Title
Go with the Flow: Graphs, Streaming and Relational Computations over Distributed Dataflow

Author
Xin, Reynold Shi

Publication Date
2018

Peer reviewed|Thesis/dissertation

Go with the Flow: Graphs, Streaming and Relational Computations over Distributed Dataflow

by

Reynold Shi Xin

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 Michael Franklin, Co-chair
Professor Ion Stoica, Co-chair
Professor Joseph Gonzalez
Professor Joshua Bloom

Spring 2018

Go with the Flow: Graphs, Streaming and Relational Computations over Distributed Dataflow

Reynold Shi Xin

Abstract

Go with the Flow: Graphs, Streaming and Relational Computations over Distributed Dataflow

by

Reynold Shi Xin

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Michael Franklin, Co-chair
Professor Ion Stoica, Co-chair

Modern data analysis is undergoing a "Big Data" transformation: organizations are generating and gathering more data than ever before, in a variety of formats covering both structured and unstructured data, and employing increasingly sophisticated techniques such as machine learning and graph computation beyond the traditional roll-up and drill-down capabilities provided by SQL. To cope with the big data challenges, we believe that data processing systems will need to provide fine-grained fault recovery across a larger cluster of machines, support both SQL and complex analytics efficiently, and enable real-time computation.

This dissertation builds on Apache Spark, a distributed dataflow engine, and creates three related systems: Spark SQL, Structured Streaming, and GraphX. Spark SQL combines relational and procedural processing through a new API called DataFrame. It also includes an extensible query optimizer to support a wide variety of data sources and analytic workloads. Structured Streaming extends Spark SQL's DataFrame API and query optimizer to automatically incrementalize queries, so users can reason about real-time stream data as batch datasets, and have the same application operate over both stream data and batch data. GraphX recasts graph specific system optimizations as dataflow optimizations, and provides an efficient framework for graph computation on top of Spark.

The three systems have enjoyed wide adoption in industry and academia, and together they laid the foundation for Spark's 2.0 release. They demonstrate the feasibility and advantages of unifying disparate, specialized data systems on top of distributed dataflow systems.

To my family

Contents

List of Figures

List of Tables

Acknowledgements

I would like to express my sincerest gratitude to my advisor, Michael Franklin, for his guidance and support during my PhD studies. He gave me enough freedom to pursue my own ideas, yet pushed gently enough to make sure I was focused and stayed on track. It is from the discussions with him that I learned what it meant to be a researcher: identifying problems, coming up with hypotheses, verifying them, disseminating knowledges, and making the world a better place.

The work in this dissertation started in a meeting I had with Ion Stoica and Matei Zaharia on porting Hive to run over Spark in 2011. These two amazing researchers have been my de facto co-advisors. Ion taught me to be relentless and think big, and this thesis would not be without Matei's initial work on Spark. Their emphasis on attacking real-world, practical problems and pursuit for simplicity have forever shaped my view towards not only research but also engineering.

Michael Armbrust has been my closest collaborator in the last four years. Although I can't think of another person with whom I have had more disagreements with over Scala (and SBT), these conflicting views have sparkled many of the ideas presented in this dissertation.

Through my work in Spark, I have been incredibly lucky to be able to work with a brilliant group of researchers and engineers. Together with Patrick Wendell, Ali Ghodsi, Xiangrui Meng, Tathagata Das, Josh Rosen, Davies Liu, Srinath Shankar, Shivaram Venkataraman, Kay Ousterhout, Joseph Gonzalez, Ankur Dave, and countless others, our work also drove Spark to what it is today.

I am also grateful for the collaborators and graduate students that helped shape not only my research but my whole graduate experience. An incomplete list of these amazing colleagues include: Alan Fekete, Alon Halevy, Donald Kossmann, Tim Kraska, Sam Madden, Scott Shenker, Sameer Agarwal, Sara Alspaugh, Peter Bailis, Neil Conway, Dan Crankshaw, Cliff Engle, Daniel Haas, Shaddi Hasan, Timothy Hunter, Haoyuan Li, David Moore, Aurojit Panda, Gene Pang, Evan Sparks, Liwen Sun, Beth Trushkowsky, Andrew Wang, Jiannan Wang, Eugene Wu, David Zhu. I would also like to thank Sam Lightstone and Renee Miller for bringing me into the database field in my undergraduate years.

Finally, it is not possible for me to adequately express how much credit my family deserve. They have been behind me and given me their unconditional support, even if that meant to sacrifice the time we spent together. I am forever indebted to their endless love, companionship, patience.

Chapter 1

Introduction

For the past few decades, data warehouses have been the primary data repositories for analytics. Different teams within an organization get together to define a data model based on their business requirements. Achieving consensus on the data model often requires months, if not years, of discussions. Once such model is created, live transactional data, presented in tabular forms, are extracted, transformed, and loaded (ETL) into data warehouses at regular intervals. Business analysts create reports and perform ad-hoc queries using SQL against the data warehouses. From a software architecture perspective, such data warehouses typically employ the Massively Parallel Processing (MPP) architecture, scaling often to only a few high-end physical machines.

1.1 Challenges of Big Data

With the rise of the Internet, e-commerce, and connected devices, modern data analysis is undergoing a "Big Data" transformation. Unlike what's described above, big data is defined by the following four key properties. The first three properties, volume, velocity, and variety, have been famously identified by Gartner as the "3Vs of Big Data" [5].

1. *Volume*: Data volumes are expanding drastically, creating the need to scale out both data storage and processing across clusters of hundreds, if not thousands, of commodity machines. Most MPP databases employ a coarse-grained recovery model, in which an entire query has to be resubmitted if a machine fails in the middle of a query. This approach works well for short queries when a retry is inexpensive, but faces significant challenges for long queries as clusters scale up [9].

2. *Velocity*: The velocity of data arriving is also increasing. Often without human in the loop, data can be generated and arrive at the speed of light. This opens up the opportunity for organizations to deploy systems that process data in

real-time. Nightly batch systems are no longer sufficient. In addition to the real-time nature of data, business requirements also change at a faster pace, making it difficult to create a commonly agreed upon data model.

3. *Variety*: Semi-structured and unstructured data, such as images, text, and voice, are becoming more common, while traditional data systems were built to handle primarily structured, tabular data.

4. *Complexity*: The complexity of analysis has also grown. Modern data analysis often employs sophisticated statistical methods, such as machine learning algorithms, that go well beyond the roll-up and drill-down capabilities of traditional database systems. While it may be possible to implement some of these functionalities using user-defined functions (UDFs) in MPP databases, these algorithms are often difficult to express and debug using UDFs. They are also computationally more expensive, exacerbating the need for systems to recover gracefully from machine failures or mitigate slowdowns that are common in large clusters.

To tackle the big data problem, the industry has created a new class of systems based on a distributed dataflow architecture. At the time this thesis first started, MapReduce [48] and Dryad [69] were the most prominent examples of such systems. They employ a fine-grained fault tolerance model suitable for large clusters, where tasks on failed or slow nodes can deterministically be re-executed on other nodes. These systems are also fairly general: [41] shows that these systems can express many statistical and machine learning algorithms, support unstructured data, and "schema-on-read" for greater flexibility.

However, dataflow engines lack many of the features that make MPP databases efficient, and thus exhibit high latencies of tens of seconds to hours, even on simple SQL queries. They are also designed primarily for batch analytics, and thus are unsuitable for real-time data processing. As such, most organizations tend to use these systems alongside MPP databases to perform complex analytics.

To provide an effective environment for big data analysis, processing systems will need to provide fine-grained fault recovery across a larger cluster of machines, support *both* SQL and complex analytics efficiently, and enable real-time computation. This dissertation develops three related systems: Spark SQL, Structured Streaming, and GraphX, that explore building effective systems for big data on top of a distributed dataflow engine.

1.2 Distributed Dataflow Frameworks

We use the term distributed dataflow framework to refer to cluster compute frameworks like MapReduce and its various generalizations. Although details vary from one framework to another, they typically satisfy the following properties:

1. a data model consisting of typed collections (*i.e.*, a generalization of tables to unstructured data).

2. a coarse-grained data-parallel programming model composed of deterministic operators which transform collections (*e.g.*, map, group-by, and join).

3. a scheduler that breaks each job into a directed acyclic graph (DAG) of tasks, where each task runs on a (horizontal) partition of data, and the edges in the graph indicate the input/output dependencies.

4. a runtime that can tolerate stragglers and partial cluster failures without restarting.

MapReduce is a special distributed dataflow framework. Its programming model exposes only two operators: *map* and *reduce* (a.k.a., group-by), and consequently each MapReduce job can contain at most two layers in its DAG of tasks. More modern frameworks such as DryadLINQ [129] and Spark [131] expose additional dataflow operators such as fold and join, and can execute tasks with multiple layers of dependencies. More generally, many shared-nothing parallel databases also satisfy the description of distributed dataflow frameworks when user-defined functions are employed, although databases typically have weaker fault-tolerance guarantees.

Distributed dataflow frameworks have enjoyed broad adoption for a wide variety of data processing tasks, including iterative machine learning. They have also been shown to scale to thousands of nodes operating on petabytes of data.

1.3 Apache Spark

In order to understand the projects and ideas in this thesis, it is important initially to examine the background of Apache Spark.

Spark is a distributed dataflow framework with APIs in Scala, Java and Python [20]. Spark was initially released in 2010 by the UC Berkeley AMPLab, and the project was donated to the Apache Software Foundation in 2013 (and thus "Apache Spark"). It has since become the most active open source project for big data processing, with over 1200 people [4] having contributed code to the project.

Spark has several features that are particularly relevant to the work in this dissertation:

1. Spark's storage abstraction called Resilient Distributed Datasets (RDDs) [131] enables applications to keep data in memory, which is essential for complex analytics such as machine learning algorithms and graph algorithms.

2. RDDs permit user-defined data partitioning, and the execution engine can exploit this to co-partition RDDs and co-schedule tasks to avoid data movement.

This primitive is essential for performance optimizations for both relational query processing and advanced analytics.

3. Spark logs the lineage of operations used to build an RDD, enabling automatic reconstruction of lost partitions upon failures. Since the lineage graph is relatively small even for long-running applications, this approach incurs negligible runtime overhead, unlike checkpointing, and can be left on without concern for performance. Furthermore, Spark supports optional in-memory distributed replication to reduce the amount of recomputation on failure.

4. Spark provides high-level programming APIs in Scala, Java, and Python that can be easily extended.

Next, we explain Resilient Distributed Datasets, the primary programming abstraction of Spark.

1.3.1 Resilient Distributed Datasets (RDDs)

Spark's main abstraction is *resilient distributed datasets* (RDDs), which are immutable, partitioned collections that can be created through various data-parallel *operators* (*e.g., map*, *group-by*, *hash-join*). Each RDD is either a collection of data stored in an external storage system, such as a file in HDFS [2], or a derived dataset created by applying operators to other RDDs. For example, given an RDD of (visitID, URL) pairs for visits to a website, we might compute an RDD of (URL, count) pairs by applying a *map* operator to turn each event into an (URL, 1) pair, and then a *reduce* to add the counts by URL.

In Spark's native API, RDD operations are invoked through a functional interface similar to DryadLINQ [129] in Scala, Java or Python. For example, the Scala code for the query above is:

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

RDDs can contain arbitrary data types as elements (since Spark runs on the JVM, these elements are Java objects), and are automatically partitioned across the cluster, but they are immutable once created, and they can only be created through Spark's deterministic parallel operators. These two restrictions, however, enable highly efficient fault recovery. In particular, instead of replicating each RDD across nodes for fault-tolerance, Spark remembers the *lineage* of the RDD (the graph of operators used to build it), and recovers lost partitions by *recomputing* them from base data [131].[1] For example, Figure 1.1 shows the lineage graph for the RDDs computed above. If Spark loses one of the partitions in the (URL, 1) RDD,

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

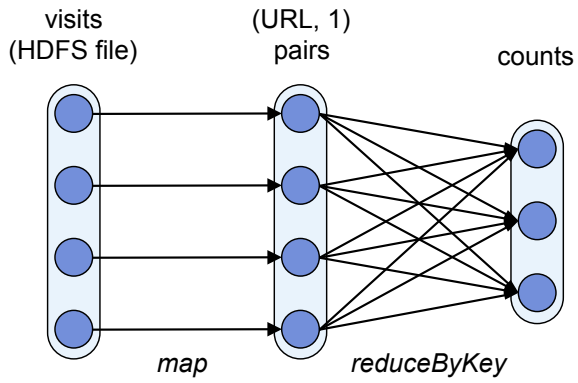


Figure 1.1: Lineage graph for the RDDs in our Spark example. Oblongs represent RDDs, while circles show partitions within a dataset. Lineage is tracked at the granularity of partitions.

for example, it can recompute it by rerunning the *map* on just the corresponding partition of the input file.

The RDD model offers several key benefits in our large-scale in-memory computing setting. First, RDDs can be written at the speed of DRAM instead of the speed of the network, because there is no need to replicate each byte written to another machine for fault-tolerance. DRAM in a modern server is over 10× faster than even a 10-Gigabit network. Second, Spark can keep just one copy of each RDD partition in memory, saving precious memory compared with a replicated system, since it can always recover lost data using lineage. Third, when a node fails, its lost RDD partitions can be rebuilt *in parallel* across the other nodes, allowing fault recovery.[2] Fourth, even if a node is just slow (a "straggler"), we can recompute necessary partitions on other nodes because RDDs are immutable so there are no consistency concerns with having two copies of a partition.

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

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

1.3.2 Fault Tolerance Guarantees

As data volume and analysis complexity increase, the runtime of data intensive programs also increases. As a result, it is more likely for stragglers or faults to occur in the course of a job. Spark provides the following fault tolerance properties:

1. Spark can tolerate the loss of *any set of worker nodes*. The execution engine will re-execute any lost tasks and recompute any lost RDD partitions using lineage.[3] This is true even within a query: Spark will rerun any failed tasks, or lost dependencies of new tasks, without aborting the query.
2. Recovery is parallelized across the cluster. If a failed node contained 100 RDD partitions, these can be rebuilt in parallel on 100 different nodes, quickly recovering the lost data.
3. The deterministic nature of RDDs also enables straggler mitigation: if a task is slow, the system can launch a speculative "backup copy" of it on another node, as in MapReduce [48].

The RDD model is expressive, fault-tolerant, and well-suited for distributed computation. In the next section, we discuss why the RDD model alone is not sufficient to capture the common analytic workloads.

1.4 Expanding Use Cases

As demonstrated in previous sections, Spark's RDD model was well suited for distributed computation. Immediately after Spark was open sourced, we saw organizations in the real world migrating their existing big data applications over from earlier generation systems such as Apache Hive [121]. As Spark's adoption grew, we started to see new applications that were previously less common. This section documents the challenges early users encountered and how we address them in this thesis.

We see primarily four classes of big data use cases among Spark users [44]:

Interactive Analysis: A common use case is to query data interactively. In this context, SQL is the standard language used by virtually all software tools and users. The lack of SQL support in Spark was a huge inhibitor, because only the most sophisticated users would be able to perform interactive analysis using Spark. Chapter 2 introduces Spark SQL and discusses executing SQL queries efficiently over Spark.

Extract, transform, load (ETL): One of the earliest and most popular use cases for Spark was to extract data from different data sources, join them, transform them,

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

and then load them into other data sources. This is also sometimes referred to as a data pipeline. These ETL jobs are typically developed and maintained by data engineers and employ custom code. Although they can sometimes be expressed in the form of SQL queries with user-defined functions, most ETL application developers apply modern software engineering techniques that make SQL as a programming language a poor fit due to the lack of proper IDEs, continuous integration and deployment tooling support.

While the RDD model provides a programmatic interface in Scala, the RDD abstraction does not differentiate the logical plan and the physical plan, making it difficult for Spark to optimize user programs. As a result, users often need to hand tune their applications for better performance. In addition to executing SQL queries, Chapter 2 shows how Spark SQL's query optimizer and execution engine can be extended to support a declarative, programmatic API called DataFrame that is more suitable for building data pipelines.

Streaming Processing: Many large-scale data sources operate in real time, including sensors, logs from mobile applications, and the Internet of Things. The RDD model was designed to capture static, batch computation. Spark Streaming [133] was the first attempt at extending the RDD model to support stream processing. Spark Streaming worked by chunking a stream of data into infinite sets of small batch datasets, and required users to implement their batch jobs and streaming jobs twice, using completely different APIs. This approach led to diverging semantics of users' batch and stream pipelines over time. Chapter 3 develops Structured Streaming, an extension to Spark SQL that automatically incrementalizes query plans, and thus enabling users to write their data pipelines once but operate on both batch and stream data.

Machine Learning and Graph Computation: As advanced analytics such as machine learning and graph computation become increasingly common, many specialized systems have been designed and implemented to support these workloads. These workloads, however, are only part of the larger analytic pipelines that often require distributed dataflow systems. Naively implementing these workloads on Spark leads to suboptimal performance that can be orders-of-magnitude slower than specialized systems. Chapter 4 presents GraphX, an efficient implementation of graph processing on top of Spark. Note that several systems to support machine learning on Spark have been developed, but are beyond the scope of this thesis. Interested readers are referred to [115] for details.

To summarize, this thesis explores designs that expand Spark to cover the aforementioned use cases efficiently and effectively. In order to accomplish that, a new relational engine is created to support both SQL and the DataFrame programming model. The same engine is extended to support incremental computation and stream processing. Last but not least, the end of the thesis develops graph computation on top of dataflow engine.

1.5 Summary and Contributions

This dissertation is organized as follows.

Chapter 2 develops Spark SQL, a novel approach to combine relational query processing with complex analytics. It unifies SQL with the DataFrame programming model popularized by R and Python, enabling its users to more effectively deal with big data applications that require a mix of processing techniques. It includes an extensible query optimizer called Catalyst to support a wide range of data sources and algorithms in big data. Spark SQL was initially open sourced and included in Spark in 2014, and has since become the most widely used component in Spark.

Chapter 3 develops Structured Streaming, an extension to Spark SQL that supports real-time and streaming applications. Specifically, Structured Streaming can automatically incrementalize queries against static, batch datasets to process streaming data. Structured Streaming is also designed to support end-to-end applications that combine streaming, batch, and ad-hoc analytics, and to make them "correct by default", with prefix consistency and exactly-once processing. Structured Streaming was open sourced in 2016, as part of Spark 2.0. We have observed multiple large-scale production use cases, the largest of which processes over 1PB of data per month.

Chapter 4 presents GraphX on Spark, a system created by this dissertation to support graph processing. Historically, graph processing systems evolved separately from distributed dataflow frameworks for better performance. GraphX is an example of unifying graph processing with distributed dataflow systems. By identifying the essential dataflow patterns in graph computation and recasting optimizations in graph processing systems as dataflow optimizations, GraphX can recover the performance of specialized graph processing systems within a general-purpose distributed dataflow framework. GraphX was merged into Spark in its 1.2 release. In Spark 2.0, it became the main graph processing system, in lieu of an older system called Bagel.

Each of the aforementioned sections also covers the related works.

Finally, Chapter 5 concludes and discusses possible areas for future work.

Chapter 2

Spark SQL: SQL and DataFrames

As discussed in the previous chapter, modern data analysis is undergoing a "Big Data" transformation, requiring data processing systems to support more than just SQL. This chapter develops Spark SQL, a new relational query processing system that combines SQL and the DataFrame programming API for complex analytics. We first describe Shark, our initial attempt at implementing SQL on top of Spark. We then describe Spark SQL's user-facing API and the core internals to support that API, followed by performance evaluation on the system. Finally, we discuss research applications and related work.

2.1 Introduction

Big data applications require a mix of processing techniques, data sources and storage formats. The earliest systems designed for these workloads, such as MapReduce, gave users a powerful, but low-level, procedural programming interface. Programming such systems was onerous and required manual optimization by the user to achieve high performance. As a result, multiple new systems sought to provide a more productive user experience by offering relational interfaces to big data. Systems like Pig, Hive, Dremel [98, 121, 88] and Shark (Section 2.2) all take advantage of declarative queries to provide automatic optimizations.

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

This chapter describes our effort to combine both models in Spark SQL, a major extension in Apache Spark [131]. Rather than forcing users to pick between a relational or a procedural API, however, Spark SQL lets users seamlessly intermix the two.

Spark SQL bridges the gap between the two models through two contributions. First, Spark SQL provides a *DataFrame API* that can perform relational operations on both external data sources and Spark's built-in distributed collections. This API is similar to the widely used data frame concept in R [108], but evaluates operations lazily so that it can perform relational optimizations. Second, to support the wide range of data sources and algorithms in big data, Spark SQL introduces a novel extensible optimizer called *Catalyst*. Catalyst makes it easy to add data sources, optimization rules, and data types for domains such as machine learning.

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

To support a wide variety of data sources and analytics workloads in Spark SQL, we designed an extensible query optimizer called Catalyst. Catalyst uses features of the Scala programming language, such as pattern-matching, to express composable rules in a Turing-complete language. It offers a general framework for transforming trees, which we use to perform analysis, planning, and runtime code generation. Through this framework, Catalyst can also be extended with new data sources, including semi-structured data such as JSON and "smart" data stores to which one can push filters (*e.g.*, HBase); with user-defined functions; and with user-defined types for domains such as machine learning. Functional languages are known to be well-suited for building compilers [123], so it is perhaps no surprise that they made it easy to build an extensible optimizer. We indeed have found Catalyst effective in enabling us to quickly add capabilities to Spark SQL, and since its release we have seen external contributors easily add them as well.

Spark SQL was released in May 2014, and is now the most actively developed component [4] in Apache Spark. Spark SQL has already been deployed in very large scale environments. For example, a large Internet company [67] uses Spark SQL to build data pipelines and run queries on an 8000-node cluster with over 100 PB of data. Each individual query regularly operates on tens of terabytes. In

addition, many users adopt Spark SQL not just for SQL queries, but in programs that combine it with procedural processing. For example, 2/3 of customers of Databricks Cloud, a hosted service running Spark, use Spark SQL within other programming languages. Performance-wise, we find that Spark SQL is competitive with SQL-only systems on Hadoop for relational queries. It is also up to 100× faster and more memory-efficient than naive Spark code in computations expressible in SQL.

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

More fundamentally, our work shows that MapReduce-like execution models can be applied effectively to SQL, and offers a promising way to combine relational and complex analytics.

2.2 Shark: The Initial SQL Implementation

In this section, we give an overview of Shark, the first implementation of SQL over Spark that attempted to combine relational query processing with complex analytics. Many of the ideas in Shark have been reimplemented and inspired major features in Spark SQL. Refer to [127] for more details on Shark.

2.2.1 System Overview

Shark is compatible with Apache Hive, enabling users to run Hive queries much faster without any changes to either the queries or the data. Thanks to its Hive compatibility, Shark can query data in any system that supports the Hadoop storage API, including HDFS and Amazon S3. It also supports a wide range of data formats such as text, binary sequence files, JSON, and XML. It inherits Hive's schema-on-read capability and nested data types [121].

Figure 2.1 shows the architecture of a Shark cluster, consisting of a single master node and a number of worker nodes, with the warehouse metadata stored in an external transactional database. When a query is submitted to the master, Shark compiles the query into operator tree represented as RDDs, as we shall discuss in the next subsection. These RDDs are then translated by Spark into a graph of tasks to execute on the worker nodes.

Cluster resources can optionally be allocated by a resource manager (*e.g.*, Hadoop YARN [2] or Apache Mesos [65]) that provides resource sharing and

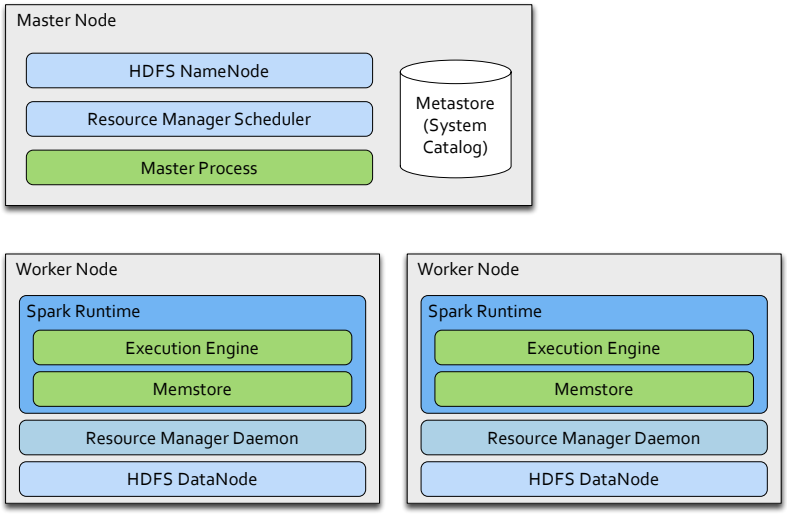


Figure 2.1: Shark Architecture

isolation between different computing frameworks, allowing Shark to coexist with engines like Hadoop.

2.2.2 Executing SQL over Spark

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

Given a query, Shark uses the Hive query compiler to parse the query and generate an abstract syntax tree. The tree is then turned into a logical plan and basic logical optimization, such as predicate pushdown, is applied. Up to this point, Shark and Hive share an identical approach. Hive would then convert the operator into a physical plan consisting of multiple MapReduce stages. In the case of Shark, its optimizer applies additional rule-based optimizations, such as pushing `LIMIT` down to individual partitions, and creates a physical plan consisting of transformations on RDDs rather than MapReduce jobs. We use a variety of operators already present in Spark, such as *map* and *reduce*, as well as new operators we implemented for Shark, such as broadcast joins. Spark's master then executes this graph using standard MapReduce scheduling techniques, such as placing tasks close to their input data, rerunning lost tasks, and performing straggler mitigation [131].

2.2.3 Engine Extensions

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

ETL. In addition, even with such a plan, naïvely executing it over Spark (or other MapReduce runtimes) can be inefficient. In this, we outline several extensions we made to Spark to efficiently store relational data and run SQL.

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. 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. The dynamic optimization is used to choose physical join execution strategies (broadcast join vs shuffle join) and to mitigate stragglers.

Columnar Memory Store: Shark implements a columnar memory store that encodes data in a compressed form using JVM primitive arrays. Compared with Spark's built-in cache, this store significantly reduces the space footprint overhead of JVM objects as well as speeding up garbage collection.

Data Co-partitioning: In some warehouse workloads, two fact 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. When joining two co-partitioned tables, Shark's optimizer constructs a DAG that avoids the expensive shuffle and instead uses map tasks to perform the join.

Partition Statistics and Map Pruning: Shark implements a form of data skipping called Map Pruning. The columnar memory store automatically tracks statistics (min value and max value) for each column for each partition. When a query is issued, the query optimizer uses this information to prune partitions that definitely do not have matches. It has been shown in [127] that this technique reduces the size of data scanned for queries by a factor of 30 in real workloads.

2.2.4 Complex Analytics Support

A key design goal of Shark is to provide a single system capable of efficient SQL query processing and complex analytics such as machine learning. 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.

As an example of Scala integration, Listing 2.1 illustrates a data analysis pipeline that performs logistic regression [63] over a user database using a combination of SQL and Scala.

```
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 2.1: Logistic Regression Example

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

The DataFrame API in Spark SQL was in part inspired by this functionality in Shark.

2.2.5 Beyond Shark

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

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

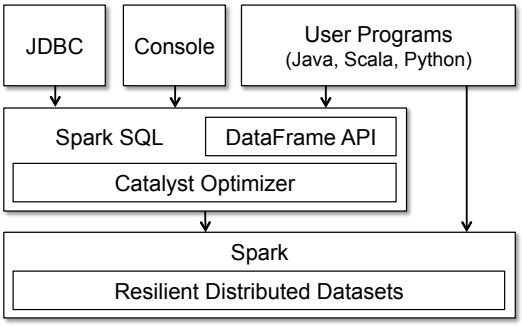


Figure 2.2: Interfaces to Spark SQL, and interaction with Spark.

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

The rest of this chapter describes the core components and innovations in Spark SQL that address these goals.

2.3 Programming Interface

Spark SQL runs as a library on top of Spark, as shown in Figure 2.2. It exposes SQL interfaces, which can be accessed through JDBC/ODBC or through a command-line console, as well as the DataFrame API integrated into Spark's supported programming languages. We start by covering the DataFrame API, which lets users intermix procedural and relational code. However, advanced functions can also be exposed in SQL through UDFs, allowing them to be invoked, for example, by business intelligence tools. We discuss UDFs in Section 2.3.7.

2.3.1 DataFrame API

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

[1] We chose the name DataFrame because it is similar to structured data libraries in R and Python, and designed our API to resemble those.

DataFrames can be constructed from tables in a system catalog (based on external data sources) or from existing RDDs of native Java/Python objects (Section 2.3.5). Once constructed, they can be manipulated with various relational operators, such as `where` and `groupBy`, which take expressions in a domain-specific language (DSL) similar to data frames in R and Python [108, 101]. Each DataFrame can also be viewed as an RDD of Row objects, allowing users to call procedural Spark APIs such as `map`.[2]

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

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

```
users = spark.table("users")
young = users.where(users("age") < 21)
println(young.count())
```

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

We next cover the details of the DataFrame API.

2.3.2 Data Model

Spark SQL uses a nested data model based on Hive [66] for tables and DataFrames. It supports all major SQL data types, including boolean, integer, double, decimal, string, date, and timestamp, as well as complex (i.e., non-atomic) data types: structs, arrays, maps and unions. Complex data types can also be nested together to create more powerful types. Unlike many traditional DBMSes, Spark SQL provides first-class support for complex data types in the query language and the API. In addition, Spark SQL also supports user-defined types, as described in Section 2.4.4.

Using this type system, we have been able to accurately model data from a variety of sources and formats, including Hive, relational databases, JSON, and native objects in Java/Scala/Python.

[2]These Row objects are constructed on the fly and do not necessarily represent the internal storage format of the data, which is typically columnar.

2.3.3 DataFrame Operations

Users can perform relational operations on DataFrames using a domain-specific language (DSL) similar to R data frames [108] and Python Pandas [101]. DataFrames support all common relational operators, including projection (`select`), filter (`where`), `join`, and aggregations (`groupBy`). These operators all take *expression* objects in a limited DSL that lets Spark capture the structure of the expression. For example, the following code computes the number of female employees in each department.

```
employees
  .join(dept, employees("deptId") === dept("id"))
  .where(employees("gender") === "female")
  .groupBy(dept("id"), dept("name"))
  .agg(count("name"))
```

Here, `employees` is a DataFrame, and `employees("deptId")` is an expression representing the `deptId` column. Expression objects have many operators that return new expressions, including the usual comparison operators (*e.g.*,`===` for equality test, `>` for greater than) and arithmetic ones (`+`, `-`, etc). They also support aggregates, such as `count("name")`. All of these operators build up an abstract syntax tree (AST) of the expression, which is then passed to Catalyst for optimization. This is unlike the native Spark API that takes functions containing arbitrary Scala/Java/Python code, which are then opaque to the runtime engine. For a detailed listing of the API, we refer readers to Spark's official documentation [20].

Apart from the relational DSL, DataFrames can be registered as temporary tables in the system catalog and queried using SQL. The code below shows an example:

```
users.where(users("age") < 21).registerTempTable("young")
ctx.sql("SELECT count(*), avg(age) FROM young")
```

SQL is sometimes convenient for computing multiple aggregates concisely, and also allows programs to expose datasets through JDBC/ODBC. The DataFrames registered in the catalog are still unmaterialized views, so that optimizations can happen *across* SQL and the original DataFrame expressions. However, DataFrames can also be materialized, as we discuss in Section 2.3.6.

2.3.4 DataFrames versus Relational Query Languages

While on the surface, DataFrames provide the same operations as relational query languages like SQL and Pig [98], we found that they can be significantly easier for users to work with thanks to their integration in a full programming language. For example, users can break up their code into Scala, Java or Python functions that pass DataFrames between them to build a logical plan, and will still benefit from optimizations across the *whole* plan when they run an output operation. Likewise, developers can use control structures like if statements and loops to structure their work.

To simplify programming in DataFrames, we also made Spark SQL analyze logical plans *eagerly* (*i.e.,* to identify whether the column names used in expressions exist in the underlying tables, and whether their data types are appropriate), even though query results are computed lazily. Thus, Spark SQL reports an error as soon as user types an invalid line of code instead of waiting until execution. This is again easier to work with than a large SQL statement.

2.3.5 Querying Native Datasets

Real-world pipelines often extract data from heterogeneous sources and run a wide variety of algorithms from different programming libraries. To interoperate with procedural Spark code, Spark SQL allows users to construct DataFrames directly against RDDs of objects native to the programming language. Spark SQL can automatically infer the schema of these objects using reflection. In Scala and Java, the type information is extracted from the language's type system (from JavaBeans and Scala case classes). In Python, Spark SQL samples the dataset to perform schema inference due to the dynamic type system.

For example, the Scala code below defines a DataFrame from an RDD of `User` objects. Spark SQL automatically detects the names ("name" and "age") and data types (string and int) of the columns.

```
case class User(name: String, age: Int)

// Create an RDD of User objects
usersRDD = spark.parallelize(
  List(User("Alice", 22), User("Bob", 19)))

// View the RDD as a DataFrame
usersDF = usersRDD.toDF
```

Internally, Spark SQL creates a logical data scan operator that points to the RDD. This is compiled into a physical operator that accesses fields of the native objects. It is important to note that this is very different from traditional object-relational mapping (ORM). ORMs often incur expensive conversions that translate an entire object into a different format. In contrast, Spark SQL accesses the native objects in-place, extracting only the fields used in each query.

The ability to query native datasets lets users run optimized relational operations within existing Spark programs. In addition, it makes it simple to combine RDDs with external structured data. For example, we could join the `users` RDD with a table in Hive:

```
views = ctx.table("pageviews")
usersDF.join(views, usersDF("name") === views("user"))
```

2.3.6 In-Memory Caching

Like Shark before it, Spark SQL can materialize (often referred to as "cache") hot data in memory using columnar storage. Compared with Spark's native cache, which simply stores data as JVM objects, the columnar cache can reduce memory footprint by an order of magnitude because it applies columnar compression schemes such as dictionary encoding and run-length encoding. Caching is particularly useful for interactive queries and for the iterative algorithms common in machine learning. It can be invoked by calling `cache()` on a DataFrame.

2.3.7 User-Defined Functions

User-defined functions (UDFs) have been an important extension point for database systems. For example, MySQL relies on UDFs to provide basic support for JSON data. A more advanced example is MADLib's use of UDFs to implement machine learning algorithms for Postgres and other database systems [42]. However, database systems often require UDFs to be defined in a separate programming environment that is different from the primary query interfaces. Spark SQL's DataFrame API supports inline definition of UDFs, without the complicated packaging and registration process found in other database systems. This feature has proven crucial for the adoption of the API.

In Spark SQL, UDFs can be registered inline by passing Scala, Java or Python functions, which may use the full Spark API internally. For example, given a `model` object for a machine learning model, we could register its prediction function as a UDF:

```
val model: LogisticRegressionModel = ...

ctx.udf.register("predict",
  (x: Float, y: Float) => model.predict(Vector(x, y)))

ctx.sql("SELECT predict(age, weight) FROM users")
```

Once registered, the UDF can also be used via the JDBC/ODBC interface by business intelligence tools. In addition to UDFs that operate on scalar values like the one here, one can define UDFs that operate on an entire table by taking its name, as in MADLib [42], and use the distributed Spark API within them, thus exposing advanced analytics functions to SQL users. Finally, because UDF definitions and query execution are expressed using the same general-purpose language (*e.g.*, Scala or Python), users can debug or profile the entire program using standard tools.

The example above demonstrates a common use case in many pipelines, *i.e.*, one that employs both relational operators and advanced analytics methods that are cumbersome to express in SQL. The DataFrame API lets developers seamlessly mix these methods.

2.4 Catalyst Optimizer

To implement Spark SQL, we designed a new extensible optimizer, Catalyst, based on functional programming constructs in Scala. Catalyst's extensible design had two purposes. First, we wanted to make it easy to add new optimization techniques and features to Spark SQL, especially to tackle various problems we were seeing specifically with "big data" (*e.g.*, semistructured data and advanced analytics). Second, we wanted to enable external developers to extend the optimizer—for example, by adding data source specific rules that can push filtering or aggregation into external storage systems, or support for new data types. Catalyst supports both rule-based and cost-based optimization.

While extensible optimizers have been proposed in the past, they have typically required a complex domain specific language to specify rules, and an "optimizer compiler" to translate the rules into executable code [59, 60]. This leads to a significant learning curve and maintenance burden. In contrast, Catalyst uses standard features of the Scala programming language, such as pattern-matching [49], to let developers use the full programming language while still making rules easy to specify. Functional languages were designed in part to build compilers, so we found Scala well-suited to this task. Nonetheless, Catalyst is, to our knowledge, the first production-quality query optimizer built on such a language.

At its core, Catalyst contains a general library for representing *trees* and applying *rules* to manipulate them.[3] On top of this framework, we have built libraries specific to relational query processing (*e.g.*, expressions, logical query plans), and several sets of rules that handle different phases of query execution: analysis, logical optimization, physical planning, and code generation to compile parts of queries to Java bytecode. For the latter, we use another Scala feature, quasiquotes [113], that makes it easy to generate code at runtime from composable expressions. Finally, Catalyst offers several public extension points, including external data sources and user-defined types.

2.4.1 Trees

The main data type in Catalyst is a *tree* composed of *node* objects. Each node has a node type and zero or more children. New node types are defined in Scala as subclasses of the `TreeNode` class. These objects are immutable and can be manipulated using functional transformations, as discussed in the next subsection.

As a simple example, suppose we have the following three node classes for a very simple expression language:[4]

[3] Cost-based optimization is performed by generating multiple plans using rules, and then computing their costs.

[4] We use Scala syntax for classes here, where each class's fields are defined in parentheses, with their types given using a colon.

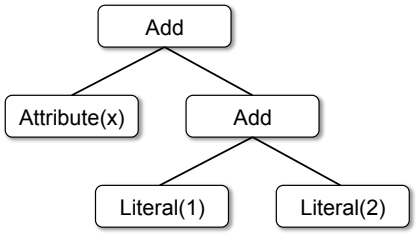


Figure 2.3: Catalyst tree for the expression `x+(1+2)`.

- `Literal(value: Int)`: a constant value
- `Attribute(name: String)`: an attribute from an input row, *e.g.,* "x"
- `Add(left: TreeNode, right: TreeNode)`: sum of two expressions.

These classes can be used to build up trees; for example, the tree for the expression `x+(1+2)`, shown in Figure 2.3, would be represented in Scala code as follows:

```
Add(Attribute(x), Add(Literal(1), Literal(2)))
```

2.4.2 Rules

Trees can be manipulated using *rules*, which are functions from a tree to another tree. While a rule can run arbitrary code on its input tree (given that this tree is just a Scala object), the most common approach is to use a set of *pattern matching* functions that find and replace subtrees with a specific structure.

Pattern matching is a feature of many functional languages that allows extracting values from potentially nested structures of algebraic data types. In Catalyst, trees offer a `transform` method that applies a pattern matching function recursively on all nodes of the tree, transforming the ones that match each pattern to a result. For example, we could implement a rule that folds Add operations between constants as follows:

```
tree.transform {
  case Add(Literal(c1), Literal(c2)) => Literal(c1+c2)
}
```

Applying this to the tree for `x+(1+2)`, in Figure 2.3, would yield the new tree `x+3`. The `case` keyword here is Scala's standard pattern matching syntax [49], and can be used to match on the type of an object as well as give names to extracted values (`c1` and `c2` here).

The pattern matching expression that is passed to `transform` is a *partial function*, meaning that it only needs to match to a subset of all possible input trees. Catalyst will tests which parts of a tree a given rule applies to, automatically skipping over and descending into subtrees that do not match. This ability means that rules only need to reason about the trees where a given optimization applies and not those

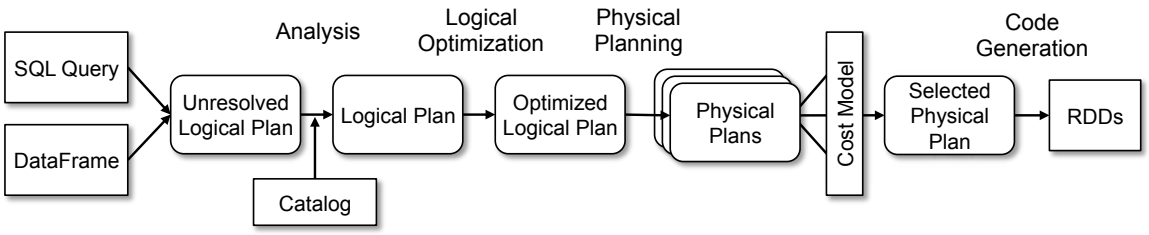


Figure 2.4: Phases of query planning in Spark SQL. Rounded rectangles represent Catalyst trees.

that do not match. Thus, rules do not need to be modified as new types of operators are added to the system.

Rules (and Scala pattern matching in general) can match multiple patterns in the same `transform` call, making it very concise to implement multiple transformations at once:

```
tree.transform {
  case Add(Literal(c1), Literal(c2)) => Literal(c1+c2)
  case Add(left, Literal(0)) => left
  case Add(Literal(0), right) => right
}
```

In practice, rules may need to execute multiple times to fully transform a tree. Catalyst groups rules into *batches*, and executes each batch until it reaches a *fixed point*, that is, until the tree stops changing after applying its rules. Running rules to fixed point means that each rule can be simple and self-contained, and yet still eventually have larger global effects on a tree. In the example above, repeated application would constant-fold larger trees, such as `(x+0)+(3+3)`. As another example, a first batch might analyze an expression to assign types to all of the attributes, while a second batch might use these types to do constant folding. After each batch, developers can also run sanity checks on the new tree (*e.g.,* to see that all attributes were assigned types), often also written via recursive matching.

Finally, rule conditions and their bodies can contain arbitrary Scala code. This gives Catalyst more power than domain specific languages for optimizers, while keeping it concise for simple rules.

In our experience, functional transformations on immutable trees make the whole optimizer very easy to reason about and debug. They also enable parallelization in the optimizer, although we do not yet exploit this.

2.4.3 Using Catalyst in Spark SQL

We use Catalyst's general tree transformation framework in four phases, shown in Figure 2.4: (1) analyzing a logical plan to resolve references, (2) logical plan optimization, (3) physical planning, and (4) code generation to compile parts of the query to Java bytecode. In the physical planning phase, Catalyst may generate

multiple plans and compare them based on cost. All other phases are purely rule-based. Each phase uses different types of tree nodes; Catalyst includes libraries of nodes for expressions, data types, and logical and physical operators. We now describe each of these phases.

Analysis

Spark SQL begins with a relation to be computed, either from an abstract syntax tree (AST) returned by a SQL parser, or from a DataFrame object constructed using the API. In both cases, the relation may contain unresolved attribute references or relations: for example, in the SQL query `SELECT col FROM sales`, the type of `col`, or even whether it is a valid column name, is not known until we look up the table `sales`. An attribute is called unresolved if we do not know its type or have not matched it to an input table (or an alias). Spark SQL uses Catalyst rules and a Catalog object that tracks the tables in all data sources to resolve these attributes. It starts by building an "unresolved logical plan" tree with unbound attributes and data types, then applies rules that do the following:

- Looking up relations by name from the catalog.
- Mapping named attributes, such as `col`, to the input provided given operator's children.
- Determining which attributes refer to the same value to give them a unique ID (which later allows optimization of expressions such as `col = col`).
- Propagating and coercing types through expressions: for example, we cannot know the type of `1 + col` until we have resolved `col` and possibly cast its subexpressions to compatible types.

In total, the rules for the analyzer are about 1000 lines of code.

Logical Optimization

The logical optimization phase applies standard rule-based optimizations to the logical plan. These include constant folding, predicate pushdown, projection pruning, null propagation, Boolean expression simplification, and other rules. In general, we have found it extremely simple to add rules for a wide variety of situations. For example, when we added the fixed-precision `DECIMAL` type to Spark SQL, we wanted to optimize aggregations such as sums and averages on `DECIMAL`s with small precisions; it took 12 lines of code to write a rule that finds such decimals in `SUM` and `AVG` expressions, and casts them to unscaled 64-bit `LONG`s, does the aggregation on that, then converts the result back. A simplified version of this rule that only optimizes SUM expressions is reproduced below:

```
object DecimalAggregates extends Rule[LogicalPlan] {
  /** Maximum number of decimal digits in a Long */
  val MAX_LONG_DIGITS = 18

  def apply(plan: LogicalPlan): LogicalPlan = {
    plan transformAllExpressions {
      case Sum(e @ DecimalType.Expression(prec, scale))
        if prec + 10 <= MAX_LONG_DIGITS =>
        MakeDecimal(Sum(LongValue(e)), prec + 10, scale)
    }
}
```

As another example, a 12-line rule optimizes `LIKE` expressions with simple regular expressions into `String.startsWith` or `String.contains` calls. The freedom to use arbitrary Scala code in rules made these kinds of optimizations, which go beyond pattern-matching the structure of a subtree, easy to express. In total, the logical optimization rules are 800 lines of code.

Physical Planning

In the physical planning phase, Spark SQL takes a logical plan and generates one or more physical plans, using physical operators that match the Spark execution engine. It then selects a plan using a cost model. At the moment, cost-based optimization is only used to select join algorithms: for relations that are known to be small, Spark SQL uses a broadcast join, using a peer-to-peer broadcast facility available in Spark.[5] The framework supports broader use of cost-based optimization, however, as costs can be estimated recursively for a whole tree using a rule. We thus intend to implement richer cost-based optimization in the future.

The physical planner also performs rule-based physical optimizations, such as pipelining projections or filters into one Spark `map` operation. In addition, it can push operations from the logical plan into data sources that support predicate or projection pushdown. We will describe the API for these data sources in Section 2.4.4.

In total, the physical planning rules are about 500 lines of code.

Code Generation

The final phase of query optimization involves generating Java bytecode to run on each machine. Because Spark SQL often operates on in-memory datasets, where processing is CPU-bound, we wanted to support code generation to speed up execution. Nonetheless, code generation engines are often complicated to build, amounting essentially to a compiler. Catalyst relies on a special feature of the Scala language, quasiquotes [113], to make code generation simpler. Quasiquotes allow the programmatic construction of abstract syntax trees (ASTs) in the Scala language, which can then be fed to the Scala compiler at runtime to generate bytecode. We use

[5] Table sizes are estimated if the table is cached in memory or comes from an external file, or if it is the result of a subquery with a `LIMIT`.

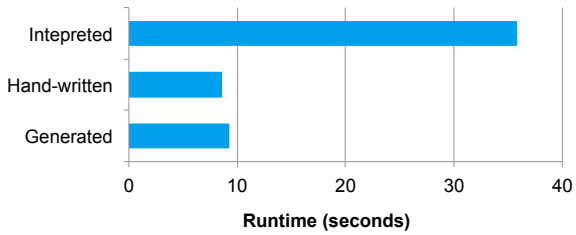


Figure 2.5: A comparision of the performance evaluating the expresion `x+x+x`, where `x` is an integer, 1 billion times.

Catalyst to transform a tree representing an expression in SQL to an AST for Scala code to evaluate that expression, and then compile and run the generated code.

As a simple example, consider the Add, Attribute and Literal tree nodes introduced in Section 2.4.2, which allowed us to write expressions such as `(x+y)+1`. Without code generation, such expressions would have to be interpreted for each row of data, by walking down a tree of Add, Attribute and Literal nodes. This introduces large amounts of branches and virtual function calls that slow down execution. With code generation, we can write a function to translate a specific expression tree to a Scala AST as follows:

```
def compile(node: Node): AST = node match {
  case Literal(value) => q"$value"
  case Attribute(name) => q"row.get($name)"
  case Add(left, right) =>
    q"${compile(left)} + ${compile(right)}"
}
```

The strings beginning with `q` are quasiquotes, meaning that although they look like strings, they are parsed by the Scala compiler at compile time and represent ASTs for the code within. Quasiquotes can have variables or other ASTs spliced into them, indicated using `$` notation. For example, `Literal(1)` would become the Scala AST for `1`, while `Attribute("x")` becomes `row.get("x")`. In the end, a tree like `Add(Literal(1), Attribute("x"))` becomes an AST for a Scala expression like `1+row.get("x")`.

Quasiquotes are type-checked at compile time to ensure that only appropriate ASTs or literals are substituted in, making them significantly more useable than string concatenation, and they result directly in a Scala AST instead of running the Scala parser at runtime. Moreover, they are highly composable, as the code generation rule for each node does not need to know how the trees returned by its children were built. Finally, the resulting code is further optimized by the Scala compiler in case there are expression-level optimizations that Catalyst missed. Figure 2.5 shows that quasiquotes let us generate code with performance similar to hand-tuned programs.

We have found quasiquotes very straightforward to use for code generation, and we observed that even new contributors to Spark SQL could quickly add rules for new types of expressions. Quasiquotes also work well with our goal of running on

native Java objects: when accessing fields from these objects, we can code-generate a direct access to the required field, instead of having to copy the object into a Spark SQL Row and use the Row's accessor methods. Finally, it was straightforward to combine code-generated evaluation with interpreted evaluation for expressions we do not yet generate code for, since the Scala code we compile can directly call into our expression interpreter.

In total, Catalyst's code generator is about 700 lines of code.

2.4.4 Extension Points

Catalyst's design around composable rules makes it easy for users and third-party libraries to extend. Developers can add batches of rules to each phase of query optimization at runtime, as long as they adhere to the contract of each phase (*e.g.*, ensuring that analysis resolves all attributes). However, to make it even simpler to add some types of extensions without understanding Catalyst rules, we have also defined two narrower public extension points: data sources and user-defined types. These still rely on facilities in the core engine to interact with the rest of the rest of the optimizer.

Data Sources

Developers can define a new data source for Spark SQL using several APIs, which expose varying degrees of possible optimization. All data sources must implement a `createRelation` function that takes a set of key-value parameters and returns a `BaseRelation` object for that relation, if one can be successfully loaded. Each `BaseRelation` contains a schema and an optional estimated size in bytes.[6] For instance, a data source representing MySQL may take a table name as a parameter, and ask MySQL for an estimate of the table size.

To let Spark SQL read the data, a `BaseRelation` can implement one of several interfaces that let them expose varying degrees of sophistication. The simplest, `TableScan`, requires the relation to return an RDD of `Row` objects for all of the data in the table. A more advanced `PrunedScan` takes an array of column names to read, and should return Rows containing only those columns. A third interface, `PrunedFilteredScan`, takes both desired column names and an array of Filter objects, which are a subset of Catalyst's expression syntax, allowing predicate pushdown.[7] The filters are advisory, *i.e.*, the data source should attempt to return only rows passing each filter, but it is allowed to return false positives in the case of filters that it cannot evaluate. Finally, a CatalystScan interface is given a complete sequence

[6] Unstructured data sources can also take a desired schema as a parameter; for example, there is a CSV file data source that lets users specify column names and types.

[7] At the moment, Filters include equality, comparisons against a constant, and `IN` clauses, each on one attribute.

of Catalyst expression trees to use in predicate pushdown, though they are again advisory.

These interfaces allow data sources to implement various degrees of optimization, while still making it easy for developers to add simple data sources of virtually any type. We and others have used the interface to implement the following data sources:

- CSV files, which simply scan the whole file, but allow users to specify a schema.
- Avro [18], a self-describing binary format for nested data.
- Parquet [19], a columnar file format for which we support column pruning as well as filters.
- A JDBC data source that scans ranges of a table from an RDBMS in parallel and pushes filters into the RDBMS to minimize communication.

To use these data sources, programmers specify their package names in SQL statements, passing key-value pairs for configuration options. For example, the Avro data source takes a path to the file:

```
CREATE TEMPORARY TABLE messages
USING com.databricks.spark.avro
OPTIONS (path "messages.avro")
```

All data sources can also expose network locality information, *i.e.*, which machines each partition of the data is most efficient to read from. This is exposed through the RDD objects they return, as RDDs have a built-in API for data locality [131].

Finally, similar interfaces exist for writing data to an existing or new table. These are simpler because Spark SQL just provides an RDD of Row objects to be written.

User-Defined Types (UDTs)

One feature we wanted to allow advanced analytics in Spark SQL was user-defined types. For example, machine learning applications may need a vector type, and graph algorithms may need types for representing a graph, which is possible over relational tables [126]. Adding new types can be challenging, however, as data types pervade all aspects of the execution engine. For example, in Spark SQL, the built-in data types are stored in a columnar, compressed format for in-memory caching (Section 2.3.6), and in the data source API from the previous section, we need to expose all possible data types to data source authors.

In Catalyst, we solve this issue by mapping user-defined types to structures composed of Catalyst's built-in types, described in Section 2.3.2. To register a Scala

type as a UDT, users provide a mapping from an object of their class to a Catalyst Row of built-in types, and an inverse mapping back. In user code, they can now use the Scala type in objects that they query with Spark SQL, and it will be converted to built-in types under the hood. Likewise, they can register UDFs (see Section 2.3.7) that operate directly on their type.

As a short example, suppose we want to register two-dimensional points (x, y) as a UDT. We can represent such vectors as two `DOUBLE` values. To register the UDT, we write the following:

```
class PointUDT extends UserDefinedType[Point] {
  def dataType = StructType(Seq( // Our native structure
    StructField("x", DoubleType),
    StructField("y", DoubleType)
  ))
  def serialize(p: Point) = Row(p.x, p.y)
  def deserialize(r: Row) =
    Point(r.getDouble(0), r.getDouble(1))
}
```

After registering this type, Points will be recognized within native objects that Spark SQL is asked to convert to DataFrames, and will be passed to UDFs defined on Points. In addition, Spark SQL will store Points in a columnar format when caching data (compressing `x` and `y` as separate columns), and Points will be writable to all of Spark SQL's data sources, which will see them as pairs of `DOUBLE`s. We use this capability in Spark's machine learning library, as we describe in Section 2.5.2.

2.5 Advanced Analytics Features

In this section, we describe three features we added to Spark SQL specifically to handle challenges in "big data" environments. First, in these environments, data is often unstructured or semistructured. While parsing such data procedurally is possible, it leads to lengthy boilerplate code. To let users query the data right away, Spark SQL includes a schema inference algorithm for JSON and other semistructured data. Second, large-scale processing often goes beyond aggregation and joins to machine learning on the data. We describe how Spark SQL is being incorporated into a new high-level API for Spark's machine learning library [89]. Last, data pipelines often combine data from disparate storage systems. Building on the data sources API in Section 2.4.4, Spark SQL supports query federation, allowing a single program to efficiently query disparate sources. These features all build on the Catalyst framework.

2.5.1 Schema Inference for Semistructured Data

Semistructured data is common in large-scale environments because it is easy to produce and to add fields to over time. Among Spark users, we have seen very high usage of JSON for input data. Unfortunately, JSON is cumbersome to work

```
{
  "text": "This is a tweet about #Spark",
  "tags": ["#Spark"],
  "loc": {"lat": 45.1, "long": 90}
}

{
  "text": "This is another tweet",
  "tags": [],
  "loc": {"lat": 39, "long": 88.5}
}

{
  "text": "A #tweet without #location",
  "tags": ["#tweet", "#location"]
}
```

Figure 2.6: A sample set of JSON records, representing tweets.

```
text STRING NOT NULL,
tags ARRAY<STRING NOT NULL> NOT NULL,
loc STRUCT<lat FLOAT NOT NULL, long FLOAT NOT NULL>
```

Figure 2.7: Schema inferred for the tweets in Figure 2.6.

with in a procedural environment like Spark or MapReduce: most users resorted to ORM-like libraries (*e.g.,* Jackson [71]) to map JSON structures to Java objects, or some tried parsing each input record directly with lower-level libraries.

In Spark SQL, we added a JSON data source that automatically infers a schema from a set of records. For example, given the JSON objects in Figure 2.6, the library infers the schema shown in Figure 2.7. Users can simply register a JSON file as a table and query it with syntax that accesses fields by their path, such as:

```
SELECT loc.lat, loc.long FROM tweets
WHERE text LIKE '%Spark%' AND tags IS NOT NULL
```

Our schema inference algorithm works in one pass over the data, and can also be run on a sample of the data if desired. It is related to prior work on schema inference for XML and object databases [27, 64, 95], but simpler because it only infers a static tree structure, without allowing recursive nesting of elements at arbitrary depths.

Specifically, the algorithm attempts to infer a tree of `STRUCT` types, each of which may contain atoms, arrays, or other `STRUCT`s. For each field defined by a distinct path from the root JSON object (*e.g.,*`tweet.loc.latitude`), the algorithm finds the most specific Spark SQL data type that matches observed instances of the field. For example, if all occurrences of that field are integers that fit into 32 bits, it will infer `INT`; if they are larger, it will use `LONG` (64-bit) or `DECIMAL` (arbitrary precision); if there are also fractional values, it will use `FLOAT`. For fields that display multiple types, Spark SQL uses `STRING` as the most generic type, preserving the original JSON representation. And for fields that contain arrays, it uses the same "most specific

supertype" logic to determine an element type from all the observed elements. We implement this algorithm using a single `reduce` operation over the data, which starts with schemata (*i.e.*, trees of types) from each individual record and merges them using an associative "most specific supertype" function that generalizes the types of each field. This makes the algorithm both single-pass and communication-efficient, as a high degree of reduction happens locally on each node.

As a short example, note how in Figures 2.6 and 2.7, the algorithm generalized the types of `loc.lat` and `loc.long`. Each field appears as an integer in one record and a floating-point number in another, so the algorithm returns `FLOAT`. Note also how for the `tags` field, the algorithm inferred an array of strings that cannot be null.

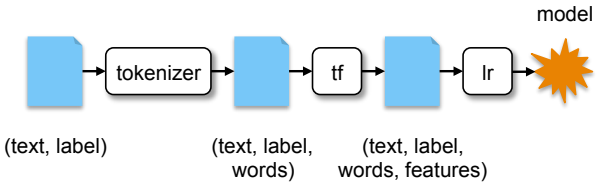
In practice, we have found this algorithm to work well with real-world JSON datasets. For example, it correctly identifies a usable schema for JSON tweets from Twitter's firehose, which contain around 100 distinct fields and a high degree of nesting. Multiple Databricks customers have also successfully applied it to their internal JSON formats.

In Spark SQL, we also use the same algorithm for inferring schemas of RDDs of Python objects (see Section 2.3), as Python is not statically typed so an RDD can contain multiple object types. In the future, we plan to add similar inference for CSV files and XML. Developers have found the ability to view these types of datasets as tables and immediately query them or join them with other data extremely valuable for their productivity.

2.5.2 Integration with Spark's Machine Learning Library

As an example of Spark SQL's utility in other Spark modules, MLlib, Spark's machine learning library, introduced a new high-level API that uses DataFrames [89]. This new API is based on the concept of machine learning *pipelines*, an abstraction in other high-level ML libraries like SciKit-Learn [112]. A pipeline is a graph of transformations on data, such as feature extraction, normalization, dimensionality reduction, and model training, each of which exchange *datasets*. Pipelines are a useful abstraction because ML workflows have many steps; representing these steps as composable elements makes it easy to change parts of the pipeline or to search for tuning parameters at the level of the whole workflow.

To exchange data between pipeline stages, MLlib's developers needed a format that was compact (as datasets can be large) yet flexible, allowing multiple types of fields to be stored for each record. For example, a user may start with records that contain text fields as well as numeric ones, then run a featurization algorithm such as TF-IDF on the text to turn it into a vector, normalize one of the other fields, perform dimensionality reduction on the whole set of features, etc. To represent datasets, the new API uses DataFrames, where each column represents a feature of the data. All algorithms that can be called in pipelines take a name for the input column(s) and output column(s), and can thus be called on any subset of the fields and produce new ones. This makes it easy for developers to build complex pipelines




```
data = <DataFrame of (text, label) records>

tokenizer = Tokenizer()
  .setInputCol("text").setOutputCol("words")
tf = HashingTF()
  .setInputCol("words").setOutputCol("features")
lr = LogisticRegression()
  .setInputCol("features")

pipeline = Pipeline().setStages([tokenizer, tf, lr])
model = pipeline.fit(data)
```

Figure 2.8: A short MLlib pipeline and the Python code to run it. We start with a DataFrame of (text, label) records, tokenize the text into words, run a term frequency featurizer (`HashingTF`) to get a feature vector, then train logistic regression.

while retaining the original data for each record. To illustrate the API, Figure 2.8 shows a short pipeline and the schemas of DataFrames created.

The main piece of work MLlib had to do to use Spark SQL was to create a user-defined type for vectors. This vector UDT can store both sparse and dense vectors, and represents them as four primitive fields: a boolean for the type (dense or sparse), a size for the vector, an array of indices (for sparse coordinates), and an array of double values (either the non-zero coordinates for sparse vectors or all coordinates otherwise). Apart from DataFrames' utility for tracking and manipulating columns, we also found them useful for another reason: they made it much easier to expose MLlib's new API in all of Spark's supported programming languages. Previously, each algorithm in MLlib took objects for domain-specific concepts (*e.g.*, a labeled point for classification, or a (user, product) rating for recommendation), and each of these classes had to be implemented in the various languages (*e.g.*, copied from Scala to Python). Using DataFrames everywhere made it much simpler to expose all algorithms in all languages, as we only need data conversions in Spark SQL, where they already exist. This is especially important as Spark adds bindings for new programming languages.

Finally, using DataFrames for storage in MLlib also makes it very easy to expose all its algorithms in SQL. We can simply define a MADlib-style UDF, as described in Section 2.3.7, which will internally call the algorithm on a table. We are also exploring APIs to expose pipeline construction in SQL.

2.5.3 Query Federation to External Databases

Data pipelines often combine data from heterogeneous sources. For example, a recommendation pipeline might combine traffic logs with a user profile database and users' social media streams. As these data sources often reside in different machines or geographic locations, naively querying them can be prohibitively expensive. Spark SQL data sources leverage Catalyst to push predicates down into the data sources whenever possible.

For example, the following uses the JDBC data source and the JSON data source to join two tables together to find the traffic log for the most recently registered users. Conveniently, both data sources can automatically infer the schema without users having to define it. The JDBC data source will also push the filter predicate down into MySQL to reduce the amount of data transferred.

```
CREATE TEMPORARY TABLE users USING jdbc
OPTIONS(driver "mysql" url "jdbc:mysql://userDB/users")

CREATE TEMPORARY TABLE logs
USING json OPTIONS (path "logs.json")

SELECT users.id, users.name, logs.message
FROM users JOIN logs WHERE users.id = logs.userId
AND users.registrationDate > "2015-01-01"
```

Under the hood, the JDBC data source uses the PrunedFilteredScan interface in Section 2.4.4, which gives it both the names of the columns requested and simple predicates (equality, comparison and `IN` clauses) on these columns. In this case, the JDBC data source will run the following query on MySQL:[8]

```
SELECT users.id, users.name FROM users
WHERE users.registrationDate > "2015-01-01"
```

In future Spark SQL releases, we are also looking to add predicate pushdown for key-value stores such as HBase and Cassandra, which support limited forms of filtering.

2.6 Performance Evaluation

We evaluate the performance of Spark SQL on two dimensions: SQL query processing performance and Spark program performance. In particular, we demonstrate that Spark SQL's extensible architecture not only enables a richer set of functionalities, but brings substantial performance improvements over previous Spark-based SQL engines. In addition, for Spark application developers, the DataFrame API can bring substantial speedups over the native Spark API while making Spark programs more concise and easier to understand. Finally, applications that combine

[8] The JDBC data source also supports "sharding" a source table by a particular column and reading different ranges of it in parallel.

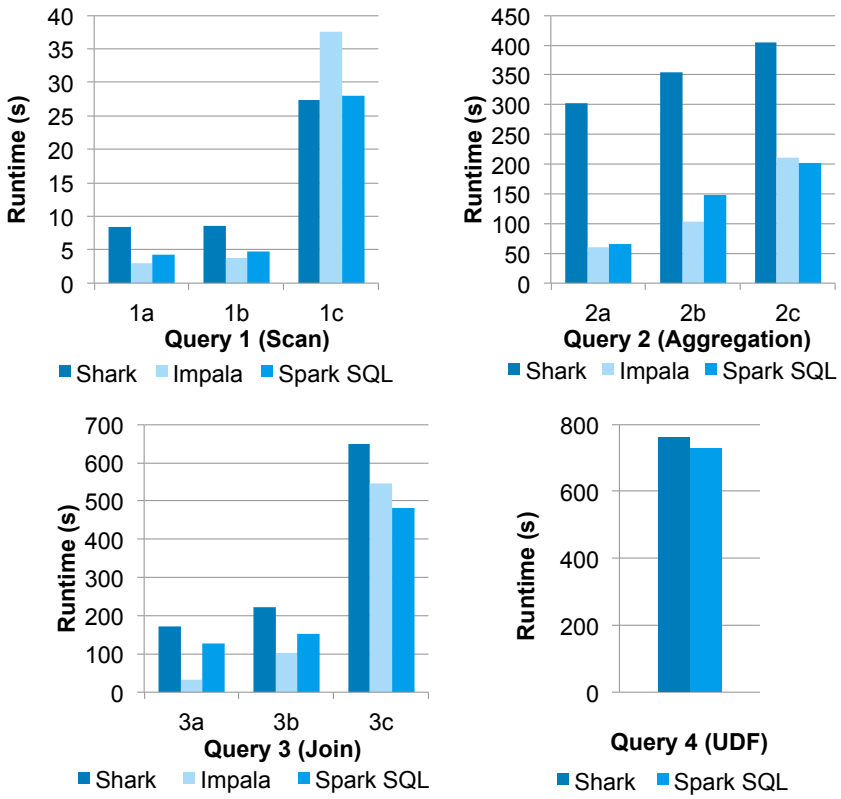


Figure 2.9: Performance of Shark, Impala and Spark SQL on the big data benchmark queries [104].

relational and procedural queries run faster on the integrated Spark SQL engine than by running SQL and procedural code as separate parallel jobs.

2.6.1 SQL Performance

We compared the performance of Spark SQL against Shark and Impala [3] using the AMPLab big data benchmark [124], which uses a web analytics workload developed by Pavlo et al. [104]. The benchmark contains four types of queries with different parameters performing scans, aggregation, joins and a UDF-based MapReduce job. We used a cluster of six EC2 `i2.xlarge` machines (one master, five workers) each with 4 cores, 30 GB memory and an 800 GB SSD, running HDFS 2.4, Spark 1.3, Shark 0.9.1 and Impala 2.1.1. The dataset was 110 GB of data after compression using the columnar Parquet format [19].

Figure 2.9 shows the results for each query, grouping by the query type. Queries 1–3 have different parameters varying their selectivity, with 1a, 2a, etc being the most selective and 1c, 2c, etc being the least selective and processing more data. Query 4 uses a Python-based Hive UDF that was not directly supported in Impala, but was largely bound by the CPU cost of the UDF.

We see that in all queries, Spark SQL is substantially faster than Shark and generally competitive with Impala. The main reason for the difference with Shark is code generation in Catalyst (Section 2.4.3), which reduces CPU overhead. This feature makes Spark SQL competitive with the C++ and LLVM based Impala engine in many of these queries. The largest gap from Impala is in query 3a where Impala chooses a better join plan because the selectivity of the queries makes one of the tables very small.

2.6.2 DataFrames vs. Native Spark Code

In addition to running SQL queries, Spark SQL can also help non-SQL developers write simpler and more efficient Spark code through the DataFrame API. Catalyst can perform optimizations on DataFrame operations that are hard to do with hand written code, such as predicate pushdown, pipelining, and automatic join selection. Even without these optimizations, the DataFrame API can result in more efficient execution due to code generation. This is especially true for Python applications, as Python is typically slower than the JVM.

For this evaluation, we compared two implementations of a Spark program that does a distributed aggregation. The dataset consists of 1 billion integer pairs, `(a, b)` with 100,000 distinct values of `a`, on the same five-worker `i2.xlarge` cluster as in the previous section. We measure the time taken to compute the average of `b` for each value of `a`. First, we look at a version that computes the average using the `map` and `reduce` functions in the Python API for Spark:

```
sum_and_count = \
  data.map(lambda x: (x.a, (x.b, 1))) \
      .reduceByKey(lambda x, y: (x[0]+y[0], x[1]+y[1])) \
      .collect()
[(x[0], x[1][0] / x[1][1]) for x in sum_and_count]
```

In contrast, the same program can written as a simple manipulation using the DataFrame API:

```
df.groupBy("a").avg("b")
```

Figure 2.10, shows that the DataFrame version of the code outperforms the hand written Python version by 12×, in addition to being much more concise. This is because in the DataFrame API, only the logical plan is constructed in Python, and all physical execution is compiled down into native Spark code as JVM bytecode, resulting in more efficient execution. In fact, the DataFrame version also outperforms a Scala version of the Spark code above by 2×. This is mainly due to code generation: the code in the DataFrame version avoids expensive allocation of key-value pairs that occurs in hand-written Scala code.

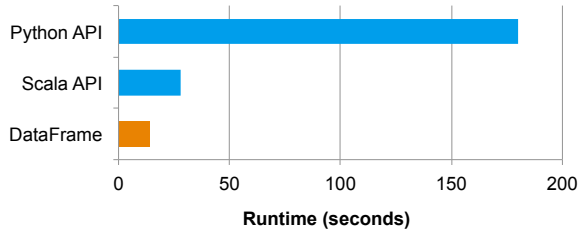


Figure 2.10: Performance of an aggregation written using the native Spark Python and Scala APIs versus the DataFrame API.

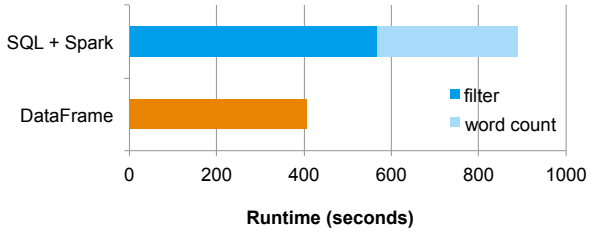


Figure 2.11: Performance of a two-stage pipeline written as a separate Spark SQL query and Spark job (above) and an integrated DataFrame job (below).

2.6.3 Pipeline Performance

The DataFrame API can also improve performance in applications that combine relational and procedural processing, by letting developers write all operations in a single program and pipelining computation across relational and procedural code. As a simple example, we consider a two-stage pipeline that selects a subset of text messages from a corpus and computes the most frequent words. Although very simple, this can model some real-world pipelines, *e.g.*, computing the most popular words used in tweets by a specific demographic.

In this experiment, we generated a synthetic dataset of 10 billion messages in HDFS. Each message contained on average 10 words drawn from an English dictionary. The first stage of the pipeline uses a relational filter to select roughly 90% of the messages. The second stage computes the word count.

First, we implemented the pipeline using a separate SQL query followed by a Scala-based Spark job, as might occur in environments that run separate relational and procedural engines (*e.g.*, Hive and Spark). We then implemented a combined pipeline using the DataFrame API, *i.e.*, using DataFrame's relational operators to perform the filter, and using the RDD API to perform a word count on the result. Compared with the first pipeline, the second pipeline avoids the cost of saving the whole result of the SQL query to an HDFS file as an intermediate dataset before passing it into the Spark job, because SparkSQL pipelines the `map` for the word count with the relational operators for the filtering. Figure 2.11 compares the runtime performance of the two approaches. In addition to being easier to understand and operate, the DataFrame-based pipeline also improves performance by 2×.

2.7 Research Applications

In addition to the immediately practical production use cases of Spark SQL, we have also seen significant interest from researchers working on more experimental projects. We outline two research projects that leverage the extensibility of Catalyst: one in approximate query processing and one in genomics.

2.7.1 Generalized Online Aggregation

Zeng et al. have used Catalyst in their work on improving the generality of online aggregation [134]. This work generalizes the execution of online aggregation to support arbitrarily nested aggregate queries. It allows users to view the progress of executing queries by seeing results computed over a fraction of the total data. These partial results also include accuracy measures, letting the user stop the query when sufficient accuracy has been reached.

In order to implement this system inside of Spark SQL, the authors add a new operator to represent a relation that has been broken up into sampled batches. During query planning a call to `transform` is used to replace the original full query with several queries, each of which operates on a successive sample of the data.

However, simply replacing the full dataset with samples is not sufficient to compute the correct answer in an online fashion. Operations such as standard aggregation must be replaced with stateful counterparts that take into account both the current sample and the results of previous batches. Furthermore, operations that might filter out tuples based on approximate answers must be replaced with versions that can take into account the current estimated errors.

Each of these transformations can be expressed as Catalyst rules that modify the operator tree until it produces correct online answers. Tree fragments that are not based on sampled data are ignored by these rules and can execute using the standard code path. By using Spark SQL as a basis, the authors were able to implement a fairly complete prototype in approximately 2000 lines of code.

2.7.2 Computational Genomics

A common operation in computational genomics involves inspecting overlapping regions based on a numerical offsets. This problem can be represented as a join with inequality predicates. Consider two datasets, `a` and `b`, with a schema of `(start LONG, end LONG)`. The range join operation can be expressed in SQL as follows:

```
SELECT * FROM a JOIN b
WHERE a.start < a.end
  AND b.start < b.end
  AND a.start < b.start
  AND b.start < a.end
```

Without special optimization, the preceding query would be executed by many systems using an inefficient algorithm such as a nested loop join. In contrast, a specialized system could compute the answer to this join using an interval tree. Researchers in the ADAM project [97] were able to build a special planning rule into a version of Spark SQL to perform such computations efficiently, allowing them to leverage the standard data manipulation abilities alongside specialized processing code. The changes required were approximately 100 lines of code.

2.8 Discussion

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

2.8.1 Why are Previous MapReduce-Based Systems Slow?

Conventional wisdom is that MapReduce is slower than MPP databases for several reasons: expensive data materialization for fault tolerance, inferior data layout (*e.g.,* lack of indices), and costlier execution strategies [103, 118]. Our exploration of Hive confirms these reasons, but also shows that a combination of conceptually simple "engineering" changes to the engine (*e.g.,* in-memory storage) and more involved architectural changes (*e.g.,* partial DAG execution) can alleviate them. We also find that a somewhat surprising variable not considered in detail in MapReduce systems, the task scheduling overhead, actually has a dramatic effect on performance, and greatly improves load balancing if minimized. There is no fundamental reason why this overhead needs to be large, and Spark makes it several orders of magnitude lower than Hadoop.

Intermediate Outputs: MapReduce-based query engines, such as Hive, materialize intermediate data to disk in two situations. First, *within* a MapReduce job, the map tasks save their output in case a reduce task fails [48]. Second, many queries need to be compiled into *multiple* MapReduce steps, and engines rely on replicated file systems, such as HDFS, to store the output of each step.

For the first case, we note that map outputs were stored on disk primarily as a convenience to ensure there is sufficient space to hold them in large batch jobs. Map outputs are *not* replicated across nodes, so they will still be lost if the mapper node fails [48]. Thus, if the outputs fit in memory, it makes sense to store them in

memory initially, and only spill them to disk if they are large. Spark SQL's shuffle implementation does this by default, and sees far faster shuffle performance (and no seeks) when the outputs fit in RAM. This is often the case in aggregations and filtering queries that return a much smaller output than their input.[9] Another hardware trend that may improve performance, even for large shuffles, is SSDs, which would allow fast random access to a larger space than memory.

For the second case, engines that extend the MapReduce execution model to general task DAGs can run multi-stage jobs without materializing any outputs to HDFS. Many such engines have been proposed, including Dryad, Tenzing and Spark [69, 38, 131].

Data Format and Layout: While the naïve pure schema-on-read approach to MapReduce incurs considerable processing costs, many systems use more efficient storage formats within the MapReduce model to speed up queries. Hive itself supports "table partitions" (a basic index-like system where it knows that certain key ranges are contained in certain files, so it can avoid scanning a whole table), as well as column-oriented representation of on-disk data [121]. We go further in Spark SQL by using fast in-memory columnar representations within Spark. Spark SQL does this without modifying the Spark runtime by simply representing a block of tuples as a single Spark record (one Java object from Spark's perspective), and choosing its own representation for the tuples within this object.

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

Finally, one capability of RDDs that we do not yet exploit is random reads. While RDDs only support coarse-grained operations for their *writes*, *read* operations on them can be fine-grained, accessing just one record [131]. This would allow RDDs to be used as indices. Tenzing can use such remote-lookup reads for joins [38].

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

To truly optimize the execution of relational queries, however, we found it necessary to select execution plans based on data statistics. This becomes difficult in the presence of UDFs and complex analytics functions, which we seek to support as first-class citizens in Spark SQL. To address this problem, we proposed partial DAG execution (PDE), which allows our modified version of Spark to *change* the downstream portion of an execution graph once each stage completes based on data statistics. PDE goes beyond the runtime graph rewriting features in previous

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

systems, such as DryadLINQ [129], by collecting fine-grained statistics about ranges of keys and by allowing switches to a completely different join strategy, such as broadcast join, instead of just selecting the number of reduce tasks.

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

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

Sub-second tasks allow the engine to balance work across nodes extremely well, even when some nodes incur unpredictable delays (*e.g.*, network delays or JVM garbage collection). They also help dramatically with skew. Consider, for example, a system that needs to run a hash aggregation on 100 cores. If the system launches 100 reduce tasks, the key range for each task needs to be carefully chosen, as any imbalance will slow down the entire job. If it could split the work among 1000 tasks, then the slowest task can be as much as 10× slower than the average with negligible on the job response time! After implementing skew-aware partition selection in PDE, we were somewhat disappointed that it did not help compared to just having a higher number of reduce tasks in most workloads, because Spark could comfortably support thousands of such tasks. However, this property makes the engine highly robust to unexpected skew.

In this way, Spark stands in contrast to Hadoop/Hive, where using the wrong number of tasks was sometimes 10× slower than an optimal plan, and there has been considerable work to automatically choose the number of reduce tasks [77, 62]. Figure 2.12 shows how job execution times vary as the number of reduce tasks launched by Hadoop and Spark in a simple aggregation query on a 100-node cluster. Since a Spark job can launch thousands of reduce tasks without incurring much overhead, partition data skew can be mitigated by always launching many tasks.

More fundamentally, there are few reasons why sub-second tasks should not be feasible even at higher scales than we have explored, such as tens of thousands of nodes. Systems like Dremel [88] routinely run sub-second, multi-thousand-node jobs. Indeed, even if a single master cannot keep up with the scheduling decisions, the scheduling could be delegated across "lieutenant" masters for subsets of the cluster. Fine-grained tasks also offer many advantages over coarser-grained execution graphs beyond load balancing, such as faster recovery (by spreading out

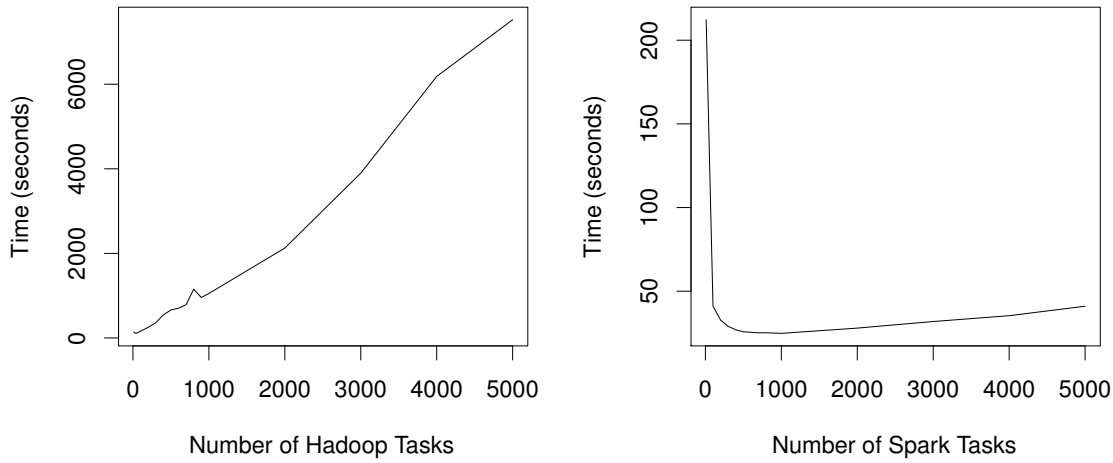


Figure 2.12: Task launching overhead

lost tasks across more nodes) and query elasticity [99]; we discuss some of these next.

2.8.2 Other Benefits of the Fine-Grained Task Model

The fine-grained task model also provides other attractive properties. We wish to point out two benefits that have been explored in MapReduce-based systems.

Elasticity: In traditional MPP databases, a distributed query plan is selected once, and the system needs to run at that level of parallelism for the whole duration of the query. In a fine-grained task system, however, nodes can appear or go away during a query, and pending work will automatically be spread onto them. This enables the database engine to naturally be elastic. If an administrator wishes to remove nodes from the engine (*e.g.*, in a virtualized corporate data center), the engine can simply treat those as failed, or (better yet) proactively replicate their data to other nodes if given a few minutes' warning. Similarly, a database engine running on a cloud could scale *up* by requesting new VMs if a query is expensive. Amazon's Elastic MapReduce [1] already supports resizing clusters at runtime.

Multitenancy: The same elasticity, mentioned above, enables dynamic resource sharing between users. In some traditional MPP databases, if an important query arrives while another large query is using most of the cluster, there are few options beyond canceling the earlier query. In systems based on fine-grained tasks, one can simply wait a few seconds for the current tasks from the first query to finish, and start giving the nodes tasks from the second query. For instance, Facebook and Microsoft have developed fair schedulers for Hadoop and Dryad that allow large historical queries, compute-intensive machine learning jobs, and short ad-hoc queries to safely coexist [130, 70].

2.9 Related Work

Programming Model Several systems have sought to combine relational processing with the procedural processing engines initially used for large clusters. As discussed earlier, Shark is the predecessor to Spark SQL, running on the same engine and offering the same combination of relational queries and advanced analytics. Spark SQL improves on Shark through a richer and more programmer-friendly API, DataFrames, where queries can be combined in a modular way using constructs in the host programming language (see Section 2.3). It also allows running relational queries directly on native RDDs, and supports a wide range of data sources beyond Hive.

One system that inspired Spark SQL's design was DryadLINQ [129], which compiles language-integrated queries in C# to a distributed DAG execution engine. LINQ queries are also relational but can operate directly on C# objects. Spark SQL goes beyond DryadLINQ by also providing a DataFrame interface similar to common data science libraries [108, 101], an API for data sources and types, and support for iterative algorithms through execution on Spark.

Other systems use only a relational data model internally and relegate procedural code to UDFs. For example, Hive and Pig [121, 98] offer relational query languages but have widely used UDF interfaces. ASTERIX [26] has a semi-structured data model internally. Stratosphere [13] also has a semi-structured model, but offers APIs in Scala and Java that let users easily call UDFs. PIQL [23] likewise provides a Scala DSL. Compared to these systems, Spark SQL integrates more closely with native Spark applications by being able to directly query data in user-defined classes (native Java/Python objects), and lets developers mix procedural and relational APIs in the same language. In addition, through the Catalyst optimizer, Spark SQL implements both optimizations (*e.g.*, code generation) and other functionality (*e.g.*, schema inference for JSON and machine learning data types) that are not present in most large-scale computing frameworks. We believe that these features are essential to offering an integrated, easy-to-use environment for big data.

Finally, data frame APIs have been built both for single machines [108, 101] and clusters [46, 28]. Unlike previous APIs, Spark SQL optimizes DataFrame computations with a relational optimizer.

Extensible Optimizers The Catalyst optimizer shares similar goals with extensible optimizer frameworks such as EXODUS [59] and Cascades [60]. Traditionally, however, optimizer frameworks have required a domain-specific language to write rules in, as well as an "optimizer compiler" to translate them to runnable code. Our major improvement here is to build our optimizer using standard features of a functional programming language, which provide the same (and often greater) expressivity while decreasing the maintenance burden and learning curve. Advanced language features helped with many areas of Catalyst—for example, our approach to code generation using quasiquotes (Section 2.4.3) is one of the simplest and most

composable approaches to this task that we know. While extensibility is hard to measure quantitatively, one promising indication is that Spark SQL had over 50 external contributors in the first 8 months after its release.

For code generation, LegoBase [75] recently proposed an approach using generative programming in Scala, which would be possible to use instead of quasiquotes in Catalyst.

Advanced Analytics Spark SQL builds on recent work to run advanced analytics algorithms on large clusters, including platforms for iterative algorithms [131] and graph analytics [126, 82]. The desire to expose analytics functions is also shared with MADlib [42], though the approach there is different, as MADlib had to use the limited interface of Postgres UDFs, while Spark SQL's UDFs can be full-fledged Spark programs. Finally, techniques including Sinew and Invisible Loading [120, 10] have sought to provide and optimize queries over semi-structured data such as JSON. We hope to apply some of these techniques in our JSON data source.

2.10 Conclusion

We have developed Spark SQL, a new module in Apache Spark providing rich integration with relational processing. Spark SQL builds on the lessons learned from its predecessor, Shark, the initial SQL implementation on Spark that is heavily tied to Hive internals. Spark SQL extends Spark with a declarative DataFrame API to allow relational processing, offering benefits such as automatic optimization, and letting users write complex pipelines that mix relational and complex analytics. It supports a wide range of features tailored to large-scale data analysis, including semi-structured data, query federation, and data types for machine learning. To enable these features, Spark SQL is based on an extensible optimizer called Catalyst that makes it easy to add optimization rules, data sources and data types by embedding into the Scala programming language. User feedback and benchmarks show that Spark SQL makes it significantly simpler and more efficient to write data pipelines that mix relational and procedural processing, while offering substantial speedups over previous SQL-on-Spark engines.

Chapter 3

Structured Streaming: Declarative Real-Time Applications

This chapter develops Structured Streaming, an extension to Spark SQL to support stream processing. We first describe the technical and operational challenges from stream processing, and present how Structured Streaming's ability to incrementalize query plans addresses those challenges. We also present use cases we have seen in production and conduct performance evaluation on the system. We end the chapter by surveying related work.

3.1 Introduction

Many large-scale data sources operate in real time, including sensors, logs from mobile applications, and the Internet of Things. As organizations have gotten better at capturing this data, they also want to process it in real time, whether to give human analysts the freshest possible data or drive automated decisions. Enabling broad access to streaming computation requires systems that are scalable, easy to use and easy to integrate into business applications.

While there has been tremendous progress in distributed stream processing systems in the past few years [17, 52, 12, 93, 43], these systems still remain fairly challenging to use in practice. In this chapter, we begin by describing these challenges, based on our experience with Spark Streaming [133], one of the earliest stream processing systems to provide a relatively high-level, functional API. We find that two challenges frequently came up with users. First, streaming systems often asked users to think in terms of low-level, physical execution concepts, such as at-least-once delivery, state storage and triggering modes, that are unique to streaming. Second, many systems focused *only* on streaming computation, but in real use cases, streaming was a part of a larger business application that also includes batch processing, joins with static data, and interactive queries. Integrating

streaming systems with these other systems (e.g., maintaining transactionality) requires significant engineering.

Motivated by these challenges, we describe Structured Streaming, a new high-level API for stream processing that was added to Apache Spark in summer 2016. Structured Streaming builds on many ideas in recent stream processing systems, such as separating processing time from event time and triggers in Google Dataflow [12], using a relational execution engine for performance [36], and offering a language-integrated API [52, 133], but aims to make them simpler to use and integrated with the rest of Apache Spark. Specifically, Structured Streaming differs from other widely used open source streaming APIs in two ways:

- **Incremental query model:** Structured Streaming automatically incrementalizes queries on static datasets expressed through Spark's SQL and DataFrame APIs, meaning that users often only need to understand Spark's batch APIs to write a streaming query, similar to [56]. Event time concepts are especially easy to express in this model, making them easier to understand. Although incremental query execution and view maintenance are well studied [29, 107, 135], we believe Structured Streaming is the first effort to adopt them in a widely used open source system.
- **Support for end-to-end applications:** Structured Streaming's API and built-in connectors aim to make it easy to write code that is "correct by default" when interacting with external systems and their code can be integrated into larger applications using Spark and other systems. Data sources and sinks follow a simple transactional model that enables "exactly-once" computation by default. The incrementalization based API also allows users to run a streaming query as a batch job or develop hybrid applications that join streams with static data computed through Spark's batch APIs. In addition, users can manage multiple streaming queries dynamically and run interactive queries on consistent snapshots of stream output, making it possible to write applications that go beyond computing a fixed result to let users refine and drill into streaming data.

Beyond these benefits, we have made several design choices in Structured Streaming that simplify operation and increase performance. First, Structured Streaming reuses the Spark SQL execution engine, including its optimizer and runtime code generator. This leads to high throughput compared to other streaming systems (e.g., 4× the throughput of Apache Flink in the Yahoo! Streaming Benchmark [40]), as in Trill [36], and also lets Structured Streaming automatically leverage new SQL functions added to Spark. The engine runs in a microbatch execution mode by default [133] but we are also extending it to support low-latency continuous operators for some queries because the API is agnostic to execution strategy [21].

Second, we found that operating a streaming application can be challenging, and designed the engine to support failures, code updates and recomputation of

already outputted data. For example, one common issue is that new data in a stream causes an application to crash, or worse, to output an incorrect result that users do not notice until much later (e.g., due to mis-parsing an input field). In Structured Streaming, each application maintains a write-ahead event log in human-readable JSON format that operators can use to restart the application from an arbitrary point. If the application crashes due to an error in a user-defined function, operators can update the UDF and restart from where it left off, which happens automatically when the restarted application reads the log. If it was outputting incorrect data instead, operators can restart from a point before the bad data started arriving and recompute results from there; this works provided old enough checkpoint information was kept.

Since Structured Streaming became part of open source Spark in 2016, we have observed a number of use cases, through the open source mailing lists as well as production use cases on Databricks' cloud service. We end the chapter with some example use cases. Production applications range from interactive network security analysis and automated alerts to incremental Extract, Transform and Load (ETL). Users often leverage the design of the engine in interesting ways, e.g., running a streaming query "discontinuously" as a series of single-microbatch jobs to leverage Structured Streaming's transactioal input and output without having to pay for cloud servers running 24/7. The largest production applications we discuss process over 1 PB of data per month on hundreds of machines. We also show that Structured Streaming outperforms Apache Flink and Kafka Streams by 4× and 90× respectively in the widely used Yahoo! Streaming Benchmark [40].

The rest of this chapter is organized as follows. We start by discussing stream processing challenges based on our experience with Spark Streaming in Section 3.2. Next, we give an overview of Structured Streaming (Section 3.3), then describe its API (Section 3.4), query planning (Section 3.5) and execution and operation (Section 3.6). In Section 3.7, we describe several large use cases. We then measure the system's performance in Section 3.8, discuss related work in Section 3.9 and conclude in Section 4.8.

3.2 Stream Processing Challenges

Despite extensive progress in the past few years, distributed streaming applications are still still generally considered hard to develop and operate. Before designing Structured Streaming, we spent time discussing the challenges in this domain with users and designers of other streaming systems, including Spark Streaming, Truviso, Storm, Flink, Dataflow and others. This section discusses the main challenges we saw.

3.2.1 Low-Level APIs

Streaming systems were invariably considered more difficult to use than batch ones due to complex API semantics. Some complexity is to be expected due to new concerns that arise only in streaming: for example, the user needs to think about what type of intermediate results the system should output before it has received all the data relevant to a particular entity, e.g., to a customer's browsing session on a website. However, other complexity arises due to the *low-level* nature of many streaming APIs: these APIs often ask users to specify applications at the level of *physical operators* with complex semantics instead of a more declarative level.

As a concrete example, the Google Dataflow model [12] has a powerful API with a rich set of options for handling event time aggregation, windowing and out-of-order data. However, in this model, users need to specify a windowing mode, triggering mode and trigger refinement mode (essentially, whether the operator outputs deltas or accumulated results) for each aggregation operator. Adding an operator that expects deltas after an aggregation that outputs accumulated results will lead to unexpected results. In essence, the raw API [25] asks the user to write a *physical* operator graph, not a logical query, so every user of the system needs to understand the intricacies of incremental processing.

Other APIs, such as Spark Streaming [133] and Flink's DataStream API [53], are also based on writing DAGs of physical operators and offer a complex array of options for managing state [55]. In addition, reasoning about applications becomes even more complex in systems that relax exactly-once semantics [17], effectively requiring the user to design and implement a consistency model.

To address this issue, we designed Structured Streaming to make simple applications simple to express using its incremental query model. In addition, we found that adding customizable *stateful processing* operators to this model still enabled advanced users to build their own processing logic, such as custom session-based windows, while staying within the incremental model (e.g., these same operators also work in batch jobs). Other open source systems have also recently been adding incremental SQL queries [54, 43], and of course such queries have long been studied [29, 107, 135].

3.2.2 Integration in End-to-End Applications

The second challenge we found was that nearly every streaming workload must run in the context of a larger application, and this integration often required significant engineering effort. Many streaming APIs focus primarily on reading streaming input from a source and writing streaming output to a sink, but end-to-end business applications need to perform other tasks. Examples include:

- The business purpose of the application may be to enable interactive queries on fresh data. In this case, streaming is used to compute and update summary

tables in a structured storage system such as a SQL database or Apache Hive [121]. It is important that when the streaming job updates its result, it does so atomically, so queries do not see partially written results. This can be challenging with file-based big data systems like Apache Hive, where multiple files need to "appear" in parallel, or even with parallel load commands in a data warehouse.

- An Extract, Transform and Load (ETL) job might need to join a stream with static data loaded from another storage system or transformed using other Spark code. In this case, it is important to be able to reason about consistency across the two systems (what happens when the static data is updated?), and it is helpful to write transformations on both datasets in the same API.
- A team may occasionally need to run its streaming business logic in a batch application, e.g., to backfill a result on old data or test alternate versions of the code. Rewriting the code in a separate system would be time-consuming and error-prone.

We address this challenge by integrating Structured Streaming closely with Spark's batch and interactive APIs.

3.2.3 Operational Challenges

One of the largest challenges to deploying streaming applications in practice is management and operation. Some key issues include:

- **Failures:** This is the most heavily studied issue in the research literature. In addition to single node failures, systems also need to support graceful shutdown and restart of the whole application, e.g., to let operators migrate it to a new cluster.
- **Code Updates:** Applications are rarely perfect, so developers may need to update their code. After an update, they may want the application to restart where it left off, or possibly to *recompute* past results that were erroneous due to a bug. Both cases need to be supported in the streaming system's state management and fault recovery mechanisms. In addition, updates to the *streaming system* itself (e.g., a patch to Spark) should be supported as much as possible.
- **Rescaling:** Applications see varying load over time, and generally *increasing* load in the long term, so operators may want to scale them up and down dynamically, especially in the cloud. Systems based on a static communication topology, while conceptually simple, are difficult to scale dynamically.
- **Stragglers:** Instead of outright failing, nodes in the streaming system can slow down due to hardware or software issues and degrade the throughput of the whole application. Systems should automatically handle this situation.

- **Monitoring:** Streaming systems need to give operators clear visibility into per-node load, backlogs, state size, etc. to let them make good operational decisions.

3.2.4 Cost and Performance

Beyond operational and engineering issues, the cost-performance of streaming applications can be an obstacle because these applications run 24/7. For example, without dynamic rescaling, an application will always consume its peak resources, and even with rescaling, it may be more expensive to update a result continuously than to run a periodic batch job. We thus designed Structured Streaming to leverage all the execution optimizations in Spark SQL.

So far, we chose to optimize *throughput* as our main performance metric because we found that it was often the most important metric in large-scale streaming applications. Applications that require a *distributed* streaming system usually work with large data volumes coming from external sources (e.g., mobile devices, sensors or IoT), where data may already incur a delay just getting to the system. This is one reason why event time processing is an important feature in these systems [12]. In contrast, latency-sensitive applications such as high-frequency trading or physical system control loops often run on a single scale-up processor, or even custom hardware like ASICs and FPGAs [14]. However, we also designed Structured Streaming to support executing over latency-optimized engines, and are building a continuous processing mode for this task [21], which we describe in Section 3.6.3. This is a change over Spark Streaming, where microbatching was "baked into" the API.

3.3 Structured Streaming Overview

Structured Streaming aims to tackle the stream processing challenges we identified through a combination of API design and execution engine design, including features to simplify operation. In this section, we give a brief overview of the engine before diving into each of these topics in the following sections. Figure 3.1 shows an overview of the Structured Streaming system.

Input and Output. Structured Streaming connects to a variety of *input sources* and *output sinks* for I/O. To provide "exactly-once" output and fault tolerance, it places two restrictions on sources and sinks, which are similar to other exactly-once systems [133, 52]:

1. Input sources must be *replayable*, allowing the system to re-read recent input data if a node crashes. In practice, organizations use a reliable message bus such as Amazon Kinesis or Apache Kafka [16, 73] for this purpose, or simply a durable file system.

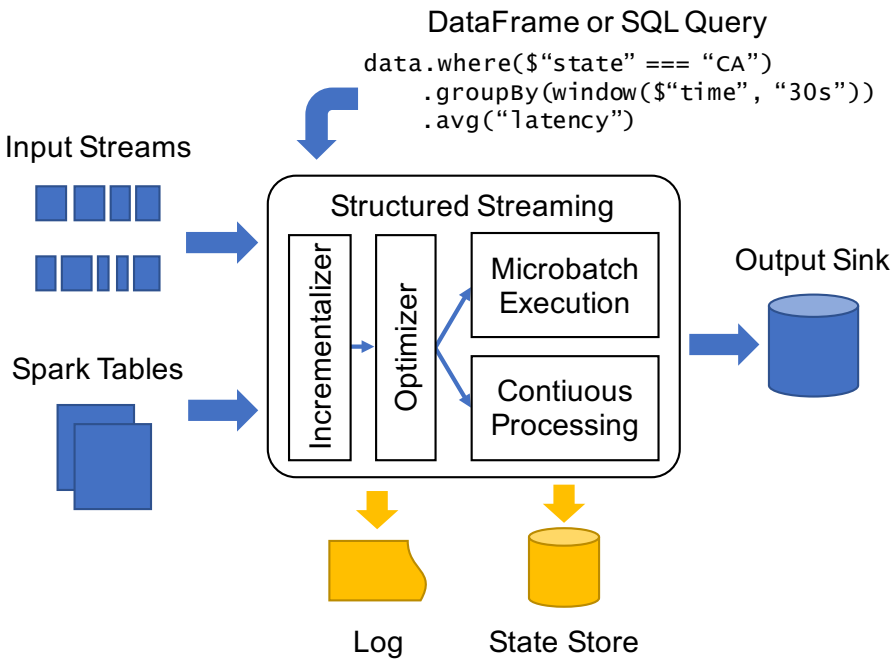


Figure 3.1: The components of Structured Streaming.

2. Output sinks must support *idempotent* writes, to ensure reliable recovery if a node fails while writing. Structured Streaming can also provide *atomic* output for some sinks that support it, where any update to the job's output appears at once even if it was written by multiple nodes working in parallel.

In addition to external systems, Structured Streaming also supports input and output from tables in Spark SQL. For example, users can compute a static table from any of Spark's batch input sources and join it with a stream, or ask Structured Streaming to output to an in-memory Spark table that users can query interactively.

API. Users program Structured Streaming by writing a query against one or more streams and tables using Spark SQL's batch APIs—SQL and DataFrames. This query defines an *output table* that the user wants to compute, assuming that each input stream is replaced by a table holding all the data received from that stream so far. The engine then determines how to compute and materialize this output table in a sink *incrementally*, using similar techniques to incremental view maintenance [29, 107]. Different sinks also support different *output modes*, which determine how the system may update their results—for example, some sinks are by nature append-only, while others allow updating records in place by key.

To support streaming specifically, Structured Streaming also adds several API features that fit in the existing Spark SQL API:

- *Triggers* control how often the engine will attempt to compute a new result and update the output sink, as in Dataflow [12].

- Users can mark a column as denoting *event time* (a timestamp set at the data source), and set a *watermark* policy to determine when enough data has been received to output a result for a specific event time, as in [12].
- *Stateful operators* allow users to track and update mutable state by key in order to implement complex processing, such as custom session-based windows. These are similar to Spark Streaming's `updateStateByKey` API [133].

Note that windowing, another key feature for streaming, is done using Spark SQL's existing aggregation operators. In addition, all the new APIs in Structured Streaming also work in batch jobs.

Execution. Once it has received a query, Structured Streaming optimizes it, incrementalizes it, and begins executing it. The system uses a microbatch model similar to Discretized Streams in Spark Streaming by default, which supports dynamic load balancing, rescaling, fault recovery and straggler mitigation by dividing work into small tasks [133]. In addition, a new continuous processing mode uses more traditional long-running operators [57] to provide lower latency at the cost of more static scheduling (Section 3.6.3).

In both cases, Structured Streaming uses two forms of durable storage to achieve fault tolerance. First, a *write-ahead log* keeps track of which data has been processed and reliably written to the output sink from each input source. For some output sinks, this log can be integrated with the sink to make updates to the sink atomic. Second, the system uses a larger-scale *state store* to hold snapshots of operator states for long-running aggregation operators. These are written asynchronously, and may be "behind" the latest data written to the output sink; the system will automatically track which state it has last updated in its log, and recompute state starting from that point in the data on failure. Both the log and state store can run over pluggable storage systems (e.g., HDFS or S3).

Operational Features. Using the durability of the write-ahead log and state store, users can achieve several forms of rollback and recovery. An *entire* Structured Streaming application can be shut down and restarted on new hardware. Running applications also tolerate node crashes, additions and stragglers automatically, by sending tasks to new nodes. For code updates to UDFs, it is sufficient to stop and restart the application, and it will begin using the new code. In addition, users can manually roll back the application to a previous point in the log and redo the part of the computation starting then, beginning from an older snapshot of the state store.

In addition, Structured Streaming's ability to execute with microbatches lets it "adaptively batch" data so that it can quickly catch up with input data if the load spikes or if a job is rolled back, then return to low latency later. This makes operation significantly simpler (e.g., operators can safely update code more often).

The next sections go into detail about Structured Streaming's API (§3.4), query planning (§3.5) and job execution and operation (§3.6).

3.4 Programming Model

Structured Streaming combines elements of Google Dataflow [12], incremental queries [29, 107, 135] and Spark Streaming [133] to enable stream processing beneath the Spark SQL API. In this section, we start by showing a short example, then describe the semantics of the model and the streaming-specific operators we added in Spark SQL to support streaming use cases (e.g., stateful operators).

3.4.1 A Short Example

Structured Streaming operates within Spark's structured data APIs—SQL, DataFrames and Datasets. The main abstraction users work with is *tables* (called DataFrames or Datasets in the language APIs), which each represent a view to be computed from input sources to the system.[1] When users create a table/DataFrame from a streaming input source, and attempt to compute it, Spark will automatically launch a streaming computation.

As a simple example, let us start with a *batch* job that counts clicks by country of origin for a web application. Suppose that the input data is JSON files and the output should be Parquet. This job can be written with Spark DataFrames in Scala as follows:

```
// Define a DataFrame to read from static data
data = spark.read.format("json").load("/in")

// Transform it to compute a result
counts = data.groupBy($"country").count()

// Write to a static data sink
counts.write.format("parquet").save("/counts")
```

Changing this job to use Structured Streaming only requires modifying the input and output sources, not the transformation in the middle. For example, if new JSON files are going to *continually* be uploaded to the `/in` directory, we can modify our job to continually update `/counts` by changing only the first and last lines:

```
// Define a DataFrame to read streaming data
data = spark.readStream.format("json").load("/in")

// Transform it to compute a result
counts = data.groupBy($"country").count()

// Write to a streaming data sink
counts.writeStream.format("parquet")
```

[1] Spark SQL offers several slightly different APIs that map to the same underlying query engine. The DataFrame API, modeled after data frames in R and Pandas [108, 101], offers a simple interface to build relational queries programmatically that is familiar to many users. The Dataset API adds static typing over DataFrames, similar to RDDs [131]. Alternatively, users can write SQL. All APIs produce a relational query plan.

```
.outputMode("complete").start("/counts")
```

The output mode parameter on the sink here specifies how Structured Streaming should update the sink. In this case, the complete mode means to write a complete result file for each update, because the file output sink chosen does not support fine-grained updates. However, other sinks, such as key-value stores, support more sophisticated output modes (e.g., updating just changed keys).

Under the hood, Structured Streaming will automatically incrementalize the query specified by the transformation(s) from input sources to data sinks, and execute it in a streaming fashion. The engine will also automatically maintain state and checkpoint it to external storage as needed—in this case, for example, we have a running count aggregation since the start of the stream, so the engine will keep track of the running counts for each country.

Finally, the API also naturally supports windowing and event time through Spark SQL's existing support aggregation operators. For example, instead of counting data by country, we could count it in 1-hour sliding windows advancing every 5 minutes by changing just the middle line of the computation as follows:

```
// Count events by windows on the "time" field
data.groupBy(window($"time","1h","5min")).count()
```

The `time` field here (event time) is just a field in the data, similar to `country` earlier. Users can also set a watermark on this field to let the system forget state for old windows after a timeout (§3.4.3).

3.4.2 Model Semantics

Formally, we define the semantics of Structured Streaming's execution model as follows:

1. Each input source provides a partially ordered set of records over time. We assume partial orders here because some message bus systems are parallel and do not define a total order across records—for example, Kafka divides streams into "partitions" that are each ordered.

2. The user provides a query to execute across the input data that can output a *result table* at any given point in processing time. Structured Streaming will always produce results consistent with running this query on a *prefix of the data in all input sources*. That is, it will never show results that incorporate one input record but do not incorporate its ancestors in the partial order. Moreover, these prefixes will be increasing over time.

3. *Triggers* tell the system *when* to run a new incremental computation and update the result table. For example, in microbatch mode, the user may wish to trigger an incremental update every minute (in processing time).

4. The sink's *output mode* specifies how the result table is written to the output system. The engine supports three distinct modes:

    - *Complete:* The engine writes the whole result table at once, e.g., replacing a whole file in HDFS with a new version. This is of course inefficient when the result is large.
    - *Append:* The engine can only add records to the sink. For example, a map-only job on a set of input files results in monotonically increasing output.
    - *Update:* The engine updates the sink in place based on a key for each record, updating only keys whose values changed.

Figure 3.2 illustrates the model visually. One attractive property of the model is that the *contents* of the result table (which is logically just a view that need never be materialized) are defined independently of the output mode (whether we output the whole table on every trigger, or only deltas). In contrast, APIs such as Dataflow require the equivalent of an output mode on every operator, so users must plan the whole operator DAG keeping in mind whether each operator is outputting complete results or positive or negative deltas, effectively incrementalizing the query by hand.

A second attractive property is that the model has strong consistency semantics, which we call *prefix consistency*. First, it guarantees that when input records are relatively ordered within a source (e.g., log records from the same device), the system will only produce results that incorporate them in the same records (e.g., never skipping a record). In addition, because the result table is defined based on *all* data in the input prefix at once, we know that all rows in the result table reflect each input records. For example, in some systems based on message-passing between nodes, the node that receives a record might send an update to two downstream nodes, but there is no guarantee that the outputs from these are synchronized.

In summary, with the Structured Streaming models, as long as users understand a regular Spark or DataFrame query, they can understand the content of the result table for their job and the values that will be written to the sink. Users need not worry about consistency, failures or incorrect processing orders.

Finally, the reader might notice that some of the output modes we defined are incompatible with certain types of query. For example, suppose we are aggregating counts by country, as in our code example in the previous section, and we want to use the append output mode. There is no way for the system to guarantee it has stopped receiving records for a given country, so this combination of query and output mode will not be allowed by the system. We describe which combinations are allowed in Section 3.5.1.

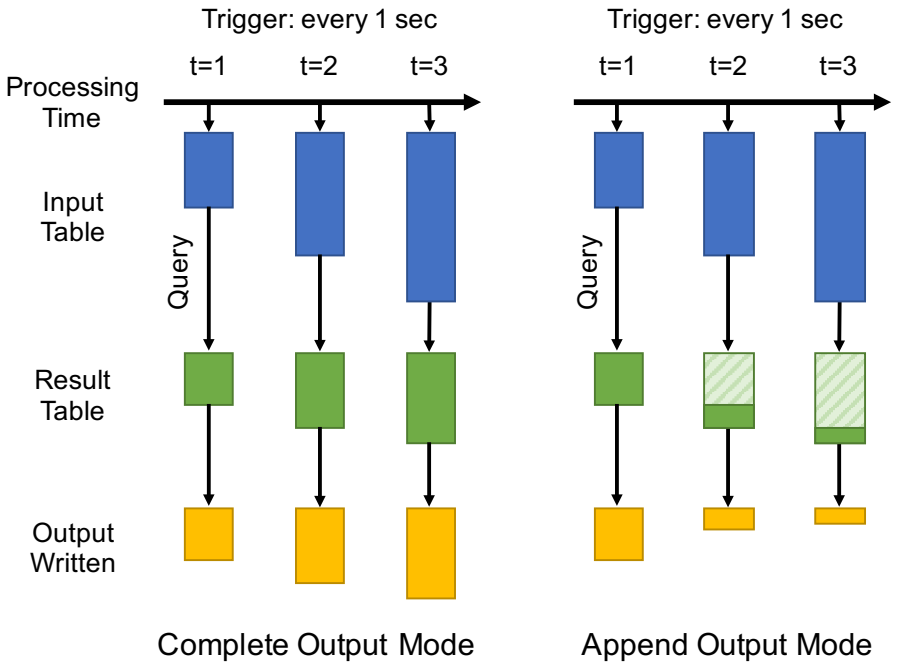


Figure 3.2: Structured Streaming's semantics for two output modes. Logically, all input data received up to a point in processing time is viewed as a large input table, and the user provides a query that defines a result table based on this input. Physically, Structured Streaming computes changes to the result table *incrementally* (without having to store all input data) and outputs results based on its output mode. For complete mode, it outputs the whole result table (left), while for append mode, it only outputs newly added records (right).

3.4.3 Streaming Specific Operators

Many Structured Streaming queries can be written using just the standard operators built intp Spark SQL, such as selection, joins and aggregation. However, to support some new requirements unique to streaming, we added two new types of operators to Spark SQL: *watermarking* operators tell the system when to "close" an event time window and output results or forget state, and *stateful operators* let users write custom logic to implement custom complex processing. Crucially, both of these new operators still fit within Structured Streaming's incremental semantics (§3.4.2), and both can also be used in batch applications.

Event Time Watermarks

From a logical point of view, the key idea in event time is to treat application-specified timestamps as an arbitrary field in the data, allowing records to arrive out-of-order [76, 12]. We can then use standard operators and incremental processing to update results grouped by event time. In practice, however, it is useful for the

processing system to have some loose bounds on how late data can arrive, for two reasons:

1. Allowing arbitrarily late data might require storing arbitrarily large state. For example, if we count data by 1-minute event time window, the system needs to remember a count for every 1-minute window since the application began, because a late record might still arrive for any particular minute. This can quickly lead to large amounts of state, especially if combined with another grouping key (e.g., group by minute and city).

2. Some sinks do not support data retraction, making it useful to be able to write the results for a given event time after a timeout. For example, many downstream applications want to start working with a "final" result from streaming analytics eventually, and might not be able to handle retractions. Append-mode sinks also do not support retractions.

Structured Streaming lets developers set a watermark [12] for event time columns using the `withWatermark` operator. This operator gives the system a delay threshold t_C for a given timestamp column C. At any point in time, the watermark for C is $\max(C) - t_C$, that is, t_C seconds before the maximum event time seen so far in C. Note that this choice of watermark is naturally robust to backlogged data: if the system cannot keep up with the input rate for a period of time, the watermark will not move forward arbitrarily during that time, and all events that arrived within at most T seconds of being produced will still be processed.

When present, watermarks affect when stateful operators can forget old state (e.g., if grouping by a window derived from a watermarked column), and when Structured Streaming will output data with an event time key to append-mode sinks. Different input streams can have different watermarks.

Stateful Operators

For developers who want to write custom stream processing logic, Structured Streaming's stateful operators are "UDFs with state" that give users control over the computation while fitting into Structured Streaming's semantics and fault tolerance mechanisms. There are two stateful operators, `mapGroupsWithState` and `flatMapGroupsWithState`. Both operators act on data that has been assigned a *key* using `groupByKey`, and let the developers track and update a *state* for each key using custom logic, as well as output records for each key. They are closely based on Spark Streaming's `updateStateByKey` operator [133].

The `mapGroupsWithState` operator, on a grouped dataset with keys of type K and values of type V, takes as argument an *update function* with the following arguments:

- `key` of type K
- `newValues` of type Iterator$[V]$

```
// Define an update function that simply tracks the
// number of events for each key as its state, returns
// that as its result, and times out keys after 30 min.
def updateFunc(key: UserId, newValues: Iterator[Event],
               state: GroupState[Int]): Int = {
  val totalEvents = state.get() + newValues.size()
  state.update(totalEvents)
  state.setTimeoutDuration("30 min")
  return totalEvents
}

// Use this update function on a stream, returning a
// new table lens that contains the session lengths.
lens = events.groupByKey(event => event.userId)
             .mapGroupsWithState(updateFunc)
```

Figure 3.3: Using `mapGroupsWithState` to track the number of events per session, timing out sessions after 30 minutes.

- `state` of type GroupState$[S]$, where S is a user-specified class.

The operator will invoke this function whenever one or more new values are received for a key. On each call, the function receives all of the values that were received for that key since the last call (multiple values may be batched for efficiency). It also receives a `state` object that wraps around a user-defined data type S, and allows the user to update the state, drop this key from state tracking, or set a timeout for this specific key (either in event time or processing time). This allows the user to store arbitrary data for the key, as well as implement custom logic for dropping state (e.g., custom exit conditions when implementing session-based windows).

Finally, the update function returns a user-specified return type R for its key. The return value of `mapGroupsWithState` is a new table with the final R record outputted for each group in the data (when the group is closed or times out). For example, the developer may wish to track user sessions on a website using `mapGroupsWithState`, and output the total number of pages clicked for each session.

To illustrate, Figure 3.3 shows how to use `mapGroupsWithState` to track user sessions, where a session is defined as a series of events with the same `userId` and gaps less than 30 minutes between them. We output the final number of events in each session as our return value R. A job could then compute metrics such as the average number of events per session by aggregating the result table `lens`.

The second stateful operator, `flatMapGroupsWithState`, is very similar to `mapGroupsWithState`, except that the update function can return *zero or more* values of type R per update instead of one. For example, this operator could be used to manually implement a stream-to-table join. The return values can either be returned all at once, when the group is closed, or incrementally across calls to the

update function. Both operators also work in batch mode, in which case the update function will only be called once.

3.5 Query Planning

We implemented Structured Streaming's query planning using the Catalyst extensible optimizer in Spark SQL, which allows writing composable rules using pattern matching in Scala, as described in Section [**?**]. Query planning proceeds in three stages: analysis to determine whether the query is valid, incrementalization and optimization.

3.5.1 Analysis

The first stage of query planning is analysis, where the engine validates the user's query and resolves the attributes and data types referred to in the query. Structured Streaming uses Spark SQL's existing analysis passes to resolve attributes and types, but adds new rules to check that the query can be executed incrementally by the engine. It also checks that the user's chosen output mode is valid for this specific query.

The rules for when each output mode is allowed are as follows:

- **Complete mode:** Only aggregation queries are allowed, where the amount of state that needs to be tracked is proportional to the number of keys in the result as opposed to the total number of input records. Queries that only do selection operations, for example, are not allowed because they would require storing all the input data.[2]
- **Append mode:** In this mode, the engine must be able to guarantee that an output record will not be updated once it writes it. (More formally, queries here should be monotonic [15].) Only selections, joins with static tables, and aggregations over event time on streams with watermarks are allowed. For aggregations over event time, Structured Streaming only outputs a key when the watermark has moved past its event time.
- **Update mode:** All queries support this mode, which allows updating past records based on a key.

These rules aim to establish two main goals: (1) that Structured Streaming need not remember an indefinitely large amount of input, which would be virtually guaranteed to be a bug in the user's application, and (2) that it respects the limitations of the output sink (e.g., append only). A full description of the supported modes is available in the Spark documentation [114].

[2] Although input sources in Structured Streaming should support replaying recent data on failure, most will only retain data for a limited time (e.g., Kafka and Kinesis).

3.5.2 Incrementalization

As of Spark 2.2.0, Structured Streaming can incrementalize a restricted set of queries, which can contain:

- Any number of selections, projections and `SELECT DISTINCT`s.
- Any number of inner or right outer joins between a stream and a static table.
- Stateful operators like `mapGroupsWithState` (§3.4.3).
- Up to one aggregation (possibly on compound keys).
- Sorting after an aggregation, only in complete output mode.

Work is also under way to support other types of queries, such as stream-to-stream joins on event time windows.

These supported queries are mapped, using transformation rules, to physical operators that perform both computation and state management. For example, an aggregation in the user query might be mapped to a StatefulAggregate operator that tracks open groups inside Structured Streaming's state store (§3.6.1) and outputs the desired result. Internally, Structured Streaming also tracks an output mode for each physical operator in the DAG produced during incrementalization, similar to the refinement mode for aggregation operators in Dataflow [12]. For example, some operators may update emitted records (equivalent to update mode), while others may only emit new records (append mode). Crucially, however, users do not have to specify these intra-DAG modes manually.

Incrementalization is an active area of work in Structured Streaming, but we have found that even the fairly simple set of queries supported today is broadly applicable in many use cases (§3.7). We expect to add more advanced incrementalization techniques later.

3.5.3 Optimization

The final stage of planning is optimization. Structured Streaming applies most of the optimization rules in Spark SQL, such as predicate pushdown, projection pushdown, expression simplification and others. In addition, it uses Spark SQL's Tungsten binary format for data in memory (avoiding the overhead of Java objects), and its runtime code generator to compile chains of operators to Java bytecode that runs over this format. This design means that most of the work in logical and execution optimization for analytical workloads in Spark SQL automatically applies to streaming.

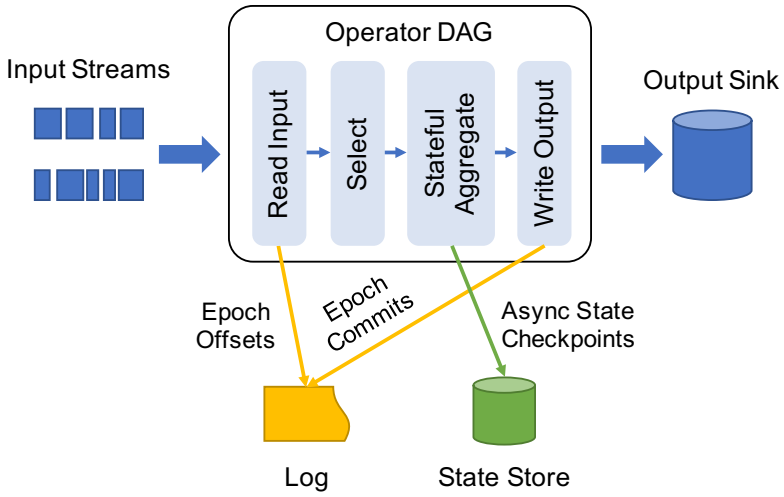


Figure 3.4: State management during the execution of Structured Streaming. Input operators are responsible for defining epochs in each input source and saving information about them (e.g., offsets) reliably in the write-ahead log. Stateful operators also checkpoint state asynchronously, marking it with its epoch, but this does not need to happen on every epoch. Finally, output operators log which epochs' outputs have been reliably committed to the idempotent output sink; the very last epoch may be rewritten on failure.

3.6 Application Execution

The final component of Structured Streaming is its execution strategy. In this section, we describe how the engine tracks state, and then the two execution modes: microbatching via fine-grained tasks and the in-development continuous processing mode. We then discuss operational features to simplify management.

3.6.1 State Management and Recovery

At a high level, Structured Streaming tracks state in a manner similar to Spark Streaming [133], in both its microbatch and continuous modes. The state of an application is tracked using two external storage systems: a *write-ahead log* that supports durable, atomic writes at low latency, and a *state store* that can store larger amounts of data durably and allows parallel access (e.g., S3 or HDFS). Structured Streaming uses these systems together to recover on failure.

The engine places two requirements on input sources and output sinks to provide fault tolerance. First, input sources should be *replayable*, i.e., allow re-reading recent data using some form of identifier, such as a stream offset. Durable message bus systems like Kafka and Kinesis meet this need. Second, output sinks should be *idempotent*, allowing Structured Streaming to rewrite some already written data on failure. Sinks can implement this in different ways.

Given these properties, Structured Streaming performs state tracking using the following mechanism, as shown in Figure 3.4:

1. As input operators read data, the master node of the Spark application defines *epochs* based on offsets in each input source. For example, Kafka and Kinesis present topics as a series of partitions, each of which are byte streams, and allow reading data using offsets in these partitions. The master writes the start and end offsets of each epoch durably to the log.

2. Any operators requiring state checkpoint their state periodically and asynchronosuly to the state store, using incremental checkpoints when possible. They store the epoch ID along with each checkpoint written. These checkpoints do not need to happen on every epoch or to block processing.[3]

3. Output operators write the epochs they committed to the log. The master waits for all nodes running an operator to report a commit for a given epoch before allowing commits for the next epoch. Depending on the sink, the master can also run an operation to finalize the writes from multiple nodes if the sink supports this. This means that if the streaming application fails, only one epoch may be partially written.[4]

4. Upon recovery, the new instance of the application starts by reading the log to find the last epoch that has not been committed to the sink, including its start and end offsets. It then uses the offsets of earlier epochs to reconstruct the application's in-memory state from the last epoch written to the state store. This just requires loading the old state and running those epochs with the same offsets while disabling output. Finally, the system reruns the last epoch and relies on the sink's idempotence to write its results, then starts defining new epochs.

Finally, all of the state management in this design is transparent to user code. Both the aggregation operators and custom stateful processing operators (e.g., `mapGroupsWithState`) automatically checkpoint state to the state store, without requiring custom code to do it. The user's data types only need to be serializable.

3.6.2 Microbatch Execution Mode

Structured Streaming jobs can execute in two modes: microbatching or continuous operators. The microbatch mode uses the *discretized streams* execution model

[3] In Spark 2.2.0, we actually make one checkpoint per epoch, but we plan to make them less frequent in a future release, as is already done in Spark Streaming.

[4] Some sinks, such as Amazon S3, provide no way to atomically commit multiple writes from different writer nodes. In such cases, we have also created *data sources* for Spark that check both the storage system and the streaming application's log to read only complete epochs. This lets other Spark applications still see a correct view [22].

from Spark Streaming [133], and inherits its benefits, such as dynamic load balancing, rescaling, straggler mitigation and fault recovery without whole-system rollback.

In this mode, epochs are typically set to be a few hundred milliseconds to a few seconds, and each epoch executes as a traditional Spark job composed of a DAG of independent tasks [131]. For example, a query doing selection followed by stateful aggregation might execute as a set of "map" tasks for the selection and "reduce" tasks for the aggregation, where the reduce tasks track state in memory on worker nodes and periodically checkpoint it to the state store. As in Spark Streaming, this mode provides the following benefits:

- **Dynamic load balancing:** Each operator's work is divided into small, independent tasks that can be scheduled on any node, so the system can automatically balance these across nodes if some are executing slower than others.
- **Fine-grained fault recovery:** If a node fails, only its tasks need to be rerun, instead of having to roll back the whole cluster to a checkpoint as in most systems based on topologies of long-lived operators. Moreover, the lost tasks can be rerun *in parallel*, further reducing recovery time [133].
- **Straggler mitigation:** Spark will launch backup copies of slow tasks as it does in batch jobs, and downstream tasks will simply use the output from whichever copy finishes first.
- **Rescaling:** Adding or removing a node is simple as tasks will automatically be scheduled on all the available nodes.
- **Scale and throughput:** Because this mode reuses Spark's batch execution engine, it inherits all the optimizations in this engine, such as a high-performance shuffle implementation [125] and the ability to run on thousands of nodes.

The main disadvantage of this mode is a higher minimum latency, as there is overhead to launching a DAG of tasks in Spark. In practice, however, latencies of a few seconds are achievable even on large clusters and are acceptable for many applications where data takes a similar amount of time to even be uploaded to the system (e.g., those collecting data from mobile devices).

3.6.3 Continuous Processing Mode

A new continuous processing mode under active development for Spark 2.3 [21] enables execution Structured Streaming jobs using long-lived operators as in a traditional streaming systems such as TelegraphCQ and Borealis [37, 6]. This mode enables lower latency at a cost of less operational flexibility (e.g., limited support for rescaling the job at runtime).

The key enabler for this execution mode was choosing a declarative API for Structured Streaming that is not tied to the execution strategy. For example, the original Spark Streaming API had some operators based on processing time that leaked the concept of microbatches into the programming model, making it hard to move programs to another type of engine. In contrast, Structured Streaming's API and semantics are independent of the execution engine: continuous execution is similar to having a much larger number of triggers. Note that unlike systems based purely on unsynchronized message passing, such as Storm [17], we do retain the concept of triggers and epochs in this mode so the output from multiple nodes can be coordinated and committed together to the sink.

Because the API supports fine-grained execution, Structured Streaming jobs could theoretically run on any existing distributed streaming engine design [52, 6, 37]. In continuous processing, we sought to implement a simple continuous operator engine that lives inside Spark and can reuse Spark's scheduling infrastructure and per-node operators (e.g., code-generated operators). The first version, currently under development, only support "map" jobs (i.e., no shuffle operations), which were one of the most common scenarios where users wanted lower latency. The design can also be extended to support shuffles, however. Compared to microbatch execution, there are two differences when using continuous processing:

- The master launches long-running tasks on each partition using Spark's scheduler that each read one partition of the input source (e.g., Kinesis stream) but execute *multiple* epochs. If one of these tasks fails, e.g., due to a node crash, Spark will automatically relaunch it on another node, which will start processing that same partition.

- Epochs are coordinated differently. The master periodically tells nodes to start a new epoch, and receives a start offset for the epoch on each input partition, which it inserts into the write-ahead log. When it asks them to start the next epoch, it also receives end offsets for the previous one, writes these to the log, and tells nodes to commit the epoch when it has written all the end offsets. Thus, the master is not on the critical path for inspecting all the input sources and defining start/end offsets.

In practice, we found that the main use case where organizations wanted low latency *and* the scale of a distributed processing engine was "stream to stream" map operations to transform data before it is used in other streaming applications. For example, an organization might upload events to Kafka, run some simple ETL transformations as a streaming map job, and write the transformed to Kafka again for consumption by multiple downstream applications. In this type of architecture, every streaming transformation job will add latency to all downstream applications, so organizations wish to minimize this latency. For this reason, we have chosen to only focus on map jobs in the first version of continuous processing, but in general it is possible to use a full distributed streaming design.

3.6.4 Operational Features

We used several properties of our execution strategy and API to design a number of operational features in Structured Streaming that tackle common problems in deployments.

Code Updates

Developers can update User-Defined Functions (UDFs) in their program and simply restart the application to use the new version of the code. For example, if a UDF is crashing on a particular input record, that epoch of processing will fail, so the developer can update the code and restart the application again to continue processing. This also applies to stateful operator UDFs, which can be updated as long as they retain the same schema for their state objects. We also designed Spark's log and state store formats to be binary compatible across Spark framework updates.

Manual Rollback

Sometimes, an application outputs *wrong* results for some time before a user notices: for example, a field that fails to parse might simply be reported as `NULL`. Therefore, rollbacks are a fact of life for many operators. In Structured Streaming, it is easy to determine which records went into each epoch from the write-ahead log and roll back the application to the epoch where a problem started occurring. We chose to store the write-ahead log as JSON to let administrators perform these operations manually.[5] As long as the input sources and state store still have data from the failed epoch, the job can start again from a previous point. Message buses like Kafka are typically configured for several weeks of retention so rollbacks are often possible. Moreover, Structured Streaming's support for running the same code as a batch job and for rescaling means that administrators can run the recovery on a temporarily larger cluster to catch up quickly.

Hybrid Batch and Streaming Execution

The most obvious benefit of Structured Streaming's unified API is that users can share code between batch and streaming applications, or run the same program as a batch job for testing. However, we have also found this useful in purely streaming applications, in two scenarios:

- *"Run-once" triggers for cost savings:* Many Spark users wanted the transactionality and state management properties of a streaming engine *without* running

[5] One additional step they may have to do is remove faulty data from the output sink, depending on the sink chosen. For the file sink, for example, it's straightforward to find which files were written in a particular epoch and remove those.

servers 24/7. Virtually all ETL workloads require tracking how far in the input one has gotten and which results have been saved reliably, which can be difficult to implement by hand. These functions are exactly what Structured Streaming's state management provides. Thus, several organizations implemented ETL jobs by running a *single* epoch of a Structured Streaming job every few hours as a batch computation, using the provided "run once" trigger that was originally designed for testing. This leads to significant cost savings (in one case, up to 10× [128]) for lower-volume applications. With all the major cloud providers now supporting per-second or per-minute billing [24], we believe this type of "discontinuous processing" will become more common.

- *Adaptive batching:* Even streaming applications occasionally experience large backlogs. For example, a link between two datacenters might go down, temporarily delaying data transfer, or there might simply be a spike in user activity. In these cases, Structured Streaming will automatically execute longer epochs in order to catch up with the input streams, often achieving similar throughput to Spark's batch jobs. This will not greatly increase latency, given that data is already backlogged, but will let the system catch up faster. In cloud environments, operators can also add extra nodes to the cluster temporarily.

Monitoring

Structured Streaming uses Spark's existing metrics API and structured event log to report information such as number of records processed, bytes shuffled across the network, etc. These interfaces are familiar to operators and easy to connect to a variety of monitoring tools using existing connectors.

Fault and Straggler Recovery

As discussed in §3.6.2, Structured Streaming's microbatch mode can recover from node failures, stragglers and load imbalances using Spark's fine-grained task execution model. The continuous processing mode recovers from node failures by launching the failed partition on a new node, but does not yet protect against stragglers or load imbalance.

Perhaps most importantly for operators, we aimed to make both Structured Streaming's semantics and its fault tolerance model easy to understand. With a simple design, operators can form an accurate model of how a system runs and what various actions will do without having know all its internal details.

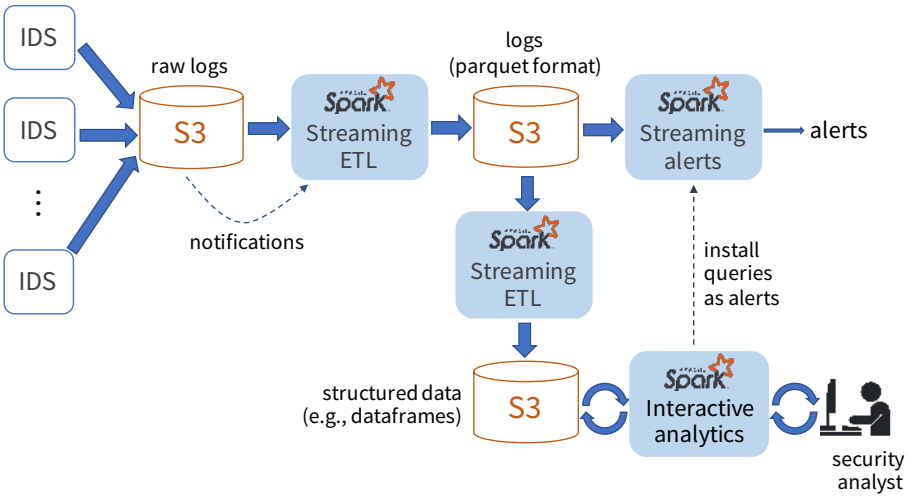


Figure 3.5: Information security platform use case.

3.7 Use Cases

First released in 2016, Structured Streaming has been part of Apache Spark and are supported by various cloud vendors. Databricks started supporting Structured Streaming on its managed cloud service [45] since 2016, and today, the cloud service is running hundreds of production streaming applications at a given time (i.e., applications running 24/7). The largest of these applications ingest over 1 PB of data per month and run on hundreds of servers. Structured Streaming is also used internally by Databricks to monitor services, including the execution of Structured Streaming itself. In this section, we describe three production workloads that leverage various aspects of Structured Streaming, as well as the Databricks internal use case.

3.7.1 Information Security Platform

A large organization has used Structured Streaming to develop a large-scale security platform to enable over 100 analysts to scour through network traffic logs to quickly identify and respond to security incidents, as well as to generate automated alerts. This platform combines streaming with batch and interactive queries and is thus a great example of the system's support for *end-to-end* applications.

Figure 3.5 shows the architecture of the platform. Intrusion Detection Systems (IDSes) monitor all the network traffic in the organization, and output logs to S3. From here, a Structured Streaming jobs ETLs these logs into the Parquet columnar format to reduce the size and improve query speed. Other Structured Streaming jobs process the Parquet logs into additional structured tables (e.g., by joining with other data). Analysts query these tables interactively, either using SQL or Dataframes, to detect and diagnose new attack patterns. If they identify a compromise, they

also look back through historical data to trace previous actions from that attacker. Finally, in parallel, the Parquet logs are processed by another Structured Streaming cluster that generates alerts based on pre-written rules.

The key challenges in realizing this platform are (1) building a robust and scalable streaming pipeline, while (2) providing the analysts with an effective environment to query both fresh and historical data. Using standard tools and services available on AWS, a team of 20 people took over six months to build and deploy a previous version of this platform in production. This previous version had several limitations, including only being able to store a small amount of data for historical queries due to using a traditional data warehouse for the interactive queries. In contrast, a team of five engineers was able to reimplement the platform using Structured Streaming in two weeks. The new platform was simultaneously more scalable and able to support more complex analysis using Spark's ML APIs. Next, we provide a few examples to illustrate the advantages of Structured Streaming that made this possible.

First, Structured Streaming's ability to adaptively vary the batch size enabled the developers to build a streaming pipeline that deals not only with spikes in the workload, but also with failures and code upgrades. Consider a streaming job that goes offline either due to failure or upgrades. When the cluster is brought back online, it will start automatically to process the data all the way back from the moment it went offline. Initially, the cluster will use large batches to maximize the throughput. Once it catches up, the cluster switches to small batches for low latency. This allows operators to regularly upgrade clusters without the fear of excessive downtime.

Second, the ability to join a stream with other streams, as well as with historical tables, has considerably simplified the analysis. Consider the simple task of figuring out which device a TCP connection originates at. It turns out that this task is challenging in the presence of mobile devices, as these devices are given *dynamic* IP addresses every time they join the network. Hence, from TCP logs alone, is not possible to track down the end-points of a connection. With Structured Streaming, an analyst can easily solve this problem. She can simply join the TCP logs with DHCP logs to map the IP address to the MAC address, and then use the organization's internal database of network devices to map the MAC address to a particular machine and user. In addition, users were able to do this join in real time using stateful operators as both the TCP and DHCP logs were being streamed in.

Finally, using the same system for streaming, interactive queries and ETL has provided developers with the ability to quickly iterate and deploy new alerts. In particular, it enables analysts to build and test queries for detecting new attacks on offline data, and then deploy these queries directly on the alerting cluster. In one example, an analyst developed a query to identify exfiltration attacks via DNS. In this attack, malware leaks confidential information from the compromised host by piggybacking this information into DNS requests sent to an external DNS server owned by the attacker. One simplified query to detect such an attack essentially

computes the aggregate size of the DNS requests sent by every host over a time interval. If the aggregate is greater than a given threshold, the query flags the corresponding host as potentially being compromised. The analyst used historical data to set this threshold, so as to achieve the desired balance between false positive and false negative rates. Once satisfied with the result, the analyst simply pushed the query to the alerting cluster. The ability to use the same system and the same API for data analysis and for implementing the alerts led not only to significant engineering cost savings, but also to better security, as it is significantly easier to deploy new rules.

3.7.2 Live Video Stream Monitoring

A large media company is using Structured Streaming to compute quality metrics for their live video traffic and interactively identify delivery problems. Live video delivery is especially challenging because network problems can severely disrupt utility. For pre-recorded video, clients can use large buffers to mask issues, and a degradation at most results in extra buffering time; but for live video, a problem may mean missing a critical moment in a sports match or similar event. This organization collects video quality metrics from clients in real time, performs ETL operations and aggregation using Structured Streaming, then stores the results in a data warehouse. This allows operations engineers to interactively query fresh data to detect and diagnose quality issues (e.g., determine whether an issue is tied to a specific ISP, video server or other cause).

3.7.3 Online Game Performance Analysis

A large gaming company uses Structured Streaming to monitor the latency experienced by players in a popular online game with tens of millions of monthly active users. As in the video use case, high network performance is essential for the user experience when gaming, and repeated problems can quickly lead to player churn. This organization collects latency logs from its game clients to cloud storage and then performs a variety of streaming analyses. For example, one job joins the measurements with a table of Internet Autonomous Systems (ASes) and then aggregates the performance by AS over time to identify poorly performing ASes. When such an AS is identified, the streaming job triggers an alert, and operators can contact the AS in question to remediate the issue.

3.7.4 Databricks Internal Data Pipelines

Databricks has been using Spark since the start of the company to monitor its cloud service, understand the workload, trigger alerts and let engineers interactively debug issues. The monitoring pipeline produces dozens of interactive dashboards as well as structured Parquet tables for ad-hoc queries. These dashboards also play

a key role for business users to understand which customers have increasing or decreasing usage, prioritize feature development, and proactively identify customers experiencing problems.

Databricks built at least three versions of a monitoring pipeline using a combination of batch and streaming APIs starting three years ago, and in all the cases, the major challenges were operational. Despite substantial engineering investment, pipelines could be brittle, experiencing frequent failures when aspects of input data changed (e.g., new schemas or reading from more locations than before), and upgrading them was a daunting exercise. Worse yet, failures and upgrades often resulted in missing data, so manual re-run of jobs to reconstruct the missing data was frequent. Testing pipelines was also challenging due to their reliance on multiple distinct Spark jobs and storage systems. Databricks' experience with Structured Streaming shows that it successfully addresses many of these challenges. Not only were they able to reimplement their pipelines in weeks, but the management overhead decreased drastically. Restartability coupled with adaptive batching, transactional sources/sinks and well-defined consistency semantics have enabled simpler fault recovery, upgrades, and rollbacks to repair old results. Moreover, we can test the same code in batch mode on data samples or use many of the same functions in interactive queries.

Databricks' pipelines with Structured Streaming also combine its batch and streaming capabilities. For example, the pipeline to monitor streaming jobs starts with an ETL job that reads JSON events from Kafka and writes them to a columnar Parquet table in S3. Dozens of other batch and streaming jobs then query this table to produce dashboards and other reports. Because Parquet is a compact and column-oriented format, this architecture consumes drastically *fewer* resources than having every job read directly from Kafka, and simultaneously places less load on the Kafka brokers. Overall, streaming jobs' latencies range from seconds to minutes, and users can also query the Parquet table interactively in seconds.

3.8 Performance Evaluation

In this section, we measure the performance of Structured Streaming using controlled benchmarks. We study performance vs. other systems on the Yahoo! Streaming Benchmark [40], scalability, and the throughput-latency tradeoff with continuous processing.

3.8.1 Performance vs. Other Streaming Systems

To evaluate performance compared to other streaming engines, we used the Yahoo! Streaming Benchmark [40], a widely used workload that has also been evaluated in other open source systems. This benchmark requires systems to read

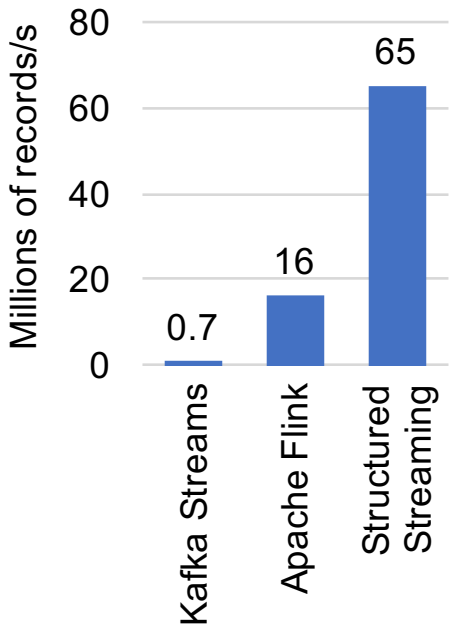


Figure 3.6: vs. Other Systems

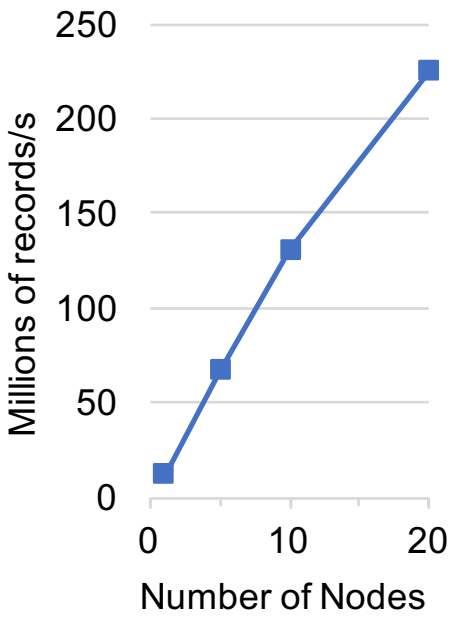


Figure 3.7: System Scaling

Figure 3.8: Throughput results on the Yahoo! benchmark.

ad click events, join them against a static table of ad campaigns by campaign ID, and output counts by campaign on 10-second event-time windows.

We compared Kafka Streams 0.10.2, Apache 1.2.1 and Spark 2.2.0 on a cluster with ten `r3.xlarge` Amazon EC2 workers (each with 4 virtual cores and 30 GB RAM) and one master. For Flink, we used the optimized version of the benchmark published by dataArtisans for a similar cluster [61]. Like in that benchmark, the systems read data from a Kafka cluster running on the workers with 40 partitions (one per core), and write results to Kafka. The original Yahoo! benchmark used Redis to hold the static table for joining ad campaigns, but we found that Redis could be a bottleneck, so we replaced it with a table in each system (a `KTable` in Kafka, a DataFrame in Spark, and an in-memory hash map in Flink).

Figure 3.6 shows each system's maximum stable throughput, i.e., the throughput it can process before a backlog begins to form. We see that streaming system performance can vary significantly. Kafka Streams implements a simple message-passing model through the Kafka message bus, but only attains 700,000 records/second on our 40-core cluster. Apache Flink reaches 16 million records/s, which is similar to the 15 million reported by dataArtisans on a similar cluster [61], though we were only able to achieve this number when we changed the input data to have just one ad per campaign (the default of 10 ads per campaign resulted in lower throughput). Finally, Structured Streaming reaches 65 million records/s, more than 4× the throughput of Flink. This particular Structured Streaming query is implemented using just DataFrame operations with no UDF code. The performance thus comes

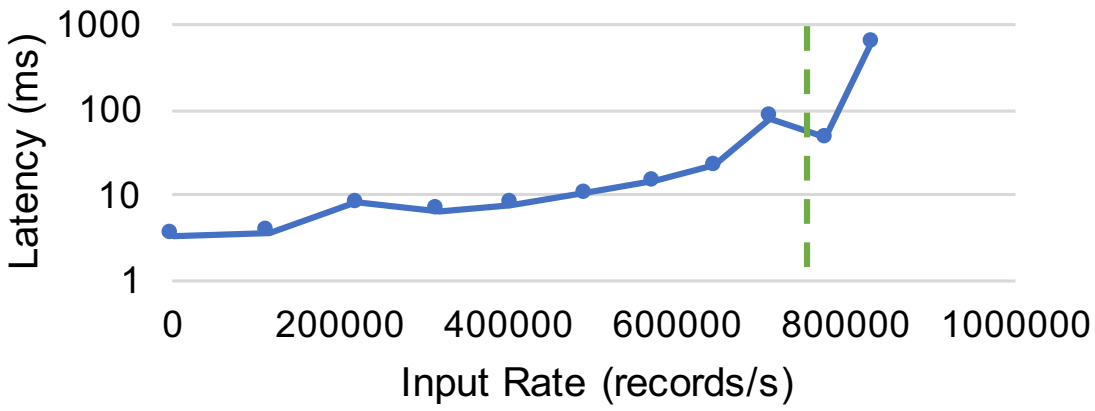


Figure 3.9: Latency of continuous processing vs. input rate. Dashed line shows max throughput in microbatch mode.

solely from Spark SQL's built in execution optimizations, including storing data in a compact binary format and runtime code generation. As pointed out by the authors of Trill [36] and others, execution optimizations can make a large difference in streaming workloads, and many systems based on per-record operations do not maximize performance.

3.8.2 Scalability

Figure 3.7 shows how Structured Streaming's performance scales for the Yahoo! benchmark as we vary the size of our cluster. We used 1, 5, 10 and 20 `c4.2xlarge` Amazon EC2 workers (with 8 virtual cores and 15 GB RAM each) and the same experimental setup as in §3.8.1, including one Kafka partition per core. We see that throughput scales close to linearly, from 11.5 million records/s on 1 node to 225 million records/s on 20 nodes (i.e., 160 cores).

3.8.3 Continuous Processing

We benchmarked Spark's in-development continuous processing mode on a 4-core server to show the latency-throughput tradeoffs it can achieve. (Since partitions run independently in this mode, we expect the latency to stay the same as more nodes are added.) Figure 3.9 shows the results for a `map` operation reading from Kafka, with the dashed line showing the maximum throughput achievable by microbatch mode. We see that continuous mode is able to achieve much lower latency without a large drop in throughput (e.g., less than 10 ms latency at half the maximum throughput of microbatching). Its maximum stable throughput is also slightly higher because microbatch mode incurs latency due to task scheduling.

3.9 Related Work

Structured Streaming builds on many existing systems for stream processing and big data analytics, including Spark SQL's DataFrame API, Spark Streaming [133], Dataflow [12], incremental query systems [29, 107, 135] and distributed stream processing [57]. At a high level, the main contributions of this work are:

- An account of real-world user challenges with streaming systems, including operational challenges that are not always discussed in research (§3.2).
- A simple, declarative programming model that *incrementalizes* a widely used batch API (Spark DataFrames/SQL) to provide similar capabilities to Dataflow [12] and other streaming systems, and experience with this model in real-world applications.
- An execution engine providing high throughput, fault tolerance, and unique operational features such as rollback and hybrid batch and streaming execution (§3.6.4). With the rest of Spark, this lets users easily build *end-to-end* business applications.

From an API standpoint, the closest work is incremental query systems [29, 107, 135], including recent distributed systems such as Stateful Bulk Processing [80] and Naiad [87]. Structured Streaming's API is an extension of Spark SQL, including its declarative DataFrame interface for programmatic construction of relational queries. Apache Flink also recently added a table API (currently in beta) for defining relational queries that can map to either streaming or batch execution [54], but this API lacks some of the features of Structured Streaming, such as custom stateful operators (§3.4.3).

Other recent streaming systems have language-integrated APIs that operate at a lower, more "imperative" level. In particular, Spark Streaming [133], Google Dataflow [12] and Flink's DataStream API [53] provide various functional operators but require users to choose the right DAG of operators to implement a particular incrementalization strategy (e.g., when to pass on deltas versus complete results); essentially, these are equivalent to writing a physical execution plan. Structured Streaming's API is simpler for users who are not experts on incrementalization. Structured Streaming adopts the definitions of event time, processing time, watermarks and triggers from Dataflow but incorporates them in an incremental model.

For execution, Structured Streaming uses concepts similar to discretized streams for microbatch mode [133] and traditional streaming engines for continuous processing mode [57, 37, 6]. It also builds on an analytical engine for performance like Trill [36]. The most unique contribution here is the integration of batch and streaming queries to enable sophisticated end-to-end applications. As described in §3.7, Structured Streaming users can easily write applications that combine batch, interactive and stream processing using the same code (e.g., security log analysis).

In addition, they leverage powerful operational features such as run-once triggers (running a streaming application "discontinuously" as batch jobs to retain its transactional features but lower costs), code updates, and batch processing to handle backlogs or code rollbacks (§3.6.4).

3.10 Conclusion

Stream processing is a powerful tool, but streaming systems are still difficult to use, operate and integrate into larger applications. We designed Structured Streaming to simplify all three of these tasks while integrating with the rest of Apache Spark. Structured Streaming provides a simple declarative API that just incrementalizes a Spark SQL computation, and a fast execution engine built on Spark SQL that outperforms other open source systems. Experience across hundreds of production use cases shows that users can leverage the system to build sophisticated business applications that combine batch, streaming and interactive queries in complex ways.

Chapter 4

GraphX: Graph Computation on Spark

The previous two chapters develop relational query processing in a new context, with programming APIs beyond SQL and continuous execution modes against both batch and streaming data. This chapter develops GraphX and explores the design space in building a graph computation framework on top of a distributed dataflow framework. We first provide background in the property graph data model and the graph-parallel abstraction. We then recast graph-parallel operations as dataflow operations to enable graph processing on top of Spark. We conduct experiments to demonstrate that GraphX matches the performance of specialized graph processing systems, while enabling a wider range of computation.

4.1 Introduction

The growing scale and importance of graph data has driven the development of numerous specialized *graph processing* systems including Pregel [85], PowerGraph [58], and many others [34, 39, 119]. By exposing specialized abstractions backed by graph-specific optimizations, these systems can naturally express and efficiently execute iterative graph algorithms like PageRank [100] and community detection [79] on graphs with billions of vertices and edges. As a consequence, graph processing systems typically outperform general-purpose distributed dataflow frameworks like Hadoop MapReduce by orders of magnitude [58, 83].

While the restricted focus of these systems enables a wide range of system optimizations, it also comes at a cost. Graphs are only part of the larger analytics process which often combines graphs with unstructured and tabular data. Consequently, analytics pipelines (*e.g.,* Figure 4.11) are forced to compose multiple systems which increases complexity and leads to unnecessary data movement and duplication. Furthermore, in pursuit of performance, graph processing systems often abandon fault tolerance in favor of snapshot recovery. Finally, as specialized systems, graph

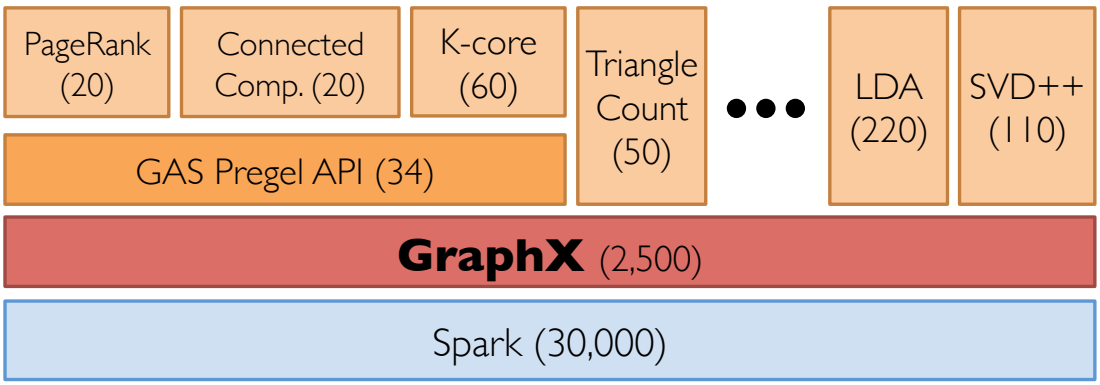


Figure 4.1: **GraphX** is a thin layer on top of the Spark general-purpose dataflow framework (lines of code).

processing frameworks do not generally enjoy the broad support of distributed dataflow frameworks.

In contrast, general-purpose distributed dataflow frameworks (*e.g.*, MapReduce [47], Spark [132], Dryad [68]) expose rich dataflow operators (*e.g.*, map, reduce, group-by, join), are well suited for analyzing unstructured and tabular data, and are widely adopted. However, *directly* implementing iterative graph algorithms using dataflow operators can be challenging, often requiring multiple stages of complex joins. Furthermore, the *general-purpose* join and aggregation strategies defined in distributed dataflow frameworks do not leverage the common patterns and structure in iterative graph algorithms and therefore miss important optimization opportunities.

Historically, graph processing systems evolved separately from distributed dataflow frameworks for several reasons. First, the early emphasis on single stage computation and on-disk processing in distributed dataflow frameworks (*e.g.*, MapReduce) limited their applicability to iterative graph algorithms which repeatedly and randomly access subsets of the graph. Second, early distributed dataflow frameworks did not expose fine-grained control over the data partitioning, hindering the application of graph partitioning techniques. However, new in-memory distributed dataflow frameworks (*e.g.*, Spark and Naiad) expose control over data partitioning and in-memory representation, addressing some of these limitations.

Given these developments, we believe there is an opportunity to unify advances in graph processing systems with advances in dataflow systems enabling a single system to address the entire analytics pipeline. In this chapter we explore the design of graph processing systems on top of general purpose distributed dataflow systems. We argue that by identifying the essential dataflow patterns in graph computation and recasting optimizations in graph processing systems as dataflow optimizations we can recover the advantages of specialized graph processing systems within a general-purpose distributed dataflow framework. To support this argument we

introduce GraphX, an efficient graph processing framework embedded within the Spark [132] distributed dataflow system.

GraphX presents a familiar, expressive graph API (Section 4.3). Using the GraphX API we implement a variant of the popular Pregel abstraction as well as a range of common graph operations. Unlike existing graph processing systems, the GraphX API enables the *composition* of graphs with unstructured and tabular data and permits the same physical data to be viewed both as a graph and as collections without data movement or duplication. For example, using GraphX it is easy to join a social graph with user comments, apply graph algorithms, and expose the results as either collections or graphs to other procedures (*e.g.,* visualization or rollup). Consequently, GraphX enables users to adopt the computational pattern (graph or collection) that is best suited for the current task without sacrificing performance or flexibility.

We built GraphX as a library on top of Spark (Figure 4.1) by encoding graphs as collections and then expressing the GraphX API on top of standard dataflow operators. GraphX requires no modifications to Spark, revealing a general method to embed graph computation within distributed dataflow frameworks and distill graph computation to a specific *join–map–group-by* dataflow pattern. By reducing graph computation to a specific pattern we identify the critical path for system optimization.

However, naively encoding graphs as collections and executing iterative graph computation using general-purpose dataflow operators can be slow and inefficient. To achieve performance parity with specialized graph processing systems, GraphX introduces a range of optimizations (Section 4.4) both in how graphs are encoded as collections as well as the execution of the common dataflow operators. Flexible vertex-cut partitioning is used to encode graphs as horizontally partitioned collections and match the state of the art in distributed graph partitioning. GraphX recasts system optimizations developed in the context of graph processing systems as join optimizations (*e.g.,* CSR indexing, join elimination, and join-site specification) and materialized view maintenance (*e.g.,* vertex mirroring and delta updates) and applies these techniques to the Spark dataflow operators. By leveraging logical partitioning and lineage, GraphX achieves low-cost fault tolerance. Finally, by exploiting immutability GraphX reuses indices across graph and collection views and over multiple iterations, reducing memory overhead and improving system performance.

We evaluate GraphX on real-world graphs and compare against direct implementations of graph algorithms using the Spark dataflow operators as well as implementations using specialized graph processing systems. We demonstrate that GraphX can achieve performance parity with specialized graph processing systems while preserving the advantages of a general-purpose dataflow framework. In summary, the contributions of this chapter are:

1. an integrated graph and collections API which is sufficient to express existing graph abstractions and enable a much wider range of computation.

2. an embedding of vertex-cut partitioned graphs in horizontally partitioned collections and the GraphX API in a small set of general-purpose dataflow operators.
3. distributed join and materialized view optimizations that enable general-purpose distributed dataflow frameworks to execute graph computation at performance parity with specialized graph systems.
4. a large-scale evaluation on real graphs and common benchmarking algorithms comparing GraphX against widely used graph processing systems.

4.2 Background

In this section we review the design trade-offs and limitations of graph processing systems and distributed dataflow frameworks. At a high level, graph processing systems define computation at the granularity of vertices and their neighborhoods and exploit the sparse dependency structure pre-defined by the graph. In contrast, general-purpose distributed dataflow frameworks define computation as dataflow operators at either the granularity of individual items (*e.g.,* filter, map) or across entire collections (*i.e.,* operations like non-broadcast join that require a shuffle).

4.2.1 The Property Graph Data Model

Graph processing systems represent graph structured data as a **property graph** [109], which associates user-defined properties with each vertex and edge. The properties can include meta-data (*e.g.,* user profiles and time stamps) and program state (*e.g.,* the PageRank of vertices or inferred affinities). Property graphs derived from natural phenomena such as social networks and web graphs often have highly skewed, power-law degree distributions and orders of magnitude more edges than vertices [79].

In contrast to dataflow systems whose operators (*e.g.,* join) can span multiple collections, operations in graph processing systems (*e.g.,* vertex programs) are typically defined with respect to a *single* property graph with a *pre-declared, sparse structure*. While this restricted focus facilitates a range of optimizations (Section 4.2.3), it also complicates the expression of analytics tasks that may span multiple graphs and sub-graphs.

4.2.2 The Graph-Parallel Abstraction

Algorithms ranging from PageRank to latent factor analysis iteratively transform vertex properties based on the properties of adjacent vertices and edges. This common pattern of *iterative local transformations* forms the basis of the graph-parallel abstraction. In the graph-parallel abstraction [58], a user-defined **vertex program**

Listing 4.1: **PageRank in Pregel**: computes the sum of the inbound messages, updates the PageRank value for the vertex, and then sends the new weighted PageRank value to neighboring vertices. Finally, if the PageRank did not change the vertex program votes to halt.

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

is instantiated concurrently for each vertex and interacts with adjacent vertex programs through messages (*e.g.,* Pregel [85]) or shared state (*e.g.,* PowerGraph [58]). Each vertex program can read and modify its vertex property and in some cases [58, 83] adjacent vertex properties. When all vertex programs vote to halt the program terminates.

As a concrete example, in Listing 4.1 we express the PageRank algorithm as a Pregel vertex program. The vertex program for the vertex v begins by summing the messages encoding the weighted PageRank of neighboring vertices. The PageRank is updated using the resulting sum and is then broadcast to its neighbors (weighted by the number of links). Finally, the vertex program assesses whether it has converged (locally) and votes to halt.

The extent to which vertex programs run concurrently differs across systems. Most systems (*e.g.,* [34, 58, 85, 110]) adopt the bulk synchronous execution model, in which all vertex programs run concurrently in a sequence of super-steps. Some systems (*e.g.,* [58, 83, 119]) also support an asynchronous execution model that mitigates the effect of stragglers by running vertex programs as resources become available. However, the gains due to an asynchronous programming model are often offset by the additional complexity and so we focus on the bulk-synchronous model and rely on system level techniques (*e.g.,* pipelining and speculation) to address stragglers.

While the graph-parallel abstraction is well suited for iterative graph algorithms that respect the static neighborhood structure of the graph (*e.g.,* PageRank), it is not well suited to express computation where disconnected vertices interact or where computation changes the graph structure. For example, tasks such as graph construction from raw text or unstructured data, graph coarsening, and analysis

Listing 4.2: **Gather-Apply-Scatter (GAS) PageRank**: The gather phase combines inbound messages. The apply phase consumes the final message sum and updates the vertex property. The scatter phase defines the message computation for each edge.

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

that spans multiple graphs are difficult to express in the vertex centric programming model.

4.2.3 Graph System Optimizations

The restrictions imposed by the graph-parallel abstraction along with the sparse graph structure enable a range of important system optimizations.

The GAS Decomposition: Gonzalez et al. [58] observed that most vertex programs interact with neighboring vertices by collecting messages in the form of a generalized commutative associative sum and then broadcasting new messages in an inherently parallel loop. They proposed the GAS decomposition which splits vertex programs into three *data-parallel* stages: Gather, Apply, and Scatter. In Listing 4.2 we decompose the PageRank vertex program into the Gather, Apply, and Scatter stages.

The GAS decomposition leads to a *pull-based* model of message computation: the system asks the vertex program for value of the message between *adjacent vertices* rather than the user sending messages directly from the vertex program. As a consequence, the GAS decomposition enables vertex-cut partitioning, improved work balance, serial edge-iteration [110], and reduced data movement. However, the GAS decomposition also prohibits direct communication between vertices that are not adjacent in the graph and therefore hinders the expression of more general communication patterns.

Graph Partitioning: Graph processing systems apply a range of graph-partitioning algorithms [74] to minimize communication and balance computation. Gonzalez et al. [58] demonstrated that vertex-cut [51] partitioning performs well on many large natural graphs. Vertex-cut partitioning evenly assigns edges to machines in a way that minimizes the number of times each vertex is cut.

Mirror Vertices: Often high-degree vertices will have multiple neighbors on the same remote machine. Rather than sending multiple, typically identical, messages across the network, graph processing systems [58, 83, 90, 106] adopt mirroring techniques in which a single message is sent to the mirror and then forwarded to

all the neighbors. Graph processing systems exploit the static graph structure to reuse the mirror data structures.

Active Vertices: As graph algorithms proceed, vertex programs within a graph converge at different rates, leading to rapidly shrinking working sets (the collection of active vertex programs). Recent systems [50, 58, 83, 85] track active vertices and eliminate data movement and unnecessary computation for vertices that have converged. In addition, these systems typically maintain efficient densely packed data-structures (*e.g.,* compressed sparse row (CSR)) that enable constant-time access to the local edges adjacent to active vertices.

4.3 The GraphX Programming Abstraction

We now revisit graph computation from the perspective of a general-purpose dataflow framework. We recast the property graph data model as collections and the graph-parallel abstraction as a specific pattern of dataflow operators. In the process we reveal the essential structure of graph-parallel computation and identify the key operators required to execute graph algorithms efficiently.

4.3.1 Property Graphs as Collections

The property graph, described in Section 4.2.1, can be logically represented as a pair of vertex and edge property collections. The *vertex collection* contains the vertex properties uniquely keyed by the vertex identifier. In the GraphX system, vertex identifiers are 64-bit integers which may be derived externally (*e.g.,* user ids) or by applying a hash function to the vertex property (*e.g.,* page URL). The *edge collection* contains the edge properties keyed by the source and destination vertex identifiers.

By reducing the property graph to a pair of collections we make it possible to compose graphs with other collections in a distributed dataflow framework. Operations like adding additional vertex properties are naturally expressed as joins against the vertex property collection. The process of analyzing the results of graph computation (*i.e.,* the final vertex and edge properties) and comparing properties across graphs becomes as simple as analyzing and joining the corresponding collections. Both of these tasks are routine in the broader scope of graph analytics but are not well served by the graph parallel abstraction.

New property graphs can be constructed by composing different vertex and edge property collections. For example, we can construct logically distinct graphs with separate vertex properties (*e.g.,* one storing PageRanks and another storing connected component membership) while sharing the same edge collection. This may appear to be a small accomplishment, but the tight integration of vertices and edges in specialized graph processing systems often hinders even this basic form of reuse. In addition, graph-specific index data structures can be shared across

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

Listing 4.3: **Constructing Triplets in SQL**: The column `P` represents the properties in the vertex and edge property collections.

graphs with common vertex and edge collections, reducing storage overhead and improving performance.

4.3.2 Graph Computation as Dataflow Ops.

The normalized representation of a property graph as a pair of vertex and edge property collections allows us to embed graphs in a distributed dataflow framework. In this section we describe how dataflow operators can be composed to express graph computation.

Graph-parallel computation, introduced in Section 4.2.2, is the process of computing aggregate properties of the neighborhood of each vertex (*e.g.*, the sum of the PageRanks of neighboring vertices weighted by the edge values). We can express graph-parallel computation in a distributed dataflow framework as a sequence of *join stages* and *group-by stages* punctuated by map operations.

In the *join stage*, vertex and edge properties are joined to form the *triplets view*[1] consisting of each edge and its corresponding source and destination vertex properties. The triplets view is best illustrated by the SQL statement in Listing 4.3, which constructs the triplets view as a three way join keyed by the source and destination vertex ids.

In the *group-by stage*, the triplets are grouped by source or destination vertex to construct the *neighborhood* of each vertex and compute aggregates. For example, to compute the PageRank of a vertex we would execute:

```
SELECT t.dstId, 0.15+0.85*sum(t.srcP*t.eP)
FROM triplets AS t GROUP BY t.dstId
```

By iteratively applying the above query to update the vertex properties until they converge, we can calculate the PageRank of each vertex.

These two stages capture the GAS decomposition described in Section 4.2.3. The group-by stage *gathers* messages destined to the same vertex, an intervening map operation *applies* the message sum to update the vertex property, and the join stage *scatters* the new vertex property to all adjacent vertices.

Similarly, we can implement the GAS decomposition of the Pregel abstraction by

[1] The *triplet* terminology derives from the classic Resource Description Framework (RDF), discussed in Section 4.6.

Listing 4.4: **Graph Operators:** transform vertex and edge collections.

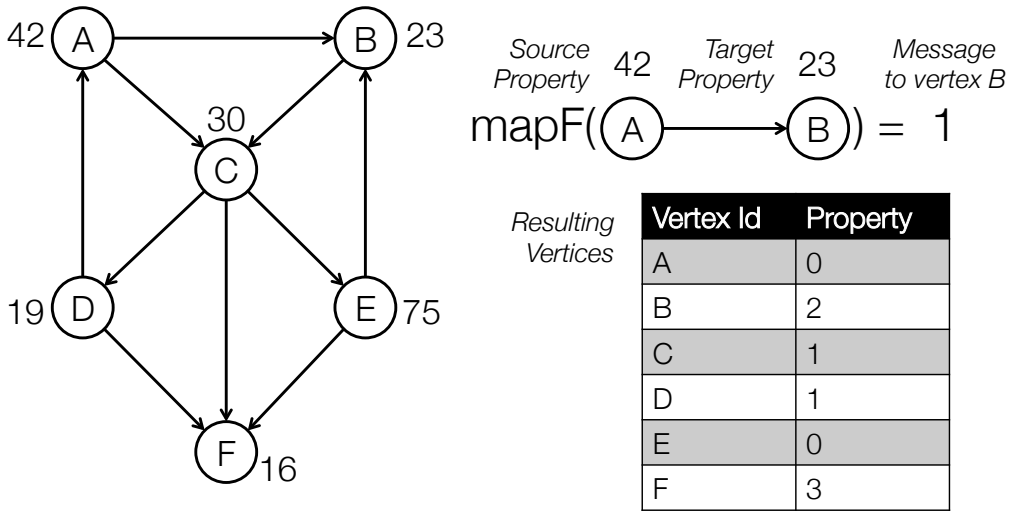
```
class Graph[V, E] {
  // Constructor
  def Graph(v: Collection[(Id, V)],
            e: Collection[(Id, Id, E)])
  // Collection views
  def vertices: Collection[(Id, V)]
  def edges: Collection[(Id, Id, E)]
  def triplets: Collection[Triplet]
  // Graph-parallel computation
  def mrTriplets(f: (Triplet) => M,
      sum: (M, M) => M): Collection[(Id, M)]
  // Convenience functions
  def mapV(f: (Id, V) => V): Graph[V, E]
  def mapE(f: (Id, Id, E) => E): Graph[V, E]
  def leftJoinV(v: Collection[(Id, V)],
      f: (Id, V, V) => V): Graph[V, E]
  def leftJoinE(e: Collection[(Id, Id, E)],
      f: (Id, Id, E, E) => E): Graph[V, E]
  def subgraph(vPred: (Id, V) => Boolean,
      ePred: (Triplet) => Boolean)
    : Graph[V, E]
  def reverse: Graph[V, E]
}
```

iteratively composing the join and group-by stages with data-parallel map stages. Each iteration begins by executing the join stage to bind active vertices with their outbound edges. Using the triplets view, messages are computed along each triplet in a map stage and then aggregated at their destination vertex in a group-by stage. Finally, the messages are received by the vertex programs in a map stage over the vertices.

The dataflow embedding of the Pregel abstraction demonstrates that graph-parallel computation can be expressed in terms of a simple sequence of *join* and *group-by* dataflow operators. Additionally, it stresses the need to efficiently maintain the triplets view in the join stage and compute the neighborhood aggregates in the group-by stage. Consequently, these stages are the focus of performance optimization in graph processing systems. We describe how to implement them efficiently in Section 4.4.

4.3.3 GraphX Operators

The GraphX programming abstraction extends the Spark dataflow operators by introducing a small set of specialized *graph operators*, summarized in Listing 4.4.



Vertex Id	Property
A	0
B	2
C	1
D	1
E	0
F	3

```
val graph: Graph[User, Double]
def mapUDF(t: Triplet[User, Double]) =
  if (t.src.age > t.dst.age) 1 else 0
def reduceUDF(a: Int, b: Int): Int = a + b
val seniors: Collection[(Id, Int)] =
  graph.mrTriplets(mapUDF, reduceUDF)
```

Figure 4.2: **Example use of mrTriplets**: Compute the number of older followers of each vertex.

The `Graph` constructor logically binds together a pair of vertex and edge property collections into a property graph. It also verifies the integrity constraints: that every vertex occurs only once and that edges do not link missing vertices. Conversely, the `vertices` and `edges` operators expose the graph's vertex and edge property collections. The `triplets` operator returns the triplets view (Listing 4.3) of the graph as described in Section 4.3.2. If a triplets view already exists, the previous triplets are incrementally maintained to avoid a full join (see Section 4.4.2).

The `mrTriplets` (Map Reduce Triplets) operator encodes the essential two-stage process of graph-parallel computation defined in Section 4.3.2. Logically, the `mrTriplets` operator is the composition of the *map* and *group-by* dataflow operators on the triplets view. The user-defined map function is applied to each triplet, yielding a value (*i.e.,* a message of type `M`) which is then aggregated at the destination vertex using the user-defined binary aggregation function as illustrated in the following:

```
SELECT t.dstId, reduceF(mapF(t)) AS msgSum
FROM triplets AS t GROUP BY t.dstId
```

The `mrTriplets` operator produces a collection containing the sum of the inbound messages keyed by the destination vertex identifier. For example, in Figure 4.2 we use the `mrTriplets` operator to compute a collection containing the number of

Listing 4.5: **GraphX Enhanced Pregel:** An implementation of the Pregel abstraction using the GraphX API.

```
def Pregel(g: Graph[V, E],
     vprog: (Id, V, M) => V,
     sendMsg: (Triplet) => M,
     gather: (M, M) => M): Collection[V] = {
 // Set all vertices as active
 g = g.mapV((id, v) => (v, halt=false))
 // Loop until convergence
 while (g.vertices.exists(v => !v.halt)) {
   // Compute the messages
   val msgs: Collection[(Id, M)] =
     // Restrict to edges with active source
     g.subgraph(ePred=(s,d,sP,eP,dP)=>!sP.halt)
     // Compute messages
      .mrTriplets(sendMsg, gather)
   // Receive messages and run vertex program
   g = g.leftJoinV(msgs).mapV(vprog)
 }
 return g.vertices
}
```

older followers for each user in a social network. Because the resulting collection contains a subset of the vertices in the graph it can reuse the same indices as the original vertex collection.

Finally, Listing 4.4 contains several functions that simply perform a dataflow operation on the vertex or edge collections. We define these functions only for caller convenience; they are not essential to the abstraction and can easily be defined using standard dataflow operators. For example, `mapV` is defined as follows:

```
g.mapV(f) ≡ Graph(g.vertices.map(f), g.edges)
```

In Listing 4.5 we use the GraphX API to implement a GAS decomposition of the Pregel abstraction. We begin by initializing the vertex properties with an additional field to track active vertices (those that have not voted to halt). Then, while there are active vertices, messages are computed using the `mrTriplets` operator and the vertex program is applied to the resulting message sums.

By expressing message computation as an *edge-parallel* map operation followed by a commutative associative aggregation, we leverage the GAS decomposition to mitigate the cost of high-degree vertices. Furthermore, by exposing the entire triplet to the message computation we can simplify algorithms like connected components. However, in cases where the entire triplet is not needed (*e.g.,* PageRank which requires only the source property) we rely on UDF bytecode inspection (see Section 4.4.3) to automatically drop unused fields from join.

Listing 4.6: **Connected Components:** For each vertex we compute the lowest reachable vertex id using Pregel.

```
def ConnectedComp(g: Graph[V, E]) = {
  g = g.mapV(v => v.id) // Initialize vertices
  def vProg(v: Id, m: Id): Id = {
    if (v == m) voteToHalt(v)
    return min(v, m)
  }
  def sendMsg(t: Triplet): Id =
    if (t.src.cc < t.dst.cc) t.src.cc
    else None // No message required
  def gatherMsg(a: Id, b: Id): Id = min(a, b)
  return Pregel(g, vProg, sendMsg, gatherMsg)
}
```

In Listing 4.6 we use the GraphX variant of Pregel to implement the connected components algorithm. The connected components algorithm computes the lowest reachable vertex id for each vertex. We initialize the vertex property of each vertex to equal its id using `mapV` and then define the three functions required to use the GraphX Pregel API. The `sendMsg` function leverages the triplet view of the edge to only send a message to neighboring vertices when their component id should change. The `gatherMsg` function computes the minimum of the inbound message values and the vertex program (`vProg`) determines the new component id.

Combining Graph and Collection Operators: Often groups of connected vertices are better modeled as a single vertex. In these cases, it can be helpful coarsen the graph by aggregating connected vertices that share a common characteristic (*e.g.,* web domain) to derive a new graph (*e.g.,* the domain graph). We use the GraphX abstraction to implement graph coarsening in Listing 4.7.

The coarsening operation takes an edge predicate and a vertex aggregation function and collapses all edges that satisfy the predicate, merging their respective vertices. The edge predicate is used to first construct the subgraph of edges that are to be collapsed (*i.e.,* modifying the graph structure). Then the graph-parallel connected components algorithm is run on the subgraph. Each connected component corresponds to a super-vertex in the new coarsened graph with the component id being the lowest vertex id in the component. The super-vertices are constructed by aggregating all the vertices with the same component id using a data-parallel aggregation operator. Finally, we update the edges to link together super-vertices and generate the new graph for subsequent graph-parallel computation.

The *coarsen* operator demonstrates the power of a unified abstraction by combining both data-parallel and graph-parallel operators in a single graph-analytics task.

Listing 4.7: **Coarsen:** The *coarsening* operator merges vertices connected by edges that satisfy the edge predicate.

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

4.4 The GraphX System

GraphX achieves performance parity with specialized graph processing systems by recasting the graph-specific optimizations of Section 4.2.3 as optimizations on top of a small set of standard dataflow operators in Spark. In this section we describe these optimizations in the context of classic techniques in traditional database systems including indexing, incremental view maintenance, and join optimizations. Along the way, we quantify the effectiveness of each optimization; readers are referred to Section 4.5 for details on datasets and experimental setup.

4.4.1 Distributed Graph Representation

GraphX represents graphs internally as a pair of vertex and edge collections built on the Spark RDD abstraction. These collections introduce indexing and graph-specific partitioning as a layer on top of RDDs. Figure 4.3 illustrates the physical representation of the horizontally partitioned vertex and edge collections and their indices.

The **vertex collection** is hash-partitioned by the vertex ids. To support frequent joins across vertex collections, vertices are stored in a local hash index within each

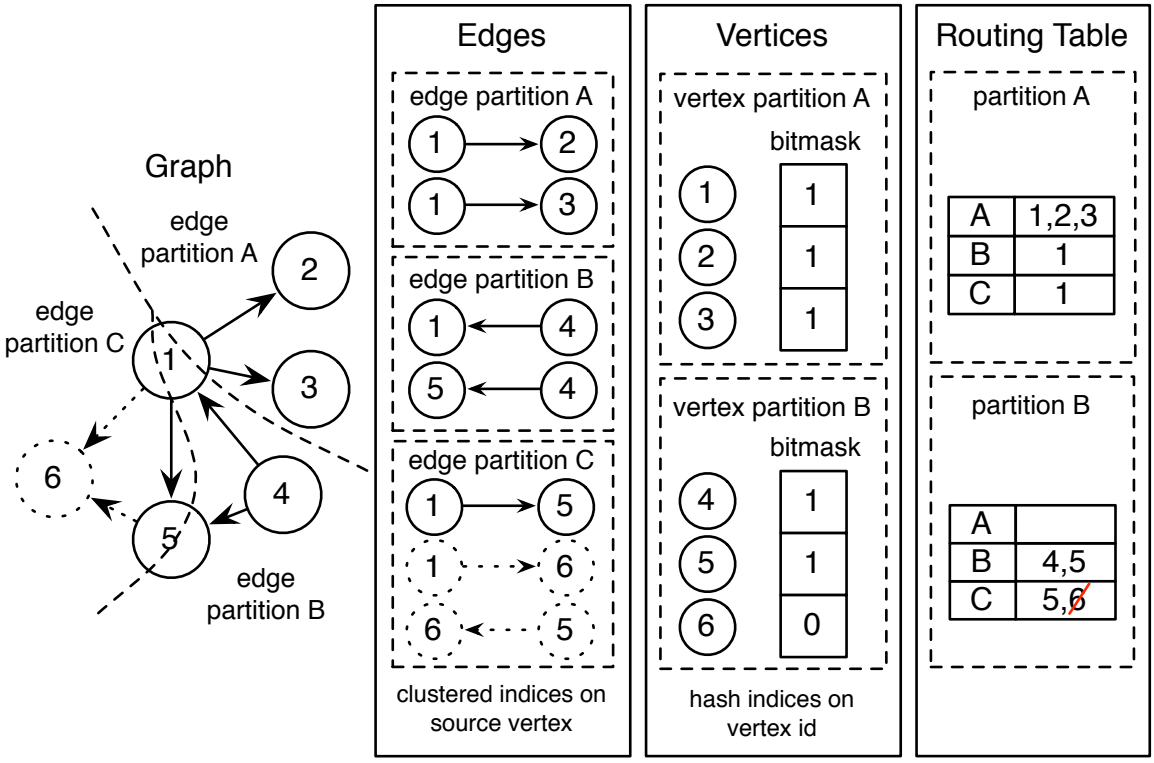


Figure 4.3: **Distributed Graph Representation**: The graph (left) is represented as a vertex and an edge collection (right). The edges are divided into three edge partitions by applying a partition function (*e.g.,* 2D Partitioning). The vertices are partitioned by vertex id. Co-partitioned with the vertices, GraphX maintains a routing table encoding the edge partitions for each vertex. If vertex 6 and adjacent edges (shown with dotted lines) are restricted from the graph (*e.g.,* by `subgraph`), they are removed from the corresponding collection by updating the bitmasks thereby enabling index reuse.

partition (Section 4.4.2). Additionally, a bitmask stores the visibility of each vertex, enabling soft deletions to promote index reuse (Section 4.4.3).

The **edge collection** is horizontally partitioned by a user-defined partition function. GraphX enables *vertex-cut* partitioning, which minimizes communication in natural graphs such as social networks and web graphs [58]. By default edges are assigned to partitions based on the partitioning of the input collection (*e.g.,* the original placement on HDFS). However, GraphX provides a range of built-in partitioning functions, including a 2D hash partitioner with strong upper bounds [35] on the communication complexity of operators like `mrTriplets`. This flexibility in edge placement is enabled by the *routing table*, described in Section 4.4.2. For efficient lookup of edges by their source and target vertices, the edges within a partition are clustered by source vertex id using a *compressed sparse row* (CSR) [111]

representation and hash-indexed by their target id. Section 4.4.3 discusses how these indices accelerate iterative computation.

Index Reuse: GraphX inherits the immutability of Spark and therefore all graph operators *logically* create new collections rather than destructively modifying existing ones. As a result, derived vertex and edge collections can often share indices to reduce memory overhead and accelerate local graph operations. For example, the hash index on vertices enables fast aggregations, and the resulting aggregates share the index with the original vertices.

In addition to reducing memory overhead, shared indices enable faster joins. Vertex collections sharing the same index (*e.g.*, the vertices and the messages from `mrTriplets`) can be joined by a coordinated scan, similar to a merge join, without requiring any index lookups. In our benchmarks, index reuse reduces the per-iteration runtime of PageRank on the Twitter graph by 59%.

The GraphX operators try to maximize index reuse. Operators that do not modify the graph structure (*e.g.*, `mapV`) automatically preserve indices. To reuse indices for operations that *restrict* the graph structure (*e.g.*, `subgraph`), GraphX relies on bitmasks to construct restricted views. In cases where index reuse could lead to decreased efficiency (*e.g.*, when a graph is highly filtered), GraphX uses the *reindex* operator to build new indices.

4.4.2 Implementing the Triplets View

As described in Section 4.3.2, a key stage in graph computation is constructing and maintaining the *triplets view*, which consists of a three-way join between the source and destination vertex properties and the edge properties.

Vertex Mirroring: Because the vertex and edge property collections are partitioned independently, the join requires data movement. GraphX performs the three-way join by shipping the vertex properties across the network to the edges, thus setting the edge partitions as the *join sites* [84]. This approach substantially reduces communication for two reasons. First, real-world graphs commonly have orders of magnitude more edges than vertices. Second, a single vertex may have many edges in the same partition, enabling substantial reuse of the vertex property.

Multicast Join: While *broadcast join* in which all vertices are sent to each edge partition would ensure joins occur on edge partitions, it could still be inefficient since most partitions require only a small subset of the vertices to complete the join. Therefore, GraphX introduces a *multicast join* in which each vertex property is sent only to the edge partitions that contain adjacent edges. For each vertex GraphX maintains the set of edge partitions with adjacent edges. This join site information is stored in a *routing table* which is co-partitioned with the vertex collection (Figure 4.3). The routing table is associated with the edge collection and constructed lazily upon first instantiation of the triplets view.

The flexibility in partitioning afforded by the multicast join strategy enables

more sophisticated application-specific graph partitioning techniques. For example, by adopting a per-city partitioning scheme on the Facebook social network graph Ugander et al. [122] showed a 50.5% reduction in query time. In Section 4.5.1 we exploit the optimized partitioning of our sample datasets to achieve up to 56% reduction in runtime and 5.8× reduction in communication compared to a 2D hash partitioning.

Partial Materialization: Vertex replication is performed eagerly when vertex properties change, but the local joins at the edge partitions are left unmaterialized to avoid duplication. Instead, mirrored vertex properties are stored in hash maps on each edge partition and referenced when constructing triplets.

Incremental View Maintenance: Iterative graph algorithms often modify only a subset of the vertex properties in each iteration. We therefore apply *incremental view maintenance* to the triplets view to avoid unnecessary movement of unchanged data. After each graph operation, we track which vertex properties have changed since the triplets view was last constructed. When the triplets view is next accessed, only the changed vertices are re-routed to their edge-partition join sites and the local mirrored values of the unchanged vertices are reused. This functionality is managed automatically by the graph operators.

Figure 4.4 illustrates the impact of incremental view maintenance for both PageRank and connected components on the Twitter graph. In the case of PageRank, where the number of active vertices decreases slowly because the convergence threshold was set to 0, we see only moderate gains. In contrast, for connected components most vertices are within a short distance of each other and converge quickly, leading to a substantial reduction in communication from incremental view maintenance. Without incremental view maintenance, the triplets view would need to be reconstructed from scratch every iteration, and communication would remain at its peak throughout the computation.

4.4.3 Optimizations to mrTriplets

GraphX incorporates two additional query optimizations for the `mrTriplets` operator: *filtered index scanning* and *automatic join elimination*.

Filtered Index Scanning

The first stage of the `mrTriplets` operator logically involves a scan of the triplets view to apply the user-defined map function to each triplet. However, as iterative graph algorithms converge, their working sets tend to shrink, and the map function skips all but a few triplets. In particular, the map function only needs to operate on triplets containing vertices in the *active set*, which is defined by an application-specific predicate. Directly scanning all triplets becomes increasingly wasteful as the active set shrinks. For example, in the last iteration of connected components on the Twitter graph, only a few of the vertices are still active. However, to execute

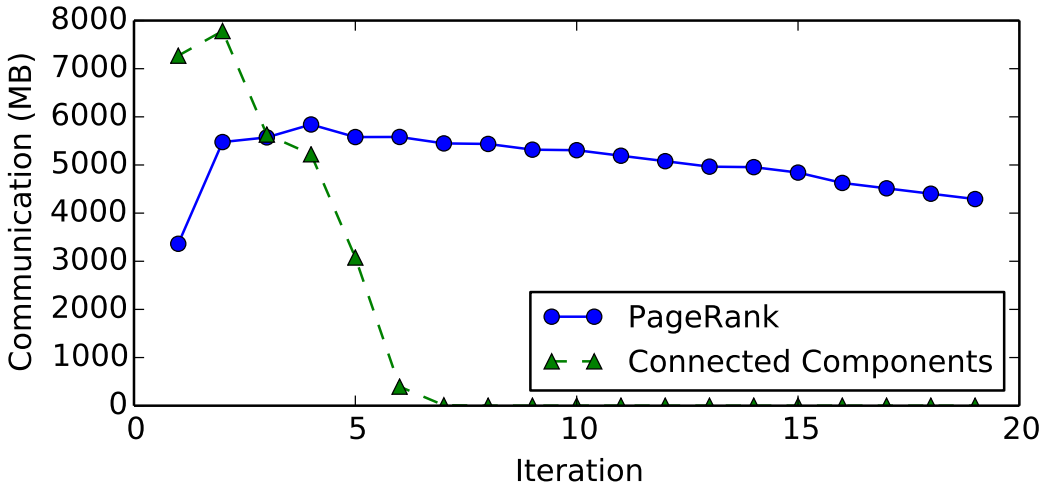


Figure 4.4: **Impact of incrementally maintaining the triplets view**: For both PageRank and connected components, as vertices converge, communication decreases due to incremental view maintenance. The initial rise in communication is due to message compression (Section 4.4.4); many PageRank values are initially the same.

`mrTriplets` we still must sequentially scan 1.5 billion edges and check whether their vertices are in the active set.

To address this problem, we introduced an indexed scan for the triplets view. The application expresses the current active set by restricting the graph using the `subgraph` operator. The vertex predicate is pushed to the edge partitions, where it can be used to filter the triplets using the CSR index on the source vertex id (Section 4.4.1). We measure the selectivity of the vertex predicate and switch from sequential scan to clustered index scan when the selectivity is less than 0.8.

Figure 4.5 illustrates the benefit of index scans in PageRank and connected components. As with incremental view maintenance, index scans lead to a smaller improvement in runtime for PageRank and a substantial improvement in runtime for connected components. Interestingly, in the initial iterations of connected components, when the majority of the vertices are active, a sequential scan is slightly faster as it does not require the additional index lookup. It is for this reason that we dynamically switch between full and indexed scans based on the fraction of active vertices.

Automatic Join Elimination

In some cases, operations on the triplets view may access only one of the vertex properties or none at all. For example, when `mrTriplets` is used to count the degree of each vertex, the map UDF does not access any vertex properties. Similarly, when computing messages in PageRank only the source vertex properties are used.

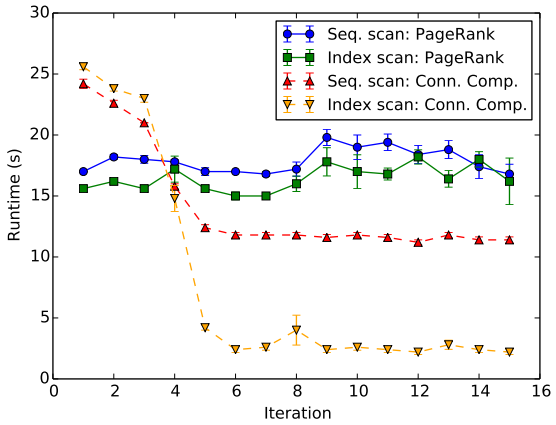


Figure 4.5: **Sequential scan vs index scan**: Connected components on the Twitter graph benefits greatly from switching to index scan after the 4th iteration, while PageRank benefits only slightly because the set of active vertices is large even at the 15th iteration.

GraphX uses a JVM bytecode analyzer to inspect user-defined functions at runtime and determine whether the source or target vertex properties are referenced. If only one property is referenced, and if the triplets view has not already been materialized, GraphX automatically rewrites the query plan for generating the triplets view from a three-way join to a two-way join. If none of the vertex properties are referenced, GraphX eliminates the join entirely. This modification is possible because the triplets view follows the lazy semantics of RDDs in Spark. If the user never accesses the triplets view, it is never materialized. A call to `mrTriplets` is therefore able to rewrite the join needed to generate the relevant part of the triplets view.

Figure 4.6 demonstrates the impact of this physical execution plan rewrite on communication and runtime for PageRank on the Twitter follower graph. We see that join elimination cuts the amount of data transferred in half, leading to a significant reduction in overall runtime. Note that on the first iteration there is no reduction in communication. This is due to compression algorithms that take advantage of all messages having exactly the same initial value. However, compression and decompression still consume CPU time so we still observe nearly a factor of two reduction in overall runtime.

4.4.4 Additional Optimizations

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

Memory-based Shuffle: Spark's default shuffle implementation materializes the

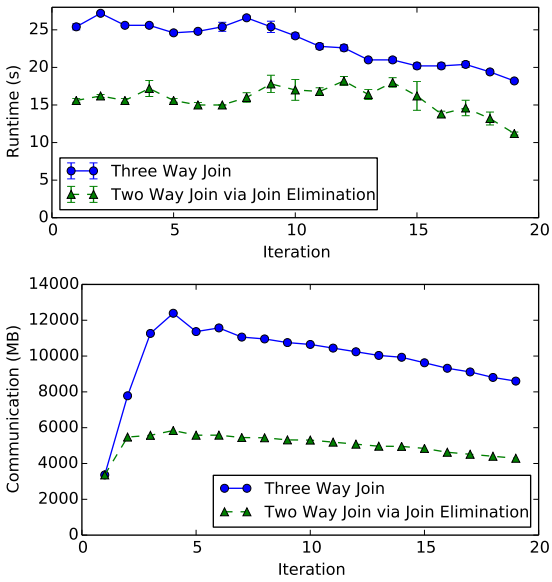


Figure 4.6: **Impact of automatic join elimination on communication and runtime**: We ran PageRank for 20 iterations on the Twitter dataset with and without join elimination and found that join elimination reduces the amount of communication by almost half and substantially decreases the total execution time.

temporary data to disk. We modified the shuffle phase to materialize map outputs in memory and remove this temporary data using a timeout.

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

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

4.5 Performance Evaluation

In this section we demonstrate that, for iterative graph algorithms, GraphX is over an order of magnitude faster than directly using the general-purpose dataflow

operators described in Section 4.3.2 and is comparable to or faster than specialized graph processing systems.

We evaluate the performance of GraphX on several graph-analytics tasks, comparing it with the following:

1. **Apache Spark 0.9.1**: the base distributed dataflow system for GraphX. We compare against Spark to demonstrate the performance gains relative to the baseline distributed dataflow framework.
2. **Apache Giraph 1.1**: an open source graph computation system based on the Pregel abstraction.
3. **GraphLab 2.2 (PowerGraph)**: the open-source graph computation system based on the GAS decomposition of vertex programs. Because GraphLab is implemented in C++ and all other systems run on the JVM, given identical optimizations, we would expect GraphLab to have a slight performance advantage.

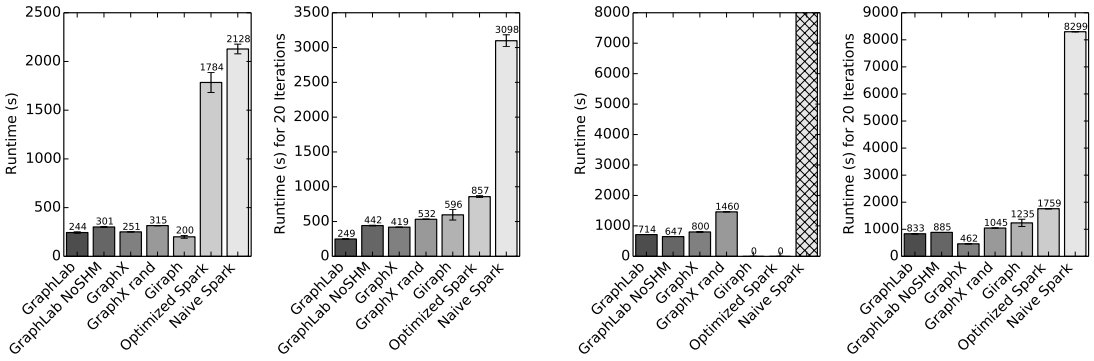
We also compare against GraphLab without shared-memory parallelism (denoted **GraphLab NoSHM**). GraphLab communicates between workers on the same machine using shared data structures. In contrast, Giraph, Spark, and GraphX adopt a shared-nothing worker model incurring extra serialization overhead between workers. To isolate this overhead, we disabled shared-memory by forcing GraphLab workers to run in separate processes.

It is worth noting that the shared data structures in GraphLab increase the complexity of the system. Indeed, we encountered and fixed a critical bug in one of the GraphLab shared data structures. The resulting patch introduced an additional lock which led to a small increase in thread contention. As a consequence, in some cases (*e.g.,* Figure 4.7c) disabling shared memory contributed to a small improvement in performance.

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

4.5.1 System Comparison

Cross-system benchmarks are often unfair due to the difficulty in tuning each system equitably. We have endeavored to minimize this effect by working closely with experts in each of the systems to achieve optimal configurations. We emphasize that we are *not* claiming GraphX is fundamentally faster than GraphLab or Giraph; these systems could in theory implement the same optimizations as GraphX. Instead, we aim to show that it is possible to achieve comparable performance to specialized graph processing systems using a general dataflow engine while gaining common dataflow features such as fault tolerance.



(a) Conn. Comp. Twitter (b) PageRank Twitter (c) Conn. Comp. uk-2007-05* (d) PageRank uk-2007-05*

Figure 4.7: **System Performance Comparison.** (*c*) Spark did not finish within 8000 seconds, Giraph and Spark + Part. ran out of memory.

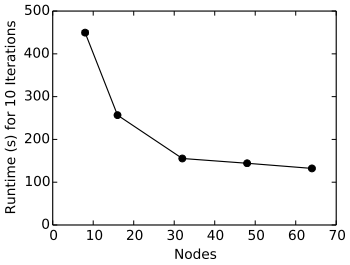


Figure 4.8: **Strong scaling for PageRank on Twitter (10 Iterations)**

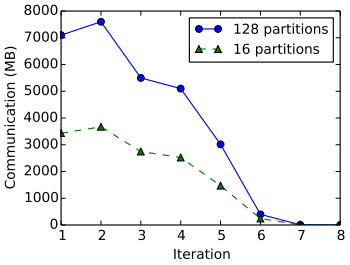


Figure 4.9: **Effect of partitioning on communication**

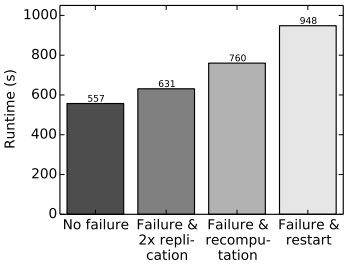


Figure 4.10: **Fault tolerance for PageRank on uk-2007-05**

While we have implemented a wide range of graph algorithms on top of GraphX, we restrict our performance evaluation to PageRank and connected components. These two representative graph algorithms are implemented in most graph processing systems, have well-understood behavior, and are simple enough to serve as an effective measure of the system's performance. To ensure a fair comparison, our PageRank implementation is based on Listing 4.1; it does not exploit delta messages and therefore benefits less from indexed scans and incremental view maintenance. Conversely, the connected components implementation only sends messages when a vertex must change component membership and therefore does benefit from incremental view maintenance.

For each system, we ran both algorithms on the twitter-2010 and uk-2007-05 graphs (Table 4.1). For Giraph and GraphLab we used the included implementations of these algorithms. For Spark we implemented the algorithms both using idiomatic dataflow operators (**Naive Spark**, as described in Section 4.3.2) and using an optimized implementation (**Optimized Spark**) that eliminates movement

Dataset	Edges	Vertices
twitter-2010 [32, 31]	1,468,365,182	41,652,230
uk-2007-05 [32, 31]	3,738,733,648	105,896,555

Table 4.1: **Graph Datasets.** Both graphs have highly skewed power-law degree distributions.

of edge data by pre-partitioning the edges to match the partitioning adopted by GraphX.

Both GraphLab and Giraph partition the graph according to specialized partitioning algorithms. While GraphX supports arbitrary user defined graph partitioners including those used by GraphLab and Giraph, the default partitioning strategy is to construct a vertex-cut that matches the input edge data layout thereby minimizing edge data movement when constructing the graph. However, as point of comparison we also tested GraphX using a randomized vertex-cut (**GraphX Rand**). We found (see Figure 4.7) that for the specific datasets used in our experiments the input partitioning, which was determined by a specialized graph compression format [31], actually resulted in a more communication-efficient vertex-cut partitioning.

Figures 4.7a and 4.7c show the total runtimes for connected components algorithm. We have excluded *Giraph* and *Optimized Spark* from Figure 4.7c because they were unable to scale to the larger web-graph in the allotted memory of the cluster. While the basic Spark implementation did not crash, it was forced to recompute blocks from disk and exceeded 8000 seconds per iteration. We attribute the increased memory overhead to the use of edge-cut partitioning and the need to store bi-directed edges and messages for the connected components algorithm.

Figures 4.7b and 4.7d show the total runtimes for PageRank for 20 iterations on each system. In Figure 4.7b, GraphLab outperforms GraphX largely due to shared-memory parallelism; GraphLab without shared memory parallelism is much closer in performance to GraphX. In 4.7d, GraphX outperforms GraphLab because the input partitioning of uk-2007-05 is highly efficient, resulting in a 5.8x reduction in communication per iteration.

4.5.2 GraphX Performance

Scaling: In Figure 4.8 we evaluate the strong scaling performance of GraphX running PageRank on the Twitter follower graph. As we move from 8 to 32 machines (a factor of 4) we see a 3x speedup. However as we move to 64 machines (a factor of 8) we only see a 3.5x speedup. While this is hardly linear scaling, it is actually slightly better than the 3.2x speedup reported by GraphLab [58]. The poor scaling performance of PageRank has been attributed by [58] to high communication overhead relative to computation for the PageRank algorithm.

It may seem surprising that GraphX scales slightly better than GraphLab given that Spark does not exploit shared memory parallelism and therefore forces the

graph to be partitioned across processors rather than machines. However, Figure 4.9 shows the communication of GraphX as a function of the number of partitions. Going from 16 to 128 partitions (a factor of 8) yields only an approximately 2-fold increase in communication. Returning to the analysis of vertex-cut partitioning conducted by [58], we find that the vertex-cut partitioning adopted by GraphX mitigates the 8-fold increase in communication.

Fault tolerance: Existing graph systems only support checkpoint-based fault tolerance, which most users leave disabled due to the performance overhead. GraphX is built on Spark, which provides lineage-based fault tolerance with negligible overhead as well as optional dataset replication. We benchmarked these fault tolerance options for PageRank on uk-2007-05 by killing a worker in iteration 11 of 20, allowing Spark to recover by using the remaining copies of the lost partitions or recomputing them, and measuring how long the job took in total. For comparison, we also measured the end-to-end time for running until failure and then restarting from scratch on the remaining nodes using a driver script, as would be necessary in existing graph systems. Figure 10 shows that in case of failure, both replication and recomputation are faster than restarting the job from scratch, and moreover they are performed transparently by the dataflow engine.

4.6 Related Work

In Section 4.2 we described the general characteristics shared across many of the earlier graph processing systems. However, there are some exceptions to many of these characteristics that are worth noting.

While most of the work on large-scale distributed graph processing has focused on static graphs, several systems have focused on various forms of stream processing. One of the earlier examples is Kineograph [39], a distributed graph processing system that constructs incremental snapshots of the graph for offline static graph analysis. In the multicore setting, GraphChi [78] and later X-Stream [110] introduced support for the addition of edges between existing vertices and between computation stages. Although conceptually GraphX could support the incremental introduction of edges (and potentially vertices), the existing data-structures would require additional optimization. Instead, GraphX focuses on efficiently supporting the *removal* of edges and vertices: essential functionality for offline sub-graph analysis.

Most of the optimizations and programming models of earlier graph processing systems focus on a single graph setting. While some of these systems [81, 58, 110] are capable of operating on multiple graphs *independently*, they do not expose an API or present optimizations for operations spanning graphs (or tables). One notable exception is CombBLAS [34] which treats graphs (and data more generally) as matrices and supports generalized binary algebraic operators. In contrast GraphX

preserves the native semantics of graphs and tables and provides a simple API to combine data across these representations.

The triplets view in GraphX is related to the classic *Resource Description Framework* [86] (RDF) data model which encodes graph structured data as *subject-predicate-object* triplets (*e.g., NYC-isA-city*). Numerous systems [7, 33, 96] have been proposed for storing and executing SPARQL [105] subgraph queries against RDF triplets. Like GraphX, these systems rely heavily on indexing and clustering for performance. Unlike GraphX, these systems are not distributed or do not address iterative graph algorithms. Nonetheless, we believe that the optimizations techniques developed for GraphX may benefit the design of distributed graph query processing.

There have been several recent efforts at exploring graph algorithms within dataflow systems. Najork et al. [94], compares implementations of a range of graph algorithms on the DryadLINQ [68] and SQL Server dataflow systems. However, the resulting implementations are fairly complex and specialized, and little is discussed about graph-specific optimizations. Both Ewen et al. [50] and Murray et al. [92] proposed dataflow systems geared towards incremental iterative computation and demonstrated performance gains for specialized implementations of graph algorithms. While this work highlights the importance of incremental updates in graph computation, neither proposed a general method to express graph algorithms or graph specific optimizations beyond incremental dataflows. Nonetheless, we believe that the GraphX system could be ported to run on-top of these dataflow frameworks and would potentially benefit from advances like timely dataflows [92].

At the time of publication, the Microsoft Naiad team had announced initial work on a system called GraphLINQ [91], a graph processing framework on-top of Naiad which shares many similarities to GraphX. Like GraphX, GraphLINQ aims to provides rich graph functionality within a general-purpose dataflow framework. In particular GraphLINQ presents a `GraphReduce` operator that is semantically similar to the `mrTriplets` operator in GraphX except that it operates on streams of vertices and edges. The emphasis on stream processing exposes opportunities for classic optimizations in the stream processing literature as well as recent developments like the Naiad timely dataflows [92]. We believe this further supports the advantages of embedding graph processing within more general-purpose data processing systems.

Others have explored join optimizations in distributed dataflow frameworks. Blanas et al. [30] show that broadcast joins and semi-joins compare favorably with the standard MapReduce style shuffle joins when joining a large table (*e.g.,* edges) with a smaller table (*e.g.,* vertices). Closely related is the work by Afrati et al. [11] which explores optimizations for multi-way joins in a MapReduce framework. They consider joining a large relation with multiple smaller relations and provide a partitioning and replication strategy similar to classic 2D partitioning [35]. However, in contrast to our work, they do not construct a routing table forcing the system to broadcast the smaller relations (*e.g.,* the vertices) to all partitions of the larger relation (*e.g.,* the edges) that *could* have matching tuples. Furthermore, they force a

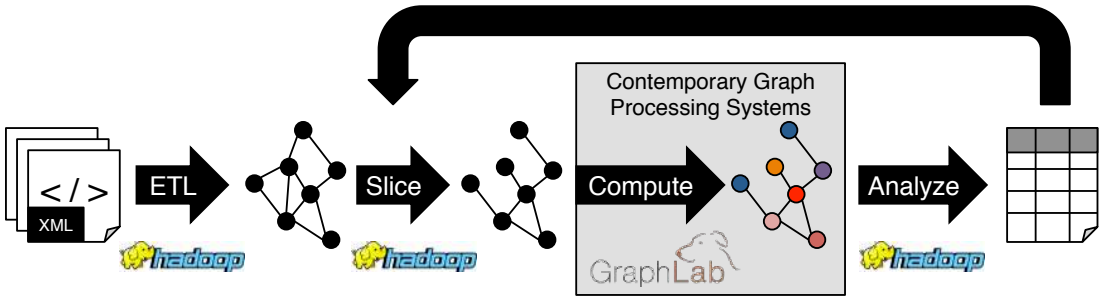


Figure 4.11: **Graph Analytics Pipeline:** requires multiple collection and graph views of the same data.

particular hash partitioning on the larger relation precluding the opportunity for user defined graph partitioning algorithms (*e.g.,* [74, 122, 117]).

4.7 Discussion

The work on GraphX addressed several key themes in data management systems and system design:

Physical Data Independence: GraphX allows the same physical data to be viewed as collections and as graphs without data movement or duplication. As a consequence the user is free to adopt the best view for the immediate task. We demonstrated that operations on collections and graphs can be efficiently implemented using the same physical representation and underlying operators. Our experiments show that this common substrate can match the performance of specialized graph systems.

Graph Computation as Joins and Group-By: The design of the GraphX system reveals the strong connection between distributed graph computation and distributed join optimizations. When viewed through the lens of dataflow operators, graph computation reduces to join and group-by operators. These two operators correspond to the Scatter and Gather stages of the GAS abstraction. Likewise, the optimizations developed for graph processing systems reduce to indexing, distributed join site selection, multicast joins, partial materialization, and incremental view maintenance.

The Narrow Waist: In designing the GraphX abstraction, we sought to develop a thin extension on top of dataflow operators with the goal of identifying the essential data model and core operations needed to support graph computation. We aimed for a portable framework that could be embedded in a range of dataflow frameworks. We believe that the GraphX design can be adopted by other dataflow systems, including MPP databases, to efficiently support a wide range of graph computations.

Analytics Pipelines: GraphX provides the ability to stay within a single frame-

work throughout the analytics process, eliminating the need to learn and support multiple systems (*e.g.,* Figure 4.11) or write data interchange formats and plumbing to move between systems. As a consequence, it is substantially easier to iteratively slice, transform, and compute on large graphs as well as to share data-structures across stages of the pipeline. The gains in performance and scalability for graph computation translate to a tighter analytics feedback loop and therefore a more efficient work flow.

Adoption: GraphX was publicly released as part of the 0.9.0 release of the Apache Spark open-source project. It has since generated substantial interest in the community and has been used in production at various places [67]. Despite its nascent state, there has been considerable open-source contribution to GraphX with contributors providing some of the core graph functionality. We attribute this to its wide applicability and the simple abstraction built on top of an existing, popular dataflow framework.

4.8 Conclusion

In this work we introduced GraphX, an efficient graph processing system that enables distributed dataflow frameworks such as Spark to naturally express and efficiently execute iterative graph algorithms. We identified a simple pattern of *join–map–group-by* dataflow operators that forms the basis of graph-parallel computation. Inspired by this observation, we proposed the GraphX abstraction, which represents graphs as horizontally-partitioned collections and graph computation as dataflow operators on those collections. Not only does GraphX support existing graph-parallel abstractions and a wide range of iterative graph algorithms, it enables the composition of graphs and collections, freeing the user to adopt the most natural view without concern for data movement or duplication.

Guided by the connection between graph computation and dataflow operators, we recast recent advances in graph processing systems as range of classic optimizations in database systems. We recast vertex-cut graph partitioning as horizontally-partitioned vertex and edge collections, active vertex tracking as incremental view maintenance, and vertex mirroring as multicast joins with routing tables. As a result, for graph algorithms, GraphX is over an order of magnitude faster than the base dataflow system and is comparable to or faster than specialized graph processing systems. In addition, GraphX benefits from features provided by recent dataflow systems such as low-cost fault tolerance and transparent recovery.

Chapter 5

Conclusion

This chapter concludes the dissertation. We summarize the contributions and broader industry impact from each of the three systems. Finally, we discuss future work.

5.1 Innovation Highlights and Broader Impact

All three systems, Spark SQL, Structured Streaming, and GraphX, developed in this dissertation, share a common theme: modern analytics are becoming more diverse. The types of analytics are becoming more diverse: complex analytics such as machine learning and graph processing are becoming increasingly common in addition to relational query processing. The programming languages are becoming more diverse: data engineers use Scala and Java, data scientists use Python and R, while business analysts still predominantly use SQL. Finally, the frequency of analytics are becoming more diverse as well: many organizations that started with batch analytics that happen once a day are now employing real-time streaming data pipelines to increase the velocity of data-driven decisions.

These three systems have been widely adopted in industry. Combined together, the three systems laid the foundation for Spark 2.0.

5.1.1 Spark SQL

Spark SQL integrates relational processing with Spark's functional programming API for complex analytics. Built on our experience with Shark, Spark SQL lets Spark programmers leverage the benefits of relational processing (*e.g.,* declarative queries and optimized storage), and lets SQL users call complex analytics libraries in Spark (*e.g.,* machine learning). Compared to previous systems, Spark SQL makes two main additions. First, it offers much tighter integration between relational and procedural processing, through a declarative DataFrame API that integrates with procedural Spark code. Second, it includes a highly extensible optimizer, Catalyst, built using

features of the Scala programming language, that makes it easy to add composable rules, control code generation, and define extension points. Using Catalyst, we have built a variety of features (*e.g.*, schema inference for JSON, machine learning data types, and query federation to external databases) tailored for the complex needs of modern data analysis. We see Spark SQL as an evolution of both SQL-on-Spark and of Spark itself, offering richer APIs and optimizations while keeping the benefits of the Spark programming model. More fundamentally, Spark SQL and its predecessor Shark show that dataflow execution models can be applied effectively to SQL, and offer a promising way to combine relational and complex analytics.

Since its initial experimental open source release in 2014, Spark SQL has become the most widely used and developed module in Spark. The DataFrame API became the de facto programming API in Spark's 2.0 release. While the initial RDD API remains as an important lower level API, most user applications have migrated over from the RDD API to the DataFrame API. Given that Spark is one of the most popular open source projects in Big Data, it is very likely that the DataFrame API is the most widely used API in big data.

5.1.2 Structured Streaming

With the ubiquity of streaming data, organizations need stream processing systems that are scalable, easy to use, and easy to integrate into business applications. Structured Streaming builds on Spark SQL, and exposes a high-level, declarative streaming API. Structured Streaming differs from other recent streaming APIs, such as Google Dataflow, in two main ways. First, it is a purely *declarative* API based on automatically incrementalizing a static relational query (expressed using SQL or DataFrames), as opposed to APIs that ask the user to design a DAG of physical operators. This makes it accessible to non-experts. Second, Structured Streaming is designed to support *end-to-end* real-time applications that combine streaming with batch and interactive analysis. We found that many real-world applications required this type of integration, and this integration was often the most challenging part of building them. Structured Streaming achieves high performance using Spark SQL's code generation engine and outperforms Apache Flink by up to 4× and Kafka Streams by 90×. It also offers powerful operational features such as rollbacks, code updates, and mixed streaming/batch execution.

Structured Streaming was initially released in 2016. We have observed several hundred large-scale production use cases, the largest of which processes over 1 PB of data per month.

5.1.3 GraphX

GraphX is an embedded graph processing system built on top of Apache Spark. GraphX presents a familiar composable graph abstraction that is sufficient to express

existing graph APIs, yet can be implemented using only a few basic dataflow operators (e.g., join, map, group-by).

In pursuit of graph processing performance, the systems community has largely abandoned general-purpose distributed dataflow frameworks in favor of specialized graph processing systems that provide tailored programming abstractions and accelerate the execution of iterative graph algorithms. In this paper we argue that many of the advantages of specialized graph processing systems can be recovered in a modern general-purpose distributed dataflow system.

To demonstrate this, we implemented GraphX on top of Spark, a dataflow framework. To achieve performance parity with specialized graph systems, GraphX recasts graph-specific optimizations as distributed join optimizations and materialized view maintenance. By leveraging advances in distributed dataflow frameworks, GraphX brings low-cost fault tolerance to graph processing. We evaluate GraphX on real workloads and demonstrate that GraphX achieves an order of magnitude performance gain over the base dataflow framework and matches the performance of specialized graph processing systems while enabling a wider range of computation.

GraphX was merged into Spark in version 1.2. In Spark 2.0, it became the main graph processing library, in lieu of Bagel.

5.2 Future Work

Our work on these systems points to a larger research agenda in the unification of specialized data processing systems. Recent advances in specialized systems for topic modeling, graph processing, stream processing, and deep learning have revealed a range of new system optimizations and design trade-offs. However, the full potential of these systems is often realized in their integration (*e.g.,* applying deep learning to text and images in a social network). By casting these systems within a common paradigm we may reveal common patterns and enable new analytic capabilities. In this context, the main areas of future work are:

AI, Deep Learning, and Machine Learning on Dataflows: One cannot avoid drawing attention to AI, deep learning, and machine learning when discussing data analytics in 2018. Many systems, both hardware [72] and software [8, 102], have been built specifically for developing and executing deep learning and machine learning algorithms. These systems achieve high performance for numeric computation through highly optimized kernels that employ data-parallel CPU and GPU instructions.

There are two general approaches in using dataflow systems for machine learning. The first approach uses the dataflow systems across the entire pipeline, from data loading, to feature engineering, to training. In this approach, the training algorithm is implemented entirely in these dataflow systems, using dataflow or relational operators. The second approach is to "integrate". In this approach, users

leverage dataflow systems for data loading, feature engineering, and parallelization, but run the specialized software (*e.g.,* PyTorch) for training. Compared with the first approach, the second approach typically achieves higher performance for the training part, at the cost of increased operational complexity and lower performance in data exchange.

One fruitful area of research is to explore whether a single dataflow system can efficiently and effectively support machine learning pipelines end-to-end, from data loading, to feature engineering, to training. This will require the development of new code generation engines that can combine relational processing with highly efficient numeric computations. KeystoneML [116] is an early attempt at this.

GraphFrames: The graph abstraction in GraphX is analogous to the RDD abstraction for distributed collection of data. It is a lower level, expressive API, but lacks optimization opportunities due to the extensive use of user-defined functions as well as coupling of physical plans and logical plans. A possible extension is to combine the ideas in Spark SQL and GraphX to create GraphFrames, a DataFrame-based approach for programming graph applications. This new approach can enable graph specific optimizations by extending the Catalyst query optimizer.

Streaming, Incremental Graph Computation: Similar to streaming data that are constantly changing, real-life graph structures are rarely static. Members on social networks make new connections all the time, while an interest graph would be updated each time a user "likes" an item. All operations in GraphX are batch-oriented, and any update on a graph would require recomputing the entire data pipeline.

Incremental graph computation will be a fruitful research area, by minimizing the computation cost of small graph updates. This will require both API changes to restrict operations to be algebraic, as well as system changes that can automatically incrementalize graph computation plans.

Unifying multi-core with distributed dataflows: In the course of this dissertation, the increase of CPU clock speed has largely plateaued. Instead, CPUs are increasing in core-count. It is not uncommon to see machines with 128 cores. With such high core counts, future multi-core systems resemble distributed systems more than they do traditional single-node systems. We believe there is a tremendous opportunity to design simple software architectures, based on dataflows, that can scale well from single-node multi-core systems, for small to medium amounts of data, to clusters of thousands of machines, for large amounts of data.

We hope the lessons learned in this dissertation can help facilitate further research in the above areas, and create innovative solutions that shape new system directions.

Bibliography

[1] http://aws.amazon.com/elasticmapreduce/.

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

[3] Apache Impala project. https://github.com/cloudera/impala.

[4] Apache Spark project contributor graph. https://github.com/apache/spark/graphs/contributors.

[5] Deja VVVu: Others Claiming Gartners Construct for Big Data. Gartner Blog Network, 2012.

[6] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Mitch Cherniack, Jeong hyon Hwang, Wolfgang Lindner, Anurag S. Maskey, Er Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. The design of the borealis stream processing engine. In *In CIDR*, pages 277–289, 2005.

[7] Daniel J. Abadi, Adam Marcus, Samuel R. Madden, and Kate Hollenbach. SW-Store: A vertically partitioned DBMS for semantic web data management. *PVLDB*, 18(2):385–406, 2009.

[8] Martin Abadi, Paul Barham, et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[9] Azza Abouzeid et al. Hadoopdb: an architectural hybrid of mapreduce and dbms technologies for analytical workloads. *VLDB*, 2009.

[10] Azza Abouzied, Daniel J. Abadi, and Avi Silberschatz. Invisible loading: Access-driven data transfer from raw files into database systems. In *EDBT*, 2013.

[11] Foto N. Afrati and Jeffrey D. Ullman. Optimizing joins in a map-reduce environment. In *EDBT*, pages 99–110, 2010.

[12] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. The dataflow model: A practical

approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proc. VLDB Endow.*, 8(12):1792–1803, August 2015.

[13] Alexander Alexandrov et al. The Stratosphere platform for big data analytics. *The VLDB Journal*, 23(6):939–964, December 2014.

[14] Intel Altera. Financial/hpc – financial offload. `https://www.altera.com/solutions/industry/computer-and-storage/applications/computer/financial-offload.html`, 2017.

[15] Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and William R. Marczak. Consistency analysis in bloom: A calm and collected approach. In *In Proceedings 5th Biennial Conference on Innovative Data Systems Research*, pages 249–260, 2011.

[16] Amazon. Amazon Kinesis. `https://aws.amazon.com/kinesis/`, 2017.

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

[18] Apache Avro project. http://avro.apache.org.

[19] Apache Parquet project. http://parquet.incubator.apache.org.

[20] Apache Spark project. http://spark.apache.org.

[21] Michael Armbrust. SPARK-20928: Continuous processing mode for structured streaming. https://issues.apache.org/jira/browse/SPARK-20928, 2017.

[22] Michael Armbrust, Bill Chambers, and Matei Zaharia. Databricks delta: A unified data management system for real-time big data. `https://databricks.com/blog/2017/10/25/databricks-delta-a-unified-management-system.html`, 2017.

[23] Michael Armbrust, Nick Lanham, Stephen Tu, Armando Fox, Michael J Franklin, and David A Patterson. The case for PIQL: a performance insightful query language. In *SOCC*, 2010.

[24] Jeff Barr. New – per-second billing for EC2 instances and EBS volumes. `https://aws.amazon.com/blogs/aws/new-per-second-billing-for-ec2-instances-and-ebs-volumes/`, 2017.

[25] Apache Beam. Apache Beam programming guide. `https://beam.apache.org/documentation/programming-guide/`, 2017.

[26] Alexander Behm et al. Asterix: towards a scalable, semistructured data platform for evolving-world models. *Distributed and Parallel Databases*, 29(3):185–216, 2011.

[27] Greet Jan Bex, Frank Neven, and Stijn Vansummeren. Inferring XML schema definitions from XML data. In *VLDB*, 2007.

[28] BigDF project. https://github.com/AyasdiOpenSource/bigdf.

[29] Jose A. Blakeley, Per-Ake Larson, and Frank Wm Tompa. Efficiently updating materialized views. *SIGMOD Rec.*, 15(2):61–71, June 1986.

[30] Spyros Blanas, Jignesh M. Patel, Vuk Ercegovac, Jun Rao, Eugene J. Shekita, and Yuanyuan Tian. A comparison of join algorithms for log processing in MapReduce. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 975–986, New York, NY, USA, 2010. ACM.

[31] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *WWW*, pages 587–596, 2011.

[32] Paolo Boldi and Sebastiano Vigna. The WebGraph framework I: Compression techniques. In *WWW'04*.

[33] Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen. Sesame: A generic architecture for storing and querying rdf and rdf schema. In *Proceedings of the First International Semantic Web Conference on The Semantic Web*, ISWC '02, pages 54–68, 2002.

[34] Aydin Buluç and John R. Gilbert. The combinatorial BLAS: design, implementation, and applications. *IJHPCA*, 25(4):496–509, 2011.

[35] Ümit V. Çatalyürek, Cevdet Aykanat, and Bora Uçar. On two-dimensional sparse matrix partitioning: Models, methods, and a recipe. *SIAM J. Sci. Comput.*, 32(2):656–683, 2010.

[36] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, Robert DeLine, Danyel Fisher, John C. Platt, James F. Terwilliger, and John Wernsing. Trill: A high-performance incremental query processor for diverse analytics. *Proc. VLDB Endow.*, 8(4):401–412, December 2014.

[37] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R. Madden, Fred Reiss, and Mehul A. Shah. Telegraphcq: Continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 668–668, New York, NY, USA, 2003. ACM.

[38] B. Chattopadhyay, , et al. Tenzing a sql implementation on the mapreduce framework. *PVLDB*, 4(12):1318–1327, 2011.

[39] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. Kineograph: taking the pulse of a fast-changing and connected world. In *EuroSys*, pages 85–98, 2012.

[40] Sanket Chintapalli, Derek Dagit, Bobby Evans, Reza Farivar, Tom Graves, Mark Holderbaugh, Zhuo Liu, Kyle Nusbaum, Kishorkumar Patil, Boyang Peng, and Paul Poulosky. Benchmarking streaming computation engines at Yahoo! `https://yahooeng.tumblr.com/post/135321837876/benchmarking-streaming-computation-engines-at`, 2015.

[41] C. Chu et al. Map-reduce for machine learning on multicore. *Advances in neural information processing systems*, 19:281, 2007.

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

[43] Confluent. Ksql: Streaming sql for kafka. `https://www.confluent.io/product/ksql/`, 2017.

[44] Databricks. Apache Spark Survey 2016 Results Now Available. `https://databricks.com/blog/2016/09/27/spark-survey-2016-released.html`, 2016.

[45] Databricks. Databricks unified analytics platform. `https://databricks.com/product/unified-analytics-platform`, 2017.

[46] DDF project. http://ddf.io.

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

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

[49] Burak Emir, Martin Odersky, and John Williams. Matching objects with patterns. In *ECOOP 2007 – Object-Oriented Programming, volume 4609 of LNCS*, pages 273–298. Springer, 2007.

[50] Stephan Ewen, Kostas Tzoumas, Moritz Kaufmann, and Volker Markl. Spinning fast iterative data flows. *Proc. VLDB*, 5(11):1268–1279, July 2012.

[51] Uriel Feige, MohammadTaghi Hajiaghayi, and James R. Lee. Improved approximation algorithms for minimum-weight vertex separators. In *Proceedings of the Thirty-seventh Annual ACM Symposium on Theory of Computing*, STOC '05, pages 563–572, New York, NY, USA, 2005. ACM.

[52] Apache Flink. Apache flink. `http://flink.apache.org`, 2017.

[53] Apache Flink. Flink datastream api programming guide. https://ci.apache.org/projects/flink/flink-docs-release-1.3/dev/datastream_api.html, 2017.

[54] Apache Flink. Flink table & sql api beta. https://ci.apache.org/projects/flink/flink-docs-release-1.3/dev/table/index.html, 2017.

[55] Apache Flink. Working with state. https://ci.apache.org/projects/flink/flink-docs-release-1.3/dev/stream/state.html, 2017.

[56] 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.

[57] Lukasz Golab and M. Tamer Özsu. Issues in data stream management. *SIGMOD Rec.*, 32(2):5–14, June 2003.

[58] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. OSDI'12, pages 17–30, Berkeley, CA, USA, 2012. USENIX Association.

[59] G. Graefe and D.J. DeWitt. The EXODUS optimizer generator. In *SIGMOD*, 1987.

[60] Goetz Graefe. The Cascades framework for query optimization. *IEEE Data Engineering Bulletin*, 18(3), 1995.

[61] Jamie Grier. Extending the Yahoo! streaming benchmark. https://data-artisans.com/blog/extending-the-yahoo-streaming-benchmark, 2016.

[62] B. Guffler et al. Handling data skew in mapreduce. In *CLOSER'11*.

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

[64] Jan Hegewald, Felix Naumann, and Melanie Weis. XStruct: efficient schema extraction from multiple and large XML documents. In *ICDE Workshops*, 2006.

[65] 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.

[66] Hive data definition language. https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DDL.

[67] Andy Huang and Wei Wu. Mining ecommerce graph data with spark at alibaba taobao. http://databricks.com/blog/2014/08/14/mining-graph-data-with-spark-at-alibaba-taobao.html, 2014.

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

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

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

[71] Jackson JSON processor. http://jackson.codehaus.org.

[72] Norman P. Jouppi, Cliff Young, et al. In-datacenter performance analysis of a tensor processing unit. *CoRR*, abs/1704.04760, 2017.

[73] Apache Kafka. Kafka. `http://kafka.apache.org`, 2017.

[74] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *J. Parallel Distrib. Comput.*, 48(1):96–129, 1998.

[75] Yannis Klonatos, Christoph Koch, Tiark Rompf, and Hassan Chafi. Building efficient query engines in a high-level language. *PVLDB*, 7(10):853–864, 2014.

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

[77] YongChul Kwon et al. Skewtune: mitigating skew in mapreduce applications. In *SIGMOD '12*, 2012.

[78] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: Large-scale graph computation on just a PC. In *OSDI*, 2012.

[79] J. Leskovec, K. J. Lang, A. Dasgupta, , and M. W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6(1):29–123, 2008.

[80] Dionysios Logothetis, Christopher Olston, Benjamin Reed, Kevin C. Webb, and Ken Yocum. Stateful bulk processing for incremental analytics. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 51–62, New York, NY, USA, 2010. ACM.

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

[82] 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.

[83] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and J. M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *PVLDB*, 2012.

[84] Lothar F. Mackert and Guy M. Lohman. R* optimizer validation and performance evaluation for distributed queries. In *VLDB'86*, pages 149–159, 1986.

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

[86] Frank Manola and Eric Miller. RDF primer. *W3C Recommendation*, 10:1–107, 2004.

[87] Frank McSherry, Derek Murray, Rebecca Isaacs, and Michael Isard. Differential dataflow. In *Proceedings of CIDR 2013*, January 2013.

[88] 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.

[89] Xiangrui Meng, Joseph Bradley, Evan Sparks, and Shivaram Venkataraman. ML pipelines: a new high-level API for MLlib. https://databricks.com/blog/2015/01/07/ml-pipelines-a-new-high-level-api-for-mllib.html.

[90] Jayanta Mondal and Amol Deshpande. Managing large dynamic graphs efficiently. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 145–156, New York, NY, USA, 2012. ACM.

[91] Derek Murray. Building new frameworks on Naiad, April 2014.

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

[93] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A timely dataflow system. pages 439–455, 2013.

[94] Marc Najork, Dennis Fetterly, Alan Halverson, Krishnaram Kenthapadi, and Sreenivas Gollapudi. Of hammers and nails: An empirical comparison of three paradigms for processing large graphs. In *Proceedings of the Fifth ACM International Conference on Web Search and Data Mining*, WSDM '12, pages 103–112. ACM, 2012.

[95] S. Nestorov, S. Abiteboul, and R. Motwani. Extracting schema from semistructured data. In *ICDM*, 1998.

[96] Thomas Neumann and Gerhard Weikum. RDF-3X: A RISC-style engine for RDF. *VLDB'08*.

[97] Frank Austin Nothaft, Matt Massie, Timothy Danford, Zhao Zhang, Uri Laserson, Carl Yeksigian, Jey Kottalam, Arun Ahuja, Jeff Hammerbacher, Michael Linderman, Michael J. Franklin, Anthony D. Joseph, and David A. Patterson. Rethinking data-intensive science using scalable analytics systems. In *SIGMOD*, 2015.

[98] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD '08*, pages 1099–1110.

[99] Kay Ousterhout et al. The case for tiny tasks in compute clusters. In *HotOS'13*.

[100] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, 1999.

[101] `pandas` Python data analysis library. http://pandas.pydata.org.

[102] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[103] A. Pavlo et al. A comparison of approaches to large-scale data analysis. In *SIGMOD*, 2009.

[104] 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.

[105] Eric Prud'hommeaux and Andy Seaborne. SPARQL query language for RDF. Latest version available as `http://www.w3.org/TR/rdf-sparql-query/`, January 2008.

[106] Josep M. Pujol, Vijay Erramilli, Georgos Siganos, Xiaoyuan Yang, Nikos Laoutaris, Parminder Chhabra, and Pablo Rodriguez. The little engine(s) that could: scaling online social networks. In *SIGCOMM*, pages 375–386, 2010.

[107] X. Qian and Gio Wiederhold. Incremental recomputation of active relational expressions. *IEEE Trans. on Knowl. and Data Eng.*, 3(3):337–341, September 1991.

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

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

[110] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. SOSP '13, pages 472–488. ACM, 2013.

[111] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edition, 2003.

[112] scikit-learn: machine learning in Python. http://scikit-learn.org.

[113] Denys Shabalin, Eugene Burmako, and Martin Odersky. Quasiquotes for Scala, a technical report. Technical Report 185242, École Polytechnique Fédérale de Lausanne, 2013.

[114] Apache Spark. Spark documentation. `http://spark.apache.org/docs/latest`, 2017.

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

[116] Evan R. Sparks, Shivaram Venkataraman, Tomer Kaftan, Michael J. Franklin, and Benjamin Recht. Keystoneml: Optimizing pipelines for large-scale advanced analytics. In *IEEE ICDE 2017*, pages 535–546, 2017.

[117] Isabelle Stanton and Gabriel Kliot. Streaming graph partitioning for large distributed graphs. Technical Report MSR-TR-2011-121, Microsoft Research, November 2011.

[118] Michael Stonebraker et al. Mapreduce and parallel dbmss: friends or foes? *Commun. ACM*.

[119] Philip Stutz, Abraham Bernstein, and William Cohen. Signal/collect: graph algorithms for the (semantic) web. In *ISWC*, 2010.

[120] Daniel Tahara, Thaddeus Diamond, and Daniel J. Abadi. Sinew: A SQL system for multi-structured data. In *SIGMOD*, 2014.

[121] 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.

[122] Johan Ugander and Lars Backstrom. Balanced label propagation for partitioning massive graphs. In *Proceedings of the Sixth ACM International Conference on Web Search and Data Mining*, WSDM '13, pages 507–516, New York, NY, USA, 2013. ACM.

[123] Philip Wadler. Monads for functional programming. In *Advanced Functional Programming*, pages 24–52. Springer, 1995.

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

[125] Reynold Xin et al. GraySort on Apache Spark by Databricks. http://sortbenchmark.org/ApacheSpark2014.pdf.

[126] 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.

[127] 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.

[128] Burak Yavuz and Tyson Condie. Running streaming jobs once a day for 10x cost savings. https://databricks.com/blog/2017/05/22/running-streaming-jobs-day-10x-cost-savings.html, 2017.

[129] 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.

[130] 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.

[131] 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. In *NSDI*, 2012.

[132] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. NSDI'12, 2012.

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

[134] Kai Zeng et al. G-OLA: Generalized online aggregation for interactive analysis on big data. In *SIGMOD*, 2015.

[135] Yue Zhuge, Héctor García-Molina, Joachim Hammer, and Jennifer Widom. View maintenance in a warehousing environment. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, SIGMOD '95, pages 316–327, New York, NY, USA, 1995. ACM.