SPARK SQL

Relational Data Processing in Spark

Earlier Attempts

- MapReduce
  - Powerful, **low-level**, procedural programming interface.
  - **Onerous** and require **manual optimization**
- Pig, Hive, **Dremel,** Shark
  - *Take advantage of **declarative queries** to provide richer **automatic optimizations.***
  - *Relational approach is insufficient for big data applications*
    - ETL to/from semi-/unstructured data sources (e.g. JSON) requires custom code
    - Advanced analytics(ML and graph processing) are challenging to express in relational system.

Spark SQL(2014)

- A new module in Apache Spark that integrates relational processing with Spark's functional programming API.
- Offers much tighter **integration between relational and procedural** in processing, through a **declarative** DataFrame API.
- Includes a highly extensible optimizer, Catalyst, that makes it easy to **add data sources, optimization rules, and data types**.

Apache Spark(2010)

- General cluster computing system.
- One of the most widely-used systems with a **“language-integrated” API.**
- One of the most active open source project for **big data processing.**
- Manipulates(e.g. map, filter, reduce) distributed collections called Resilient Distributed Datasets (RDD).
- RDDs are evaluated lazily.

Scala RDD Example:
Counts lines starting with "ERROR" in an HDFS file

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

- Each RDD(lines, errors) represents a "logical plan" to compute a dataset, but Spark waits until certain output operations, **count**, to launch a computation.
- Spark will pipeline reading lines, applying filter and computer counts.
- No intermediate materialization needed.
- Useful but limited.

Shark

- First effort to build a relational interface on Spark.
- Modified the Apache Hive system with traditional RDBMS optimizations,
- Shows good performance and opportunities for integration with Spark programs.
- Challenges
  - *Only query external data stored in the Hive catalog, and was thus not useful for relational queries on data inside a Spark program(e.g. RDD errors).*
  - *Inconvenient and error-prone to work with.*
  - *Hive optimizer was tailored for MapReduce and difficult to extend.*

Goals for Spark SQL

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

Programming Interface

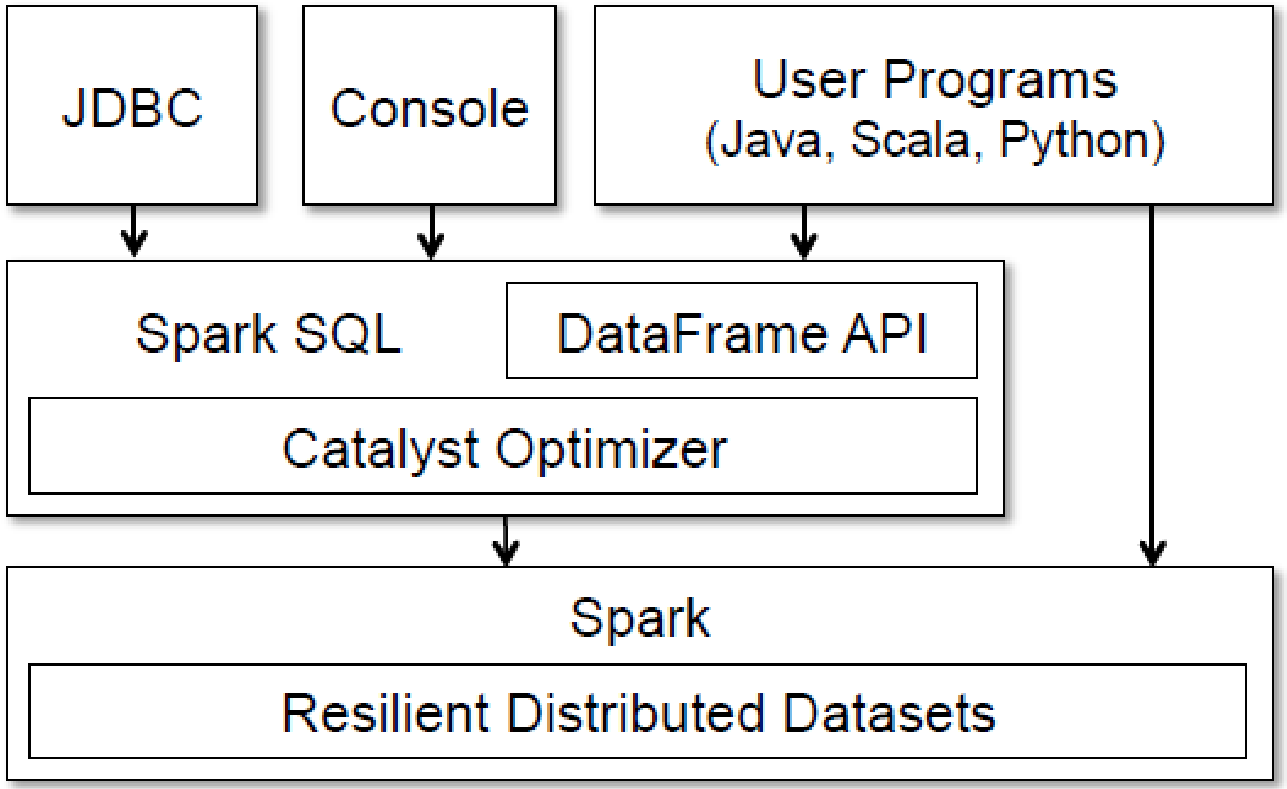


Figure 1: Interfaces to Spark SQL, and interaction with Spark.

DataFrame API:

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

- Equivalent to a table in relational database

- Can be manipulated in similar ways to the “native” RDD.

Data Model

- Uses a nested data model based on Hive for tables and DataFrames
  - *Supports all major SQL data types*

- Supports user-defined types

- Able to model data from a variety sources and formats(e.g. Hive, RDB, JSON, and native objects in Java/Dcala/Python)

DataFrame Operations

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

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

- All of these operators build up an abstract syntax tree (AST) of the expression, which is then passed to **Catalyst** for optimization.
- The DataFrames registered in the catalog can still be **unmaterialized views**, so that optimizations can happen across SQL and the original DataFrame expressions.
- Integration in a full programming language(DataFrames can be passed Inter-language but still benefit from optimization across the whole plan).

Querying Native Datasets

- Allows users to construct DataFrames directly against RDDs of objects native to the programming language.
- Automatically infer the schema and types of the objects.
- Accesses the native objects in-place, extracting only the fields used in each query (avoid expensive conversions).

```
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
```

In-Memory Caching

Columnar cache can reduce memory footprint by an order of magnitude

User-Defined Functions

supports inline definition of UDFs (avoid complicated packaging and registration process)

Catalyst Optimizer

- Based on functional programming constructs in Scala.
- Easy to add new optimization techniques and features,
  - *Especially to tackle various problems when dealing with "big data"(e.g. semi-structured data and advanced analytics)*
- Enable external developers to extend the optimizer.
  - *Data source specific rules that can push filtering or aggregation into external storage systems*
  - *Support for new data type*
- Supports rule-based and cost-based optimization

- First production-quality query optimizer built on such a language (Scala).

Trees

Scala Code :Add(Attribute(x), Add(Literal(1), Literal(2)))

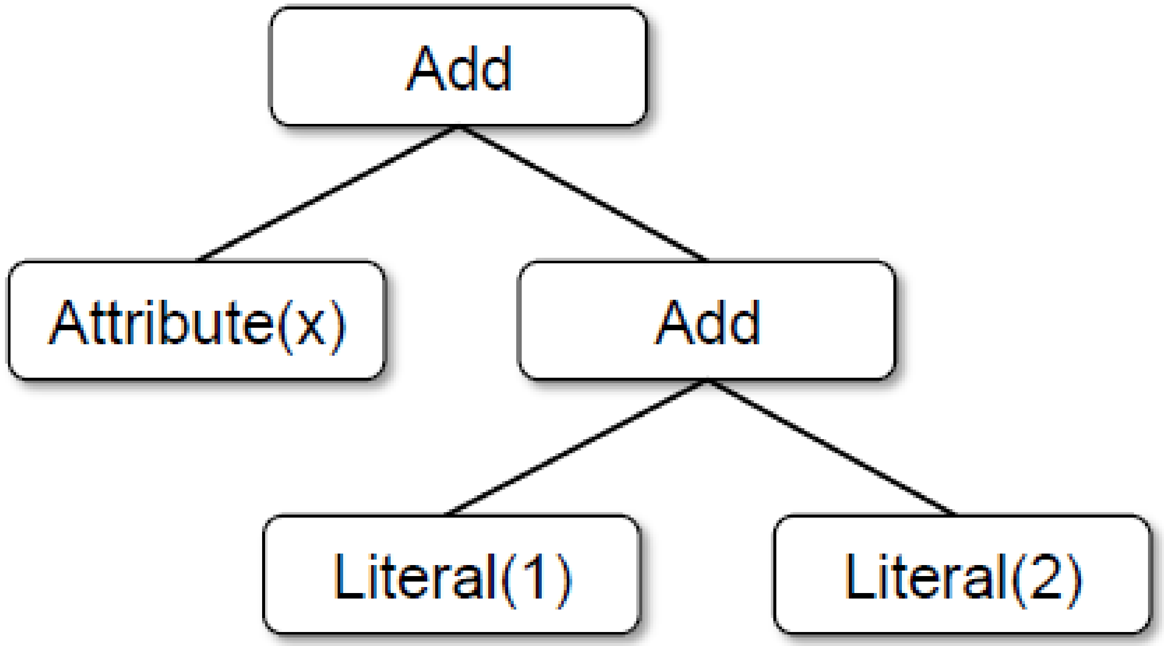


Figure 2: Catalyst tree for the expression `x+(1+2)`**.**

Rules

- Trees can be manipulated using rules, which are functions from a tree to another tree.
  - *Use a set of **pattern matching** functions that find and replace subtrees with a specific structure.*
  - *tree.transform{*

```
        case Add(Literal(c1), Literal(c2)) => Literal(c1+c2)
 }
```

  - *tree.transform {*

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

- Catalyst groups rules into batches, and executes each batch until it reaches a fixed point.

Using Catalyst

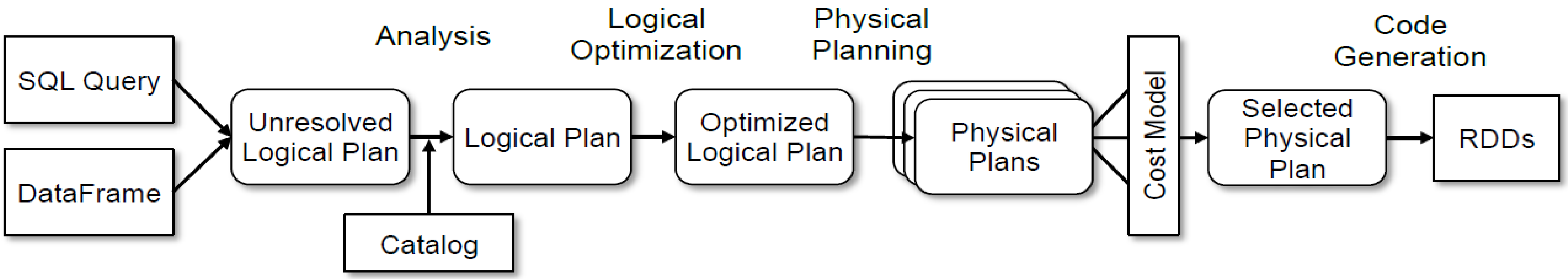


Figure 3: Phases of query planning in Spark SQL. Rounded rectangles represent Catalyst trees.

Analysis

SELECT col FROM sales

- Takes input from SQL parser or DataFrame object
- Unresolved : have not matched it to input table or do not know type
- Catalog object tracks the tables in all data sources
- Around 1000 lines of rules

Logical Optimization

- Applies standard rule-based optimizations to the logical plan
  - *Constant folding*
  - *Predicate pushdown*
  - *Projection pruning*
  - *Null propagation*
  - *Boolean expression simplification*
  - *…*
- Extremely easy to add rules for specific situation
- Around 800 lines of rules

Physical Planning

- Take a Logical Plan and generates one or more physical plans.
- Cost-based
  - *selects a plan using a cost model.(currently only used to select join algorithm)*
- Rule-based:
  - *Pipelining projections or filter into one Spark map operation*
  - *Push operations from the logical plan into data sources that support predicate or projection pushdown.*
- Around 500 lines of rules.

Code Generation

- Generates Java bytecode to run on each machine.
- Relies on quasiquotes of Scala to wrap codes into trees
- Transform a tree representing an expression in SQL to an AST for Scala to evaluate that expression.
- Compile(optimized by Scala again) and run the generated code.
- Around 700 lines of rules

```
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)}"
}
```

Performance by using quasiquotes

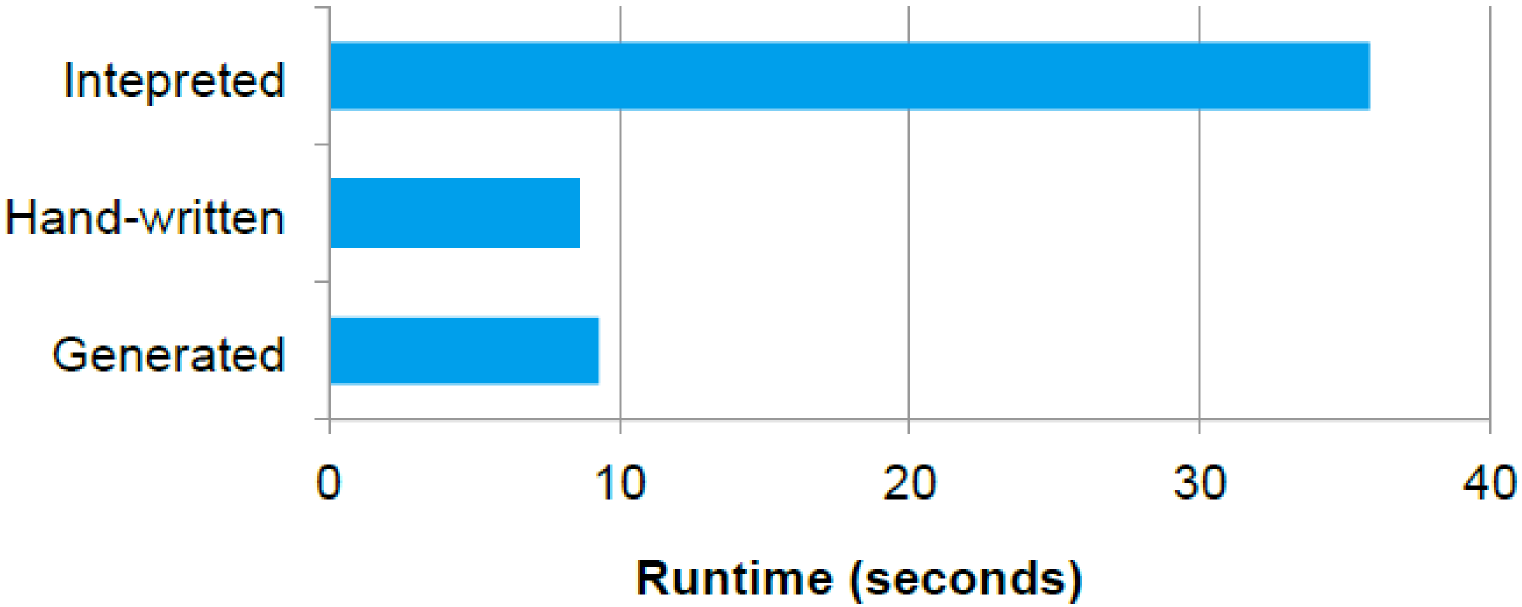


Figure 4: A comparision of the performance evaluating the expresion x+x+x, where x is an integer, 1 billion times.

Extension Points

- Catalyst’s design around composable rules makes it easy to extend.

- Data Source
  - *CSV, Avro, Parquet, etc.*
- User-Defined Types (UDTs)
  - *Mapping user-defined types to structures composed of Catalyst’s built-in types.*

Advanced Analytics Features

Specifically designed to handle “big data”

- A schema inference algorithm for JSON and other semi-structured data.
- A new high-level API for Spark’s machine learning library.
- Supports query federation, allowing a single program to efficiently query disparate sources.

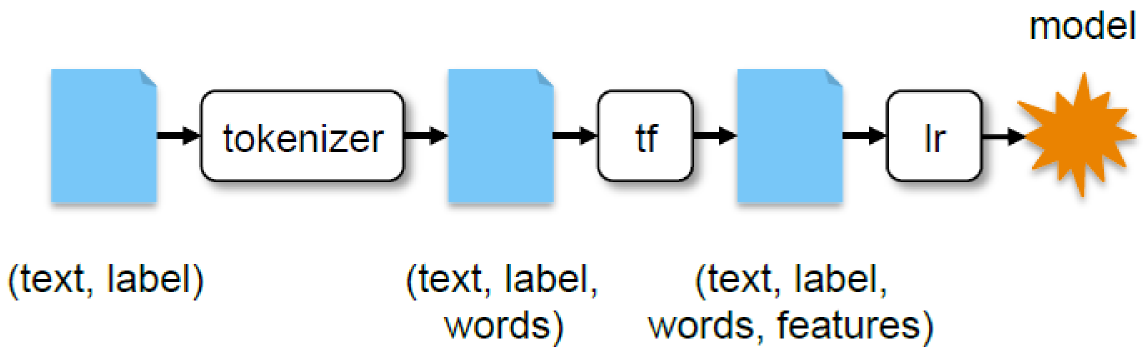
```
{
  "text": "This is a tweet about #Spark",
  "tags": ["#Spark"],
  "loc": {"lat": 45.1, "long": 90}
}

{
  "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.

Integration with Spark's Machine Learning Library




```
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.

SQL Performance

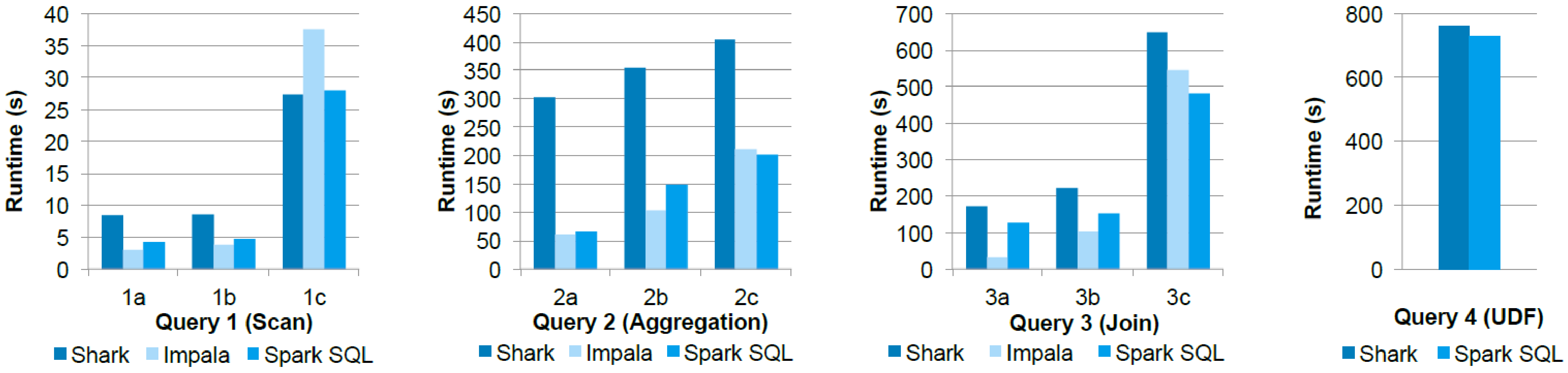


Figure 8: Performance of Shark, Impala and Spark SQL on the big data benchmark queries [31].

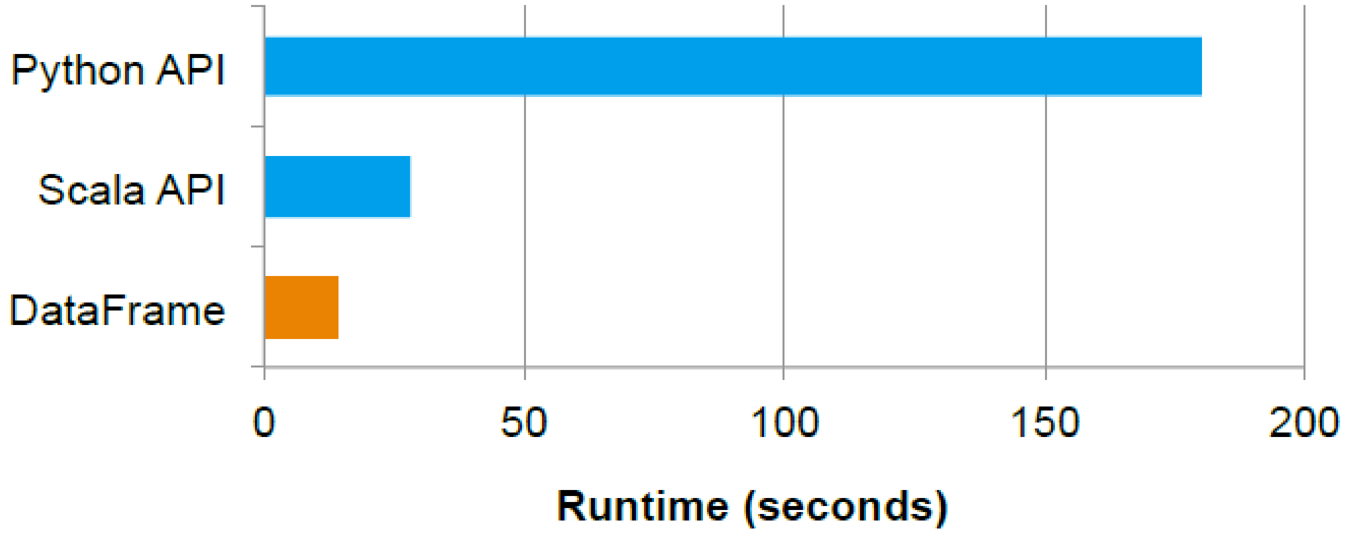
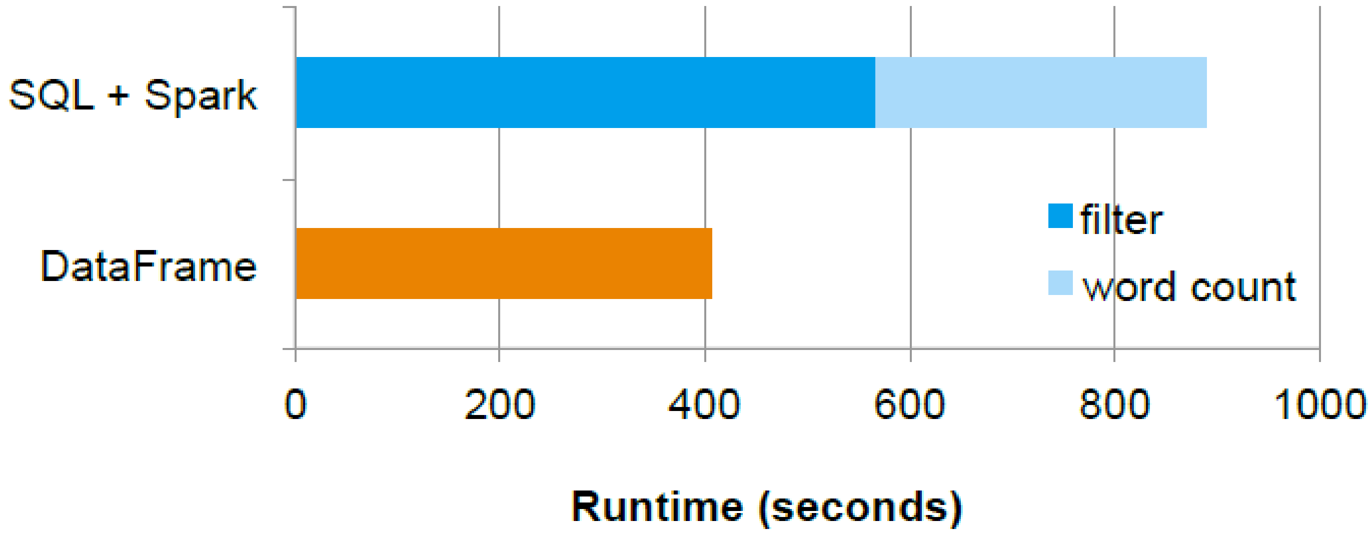


Figure 9: Performance of an aggregation written using the native Spark Python and Scala APIs versus the DataFrame API.



Conclusion

- 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.
- Supports a wide range of features tailored to large-scale data analysis, including semi-structured data, query federation, and data types for machine learning.