: Google's Pregel [72] 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 [118] for more details.

Iterative MapReduce: Several recently proposed systems, including HaLoop [22] and Twister [37], 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.

2.7.2 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. In fact, most of today's MapReduce applications run over filesystems that do not allow updates to files, such as HDFS. In later chapters (3 and 5), we study the expressiveness of RDDs in more detail.

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.

2.7.3 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 distributed systems [51], 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 [118].

2.8 Related Work

Cluster Programming Models: Related work in cluster programming models falls into several classes. First, data flow models such as MapReduce [36], Dryad [61] and CIEL [77] 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 [115] and FlumeJava [25], 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 [72] supports iterative graph applications, while Twister [37] and HaLoop [22] 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 [86] lets users run parallel functions

[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 [81] would still require data replication and serialization, which can be costly for some applications, as shown in §2.6.1.

that read and update cells in a distributed hash table. Distributed shared memory (DSM) systems [79] and key-value stores like RAMCloud [81] 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 2.2.3, RDDs also provide other advantages over DSM, such as straggler mitigation.

Caching Systems: Nectar [50] can reuse intermediate results across DryadLINQ jobs by identifying common subexpressions with program analysis [55]. 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 [77] and FlumeJava [25] 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 [6]. 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 [20] and [28] 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 [89]. 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 consistency. These overheads are not required with the coarse-grained transformation model of RDDs.

2.9 Summary

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 80× in iterative applications and can be used interactively to query hundreds of gigabytes of data.

Spark is now hosted at the Apache Software Foundation as an open source project, at `spark.incubator.apache.org`. Since the project came out, it has been used and enhanced by a large open source community—at the time of this writing, over 100 developers have contributed to Spark.

While this chapter covered some simple existing programming models that can quickly be implemented on Spark, RDDs can express more sophisticated computations as well, along with popular optimizations within various models (*e.g.*, column-oriented storage). The next two chapters cover such models.

Chapter 3

Models Built over RDDs

3.1 Introduction

Although the previous chapter covered simple programming models built on RDDs, such as Pregel and iterative MapReduce, the RDD abstraction can be used to implement more complex ones as well, including some of the key optimizations used in specialized engines (*e.g.*, column-oriented processing or indexes). Since Spark was released, we and others have implemented several such models, including the Shark SQL engine [113], the GraphX graph processing system [112], and the MLlib machine learning library [96]. In this chapter, we sketch the techniques used to implement some of these systems, diving more deeply into Shark as an example.

To recap previous chapter, RDDs provide the following facilities:

- Immutable storage of arbitrary records across a cluster (in Spark, the records are Java objects).
- Control over data partitioning, using a key for each record.
- Coarse-grained operators that take into account partitioning.
- Low latency due to in-memory storage.

We next show how these have been used to implement more sophisticated processing and storage.

3.2 Techniques for Implementing Other Models on RDDs

In specialized engines, not only the operators on data but its storage format and access methods are optimized. For example, a SQL engine like Shark may process data in a column-oriented format, while a graph engine like GraphX may

perform best when data is indexed. We discuss common ways that have emerged to implement these optimizations with RDDs, leveraging the fault tolerance and composition benefits of the RDD model while keeping the performance of specialized systems.

3.2.1 Data Format Within RDDs

Although RDDs store simple, flat records of data we found that one effective strategy for achieving richer storage formats is to store *multiple* data items in the same RDD record, implementing more sophisticated storage within each record. Batching even a few thousand records this way is enough to achieve most of the efficiency gains of specialized data structures, while still keeping the size of each RDD "record" to a few megabytes.

For example, in analytical databases, one common optimization is column-oriented storage and compression. In the Shark system, we store multiple database records in one Spark record and apply these optimizations. The degree of compression in a batch of 10,000 to 100,000 records is so close to the degree one would get by storing the whole database in column-oriented format that we still get significant gains. As a more advanced example, GraphX [112] needs to perform very sparse joins against datasets representing vertices and edges of a graph. It does this by storing a *hash table* containing multiple graph records within each RDD record, enabling fast lookups of particular vertices or edges when joining with new data. This is important because in many graph algorithms, the last few iterations only have a few edges or vertices active, but they still need to be joined with the entire dataset. The same mechanism could be used for a more efficient implementation of Spark Streaming's *updateStateByKey* (Section 4.3.3).

There are two aspects of the RDD model that this make this approach efficient. First, because RDDs are most commonly in memory, *pointers* can be used to read only the relevant part of a "composite" record for each operation. For example, a column-oriented representation of a set of (Int, Float) records might be implemented as an array of Ints and an array of Floats. If we only wish to read the integer field, we can follow the pointer to the first array and avoid scanning the memory of the second. Similarly, in the RDD of hash tables above, we can look up just the desired records.[1]

Second, a natural question might be how to combine processing types efficiently if each computation model has its own representation of batched records. Fortunately, the low-level interface to RDDs is based on *iterators* (see the *compute* method in Section 2.4), which enable fast, pipelined conversions between formats. An RDD containing composite records can efficiently return an iterator over unpacked records through a *flatMap* operation on the batches, and this can be pipelined with further narrow transformations on the unpacked records, or with a transformation

[1] It would be interesting to add an API offering similar properties for on-disk records, similar to database storage system APIs, but we have not yet done this.

that re-packs them into another format, minimizing the amount of unpacked data in-flight. Iterators in general are an efficient interface for in-memory data because they typically perform scans. As long as each batched record fits in the CPU cache, this leads to fast transformations and conversions.

3.2.2 Data Partitioning

A second optimization common in specialized models is *partitioning* data across a cluster in domain-specific ways to speed up an application. For example, Pregel and HaLoop keep data partitioned using a possibly user-defined function to speed up joins against a dataset. Parallel databases commonly provide various forms of partitioning as well.

With RDDs, partitioning can easily be achieved through a key for each record. (Indeed, this is the only notion of structure within a record that RDDs have.) Note that even though the key is for the whole record, "composite" records that match multiple underlying data items can still have a meaningful key. For example, in GraphX, the records in each partition have the same hash code modulo the number of partitions, but are still hashed within it to enable efficient lookup. And when systems using composite records perform shuffle operations like *groupBy*, they can hash each outgoing record to the key of the target composite record.

A final interesting use case of partitioning is *compound* data structures, where some fields of the data structure are updated over time and some are not. For example, in the PageRank application in Section 2.3.2, there are two pieces of data for each page: a list of links, which is immutable, and a rank, which is mutable. We represent this as two RDDs, one of (ID, links) pairs and one of (ID, ranks) pairs, that are co-partitioned by ID. The system places the ranks and links for each ID preferentially on the same machine, but we can update the ranks separately without changing the links. Similar techniques are used to keep vertex and edge state in GraphX.

In general, one can think of RDDs as a more declarative memory abstraction for the cluster environment. When using memory on a single machine, programmers worry about data layout mainly to optimize lookups and to maximizes colocation of information commonly accessed together. RDDs give control over the same concerns for distributed memory, by letting users select a partitioning function and co-partition datasets, but they avoid asking having users specify exactly *where* each partition will be located. Thus, the runtime can efficiently place partitions based on the available resources or move them on failure, while the programmer still controls access performance.

3.2.3 Dealing with Immutability

A third difference between the RDD model and most specialized systems is that RDDs are immutable. Immutability is important to reason about lineage and fault

recovery, although fundamentally it is no different from having mutable datasets and tracking version numbers for these purposes. However, one might ask whether it leads to inefficiency.

While some overhead will certainly occur from immutability and fault tolerance, we found that two techniques can yield good performance in many cases:

1. Compound data structures represented as multiple co-partitioned RDDs, as in the PageRank example in the previous section, allow the program to mutate only the part of the state that need to change. In many algorithms, some fields of records never change while others change on almost every iteration, so this approach can capture them efficiently.

2. Even within a record, pointers can be used to reuse state from the previous "version" of the record when the internal data structures are immutable. For example, strings in Java are immutable, so a *map* on a (Int, String) record that changes the Int but keeps the same String will just have a pointer to the previous String object, without copying it. More generally, *persistent data structures* from functional programming [64] can be used to represent incremental updates to other forms of data (*e.g.*, hash tables) as a delta from a previous version. It is quite pleasant that many ideas from functional programming can directly help RDDs.

In future work, we plan to explore other ways to track multiple versions of state while approaching the performance of mutable-state systems.

3.2.4 Implementing Custom Transformations

A final technique we found useful in some applications is implementing custom dependency patterns and transformations using the low-level RDD interface (Section 2.4). The interface is very simple, requiring only a list of dependencies on parent RDDs and a function for computing an iterator for a partition of the RDD given iterators from the parents. Our experience implementing some of these custom operators in the first versions of Shark and GraphX led to many new operators being added to Spark itself. For example, one useful one is *mapPartitions*, which, given an RDD[T] and a function from Iterator[T] to Iterator[U], returns an RDD[U] by applying it to each partition. This is very close to the lowest-level interface to RDDs and allows transformations to perform non-functional operations within each partition (*e.g.*, use mutable state). Shark also contained custom versions of *join* and *groupBy* that we are working to replace with the built-in operators. Nonetheless, note that even when applications implement custom transformations, they still automatically benefit from the fault tolerance, multitenancy, and composition benefits of the RDD model, and remain much simpler to develop than standalone systems.

3.3 Shark: SQL on RDDs

As an example of implementing sophisticated storage and processing on RDDs, we sketch the Shark system. Shark achieves good performance in the heavily studied field of parallel databases while offering fault tolerance and complex analytics capabilities that traditional databases lack.

3.3.1 Motivation

Modern data analysis faces several challenges. First, data volumes are expanding quickly, creating the need to scale out across clusters of hundreds of commodity machines. Second, this 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 [36] and various generalizations [61, 27], 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 [31]. 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 [27], or that combine it with a traditional database on each node, such as HadoopDB [1], report a minimum latency of 10 seconds. As such, MapReduce approaches have largely been dismissed for interactive-speed queries [84], and even Google is developing new engines for such workloads [75, 95].

Instead, most MPP analytic databases (*e.g.*, Vertica, Greenplum, Teradata) and several of the new low-latency engines built for MapReduce environments (*e.g.*, Google F1 [95], Impala [60]) employ a coarser-grained recovery model, where an entire query must be resubmitted if a machine fails. This works well for short queries where a retry is cheap, but faces significant challenges for long queries as clusters scale up [1]. In addition, these systems often lack the rich analytics functions that are easy to implement in MapReduce, such as machine learning and graph algorithms. Indeed, 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 offer fine-grained fault recovery across both types of operations. We present a new system that meets these goals, called Shark.

Shark uses the RDD model 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. Second, even traditional SQL warehouse workloads exhibit strong temporal and spatial locality, because more-recent fact table data 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 [7].

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. First, to store and process relational data efficiently, we implemented in-memory columnar storage and compression. This reduced both the data size and the processing time by as much as 5× over naïvely storing records. 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 [104], 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× faster than Hive, runs iterative machine learning algorithms more than 100× 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 [84], but it offers fine-grained recovery and complex analytics features that these systems lack.

3.4 Implementation

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 Apache 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 DAG scheduling techniques, such as placing tasks close to their input data, rerunning lost tasks, and performing straggler mitigation (Section 2.5.1).

While this basic approach makes it possible to run SQL on 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 such a plan, naïvely executing it over RDDs can be inefficient. In this section, we discuss several optimizations we made within the RDD model to efficiently run SQL.

3.4.1 Columnar 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 default approach in Spark's 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 [100], 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.

3.4.2 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. It adds a `DISTRIBUTE BY` clause in the table creation statement that takes a column to distribute on.

3.4.3 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, *e.g.,* to search only for events over a typical date range or news article.

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 driver node 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, and prunes partitions that cannot match the predicates. This is done by simply building an RDD that only depends on some of the parent's partitions.

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 3.5 provides more details on this workload.

3.4.4 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.

Examples of optimizations performed currently implemented with PDE include:

- **Join algorithm selection.** When joining two tables, Shark uses PDE to select whether to run a shuffle join (hashing both collections of records across the network based on their key) or broadcast join (sending a copy of the smaller table to all nodes with data from the larger one). The optimal algorithm depends on the sizes of the tables: if one table is much smaller than the other, broadcast join uses less communication. Because the table sizes may not be known in advance (*e.g.,* due to UDFs), choosing the algorithm at runtime leads to better decisions.

- **Degree of parallelism.** The degree of parallelism for reduce tasks can have a large performance impact in MapReduce-like systems: launching too few reducers may overload reducers' network connections and exhaust their memories, while launching too many may prolong the job due to scheduling overhead [17]. 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.

- **Skew handling.** In a similar way that pre-partitioning the map outputs into many small buckets can help us choose the number of reduce tasks, it can also help us identify popular keys and treat them specially. These popular partitions can go to a single reduce task, while other buckets might be coalesced to form larger tasks.

Partial DAG execution complements existing adaptive query optimization techniques that typically run in a single-node system [16, 63, 107], 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 [115].

While PDE is currently only implementd in our Shark prototype, we plan to add it to the core Spark engine in the future to benefit from these optimizations.

3.5 Performance

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 [84].

2. TPC-H Dataset: 100 GB and 1 TB datasets generated by the DBGEN program [106].

3. Real Hive Warehouse: 1.7 TB of sampled Hive warehouse data from an early (anonymized) industrial user of Shark.

Overall, our results show that Shark can perform more than 100× faster than Hive and Hadoop. In particular, Shark provides comparable performance gains to those reported for MPP databases in Pavlo et al.'s comparison [84]. In some cases where data fits in memory, Shark exceeds the performance reported for MPP databases there.

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 Java VM. 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 complex analytics (*e.g.*, machine learning) on the same data, which we believe will be essential for future analytics workloads.

3.5.1 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.

3.5.2 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 [84]. We reused the dataset and queries from their benchmarks to compare Shark against Hive.

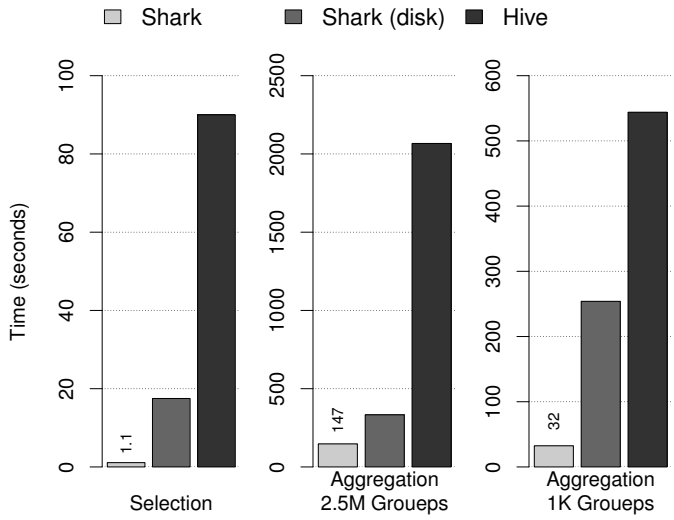


Figure 3.1. Selection and aggregation query runtimes (seconds) from Pavlo et al. benchmark

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 3.1 and 3.2. In this subsection, we hand-tuned Hive's number of reduce tasks to produce optimal results for Hive. Despite this tuning, Shark outperformed Hive in all cases by a wide margin.

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

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

In [84], 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× faster than Hive for in-memory data, and 5× on data read from HDFS.

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);
```

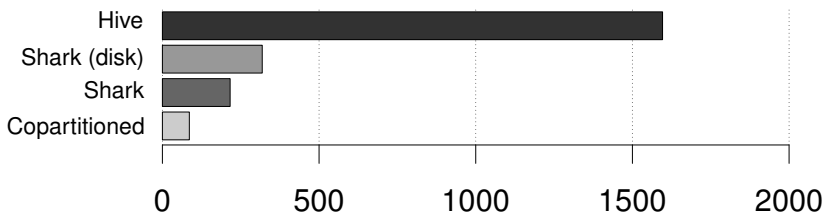


Figure 3.2. Join query runtime (seconds) from Pavlo benchmark

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.

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 3.2 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.

Data Loading Hadoop was shown by [84] 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 3.4, 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× higher in Shark's memory store than that of HDFS.

3.5.3 Microbenchmarks

To understand the factors affecting Shark's performance, we conducted a sequence of microbenchmarks. We generated 100 GB and 1 TB of data using the DBGEN program provided by TPC-H [106]. We chose this dataset because it contains tables and columns of varying cardinality and can be used to create a myriad of microbenchmarks 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.

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.

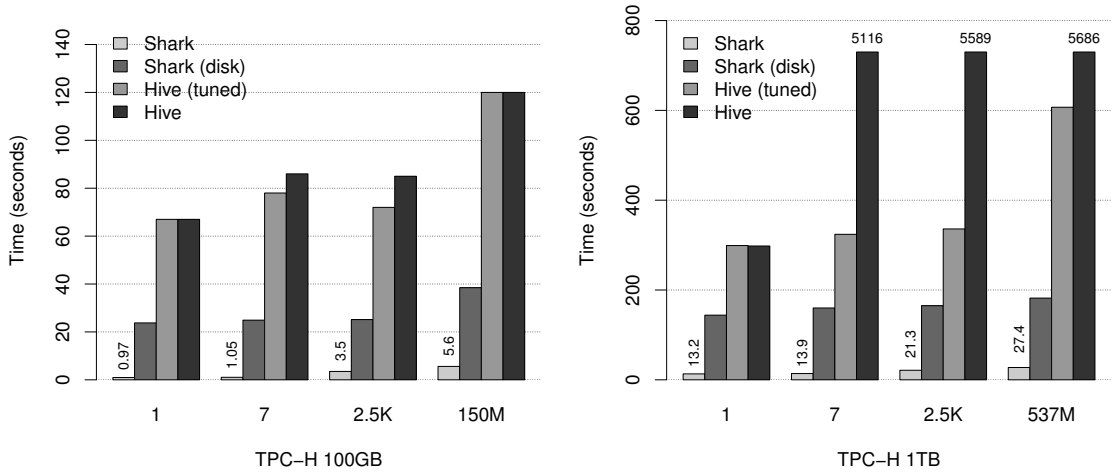


Figure 3.3. Aggregation queries on *lineitem* table. X-axis indicates the number of groups for each aggregation query.

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 3.3 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× faster than hand-tuned Hive for queries with small numbers of groups, and 20× 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. Shark's execution engine can launch thousands of tasks a second to maximize the degree of parallelism available, and it only needs to perform hash based shuffle for aggregation queries. In the case of Hive, the only shuffle mechanism provided by Hadoop MapReduce is a sort-based shuffle, which is computationally more expensive than hash.

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 3.4 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

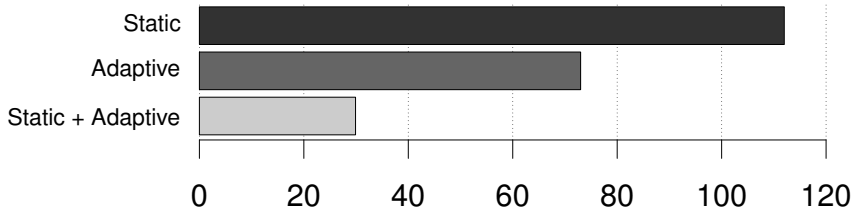


Figure 3.4. Join strategies chosen by optimizers (seconds)

and partial DAG execution led to a 3× performance improvement over a naïve, statically chosen plan.

3.5.4 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 3.5 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).

After this recovery, subsequent queries operated against the recovered dataset, albeit with fewer machines. In Figure 3.5, 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. corollary

3.5.5 Real Hive Warehouse Queries

An early (anonymized) 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,

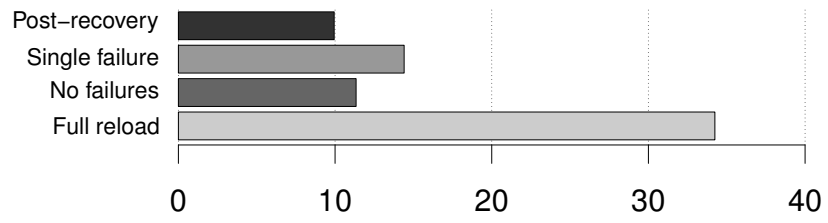


Figure 3.5. Query time with failures (seconds)

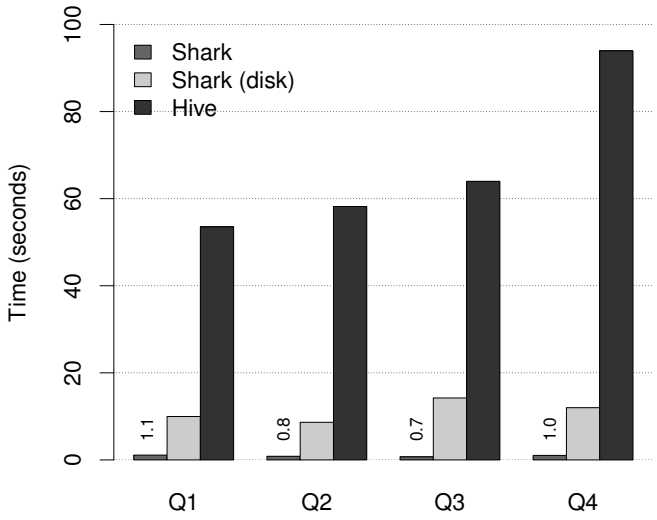


Figure 3.6. Real Hive warehouse workloads

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 3.6 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 3.4.3. The map pruning technique, on average, reduced the amount of data scanned by a factor of 30.

3.6 Combining SQL with Complex Analytics

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 supports machine learning as a first-class feature. 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 [33, 41] 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.

3.6.1 Language Integration

Shark's complex analytics capabilities are available to users in two ways. First, Shark offers an API in Scala that can be called in Spark programs to extract Shark data as an RDD. Users can then write arbitrary Spark computations on the RDD, where they get automatically pipelined with the SQL ones. Second, we have also extended the Hive dialect of SQL to allow calling Scala functions that operate on RDDs, making it easy to expose existing Scala libraries to Shark.

As an example of Scala integration, Listing 3.1 illustrates a data analysis pipeline that performs logistic regression [53] over a user database using a combination of SQL and Scala. 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.

```
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 3.1. Logistic Regression Example

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 `map`, `mapRows`, and `reduce` functions are automatically parallelized by Shark and Spark to execute across a cluster, and the master program simply collects the output of the `reduce` function to update w. They are also pipelined with the reduce step of the join operation in SQL, passing column-oriented data from SQL to Scala code through an iterator interface as discussed in Section 3.2.1.

We also offer an API to call Scala functions from SQL. Given a Scala function on RDDs, such as K-means clustering or logistic regression, users can annotate it as callable from SQL, then write SQL code like the following:

```
CREATE TABLE user_features AS SELECT age, country FROM user;

GENERATE KMeans(user_features, 10) AS TABLE user_features_clustered;
```

In this case, the `user_features_clustered` table will contain the age, country, and a new field for the cluster ID of each row. The `10` passed to `KMeans` is the number of clusters.

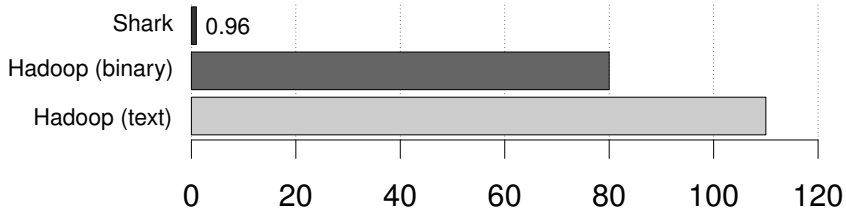


Figure 3.7. Logistic regression, per-iteration runtime (seconds)

3.6.2 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.

3.6.3 Performance

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 3.7 and 3.8 show the time to execute a single iteration 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× faster than Hive and Hadoop for logistic regression and 30× 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.

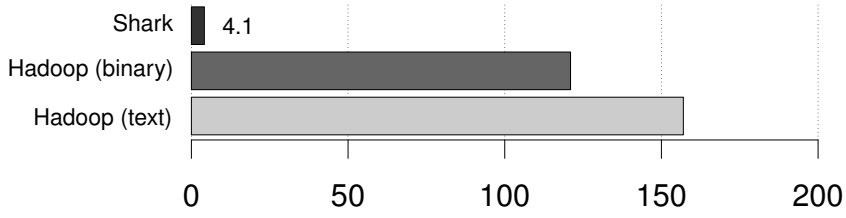


Figure 3.8. K-means clustering, per-iteration runtime (seconds)

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 3.7 and 3.8. In the case of Hive and Hadoop, every iteration took the reported time because data was loaded from HDFS for every iteration.

3.7 Summary

In this chapter, we have presented techniques for implementing more sophisticated processing and storage optimizations with RDDs, and illustrated them through the Shark data warehousing systems. Similar techniques, including batching data within Spark records, indexing it, and optimizing partitioning, have been used in GraphX [112], MLlib [96], MLI [98] and other projects. Together, these techniques have allowed RDD-based systems to achieve similar performance to specialized systems in each domain, while providing much higher performance in applications that *combine* processing types, and fault tolerance across these types of computations.

Chapter 4

Discretized Streams

4.1 Introduction

This chapter applies the RDD model in Chapter 2 to perhaps the most surprising area: large-scale stream processing. While the resulting design runs counter to traditional streaming systems, it offers rich fault recovery benefits and powerful composition with the other processing types.

The motivation for large-scale stream processing is simple: 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 applications, there is a need for stream processing models that scale to large clusters.

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, faults and stragglers become important problems [36], and streaming applications in particular must recover quickly. Indeed, fast recovery is *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 [14], TimeStream [87], MapReduce Online [34], and streaming databases [18, 26, 29], 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 [58]: *replication*, where there are two copies of each node [18, 93], or *upstream backup*, where nodes buffer sent messages and replay them to a new copy of a failed node [87, 34, 14]. Neither approach is attractive in large clusters:

replication costs 2× 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 [93] to coordinate replicas, so a straggler will slow down both replicas.

We propose a new stream processing model, *discretized streams* (D-Streams), that overcomes these challenges. Instead of managing long-lived operators, D-Streams structure streaming computations 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 visible 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 build on the RDD data structure introduced in Chapter 2, which keeps data in memory and can recover it without replication by tracking the *lineage graph* of operations used to build it. With RDDs, we show that we can attain end-to-end latencies below a second. 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 [93]),[1] but becomes simple with the fully deterministic D-Stream model. In a similar way, D-Streams can recover from stragglers using speculative execution [36], while previous streaming systems do not handle them.

We have implemented D-Streams in a system called Spark Streaming, built on Spark. 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

[1] The only parallel recovery algorithm we are aware of, by Hwang *et al.* [59], only tolerates one node failure and cannot handle stragglers.

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.

4.2 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:

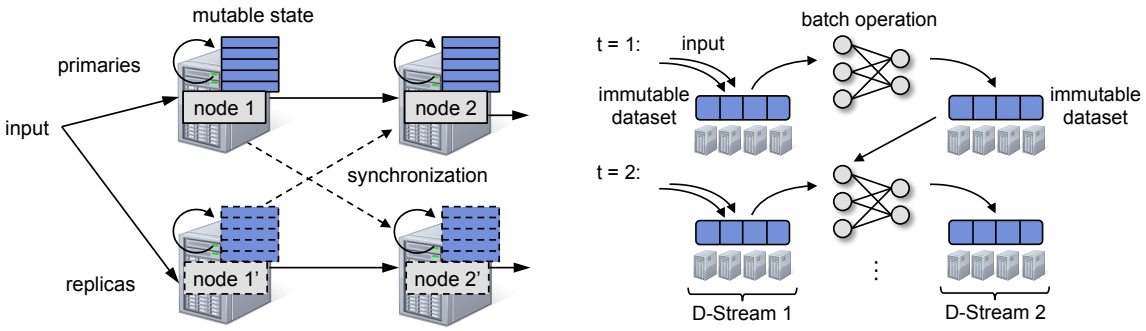
- **Site activity statistics:** Facebook built a distributed aggregation system called Puma that gives advertisers statistics about users clicking their pages within 10–30 seconds and processes 10^6 events/s [94].
- **Cluster monitoring:** Datacenter operators often collect and mine program logs to detect problems, using systems like Flume [9] on hundreds of nodes [52].
- **Spam detection:** A social network such as Twitter may wish to identify new spam campaigns in real time using statistical learning algorithms [102].

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.

4.2.1 Goals

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

1. Scalability to hundreds of nodes.



(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 [93, 18] 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 4.1. Comparison of traditional continuous stream processing (a) with discretized streams (b).

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 [87, 110], and neither tolerates stragglers.

4.2.2 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 [29]. Figure 4.1(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 the state of operators on a lost, or slow, node. Previous

systems use one of two schemes, *replication* and *upstream backup* [58], 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 [93] or Borealis's DPC [18], 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 [87] and MapReduce Online [34] use this model. Popular message queueing systems, like Storm [14], 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.

4.3 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

[2] Storm's Trident layer [73] 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.

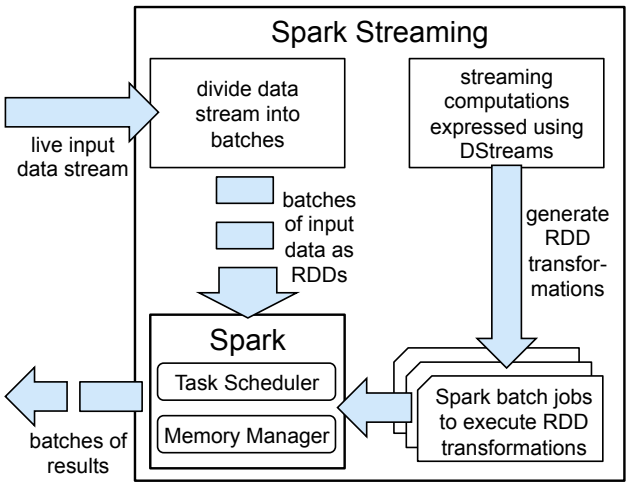


Figure 4.2. High-level overview of the Spark Streaming system. Spark Streaming divides input data streams into batches and stores them in Spark's memory. It then executes a streaming application by generating Spark jobs to process the batches.

tolerance, the D-Stream model gives other benefits, such as powerful unification with batch processing.

4.3.1 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 system in a distributed manner. In the latter case, the intermediate state is stored as *resilient distributed datasets (RDDs)*, 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 4.1(b) shows our model.

We implemented our system, Spark Streaming, based on this model. We used Spark as our batch processing engine for each batch of data. Figure 4.2 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

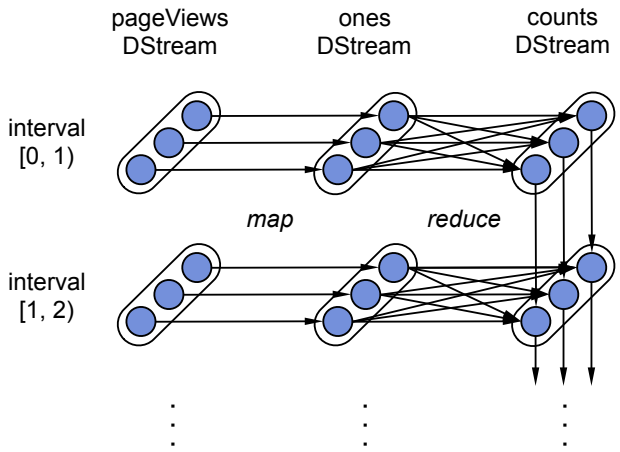


Figure 4.3. Lineage graph for RDDs in the view count program. Each oval is an RDD, with partitions shown as circles. Each sequence of RDDs is a D-Stream.

count of view events by URL. Spark Streaming exposes D-Streams through a functional API similar to LINQ [115, 3] 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.

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 4.2). 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. Spark tracks this information at the level of *partitions* within each distributed dataset, as shown in Figure 4.3. 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

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

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 [36], 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.

2. The lineage graph often enables data from different timesteps to be rebuilt in parallel. For example, in Figure 4.3, 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 (§4.6.2).

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 4.4.

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

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

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 [67].

These timing concerns are inherent to stream processing, as any system must handle external delays. They have been studied in detail in databases [67, 99]. 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 work.

4.3.3 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 [29, 15], 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.

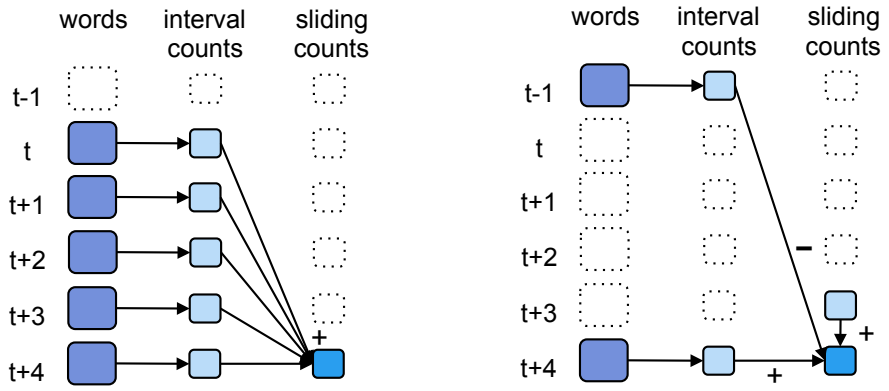
In Spark Streaming, users register one or more streams using a functional API. The program can define *input* streams to be read from outside, which receive data either by having nodes listen on a port or by loading it periodically from a storage system (*e.g.,* HDFS). It can then apply two types of 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 [36, 115], including *map*, *reduce*, *groupBy*, and *join*. We provide all the operations in Spark. 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 [29, 15]. These include:



(a) Associative only (b) Associative & invertible

Figure 4.4. *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 $[t, t+5)$.

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 4.4(a). If the aggregation function is also *invertible*, a more efficient version also takes a function for "subtracting" values and maintains the state incrementally (Figure 4.4(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 *updateStateByKey* operation that transforms streams of

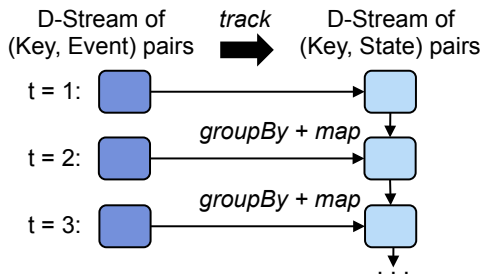


Figure 4.5. RDDs created by the *updateStateByKey* operation.

(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) => (ev == Exit ? null : 1),    // update function
  "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 4.5 shows the RDDs built by *updateStateByKey*, 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)))`.

4.3.4 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 [18], 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.

4.3.5 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. 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

Aspect	D-Streams	Continuous processing 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 [18, 87]
Late records	Slack time or app-level correction	Slack time, out of order processing [67, 99]
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 [43]; not in message queueing systems

Table 4.1. Comparing D-Streams with continuous operator systems.

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.

4.3.6 Summary

To end our overview of D-Streams, we compare them with continuous operator systems in Table 4.1. 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 [87, 18], *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.

4.4 System Architecture

We have implemented D-Streams in a system called Spark Streaming, based on a modified version of the Spark processing engine. Spark Streaming consists of three components, shown in Figure 4.6:

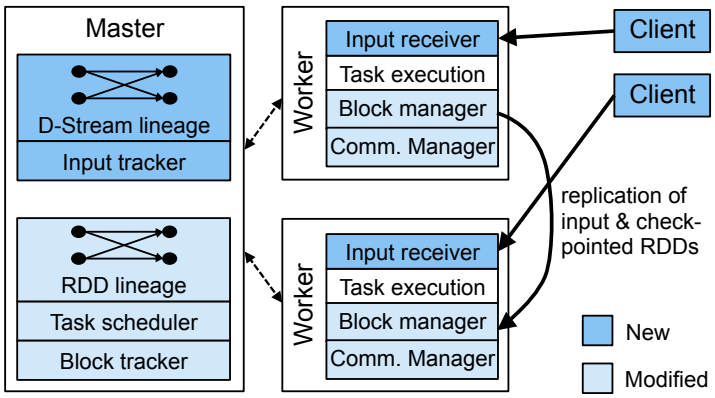


Figure 4.6. Components of Spark Streaming, showing what we modified over the original version of Spark.

- 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 4.4.2.

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 computed 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.

4.4.1 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 [9]. 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 [61], it simply starts each task whenever its parents are finished.

Spark Streaming relies on Spark's existing batch scheduler within each timestep (Section 2.5.1), and performs many of the optimizations in systems like DryadLINQ [115]:

- 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 Section 2.5.1.

4.4.2 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 4.3. 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 4.5.3).

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

4.4.3 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 applications 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.

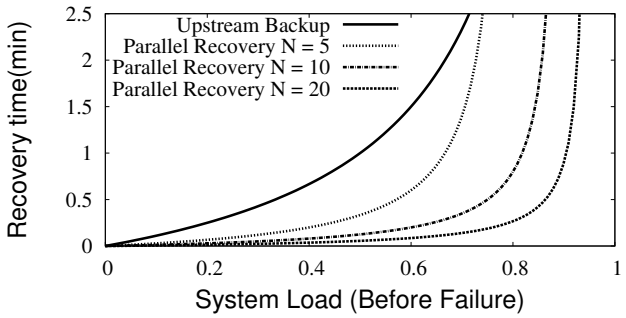


Figure 4.7. Recovery time for single-node upstream backup vs. parallel recovery on N nodes, as a function of the load before a failure. We assume the time since the last checkpoint is 1 min.

4.5 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.

4.5.1 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 4.3, 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 4.7 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 λ. Then during each minute of

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

recovery, the backup node can do 1 min of work, but receives λ minutes of new work. Thus, it fully recovers from the λ units of work that the failed node did since the last checkpoint at a time t_{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 λ/N work, but also receive new data at a rate of $\frac{N}{N-1}\lambda$. The time t_{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.

4.5.2 Straggler Mitigation

Besides failures, another concern in large clusters is stragglers [36]. 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 [93, 18], focus on *synchronizing* two replicas.

In our implementation, we use a simple threshold to detect stragglers: whenever a task runs more than 1.4× 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.

4.5.3 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.

[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.

4.6 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 [57]. These latter applications also leverage D-Streams' unification with batch processing, as we shall discuss.

4.6.1 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 [78, 14]. We ran these applications on "m1.xlarge" nodes on Amazon EC2, each with 4 cores and 15 GB RAM.

Figure 4.8 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

[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).

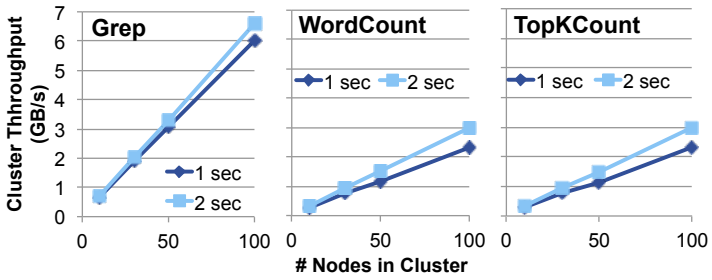


Figure 4.8. Maximum throughput attainable under a given latency bound (1 s or 2 s) by Spark Streaming.

latency improves throughput slightly, but even the performance at sub-second latency is high.

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 [82], StreamBase reports 245,000 records/s on 8 cores [105], and Esper reports 500,000 records/s on 4 cores [38]. 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 4.9, 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.

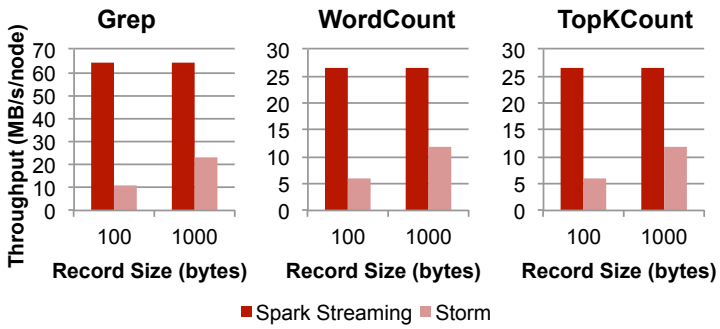


Figure 4.9. Throughput vs Storm on 30 nodes.

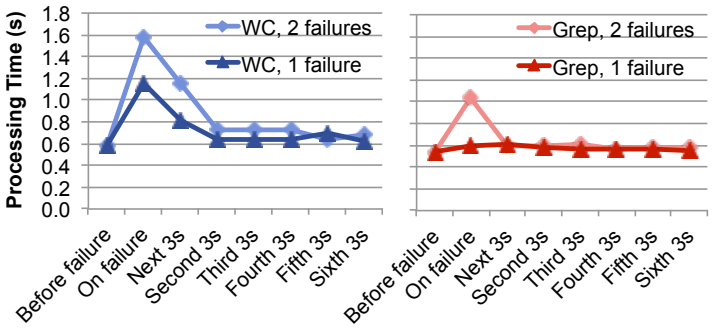


Figure 4.10. Interval processing times for WordCount (WC) and Grep under failures. We show the average time to process each 1s batch of data before a failure, during the interval of the failure, and during 3-second periods after. Results are over 5 runs.

4.6.2 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 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 4.10. 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

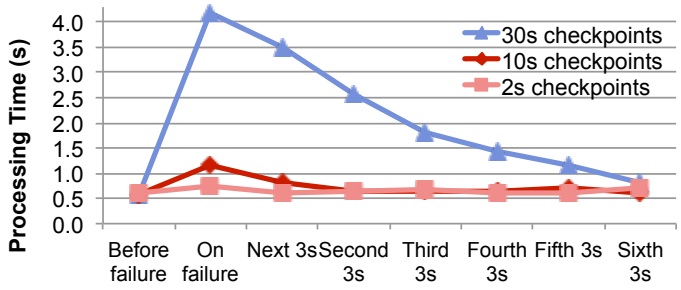


Figure 4.11. Effect of checkpoint time in WordCount.

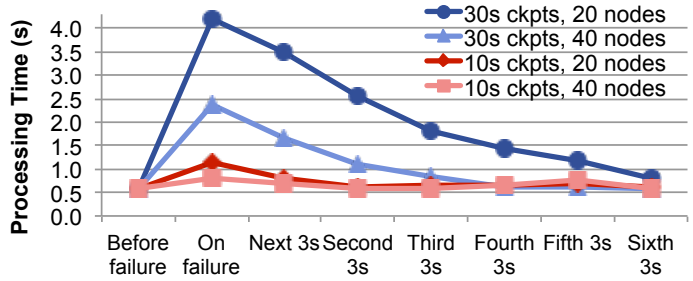


Figure 4.12. Recovery of WordCount on 20 & 40 nodes.

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 4.11 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 40 nodes. As Figure 4.12 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 4.4), 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 4.13 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

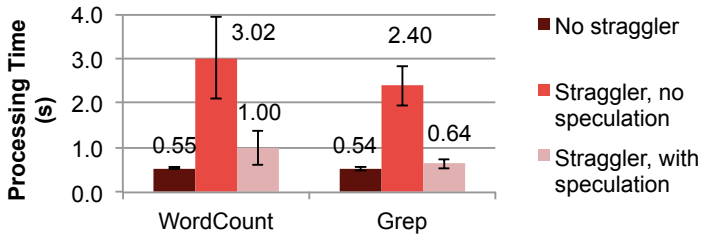


Figure 4.13. Processing time of intervals in Grep and WordCount in normal operation, as well as in the presence of a straggler, with and without speculation.

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.

4.6.3 Real Applications

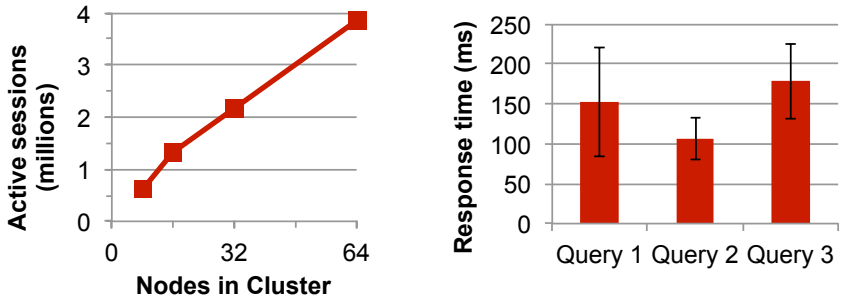
We evaluated the expressiveness of D-Streams by porting two real applications. Both applications are significantly more complex than the test programs shown so far, and both took advantage of D-Streams to perform batch or interactive processing in addition to streaming.

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 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



(a) Scalability (b) Ad-hoc queries

Figure 4.14. 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.

able to compute all the metrics (a 2-stage MapReduce job) in batches as small as 2 seconds. Our code uses the *updateStateByKey* operator described in Section 4.3.3 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.

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 4.14(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 4.14(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.

Crowdsourced Traffic Estimation

We applied the D-Streams to the *Mobile Millennium* traffic information system [57], 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

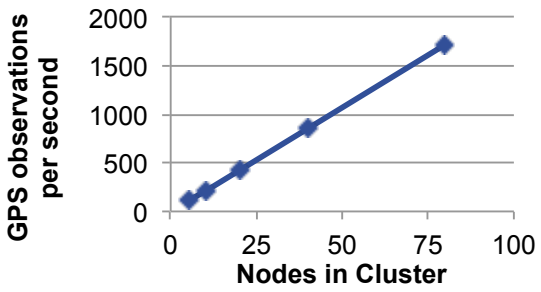


Figure 4.15. Scalability of the *Mobile Millennium* job.

for each road link. The previous implementation [57] 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 4.15 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]

4.7 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 [18, 87], 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.

[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.

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 [15] or Complex Event Processing models [39] 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.

Checkpointing and fault tolerance strategies: Selecting how often to checkpoint each D-Stream automatically would clearly be valuable due to the high cost of checkpoints. In addition, D-Streams allow a wide range of fault tolerance strategies beyond checkpointing, such as *partial replication* of the computation, where a subset of the tasks are actively replicated (*e.g.*, the reduce tasks that we replicated in Section 4.6.2). Applying these strategies automatically would also be interesting.

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.

4.8 Related Work

Streaming Databases Streaming databases such as Aurora, Telegraph, Borealis, and STREAM [23, 26, 18, 15] 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 [58]. 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 [18] or Flux [93]) and upstream backup [58]. The only parallel recovery protocol we are aware of, by Hwang et al [59], only tolerates one node failure, and cannot handle stragglers.

Second, D-Streams also tolerate stragglers, using speculative execution [36]. 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 [87] runs the continuous, stateful operators in Microsoft StreamInsight [3] 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 operator, and takes time proportional to that operator's processing window (*e.g.*, 30 seconds for a 30-second sliding window) [87]. 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 [74, 76] 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 [2] runs stateful computations using an event-driven API but handles reliability by writing all state to a replicated storage system like BigTable.

MapReduce Online [34] 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 check-

point such state into an external system), and does not handle stragglers. Meteor Shower [110] also uses upstream backup, and can take tens of seconds to recover state. iMR [70] offers a MapReduce API for log processing, but can lose data on failure. Percolator [85] 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 [43].

Message Queueing Systems Systems like Storm, S4, and Flume [14, 78, 9] 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 [101]. Trident [73] 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 [69] and Comet [54] 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 [19] 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 [24], 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 [36, 44, 81], 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.

4.9 Summary

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 go against conventional streaming wisdom by *batching* data in small time intervals. This enables highly efficient recovery mechanisms that exploit parallelism across both 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 the Apache Spark project. The code is available at `http://spark.incubator.apache.org`.

Chapter 5

Generality of RDDs

5.1 Introduction

The previous four chapters covered the RDD model and several applications of RDDs to implement some previously specialized computation types. With some relatively simple optimizations within the model, and by pushing latency down, RDDs could match the performance of specialized systems, while offering more efficient composition. Nonetheless, the question remains: why were RDDs so general? Why were they able to approach the performance of specialized systems, and what are the limitations of the model?

In this chapter, we study these questions by exploring the generality of RDDs from two perspectives. First, from an expressiveness point of view, we show that RDDs can emulate *any* distributed system, and will do so *efficiently* in many cases unless the system is sensitive to network latency. In particular, adding data sharing to MapReduce enables this efficient emulation. Second, from a systems point of view, we show that RDDs gave applications control over the most common bottleneck resources in cluster applications (specifically network and storage I/O), which makes it possible to express the same optimizations for these resources that specialized systems have, and hence achieve similar performance. Finally, these explorations of RDDs' generality also identify limitations of the model where it may not emulate other distributed systems efficiently, and lead to several possible areas for extensions.

5.2 Expressiveness Perspective

To characterize the expressiveness of RDDs from a theoretical perspective, we start by comparing RDDs to the MapReduce model, which they derive from and build upon. MapReduce was initially used for a wide range of cluster computations, from SQL [104] to machine learning [12], but was gradually replaced by specialized systems.

5.2.1 What Computations can MapReduce Capture?

The first question we look at is what computations MapReduce itself can express. Although there have been numerous discussions about the limitations of MapReduce and multiple systems extending it, the surprising answer here is that ***MapReduce can emulate any distributed computation***.[1]

To see this, note that any distributed system is composed of nodes that perform local computation and occasionally exchange messages. MapReduce offers the *map* operation, which allows local computation, and *reduce*, which allows all-to-all communication. Thus, any distributed system can be emulated (perhaps somewhat inefficiently) by breaking down its computation into timesteps, running maps to perform the local computation in each timestep, and batching and exchanging messages at the end of each timestep. A series of MapReduce steps is thus enough to capture the whole result. Figure 5.1 shows how such steps would execute.

Two aspects make this kind of emulation inefficient. First, as discussed in in the rest of the dissertation, MapReduce is inefficient at *sharing data* across timesteps because it relies on replicated external storage systems for this. Thus, our emulated distributed system may become slower due to writing out its state on each timestep. Second, the *latency* of the MapReduce steps will determine how well our emulation will match a real network, and most MapReduce implementations were designed for batch environments with minutes to hours of latency.

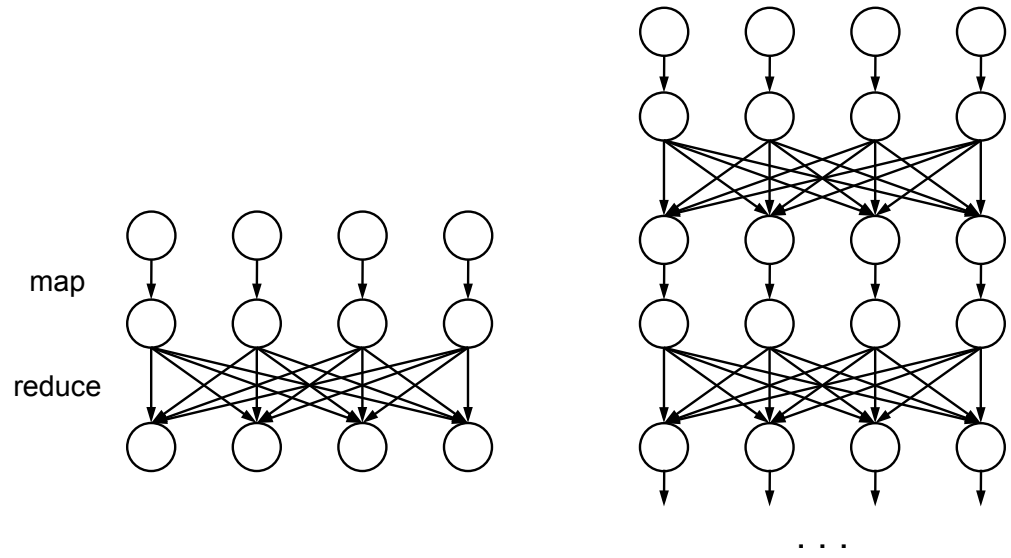
The RDD architecture and Spark system address both of these limitations. For the data sharing component, RDDs make data sharing fast by avoiding replication, and can closely emulate the in-memory "data sharing" across time that would happen in a real distributed system composed of long-running processes. For the latency component, Spark shows that it is possible to run MapReduce computations on 100+ node commodity clusters with 100 ms latency—nothing intrinsic to the MapReduce model prevents this. While some applications will need finer-grained time steps and communication, this 100 ms latency is enough to implement many data-intensive computations, where the amount of computation that can be batched before a communication step is high.

5.2.2 Lineage and Fault Recovery

One interesting property of the RDD-based emulation above is that it also provides fault tolerance. In particular, the RDD at each step of the computation only has a *constant-size* lineage over the previous step, which means that it is inexpensive to store the lineage and perform fault recovery.

Recall that we are emulating a distributed system composed of multiple single-node processes that exchange messages by running a fixed number of steps of the

[1] In particular, we consider work-preserving emulation [88], where the total time on the emulating machine is within a constant multiple of the emulated one, assuming they have the same number of processors.



(a) Single MapReduce step. (b) Multi-step communication.

Figure 5.1. Emulating an arbitrary distributed system with MapReduce. MapReduce provides primitives for local computation and all-to-all communication (a). By chaining these steps together, we can emulate any distributed system (b). The main costs to this emulation will be the latency of the rounds and the overhead of passing state across steps.

local event-processing loop of each process in a "map" function (having its old state as input and new state as output), and then exchanging messages between timesteps with a "reduce" function. As long as the program counter of each process is stored in its state, these map and reduce functions are *the same* on every timestep: they just read the program counter, received messages, and state, and emulate process execution. They can thus be encoded in constant space. While adding the program counter as part of the state might introduce some overhead, it is usually a small fraction of the size of the state, and the state sharing is just local to one node.[2]

By default, the lineage graph for the emulation above might make it somewhat expensive to recover state, because each step adds a new all-to-all shuffle dependency. However, if the application exposes the semantics of the computation more precisely (*e.g.,* indicating that some steps only create narrow dependencies), or subdivides the work of each node into multiple tasks, we can also parallelize recovery across the cluster.

One final concern with the RDD-based emulation of a distributed system is the cost of maintaining multiple versions of state locally as transformations apply to

[2] This argument most easily applies to systems that do not receive input from outside once started. If the system does receive input, we need this input to be stored reliably and divided into timesteps the same way we handled it in discretized streams (Chapter 4), but this requirement is already necessary if a system wants to react to input reliably (*i.e.,* not lose any piece of input if a node goes down).

them, and maintaining copies of outgoing messages, in order to facilitate recovery. This is a nontrivial cost, but in many cases, we can limit it by checkpointing state asynchronously once in a while (*e.g.*, if the available checkpointing bandwidth is 10× lower than memory bandwidth, we might checkpoint every 10 steps) or by storing deltas between different versions of state (as discussed in Section 3.2.3). As long as the amount of "fast" storage on each machine is enough to store outgoing messages and a few versions of state, we can achieve similar performance to the original system.

5.2.3 Comparison with BSP

As a second example of the generality of MapReduce and RDDs, we note that the "local computation and all-to-all communication" model discussed above very closely matches Valiant's Bulk Synchronous Parallel (BSP) model [108]. BSP is a "bridging model" designed to capture the most salient aspects of real hardware (namely that communication has latency, and synchronization is expensive) while being simple enough to analyze mathematically. Thus, not only has it been used directly to design some parallel algorithms, but its cost factors (namely the number of communication steps, amount of local computation on each step, and amount of data communicated by each processor on each step) are natural factors to optimize in most parallel applications. In this sense, we can expect that algorithms that map well to BSP can be evaluated efficiently with RDDs.

Note that this emulation argument for RDDs can thus apply to distributed runtimes based on BSP, such as Pregel [72], as well. The RDD abstraction adds two benefits over Pregel. First, Pregel as described in Google's paper [72] only supports checkpoint-and-rollback fault recovery of the whole system, which scales inefficiently as systems grow and failures become more common. The paper does describe a "confined recovery" mode in development that logs outgoing messages and recovers lost state in parallel, similar to parallel recovery in RDDs. Second, because RDDs have an iterator-based interface, they can more efficiently support pipelining between computations written as different libraries, which is useful in composing programs. More generally, from a programming interface point of view, we found that RDDs offer the right mix of letting users work with high-level abstractions (*e.g.*, divide state into multiple co-partitioned datasets, or set up patterns of narrow and wide dependencies that aren't all-to-all communication on each step) while retaining a simple common interface that enables the data sharing discussed above.

5.3 Systems Perspective

Completely independent of the emulation approach to characterizing RDDs' generality, we can take a systems approach: what are the bottleneck resources in

most cluster computations, and can RDDs tackle them efficiently? In this light, the clearest bottleneck in most cluster applications is the communication and storage hierarchy. We show that the partitioning and locality features in RDDs give applications enough control to emulate common optimizations for these resources, leading to similar performance in many applications.

5.3.1 Bottleneck Resources

Although cluster applications are diverse, they are all bound by the same properties of the underlying hardware. Current datacenters have a very steep storage hierarchy, which will limit most applications in similar ways. For example, a typical datacenter today might have the following hardware characteristics:

- Each node has local memory with around 50 GB/s of memory bandwidth, as well as multiple disks—typically 12-24 in a Hadoop cluster [80], which would mean around 2 GB/s bandwidth assuming 20 disks at 100 MB/s each.
- Each node has a 10 Gbps (1.3 GB/s) outlink, which is about 40× smaller than memory bandwidth and 2× smaller than its aggregate disk bandwidth.
- Nodes are organized into racks of 20-40 machines, with 20–40 Gbps bandwidth out of each rack, or 10× lower than the in-rack network performance.

Given these properties, the most important performance concern is in many applications is to control network placement and communication. Fortunately, RDDs provide the facilities for this: the interface leaves room for runtimes to place computations near input data, as in the map tasks in MapReduce (in fact RDDs explicitly have a "preferred locations" API in our definition in Section 2.4), and RDDs provide control over data partitioning and co-location. Unlike data sharing in MapReduce, which always implicitly has to go over the network, RDDs do not cause network traffic unless the user explicitly invokes a cross-node operation or checkpoints a dataset.

From this perspective, if most applications are network-bandwidth-bound, then local efficiency within a node (*e.g.*, data structures or CPU overhead) matters less than controlling communication. In our experience, many Spark applications, especially if the data fits in memory, are network-bandwidth-bound. When data does not fit in memory, applications are I/O-bound, and data locality in scheduling is the most important factor. And applications that are CPU-bound are generally easier to execute overall (*e.g.*, many of them would work well on vanilla MapReduce). Much like the discussion in the previous section, the only area where RDDs clearly add a cost is network latency, but our work in Spark shows that this latency can be made sufficiently small for many applications if one optimizes the MapReduce engine for this metric, even small enough to support streaming.

5.3.2 The Cost of Fault Tolerance

One final observation from a systems perspective is that RDD-based systems incur extra costs over some of today's specialized systems due to fault tolerance. For example, in Spark, the "map" tasks in each shuffle shuffle operation save their output to the local filesystem on the machine where they ran, so that "reduce" tasks can re-fetch it later. In addition, Spark (like the original MapReduce) implements a barrier at shuffle stages, so that the reduce tasks do not start until all the mappers have finished. This avoids some of the complexity of needed for fault recovery if one "pushed" records directly from the mappers to the reducers in a pipelined fashion.[3]

Even though removing some of these inefficiencies would speed up the system, and we plan to do so in future work, Spark often performed competitively despite them. The main reason is an argument similar to the one in the previous section: many application are bound by an external I/O operation, *e.g.,* shuffling a large amount of data across the network or reading it from disk, and beyond this optimizations such as pipelining only add a modest benefit. For example, consider the benefit of pushing data directly from mappers to reducers in a shuffle instead of having a barrier to wait for all the maps to finish before launching reducers: at best, if the CPU time of the mappers overlapped exactly with the network transfer time of the shuffle, this would speed the job by 2×. This is certainly a useful optimization but not as crucial as placing the map tasks close to input data and avoiding replication of intermediate state. Furthermore, if any other component of the running time dominates (*e.g.,* the mappers spend a long time reading from disk, or the shuffle is much slower than the computation), the benefit decreases.

Finally, we note that even if faults *per se* do not occur often, a Spark and MapReduce-like design that divides work into fine-grained, independent tasks has other important benefits in a cluster setting. First, it also allows mitigating stragglers, which is more complicated in a traditional push-based data flow design (similar to our comparison of D-streams with continuous operators in Chapter 4). Stragglers are more common than faults even on small clusters, especially in virtualized settings [120]. Second, the independent task model facilitates *multitenancy*: multiple applications from different users can dynamically share resources, leading to interactive performance for all users. Much of our work parallel to programming models has been to enable dynamic resource sharing between cluster users, as large clusters are inevitably multi-user and all the applications in these clusters need to achieve fast placement and data locality [117, 56, 116]. For these reasons, a fine-grained task design might lead to *better* performance for most users in a multi-user setting, even if performance on a single application is worse.

Given these observations about the cost of fault tolerance and other benefits of the independent task model, we do believe that cluster system designers should

[3] There are other benefits to this as well, especially in a multi-user setting, as it avoids committing some machines to run reduce tasks only to find out later that they are mostly idle if the maps are not producing data quickly [116].

consider making fault tolerance and elasticity a first-class aspect even when only targeting short job workloads. Systems that provide these properties will be able to scale more easily to large queries as well, and to multi-user settings.

5.4 Limitations and Extensions

While the previous section characterized situations in which RDDs can emulate a distributed system efficiently, it also highlighted cases when they do not. We now survey some of the key limitations and discuss several possible extensions to the model that would bypass them.

5.4.1 Latency

As explained in the previous sections, the main performance metric in which RDD-based emulation of a distributed system can fall behind the real system is latency. Because RDD operations are deterministic and synchronized across the whole cluster, there is inherently more latency in launching each "timestep" of the computation, so computations with many timesteps will be slower.

Two examples of such applications are very low-latency streaming systems (*e.g.*, with millisecond-scale requirements) and fine-grained time-stepped simulation (*e.g.*, scientific simulations). While we showed that in practice, RDD operations can have a latency as low as 100 ms, this may not be enough for some applications. In streaming settings, the latency is likely low enough to track events at a "human" time-scale (*e.g.*, trends in web click rates or social media), and to match the wide-area latency of the Internet. In simulation applications, recent work on jitter-tolerant execution applies well to RDDs [121]. This work simulates multiple timesteps in a grid area locally on each machine, taking advantage of the fact that information takes several steps to move across grid cells in most simulations, and leaving more time to synchronize with other nodes. It was designed specifically to handle straggler-prone cloud environments.

5.4.2 Communication Patterns Beyond All-to-All

Section 5.2.1 showed that MapReduce and RDDs can emulate a distributed system where the nodes do point-to-point communication between any pair of nodes, using the reduce operation. While this is a common scenario, real networks may also have other efficient primitives, like broadcast or in-network aggregation. Some data-intensive applications benefit significantly from these (*e.g.*, when broadcasting the current model in a machine learning algorithm or collecting back results). These primitives are usually costly to emulate with just point-to-point message passing, so extending RDDs with direct support for them would help. For example, Spark

already contains an efficient "broadcast" operation that can be implemented with a variant of BitTorrent [30].

5.4.3 Asynchrony

RDD operations such as an all-to-all *reduce* are synchronous to provide determinism. This can slow computations down when work is imbalanced across nodes or some nodes are slower. Recently, a number of cluster computing systems have been proposed that let nodes send messages *asynchronously* so that the bulk of the computation can continue even if some nodes are slow [71, 48, 32]. It would be interesting to explore whether an RDD-like model could support such operations while still preserving fault recovery. For example, nodes might reliably log which message IDs they received on each iteration of the computation, which might still be less expensive than logging the messages themselves. Note that for general computations, it is usually not useful to rebuild a different state from the one lost on failure, because subsequent computations on the lost nodes might have used the lost state.

Some algorithms, such as statistical optimization, can also continue even from corrupted state without completely losing all progress [71]. In this case, an RDD-based system could run these algorithms in a special mode that does not perform recovery, but checkpoint the result to allow later computation on it (*e.g.*, interactive queries) to see a consistent result.

5.4.4 Fine-Grained Updates

RDDs are not efficient when a user performs many fine-grained update operations on them, as the cost of logging lineage for each operation is high. For example, they would not be a good fit for implementing a distributed key-value store. In some cases it may be possible to *batch* operations together and execute multiple updates as a single coarse-grained operation, similar to how we "discretized" stream processing in D-streams.[4] While this is still unlikely to be low enough latency for a key-value store in the current Spark system, it would be interesting to implement this model in a lower-latency system. This approach is similar to the Calvin distributed database [103], which batches transactions and executes them deterministically to achieve better wide-scale performance and reliability.

5.4.5 Immutability and Version Tracking

As discussed earlier in this chapter, immutability may add overhead as more data needs to be copied in an RDD-based system. The main reason RDDs were

[4] Note that in this case the sequence of updates still needs to be replicated for reliability, but this is the case in any reliable key-value store, where write operations need to be replicated.

designed to be immutable is to track dependencies between different versions of a dataset and recover state that relied on old versions, but it would also be possible for a runtime to track these dependencies in another way and use mutable state under the RDD abstraction. This would be an especially interesting addition when dealing with streaming or fine-grained updates. As discussed in Section 3.2.3, there are also ways for users to manually divide state into multiple RDDs or reuse old date using persistent data structures.

5.5 Related Work

Several other projects have sought to generalize MapReduce to support more computations efficiently. For example, Dryad [61] extends the model to arbitrary task DAGs, and CIEL [77] allows tasks to modify the DAG at runtime (*e.g.*, spawn other tasks) based on the data they read. However, to our knowledge, none of these research efforts has tried to formally qualify the power of its model (*e.g.*, by emulating general parallel machines), beyond showing that some applications can run faster.

Our work in this chapter is closely tied to research on on parallel machine models, such as PRAM [42], BSP [108], QSM [46] and LogP [35]. Much of this work has focused on defining new models that are more similar to real machines, and determining which models can emulate one another. For example, BSP was shown to be able to simulate PRAM within a constant factor if there is enough parallel slack [108], and QSM, BSP and LogP have been shown to emulate each other within polylog factors [88]. In this context, we have shown that RDDs can efficiently implement the BSP model and, more informally, will mostly increase the network latency seen by an application (*e.g.*, in a LogP model). Informally, this means that many algorithms designed for these models are likely to work well with RDDs.

One interesting aspect of using RDDs to implement these computations is that they are fault-tolerant, and can parallelize the recovery over multiple nodes if there is parallel slack (*i.e.*, multiple tasks running on the same real processor). The only work we are aware of that does this in more traditional computing models is by Savva and Nanya [91], which shows how to implement a shared memory model over BSP fault-tolerantly, but do so by duplicating the contents of memory.

More recently, several theoretical articles have looked at complexity measures and bounds on MapReduce [40, 65, 49, 90, 47]. These works largely focus on what can be computed with a small number of MapReduce steps, acknowledging the cost of launching a new step. Several of them also show that MapReduce can emulate BSP or PRAM computations [65, 49]. Because RDDs are a practical computing abstraction that makes multi-round computation less expensive, it may be interesting to see whether these analyses change if we use them.

Finally, other current computing systems can implement BSP, including

Pregel [72], iterative MapReduce models [37, 22], and GraphLab [71]. However, to our knowledge, these systems have not been used directly to emulate general computations.

5.6 Summary

This section explored why RDDs were able to be a general programming model from two perspectives: expressiveness in terms of emulating other distributed systems, and a practical perspective in terms of letting users control the key performance factors in a cluster setting. In the first perspective, we showed that RDDs can emulate *any* distributed system, and only primarily add a cost in network latency, which is tolerable in many applications. From the second perspective, RDDs were successful because they let users control the most same aspects of network and data placement that previous specialized systems optimized. Using the properties of RDDs, we also obtained possible extensions to the model.

Chapter 6

Conclusion

How should we design computing platforms for the new era of massively parallel clusters? This dissertation shows that the answer can, in many cases, be quite simple: a single abstraction for computation, based on coarse-grained operations with efficient data sharing, can achieve state-of-the-art performance in a wide range workloads, sometimes offering properties that previous systems lack.

While cluster computing systems will undoubtedly evolve, we hope that the RDD architecture proposed here can offer, at the very least, a useful reference point. The Spark engine is currently only 34,000 lines of code (the first published version was 14,000), and other models built on it are an order of magnitude smaller than standalone systems. In an area where application requirements are fast evolving, we believe that this kind of small, general abstraction over the difficult parts of cluster computing (fault tolerance, scheduling, multitenancy) can enable rapid innovation.

Even if the RDD model is not adopted for its performance, we believe that its main contribution is to enable previously disparate cluster workloads to be *composed*. As cluster applications grow in complexity, they will need to combine different types of computation (*e.g.,* machine learning and SQL) and processing modes (*e.g.,* interactive and streaming). Combining these computations with today's specialized models is highly expensive due to the cost to share data between systems. RDDs avoid this cost by enabling lineage-based fault recovery *across* systems, achieving memory-speed data sharing. And at the cluster level, fine-grained execution allows RDD-based applications to efficiently coexist, improving productivity for all users. In real deployments, the efficiency gains from composition and multitenancy can outweigh performance on any individual computation.

In the rest of this chapter, we summarize a few of the lessons that influenced this work. We then discuss some of its impact in industry. Finally, we sketch areas for future work.

6.1 Lessons Learned

The importance of data sharing. The main thread underlying our work is how important data sharing is to performance, both within a single computation (*e.g.*, an iterative algorithm or streaming job) and across computations in applications that compose them. For "big data" applications in particular, datasets are very expensive to move, so sharing them efficiently is crucial for application developers. Whereas previous systems have mostly focused on enabling specific data flow patterns, RDDs make datasets a first-class primitive and offer users enough mechanisms to control their properties (*e.g.*, partitioning and persistence) while keeping the interface abstract enough to automatically provide fault tolerance.

Because of the continuing gaps between network bandwidth, storage bandwidth and computing power on each machine, we believe that data sharing will remain a concern in most distributed applications, and parallel processing platforms will need to address it.

Value performance in a shared setting over single-application. While it is rewarding to optimize execution engines for specific applications, another lesson from our work is that real-world deployments are often more complex, and it is performance in these complex settings that matters. In particular:

- Most workflows will *combine* processing of different types, *e.g.*, parsing a log file using MapReduce and then running a machine learning algorithm on the result.
- Most deployments will be *shared* among multiple applications, requiring execution models that behave well under dynamic resource sharing, revocation, and stragglers.

For example, suppose that a specialized implementation of a machine learning algorithm, using a more rigid execution model like MPI (where the application has a static allocation for its whole duration), was 5× faster than a Spark implementation. The specialized system might still be slower in an *end-to-end* workflow that involves parsing a data file with a MapReduce script and then running the learning algorithm, because it would require an extra export of the parsed dataset to a reliable storage system to share it between systems. And in a multi-user cluster, the specialized system would require choosing a fixed allocation for the application in advance, leading to either queueing of applications or underutilization, and decreased responsiveness for all users of the cluster compared to a fine-grained task execution model like RDDs.[1]

[1] Although not covered in this dissertation, the author has also worked on a number of scheduling algorithms in the fine-grained task model to show that efficient and dynamic resource sharing can actually be achieved [117, 45, 56].

We believe that due to the same point as in our first lesson above (that data is expensive to move), clusters will continue to be shared dynamically, requiring applications to scale up and down responsively and take turns accessing data on each node. In these types of environments, we believe that computer systems will have to be optimized *for the shared case* to provide tangible performance benefits in most deployments.

Optimize the bottlenecks that matter. One interesting lesson in how to design general processing engines is to look at bottlenecks. In many cases, a few resources ultimately limit the performance of the application, so giving users control to optimize these resources can lead to good performance. For example, when Cloudera released its Impala SQL engine, the Berkeley AMPLab compared it to Shark and found that, in many queries, the performance was nearly identical [111]. Why is that? These queries were either I/O- or network-bound, and both systems saturated the available bandwidth.

This is an interesting way to approach generality because it means that general abstractions do not need to be low-level. For example, RDDs gave users the ability to optimize network usage (the most common bottleneck) through controlled partitioning. But they do this in a way that enables common patterns (*e.g.*, co-partitioning) without requiring users to choose manually which machine each piece of data is on, and can thus automatically handle rebalancing and fault tolerance.

Simple designs compose. Finally, part of what made both Spark and RDDs so general is that they were built on a small set of core interfaces. At the interface level, RDDs distinguish between only two types of transformations (narrow and wide). During execution, they pass data through a single, standard interface (an iterator over records), allowing efficient intermixing of different data formats and processing functions. And for the purpose of task scheduling, they map down to a very simple model (fine-grained, stateless tasks) for which a wide set of algorithms have been developed to provide fair resource sharing [45], data locality [117], straggler mitigation [120, 8], and scalable execution [83]. These scheduling algorithms *themselves* compose with each other, partly due to the simplicity of the fine-grained task model. The result is a system that can benefit from both the latest parallel processing algorithms and the most powerful runtime algorithms for executing these computations.

6.2 Broader Impact

Since Spark was released in 2010, it has become a growing open source community, with over 100 developers from 25 companies contributing to the project. The higher-level projects in the Spark stack, such as Shark, have also attracted significant contributions and interest (for instance, 29 developers have contributed to

Shark since 2011). These external developers have contributed numerous important features to the core engine, as well as ideas and use cases that continue to drive the design of the project. For an overview of some Spark industry use cases, we invite the reader to look at the presentations from the 2013 Spark Summit [97].

6.3 Future Work

As described in Section 5.4, our analysis of RDDs also shows limitations of the model that might be interesting to study for further generalizations. To reiterate that section, the main areas for extensions are:

- **Communication latency:** the main drawback in RDD-based emulation of an arbitrary distributed system is the additional latency to synchronize each time step and make the computation deterministic. There are two interesting lines of future work here. The first would be a systems challenge of seeing how low the latency for an in-memory cluster computing system can actually be—newer datacenter networks can have latency on the order of microseconds, and for an optimized codebase, it is plausible that the timestep latency could be just a few milliseconds. (In the current JVM-based Spark system, the Java runtime makes it higher.) The second line of work would be using latency hiding techniques [121] to execute even tightly synchronized applications over RDDs by dividing work across blocks or speculating the responses of other machines.

- **New communication patterns:** RDDs currently only provide the point-to-point "shuffle" pattern for communication across nodes, but there are parallel computations that benefit from other patterns, such as broadcast or all-to-one aggregation. Exposing these patterns could improve application performance and create opportunities for new runtime optimizations and fault recovery schemes.

- **Asynchrony:** Although RDD-based computations are synchronized and deterministic, it may be possible to fit in asynchronous computation steps within the model while preserving some fault recovery guarantees between them.

- **Fine-grained updates:** RDDs excel at coarse-grained, data-parallel operations, but as discussed in Section 5.4.4, it would also be possible to make them emulate a system with fine-grained operations, such as reads and writes to a key-value store, by grouping such updates into batches. Especially with a lower-latency runtime, it would be interesting to see how far this approach can go compared to a traditional database design, and what benefits it can offer by allowing transactional and analytics workloads to coexist.

- **Version tracking:** RDDs are defined as immutable datasets to allow lineage dependencies to point to specific versions, but there is significant room to use

mutable storage and more efficient version-tracking schemes underneath this abstraction.

In addition to these ways to extend the RDD programming model, our experience with Spark also points to practical systems concerns that users face that can make for interesting future research. Some important areas are:

- **Correctness debugging:** Distributed applications are complicated to debug and to test for correctness, especially when they operate on large, uncurated datasets. One idea we have explored within Spark is to use RDDs' lineage information to enable efficient *replay debugging* of parts of the application (*e.g.*, just the tasks that crashed with an exception, or just the portion of the execution graph that led to a particular output). The same tool could also modify the lineage graph on a second run, *e.g.*, to add logging code to user functions or augment records with taint tracking.

- **Performance debugging:** The most common debugging inquiries on the Spark mailing list are about performance instead of correctness. Distributed applications are very difficult to tune, partly because users have much less intuition about what makes good performance. If a PageRank implementation on a 100 GB dataset takes 30 minutes on a 5-node cluster, is this performance good? Could the application have taken 2 minutes, or 10 seconds? Factors such as communication cost, memory overhead of various data representations, and data skew can significantly impact parallel applications. It would be interesting and challenging to develop tools that can automatically detect these inefficiencies, or even monitoring tools that show a user enough about how an application performs to identify problems.

- **Scheduling policies:** Although the RDD model supports very flexible runtime placement through fine-grained tasks, and this has already been used to implement mechanisms such as fair sharing, finding the right policies for applications written in this model remains an open challenge. For example, in a Spark Streaming application where there are multiple streams, how can we schedule computations to enforce deadlines or priorities? What if the same application is also performing interactive queries on the streams? Similarly, given a graph of RDDs or D-Streams, can we automatically determine which to checkpoint to minimize expected execution time? As applications grow more complex and users start demanding more responsive interfaces, these policies will be important to maintain good performance.

- **Memory management:** Allocating the limited memory in most clusters is an interesting challenge that also depends on application defined priorities and usage patterns. The problem is especially interesting because there are different "levels" of storage possible that trade off memory size versus access speed. For example, data in RAM can be compressed, which might make it more expensive to compute upon, but smaller; or it can be spilled to an

SSD, or a spinning disk. Some RDDs might be more efficient to not persist at all, but rather recompute on-the-fly by applying a map function on the previous RDD (the result might be fast enough to compute for most users that it would be worth saving space). Some RDDs might be so expensive to compute that they should always be replicated. Particularly because RDDs can always recompute lost data from scratch, they leaves significant room for rich storage management policies.

We hope that continued experience with the open source Spark system will help us address these challenges, and lead to solutions that are applicable both within Spark and to other cluster computing systems.

Bibliography

[1] Azza Abouzeid et al. HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads. *VLDB*, 2009.

[2] Tyler Akidau, Alex Balikov, Kaya Bekiroglu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. MillWheel: Fault-tolerant stream processing at internet scale. In *VLDB*, 2013.

[3] 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, Ying Li, V. Di Nicola, X. Wang, David Maier, S. Grell, O. Nano, and I. Santos. Microsoft CEP server and online behavioral targeting. *Proc. VLDB Endow.*, 2(2):1558, August 2009.

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

[5] Amazon Redshift. `http://aws.amazon.com/redshift/`.

[6] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. Disk-locality in datacenter computing considered irrelevant. In *HotOS '11*, 2011.

[7] Ganesh Ananthanarayanan, Ali Ghodsi, Andrew Wang, Dhruba Borthakur, Srikanth Kandula, Scott Shenker, and Ion Stoica. Pacman: Coordinated memory caching for parallel jobs. In *NSDI'12*, 2012.

[8] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. Reining in the outliers in map-reduce clusters using mantri. In *OSDI'10*, 2010.

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

[10] Apache Giraph. `http://giraph.apache.org`.

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

[12] Apache Mahout. `http://mahout.apache.org`.

[13] Apache Spark. `http://spark.incubator.apache.org`.

[14] Apache Storm. `http://storm-project.net`.

[15] Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Itaru Nishizawa, Justin Rosenstein, and Jennifer Widom. STREAM: The Stanford stream data management system. In *SIGMOD*, 2003.

[16] Ron Avnur and Joseph M. Hellerstein. Eddies: continuously adaptive query processing. In *SIGMOD*, 2000.

[17] Shivnath Babu. Towards automatic optimization of MapReduce programs. In *SoCC'10*, 2010.

[18] Magdalena Balazinska, Hari Balakrishnan, Samuel R. Madden, and Michael Stonebraker. Fault-tolerance in the Borealis distributed stream processing system. *ACM Trans. Database Syst.*, 2008.

[19] Pramod Bhatotia, Alexander Wieder, Rodrigo Rodrigues, Umut A. Acar, and Rafael Pasquin. Incoop: MapReduce for incremental computations. In *SOCC '11*, 2011.

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

[21] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. In *WWW*, 1998.

[22] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. HaLoop: efficient iterative data processing on large clusters. *Proc. VLDB Endow.*, 3:285–296, September 2010.

[23] Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Monitoring streams: a new class of data management applications. In *VLDB '02*, 2002.

[24] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *SIGMOD*, 2013.

[25] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. FlumeJava: easy, efficient data-parallel pipelines. In *PLDI*, 2010.

[26] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Sam Madden, Vijayshankar Raman, Fred Reiss, and Mehul Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[27] Biswapesh Chattopadhyay, Liang Lin, Weiran Liu, Sagar Mittal, Prathyusha Aragonda, Vera Lychagina, Younghee Kwon, and Michael Wong. Tenzing a SQL implementation on the MapReduce framework. In *Proceedings of VLDB*, 2011.

[28] James Cheney, Laura Chiticariu, and Wang-Chiew Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 1(4):379–474, 2009.

[29] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Donald Carney, Ugur Cetintemel, Ying Xing, and Stanley B. Zdonik. Scalable distributed stream processing. In *CIDR*, 2003.

[30] Mosharaf Chowdhury, Matei Zaharia, Justin Ma, Michael I. Jordan, and Ion Stoica. Managing data transfers in computer clusters with Orchestra. In *SIGCOMM*, 2011.

[31] Cheng T. Chu, Sang K. Kim, Yi A. Lin, Yuanyuan Yu, Gary R. Bradski, Andrew Y. Ng, and Kunle Olukotun. Map-reduce for machine learning on multicore. In *NIPS '06*, pages 281–288. MIT Press, 2006.

[32] James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Gregory R. Ganger, Garth Gibson, Kimberly Keeton, and Eric Xing. Solving the straggler problem with bounded staleness. In *HotOS*, 2013.

[33] J. Cohen, B. Dolan, M. Dunlap, J.M. Hellerstein, and C. Welton. Mad skills: new analysis practices for big data. *VLDB*, 2009.

[34] Tyson Condie, Neil Conway, Peter Alvaro, and Joseph M. Hellerstein. MapReduce online. *NSDI*, 2010.

[35] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. Logp: Towards a realistic model of parallel computation. *SIGPLAN Not.*, 28(7):1–12, July 1993.

[36] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[37] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: a runtime for iterative mapreduce. In *HPDC '10*, 2010.

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

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

[40] Jon Feldman, S. Muthukrishnan, Anastasios Sidiropoulos, and Cliff Stein. On distributing symmetric streaming computations. In *SODA*, 2009.

[41] Xixuan Feng et al. Towards a unified architecture for in-rdbms analytics. In *SIGMOD*, 2012.

[42] Steven Fortune and James Wyllie. Parallelism in random access machines. In *STOC*, 1978.

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

[44] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of SOSP '03*, 2003.

[45] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andrew Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: fair allocation of multiple resource types. In *NSDI*, 2011.

[46] Phillip B. Gibbons, Yossi Matias, and Vijaya Ramachandran. Can shared-memory model serve as a bridging model for parallel computation? In *SPAA*, pages 72–83, 1997.

[47] Ashish Goel and Kamesh Munagala. Complexity measures for map-reduce, and comparison to parallel computing. *CoRR*, abs/1211.6526, 2012.

[48] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *OSDI'12*, 2012.

[49] Michael T. Goodrich, Nodari Sitchinava, and Qin Zhang. Sorting, searching, and simulation in the MapReduce framework. In Takao Asano, Shin-ichi Nakano, Yoshio Okamoto, and Osamu Watanabe, editors, *Algorithms and Computation*, volume 7074 of *Lecture Notes in Computer Science*, pages 374–383. Springer Berlin Heidelberg, 2011.

[50] Pradeep Kumar Gunda, Lenin Ravindranath, Chandramohan A. Thekkath, Yuan Yu, and Li Zhuang. Nectar: automatic management of data and computation in datacenters. In *OSDI '10*, 2010.

[51] Zhenyu Guo, Xi Wang, Jian Tang, Xuezheng Liu, Zhilei Xu, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. R2: an application-level kernel for record and replay. In *OSDI*, 2008.

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

[53] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer Publishing Company, New York, NY, 2009.

[54] Bingsheng He, Mao Yang, Zhenyu Guo, Rishan Chen, Bing Su, Wei Lin, and Lidong Zhou. Comet: batched stream processing for data intensive distributed computing. In *SoCC*, 2010.

[55] Allan Heydon, Roy Levin, and Yuan Yu. Caching function calls using precise dependencies. In *ACM SIGPLAN Notices*, pages 311–320, 2000.

[56] Benjamin Hindman, Andrew Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy H. Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, 2011.

[57] Timothy Hunter, Teodor Moldovan, Matei Zaharia, Samy Merzgui, Justin Ma, Michael J. Franklin, Pieter Abbeel, and Alexandre M. Bayen. Scaling the Mobile Millennium system in the cloud. In *SOCC '11*, 2011.

[58] Jeong-Hyon Hwang, Magdalena Balazinska, Alexander Rasin, Ugur Cetintemel, Michael Stonebraker, and Stan Zdonik. High-availability algorithms for distributed stream processing. In *ICDE*, 2005.

[59] Jeong hyon Hwang, Ying Xing, and Stan Zdonik. A cooperative, self-configuring high-availability solution for stream processing. In *ICDE*, 2007.

[60] Cloudera Impala. `http://www.cloudera.com/content/cloudera/en/products-and-services/cdh/impala.html`.

[61] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys*, 2007.

[62] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *SOSP*, November 2009.

[63] Navin Kabra and David J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *SIGMOD*, 1998.

[64] Haim Kaplan. Persistent data structures. In Dinesh Mehta and Sartaj Sanhi, editors, *Handbook on Data Structures and Applications*. CRC Press, 1995.

[65] Howard Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for MapReduce. In *SODA*, 2010.

[66] Steven Y. Ko, Imranul Hoque, Brian Cho, and Indranil Gupta. On availability of intermediate data in cloud computations. In *HotOS '09*, 2009.

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

[68] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Eric Baldeschwieler, Scott Shenker, and Ion Stoica. Tachyon: Memory throughput I/O for cluster computing frameworks. In *LADIS*, 2013.

[69] Dionysios Logothetis, Christopher Olston, Benjamin Reed, Kevin C. Webb, and Ken Yocum. Stateful bulk processing for incremental analytics. In *SoCC*, 2010.

[70] Dionysios Logothetis, Chris Trezzo, Kevin C. Webb, and Kenneth Yocum. In-situ MapReduce for log processing. In *USENIX ATC*, 2011.

[71] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. Distributed GraphLab: A framework for machine learning and data mining in the cloud, April 2012.

[72] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, 2010.

[73] Nathan Marz. Trident: a high-level abstraction for real-time computation. `http://engineering.twitter.com/2012/08/trident-high-level-abstraction-for.html`.

[74] Frank McSherry, Derek G. Murray, Rebecca Isaacs, and Michael Isard. Differential dataflow. In *Conference on Innovative Data Systems Research (CIDR)*, 2013.

[75] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: interactive analysis of web-scale datasets. *Proc. VLDB Endow.*, 3:330–339, Sept 2010.

[76] Derek Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A timely dataflow system. In *SOSP '13*, 2013.

[77] Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. Ciel: a universal execution engine for distributed data-flow computing. In *NSDI*, 2011.

[78] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. S4: Distributed stream computing platform. In *Intl. Workshop on Knowledge Discovery Using Cloud and Distributed Computing Platforms (KDCloud)*, 2010.

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

[80] Kevin O'Dell. How-to: Select the right hardware for your new hadoop cluster. Cloudera blog, http://blog.cloudera.com/blog/2013/08/how-to-select-the-right-hardware-for-your-new-hadoop-cluster/.

[81] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John K. Ousterhout, and Mendel Rosenblum. Fast crash recovery in RAMCloud. In *SOSP*, 2011.

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

[83] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, low latency scheduling. In *SOSP '13*, 2013.

[84] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD '09*, 2009.

[85] Daniel Peng and Frank Dabek. Large-scale incremental processing using distributed transactions and notifications. In *OSDI*, 2010.

[86] Russel Power and Jinyang Li. Piccolo: Building fast, distributed programs with partitioned tables. In *Proc. OSDI 2010*, 2010.

[87] Zhengping Qian, Yong He, Chunzhi Su, Zhuojie Wu, Hongyu Zhu, Taizhi Zhang, Lidong Zhou, Yuan Yu, and Zheng Zhang. Timestream: Reliable stream computation in the cloud. In *EuroSys '13*, 2013.

[88] Vijaya Ramachandran, Brian Grayson, and Michael Dahlin. Emulations between qsm, bsp and logp: A framework for general-purpose parallel algorithm design. *J. Parallel Distrib. Comput.*, 63(12):1175–1192, December 2003.

[89] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, Inc., 3 edition, 2003.

[90] Anish Das Sarma, Foto N. Afrati, Semih Salihoglu, and Jeffrey D. Ullman. Upper and lower bounds on the cost of a map-reduce computation. In *VLDB'13*, 2013.

[91] A. Savva and T. Nanya. A gracefully degrading massively parallel system using the bsp model, and its evaluation. *Computers, IEEE Transactions on*, 48(1):38–52, 1999.

[92] Scala Programming Language. http://www.scala-lang.org.

[93] Mehul Shah, Joseph Hellerstein, and Eric Brewer. Highly available, fault-tolerant, parallel dataflows. *SIGMOD*, 2004.

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

[95] Jeff Shute, Radek Vingralek, Bart Samwel, Ben Handy, Chad Whipkey, Eric Rollins, Mircea Oancea, Kyle Littlefield, David Menestrina, Stephan Ellner, John Cieslewicz, Ian Rae, Traian Stancescu, and Himani Apte. F1: A distributed sql database that scales. *Proc. VLDB Endow.*, 6(11), August 2013.

[96] Spark machine learning library (MLlib). http://spark.incubator.apache.org/docs/latest/mllib-guide.html.

[97] Spark summit 2013. http://spark-summit.org/summit-2013/.

[98] Evan Sparks, Ameet Talwalkar, Virginia Smith, Xinghao Pan, Joseph Gonzalez, Tim Kraska, Michael I. Jordan, and Michael J. Franklin. MLI: An API for distributed machine learning. In *ICDM*, 2013.

[99] Utkarsh Srivastava and Jennifer Widom. Flexible time management in data stream systems. In *PODS*, 2004.

[100] Mike Stonebraker et al. C-store: a column-oriented dbms. In *VLDB'05*, 2005.

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

[102] Kurt Thomas, Chris Grier, Justin Ma, Vern Paxson, and Dawn Song. Design and evaluation of a real-time URL spam filtering service. In *IEEE Symposium on Security and Privacy*, 2011.

[103] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *SIGMOD '12*, 2012.

[104] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Antony, Hao Liu, and Raghotham Murthy. Hive-a petabyte scale data warehouse using Hadoop. In *ICDE*, 2010.

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

[106] Transaction Processing Performance Council. *TPC BENCHMARK H*.

[107] Tolga Urhan, Michael J. Franklin, and Laurent Amsaleg. Cost-based query scrambling for initial delays. In *SIGMOD*, 1998.

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

[109] Vinod K. Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache Hadoop YARN: Yet Another Resource Negotiator. In *SoCC*, 2013.

[110] Huayong Wang, Li-Shiuan Peh, Emmanouil Koukoumidis, Shao Tao, and Mun Choon Chan. Meteor shower: A reliable stream processing system for commodity data centers. In *IPDPS '12*, 2012.

[111] Patrick Wendell. Comparing large scale query engines. `https://amplab.cs.berkeley.edu/2013/06/04/comparing-large-scale-query-engines/`.

[112] Reynold S. Xin, Joseph E. Gonzalez, Michael J. Franklin, and Ion Stoica. Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*, GRADES '13, 2013.

[113] Reynold S. Xin, Josh Rosen, Matei Zaharia, Michael J. Franklin, Scott Shenker, and Ion Stoica. Shark: Sql and rich analytics at scale. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, 2013.

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

[115] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI '08*, 2008.

[116] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Job scheduling for multi-user mapreduce clusters. Technical Report UCB/EECS-2009-55, University of California, Berkeley, Apr 2009.

[117] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys 10*, 2010.

[118] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. Technical Report UCB/EECS-2011-82, University of California, Berkeley, 2011.

[119] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *SOSP*, 2013.

[120] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving MapReduce performance in heterogeneous environments. In *OSDI 2008*, December 2008.

[121] Tao Zou, Guozhang Wang, Marcos Vaz Salles, David Bindel, Alan Demers, Johannes Gehrke, and Walker White. Making time-stepped applications tick in the cloud. In *SOCC '11*, 2011.