Spark SQL: Relational Data Processing in Spark

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

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

ABSTRACT

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

Categories and Subject Descriptors

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

Keywords

Databases; Data Warehouse; Machine Learning; Spark; Hadoop

1 Introduction

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

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

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

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

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

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

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

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

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

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

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

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

2 Background and Goals

2.1 Spark Overview

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

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

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

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

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

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

2.2 Previous Relational Systems on Spark

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

2.3 Goals for Spark SQL

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

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

3 Programming Interface

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

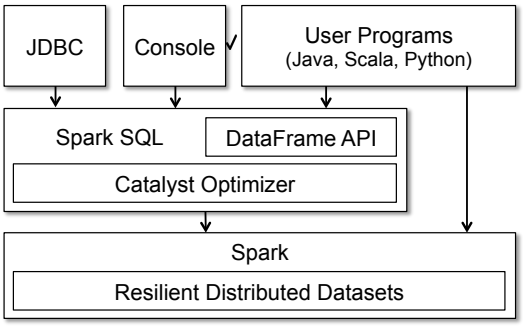


Figure 1: Interfaces to Spark SQL, and interaction with Spark.

3.1 DataFrame API

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

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

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

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

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

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

We next cover the details of the DataFrame API.

3.2 Data Model

Spark SQL uses a nested data model based on Hive [19] for tables and DataFrames. It supports all major SQL data types, including boolean, integer, double, decimal, string, date, and timestamp, as well as complex (i.e., non-atomic) data types: structs, arrays, maps and unions. Complex data types can also be nested together to create more powerful types. Unlike many traditional DBMSes, Spark SQL provides first-class support for complex data types in the query language and the API. In addition, Spark SQL also supports user-defined types, as described in Section 4.4.2.

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

3.3 DataFrame Operations

Users can perform relational operations on DataFrames using a domain-specific language (DSL) similar to R data frames [32] and Python Pandas [30]. DataFrames support all common relational operators, including projection (`select`), filter (`where`), `join`, and aggregations (`groupBy`). These operators all take *expression* objects in a limited DSL that lets Spark capture the structure of the expression. For example, the following code computes the number of female employees in each department.

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

Here, `employees` is a DataFrame, and `employees("deptId")` is an expression representing the `deptId` column. Expression objects have many operators that return new expressions, including the usual comparison operators (*e.g.,* === for equality test, > for greater than) and arithmetic ones (+, -, etc). They also support aggregates, such as `count("name")`. All of these operators build up an abstract syntax tree (AST) of the expression, which is then passed to Catalyst for optimization. This is unlike the native Spark API that takes functions containing arbitrary Scala/Java/Python code, which are then opaque to the runtime engine. For a detailed listing of the API, we refer readers to Spark's official documentation [6].

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

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

SQL is sometimes convenient for computing multiple aggregates concisely, and also allows programs to expose datasets through JDBC/ODBC. The DataFrames registered in the catalog are still unmaterialized views, so that optimizations can happen *across* SQL and the original DataFrame expressions. However, DataFrames can also be materialized, as we discuss in Section 3.6.

3.4 DataFrames versus Relational Query Languages

While on the surface, DataFrames provide the same operations as relational query languages like SQL and Pig [29], we found that they can be significantly easier for users to work with thanks to their integration in a full programming language. For example, users can break up their code into Scala, Java or Python functions that pass DataFrames between them to build a logical plan, and will still benefit from optimizations across the *whole* plan when they run an output operation. Likewise, developers can use control structures like if statements and loops to structure their work. One user said that the DataFrame API is "concise and declarative like SQL, except I can name intermediate results," referring to how it is easier to structure computations and debug intermediate steps.

To simplify programming in DataFrames, we also made API analyze logical plans *eagerly* (*i.e.,* to identify whether the column

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

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

names used in expressions exist in the underlying tables, and whether their data types are appropriate), even though query results are computed lazily. Thus, Spark SQL reports an error as soon as user types an invalid line of code instead of waiting until execution. This is again easier to work with than a large SQL statement.

3.5 Querying Native Datasets

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

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

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

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

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

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

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

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

3.6 In-Memory Caching

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

3.7 User-Defined Functions

User-defined functions (UDFs) have been an important extension point for database systems. For example, MySQL relies on UDFs to provide basic support for JSON data. A more advanced example is MADLib's use of UDFs to implement machine learning algorithms for Postgres and other database systems [12]. However, database systems often require UDFs to be defined in a separate programming environment that is different from the primary query interfaces. Spark SQL's DataFrame API supports inline definition of UDFs, without the complicated packaging and registration process found in other database systems. This feature has proven crucial for the adoption of the API.

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

```
val model: LogisticRegressionModel = ...

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

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

Once registered, the UDF can also be used via the JDBC/ODBC interface by business intelligence tools. In addition to UDFs that operate on scalar values like the one here, one can define UDFs that operate on an entire table by taking its name, as in MADLib [12], and use the distributed Spark API within them, thus exposing advanced analytics functions to SQL users. Finally, because UDF definitions and query execution are expressed using the same general-purpose language (*e.g.,* Scala or Python), users can debug or profile the entire program using standard tools.

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

4 Catalyst Optimizer

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

While extensible optimizers have been proposed in the past, they have typically required a complex domain specific language to specify rules, and an "optimizer compiler" to translate the rules into executable code [17, 16]. This leads to a significant learning curve and maintenance burden. In contrast, Catalyst uses standard features of the Scala programming language, such as pattern-matching [14], to let developers use the full programming language while still making rules easy to specify. Functional languages were designed in part to build compilers, so we found Scala well-suited to this task. Nonetheless, Catalyst is, to our knowledge, the first production-quality query optimizer built on such a language.

At its core, Catalyst contains a general library for representing *trees* and applying *rules* to manipulate them.[3] On top of this framework, we have built libraries specific to relational query processing (*e.g.,* expressions, logical query plans), and several sets of rules that handle different phases of query execution: analysis, logical optimization, physical planning, and code generation to compile parts of queries to Java bytecode. For the latter, we use another Scala feature, quasiquotes [34], that makes it easy to generate code at runtime from composable expressions. Finally, Catalyst offers several public extension points, including external data sources and user-defined types.

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

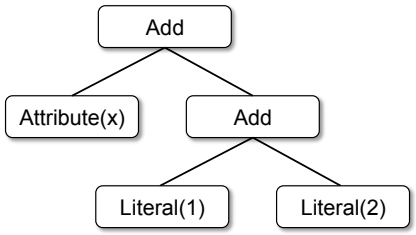


Figure 2: Catalyst tree for the expression `x+(1+2)`.

4.1 Trees

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

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

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

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

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

4.2 Rules

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

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

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

Applying this to the tree for `x+(1+2)`, in Figure 2, would yield the new tree `x+3`. The `case` keyword here is Scala's standard pattern matching syntax [14], and can be used to match on the type of an object as well as give names to extracted values (`c1` and `c2` here).

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

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

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

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

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

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

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

4.3 Using Catalyst in Spark SQL

We use Catalyst's general tree transformation framework in four phases, shown in Figure 3: (1) analyzing a logical plan to resolve references, (2) logical plan optimization, (3) physical planning, and (4) code generation to compile parts of the query to Java bytecode. In the physical planning phase, Catalyst may generate multiple plans and compare them based on cost. All other phases are purely rule-based. Each phase uses different types of tree nodes; Catalyst includes libraries of nodes for expressions, data types, and logical and physical operators. We now describe each of these phases.

4.3.1 Analysis

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

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

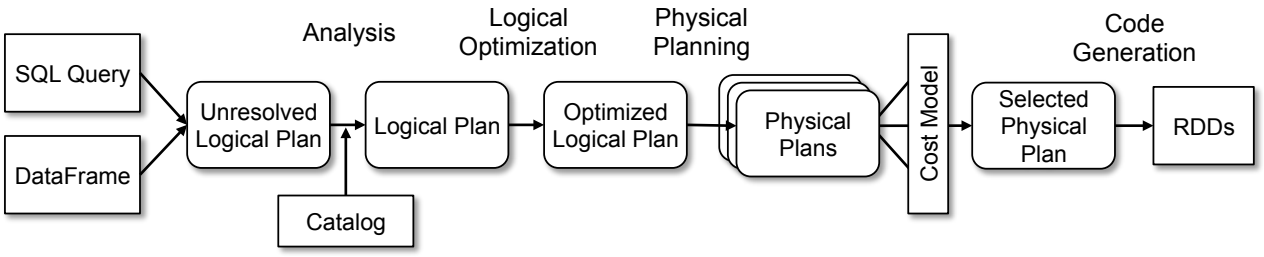


Figure 3: Phases of query planning in Spark SQL. Rounded rectangles represent Catalyst trees.

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

4.3.2 Logical Optimization

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

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

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

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

4.3.3 Physical Planning

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

The physical planner also performs rule-based physical optimizations, such as pipelining projections or filters into one Spark `map` operation. In addition, it can push operations from the logical plan into data sources that support predicate or projection pushdown. We will describe the API for these data sources in Section 4.4.1.

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

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

4.3.4 Code Generation

The final phase of query optimization involves generating Java bytecode to run on each machine. Because Spark SQL often operates on in-memory datasets, where processing is CPU-bound, we wanted to support code generation to speed up execution. Nonetheless, code generation engines are often complicated to build, amounting essentially to a compiler. Catalyst relies on a special feature of the Scala language, quasiquotes [34], to make code generation simpler. Quasiquotes allow the programmatic construction of abstract syntax trees (ASTs) in the Scala language, which can then be fed to the Scala compiler at runtime to generate bytecode. We use Catalyst to transform a tree representing an expression in SQL to an AST for Scala code to evaluate that expression, and then compile and run the generated code.

As a simple example, consider the Add, Attribute and Literal tree nodes introduced in Section 4.2, which allowed us to write expressions such as `(x+y)+1`. Without code generation, such expressions would have to be interpreted for each row of data, by walking down a tree of Add, Attribute and Literal nodes. This introduces large amounts of branches and virtual function calls that slow down execution. With code generation, we can write a function to translate a specific expression tree to a Scala AST as follows:

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

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

Quasiquotes are type-checked at compile time to ensure that only appropriate ASTs or literals are substituted in, making them significantly more useable than string concatenation, and they result directly in a Scala AST instead of running the Scala parser at runtime. Moreover, they are highly composable, as the code generation rule for each node does not need to know how the trees returned by its children were built. Finally, the resulting code is further optimized by the Scala compiler in case there are expression-level optimizations that Catalyst missed. Figure 4 shows that quasiquotes let us generate code with performance similar to hand-tuned programs.

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

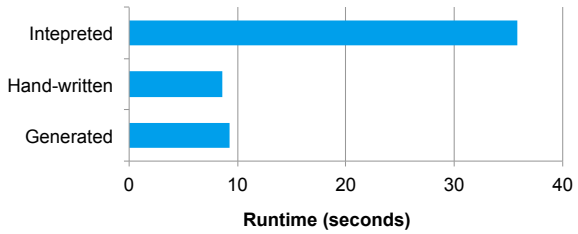


Figure 4: A comparision of the performance evaluating the expresion `x+x+x`, where `x` is an integer, 1 billion times.

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

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

4.4 Extension Points

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

4.4.1 Data Sources

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

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

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

- CSV files, which simply scan the whole file, but allow users to specify a schema.
- Avro [4], a self-describing binary format for nested data.
- Parquet [5], a columnar file format for which we support column pruning as well as filters.
- A JDBC data source that scans ranges of a table from an RDBMS in parallel and pushes filters into the RDBMS to minimize communication.

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

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

All data sources can also expose network locality information, *i.e.,* which machines each partition of the data is most efficient to read from. This is exposed through the RDD objects they return, as RDDs have a built-in API for data locality [39].

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

4.4.2 User-Defined Types (UDTs)

One feature we wanted to allow advanced analytics in Spark SQL was user-defined types. For example, machine learning applications may need a vector type, and graph algorithms may need types for representing a graph, which is possible over relational tables [15]. Adding new types can be challenging, however, as data types pervade all aspects of the execution engine. For example, in Spark SQL, the built-in data types are stored in a columnar, compressed format for in-memory caching (Section 3.6), and in the data source API from the previous section, we need to expose all possible data types to data source authors.

In Catalyst, we solve this issue by mapping user-defined types to structures composed of Catalyst's built-in types, described in Section 3.2. To register a Scala type as a UDT, users provide a mapping from an object of their class to a Catalyst Row of built-in types, and an inverse mapping back. In user code, they can now use the Scala type in objects that they query with Spark SQL, and it will be converted to built-in types under the hood. Likewise, they can register UDFs (see Section 3.7) that operate directly on their type.

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

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

After registering this type, Points will be recognized within native objects that Spark SQL is asked to convert to DataFrames, and will be passed to UDFs defined on Points. In addition, Spark SQL will store Points in a columnar format when caching data (compressing `x` and `y` as separate columns), and Points will be writable to all of Spark SQL's data sources, which will see them as pairs of `DOUBLE`s. We use this capability in Spark's machine learning library, as we describe in Section 5.2.

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

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

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

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

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

Figure 5: A sample set of JSON records, representing tweets.

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

Figure 6: Schema inferred for the tweets in Figure 5.

5 Advanced Analytics Features

In this section, we describe three features we added to Spark SQL specifically to handle challenges in "big data" environments. First, in these environments, data is often unstructured or semistructured. While parsing such data procedurally is possible, it leads to lengthy boilerplate code. To let users query the data right away, Spark SQL includes a schema inference algorithm for JSON and other semistructured data. Second, large-scale processing often goes beyond aggregation and joins to machine learning on the data. We describe how Spark SQL is being incorporated into a new high-level API for Spark's machine learning library [26]. Last, data pipelines often combine data from disparate storage systems. Building on the data sources API in Section 4.4.1, Spark SQL supports query federation, allowing a single program to efficiently query disparate sources. These features all build on the Catalyst framework.

5.1 Schema Inference for Semistructured Data

Semistructured data is common in large-scale environments because it is easy to produce and to add fields to over time. Among Spark users, we have seen very high usage of JSON for input data. Unfortunately, JSON is cumbersome to work with in a procedural environment like Spark or MapReduce: most users resorted to ORM-like libraries (*e.g.,* Jackson [21]) to map JSON structures to Java objects, or some tried parsing each input record directly with lower-level libraries.

In Spark SQL, we added a JSON data source that automatically infers a schema from a set of records. For example, given the JSON objects in Figure 5, the library infers the schema shown in Figure 6. Users can simply register a JSON file as a table and query it with syntax that accesses fields by their path, such as:

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

Our schema inference algorithm works in one pass over the data, and can also be run on a sample of the data if desired. It is related to prior work on schema inference for XML and object databases [9, 18, 27], but simpler because it only infers a static tree structure, without allowing recursive nesting of elements at arbitrary depths.

Specifically, the algorithm attempts to infer a tree of `STRUCT` types, each of which may contain atoms, arrays, or other `STRUCT`s. For each field defined by a distinct path from the root JSON object (*e.g.,* `tweet.loc.latitude`), the algorithm finds the most specific Spark SQL data type that matches observed instances of the field. For example, if all occurrences of that field are integers that fit into 32 bits, it will infer `INT`; if they are larger, it will use `LONG` (64-bit) or `DECIMAL` (arbitrary precision); if there are also fractional values, it will use `FLOAT`. For fields that display multiple types, Spark SQL uses `STRING` as the most generic type, preserving the original JSON representation. And for fields that contain arrays, it uses the same "most specific supertype" logic to determine an element type from all the observed elements. We implement this algorithm using a single `reduce` operation over the data, which starts with schemata (*i.e.,* trees of types) from each individual record and merges them using an associative "most specific supertype" function that generalizes the types of each field. This makes the algorithm both single-pass and communication-efficient, as a high degree of reduction happens locally on each node.

As a short example, note how in Figures 5 and 6, the algorithm generalized the types of `loc.lat` and `loc.long`. Each field appears as an integer in one record and a floating-point number in another, so the algorithm returns `FLOAT`. Note also how for the `tags` field, the algorithm inferred an array of strings that cannot be null.

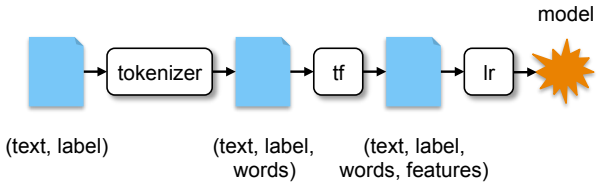
In practice, we have found this algorithm to work well with real-world JSON datasets. For example, it correctly identifies a usable schema for JSON tweets from Twitter's firehose, which contain around 100 distinct fields and a high degree of nesting. Multiple Databricks customers have also successfully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

In Spark SQL, we also use the same algorithm for inferring schemas of RDDs of Python objects (see Section 3), as Python is not statically typed so an RDD can contain multiple object types. In the future, we plan to add similar inference for CSV files and XML. Developers have found the ability to view these types of datasets as tables and immediately query them or join them with other data extremely valuable for their productivity.

5.2 Integration with Spark's Machine Learning Library

As an example of Spark SQL's utility in other Spark modules, MLlib, Spark's machine learning library, introduced a new high-level API that uses DataFrames [26]. This new API is based on the concept of machine learning *pipelines*, an abstraction in other high-level ML libraries like SciKit-Learn [33]. A pipeline is a graph of transformations on data, such as feature extraction, normalization, dimensionality reduction, and model training, each of which exchange *datasets*. Pipelines are a useful abstraction because ML workflows have many steps; representing these steps as composable elements makes it easy to change parts of the pipeline or to search for tuning parameters at the level of the whole workflow.

To exchange data between pipeline stages, MLlib's developers needed a format that was compact (as datasets can be large) yet flexible, allowing multiple types of fields to be stored for each record. For example, a user may start with records that contain text fields as well as numeric ones, then run a featurization algorithm such as TF-IDF on the text to turn it into a vector, normalize one of the other fields, perform dimensionality reduction on the whole set of features, etc. To represent datasets, the new API uses DataFrames, where each column represents a feature of the data. All algorithms that can be called in pipelines take a name for the input column(s) and output column(s), and can thus be called on any subset of the fields and produce new ones. This makes it easy for developers to build complex pipelines while retaining the original data for each record. To illustrate the API, Figure 7 shows a short pipeline and the schemas of DataFrames created.




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

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

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

Figure 7: A short MLlib pipeline and the Python code to run it. We start with a DataFrame of (text, label) records, tokenize the text into words, run a term frequency featurizer (`HashingTF`) to get a feature vector, then train logistic regression.

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

Finally, using DataFrames for storage in MLlib also makes it very easy to expose all its algorithms in SQL. We can simply define a MADlib-style UDF, as described in Section 3.7, which will internally call the algorithm on a table. We are also exploring APIs to expose pipeline construction in SQL.

5.3 Query Federation to External Databases

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

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

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

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

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

Under the hood, the JDBC data source uses the PrunedFilteredScan interface in Section 4.4.1, which gives it both the names of the columns requested and simple predicates (equality, comparison and `IN` clauses) on these columns. In this case, the JDBC data source will run the following query on MySQL:[8]

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

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

6 Evaluation

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

6.1 SQL Performance

We compared the performance of Spark SQL against Shark and Impala [23] using the AMPLab big data benchmark [3], which uses a web analytics workload developed by Pavlo et al. [31]. The benchmark contains four types of queries with different parameters performing scans, aggregation, joins and a UDF-based MapReduce job. We used a cluster of six EC2 `i2.xlarge` machines (one master, five workers) each with 4 cores, 30 GB memory and an 800 GB SSD, running HDFS 2.4, Spark 1.3, Shark 0.9.1 and Impala 2.1.1. The dataset was 110 GB of data after compression using the columnar Parquet format [5].

Figure 8 shows the results for each query, grouping by the query type. Queries 1–3 have different parameters varying their selectivity, with 1a, 2a, etc being the most selective and 1c, 2c, etc being the least selective and processing more data. Query 4 uses a Python-based Hive UDF that was not directly supported in Impala, but was largely bound by the CPU cost of the UDF.

We see that in all queries, Spark SQL is substantially faster than Shark and generally competitive with Impala. The main reason for the difference with Shark is code generation in Catalyst (Section 4.3.4), which reduces CPU overhead. This feature makes Spark SQL competitive with the C++ and LLVM based Impala engine in many of these queries. The largest gap from Impala is in query 3a where Impala chooses a better join plan because the selectivity of the queries makes one of the tables very small.

6.2 DataFrames vs. Native Spark Code

In addition to running SQL queries, Spark SQL can also help non-SQL developers write simpler and more efficient Spark code through the DataFrame API. Catalyst can perform optimizations on

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

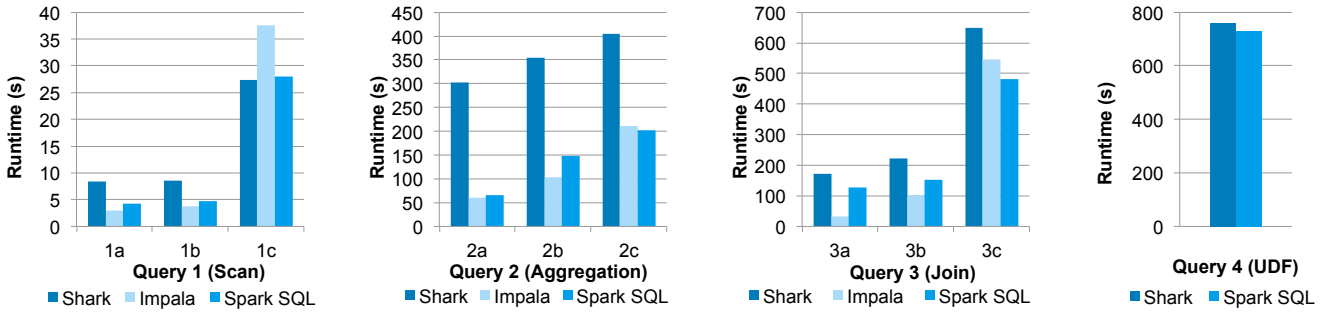


Figure 8: Performance of Shark, Impala and Spark SQL on the big data benchmark queries [31].

DataFrame operations that are hard to do with hand written code, such as predicate pushdown, pipelining, and automatic join selection. Even without these optimizations, the DataFrame API can result in more efficient execution due to code generation. This is especially true for Python applications, as Python is typically slower than the JVM.

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

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

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

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

Figure 9, shows that the DataFrame version of the code outperforms the hand written Python version by 12×, in addition to being much more concise. This is because in the DataFrame API, only the logical plan is constructed in Python, and all physical execution is compiled down into native Spark code as JVM bytecode, resulting in more efficient execution. In fact, the DataFrame version also outperforms a Scala version of the Spark code above by 2×. This is mainly due to code generation: the code in the DataFrame version avoids expensive allocation of key-value pairs that occurs in hand-written Scala code.

6.3 Pipeline Performance

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

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

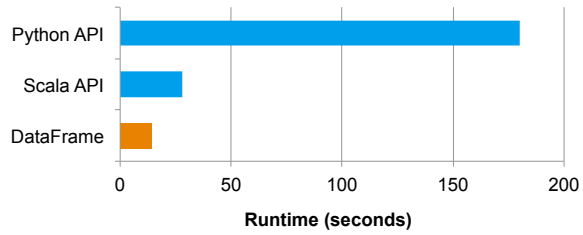


Figure 9: Performance of an aggregation written using the native Spark Python and Scala APIs versus the DataFrame API.

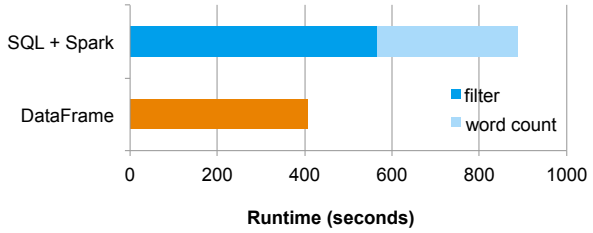


Figure 10: Performance of a two-stage pipeline written as a separate Spark SQL query and Spark job (above) and an integrated DataFrame job (below).

First, we implemented the pipeline using a separate SQL query followed by a Scala-based Spark job, as might occur in environments that run separate relational and procedural engines (*e.g.,* Hive and Spark). We then implemented a combined pipeline using the DataFrame API, *i.e.,* using DataFrame's relational operators to perform the filter, and using the RDD API to perform a word count on the result. Compared with the first pipeline, the second pipeline avoids the cost of saving the whole result of the SQL query to an HDFS file as an intermediate dataset before passing it into the Spark job, because SparkSQL pipelines the `map` for the word count with the relational operators for the filtering. Figure 10 compares the runtime performance of the two approaches. In addition to being easier to understand and operate, the DataFrame-based pipeline also improves performance by 2×.

7 Research Applications

In addition to the immediately practical production use cases of Spark SQL, we have also seen significant interest from researchers working on more experimental projects. We outline two research

projects that leverage the extensibility of Catalyst: one in approximate query processing and one in genomics.

7.1 Generalized Online Aggregation

Zeng et al. have used Catalyst in their work on improving the generality of online aggregation [40]. This work generalizes the execution of online aggregation to support arbitrarily nested aggregate queries. It allows users to view the progress of executing queries by seeing results computed over a fraction of the total data. These partial results also include accuracy measures, letting the user stop the query when sufficient accuracy has been reached.

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

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

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

7.2 Computational Genomics

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

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

Without special optimization, the preceding query would be executed by many systems using an inefficient algorithm such as a nested loop join. In contrast, a specialized system could compute the answer to this join using an interval tree. Researchers in the ADAM project [28] were able to build a special planning rule into a version of Spark SQL to perform such computations efficiently, allowing them to leverage the standard data manipulation abilities alongside specialized processing code. The changes required were approximately 100 lines of code.

8 Related Work

Programming Model Several systems have sought to combine relational processing with the procedural processing engines initially used for large clusters. Of these, Shark [38] is the closest to Spark SQL, running on the same engine and offering the same combination of relational queries and advanced analytics. Spark SQL improves on Shark through a richer and more programmer-friendly API, DataFrames, where queries can be combined in a modular way using constructs in the host programming language (see Section 3.4). It also allows running relational queries directly on native RDDs, and supports a wide range of data sources beyond Hive.

One system that inspired Spark SQL's design was DryadLINQ [20], which compiles language-integrated queries in C# to a distributed DAG execution engine. LINQ queries are also relational but can operate directly on C# objects. Spark SQL goes beyond DryadLINQ by also providing a DataFrame interface similar to common data science libraries [32, 30], an API for data sources and types, and support for iterative algorithms through execution on Spark.

Other systems use only a relational data model internally and relegate procedural code to UDFs. For example, Hive and Pig [36, 29] offer relational query languages but have widely used UDF interfaces. ASTERIX [8] has a semi-structured data model internally. Stratosphere [2] also has a semi-structured model, but offers APIs in Scala and Java that let users easily call UDFs. PIQL [7] likewise provides a Scala DSL. Compared to these systems, Spark SQL integrates more closely with native Spark applications by being able to directly query data in user-defined classes (native Java/Python objects), and lets developers mix procedural and relational APIs in the same language. In addition, through the Catalyst optimizer, Spark SQL implements both optimizations (*e.g.,* code generation) and other functionality (*e.g.,* schema inference for JSON and machine learning data types) that are not present in most large-scale computing frameworks. We believe that these features are essential to offering an integrated, easy-to-use environment for big data.

Finally, data frame APIs have been built both for single machines [32, 30] and clusters [13, 10]. Unlike previous APIs, Spark SQL optimizes DataFrame computations with a relational optimizer.

Extensible Optimizers The Catalyst optimizer shares similar goals with extensible optimizer frameworks such as EXODUS [17] and Cascades [16]. Traditionally, however, optimizer frameworks have required a domain-specific language to write rules in, as well as an "optimizer compiler" to translate them to runnable code. Our major improvement here is to build our optimizer using standard features of a functional programming language, which provide the same (and often greater) expressivity while decreasing the maintenance burden and learning curve. Advanced language features helped with many areas of Catalyst—for example, our approach to code generation using quasiquotes (Section 4.3.4) is one of the simplest and most composable approaches to this task that we know. While extensibility is hard to measure quantitatively, one promising indication is that Spark SQL had over 50 external contributors in the first 8 months after its release.

For code generation, LegoBase [22] recently proposed an approach using generative programming in Scala, which would be possible to use instead of quasiquotes in Catalyst.

Advanced Analytics Spark SQL builds on recent work to run advanced analytics algorithms on large clusters, including platforms for iterative algorithms [39] and graph analytics [15, 24]. The desire to expose analytics functions is also shared with MADlib [12], though the approach there is different, as MADlib had to use the limited interface of Postgres UDFs, while Spark SQL's UDFs can be full-fledged Spark programs. Finally, techniques including Sinew and Invisible Loading [35, 1] have sought to provide and optimize queries over semi-structured data such as JSON. We hope to apply some of these techniques in our JSON data source.

9 Conclusion

We have presented Spark SQL, a new module in Apache Spark providing rich integration with relational processing. Spark SQL extends Spark with a declarative DataFrame API to allow relational processing, offering benefits such as automatic optimization, and letting users write complex pipelines that mix relational and complex analytics. It supports a wide range of features tailored to large-scale

data analysis, including semi-structured data, query federation, and data types for machine learning. To enable these features, Spark SQL is based on an extensible optimizer called Catalyst that makes it easy to add optimization rules, data sources and data types by embedding into the Scala programming language. User feedback and benchmarks show that Spark SQL makes it significantly simpler and more efficient to write data pipelines that mix relational and procedural processing, while offering substantial speedups over previous SQL-on-Spark engines.

Spark SQL is open source at `http://spark.apache.org`.

10 Acknowledgments

We would like to thank Cheng Hao, Tayuka Ueshin, Tor Myklebust, Daoyuan Wang, and the rest of the Spark SQL contributors so far. We would also like to thank John Cieslewicz and the other members of the F1 team at Google for early discussions on the Catalyst optimizer. The work of authors Franklin and Kaftan was supported in part by: NSF CISE Expeditions Award CCF-1139158, LBNL Award 7076018, and DARPA XData Award FA8750-12-2-0331, and gifts from Amazon Web Services, Google, SAP, The Thomas and Stacey Siebel Foundation, Adatao, Adobe, Apple, Inc., Blue Goji, Bosch, C3Energy, Cisco, Cray, Cloudera, EMC2, Ericsson, Facebook, Guavus, Huawei, Informatica, Intel, Microsoft, NetApp, Pivotal, Samsung, Schlumberger, Splunk, Virdata and VMware.

11 References

[1] A. Abouzied, D. J. Abadi, and A. Silberschatz. Invisible loading: Access-driven data transfer from raw files into database systems. In *EDBT*, 2013.

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

[3] AMPLab big data benchmark. https://amplab.cs.berkeley.edu/benchmark.

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

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

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

[7] M. Armbrust, N. Lanham, S. Tu, A. Fox, M. J. Franklin, and D. A. Patterson. The case for PIQL: a performance insightful query language. In *SOCC*, 2010.

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

[9] G. J. Bex, F. Neven, and S. Vansummeren. Inferring XML schema definitions from XML data. In *VLDB*, 2007.

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

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

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

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

[14] B. Emir, M. Odersky, and J. Williams. Matching objects with patterns. In *ECOOP 2007 – Object-Oriented Programming, volume 4609 of LNCS*, pages 273–298. Springer, 2007.

[15] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. GraphX: Graph processing in a distributed dataflow framework. In *OSDI*, 2014.

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

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

[18] J. Hegewald, F. Naumann, and M. Weis. XStruct: efficient schema extraction from multiple and large XML documents. In *ICDE Workshops*, 2006.

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

[20] M. Isard and Y. Yu. Distributed data-parallel computing using a high-level programming language. In *SIGMOD*, 2009.

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

[22] Y. Klonatos, C. Koch, T. Rompf, and H. Chafi. Building efficient query engines in a high-level language. *PVLDB*, 7(10):853–864, 2014.

[23] M. Kornacker et al. Impala: A modern, open-source SQL engine for Hadoop. In *CIDR*, 2015.

[24] Y. Low et al. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *VLDB*, 2012.

[25] S. Melnik et al. Dremel: interactive analysis of web-scale datasets. *Proc. VLDB Endow.*, 3:330–339, Sept 2010.

[26] X. Meng, J. Bradley, E. Sparks, and S. Venkataraman. ML pipelines: a new high-level API for MLlib. https://databricks.com/blog/2015/01/07/ml-pipelines-a-new-high-level-api-for-mllib.html.

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

[28] F. A. Nothaft, M. Massie, T. Danford, Z. Zhang, U. Laserson, C. Yeksigian, J. Kottalam, A. Ahuja, J. Hammerbacher, M. Linderman, M. J. Franklin, A. D. Joseph, and D. A. Patterson. Rethinking data-intensive science using scalable analytics systems. In *SIGMOD*, 2015.

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

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

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

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

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

[34] D. Shabalin, E. Burmako, and M. Odersky. Quasiquotes for Scala, a technical report. Technical Report 185242, École Polytechnique Fédérale de Lausanne, 2013.

[35] D. Tahara, T. Diamond, and D. J. Abadi. Sinew: A SQL system for multi-structured data. In *SIGMOD*, 2014.

[36] A. Thusoo et al. Hive–a petabyte scale data warehouse using Hadoop. In *ICDE*, 2010.

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

[38] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: SQL and rich analytics at scale. In *SIGMOD*, 2013.

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

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