databricks

# Apache Spark™ Analytics Made Simple

Highlights from the Databricks Blog

# Apache Spark™ Analytics Made Simple

Highlights from the Databricks Blog

By Michael Armbrust, Wenchen Fan, Vida Ha, Yin Huai, Davies Liu, Kavitha Mariappan, Ion Stoica, Reynold Xin, Burak Yavuz, and Matei Zaharia

Special thanks to our guest authors Grega Kešpret from Celtra and Andrew Ray from Silicon Valley Data Science.



1st in a series from Databricks:


Databricks
160 Spear Street, 13
San Francisco, CA 94105
Contact Us

## About Databricks

Databricks' mission is to accelerate innovation for its customers by unifying Data Science, Engineering and Business. Founded by the team who created Apache Spark™, Databricks provides a Unified Analytics Platform for data science teams to collaborate with data engineering and lines of business to build data products. Users achieve faster time-to-value with Databricks by creating analytic workflows that go from ETL and interactive exploration to production. The company also makes it easier for its users to focus on their data by providing a fully managed, scalable, and secure cloud infrastructure that reduces operational complexity and total cost of ownership. Databricks, venture-backed by Andreessen Horowitz and NEA, has a global customer base that includes CapitalOne, Salesforce, Viacom, Amgen, Shell and HP. For more information, visit www.databricks.com.

# Introduction

Apache® Spark™ has rapidly emerged as the de facto standard for big data processing and data sciences across all industries. The use cases range from providing recommendations based on user behavior to analyzing millions of genomic sequences to accelerate drug innovation and development for personalized medicine.

Our engineers, including the creators of Spark, continue to drive Spark development to make these transformative use cases a reality. Through the Databricks Blog, they regularly highlight new Spark releases and features, provide technical tutorials on Spark components, in addition to sharing practical implementation tools and tips.

This e-book, the first of a series, offers a collection of the most popular technical blog posts written by leading Spark contributors and members of the Spark PMC including Matei Zaharia, the creator of Spark; Reynold Xin, Spark's chief architect; Michael Armbrust, who is the architect behind Spark SQL; Xiangrui Meng and Joseph Bradley, the drivers of Spark MLlib; and Tathagata Das, the lead developer behind Spark Streaming, just to name a few.

These blog posts highlight many of the major developments designed to make Spark analytics simpler including an introduction to the Apache Spark APIs for analytics, tips and tricks to simplify unified data access, and real-world case studies of how various companies are using Spark with Databricks to transform their business. Whether you are just getting started with Spark or are already a Spark power user, this e-book will arm you with the knowledge to be successful on your next Spark project.

# Section 1:
# An Introduction to the Apache Spark APIs for Analytics

# Spark SQL: Manipulating Structured Data Using Spark

March 26, 2014 | by Michael Armbrust and Reynold Xin

Building a unified platform for big data analytics has long been the vision of Apache Spark, allowing a single program to perform ETL, MapReduce, and complex analytics. An important aspect of unification that our users have consistently requested is the ability to more easily import data stored in external sources, such as Apache Hive. Today, we are excited to announce Spark SQL, a new component recently merged into the Spark repository.

Spark SQL brings native support for SQL to Spark and streamlines the process of querying data stored both in RDDs (Spark's distributed datasets) and in external sources. Spark SQL conveniently blurs the lines between RDDs and relational tables. Unifying these powerful abstractions makes it easy for developers to intermix SQL commands querying external data with complex analytics, all within in a single application. Concretely, Spark SQL will allow developers to:

- Import relational data from Parquet files and Hive tables
- Run SQL queries over imported data and existing RDDs
- Easily write RDDs out to Hive tables or Parquet files

## Spark SQL In Action

Now, let's take a closer look at how Spark SQL gives developers the power to integrate SQL commands into applications that also take advantage of MLlib, Spark's machine learning library. Consider an application that needs to predict which users are likely candidates for a service, based on their profile. Often, such an analysis requires joining data from multiple sources. For the purposes of illustration, imagine an application with two tables:

- Users(userId INT, name String, email STRING, age INT, latitude: DOUBLE, longitude: DOUBLE, subscribed: BOOLEAN)
- Events(userId INT, action INT)

Given the data stored in in these tables, one might want to build a model that will predict which users are good targets for a new campaign, based on users that are similar.

```
// Data can easily be extracted from existing sources,
// such as Apache Hive.
val trainingDataTable = sql("""
  SELECT e.action
         u.age,
         u.latitude,
         u.logitude
  FROM Users u
  JOIN Events e
  ON u.userId = e.userId""")
```

```
// Since `sql` returns an RDD, the results of the above
// query can be easily used in MLlib
val trainingData = trainingDataTable.map { row =>
  val features = Array[Double](row(1), row(2), row(3))
  LabeledPoint(row(0), features)
}

val model =
  new LogisticRegressionWithSGD().run(trainingData)
```

Now that we have used SQL to join existing data and train a model, we can use this model to predict which users are likely targets.

```
val allCandidates = sql("""
  SELECT userId,
         age,
         latitude,
         logitude
  FROM Users
  WHERE subscribed = FALSE""")

// Results of ML algorithms can be used as tables
// in subsequent SQL statements.
case class Score(userId: Int, score: Double)
val scores = allCandidates.map { row =>
  val features = Array[Double](row(1), row(2), row(3))
  Score(row(0), model.predict(features))
}
scores.registerAsTable("Scores")
```

In this example, Spark SQL made it easy to extract and join the various datasets preparing them for the machine learning algorithm. Since the results of Spark SQL are also stored in RDDs, interfacing with other Spark libraries is trivial. Furthermore, Spark SQL allows developers to close the loop, by making it easy to manipulate and join the output of these algorithms, producing the desired final result.

To summarize, the unified Spark platform gives developers the power to choose the right tool for the right job, without having to juggle multiple systems. If you would like to see more concrete examples of using Spark SQL please check out the programming guide.

## Optimizing with Catalyst

In addition to providing new ways to interact with data, Spark SQL also brings a powerful new optimization framework called Catalyst. Using Catalyst, Spark can automatically transform SQL queries so that they execute more efficiently. The Catalyst framework allows the developers behind Spark SQL to rapidly add new optimizations, enabling us to build a faster system more quickly. In one recent example, we found an inefficiency in Hive group-bys that took an experienced developer an entire weekend and over 250 lines of code to fix; we were then able to make the same fix in Catalyst in only a few lines of code.

## Future of Shark

The natural question that arises is about the future of Shark. Shark was among the first systems that delivered up to 100X speedup over Hive. It builds on the Apache Hive codebase and achieves performance improvements by swapping out the physical execution engine part of Hive. While this approach enables Shark users to speed up their Hive queries without modification to their existing warehouses, Shark inherits the large, complicated code base from Hive that makes it hard to optimize and maintain. As Spark SQL matures, Shark will transition to using Spark SQL for query optimization and physical execution so that users can benefit from the ongoing optimization efforts within Spark SQL.

In short, we will continue to invest in Shark and make it an excellent drop-in replacement for Apache Hive. It will take advantage of the new Spark SQL component, and will provide features that complement it, such as Hive compatibility and the standalone SharkServer, which allows external tools to connect queries through JDBC/ODBC.

## What's next

Spark SQL will be included in Spark 1.0 as an alpha component. However, this is only the beginning of better support for relational data in Spark, and this post only scratches the surface of Catalyst. Look for future blog posts on the following topics:

- Generating custom bytecode to speed up expression evaluation
- Reading and writing data using other formats and systems, include Avro and HBase
- API support for using Spark SQL in Python and Java

# What's new for Spark SQL in Spark 1.3

March 24, 2015 | by Michael Armbrust

The Spark 1.3 release represents a major milestone for Spark SQL. In addition to several major features, we are very excited to announce that the project has officially graduated from Alpha, after being introduced only a little under a year ago. In this blog post we will discuss exactly what this step means for compatibility moving forward, as well as highlight some of the major features of the release.

## Graduation from Alpha

While we know many organizations (including all of Databricks' customers) have already begun using Spark SQL in production, the graduation from Alpha comes with a promise of stability for those building applications using this component. Like the rest of the Spark stack, we now promise binary compatibility for all public interfaces through the Spark 1.X release series.

Since the SQL language itself and our interaction with Apache Hive represent a very large interface, we also wanted to take this chance to articulate our vision for how the project will continue to evolve. A large number of Spark SQL users have data in Hive metastores and legacy workloads which rely on Hive QL. As a result, Hive compatibility will remain a major focus for Spark SQL moving forward

More specifically, the HiveQL interface provided by the HiveContext remains the most complete dialect of SQL that we support and we are committed to continuing to maintain compatibility with this interface. In places where our semantics differ in minor ways from Hive's (i.e. SPARK-5680), we continue to aim to provide a superset of Hive's functionality. Additionally, while we are excited about all of the new data sources that are available through the improved native Data Sources API (see more below), we will continue to support reading tables from the Hive Metastore using Hive's SerDes.

The new DataFrames API (also discussed below) is currently marked experimental. Since this is the first release of this new interface, we wanted an opportunity to get feedback from users on the API before it is set in stone. That said, we do not anticipate making any major breaking changes to DataFrames, and hope to remove the experimental tag from this part of Spark SQL in Spark 1.4. You can track progress and report any issues at SPARK-6116.

## Improved Data Sources API

The Data Sources API was another major focus for this release, and provides a single interface for loading and storing data using Spark SQL. In addition to the sources that come prepackaged with the Apache Spark distribution, this API provides an integration point for external developers to add support for custom data sources. At Databricks, we have already

contributed libraries for reading data stored in Apache Avro or CSV and we look forward to contributions from others in the community (check out spark packages for a full list of sources that are currently available).


## Unified Load/Save Interface

In this release we added a unified interface to SQLContext and DataFrame for loading and storing data using both the built-in and external data sources. These functions provide a simple way to load and store data, independent of whether you are writing in Python, Scala, Java, R or SQL. The examples below show how easy it is to both load data from Avro and convert it into parquet in different languages.

### Scala

```
val df = sqlContext.load("/home/michael/data.avro",
"com.databricks.spark.avro")
df.save("/home/michael/data.parquet", “parquet")
```

### Python

```
df = sqlContext.load("/home/michael/data.avro",
"com.databricks.spark.avro")
df.save("/home/michael/data.parquet", “parquet")
```

### Java

```
DataFrame df = sqlContext.load("/home/michael/
data.avro", "com.databricks.spark.avro")
df.save("/home/michael/data.parquet", "parquet")
```

### SQL

```
CREATE TABLE avroData
USING com.databricks.spark.avro
OPTIONS (
  path "/home/michael/data.avro"
)

CREATE TABLE parquetData
USING parquet
OPTIONS (
  path "/home/michael/data/parquet")
AS SELECT * FROM avroData
```

## Automatic Partition Discovery and Schema Migration for Parquet

Parquet has long been one of the fastest data sources supported by Spark SQL. With its columnar format, queries against parquet tables can execute quickly by avoiding the cost of reading unneeded data.

In the 1.3 release we added two major features to this source. First, organizations that store lots of data in parquet often find themselves evolving the schema over time by adding or removing columns. With this release we add a new feature that will scan the metadata for all files, merging the schemas to come up with a unified representation of the data. This functionality allows developers to read data where the schema has changed overtime, without the need to perform expensive manual conversions.

Additionally, the parquet datasource now supports auto-discovering data that has been partitioned into folders, and then prunes which folders are scanned based on predicates in queries made against this data. This optimization means that you can greatly speed up may queries simply by breaking up your data into folders. For example:

```
/data/year=2014/file.parquet
/data/year=2015/file.parquet
…

SELECT * FROM table WHERE year = 2015
```

In Spark 1.4, we plan to provide an interface that will allow other formats, such as ORC, JSON and CSV, to take advantage of this partitioning functionality.

## Persistent Data Source Tables

Another feature that has been added in Spark 1.3 is the ability to persist metadata about Spark SQL Data Source tables to the Hive metastore. These tables allow multiple users to share the metadata about where data is located in a convenient manner. Data Source tables can live alongside native Hive tables, which can also be read by Spark SQL.

## Reading from JDBC Sources

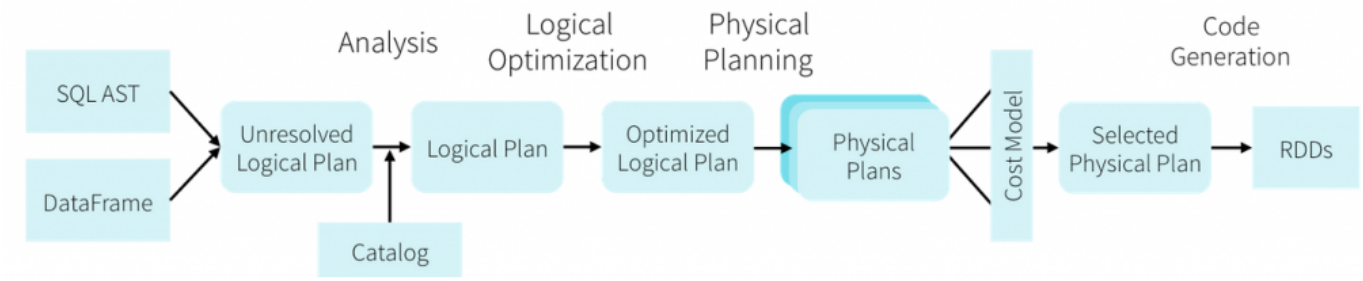
Finally, a Data Source for reading from JDBC has been added as built-in source for Spark SQL. Using this library, Spark SQL can extract data from any existing relational databases that supports JDBC. Examples include mysql, postgres, H2, and more. Reading data from one of these systems is as simple as creating a virtual table that points to the external table. Data from this table can then be easily read in and joined with any of the other sources that Spark SQL supports.

```
/data/year=2014/file.parquet
/data/year=2015/file.parquet
…

SELECT * FROM table WHERE year = 2015
```

This functionality is a great improvement over Spark's earlier support for JDBC (i.e., JdbcRDD). Unlike the pure RDD implementation, this new DataSource supports automatically pushing down predicates, converts the data into a DataFrame that can be easily joined, and is accessible from Python, Java, and SQL in addition to Scala.

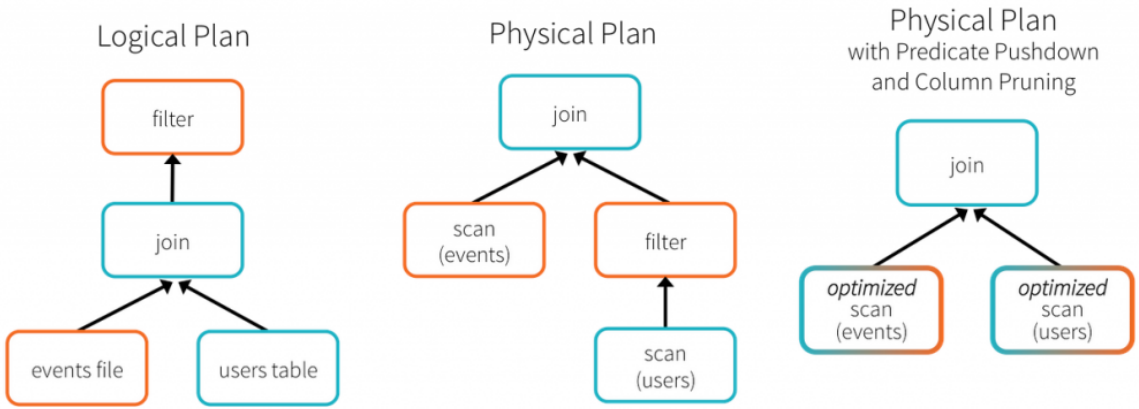
## Introducing DataFrames

While we have already talked about the DataFrames in other blog posts and talks at the Spark Summit East, any post about Spark 1.3 would be remiss if it didn't mention this important new API. DataFrames evolve Spark's RDD model, making it faster and easier for Spark developers to work with structured data by providing simplified methods for filtering, aggregating, and projecting over large datasets. Our DataFrame implementation was inspired by Pandas' and R's data frames, and are fully interoperable with these implementations. Additionally, Spark SQL DataFrames are available in Spark's Java, Scala, and Python API's as well as the upcoming (unreleased) R API.



Internally, DataFrames take advantage of the Catalyst query optimizer to intelligently plan the execution of your big data analyses. This planning permeates all the way into physical storage, where optimizations such as predicate pushdown are applied based on analysis of user programs. Since this planning is happening at the logical level, optimizations can even occur across function calls, as shown in the example to the right.

```
def add_demographics(events):
   u = sqlCtx.table("users")                    # Load partitioned Hive table
   events \
     .join(u, events.user_id == u.user_id) \    # Join on user_id
     .withColumn("city", zipToCity(df.zip))     # Run udf to add city column

events = add_demographics(sqlCtx.load("/data/events", "parquet"))
training_data = events.where(events.city == "New York").select(events.timestamp).collect()
```




In this example, Spark SQL is able to push the filtering of users by their location through the join, greatly reducing its cost to execute. This optimization is possible even though the original author of the `add_demographics` function did not provide a parameter for specifying how to filter users!

This is only example of how Spark SQL DataFrames can make developers more efficient by providing a simple interface coupled with powerful optimization.

To learn more about Spark SQL, Dataframes, or Spark 1.3, checkout the SQL programming guide on the Apache Spark website. Stay tuned to this blog for updates on other components of the Spark 1.3 release!

# Introducing Window Functions in Spark SQL

July 15, 2015 | by Yin Huai and Michael Armbrust

*Get the Notebook*

In this blog post, we introduce the new window function feature that was added in Spark 1.4. Window functions allow users of Spark SQL to calculate results such as the rank of a given row or a moving average over a range of input rows. They significantly improve the expressiveness of Spark's SQL and DataFrame APIs. This blog will first introduce the concept of window functions and then discuss how to use them with Spark SQL and Spark's DataFrame API.

## What are Window Functions?

Before 1.4, there were two kinds of functions supported by Spark SQL that could be used to calculate a single return value. *Built-in functions* or *UDFs*, such as `substr` or `round`, take values from a single row as input, and they generate a single return value for every input row. *Aggregate functions,* such as `SUM` or `MAX`, operate on a group of rows and calculate a single return value for every group.

While these are both very useful in practice, there is still a wide range of operations that cannot be expressed using these types of functions alone. Specifically, there was no way to both operate on a group of rows while still returning a single value for every input row. This limitation makes it hard to conduct various data processing tasks like calculating a moving average, calculating a cumulative sum, or accessing the values of a row appearing before the current row. Fortunately for users of Spark SQL, window functions fill this gap.

At its core, a window function calculates a return value for every input row of a table based on a group of rows, called the *Frame*. Every input row can have a unique frame associated with it. This characteristic of window functions makes them more powerful than other functions and allows users to express various data processing tasks that are hard (if not impossible) to be expressed without window functions in a concise way. Now, let's take a look at two examples.

Suppose that we have a *productRevenue* table as shown below.

productRevenue

| product | category | revenue |
|---|---|---|
| Thin | Cell phone | 6000 |
| Normal | Tablet | 1500 |
| Mini | Tablet | 5500 |
| Ultra thin | Cell phone | 5000 |
| Very thin | Cell phone | 6000 |
| Big | Tablet | 2500 |
| Bendable | Cell phone | 3000 |
| Foldable | Cell phone | 3000 |
| Pro | Tablet | 4500 |
| Pro2 | Tablet | 6500 |

We want to answer two questions:

1. What are the best-selling and the second best-selling products in every category?
2. What is the difference between the revenue of each product and the revenue of the best-selling product in the same category of that product?

To answer the first question "What are the best-selling and the second best-selling products in every category?", we need to rank products in a category based on their revenue, and to pick the best selling and the second best-selling products based the ranking. Below is the SQL query used to answer this question by using window function **dense_rank** (we will explain the syntax of using window functions in next section).

```
SELECT
  product,
  category,
  revenue
FROM (
  SELECT
    product,
    category,
    revenue,
    dense_rank() OVER (PARTITION BY category ORDER BY
revenue DESC) as rank
  FROM productRevenue) tmp
WHERE
  rank <= 2
```

The result of this query is shown below. Without using window functions, it is very hard to express the query in SQL, and even if a SQL query can be expressed, it is hard for the underlying engine to efficiently evaluate the query.

| product | category | revenue |
|---|---|---|
| Pro2 | Tablet | 6500 |
| Mini | Tablet | 5500 |
| Thin | Cell Phone | 6000 |
| Very thin | Cell Phone | 6000 |
| Ultra thin | Cell Phone | 5500 |

For the second question "*What is the difference between the revenue of each product and the revenue of the best selling product in the same category as that product?*", to calculate the revenue difference for a product, we need to find the highest revenue value from products in the same category for each product. Below is a Python DataFrame program used to answer this question.

```
import sys
from pyspark.sql.window import Window
import pyspark.sql.functions as func

windowSpec = \
  Window
    .partitionBy(df['category']) \
    .orderBy(df['revenue'].desc()) \
    .rangeBetween(-sys.maxsize, sys.maxsize)
dataFrame = sqlContext.table("productRevenue")
```

```
revenue_difference = \
  (func.max(dataFrame['revenue']).over(windowSpec) -
dataFrame['revenue'])
dataFrame.select(
  dataFrame['product'],
  dataFrame['category'],
  dataFrame['revenue'],
  revenue_difference.alias("revenue_difference"))
```

The result of this program is shown below. Without using window functions, users have to find all highest revenue values of all categories and then join this derived data set with the original productRevenue table to calculate the revenue differences.

| product | category | revenue | revenue_difference |
|---|---|---|---|
| Pro2 | Tablet | 6500 | 0 |
| Mini | Tablet | 5500 | 1000 |
| Pro | Tablet | 4500 | 2000 |
| Big | Tablet | 2500 | 4000 |
| Normal | Tablet | 1500 | 5000 |
| Thin | Cell Phone | 6000 | 0 |
| Very thin | Cell Phone | 6000 | 0 |
| Ultra thin | Cell Phone | 5500 | 500 |
| Foldable | Cell Phone | 3000 | 3000 |
| Bendable | Cell Phone | 3000 | 3000 |

## Using Window Functions

Spark SQL supports three kinds of window functions: ranking functions, analytic functions, and aggregate functions. The available ranking functions and analytic functions are summarized in the table below. For aggregate functions, users can use any existing aggregate function as a window function.

| | SQL | DataFrame API |
|---|---|---|
| **Ranking functions** | rank | rank |
| | dense_rank | denseRank |
| | percent_rank | percentRank |
| | ntile | ntile |
| | row_number | rowNumber |
| **Analytic functions** | cume_dist | cumeDist |
| | first_value | firstValue |
| | last_value | lastValue |
| | lag | lag |
| | lead | lead |

To use window functions, users need to mark that a function is used as a window function by either

- Adding an *OVER* clause after a supported function in SQL, e.g. `avg(revenue) OVER (...)`; or
- Calling the *over* method on a supported function in the DataFrame API, e.g. `rank().over(...)`.

Once a function is marked as a window function, the next key step is to define the *Window Specification* associated with this function. A window specification defines which rows are included in the frame associated with a given input row. A window specification includes three parts:

1. Partitioning Specification: controls which rows will be in the same partition with the given row. Also, the user might want to make sure all rows having the same value for the category column are collected to the same machine before ordering and calculating the frame. If no partitioning specification is given, then all data must be collected to a single machine.
2. Ordering Specification: controls the way that rows in a partition are ordered, determining the position of the given row in its partition.
3. Frame Specification: states which rows will be included in the frame for the current input row, based on their relative position to the current row. For example, "the three rows preceding the current row to the current row" describes a frame including the current input row and three rows appearing before the current row.

In SQL, the `PARTITION BY` and `ORDER BY` keywords are used to specify partitioning expressions for the partitioning specification, and ordering expressions for the ordering specification, respectively. The SQL syntax is shown below.

```
OVER (PARTITION BY ... ORDER BY …)
```

In the DataFrame API, we provide utility functions to define a window specification. Taking Python as an example, users can specify partitioning expressions and ordering expressions as follows.

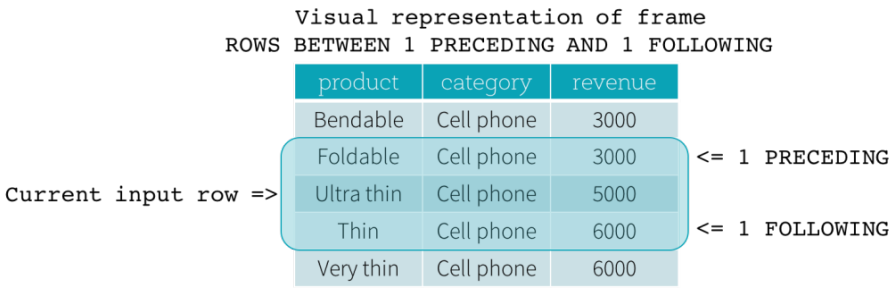
```
from pyspark.sql.window import Window

windowSpec = \
  Window \
    .partitionBy(...) \
    .orderBy(…)
```

In addition to the ordering and partitioning, users need to define the start boundary of the frame, the end boundary of the frame, and the type of the frame, which are three components of a frame specification.

There are five types of boundaries, which are `UNBOUNDED PRECEDING`, `UNBOUNDED FOLLOWING`, `CURRENT ROW`, `<value> PRECEDING`, and `<value> FOLLOWING`. `UNBOUNDED PRECEDING` and `UNBOUNDED FOLLOWING` represent the first row of the partition and the last row of the partition, respectively. For the other three types of boundaries, they specify the offset from the position of the current input row and their specific meanings are defined based on the type of the frame. There are two types of frames, *ROW* frame and *RANGE* frame.

## ROW frame

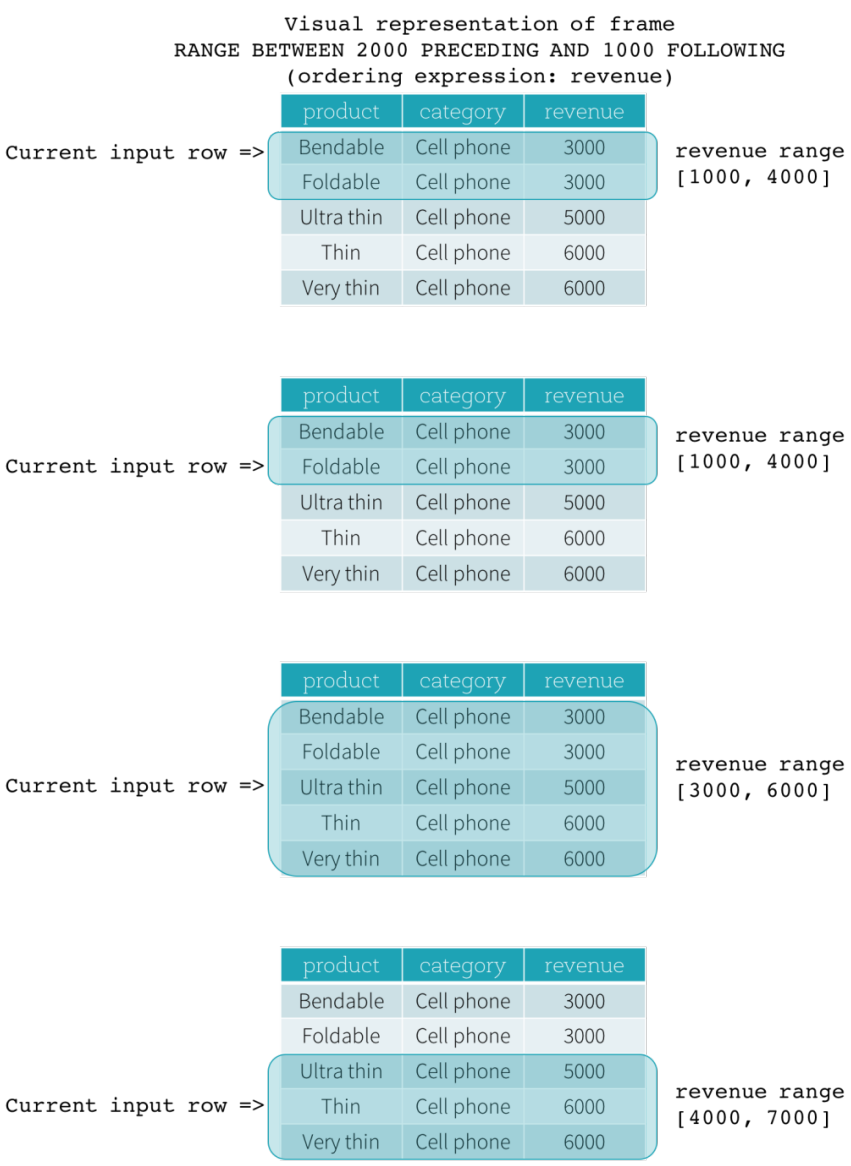
ROW frames are based on physical offsets from the position of the current input row, which means that `CURRENT ROW`, `<value> PRECEDING`, or `<value> FOLLOWING` specifies a physical offset. If `CURRENT ROW` is used as a boundary, it represents the current input row. `<value> PRECEDING` and `<value> FOLLOWING` describes the number of rows appear before and after the current input row, respectively. The following figure illustrates a ROW frame with a `1 PRECEDING` as the start boundary and `1 FOLLOWING` as the end boundary (`ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING` in the SQL syntax).

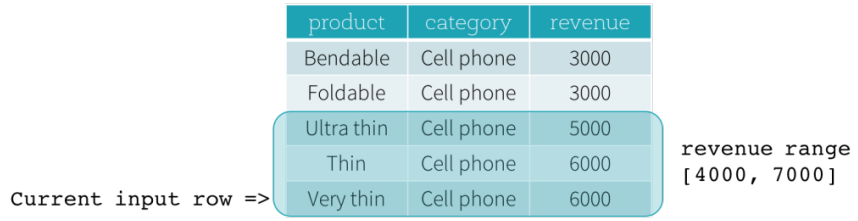



## RANGE frame

RANGE frames are based on logical offsets from the position of the current input row, and have similar syntax to the ROW frame. A logical offset is the difference between the value of the ordering expression of the current input row and the value of that same expression of the boundary row of the frame. Because of this definition, when a RANGE frame is used, only a single ordering expression is allowed. Also, for a RANGE frame, all rows having the same value of the ordering expression with the current input row are considered as same row as far as the boundary calculation is concerned.

Now, let's take a look at an example. In this example, the ordering expressions is `revenue`; the start boundary is `2000 PRECEDING;` and the end boundary is `1000 FOLLOWING` (this frame is defined as `RANGE BETWEEN 2000 PRECEDING AND 1000 FOLLOWING` in the SQL syntax). The following five figures illustrate how the frame is updated with the update of the current input row. Basically, for every current input row, based on the value of revenue, we calculate the revenue range `[current revenue value - 2000, current revenue value + 1000]`. All rows whose revenue values fall in this range are in the frame of the current input row.




In summary, to define a window specification, users can use the following syntax in SQL.

```
OVER (PARTITION BY ... ORDER BY ... frame_type BETWEEN
start AND end)
```

Here, `frame_type` can be either ROWS (for ROW frame) or RANGE (for RANGE frame); `start` can be any of `UNBOUNDED PRECEDING`, `CURRENT ROW`, `<value> PRECEDING`, and `<value> FOLLOWING`; and `end` can be any of `UNBOUNDED FOLLOWING`, `CURRENT ROW`, `<value> PRECEDING`, and `<value> FOLLOWING`.

In the Python DataFrame API, users can define a window specification as follows.

```
from pyspark.sql.window import Window

# Defines partitioning specification and ordering
specification.
windowSpec = \
  Window \
    .partitionBy(...) \
    .orderBy(...)
# Defines a Window Specification with a ROW frame.
windowSpec.rowsBetween(start, end)
# Defines a Window Specification with a RANGE frame.
windowSpec.rangeBetween(start, end)
from pyspark.sql.window import Window
```

## What's next?

Since the release of Spark 1.4, we have been actively working with community members on optimizations that improve the performance and reduce the memory consumption of the operator evaluating window functions. Some of these will be added in Spark 1.5, and others will be added in our future releases. Besides performance improvement work, there are two features that we will add in the near future to make window function support in Spark SQL even more powerful. First, we have been working on adding Interval data type support for Date and Timestamp data types (SPARK-8943). With the Interval data type, users can use intervals as values specified in `<value> PRECEDING` and `<value> FOLLOWING` for RANGE frame, which makes it much easier to do various time series analysis with window functions. Second, we have been working on adding the support for user-defined aggregate functions in Spark SQL (SPARK-3947). With our window function support, users can immediately use their user-defined aggregate functions as window functions to conduct various advanced data analysis tasks.

## Acknowledgements

The development of the window function support in Spark 1.4 is is a joint work by many members of the Spark community. In particular, we would like to thank Wei Guo for contributing the initial patch.

*Get the Notebook*

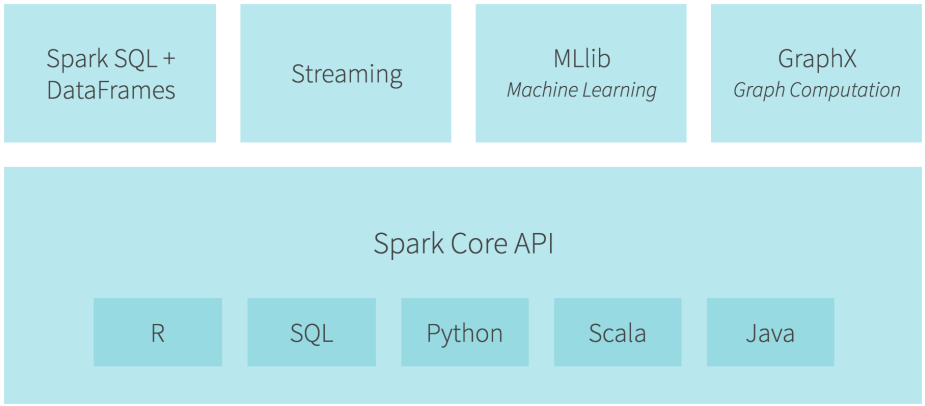
# Recent performance improvements in Apache Spark: SQL, Python, DataFrames, and More

April 24, 2015 | by Reynold Xin

In this post, we look back and cover recent performance efforts in Spark. In a follow-up blog post next week, we will look forward and share with you our thoughts on the future evolution of Spark's performance.

2014 was the most active year of Spark development to date, with major improvements across the entire engine. One particular area where it made great strides was performance: Spark set a new world record in 100TB sorting, beating the previous record held by Hadoop MapReduce by three times, using only one-tenth of the resources; it received a new SQL query engine with a state-of-the-art optimizer; and many of its built-in algorithms became five times faster.

Back in 2010, we at the AMPLab at UC Berkeley designed Spark for interactive queries and iterative algorithms, as these were two major use cases not well served by batch frameworks like MapReduce. As a result, early users were drawn to Spark because of the significant performance improvements in these workloads. However, performance optimization is a never-ending process, and as Spark's use cases have grown, so have the areas looked at for further improvement. User feedback and detailed measurements helped the Apache Spark developer community to prioritize areas to work in. Starting with the core engine, I will cover some of the recent optimizations that have been made.




*The Spark Ecosystem*

## Core engine

One unique thing about Spark is its user-facing APIs (SQL, streaming, machine learning, etc.) run over a common core execution engine. Whenever possible, specific workloads are sped up by making optimizations in the core engine. As a result, these optimizations speed up *all* components. We've often seen very surprising results this way: for example, when core developers decreased latency to introduce Spark Streaming, we also saw SQL queries become two times faster.

In the core engine, the major improvements in 2014 were in communication. First, *shuffle* is the operation that moves data point-to-point across machines. It underpins almost all workloads. For example, a SQL query joining two data sources uses shuffle to move tuples that should be joined together onto the same machine, and product recommendation algorithms such as ALS use shuffle to send user/product weights across the network.

The last two releases of Spark featured a new sort-based shuffle layer and a new network layer based on Netty with zero-copy and explicit memory management. These two make Spark more robust in very large-scale workloads. In our own experiments at Databricks, we have used this to run petabyte shuffles on 250,000 tasks. These two changes were also the key to Spark setting the current world record in large-scale sorting, beating the previous Hadoop-based record by 30 times in per-node performance.

In addition to shuffle, core developers rewrote Spark's *broadcast* primitive to use a BitTorrent-like protocol to reduce network traffic. This speeds up workloads that need to send a large parameter to multiple machines, including SQL queries and many machine learning algorithms. We have seen more than five times performance improvements for these workloads.

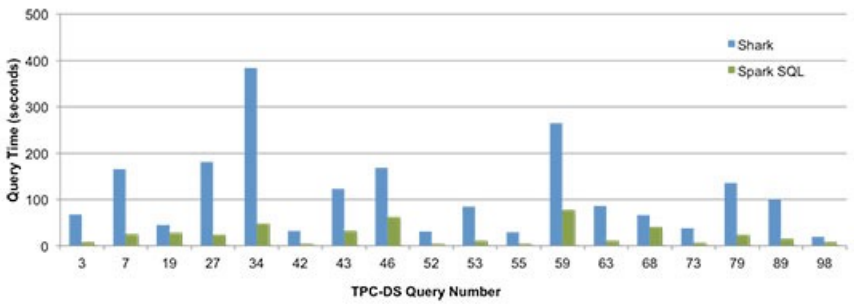
## Python API (PySpark)

Python is perhaps the most popular programming language used by data scientists. The Spark community views Python as a first-class citizen of the Spark ecosystem. When it comes to performance, Python programs historically lag behind their JVM counterparts due to the more dynamic nature of the language.

Spark's core developers have worked extensively to bridge the performance gap between JVM languages and Python. In particular, PySpark can now run on *PyPy* to leverage the just-in-time compiler, in some cases improving performance by a factor of 50. The way Python processes communicate with the main Spark JVM programs have also been redesigned to enable *worker reuse*. In addition, broadcasts are handled via a more optimized serialization framework, enabling PySpark to broadcast data larger than 2GB. The latter two have made general Python program performance two to 10 times faster.

## SQL

One year ago, Shark, an earlier SQL on Spark engine based on Hive, was deprecated and we at Databricks built a new query engine based on a new query optimizer, Catalyst, designed to run natively on Spark. It was a controversial decision, within the Apache Spark developer community as well as internally within Databricks, because building a brand new query engine necessitates astronomical engineering investments. One year later, more than 115 open source contributors have joined the project, making it one of the most active open source query engines.

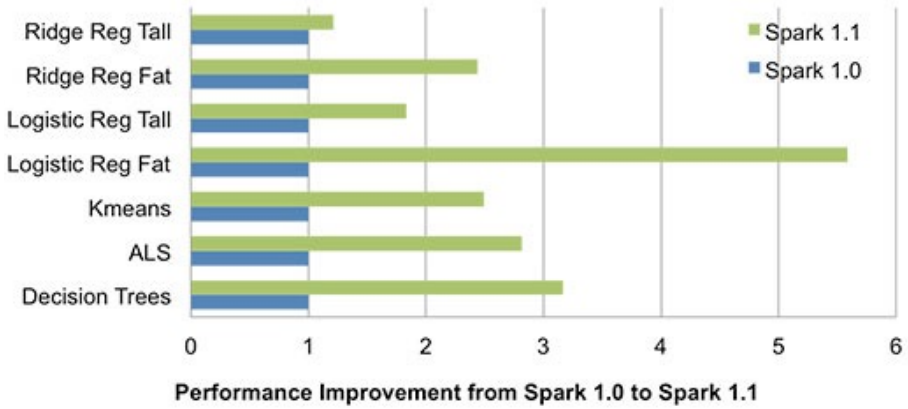



*Shark Vs. Spark Sql*

Despite being less than a year old, Spark SQL is outperforming Shark on almost all benchmarked queries. In TPC-DS, a decision-support benchmark, Spark SQL is outperforming Shark often by an order of magnitude, due to better optimizations and code generation.

## Machine learning (MLlib) and Graph Computation (GraphX)

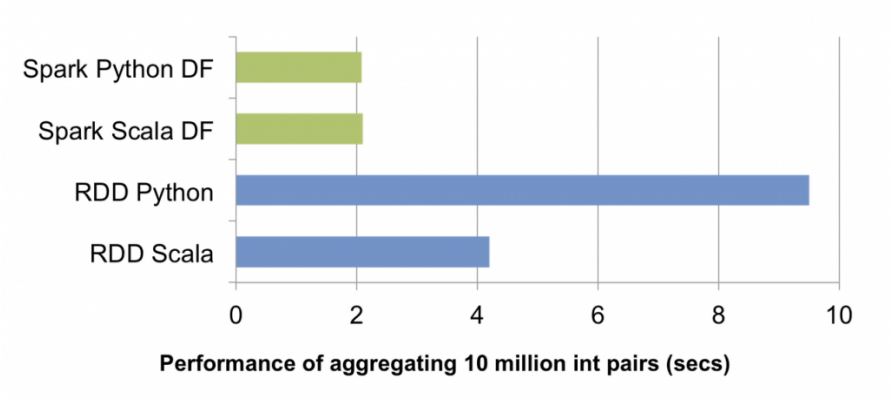
From early on, Spark was packaged with powerful standard libraries that can be optimized along with the core engine. This has allowed for a number of rich optimizations to these libraries. For instance, Spark 1.1 featured a new communication pattern for aggregating machine learning models using multi-level aggregation trees. This has reduced the model aggregation time by an order of magnitude. This new communication pattern, coupled with the more efficient broadcast implementation in core, results in speeds 1.5 to five times faster across all algorithms.




In addition to optimizations in communication, *Alternative Least Squares (ALS)*, a common collaborative filtering algorithm, was also re-implemented 1.3, which provided another factor of two speedup for ALS over what the above chart shows. In addition, all the built-in algorithms in GraphX have also seen 20% to 50% runtime performance improvements, due to a new optimized API.

## DataFrames: Leveling the Field for Python and JVM

In Spark 1.3, we introduced a new *DataFrame* API. This new API makes Spark programs more concise and easier to understand, and at the same time exposes more application semantics to the engine. As a result, Spark can use Catalyst to optimize these programs.




*Note: An earlier version of this blog post appeared on O'Reilly Radar.*

Through the new DataFrame API, Python programs can achieve the same level of performance as JVM programs because the Catalyst optimizer compiles DataFrame operations into JVM bytecode. Indeed, performance sometimes beats hand-written Scala code.

The Catalyst optimizer will also become smarter over time, picking better logical optimizations and physical execution optimizations. For example, in the future, Spark will be able to leverage schema information to create a custom physical layout of data, improving cache locality and reducing garbage collection. This will benefit both Spark SQL and DataFrame programs. As more libraries are converting to use this new DataFrame API, they will also automatically benefit from these optimizations.

The goal of Spark is to offer a single platform where users can get the best distributed algorithms for any data processing task. We will continue to push the boundaries of performance, making Spark faster and more powerful for more users.

# Introducing DataFrames in Spark for Large Scale Data Science

February 17, 2015 | by Reynold Xin, Michael Armbrust and Davies Liu

Today, we are excited to announce a new DataFrame API designed to make big data processing even easier for a wider audience.

When we first open sourced Spark, we aimed to provide a simple API for distributed data processing in general-purpose programming languages (Java, Python, Scala). Spark enabled distributed data processing through functional transformations on distributed collections of data (RDDs). This was an incredibly powerful API: tasks that used to take thousands of lines of code to express could be reduced to dozens.

As Spark continues to grow, we want to enable wider audiences beyond "Big Data" engineers to leverage the power of distributed processing. The new DataFrames API was created with this goal in mind. This API is inspired by data frames in R and Python (Pandas), but designed from the ground-up to support modern big data and data science applications. As an extension to the existing RDD API, DataFrames feature:

- Ability to scale from kilobytes of data on a single laptop to petabytes on a large cluster
- Support for a wide array of data formats and storage systems
- State-of-the-art optimization and code generation through the Spark SQL Catalyst optimizer
- Seamless integration with all big data tooling and infrastructure via Spark
- APIs for Python, Java, Scala, and R (in development via SparkR)

For new users familiar with data frames in other programming languages, this API should make them feel at home. For existing Spark users, this extended API will make Spark easier to program, and at the same time improve performance through intelligent optimizations and code-generation.

## What Are DataFrames?

In Spark, a DataFrame is a distributed collection of data organized into named columns. It is conceptually equivalent to a table in a relational database or a data frame in R/Python, but with richer optimizations under the hood. DataFrames can be constructed from a wide array of sources such as: structured data files, tables in Hive, external databases, or existing RDDs.

The following example shows how to construct DataFrames in Python. A similar API is available in Scala and Java.

```
# Constructs a DataFrame from the users table in Hive.
users = context.table("users")

# from JSON files in S3
logs = context.load("s3n://path/to/data.json", "json")
```

## How Can One Use DataFrames?

Once built, DataFrames provide a domain-specific language for distributed data manipulation. Here is an example of using DataFrames to manipulate the demographic data of a large population of users:

```
# Create a new DataFrame that contains “young users” only
young = users.filter(users.age < 21)

# Alternatively, using Pandas-like syntax
young = users[users.age < 21]

# Increment everybody’s age by 1
young.select(young.name, young.age + 1)

# Count the number of young users by gender
young.groupBy("gender").count()

# Join young users with another DataFrame called logs
young.join(logs, logs.userId == users.userId,
“left_outer")
```

You can also incorporate SQL while working with DataFrames, using Spark SQL. This example counts the number of users in the *young* DataFrame.

```
young.registerTempTable("young")
context.sql("SELECT count(*) FROM young")
```

In Python, you can also convert freely between Pandas DataFrame and Spark DataFrame:

```
# Convert Spark DataFrame to Pandas
pandas_df = young.toPandas()

# Create a Spark DataFrame from Pandas
spark_df = context.createDataFrame(pandas_df)
```

Similar to RDDs, DataFrames are evaluated lazily. That is to say, computation only happens when an action (e.g. display result, save output) is required. This allows their executions to be optimized, by applying techniques such as predicate push-downs and bytecode generation, as explained later in the section “Under the Hood: Intelligent Optimization and Code Generation”. All DataFrame operations are also automatically parallelized and distributed on clusters.

## Supported Data Formats and Sources

Modern applications often need to collect and analyze data from a variety of sources. Out of the box, DataFrame supports reading data from the

most popular formats, including JSON files, Parquet files, Hive tables. It can read from local file systems, distributed file systems (HDFS), cloud storage (S3), and external relational database systems via JDBC. In addition, through Spark SQL's external data sources API, DataFrames can be extended to support any third-party data formats or sources. Existing third-party extensions already include Avro, CSV, ElasticSearch, and Cassandra.


Formats and Sources supported by DataFrames

DataFrames' support for data sources enables applications to easily combine data from disparate sources (known as federated query processing in database systems). For example, the following code snippet joins a site's textual traffic log stored in S3 with a PostgreSQL database to count the number of times each user has visited the site.

```
users = context.jdbc("jdbc:postgresql:production",
"users")
logs = context.load("/path/to/traffic.log")
logs.join(users, logs.userId == users.userId,
"left_outer") \
  .groupBy("userId").agg({"*": "count"})
```

## Application: Advanced Analytics and Machine Learning

Data scientists are employing increasingly sophisticated techniques that go beyond joins and aggregations. To support this, DataFrames can be used directly in MLlib's machine learning pipeline API. In addition, programs can run arbitrarily complex user functions on DataFrames.

Most common advanced analytics tasks can be specified using the new pipeline API in MLlib. For example, the following code creates a simple text classification pipeline consisting of a tokenizer, a hashing term frequency feature extractor, and logistic regression.

```
tokenizer = Tokenizer(inputCol="text", outputCol="words")
hashingTF = HashingTF(inputCol="words",
outputCol="features")
lr = LogisticRegression(maxIter=10, regParam=0.01)
pipeline = Pipeline(stages=[tokenizer, hashingTF, lr])
```

Once the pipeline is setup, we can use it to train on a DataFrame directly:

```
df = context.load("/path/to/data")
model = pipeline.fit(df)
```

For more complicated tasks beyond what the machine learning pipeline API provides, applications can also apply arbitrarily complex functions on a DataFrame, which can also be manipulated using Spark's existing RDD

API. The following snippet performs a word count, the “hello world” of big data, on the “bio” column of a DataFrame.

```
df = context.load("/path/to/people.json")
# RDD-style methods such as map, flatMap are available on
DataFrames
# Split the bio text into multiple words.
words = df.select("bio").flatMap(lambda row:
row.bio.split(" "))
# Create a new DataFrame to count the number of words
words_df = words.map(lambda w: Row(word=w, cnt=1)).toDF()
word_counts = words_df.groupBy(“word").sum()
```

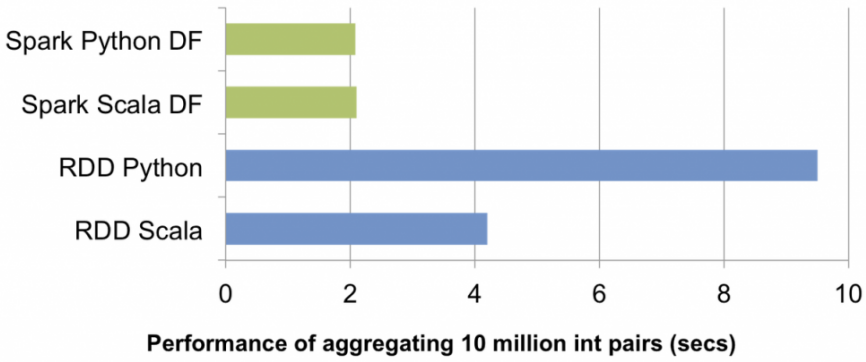
## Under the Hood: Intelligent Optimization and Code Generation

Unlike the eagerly evaluated data frames in R and Python, DataFrames in Spark have their execution automatically optimized by a query optimizer. Before any computation on a DataFrame starts, the Catalyst optimizer compiles the operations that were used to build the DataFrame into a physical plan for execution. Because the optimizer understands the semantics of operations and structure of the data, it can make intelligent decisions to speed up computation.

At a high level, there are two kinds of optimizations. First, Catalyst applies logical optimizations such as predicate pushdown. The optimizer can push filter predicates down into the data source, enabling the physical execution to skip irrelevant data. In the case of Parquet files, entire blocks can be skipped and comparisons on strings can be turned into cheaper integer comparisons via dictionary encoding. In the case of relational databases, predicates are pushed down into the external databases to reduce the amount of data traffic.

Second, Catalyst compiles operations into physical plans for execution and generates JVM bytecode for those plans that is often more optimized than hand-written code. For example, it can choose intelligently between broadcast joins and shuffle joins to reduce network traffic. It can also perform lower level optimizations such as eliminating expensive object allocations and reducing virtual function calls. As a result, we expect performance improvements for existing Spark programs when they migrate to DataFrames.

Since the optimizer generates JVM bytecode for execution, Python users will experience the same high performance as Scala and Java users.



The above chart compares the runtime performance of running group-by-aggregation on 10 million integer pairs on a single machine (source code). Since both Scala and Python DataFrame operations are compiled into JVM bytecode for execution, there is little difference between the two languages, and both outperform the vanilla Python RDD variant by a factor of 5 and Scala RDD variant by a factor of 2.

DataFrames were inspired by previous distributed data frame efforts, including Adatao's DDF and Ayasdi's BigDF. However, the main difference from these projects is that DataFrames go through the Catalyst optimizer, enabling optimized execution similar to that of Spark SQL queries. As we improve the Catalyst optimizer, the engine also becomes smarter, making applications faster with each new release of Spark.

Our data science team at Databricks has been using this new DataFrame API on our internal data pipelines. It has brought performance improvements to our Spark programs while making them more concise and easier to understand. We are very excited about it and believe it will make big data processing more accessible to a wider array of users.

This API will be released as part of Spark 1.3 in early March. If you can't wait, check out Spark from GitHub to try it out.

This effort would not have been possible without the prior data frame implementations, and thus we would like to thank the developers of R, Pandas, DDF and BigDF for their work.

# Statistical and Mathematical Functions with DataFrames in Spark

June 2, 2015 | by Burak Yavuz and Reynold Xin

*Get the Notebook*

We introduced DataFrames in Spark 1.3 to make Apache Spark much easier to use. Inspired by data frames in R and Python, DataFrames in Spark expose an API that's similar to the single-node data tools that data scientists are already familiar with. Statistics is an important part of everyday data science. We are happy to announce improved support for statistical and mathematical functions in the upcoming 1.4 release.

In this blog post, we walk through some of the important functions, including:

1. Random data generation
2. Summary and descriptive statistics
3. Sample covariance and correlation
4. Cross tabulation (a.k.a. contingency table)
5. Frequent items
6. Mathematical functions

We use Python in our examples. However, similar APIs exist for Scala and Java users as well.

## 1. Random Data Generation

Random data generation is useful for testing of existing algorithms and implementing randomized algorithms, such as random projection. We provide methods under sql.functions for generating columns that contains i.i.d. (independently and identically distributed) values drawn from a distribution, e.g., uniform (**rand**), and standard normal (**randn**).

```
In [1]: from pyspark.sql.functions import rand, randn
In [2]: # Create a DataFrame with one int column and 10
rows.
In [3]: df = sqlContext.range(0, 10)
In [4]: df.show()
+--+
|id|
+--+
| 0|
| 1|
| 2|
| 3|
| 4|
| 5|
| 6|
| 7|
| 8|
| 9|
+--+
```

```
In [4]: # Generate two other columns using uniform
distribution and normal distribution.
In [5]: df.select("id", rand(seed=10).alias("uniform"),
randn(seed=27).alias("normal")).show()
+--+------------------+--------------------+
|id|           uniform|              normal|
+--+------------------+--------------------+
| 0| 0.7224977951905031| -0.1875348803463305|
| 1| 0.2953174992603351|-0.26525647952450265|
| 2| 0.4536856090041318| -0.7195024130068081|
| 3| 0.9970412477032209|  0.5181478766595276|
| 4|0.19657711634539565|  0.7316273979766378|
| 5|0.48533720635534006| 0.07724879367590629|
| 6| 0.7369825278894753| -0.5462256961278941|
| 7| 0.5241113627472694| -0.2542275002421211|
| 8| 0.2977697066654349| -0.5752237580095868|
| 9| 0.5060159582230856|  1.0900096472044518|
+--+------------------+--------------------+
```

## 2. Summary and Descriptive Statistics

The first operation to perform after importing data is to get some sense of what it looks like. For numerical columns, knowing the descriptive summary statistics can help a lot in understanding the distribution of your data. The function `describe` returns a DataFrame containing information such as number of non-null entries (count), mean, standard deviation, and minimum and maximum value for each numerical column.

```
In [1]: from pyspark.sql.functions import rand, randn
In [2]: # A slightly different way to generate the two
random columns
In [3]: df = sqlContext.range(0, 10).withColumn('uniform',
rand(seed=10)).withColumn('normal', randn(seed=27))
In [4]: df.describe().show()
+-------+-----------------+------------------+--------------------+
|summary|               id|           uniform|              normal|
+-------+-----------------+------------------+--------------------+
|  count|               10|                10|                  10|
|   mean|              4.5| 0.5215336029384192|-0.01309370117407197|
| stddev|2.8722813232690143|  0.229328162820653|  0.5756058014772729|
|    min|                0|0.19657711634539565| -0.7195024130068081|
|    max|                9| 0.9970412477032209|  1.0900096472044518|
+-------+-----------------+------------------+--------------------+
```

If you have a DataFrame with a large number of columns, you can also run describe on a subset of the columns:

```
In [4]: df.describe('uniform', 'normal').show()
+-------+------------------+--------------------+
|summary|           uniform|              normal|
+-------+------------------+--------------------+
|  count|                10|                  10|
|   mean|0.5215336029384192|-0.01309370117407197|
| stddev| 0.229328162820653|  0.5756058014772729|
|    min|0.19657711634539565| -0.7195024130068081|
|    max|0.9970412477032209|  1.0900096472044518|
+-------+------------------+--------------------+
```

Of course, while describe works well for quick exploratory data analysis, you can also control the list of descriptive statistics and the columns they apply to using the normal select on a DataFrame:

```
In [5]: from pyspark.sql.functions import mean, min, max
In [6]: df.select([mean('uniform'), min('uniform'), max('uniform')]).show()
+-----------------+-------------------+-----------------+
|      AVG(uniform)|       MIN(uniform)|      MAX(uniform)|
+-----------------+-------------------+-----------------+
|0.5215336029384192|0.19657711634539565|0.9970412477032209|
+-----------------+-------------------+-----------------+
```

## 3. Sample covariance and correlation

Covariance is a measure of how two variables change with respect to each other. A positive number would mean that there is a tendency that as one variable increases, the other increases as well. A negative number would mean that as one variable increases, the other variable has a tendency to decrease. The sample covariance of two columns of a DataFrame can be calculated as follows:

```
In [1]: from pyspark.sql.functions import rand
In [2]: df = sqlContext.range(0, 10).withColumn('rand1',
rand(seed=10)).withColumn('rand2', rand(seed=27))

In [3]: df.stat.cov('rand1', 'rand2')
Out[3]: 0.009908130446217347

In [4]: df.stat.cov('id', 'id')
Out[4]: 9.166666666666666
```

As you can see from the above, the covariance of the two randomly generated columns is close to zero, while the covariance of the id column with itself is very high.

The covariance value of 9.17 might be hard to interpret. Correlation is a normalized measure of covariance that is easier to understand, as it provides quantitative measurements of the statistical dependence between two random variables.

```
In [5]: df.stat.corr('rand1', 'rand2')
Out[5]: 0.14938694513735398

In [6]: df.stat.corr('id', 'id')
Out[6]: 1.0
```

In the above example, id correlates perfectly with itself, while the two randomly generated columns have low correlation value.

## 4. Cross Tabulation (Contingency Table)

Cross Tabulation provides a table of the frequency distribution for a set of variables. Cross-tabulation is a powerful tool in statistics that is used to observe the statistical significance (or independence) of variables. In Spark 1.4, users will be able to cross-tabulate two columns of a DataFrame in order to obtain the counts of the different pairs that are observed in those columns. Here is an example on how to use crosstab to obtain the contingency table.

```
In [1]: # Create a DataFrame with two columns (name, item)
In [2]: names = ["Alice", "Bob", "Mike"]
In [3]: items = ["milk", "bread", "butter", "apples",
"oranges"]
In [4]: df = sqlContext.createDataFrame([(names[i % 3],
items[i % 5]) for i in range(100)], ["name", "item"])

In [5]: # Take a look at the first 10 rows.
In [6]: df.show(10)
+-----+-------+
| name|   item|
+-----+-------+
|Alice|   milk|
|  Bob|  bread|
| Mike| butter|
|Alice| apples|
|  Bob|oranges|
| Mike|   milk|
|Alice|  bread|
|  Bob| butter|
| Mike| apples|
|Alice|oranges|
+-----+-------+

In [7]: df.stat.crosstab("name", "item").show()
+---------+----+-----+------+------+-------+
|name_item|milk|bread|apples|butter|oranges|
+---------+----+-----+------+------+-------+
|      Bob|   6|    7|     7|     6|      7|
|     Mike|   7|    6|     7|     7|      6|
|    Alice|   7|    7|     6|     7|      7|
+---------+----+-----+------+------+-------+
```

One important thing to keep in mind is that the cardinality of columns we run crosstab on cannot be too big. That is to say, the number of distinct "name" and "item" cannot be too large. Just imagine if "item" contains 1 billion distinct entries: how would you fit that table on your screen?!

## 5. Frequent Items

Figuring out which items are frequent in each column can be very useful to understand a dataset. In Spark 1.4, users will be able to find the frequent items for a set of columns using DataFrames. We have implemented an one-pass algorithm proposed by Karp et al. This is a fast, approximate algorithm that always return all the frequent items that appear in a user-specified minimum proportion of rows. Note that the result might contain false positives, i.e. items that are not frequent.

```
In [1]: df = sqlContext.createDataFrame([(1, 2, 3) if i % 2 == 0
else (i, 2 * i, i % 4) for i in range(100)], ["a", "b", "c"])

In [2]: df.show(10)
+-+--+-+
|a| b|c|
+-+--+-+
|1| 2|3|
|1| 2|1|
|1| 2|3|
|3| 6|3|
|1| 2|3|
|5|10|1|
|1| 2|3|
```

```
|7|14|3|
|1| 2|3|
|9|18|1|
+-+--+-+

In [3]: freq = df.stat.freqItems(["a", "b", "c"], 0.4)
```

Given the above DataFrame, the following code finds the frequent items that show up 40% of the time for each column:

```
In [4]: freq.collect()[0]
Out[4]: Row(a_freqItems=[11, 1], b_freqItems=[2, 22],
c_freqItems=[1, 3])
```

As you can see, “11” and “1” are the frequent values for column “a”. You can also find frequent items for column combinations, by creating a composite column using the struct function:

```
In [5]: from pyspark.sql.functions import struct

In [6]: freq = df.withColumn('ab', struct('a',
'b')).stat.freqItems(['ab'], 0.4)

In [7]: freq.collect()[0]
Out[7]: Row(ab_freqItems=[Row(a=11, b=22), Row(a=1, b=2)])
```

From the above example, the combination of “a=11 and b=22”, and “a=1 and b=2” appear frequently in this dataset. Note that “a=11 and b=22” is a false positive.

## 6. Mathematical Functions

Spark 1.4 also added a suite of mathematical functions. Users can apply these to their columns with ease. The list of math functions that are supported come from this file (we will also post pre-built documentation once 1.4 is released). The inputs need to be columns functions that take a single argument, such as `cos`, `sin`, `floor`, `ceil`. For functions that take two arguments as input, such as `pow`, `hypot`, either two columns or a combination of a double and column can be supplied.

```
In [1]: from pyspark.sql.functions import *
In [2]: df = sqlContext.range(0, 10).withColumn('uniform',
rand(seed=10) * 3.14)

In [3]: # you can reference a column or supply the column
name
In [4]: df.select(
   ...:   'uniform',
   ...:   toDegrees('uniform'),
   ...:   (pow(cos(df['uniform']), 2) + pow(sin(df.uniform),
2)). \
   ...:     alias("cos^2 + sin^2")).show()
```

```
+-------------------+-----------------+------------------+
|            uniform|  DEGREES(uniform)|     cos^2 + sin^2|
+-------------------+-----------------+------------------+
|  0.7224977951905031| 41.39607437192317|               1.0|
|  0.3312021111290707|18.976483133518624|0.9999999999999999|
|  0.2953174992603351|16.920446323975014|               1.0|
|0.018326130186194667| 1.050009914476252|0.9999999999999999|
|  0.3163135293051941|18.123430232075304|               1.0|
|  0.4536856090041318| 25.99427062175921|               1.0|
|   0.873869321369476| 50.06902396043238|0.9999999999999999|
|  0.9970412477032209| 57.12625549385224|               1.0|
| 0.19657711634539565| 11.26303911544332|1.0000000000000002|
|  0.9632338825504894| 55.18923615414307|               1.0|
+-------------------+-----------------+------------------+
```

## What's Next?

All the features described in this blog post will be available in Spark 1.4 for Python, Scala, and Java, to be released in the next few days.

Statistics support will continue to increase for DataFrames through better integration with Spark MLlib in future releases. Leveraging the existing Statistics package in MLlib, support for feature selection in pipelines, Spearman Correlation, ranking, and aggregate functions for covariance and correlation.

At the end of the blog post, we would also like to thank Davies Liu, Adrian Wang, and rest of the Spark community for implementing these functions.

*Get the Notebook*

# Spark 1.5 DataFrame API Highlights: Date/Time/String Handling, Time Intervals, and UDAFs

September 16, 2015 | by Michael Armbrust, Yin Huai, Davies Liu and Reynold Xin

*Get the Notebook*

A few days ago, we announced the release of Spark 1.5. This release contains major under-the-hood changes that improve Spark's performance, usability, and operational stability. Besides these changes, we have been continuously improving DataFrame API. In this blog post, we'd like to highlight three major improvements to DataFrame API in Spark 1.5, which are:

- New built-in functions;
- Time intervals; and
- Experimental user-defined aggregation function (UDAF) interface.

## New Built-in Functions in Spark 1.5

In Spark 1.5, we have added a comprehensive list of built-in functions to the DataFrame API, complete with optimized code generation for execution. This code generation allows pipelines that call functions to take full advantage of the efficiency changes made as part of Project Tungsten. With these new additions, Spark SQL now supports a wide range of built-in functions for various use cases, including:

| Category | Functions |
|---|---|
| Aggregate Functions | approxCountDistinct, avg, count, countDistinct, first, last, max, mean, min, sum, sumDistinct |
| Collection Functions | array_contains, explode, size, sort_array |
| Date/time Functions | **Date/timestamp conversion:** unix_timestamp, from_unixtime, to_date, quarter, day, dayofyear, weekofyear, from_utc_timestamp, to_utc_timestamp |
| | **Extracting fields from a date/timestamp value:** year, month, dayofmonth, hour, minute, second |
| | **Date/timestamp calculation:** datediff, date_add, date_sub, add_months, last_day, next_day, months_between |
| | **Miscellaneous:** current_date, current_timestamp, trunc, date_format |

| | |
|---|---|
| Math Functions | abs, acros, asin, atan, atan2, bin, cbrt, ceil, conv, cos, sosh, exp, expm1, factorial, floor, hex, hypot, log, log10, log1p, log2, pmod, pow, rint, round, shiftLeft, shiftRight, shiftRightUnsigned, signum, sin, sinh, sqrt, tan, tanh, toDegrees, toRadians, unhex |
| Miscellaneous Functions | array, bitwiseNOT, callUDF, coalesce, crc32, greatest, if, inputFileName, isNaN, isnotnull, isnull, least, lit, md5, monotonicallyIncreasingId, nanvl, negate, not, rand, randn, sha, sha1, sparkPartitionId, struct, when |
| String Functions | ascii, base64, concat, concat_ws, decode, encode, format_number, format_string, get_json_object, initcap, instr, length, levenshtein, locate, lower, lpad, ltrim, printf, regexp_extract, regexp_replace, repeat, reverse, rpad, rtrim, soundex, space, split, substring, substring_index, translate, trim, unbase64, upper |
| Window Functions (in addition to Aggregate Functions) | cumeDist, denseRank, lag, lead, ntile, percentRank, rank, rowNumber |

For all available built-in functions, please refer to our API docs (Scala Doc, Java Doc, and Python Doc).

Unlike normal functions, which execute immediately and return a result, DataFrame functions return a `Column`, that will be evaluated inside of a parallel job. These columns can be used inside of DataFrame operations, such as `select`, `filter`, `groupBy`, etc. The input to a function can either be another Column (i.e. `df['columnName']`) or a literal value (i.e. a constant value). To make this more concrete, let's look at the syntax for calling the `round` function in Python.

`round` is a function that rounds a numeric value to the specified precision. When the given precision is a positive number, a given input numeric value is rounded to the decimal position specified by the precision. When the specified precision is a zero or a negative number, a given input numeric value is rounded to the position of the integral part specified by the precision.

```
# Create a simple DataFrame
data = [
  (234.5, "row1"),
  (23.45, "row2"),
  (2.345, "row3"),
  (0.2345, "row4")]
df = sqlContext.createDataFrame(data, ["i", "j"])

# Import functions provided by Spark's DataFrame API
from pyspark.sql.functions import *

# Call round function directly
df.select(
  round(df['i'], 1),
  round(df['i'], 0),
  round(df['i'], -1)).show()
```

```
+----------+----------+-----------+
|round(i,1)|round(i,0)|round(i,-1)|
+----------+----------+-----------+
|     234.5|     235.0|      230.0|
|      23.5|      23.0|       20.0|
|       2.3|       2.0|        0.0|
|       0.2|       0.0|        0.0|
+----------+----------+-----------+
```

Alternatively, all of the added functions are also available from SQL using standard syntax:

```
SELECT round(i, 1) FROM dataFrame
```

Finally, you can even mix and match SQL syntax with DataFrame operations by using the `expr` function. By using `expr`, you can construct a DataFrame column expression from a SQL expression String.

```
df.select(
  expr("round(i, 1) AS rounded1"),
  expr("round(i, 0) AS rounded2"),
  expr("round(i, -1) AS rounded3")).show()
```

## Time Interval Literals

In the last section, we introduced several new date and time functions that were added in Spark 1.5 (e.g. `datediff`, `date_add`, `date_sub`), but that is not the only new feature that will help users dealing with date or timestamp values. Another related feature is a new data type, interval, that allows developers to represent fixed periods of time (i.e. 1 day or 2 months) as interval literals. Using interval literals, it is possible to perform subtraction or addition of an arbitrary amount of time from a date or timestamp value. This representation can be useful when you want to add or subtract a time period from a fixed point in time. For example, users can now easily express queries like *"Find all transactions that have happened during the past hour"*.

An interval literal is constructed using the following syntax:

```
INTERVAL value unit
```

Breaking the above expression down, all time intervals start with the `INTERVAL` keyword. Next, the value and unit together specify the time difference. Available units are `YEAR`, `MONTH`, `DAY`, `HOUR`, `MINUTE`, `SECOND`, `MILLISECOND`, and `MICROSECOND`. For example, the following interval literal represents 3 years.

```
INTERVAL 3 YEAR
```

In addition to specifying an interval literal with a single unit, users can also combine different units. For example, the following interval literal represents a 3-year and 3-hour time difference.

```
INTERVAL 3 YEAR 3 HOUR
```

In the DataFrame API, the `expr` function can be used to create a `Column` representing an interval. The following code in Python is an example of using an interval literal to select records where `start_time` and `end_time` are in the same day and they differ by less than an hour.

```
# Import functions.
from pyspark.sql.functions import *

# Create a simple DataFrame.
data = [
  ("2015-01-01 23:59:59", "2015-01-02 00:01:02", 1),
  ("2015-01-02 23:00:00", "2015-01-02 23:59:59", 2),
  ("2015-01-02 22:59:58", "2015-01-02 23:59:59", 3)]
df = sqlContext.createDataFrame(data, ["start_time",
"end_time", "id"])
df = df.select(
  df.start_time.cast("timestamp").alias("start_time"),
  df.end_time.cast("timestamp").alias("end_time"),
  df.id)

# Get all records that have a start_time and end_time in the
# same day, and the difference between the end_time and
start_time
# is less or equal to 1 hour.
condition = \
  (to_date(df.start_time) == to_date(df.end_time)) & \
  (df.start_time + expr("INTERVAL 1 HOUR") >= df.end_time)

df.filter(condition).show()
+---------------------+---------------------+---+
|start_time           |            end_time |id |
+---------------------+---------------------+---+
|2015-01-02 23:00:00.0|2015-01-02 23:59:59.0|2  |
+---------------------+---------------------+---+
```

## User-defined Aggregate Function Interface

For power users, Spark 1.5 introduces an experimental API for user-defined aggregate functions (UDAFs). These UDAFs can be used to compute custom calculations over groups of input data (in contrast, UDFs compute a value looking at a single input row), such as calculating geometric mean or calculating the product of values for every group.

A UDAF maintains an aggregation buffer to store intermediate results for every group of input data. It updates this buffer for every input row. Once it has processed all input rows, it generates a result value based on values of the aggregation buffer.

An UDAF inherits the base class `UserDefinedAggregateFunction` and implements the following eight methods, which are:

- `inputSchema: inputSchema` returns a `StructType` and every field of this StructType represents an input argument of this UDAF.
- `bufferSchema: bufferSchema` returns a `StructType` and every field of this StructType represents a field of this UDAF's intermediate results.
- `dataType: dataType` returns a `DataType` representing the data type of this UDAF's returned value.
- `deterministic: deterministic` returns a boolean indicating if this UDAF always generate the same result for a given set of input values.

- `initialize: initialize` is used to initialize values of an aggregation buffer, represented by a `MutableAggregationBuffer`.
- `update: update` is used to update an aggregation buffer represented by a `MutableAggregationBuffer` for an input `Row`.
- `merge: merge` is used to merge two aggregation buffers and store the result to a `MutableAggregationBuffer`.
- `evaluate: evaluate` is used to generate the final result value of this UDAF based on values stored in an aggregation buffer represented by a `Row`.

Below is an example UDAF implemented in Scala that calculates the geometric mean of the given set of double values. The geometric mean can be used as an indicator of the typical value of an input set of numbers by using the product of their values (as opposed to the standard builtin mean which is based on the sum of the input values). For the purpose of simplicity, null handling logic is not shown in the following code.

```
import
org.apache.spark.sql.expressions.MutableAggregationBuffer
import
org.apache.spark.sql.expressions.UserDefinedAggregateFunction
import org.apache.spark.sql.Row
import org.apache.spark.sql.types._

class GeometricMean extends UserDefinedAggregateFunction {
  def inputSchema: org.apache.spark.sql.types.StructType =
    StructType(StructField("value", DoubleType) :: Nil)

  def bufferSchema: StructType = StructType(
    StructField("count", LongType) ::
    StructField("product", DoubleType) :: Nil
  )

  def dataType: DataType = DoubleType

  def deterministic: Boolean = true

  def initialize(buffer: MutableAggregationBuffer): Unit = {
    buffer(0) = 0L
    buffer(1) = 1.0
  }

  def update(buffer: MutableAggregationBuffer,input: Row):
Unit = {
    buffer(0) = buffer.getAs[Long](0) + 1
    buffer(1) = buffer.getAs[Double](1) *
input.getAs[Double](0)
  }

  def merge(buffer1: MutableAggregationBuffer, buffer2:
Row): Unit = {
    buffer1(0) = buffer1.getAs[Long](0) +
buffer2.getAs[Long](0)
    buffer1(1) = buffer1.getAs[Double](1) *
buffer2.getAs[Double](1)
  }

  def evaluate(buffer: Row): Any = {
    math.pow(buffer.getDouble(1), 1.toDouble /
buffer.getLong(0))
  }
}
```

A UDAF can be used in two ways. First, an instance of a UDAF can be used immediately as a function. Second, users can register a UDAF to Spark SQL's function registry and call this UDAF by the assigned name. The example code is shown below.

```
import org.apache.spark.sql.functions._
// Create a simple DataFrame with a single column called
"id"
// containing number 1 to 10.
val df = sqlContext.range(1, 11)

// Create an instance of UDAF GeometricMean.
val gm = new GeometricMean

// Show the geometric mean of values of column "id".
df.groupBy().agg(gm(col("id")).as("GeometricMean")).show()

// Register the UDAF and call it "gm".
sqlContext.udf.register("gm", gm)
// Invoke the UDAF by its assigned name.
df.groupBy().agg(expr("gm(id) as GeometricMean")).show()
```

## Summary

In this blog post, we introduced three major additions to DataFrame APIs, a set of built-in functions, time interval literals, and user-defined aggregation function interface. With new built-in functions, it is easier to manipulate string data and data/timestamp data, and to apply math operations. If your existing programs use any user-defined functions that do the same work with these built-in functions, we strongly recommend you to migrate your code to these new built-in functions to take full advantage of the efficiency changes made as part of Project Tungsten. Combining date/time functions and interval literals, it is much easier to work with date/timestamp data and to calculate date/timestamp values for various use cases. With user-defined aggregate function, users can apply custom aggregations over groups of input data in the DataFrame API.

## Acknowledgements

The development of features highlighted in this blog post has been a community effort. In particular, we would like to thank the following contributors: Adrian Wang, Tarek Auel, Yijie Shen, Liang-Chi Hsieh, Zhichao Li, Pedro Rodriguez, Cheng Hao, Shilei Qian, Nathan Howell, and Wenchen Fan.

*Get the Notebook*

# Introducing Spark Datasets

January 4, 2016 | by Michael Armbrust, Wenchen Fan, Reynold Xin and Matei Zaharia

Developers have always loved Apache Spark for providing APIs that are simple yet powerful, a combination of traits that makes complex analysis possible with minimal programmer effort. At Databricks, we have continued to push Spark's usability and performance envelope through the introduction of DataFrames and Spark SQL. These are high-level APIs for working with structured data (e.g. database tables, JSON files), which let Spark automatically optimize both storage and computation. Behind these APIs, the Catalyst optimizer and Tungsten execution engine optimize applications in ways that were not possible with Spark's object-oriented (RDD) API, such as operating on data in a raw binary form.

Today we're excited to announce Spark Datasets, an extension of the DataFrame API that provides a *type-safe, object-oriented programming interface*. Spark 1.6 includes an API preview of Datasets, and they will be a development focus for the next several versions of Spark. Like DataFrames, Datasets take advantage of Spark's Catalyst optimizer by exposing expressions and data fields to a query planner. Datasets also leverage Tungsten's fast in-memory encoding. Datasets extend these benefits with compile-time type safety – meaning production applications can be checked for errors before they are run. They also allow direct operations over user-defined classes.

In the long run, we expect Datasets to become a powerful way to write more efficient Spark applications. We have designed them to work alongside the existing RDD API, but improve efficiency when data can be represented in a structured form. Spark 1.6 offers the first glimpse at Datasets, and we expect to improve them in future releases.

## Working with Datasets

A Dataset is a strongly-typed, immutable collection of objects that are mapped to a relational schema. At the core of the Dataset API is a new concept called an encoder, which is responsible for converting between JVM objects and tabular representation. The tabular representation is stored using Spark's internal Tungsten binary format, allowing for operations on serialized data and improved memory utilization. Spark 1.6 comes with support for automatically generating encoders for a wide variety of types, including primitive types (e.g. String, Integer, Long), Scala case classes, and Java Beans.

Users of RDDs will find the Dataset API quite familiar, as it provides many of the same functional transformations (e.g. map, flatMap, filter). Consider the following code, which reads lines of a text file and splits them into words:

### RDDs

```
val lines = sc.textFile("/wikipedia")
val words = lines
  .flatMap(_.split(" "))
  .filter(_ != “”)
```

### Datasets

```
val lines = sqlContext.read.text("/wikipedia").as[String]
val words = lines
  .flatMap(_.split(" "))
  .filter(_ != "")
```

Both APIs make it easy to express the transformation using lambda functions. The compiler and your IDE understand the types being used, and can provide helpful tips and error messages while you construct your data pipeline.

While this high-level code may look similar syntactically, with Datasets you also have access to all the power of a full relational execution engine. For example, if you now want to perform an aggregation (such as counting the number of occurrences of each word), that operation can be expressed simply and efficiently as follows:

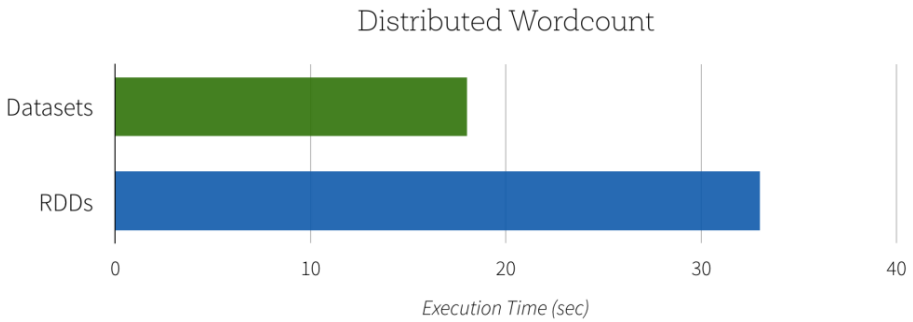
### RDDs

```
val counts = words
    .groupBy(_.toLowerCase)
    .map(w => (w._1, w._2.size))
```

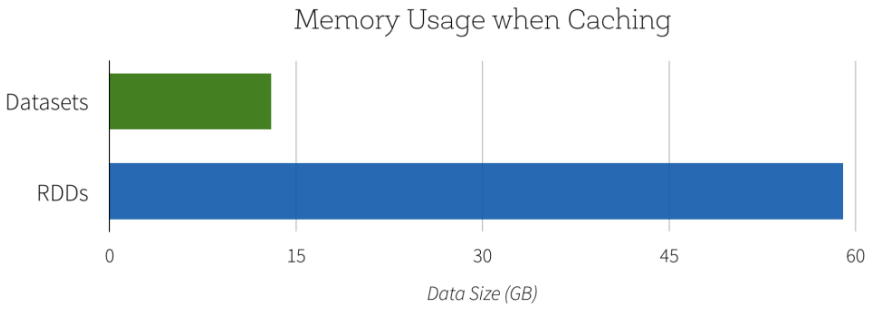
### Datasets

```
val counts = words
    .groupBy(_.toLowerCase)
    .count()
```

Since the Dataset version of word count can take advantage of the built-in aggregate **count**, this computation can not only be expressed with less code, but it will also execute significantly faster. As you can see in the graph below, the Dataset implementation runs much faster than the naive RDD implementation. In contrast, getting the same performance using RDDs would require users to manually consider how to express the computation in a way that parallelizes optimally.



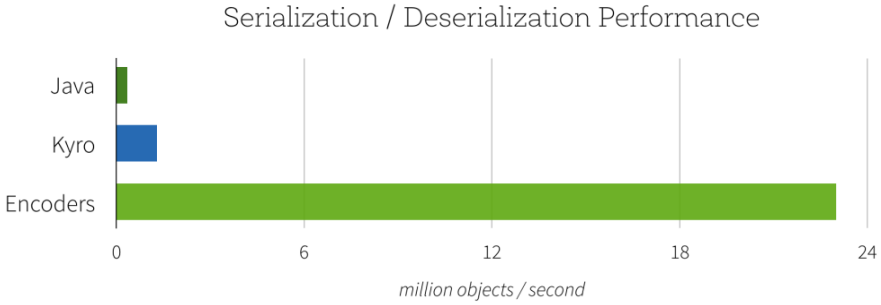

Another benefit of this new Dataset API is the reduction in memory usage. Since Spark understands the structure of data in Datasets, it can create a more optimal layout in memory when caching Datasets. In the following example, we compare caching several million strings in memory using Datasets as opposed to RDDs. In both cases, caching data can lead to significant performance improvements for subsequent queries. However, since Dataset encoders provide more information to Spark about the data being stored, the cached representation can be optimized to use 4.5x less space.




To help you get started, we've put together some example notebooks: Working with Classes, Word Count.

## Lightning-fast Serialization with Encoders

Encoders are highly optimized and use runtime code generation to build custom bytecode for serialization and deserialization. As a result, they can operate significantly faster than Java or Kryo serialization.




In addition to speed, the resulting serialized size of encoded data can also be significantly smaller (up to 2x), reducing the cost of network transfers. Furthermore, the serialized data is already in the Tungsten binary format, which means that many operations can be done in-place, without needing to materialize an object at all. Spark has built-in support for automatically generating encoders for primitive types (e.g. String, Integer, Long), Scala case classes, and Java Beans. We plan to open up this functionality and allow efficient serialization of custom types in a future release.

## Seamless Support for Semi-Structured Data

The power of encoders goes beyond performance. They also serve as a powerful bridge between semi-structured formats (e.g. JSON) and type-safe languages like Java and Scala.

For example, consider the following dataset about universities:

```
{"name": "UC Berkeley", "yearFounded": 1868, numStudents: 37581}
{"name": "MIT", "yearFounded": 1860, numStudents: 11318}
...
```

Instead of manually extracting fields and casting them to the desired type, you can simply define a class with the expected structure and map the input data to it. Columns are automatically lined up by name, and the types are preserved.

```
case class University(name: String, numStudents: Long,
yearFounded: Long)
val schools = sqlContext.read.json("/
schools.json").as[University]
schools.map(s => s"${s.name} is ${2015 - s.yearFounded}
years old")
```

Encoders eagerly check that your data matches the expected schema, providing helpful error messages before you attempt to incorrectly process TBs of data. For example, if we try to use a datatype that is too small, such that conversion to an object would result in truncation (i.e. numStudents is larger than a byte, which holds a maximum value of 255) the Analyzer will emit an AnalysisException.

```
case class University(numStudents: Byte)
val schools = sqlContext.read.json("/
schools.json").as[University]

org.apache.spark.sql.AnalysisException: Cannot upcast
`yearFounded` from bigint to smallint as it may truncate
```

When performing the mapping, encoders will automatically handle complex types, including nested classes, arrays, and maps.

## A Single API for Java and Scala

Another goal to the Dataset API is to provide a single interface that is usable in both Scala and Java. This unification is great news for Java users as it ensure that their APIs won't lag behind the Scala interfaces, code examples can easily be used from either language, and libraries no longer have to deal with two slightly different types of input. The only difference for Java users is they need to specify the encoder to use since the compiler does not provide type information. For example, if wanted to process json data using Java you could do it as follows:

```
public class University implements Serializable {
    private String name;
    private long numStudents;
    private long yearFounded;

    public void setName(String name) {...}
    public String getName() {...}
    public void setNumStudents(long numStudents) {...}
    public long getNumStudents() {...}
    public void setYearFounded(long yearFounded) {...}
    public long getYearFounded() {...}
}

class BuildString implements MapFunction<University,
String> {
    public String call(University u) throws Exception {
        return u.getName() + " is " + (2015 -
u.getYearFounded()) + " years old.";
    }
}
Dataset<University> schools = context.read().json("/
schools.json").as(Encoders.bean(University.class));
Dataset<String> strings = schools.map(new BuildString(),
Encoders.STRING());
```

## Looking Forward

While Datasets are a new API, we have made them interoperate easily with RDDs and existing Spark programs. Simply calling the `rdd()` method on a Dataset will give an RDD. In the long run, we hope that Datasets can become a common way to work with structured data, and we may converge the APIs even further.

As we look forward to Spark 2.0, we plan some exciting improvements to Datasets, specifically:

- Performance optimizations – In many cases, the current implementation of the Dataset API does not yet leverage the additional information it has and can be slower than RDDs. Over the next several releases, we will be working on improving the performance of this new API.
- Custom encoders – while we currently autogenerate encoders for a wide variety of types, we'd like to open up an API for custom objects.
- Python Support.
- Unification of DataFrames with Datasets – due to compatibility guarantees, DataFrames and Datasets currently cannot share a common parent class. With Spark 2.0, we will be able to unify these abstractions with minor changes to the API, making it easy to build libraries that work with both.

If you'd like to try out Datasets yourself, they are already available in Databricks. We've put together a few example notebooks for you to try out: Working with Classes, Word Count.

# Section 2:
# Tips and Tricks in Data Import

# An Introduction to JSON Support in Spark SQL

February 2, 2015 | by Yin Huai

*Get the Notebook*

Note: Starting Spark 1.3, SchemaRDD will be renamed to DataFrame.

In this article we introduce Spark SQL's JSON support, a feature we have been working on at Databricks to make it dramatically easier to query and create JSON data in Spark. With the prevalence of web and mobile applications, JSON has become the de-facto interchange format for web service API's as well as long-term storage. With existing tools, users often engineer complex pipelines to read and write JSON data sets within analytical systems. Spark SQL's JSON support, released in version 1.1 and enhanced in Spark 1.2, vastly simplifies the end-to-end-experience of working with JSON data.

## Existing practices

In practice, users often face difficulty in manipulating JSON data with modern analytical systems. To write a dataset to JSON format, users first need to write logic to convert their data to JSON. To read and query JSON datasets, a common practice is to use an ETL pipeline to transform JSON records to a pre-defined structure. In this case, users have to wait for this process to finish before they can consume their data. For both writing and reading, defining and maintaining schema definitions often make the ETL task more onerous, and eliminate many of the benefits of the semi-structured JSON format. If users want to consume fresh data, they either have to laboriously define the schema when they create external tables and then use a custom JSON serialization/deserialization library, or use a combination of JSON UDFs to query the data.

As an example, consider a dataset with following JSON schema:

```
[json]
{“name”:”Yin”, “address”:
{“city”:”Columbus”,”state”:”Ohio”}}
{“name”:”Michael”, “address”:{“city”:null,
“state”:”California”}}
[/json]
```

In a system like Hive, the JSON objects are typically stored as values of a single column. To access this data, fields in JSON objects are extracted and flattened using a UDF. In the SQL query shown below, the outer fields (name and address) are extracted and then the nested address field is further extracted.

*In the following example it is assumed that the JSON dataset shown above is stored in a table called people and JSON objects are stored in the column called jsonObject.*

```
SELECT
  v1.name, v2.city, v2.state
FROM people
  LATERAL VIEW json_tuple(people.jsonObject, 'name', 'address') v1
     as name, address
  LATERAL VIEW json_tuple(v1.address, 'city', 'state') v2
     as city, state;
```

## JSON support in Spark SQL

Spark SQL provides a natural syntax for querying JSON data along with automatic inference of JSON schemas for both reading and writing data. Spark SQL understands the nested fields in JSON data and allows users to directly access these fields without any explicit transformations. The above query in Spark SQL is written as follows:

```
SELECT name, age, address.city, address.state FROM people
```

## Loading and saving JSON datasets in Spark SQL

To query a JSON dataset in Spark SQL, one only needs to point Spark SQL to the location of the data. The schema of the dataset is inferred and natively available without any user specification. In the programmatic APIs, it can be done through jsonFile and jsonRDD methods provided by SQLContext. With these two methods, you can create a SchemaRDD for a given JSON dataset and then you can register the SchemaRDD as a table. Here is an example:

```
// Create a SQLContext (sc is an existing SparkContext)
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
// Suppose that you have a text file called people with the
following content:
// {"name":"Yin", "address":{"city":"Columbus","state":"Ohio"}}
// {"name":"Michael", "address":{"city":null,
"state":"California"}}
// Create a SchemaRDD for the JSON dataset.
val people = sqlContext.jsonFile("[the path to file people]")
// Register the created SchemaRDD as a temporary table.
people.registerTempTable(“people")
```

It is also possible to create a JSON dataset using a purely SQL API. For instance, for those connecting to Spark SQL via a JDBC server, they can use:

```
CREATE TEMPORARY TABLE people
USING org.apache.spark.sql.json
OPTIONS (path '[the path to the JSON dataset]’)
```

In the above examples, because a schema is not provided, Spark SQL will automatically infer the schema by scanning the JSON dataset. When a field is JSON object or array, Spark SQL will use STRUCT type and ARRAY type to represent the type of this field. Since JSON is semi-structured and different elements might have different schemas, Spark SQL will also resolve conflicts on data types of a field. To understand what is the schema of the JSON dataset, users can visualize the schema by using the method of printSchema() provided by the returned SchemaRDD in the programmatic APIs or by using DESCRIBE [table name] in SQL. For example, the schema of people visualized through people.printSchema() will be:

```
root
 |-- address: struct (nullable = true)
 |    |-- city: string (nullable = true)
 |    |-- state: string (nullable = true)
 |-- name: string (nullable = true)
```

Optionally, a user can apply a schema to a JSON dataset when creating the table using jsonFile and jsonRDD. In this case, Spark SQL will bind the provided schema to the JSON dataset and will not infer the schema. Users are not required to know all fields appearing in the JSON dataset. The specified schema can either be a subset of the fields appearing in the dataset or can have field that does not exist.

After creating the table representing a JSON dataset, users can easily write SQL queries on the JSON dataset just as they would on regular tables. As with all queries in Spark SQL, the result of a query is represented by another SchemaRDD. For example:

```
val nameAndAddress = sqlContext.sql("SELECT name, address.city,
address.state FROM people")
nameAndAddress.collect.foreach(println)
```

The result of a SQL query can be used directly and immediately by other data analytic tasks, for example a machine learning pipeline. Also, JSON datasets can be easily cached in Spark SQL's built in in-memory columnar store and be save in other formats such as Parquet or Avro.

## Saving SchemaRDDs as JSON files

In Spark SQL, SchemaRDDs can be output in JSON format through the toJSON method. Because a SchemaRDD always contains a schema (including support for nested and complex types), Spark SQL can automatically convert the dataset to JSON without any need for user-defined formatting. SchemaRDDs can themselves be created from many types of data sources, including Apache Hive tables, Parquet files, JDBC, Avro file, or as the result of queries on existing SchemaRDDs. This combination means users can migrate data into JSON format with minimal effort, regardless of the origin of the data source.

## What's next?

There are also several features in the pipeline that with further improve Spark SQL's support for semi-structured JSON data.

### Improved SQL API support to read/write JSON datasets

In Spark 1.3, we will introduce improved JSON support based on the new data source API for reading and writing various format using SQL. Users can create a table from a JSON dataset with an optional defined schema like what they can do with jsonFile and jsonRDD. Also, users can create a table and ask Spark SQL to store its rows in JSON objects. Data can inserted into this table through SQL. Finally, a CREATE TABLE AS SELECT statement can be used to create such a table and populate its data.

## Handling JSON datasets with a large number of fields

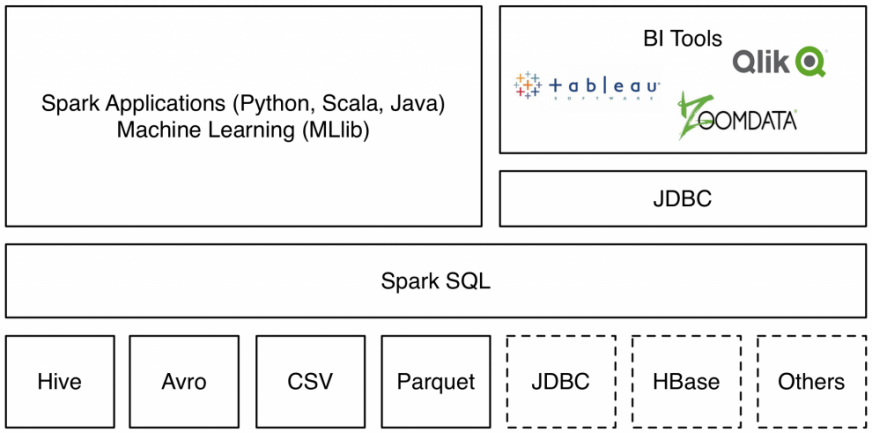
JSON data is often semi-structured, not always following a fixed schema. In the future, we will expand Spark SQL's JSON support to handle the case where each object in the dataset might have considerably different schema. For example, consider a dataset where JSON fields are used to hold key/value pairs representing HTTP headers. Each record might introduce new types of headers and using a distinct column for each one would produce a very wide schema. We plan to support auto-detecting this case and instead use a Map type. Thus, each row may contain a Map, enabling querying its key/value pairs. This way, Spark SQL will handle JSON datasets that have much less structure, pushing the boundary for the kind of queries SQL-based systems can handle.

*Get the Notebook*

# Spark SQL Data Sources API: Unified Data Access for the Spark Platform

January 9, 2015 by Michael Armbrust

Since the inception of Spark SQL in Spark 1.0, one of its most popular uses has been as a conduit for pulling data into the Spark platform. Early users loved Spark SQL's support for reading data from existing Apache Hive tables as well as from the popular Parquet columnar format. We've since added support for other formats, such as JSON. In Spark 1.2, we've taken the next step to allow Spark to integrate natively with a far larger number of input sources. These new integrations are made possible through the inclusion of the new Spark SQL Data Sources API.




The Data Sources API provides a pluggable mechanism for accessing structured data though Spark SQL. Data sources can be more than just simple pipes that convert data and pull it into Spark. The tight optimizer integration provided by this API means that filtering and column pruning can be pushed all the way down to the data source in many cases. Such integrated optimizations can vastly reduce the amount of data that needs to be processed and thus can significantly speed up Spark jobs.

Using a data sources is as easy as referencing it from SQL (or your favorite Spark language):

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

Another strength of the Data Sources API is that it gives users the ability to manipulate data in all of the languages that Spark supports, regardless of how the data is sourced. Data sources that are implemented in Scala, for example, can be used by pySpark users without any extra effort required of the library developer. Furthermore, Spark SQL makes it easy to join data from different data sources using a single interface. Taken together, these capabilities further unify the big data analytics solution provided by Spark 1.2.

Even though this API is still young, there are already several libraries built on top of it, including Apache Avro, Comma Separated Values (csv), and even dBASE Table File Format (dbf). Now that Spark 1.2 has been

officially released, we expect this list to grow quickly. We know of efforts underway to support HBase, JDBC, and more. Check out Spark Packages to find an up-to-date list of libraries that are available.

For developers that are interested in writing a library for their favorite format, we suggest that you study the reference library for reading Apache Avro, check out the example sources, or watch this meetup video.

Additionally, stay tuned for extensions to this API.  In Spark 1.3 we are hoping to add support for partitioning, persistent tables, and optional user specified schema.

# Section 3:
# Real-World Case Studies of Spark Analytics with Databricks

# Analyzing Apache Access Logs with Databricks

April 21, 2015 | by Ion Stoica and Vida Ha

*Get the Notebook*

Databricks provides a powerful platform to process, analyze, and visualize big and small data in one place. In this blog, we will illustrate how to analyze access logs of an Apache HTTP web server using Notebooks. Notebooks allow users to write and run arbitrary Spark code and interactively visualize the results. Currently, notebooks support three languages: Scala, Python, and SQL. In this blog, we will be using Python for illustration.

The analysis presented in this blog and much more is available in Databricks as part of the Databricks Guide. Find this notebook in your Databricks workspace at "`databricks_guide/Sample Applications/Log Analysis/Log Analysis in Python`" – it will also show you how to create a data frame of access logs with Python using the new Spark SQL 1.3 API. Additionally, there are also Scala & SQL notebooks in the same folder with similar analysis available.

## Getting Started

First we need to locate the log file. In this example, we are using synthetically generated logs which are stored in the "`/dbguide/sample_log`" file. The command below (typed in the notebook) assigns the log file pathname to the `DBFS_SAMPLE_LOGS_FOLDER` variable, which will be used throughout the rest of this analysis.

```
> DBFS_SAMPLE_LOGS_FOLDER = "/dbguide/sample_logs"

  Command took 0.07s
```

*Figure 1: Location Of The Synthetically Generated Logs In Your Instance Of Databricks Cloud*

## Parsing the Log File

Each line in the log file corresponds to an Apache web server access request. To parse the log file, we define `parse_apache_log_line()`, a function that takes a log line as an argument and returns the main fields of the log line. The return type of this function is a PySpark SQL Row object which models the web log access request. For this we use the "re" module which implements regular expression operations. The `APACHE_ACCESS_LOG_PATTERN` variable contains the regular expression used to match an access log line. In particular, `APACHE_ACCESS_LOG_PATTERN` matches client IP address (`ipAddress`) and identity (`clientIdentd`), user name as defined by HTTP authentication (`userId`), time when the server has finished processing the request (`dateTime`), the HTTP command issued by the client, e.g., GET (`method`), protocol, e.g., HTTP/1.0 (`protocol`), response code (`responseCode`), and the size of the response in bytes (`contentSize`).

```
> import re
from pyspark.sql import Row

APACHE_ACCESS_LOG_PATTERN = '^(\S+) (\S+) (\S+) \[([\w:/]+\s[+\-]\d{4})\] "(\S+) (\S+) (\S+)" (\d{3}) (\d+)'

# Returns a dictionary containing the parts of the Apache Access Log.
def parse_apache_log_line(logline):
    match = re.search(APACHE_ACCESS_LOG_PATTERN, logline)
    if match is None:
        # Optionally, you can change this to just ignore if each line of data is not critical.
        #   Corrupt data is common when writing to files.
        raise Error("Invalid logline: %s" % logline)
    return Row(
        ipAddress    = match.group(1),
        clientIdentd = match.group(2),
        userId       = match.group(3),
        dateTime     = match.group(4),
        method       = match.group(5),
        endpoint     = match.group(6),
        protocol     = match.group(7),
        responseCode = int(match.group(8)),
        contentSize  = long(match.group(9)))

Command took 0.04s
```

*Figure 2: Example Function To Parse The Log File In A Databricks Notebook*

## Loading the Log File

Now we are ready to load the logs into a Resilient Distributed Dataset (RDD). An RDD is a partitioned collection of tuples (rows), and is the primary data structure in Spark. Once the data is stored in an RDD, we can easily analyze and process it in parallel. To do so, we launch a Spark job that reads and parses each line in the log file using the `parse_apache_log_line()` function defined earlier, and then creates an RDD, called access_logs. Each tuple in access_logs contains the fields of a corresponding line (request) in the log file, `DBFS_SAMPLE_LOGS_FOLDER`. Note that once we create the `access_logs` RDD, we cache it into memory, by invoking the `cache()` method. This will dramatically speed up subsequent operations we will perform on `access_logs`.

```
> access_logs = (sc.textFile(DBFS_SAMPLE_LOGS_FOLDER)
                 # Call the parse_apace_log_line function on each line.
                 .map(parse_apache_log_line)
                 # Caches the objects in memory since they will be queried multiple times.
                 .cache())
  # An action must be called on the RDD to actually populate the cache.
  access_logs.count()

  Out[3]: 100000

  Command took 2.82s
```

*Figure 3: Example Code To Load The Log File In Databricks Notebook*

At the end of the above code snippet, notice that we count the number of tuples in `access_logs` (which returns 100,000 as a result).

# Log Analysis

Now we are ready to analyze the logs stored in the `access_logs` RDD. Below we give two simple examples:

1. Computing the average content size
2. Computing and plotting the frequency of each response code

## 1. Average Content Size

We compute the average content size in two steps. First, we create another RDD, `content_sizes`, that contains only the "`contentSize`" field from `access_logs`, and cache this RDD:

```
> # Create the content sizes rdd.
  content_sizes = (access_logs
                   .map(lambda row: row.contentSize)
                   .cache()) # Cache this as well since it will be queried many times.

  Command took 0.07s
```

*Figure 4: Create The Content Size Rdd In Databricks Notebook*

Second, we use the `reduce()` operator to compute the sum of all content sizes and then divide it into the total number of tuples to obtain the average:

```
> average_content_size = content_sizes.reduce(lambda x, y: x + y) / content_sizes.count()
  average_content_size
  Out[5]: 249L
```

*Figure 5: Computing The Average Content Size With The* `reduce()` *Operator*

The result is 249 bytes. Similarly we can easily compute the min and max, as well as other statistics of the content size distribution.

An important point to note is that both commands above run in parallel. Each RDD is partitioned across a set of workers, and each operation invoked on an RDD is shipped and executed in parallel at each worker on the corresponding RDD partition. For example the lambda function passed as the argument of `reduce()` will be executed in parallel at workers on each partition of the `content_sizes` RDD. This will result in computing the partial sums for each partition. Next, these partial sums are aggregated at the driver to obtain the total sum. The ability to cache RDDs and process them in parallel are the two of the main features of Spark that allows us to perform large scale, sophisticated analysis.

## 2. Computing and Plotting the Frequency of Each Response Code

We compute these counts using a map-reduce pattern. In particular, the code snippet returns an RDD `(response_code_to_count_pair_rdd)` of tuples, where each tuple associates a response code with its count.

```
> # First, calculate the response code to count pairs.
  response_code_to_count_pair_rdd = (access_logs
                                     .map(lambda row: (row.responseCode, 1))
                                     .reduceByKey(lambda x, y: x + y))
  response_code_to_count_pair_rdd
  Out[9]: PythonRDD[430] at RDD at PythonRDD.scala:43
  Command took 0.07s
```

Figure 6: Counting The Response Codes Using A Map-Reduce Pattern

Next, we take the first 100 tuples from `response_code_to_count_pair_rdd` to filter out possible bad data, and store the result in another RDD, `response_code_to_count_array`.

```
> # View the responseCodeToCount by calling take on the RDD - which outputs an array of tuples.
  # Notice the use of take(100) - just in case bad data may have slipped in and there are too many response codes.
  response_code_to_count_array = response_code_to_count_pair_rdd.take(100)
  response_code_to_count_array

  Out[10]: [(200, 71322), (500, 14355), (401, 14323)]

  Command took 0.53s
```

*Figure 7: Filter Out Possible Bad Data With* `take()`

To plot data we convert the `response_code_to_count_array` RDD into a DataFrame. A DataFrame is basically a table, and it is very similar to the DataFrame abstraction in the popular Python's pandas package. The resulting DataFrame (`response_code_to_count_data_frame`) has two columns "response code" and "count".

```
> # To call display(), the RDD of tuples must be converted to a DataFrame.
  # A simple map can accomplish that.
  response_code_to_count_row_rdd = response_code_to_count_pair_rdd.map(lambda (x, y): Row(response_code=x, count=y))
  response_code_to_count_data_frame = sqlContext.createDataFrame(response_code_to_count_row_rdd)

  Command took 0.27s
```

*Figure 8: Converting Rdd To Dataframe For Easy Data Manipulation And Visualization*

Now we can plot the count of response codes by simply invoking `display()` on our data frame.

```
> # Now, display can be called on the resulting DataFrame.
  #   For this display, a Pie Chart is chosen by selecting that icon.
  #   Then, "Plot Options..." was used to make the response_code the key and count as the value.
display(response_code_to_count_data_frame)
```

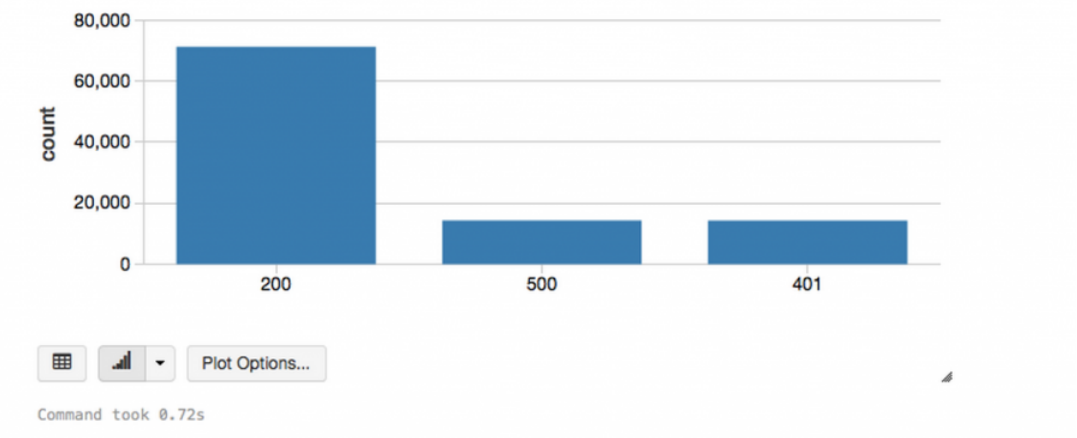



*Figure 9: Visualizing Response Codes With Display()*

If you want to change the plot size you can do so interactively by just clicking on the down arrow below the plot, and select another plot type. To illustrate this capability, here is the same data using a pie-chart.

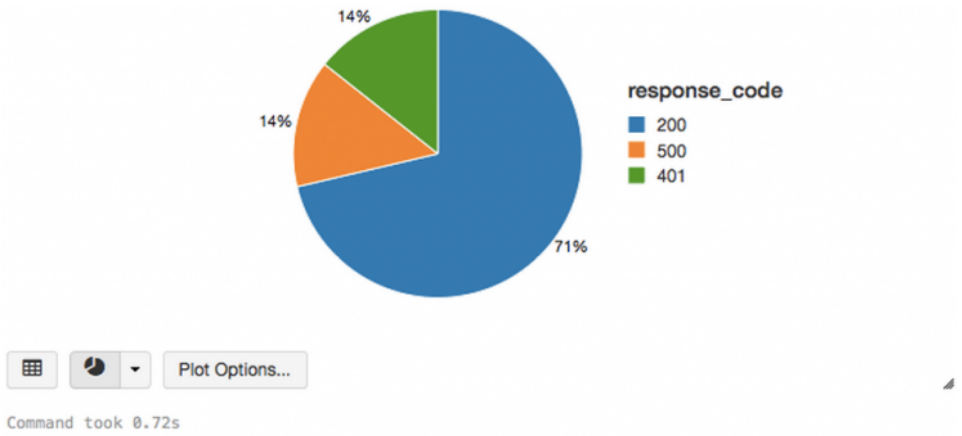



*Figure 10: Changing The Visualization Of Response Codes To A Pie Chart*

# Additional Resources

If you'd like to analyze your Apache access logs with Databricks, you can register here for an account. You can also find the source code on Github here.

Other Databricks how-tos can be found at: Easiest way to run Spark jobs

*Get the Notebook*

# Reshaping Data with Pivot in Spark

February 9, 2016 | by Andrew Ray, Silicon Valley Data Science

*This is a guest blog from our friend at Silicon Valley Data Science. Dr. Andrew Ray is passionate about big data and has extensive experience working with Spark. Andrew is an active contributor to the Apache Spark project including SparkSQL and GraphX.*

---

One of the many new features added in Spark 1.6 was the ability to pivot data, creating pivot tables, with a DataFrame (with Scala, Java, or Python). A pivot is an aggregation where one (or more in the general case) of the grouping columns has its distinct values transposed into individual columns. Pivot tables are an essential part of data analysis and reporting. Many popular data manipulation tools (pandas, reshape2, and Excel) and databases (MS SQL and Oracle 11g) include the ability to pivot data. I went over this briefly in a past post, but will be giving you a deep dive into the details here. Code for this post is available here.

## Syntax

In the course of doing the pull request for pivot, one of the pieces of research I did was to look at the syntax of many of the competing tools. I found a wide variety of syntax options. The two main competitors were pandas (Python) and reshape2 (R).

**Original DataFrame (df)**

| A | B | C | D |
|---|---|---|---|
| foo | one | small | 1 |
| foo | one | large | 2 |
| foo | one | large | 2 |
| foo | two | small | 3 |
| foo | two | small | 3 |
| bar | one | large | 4 |
| bar | one | small | 5 |
| bar | one | small | 6 |
| bar | two | large | 7 |

**Pivoted DataFrame**

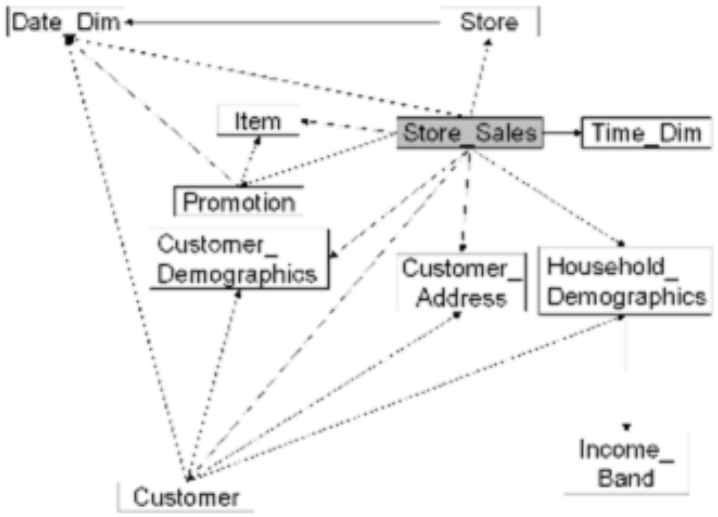
| A | B | large | small |
|---|---|---|---|
| foo | two | null | 6 |
| bar | two | 7 | 6 |
| foo | one | 4 | 1 |
| bar | one | 4 | 5 |

For example, say we wanted to group by two columns A and B, pivot on column C, and sum column D. In pandas the syntax would be `pivot_table(df, values='D', index=['A', 'B'], columns=['C'], aggfunc=np.sum)`. This is somewhat verbose, but clear. With reshape2, it is `dcast(df, A + B ~ C, sum)`, a very compact syntax thanks to the use of an R formula. Note that we did not have to specify the value column for reshape2; its inferred as the remaining column of the DataFrame (although it can be specified with another argument).

We came up with our own syntax that fit in nicely with the existing way to do aggregations on a DataFrame. To do the same group/pivot/sum in Spark the syntax is `df.groupBy("A", "B").pivot("C").sum("D")`. Hopefully this is a fairly intuitive syntax. But there is a small catch: to get better performance you need to specify the distinct values of the pivot column. If, for example, column C had two distinct values "small" and "large," then the more performant version would be `df.groupBy("A", "B").pivot("C", Seq("small", "large")).sum("D")`. Of course this is the Scala version, there are similar methods that take Java and Python lists.

## Reporting

Let's look at examples of real-world use cases. Say you are a large retailer (like my former employer) with sales data in a fairly standard transactional format, and you want to make some summary pivot tables. Sure, you could aggregate the data down to a manageable size and then use some other tool to create the final pivot table (although limited to the granularity of your initial aggregation). But now you can do it all in Spark (and you could before it just took a lot of IF's). Unfortunately, since no large retailers want to share their raw sales data with us we will have to use a synthetic example. A good one that I have used previously is the TPC-DS dataset. Its schema approximates what you would find in an actual retailer.



Since TPC-DS is a synthetic dataset that is used for benchmarking "big data" databases of various sizes, we are able to generate it in many "scale factors" that determine how large the output dataset is. For simplicity we will use scale factor 1, corresponding to about a 1GB dataset. Since the requirements are a little complicated I have a docker image that you can follow along with. Say we wanted to summarize sales by category and quarter with the later being columns in our pivot table. Then we would do the following (a more realistic query would probably have a few more conditions like time range).

```
(sql("""select *, concat('Q', d_qoy) as qoy
  from store_sales
  join date_dim on ss_sold_date_sk = d_date_sk
  join item on ss_item_sk = i_item_sk""")
  .groupBy("i_category")
  .pivot("qoy")
  .agg(round(sum("ss_sales_price")/1000000,2))
  .show)

+-----------+----+----+----+----+
| i_category|  Q1|  Q2|  Q3|  Q4|
+-----------+----+----+----+----+
|      Books|1.58|1.50|2.84|4.66|
|      Women|1.41|1.36|2.54|4.16|
|      Music|1.50|1.44|2.66|4.36|
|   Children|1.54|1.46|2.74|4.51|
|     Sports|1.47|1.40|2.62|4.30|
|      Shoes|1.51|1.48|2.68|4.46|
|    Jewelry|1.45|1.39|2.59|4.25|
|       null|0.04|0.04|0.07|0.13|
|Electronics|1.56|1.49|2.77|4.57|
|       Home|1.57|1.51|2.79|4.60|
|        Men|1.60|1.54|2.86|4.71|
+-----------+----+----+----+----+
```

Note that we put the sales numbers in millions to two decimals to keep this easy to look at. We notice a couple of things. First is that Q4 is crazy, this should come as no surprise for anyone familiar with retail. Second, most of these values within the same quarter with the exception of the null category are about the same. Unfortunately, even this great synthetic dataset is not completely realistic. Let me know if you have something better that is publicly available.

## Feature Generation

For a second example, let's look at feature generation for predictive models. It is not uncommon to have datasets with many observations of your target in the format of one per row (referred to as long form or narrow data). To build models, we need to first reshape this into one row per target; depending on the context this can be accomplished in a few ways. One way is with a pivot. This is potentially something you would not be able to do with other tools (like pandas, reshape2, or Excel), as the result set could be millions or billions of rows.

To keep the example easily reproducible, I'm going to use the relatively small MovieLens 1M dataset. This has about 1 million movie ratings from 6040 users on 3952 movies. Let's try to predict the gender of a user based on their ratings of the 100 most popular movies. In the below example the ratings table has three columns: user, movie, and rating.

```
+----+-----+------+
|user|movie|rating|
+----+-----+------+
|  11| 1753|     4|
|  11| 1682|     1|
|  11|  216|     4|
|  11| 2997|     4|
|  11| 1259|     3|
...
```

To come up with one row per user we pivot as follows:

```
val ratings_pivot =
ratings.groupBy("user").pivot("movie",
popular.toSeq).agg(expr("coalesce(first(rating),
3)").cast("double"))
```

Here, popular is a list of the most popular movies (by number of ratings) and we are using a default rating of 3. For user 11 this gives us something like:

```
+----+----+---+----+----+---+----+---+----+----+---+...
|user|2858|260|1196|1210|480|2028|589|2571|1270|593|...
+----+----+---+----+----+---+----+---+----+----+---+...
|  11| 5.0|3.0| 3.0| 3.0|4.0| 3.0|3.0| 3.0| 3.0|5.0|...
+----+----+---+----+----+---+----+---+----+----+---+...
```

Which is the wide form data that is required for modeling. See the complete example here. Some notes: I only used the 100 most popular movies because currently pivoting on thousands of distinct values is not particularly fast in the current implementation. More on this later.

## Tips and Tricks

For the best performance, specify the distinct values of your pivot column (if you know them). Otherwise, a job will be immediately launched to determine them {this is a limitation of other SQL engines as well as Spark SQL as the output columns are needed for planning}. Additionally, they will be placed in sorted order. For many things this makes sense, but for some, like the day of the week, this will not (Friday, Monday, Saturday, etc).

Pivot, just like normal aggregations, supports multiple aggregate expressions, just pass multiple arguments to the agg method. For example: `df.groupBy("A", "B").pivot("C").agg(sum("D"), avg("D"))`

Although the syntax only allows pivoting on one column, you can combine columns to get the same result as pivoting multiple columns. For example:

```
df.withColumn(“p”, concat($”p1”, $”p2”))
  .groupBy(“a”, “b”)
  .pivot(“p”)
  .agg(…)
```

Finally, you may be interested to know that there is a maximum number of values for the pivot column if none are specified. This is mainly to catch mistakes and avoid OOM situations. The config key is `spark.sql.pivotMaxValues` and its default is 10,000. You should probably not change it.

## Implementation

The implementation adds a new logical operator (`o.a.s.sql.catalyst.plans.logical.Pivot`). That logical operator is translated by a new analyzer rule (`o.a.s.sql.catalyst.analysis.Analyzer.ResolvePivot`) that currently translates it into an aggregation with lots of if statements, one expression per pivot value.

For example, `df.groupBy("A", "B").pivot("C", Seq("small", "large")).sum("D")` would be translated into the equivalent of `df.groupBy("A", "B").agg(expr("sum(if(C = 'small', D, null))"), expr("sum(if(C = 'large', D, null))"))`. You could have done this yourself but it would get long and possibly error prone quickly.

## Future Work

There is still plenty that can be done to improve pivot functionality in Spark:

- Make it easier to do in the user's language of choice by adding pivot to the R API and to the SQL syntax (similar to Oracle 11g and MS SQL).
- Add support for unpivot which is roughly the reverse of pivot.
- Speed up the implementation of pivot when there are many distinct values in the pivot column. I'm already working on an idea for this.

# An Illustrated Guide to Advertising Analytics

February 2, 2016 | by Grega Kešpret and Denny Lee

*Get the Notebook*

*This is a joint blog with our friend at Celtra. Grega Kešpret is the Director of Engineering. He leads a team of engineers and data scientists to build analytics pipeline and optimization systems for Celtra.*

---

Advertising technology companies that want to analyze their immense stores and varieties of data require a scalable, extensible, and elastic platform. With Databricks, Celtra was able to scale their Big Data analysis projects six-fold, leading to better-informed product design and quicker issue detection and resolution.

Celtra provides agencies, media suppliers, and brand leaders alike with an integrated, scalable HTML5 technology for brand advertising on smartphones, tablets, and desktop. The platform, AdCreator 4, gives clients such as MEC, Kargo, Pepsi, and Macy's the ability to easily create, manage, and traffic sophisticated data-driven dynamic ads, optimize them on the go, and track their performance with insightful analytics.

## Advertising Analytics Challenges

Like most advertising platforms, Celtra needed the ability to go far beyond calculations provided by precomputed aggregations (e.g. canned reports). They also needed:

- the flexibility to perform uniques, order statistics, and other metrics outside the boundaries of existing pre-designed data models;
- to combine their metric calculations with visualizations to more quickly and better understand their data;
- and to have short development cycles so they could experiment with different analysis much more quickly.

To complicate matters further, Celtra's data sources are diverse involving a wide variety of creative capabilities within a complex ecosystem (e.g. high cardinality). With analysis focused on consumer engagement with their clients' ads, Celtra was constantly exploring new ways to leverage this information to improve their data products.

In the past, Celtra's original environments had the following issues:

- they were dealing with complex setup and configurations that required their data scientists to focus on infrastructure work instead of focusing their data problems;
- there was a limited number of people working on their solution resulting in all of their big data analysis being bottlenecked with their analytics engineering team;
- and the lack of a collaborative environment resulted in analyses that were not reproducible nor repeatable.

## Data Sciences and Simplified Operations with Databricks

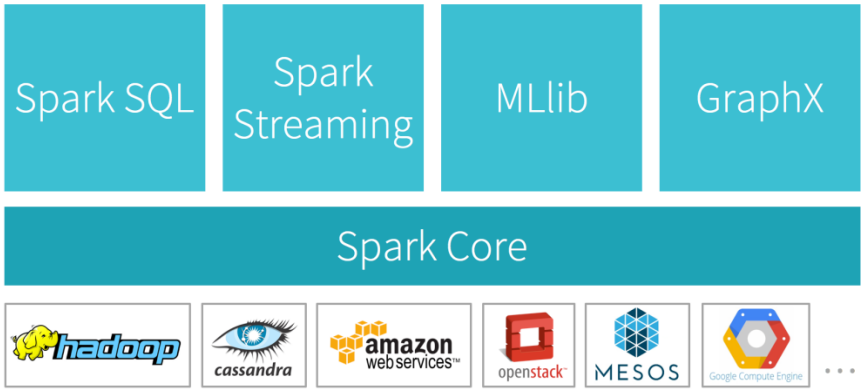
With Databricks, Celtra was able to address the challenges above and:

- They reduced the load on their analytics engineering team by expanding access to the number of people able to work with the data directly by a factor of four.
- Allowed their teams to effortlessly manage their Spark clusters and managed issues ranging from high availability to the optimized setups and configurations within AWS.
- This increased the amount of ad-hoc analysis done six-fold, leading to better-informed product design and quicker issue detection and resolution.
- With Databricks' integrated workspace, they were able to quickly build notebooks with visualizations that increased collaboration and improved reproducibility and repeatability of analysis.

While Spark allows you to solve a wide variety of data problems with multiple languages in a scalable, elastic, distributed environment; Databricks simplified operations and reduced the need for dedicated personnel to maintain the infrastructure.

## Why Spark for Event Analytics

Apache Spark™ is a powerful open-source processing engine built around speed, ease of use, and sophisticated analytics. Spark comes packaged with support for ETL, interactive queries (SQL), advanced analytics (e.g. machine learning), and streaming over large datasets.



In addition to being scalable and fault tolerant, Spark allows you to program in your language of choice including Python, Java, Scala, R, and SQL. For Celtra and other advertising customers, Spark provides distinct benefits for AdTech and event analytics including:

- the ability to solve multiple data problems (e.g. streaming, machine learning, and analytics) using the same data platform;
- access to an expressive computation layer;
- a fast pace of innovation (with 1000 total code contributors in 2015);
- and seamless integration with S3.

Particularly convenient is the ability to do event sessionization with a simple yet powerful API such as the code below:

```
def analyze(events: RDD[Event]): RDD[Session] = {
   events
      .keyBy(_.sessionId)
      .groupByKey(groupTasks)
      .values()
      .flatMap(computeSession)
}
```

For more information, please check out the webinar How Celtra Optimizes its Advertising Platform with Databricks.

## Making Sense of your Advertising Data

To make sense of your advertising weblog data, log into Databricks and you will immediately be able to begin working with a Databricks notebook. Our notebooks provide much more than just data visualization, they also support multiple languages (R, Python, Scala, SQL, and Markdown), mixing languages within the same notebook, versioning with GitHub, real-time collaboration, one-click to production (the ability to execute a notebook as a separate scheduled job), and the ability to export notebooks in multiple archive formats including HTML.

In this example, we will perform the following tasks:

1. Create an external table against a large amount of web access logs including the use of a regular expression to parse a series of logs.
2. Identify each visitor's country (ISO-3166-1 three-letter ISO country code) based on IP address by calling a REST Web service API.
3. Identify each visitor's browser and OS information based on their User-Agent string using the user-agents PyPI package.
4. Convert the Apache web logs date information, create a userid, and join back to the browser and OS information.

## Step 1: Making sense of the access logs

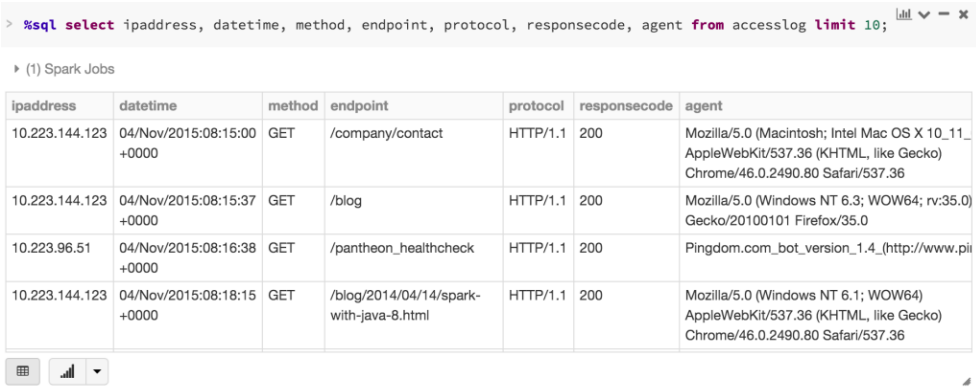
The primary data source for advertising is an Apache web access log. Below is a sample row from one of those logs.

```
10.109.100.123 - - [04/Dec/2015:08:15:00 +0000]  "GET /
company/info HTTP/1.1" 200 8572 "https://databricks.com/"
"Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_0)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/46.0.2490.80
Safari/537.36" 0.304 "11.111.111.111, 22.222.222.2222,
33.111.111.111, 44.111.111.111”
```

Traditionally, to make sense of this data, developers would need to build custom ETL processes to provide structure against this data (i.e. convert it into a table). With Spark, instead of spending a lot of resources to make sense of the Apache access log format, you can define an external table using regular expressions to parse your weblogs stored within S3.

```
CREATE EXTERNAL TABLE accesslog (
  ipaddress STRING,
  ...
)
ROW FORMAT
  SERDE 'org.apache.hadoop.hive.serde2.RegexSerDe'
WITH SERDEPROPERTIES (
  "input.regex" = '^(\\S+) (\\S+) (\\S+) \\[([\\w:/]+\\s[+\
\-]\\d{4})\\]  \\"(\\S+) (\\S+) (\\S+)\\" (\\d{3}) (\\d+) \
\"(.*)\\" \\"(.*)\\" (\\S+) \\"(\\S+), (\\S+), (\\S+), (\\S
+)\\"'
)
LOCATION
  “/mnt/mdl/accesslogs/"
```

With the creation of this table, you can execute a Spark SQL query against this data similar to how you would query any other structured data source. Note, the underlying source of this external table is still your log files that you had stored in S3.



```
%sql select ipaddress, datetime, method, endpoint, protocol, responsecode, agent from accesslog limit 10;
```

(1) Spark Jobs

| ipaddress | datetime | method | endpoint | protocol | responsecode | agent |
|---|---|---|---|---|---|---|
| 10.223.144.123 | 04/Nov/2015:08:15:00 +0000 | GET | /company/contact | HTTP/1.1 | 200 | Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_ AppleWebKit/537.36 (KHTML, like Gecko) Chrome/46.0.2490.80 Safari/537.36 |
| 10.223.144.123 | 04/Nov/2015:08:15:37 +0000 | GET | /blog | HTTP/1.1 | 200 | Mozilla/5.0 (Windows NT 6.3; WOW64; rv:35.0) Gecko/20100101 Firefox/35.0 |
| 10.223.96.51 | 04/Nov/2015:08:16:38 +0000 | GET | /pantheon_healthcheck | HTTP/1.1 | 200 | Pingdom.com_bot_version_1.4_(http://www.pi |
| 10.223.144.123 | 04/Nov/2015:08:18:15 +0000 | GET | /blog/2014/04/14/spark-with-java-8.html | HTTP/1.1 | 200 | Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/46.0.2490.80 Safari/537.36 |

In addition to Spark’s in-memory computing, Databricks makes use of the blazingly fast SSD-backed EC2 R3 instances to provide both in-memory and file caching for faster processing and querying. Prior to creating your table, you can create a Databricks File System mount (as denoted by /mnt in the above Spark SQL code) by following the Databricks Data Import How-To Guide to leverage both SSDs and Tachyon in-memory file system.

## Step 2: Identify Geo-location information based on IP address

Often included within weblogs are the client IP addresses that can potentially provide you the approximate location of the site visitor. While multiple methods exist to convert IP address to geolocation information, a quick (non-production) way to do this is to make use of an external web service such as http://freegeoip.net/. In the sample code below, we are making a web service call directly from Spark:

```
# Obtain the unique agents from the accesslog table
ipaddresses = sqlContext.sql("select distinct ip1 from \
 accesslog where ip1 is not null").rdd

# getCCA2: Obtains two letter country code based on IP
address
def getCCA2(ip):
  url = 'http://freegeoip.net/csv/' + ip
  str = urllib2.urlopen(url).read()
  return str.split(",")[1]

# Loop through distinct IP addresses and obtain two-
letter country codes
mappedIPs = ipaddresses.map(lambda x: (x[0],
getCCA2(x[0])))
```

Using Python and Spark, the first line of code creates the **ipaddresses** RDD which uses a **sqlContext** to store the distinct IP addresses from the **accesslog** external table (based on the access log data stored on S3). The subsequent **getCCA2** function is the call to the http://freegeoip.net web service which receives an IP address and returns (in this case) a comma-delimited message containing the geo-location information. The final call of this code snippet is a map function which allows Spark to loop through all of the unique IP addresses stored within **ipaddresses** RDD and make the web service call defined by **getCCA2**. This is similar to a **for** loop, except this workload is partitioned and distributed to many nodes in your Spark cluster to be completed in parallel.

## Step 3: Making sense of Browser and OS information

The user agent string (or agent) is part of a browser (or client) header when visiting a web page. This header often contains interesting information such as what browser, operating system, or device that users are using to view your ads. For example, the on-demand webinar How Celtra Optimizes its Advertising Platform with Databricks discussed how Celtra was able to troubleshoot different devices viewing their ads by analyzing their web and troubleshooting logs.

Below is an example user agent string which reveals an operating system (Mac OSX 10.11 El Capitan) and browser (Google Chrome 46.0.2490.80).

```
Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_0) AppleWebKit/
537.36 (KHTML, like Gecko) Chrome/46.0.2490.80 Safari/537.36
```

Instead of spending time writing custom code to parse this string, Databricks is extensible and allows you to include external packages such

as the user-agents via PyPI. Together with PySpark UDFs, you can add columns to an existing Spark DataFrame combining a python function and Spark SQL.

```
from user_agents import parse
from pyspark.sql.types import StringType
from pyspark.sql.functions import udf

# Create UDFs to extract out Browser Family information
def browserFamily(ua_string) : return
parse(ua_string).browser.family
udfBrowserFamily = udf(browserFamily, StringType())

# Obtain the unique agents from the accesslog table
userAgentTbl = sqlContext.sql("select distinct agent from
accesslog")

# Add new columns to the UserAgentInfo DataFrame containing
browser information
userAgentInfo = userAgentTbl.withColumn('browserFamily', \
   udfBrowserFamily(userAgentTbl.agent))

# Register the DataFrame as a table
userAgentInfo.registerTempTable("UserAgentInfo")
```

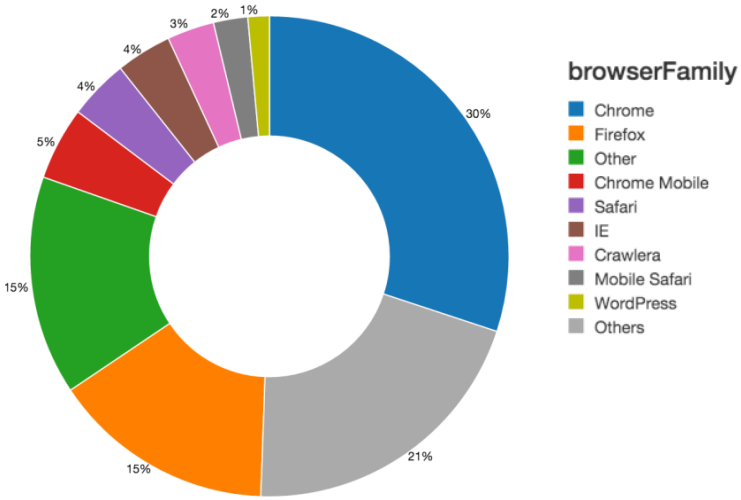
The `browserFamily` function utilizes the user-agents PyPI package parse function which takes an user agent string input and returns the browser Family information (e.g. Firefox, Safari, Chrome, etc.). The subsequent `udfBrowserFamily` UDF defines the output of the `browserFamily` function as `StringType()` so it can be properly internalized within a subsequent DataFrame.

The `userAgentTbl` is a Spark DataFrame that contains the unique agents from the `accesslog` table from Step 1. To add the browser family information as defined by the agent string, the new `UserAgentInfo` DataFrame is created by using `.withColumn` defining the column name (`browserFamily`) and the string datatype output from `udfBrowserFamily`.

Once the DataFrame has been created, you can execute a Spark SQL query within the same Python notebook. For example, to see the breakdown by `browserFamily` within the `UserAgentInfo` table, execute the following query in your notebook:

```
%sql
SELECT browserFamily, count(1)
FROM UserAgentInfo
GROUP BY browserFamily
```

Within the same notebook, you will see the following donut chart:



## Step 4: Complete our basic preparation

To put this all together, we will do the following tasks to complete our basic preparation of these web access logs for analysis:

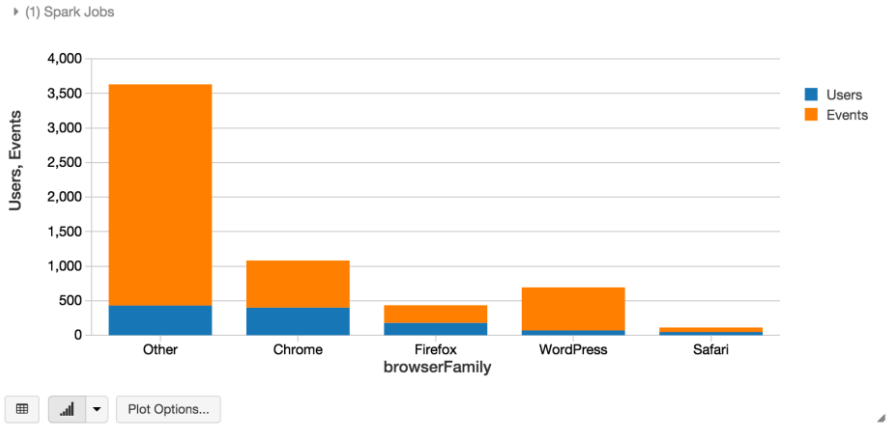
```
# Define function (converts Apache web log time)
def weblog2Time(weblog_timestr): …

# Define and Register UDF
udfWeblog2Time = udf(weblog2Time, DateType())
sqlContext.registerFunction("udfWeblog2Time", lambda x:
weblog2Time(x))

# Create DataFrame
accessLogsPrime = sqlContext.sql("select hash(a.ip1,
a.agent) as UserId, m.cca3, udfWeblog2Time(a.datetime) as
LogDateTime,... from accesslog join UserAgentInfo u on
u.agent = a.agent join mappedIP3 m on m.ip = a.ip1")
```

The **weblog2Time** function performs the task of converting the Apache web log time to an ISO-8601 date format. Within the **sqlContext**, to unique-ify the site visitors (in this case we lack a cookieId that anonymously identifies users), we can combine IP address and the user agent string as the **UserId**. To combine back to the browser and OS information as well as country (based on IP address) information, the same **sqlContext** includes a join statement to the **UserAgentInfo** and **mappedIP3** tables.

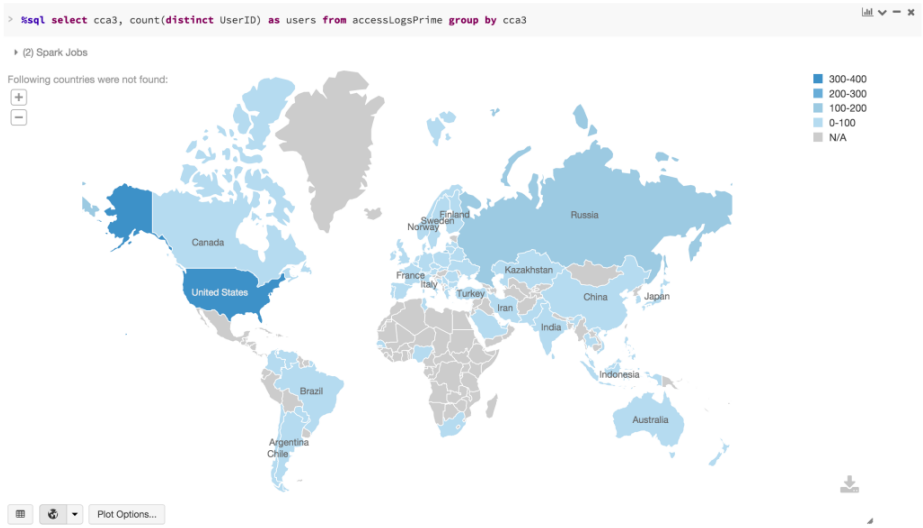
## Visualize This!

With your preparation completed, you can quickly analyze and visualize your data, all from within the same notebook. For example, with the browser information included within the **accesslogprime** DataFrame, we can quickly identify the top 5 browsers by users and events.

```
%sql select browserFamily, count(distinct UserID) as Users, count(1) as Events from accessLogsPrime group by
browserFamily order by Users desc limit 5;
```



In the same Python notebook, we can also identify where site visitors are coming from based on their IP addresses as noted in the map below:




## Conclusion

Databricks allows you to quickly jump start your advertising analysis. To access the complete notebook including the functioning code and charts, you can view the Databricks AdTech Sample Notebook (Part 1) exported HTML notebook. For more information about advertising platform optimization, check out the on-demand webinar How Celtra Optimizes Its Advertising Platform with Databricks.

For a free trial of Databricks, please sign up at Databricks Registration.

*Get the Notebook*

# Automatic Labs Selects Databricks for Primary Real-Time Data Processing

February 12, 2015 | by Kavitha Mariappan

We're really excited to share that Automatic Labs has selected Databricks as its preferred big data processing platform.

Press release: http://www.marketwired.com/press-release/automatic-labs-turns-databricks-cloud-faster-innovation-dramatic-cost-savings-1991316.htm

Automatic Labs needed to run large and complex queries against their entire data set to explore and come up with new product ideas. Their prior solution using Postgres impeded the ability of Automatic's team to efficiently explore data because queries took days to run and data could not be easily visualized, preventing Automatic Labs from bringing critical new products to market. They then deployed Databricks, our simple yet powerful unified big data processing platform on Amazon Web Services (AWS) and realized these key benefits:

- **Reduced time to bring product to market.** Minimized the time to validate a product idea from months to days by speeding up the interactive exploration over Automatic's entire data set, and completing queries in minutes instead of days.
- **Eliminated DevOps and non-core activities.** Freed up one full-time data scientist from non-core activities such as DevOps and infrastructure maintenance to perform core data science activities.
- **Infrastructure savings.** Realized savings of ten thousand dollars in one month alone on AWS costs due to the ability to instantly set up and tear-down Spark clusters

With a mission to connect all cars on the road to the internet, Automatic Labs is now able to run large and complex production workloads with Databricks to explore new product ideas and bring them to market faster, such as custom driving reports, recommendations for users regarding fuel-efficient driving and more.

Download this case study learn more about how Automatic Labs is using Databricks.

# Conclusion

Our mission at Databricks is to dramatically simplify big data processing so organizations can immediately start working on their data problems in an environment accessible to data scientists, engineers, and business users alike. We hope the collection of blog posts we've curated in this e-book will provide you with the insights and tools to help you solve your biggest data problems.

If you enjoyed the technical content in this e-book, visit the Databricks Blog for more technical tips, best practices, and case studies from the Spark experts at Databricks.

**Read all the books in this Series:**

Apache Spark Analytics Made Simple
Mastering Advanced Analytics with Apache Spark
Lessons for Large-Scale Machine Learning Deployments on Apache Spark
Building Real-Time Applications with Spark Streaming

**To learn more about Databricks, check out some of these resources:**

Databricks Primer
How-To Guide: The Easiest Way to Run Spark Jobs
Solution Brief: Making Data Warehousing Simple
Solution Brief: Making Machine Learning Simple
White Paper: Simplifying Spark Operations with Databricks

**To try out Databricks for yourself, sign-up for a 14-day free trial today!**