[Spark RDDs vs DataFrames vs SparkSQL](https://community.cloudera.com/t5/Community-Articles/Spark-RDDs-vs-DataFrames-vs-SparkSQL/ta-p/246547)

<https://community.cloudera.com/t5/Community-Articles/Spark-RDDs-vs-DataFrames-vs-SparkSQL/ta-p/246547>

Synopsis

This tutorial will demonstrate using Spark for data processing operations on a large set of data consisting of pipe delimited text files. Delimited text files are a common format seen in Data Warehousing:

1. Random lookup for a single record
2. Grouping data with aggregation and sorting the output

3 Different techniques will be used to solve the above 2 problems and then compare how they perform against each other:

1. Using RDD’s
2. Using DataFrames
3. Using SparkSQL

The purpose of this tutorial is to provide you with code snippets for the above 3 techniques and to demonstrate how RDD’s outperform DataFrames and SparkSQL for certain types of data processing. See below at the end of this article for all code.

Also, these tests are demonstrating the native functionality within Spark for RDDs, DataFrames, and SparkSQL without calling additional modules/readers for file format conversions or other optimizations.

For joining datasets, DataFrames and SparkSQL are much more intuitive to use, especially SparkSQL, and may perhaps yield better performance results than RDDs.

Source Data:

* 9 Million unique order records across 3 files in HDFS
* Total size of HDFS data files is 1.4 GB
* Each order record could be for 1 of 8 different products
* Pipe delimited text files with each record containing 11 fields
* Data is fictitious and was auto-generated programmatically

Environment:

* HDP 2.4
* Hadoop version 2.7
* Spark 1.6
* HDP Sandbox

RDDs, DataFrames, and SparkSQL:

At its core, Spark operates on the concept of Resilient Distributed Datasets, or RDD’s:

* Resilient - if data in memory is lost, it can be recreated
* Distributed - immutable distributed collection of objects in memory partitioned across many data nodes in a cluster
* Dataset - initial data can from from files, be created programmatically, from data in memory, or from another RDD

DataFrames API is a data abstraction framework that organizes your data into named columns:

* Create a schema for the data
* Conceptually equivalent to a table in a relational database
* Can be constructed from many sources including structured data files, tables in Hive, external databases, or existing RDDs
* Provides a relational view of the data for easy SQL like data manipulations and aggregations
* Under the hood, it is a row of RDD’s

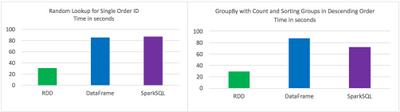
SparkSQL is a Spark module for structured data processing. You can interact with SparkSQL through:

* SQL
* DataFrames API
* Datasets API

Test results:

* RDD’s outperformed DataFrames and SparkSQL for certain types of data processing
* DataFrames and SparkSQL performed almost about the same, although with analysis involving aggregation and sorting SparkSQL had a slight advantage
* Syntactically speaking, DataFrames and SparkSQL are much more intuitive than using RDD’s
* Took the best out of 3 for each test
* Times were consistent and not much variation between tests
* Jobs were run individually with no other jobs running

1. Random lookup against 1 order ID from 9 Million unique order ID's
2. GROUP all the different products with their total COUNTS and SORT DESCENDING by product name



Code:

RDD Random Lookup

#!/usr/bin/env python

from time import time

from pyspark import SparkConf, SparkContext

conf = (SparkConf()

.setAppName("rdd\_random\_lookup")

.set("spark.executor.instances", "10")

.set("spark.executor.cores", 2)

.set("spark.dynamicAllocation.enabled", "false")

.set("spark.shuffle.service.enabled", "false")

.set("spark.executor.memory", "500MB"))

sc = SparkContext(conf = conf)

t0 = time()

path = "/data/customer\_orders\*"

lines = sc.textFile(path)

## filter where the order\_id, the second field, is equal to 96922894

print lines.map(lambda line: line.split('|')).filter(lambda line: int(line[1]) == 96922894).collect()

tt = str(time() - t0)

print "RDD lookup performed in " + tt + " seconds"

DataFrame Random Lookup

#!/usr/bin/env python

from time import time

from pyspark.sql import \*

from pyspark import SparkConf, SparkContext

conf = (SparkConf()

.setAppName("data\_frame\_random\_lookup")

.set("spark.executor.instances", "10")

.set("spark.executor.cores", 2)

.set("spark.dynamicAllocation.enabled", "false")

.set("spark.shuffle.service.enabled", "false")

.set("spark.executor.memory", "500MB"))

sc = SparkContext(conf = conf)

sqlContext = SQLContext(sc)

t0 = time()

path = "/data/customer\_orders\*"

lines = sc.textFile(path)

## create data frame

orders\_df = sqlContext.createDataFrame( lines.map(lambda l: l.split("|")) .map(lambda p: Row(cust\_id=int(p[0]), order\_id=int(p[1]), email\_hash=p[2], ssn\_hash=p[3], product\_id=int(p[4]), product\_desc=p[5], country=p[6], state=p[7], shipping\_carrier=p[8], shipping\_type=p[9], shipping\_class=p[10] ) ) )

## filter where the order\_id, the second field, is equal to 96922894

orders\_df.where(orders\_df['order\_id'] == 96922894).show()

tt = str(time() - t0)

print "DataFrame performed in " + tt + " seconds"

SparkSQL Random Lookup

#!/usr/bin/env python

from time import time

from pyspark.sql import \*

from pyspark import SparkConf, SparkContext

conf = (SparkConf()

.setAppName("spark\_sql\_random\_lookup")

.set("spark.executor.instances", "10")

.set("spark.executor.cores", 2)

.set("spark.dynamicAllocation.enabled", "false")

.set("spark.shuffle.service.enabled", "false")

.set("spark.executor.memory", "500MB"))

sc = SparkContext(conf = conf)

sqlContext = SQLContext(sc)

t0 = time()

path = "/data/customer\_orders\*"

lines = sc.textFile(path)

## create data frame

orders\_df = sqlContext.createDataFrame( lines.map(lambda l: l.split("|")) .map(lambda p: Row(cust\_id=int(p[0]), order\_id=int(p[1]), email\_hash=p[2], ssn\_hash=p[3], product\_id=int(p[4]), product\_desc=p[5], country=p[6], state=p[7], shipping\_carrier=p[8], shipping\_type=p[9], shipping\_class=p[10] ) ) )

## register data frame as a temporary table

orders\_df.registerTempTable("orders")

## filter where the customer\_id, the first field, is equal to 96922894

print sqlContext.sql("SELECT \* FROM orders where order\_id = 96922894").collect()

tt = str(time() - t0)

print "SparkSQL performed in " + tt + " seconds"

RDD with GroupBy, Count, and Sort Descending

#!/usr/bin/env python

from time import time

from pyspark import SparkConf, SparkContext

conf = (SparkConf()

.setAppName("rdd\_aggregation\_and\_sort")

.set("spark.executor.instances", "10")

.set("spark.executor.cores", 2)

.set("spark.dynamicAllocation.enabled", "false")

.set("spark.shuffle.service.enabled", "false")

.set("spark.executor.memory", "500MB"))

sc = SparkContext(conf = conf)

t0 = time()

path = "/data/customer\_orders\*"

lines = sc.textFile(path)

counts = lines.map(lambda line: line.split('|')) .map(lambda x: (x[5], 1)) .reduceByKey(lambda a, b: a + b) .map(lambda x:(x[1],x[0])) .sortByKey(ascending=False)

for x in counts.collect():

print x[1] + '\t' + str(x[0])

tt = str(time() - t0)

print "RDD GroupBy performed in " + tt + " seconds"

DataFrame with GroupBy, Count, and Sort Descending

#!/usr/bin/env python

from time import time

from pyspark.sql import \*

from pyspark import SparkConf, SparkContext

conf = (SparkConf()

.setAppName("data\_frame\_aggregation\_and\_sort")

.set("spark.executor.instances", "10")

.set("spark.executor.cores", 2)

.set("spark.dynamicAllocation.enabled", "false")

.set("spark.shuffle.service.enabled", "false")

.set("spark.executor.memory", "500MB"))

sc = SparkContext(conf = conf)

sqlContext = SQLContext(sc)

t0 = time()

path = "/data/customer\_orders\*"

lines = sc.textFile(path)

## create data frame

orders\_df = sqlContext.createDataFrame( lines.map(lambda l: l.split("|")) .map(lambda p: Row(cust\_id=int(p[0]), order\_id=int(p[1]), email\_hash=p[2], ssn\_hash=p[3], product\_id=int(p[4]), product\_desc=p[5], country=p[6], state=p[7], shipping\_carrier=p[8], shipping\_type=p[9], shipping\_class=p[10] ) ) )

results = orders\_df.groupBy(orders\_df['product\_desc']).count().sort("count",ascending=False)

for x in results.collect():

print x

tt = str(time() - t0)

print "DataFrame performed in " + tt + " seconds"

SparkSQL with GroupBy, Count, and Sort Descending

#!/usr/bin/env python

from time import time

from pyspark.sql import \*

from pyspark import SparkConf, SparkContext

conf = (SparkConf()

.setAppName("spark\_sql\_aggregation\_and\_sort")

.set("spark.executor.instances", "10")

.set("spark.executor.cores", 2)

.set("spark.dynamicAllocation.enabled", "false")

.set("spark.shuffle.service.enabled", "false")

.set("spark.executor.memory", "500MB"))

sc = SparkContext(conf = conf)

sqlContext = SQLContext(sc)

t0 = time()

path = "/data/customer\_orders\*"

lines = sc.textFile(path)

## create data frame

orders\_df = sqlContext.createDataFrame(lines.map(lambda l: l.split("|")) .map(lambda r: Row(product=r[5])))

## register data frame as a temporary table

orders\_df.registerTempTable("orders")

results = sqlContext.sql("SELECT product, count(\*) AS total\_count FROM orders GROUP BY product ORDER BY total\_count DESC")

for x in results.collect():

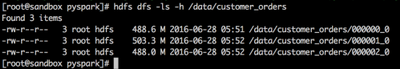
print x

tt = str(time() - t0)

print "SparkSQL performed in " + tt + " seconds"

Verification:

Source data files in HDFS:



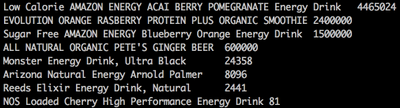
Total size of source data files in HDFS:

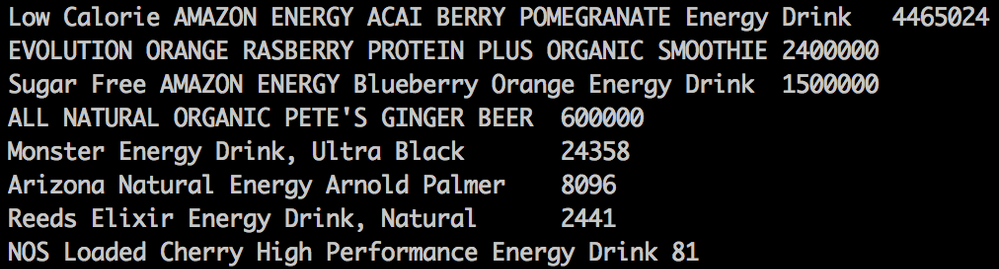
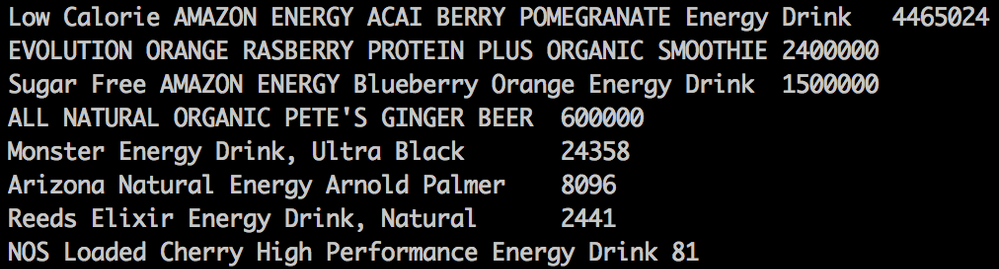
5295-screen-shot-2016-06-27-at-105801-pm.png

Order ID is second field in pipe delimited file. There are 9 Million unique order ID records:

5296-screen-shot-2016-06-27-at-110051-pm.png

Output produced by GroupBy, Count, and Sort Descending (format will not be same for all, however, numbers will be same):





<https://www.linkedin.com/pulse/spark-join-types-explained-ram-ghadiyaram?trk=related_artice_Spark%20Join%20types%20explained..._article-card_title>

<https://www.linkedin.com/pulse/scala-recursion-demo-normal-java-style-well-tail-ram-ghadiyaram?trk=portfolio_article-card_title>

<https://www.linkedin.com/today/author/ram-ghadiyaram-82734811>

Question : What is the difference in these two approaches? Is there any performance gain with using Dataframe APIs?

Answer :

There is comparative study done by horton works. [source](https://community.hortonworks.com/articles/42027/rdd-vs-dataframe-vs-sparksql.html)...

Gist is based on situation/scenario each one is right. there is no hard and fast rule to decide this. pls go through below..

### RDDs, DataFrames, and SparkSQL (infact 3 approaches not just 2):

At its core, Spark operates on the concept of Resilient Distributed Datasets, or RDD’s:

* Resilient - if data in memory is lost, it can be recreated
* Distributed - immutable distributed collection of objects in memory partitioned across many data nodes in a cluster
* Dataset - initial data can from from files, be created programmatically, from data in memory, or from another RDD

DataFrames API is a data abstraction framework that organizes your data into named columns:

* Create a schema for the data
* Conceptually equivalent to a table in a relational database
* Can be constructed from many sources including structured data files, tables in Hive, external databases, or existing RDDs
* Provides a relational view of the data for easy SQL like data manipulations and aggregations
* Under the hood, it is a row of RDD’s

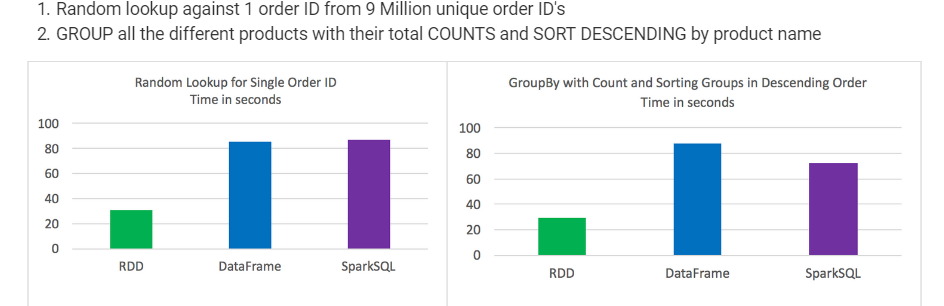
SparkSQL is a Spark module for structured data processing. You can interact with SparkSQL through:

* SQL
* DataFrames API
* Datasets API

### Test results:

* RDD’s outperformed DataFrames and SparkSQL for certain types of data processing
* DataFrames and SparkSQL performed almost about the same, although with analysis involving aggregation and sorting SparkSQL had a slight advantage
* Syntactically speaking, DataFrames and SparkSQL are much more intuitive than using RDD’s
* Took the best out of 3 for each test
* Times were consistent and not much variation between tests
* Jobs were run individually with no other jobs running

Random lookup against 1 order ID from 9 Million unique order ID's GROUP all the different products with their total COUNTS and SORT DESCENDING by product name

[](https://i.stack.imgur.com/NkrUy.png)

[share](https://stackoverflow.com/a/45438898)[improve this answer](https://stackoverflow.com/posts/45438898/edit)

# SQL at Scale with Apache Spark SQL and DataFrames — Concepts, Architecture and Examples

<https://towardsdatascience.com/sql-at-scale-with-apache-spark-sql-and-dataframes-concepts-architecture-and-examples-c567853a702f>

# Note

This article covers detailed concepts pertaining to Spark, SQL and DataFrames. Besides this we also cover a hands-on case study around working with SQL at scale using Spark SQL and DataFrames. If the article seems a bit overwhelming or really long to you at the first glance, feel free to check out the same in more bite-sized chunks in the following links at [opensource.com](https://opensource.com/)

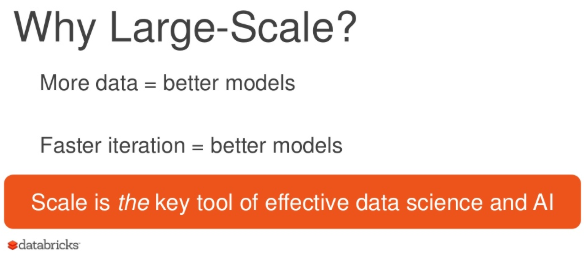
* [**Scaling relational databases with Apache Spark SQL and DataFrames**](https://opensource.com/article/19/3/sql-scale-apache-spark-sql-and-dataframes)
* [**How to use Spark SQL: A hands-on tutorial**](https://opensource.com/article/19/3/apache-spark-and-dataframes-tutorial)

I hope this helps you out on your own journey with Spark and SQL!

# Introduction

Relational Databases are here to stay, regardless of the hype as well as the advent of newer databases often popularly termed as ‘NoSQL’ databases. The simple reason is that these databases enforce essential structure, constraints and provide a nice declarative language to query data, which we love — SQL! However, scale has always been a problem with relational databases. Most enterprises now in the 21st century are loaded with rich data stores and repositories, and want to take maximum advantage of their ‘Big Data’ for actionable insights. Relational databases might be popular, but they don’t scale very well unless we invest in a proper Big Data management strategy. This involves thinking about potential data sources, data volume, constraints, schemas, ETL (extract-transform-load), access and querying patterns and much more!

https://miro.medium.com/max/60/1*dhRLxnSOeX3ri18g7z2zIA.png?q=20



Large Scale Data Science in Apache Spark: <https://www.slideshare.net/databricks/largescale-data-science-in-apache-spark-20>

This article will cover some excellent advances made for leveraging the power of relational databases, but ‘at scale,’ using some of the newer components from Apache Spark — Spark SQL and DataFrames. Most notably, we will cover the following topics.

## 1. Motivation and Challenges with Scaling Relational Databases

## 2. Understanding Spark SQL & DataFrames

* Goals
* Architecture and Features
* Performance

## 3. A real-world case study on Spark SQL with hands-on examples

Thus, we will be looking at the major challenges and motivation for people working so hard, and investing time in building new components in Apache Spark, so that we could perform SQL at scale. We will also understand major architecture, interfaces, features and performance benchmarks for Spark SQL and DataFrames. Lastly, but most importantly, we will cover a real-world case study on analyzing intrusion attacks based on [**KDD 99 Cup Data**](http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html) using Spark SQL and DataFrames by leveraging [Databricks Cloud Platform](http://databricks.com/" \t "_blank) for Spark!

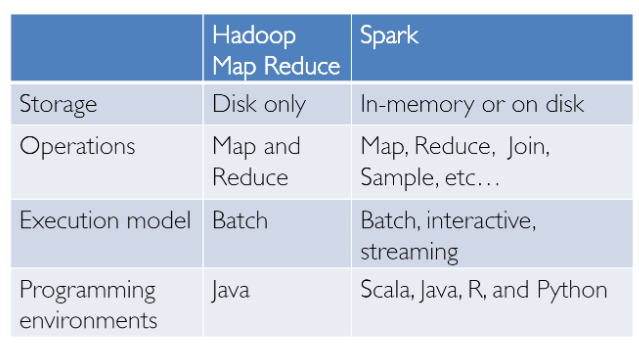
# Motivation and Challenges on Scaling Relational Databases for Big Data

Relational data stores are easy to build and query. Also, users as well as developers often prefer writing easy to interpret, declarative queries in a human-like readable language such as SQL. However, as data starts increasing in volume and variety, the relational approach does not scale well enough for building big data applications and analytical system. Following are some major challenges.

* Dealing with different types and sources of data which can be structured, semi-structured and unstructured.
* Building ETL pipelines to and from various data sources, which may lead to developing a lot of specific custom code, which increases technical debt over time.
* Capability to perform both traditional BI (Business Intelligence)-based analytics and advanced analytics (machine learning, statistical modeling etc.) the latter of which is definitely challenging to perform in relational systems.

Big Data Analytics is not something which was just invented yesterday! We have had success in this domain with Hadoop and the Map-Reduce paradigm. This was powerful, but often slow, and also gave users a low-level, procedural programming interface which required people to write a lot of code for even very simple data transformations. However, once Spark was released, it really revolutionized the way big data analytics was done with focus on in-memory computing, fault tolerance, high-level abstractions and ease of use.

https://miro.medium.com/max/60/1*L4a0rQFDwetRNCnBl0t9vA.png?q=20



From then, several frameworks and systems like Hive, Pig and Shark (which evolved into Spark SQL) provided rich relational interfaces and declarative querying mechanisms to Big Data stores. The challenge remained that these tools were either relational or procedural-based and we couldn’t have the best of both worlds.

However in the real-world, most data and analytical pipelines might involve a combination of relational and procedural code. Thus, forcing users to choose either one ends up complicating things and increasing user efforts in developing, building and maintaining different applications and systems. Apache Spark SQL builds on the previously mentioned SQL-on-Spark effort, called Shark. Instead of forcing users to pick between a relational or a procedural API, Spark SQL tries to enable users to seamlessly intermix the two and perform data querying, retrieval and analysis at scale on Big Data.

# Understanding Spark SQL & DataFrames

Spark SQL essentially tries to bridge the gap between the two models we mentioned previously — the relational and procedural models by two major components.

* Spark SQL provides a DataFrame API that can perform relational operations on both external data sources and Spark’s built-in distributed collections — at scale!
* To support the a wide variety of diverse data sources and algorithms in big data, Spark SQL introduces a novel extensible optimizer called Catalyst, which makes it easy to add data sources, optimization rules, and data types for advanced analytics such as machine learning.

Essentially, Spark SQL leverages the power of Spark to perform distributed, robust, in-memory computations at massive scale on Big Data. Spark SQL provides state-of-the-art SQL performance, and also maintains compatibility with all existing structures and components supported by **Apache Hive**(a popular Big Data Warehouse framework) including data formats, user-defined functions (UDFs) and the metastore. Besides this, it also helps in ingesting a wide variety of data formats from Big Data sources and enterprice data warehouses like JSON, Hive, Parquet and so on, and perform a combination of relational and procedural operations for more complex, advanced analytics.

## Goals

Let’s look at some of the interesting facts about Spark SQL, it’s usage, adoption and goals, some of which I will shamelessly once again copy from the excellent and original paper on [Relational Data Processing in Spark](http://people.csail.mit.edu/matei/papers/2015/sigmod_spark_sql.pdf). Spark SQL was first released in May 2014, and is perhaps now one of the most actively developed components in Spark. Apache Spark is definitely the most active open source project for big data processing, with hundreds of contributors. Besides being just an open-source project, Spark SQL has actually started seeing mainstream industry adoption! It has already been deployed in very large scale environments. An excellent case-study has been mentioned by Facebook where they talk about [**‘Apache Spark @Scale: A 60 TB+ production use case’**](https://code.fb.com/core-data/apache-spark-scale-a-60-tb-production-use-case)**—**Here, they were doing data preparation for entity ranking and their Hive jobs used to take several days and had many challenges, but they were able to successfully able to scale and increase performance using Spark. Do check out the interesting challenges they faced in this journey!

## [Apache Spark @Scale: A 60 TB+ production use case - Facebook Code](https://code.fb.com/core-data/apache-spark-scale-a-60-tb-production-use-case/?source=post_page-----c567853a702f----------------------)

### [Facebook often uses analytics for data-driven decision making. Over the past few years, user and product growth has…](https://code.fb.com/core-data/apache-spark-scale-a-60-tb-production-use-case/?source=post_page-----c567853a702f----------------------)

#### [code.fb.com](https://code.fb.com/core-data/apache-spark-scale-a-60-tb-production-use-case/?source=post_page-----c567853a702f----------------------)

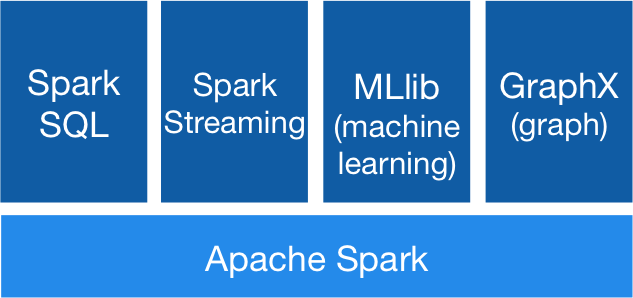
Another interesting fact is that 2/3rd of customers of Databricks Cloud (hosted service running Spark), use Spark SQL within other programming languages. We will also showcase a hands-on case study using Spark SQL on Databricks in this article. Stay tuned for that! The major goals for Spark SQL as defined by it’s creators are as follows.

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

## Architecture and Features

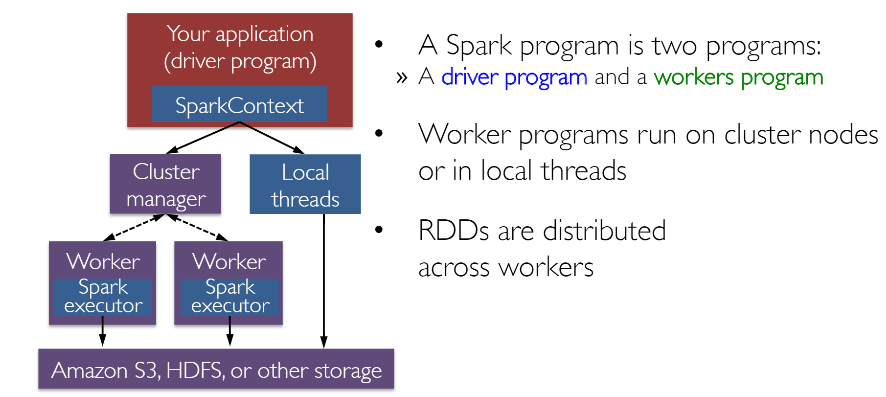
We will now take a look at the key features and architecture around Spark SQL and DataFrames. Some key concepts to keep in mind here would be around the Spark eco-system which has been constantly evolving over time.

https://miro.medium.com/max/60/0*Lq-LzsIpMWSVfTSU.png?q=20

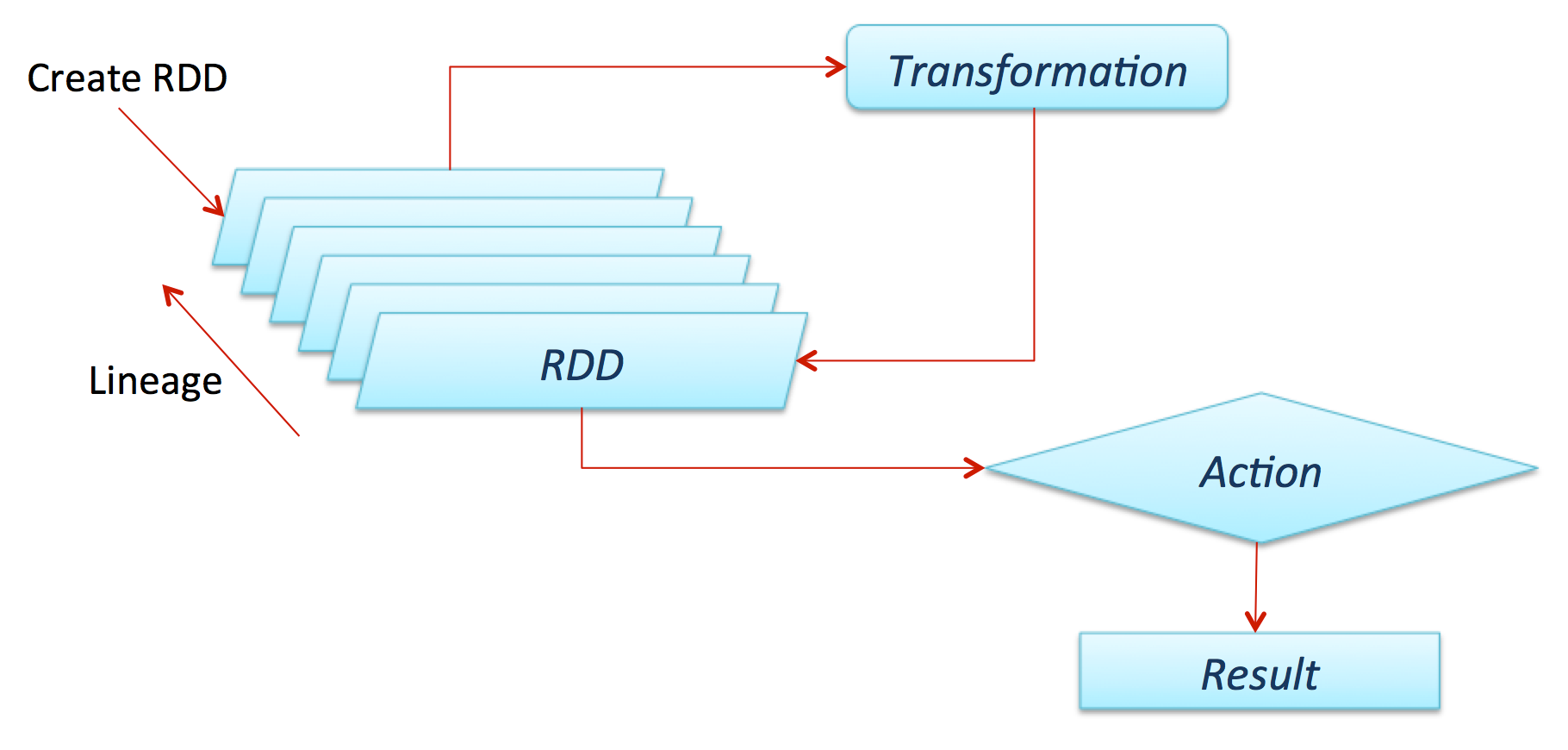


**RDDs** or “Resilient Distributed Datasets” is perhaps the biggest contributor behind all the success stories of Spark. It is basically a data structure or rather , a distributed memory abstraction to be more precise, that allows programmers to perform in-memory computations on large distributed clusters while retaining aspects like fault tolerance. You can also parallelize a lot of computations, transformations and track the whole lineage of transformations, which can help in efficiently recomputing lost data. Spark enthusiasts do read the excellent paper around RDDs, [‘Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing’](https://www2.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-82.pdf). Also, Spark works with the concept of drivers and workers as depicted in the following figure.

https://miro.medium.com/max/60/1*nGbJzH7pThsIXS8WIZP1iA.png?q=20



You can typically create an RDD by reading in data from files, databases, parallelizing existing collections or even transformations. Typically **transformations**are operations which can be used to transform the data into different aspects and dimensions depending on the way we want to wrangle and process the data. They are also lazily evaluated meaning that even if you define a transformation, the results are not computed till you apply an **action**which typically requires a result to be returned to the driver program (and it computed all applied transformations then!).



Shout out to fellow data scientist and friend [Favio Vázquez](https://towardsdatascience.com/@faviovazquez?source=post_header_lockup" \t "_blank) and his excellent [**article on Deep Learning With Apache Spark**](https://towardsdatascience.com/deep-learning-with-apache-spark-part-1-6d397c16abd) from which I got some excellent ideas and content including the preceding figure. Do check it out!

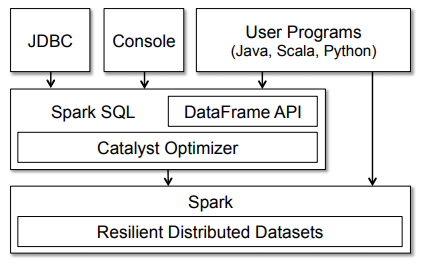
## [Deep Learning With Apache Spark — Part 1](https://towardsdatascience.com/deep-learning-with-apache-spark-part-1-6d397c16abd?source=post_page-----c567853a702f----------------------)

### [First part on a full discussion on how to do Distributed Deep Learning with Apache Spark. This part: What is Spark…](https://towardsdatascience.com/deep-learning-with-apache-spark-part-1-6d397c16abd?source=post_page-----c567853a702f----------------------)

#### [towardsdatascience.com](https://towardsdatascience.com/deep-learning-with-apache-spark-part-1-6d397c16abd?source=post_page-----c567853a702f----------------------)

Now that we know about the general architecture of how Spark works, let’s take a closer look into Spark SQL. Typically, **Spark SQL** runs as a library on top of Spark, as we have seen in the figure covering the Spark eco-system. The following figure gives a more detailed peek into the typical achitecture and interfaces of Spark SQL.

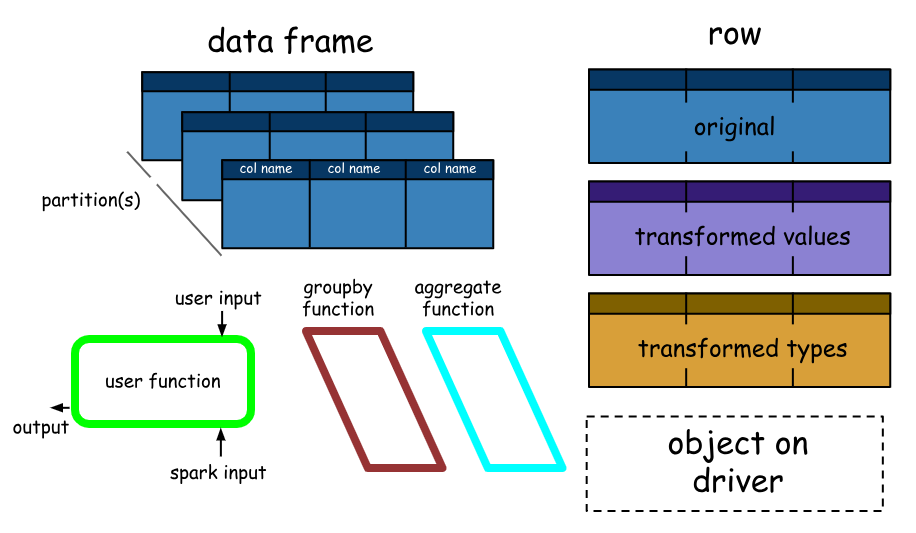
https://miro.medium.com/max/60/1*Y-CFnWiFSb4B7NYpQA-ARQ.png?q=20



The figure clearly shows us the various SQL interfaces, which can be accessed through JDBC/ODBC or through a command-line console, as well as the DataFrame API integrated into Spark’s supported programming languages (we will be using Python!). The DataFrame API is very powerful and allows users to **finally** intermix procedural and relational code! Advanced functions like UDFs (user defined functions) can also be exposed in SQL, which can be used by BI tools.

Spark DataFrames are very interesting and help us leverage the power of Spark SQL and combine its procedural paradigms as needed. A Spark DataFrame is basically a distributed collection of rows (Row types) with the same schema. It is basically a [**Spark Dataset**](https://databricks.com/blog/2016/01/04/introducing-apache-spark-datasets.html) organized into named columns. A point to note here is that [**Datasets**](https://databricks.com/glossary/what-are-datasets), are an extension of the DataFrame API that provides a type-safe, object-oriented programming interface. Hence, they are available only in Java and Scala and we will, therefore, be focusing on DataFrames.

https://miro.medium.com/max/60/1*bmI7xp82kNKLWAPtcZdl0A.png?q=20



Source: PySpark-Pictures — Jeffrey Thompson

A DataFrame is equivalent to a table in a relational database (but with more optimizations under the hood), and can also be manipulated in similar ways to the “native” distributed collections in Spark (RDDs). Spark DataFrames have some interesting properties, some of which are mentioned below.

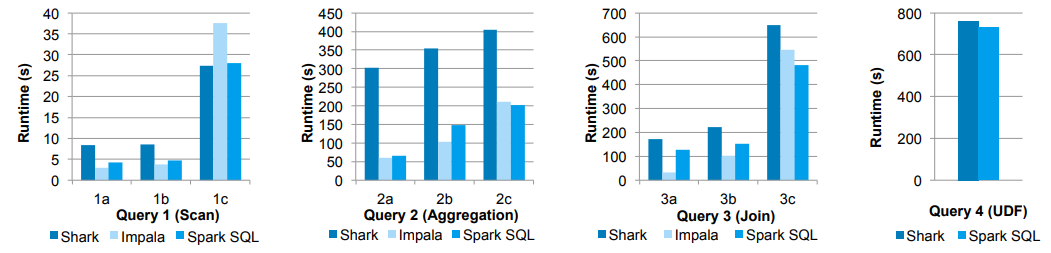
1. Unlike RDDs, DataFrames usually keep track of their schema and support various relational operations that lead to a more optimized execution.
2. DataFrames can be constructed from tables, just like existing Hive tables in your Big Data infrastructure, or even from existing RDDs.
3. DataFrames can be manipulated with direct SQL queries and also using the DataFrame DSL (domain-specific language), where we can use various relational operators and transformers such as where and groupBy
4. Also, each DataFrame can also be viewed as an RDD of Row objects, allowing users to call procedural Spark APIs such as map
5. Finally, a given, but a point to always remember, unlike traditional dataframe APIs (pandas), Spark DataFrames are lazy, in that each DataFrame object represents a logical plan to compute a dataset, but no execution occurs until the user calls a special “output operation” such as save.

This should give you enough perspective on Spark SQL, DataFrames, essential features, concepts, architecture and interfaces. Let’s wrap up this section by taking a look at performance benchmarks.

## Performance

Releasing a new feature without the right optimizations can be deadly, and the folks who built Spark did tons of performance tests and benchmarking! Let’s take a look at some interesting results. The first figure showcasing some results is depicted below.

https://miro.medium.com/max/60/1*-pnBiNr8OjNUUZPyJkva4A.png?q=20

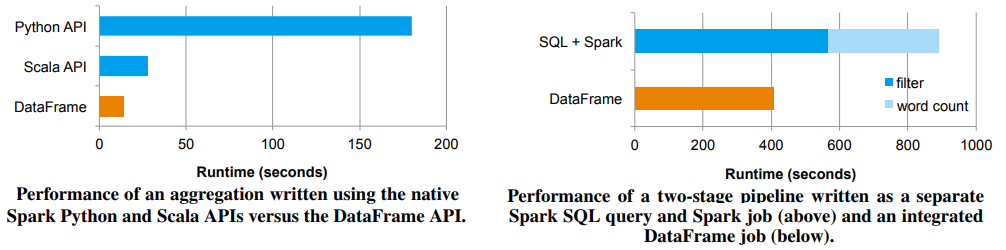


Performance of Shark, Impala and Spark SQL on Big Data benchmark queries

In these experiments, they compared the performance of Spark SQL against Shark and Impala using the AMPLab big data benchmark, which uses a web analytics workload developed by Pavlo et al. The benchmark contains four types of queries with different parameters performing scans, aggregation, joins and a UDF-based MapReduce job. The dataset was 110 GB of data after compression using the columnar Parquet format. We see that in all queries, Spark SQL is substantially faster than Shark, and generally competitive with Impala. The Catalyst optimizer is responsible for this, which reduces CPU overhead (we shall cover this briefly). This feature makes Spark SQL competitive with the C++ and LLVM-based Impala engine in many of these queries. The largest gap from Impala is in **query 3a** where Impala chooses a better join plan, because the selectivity of the queries makes one of the tables very small.

Following graphs show some more performance benchmarks for DataFrames and regular Spark APIs and Spark + SQL.

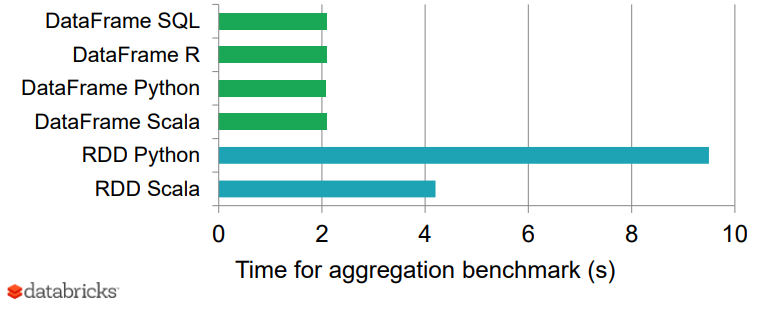
https://miro.medium.com/max/60/1*wRO9RQXfA-4XDyXSOfpoWA.png?q=20



Spark DataFrames vs RDDs and SQL

Finally, the following graph shows a nice benchmark result of DataFrames vs. RDDs in different languages, which gives an interesting perspective on how optimized DataFrames can be!

https://miro.medium.com/max/60/1*Kolwjg356xdqPv-8JmQGRQ.png?q=20

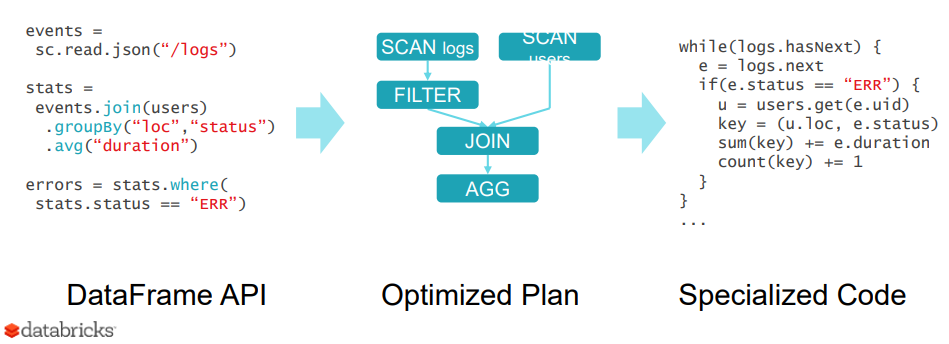


Comparing Spark DataFrames and RDDs

## Secret to Performance — The Catalyst Optimizer

Why is Spark SQL so fast and optimized? The reason is, because of a new extensible optimizer, Catalyst, based on functional programming constructs in Scala. While we won’t go into too extensive details about Catalyst here, it is worth a mention since it helps in optimizing DataFrame operations and queries.

https://miro.medium.com/max/60/1*PkHqRCgSACepcFZP5VM_dQ.png?q=20

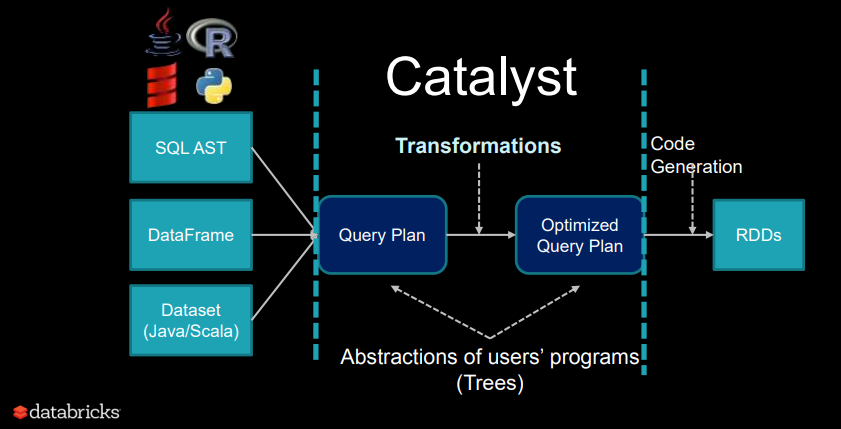


Catalyst’s extensible design has two purposes.

* Making it easy to add new optimization techniques and features to Spark SQL, especially to tackle diverse problems around ‘Big Data’, semi-structured data and advanced analytics.
* Ease of being able to extend the optimizer — for example, by adding data source-specific rules that can push filtering or aggregation into external storage systems, or support for new data types.

Catalyst supports both rule-based and cost-based optimization. While extensible optimizers have been proposed in the past, they have typically required a complex domain-specific language to specify rules. Usually, this leads to having a significant learning curve and maintenance burden. In contrast, Catalyst uses standard features of the Scala programming language, such as pattern-matching, to let developers use the full programming language while still making rules easy to specify.

https://miro.medium.com/max/60/1*XdltYoj1GlfRKiWZLooO3Q.png?q=20



At its core, Catalyst contains a general library for representing trees and applying rules to manipulate them. On top of this framework, it has libraries specific to relational query processing (e.g., expressions, logical query plans), and several sets of rules that handle different phases of query execution: analysis, logical optimization, physical planning, and code generation to compile parts of queries to Java bytecode. Interested in knowing more details about Catalyst and doing a deep-dive? You can check out [**an excellent article from Databricks!**](https://databricks.com/blog/2015/04/13/deep-dive-into-spark-sqls-catalyst-optimizer.html)

## [Deep Dive into Spark SQL's Catalyst Optimizer](https://databricks.com/blog/2015/04/13/deep-dive-into-spark-sqls-catalyst-optimizer.html?source=post_page-----c567853a702f----------------------)

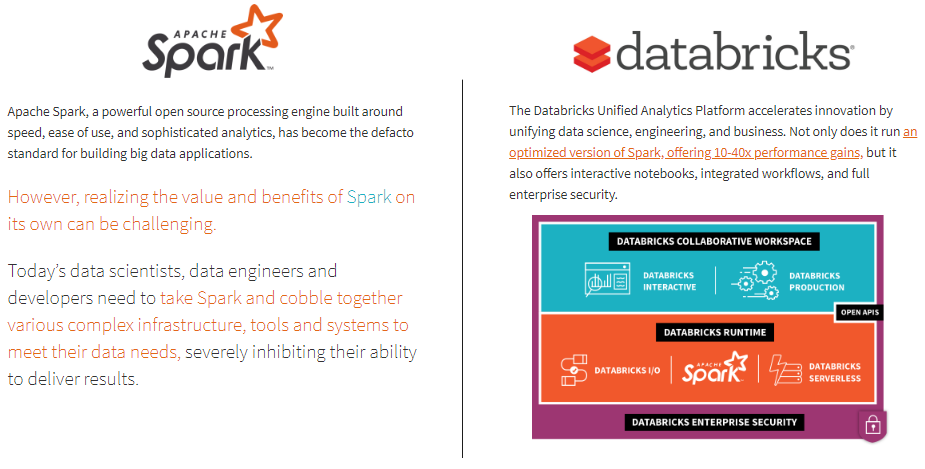
### [Spark SQL is one of the newest and most technically involved components of Spark. It powers both SQL queries and the…](https://databricks.com/blog/2015/04/13/deep-dive-into-spark-sqls-catalyst-optimizer.html?source=post_page-----c567853a702f----------------------)

#### [databricks.com](https://databricks.com/blog/2015/04/13/deep-dive-into-spark-sqls-catalyst-optimizer.html?source=post_page-----c567853a702f----------------------)

# Hands-on Case Study with Spark SQL

We will now do a simple tutorial based on a real-world dataset to look at how to use Spark SQL. We will be using Spark DataFrames but the focus will be more on using SQL. I will be covering a detailed discussion around Spark DataFrames and common operations in a separate article. I love using cloud services for my machine learning, deep learning and even Big Data Analytics needs. Instead of painfully setting up your own Spark cluster, use one of the best in the Cloud! We will be using the Databricks Platform for our Spark needs! Databricks is a company founded by the creators of Apache Spark, that aims to help clients with cloud-based big data processing using Spark.

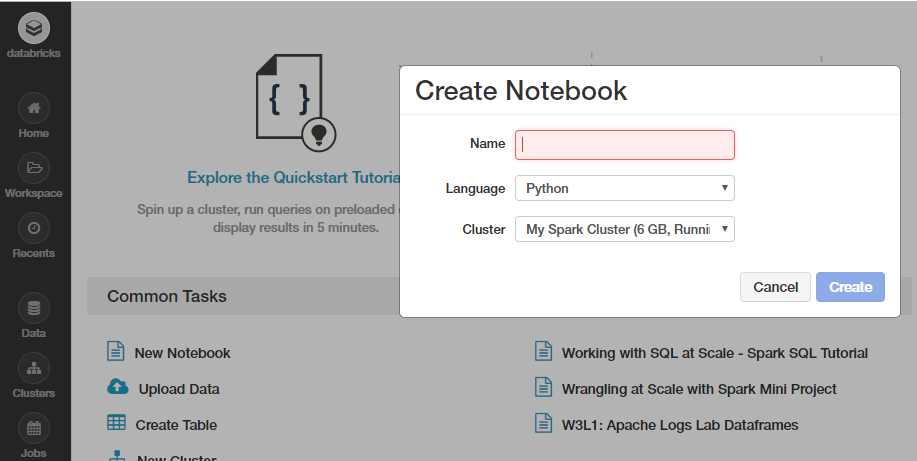
https://miro.medium.com/max/60/1*TXw24jTUHTEDlQHe7ZGFxg.png?q=20



The simplest way (and free of charge) is to go to the [**Try Databricks page**](https://databricks.com/try-databricks) and [sign up](https://databricks.com/signup#signup/community) for an account using the [**community edition**](https://databricks.com/signup#signup/community) where you get a cloud-based cluster, which is a single node cluster with 6 GB and unlimited notebooks, not bad for a free version! I definitely recommend using the Databricks Platform if you have serious needs for analyzing Big Data!

Let’s get started with our case study now, feel free to create a new notebook from your home screen in Databricks or your own Spark cluster as depicted in the following snapshot.

https://miro.medium.com/max/60/1*aOX5kmcfy0RwLQWyz0bNTw.png?q=20



You can also import my notebook containing the entire tutorial but do run every cell and play around with it and explore instead of just reading through it. Unsure of how to use Spark on Databricks? Follow [**this short but useful tutorial**](https://databricks.com/spark/getting-started-with-apache-spark) and get started today!

## [Apache Spark Tutorial: Getting Started with Apache Spark Tutorial](https://databricks.com/spark/getting-started-with-apache-spark?source=post_page-----c567853a702f----------------------)

### [Apache Spark is a powerful open-source processing engine built around speed, ease of use, and sophisticated analytics…](https://databricks.com/spark/getting-started-with-apache-spark?source=post_page-----c567853a702f----------------------)

#### [databricks.com](https://databricks.com/spark/getting-started-with-apache-spark?source=post_page-----c567853a702f----------------------)

This tutorial will familiarize you with essential Spark capabilities to deal with structured data often obtained from databases or flat files. We will explore typical ways of querying and aggregating relational data by leveraging concepts of DataFrames and SQL using Spark. We will work on an interesting dataset from the [**KDD Cup 1999**](http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html) and try to query the data using high level abstrations like the dataframe which has already been a hit in popular data analysis tools like R and Python. We will also look at how easy it is to build data queries using the SQL language, which you have learnt, and retrieve insightful information from our data. This also happens at scale without us having to do a lot more since Spark distributes these data structures efficiently in the back-end, which makes our queries scalable and as efficient as possible. We start by loading some basic dependencies.

**import pandas as pd  
import matplotlib.pyplot as plt  
plt.style.use('fivethirtyeight')**

## Data Retrieval

We will use data from the [**KDD Cup 1999**](http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html), which is the data set used for The Third International Knowledge Discovery and Data Mining Tools Competition, which was held in conjunction with KDD-99 The Fifth International Conference on Knowledge Discovery and Data Mining. The competition task was to build a network intrusion detector, a predictive model capable of distinguishing between **bad connections**, called intrusions or attacks, and **good, normal connections**. This database contains a standard set of data to be audited, which includes a wide variety of intrusions simulated in a military network environment.

We will be using the reduced dataset **kddcup.data\_10\_percent.gz** containing nearly half a million nework interactions, since we would be downloading this Gzip file from the web locally, and then work on the same. If you have a good, stable internet connection, feel free to download and work with the full dataset available as **kddcup.data.gz.**

## Working with data from the web

Dealing with datasets retrieved from the web can be a bit tricky in Databricks. Fortunately, we have some excellent utility packages like **dbutils** which help in making our job easier. Let's take a quick look at some essential functions for this module.

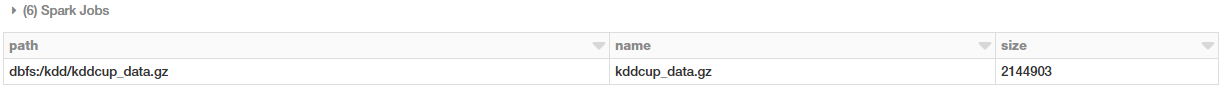
**dbutils.help()**This module provides various utilities for users to interact with the rest of Databricks.**fs: DbfsUtils** -> Manipulates the Databricks filesystem (DBFS) from the console  
**meta: MetaUtils** -> Methods to hook into the compiler (EXPERIMENTAL)  
**notebook: NotebookUtils** -> Utilities for the control flow of a notebook (EXPERIMENTAL)  
**preview: Preview** -> Utilities under preview category  
**secrets: SecretUtils** -> Provides utilities for leveraging secrets within notebooks  
**widgets: WidgetsUtils** -> Methods to create and get bound value of input widgets inside notebooks

## Retrieve and store data in Databricks

We will now leverage the python **urllib** library to extract the KDD Cup 99 data from their web repository, store it in a temporary location, and then move it to the Databricks filesystem, which can enable easy access to this data for analysis

***Note:****If you skip this step and download the data directly, you may end up getting a****InvalidInputException: Input path does not exist****error*

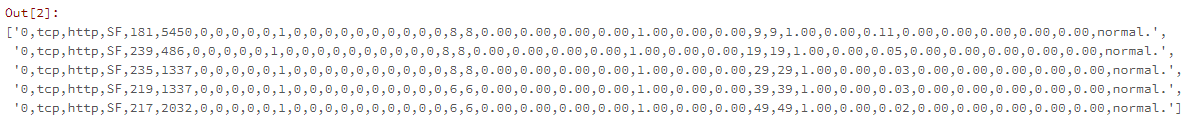
https://miro.medium.com/max/60/1*YgaPjGDEzsU4D0l70JcFKA.png?q=20



## Building the KDD Dataset

Now that we have our data stored in the Databricks filesystem, let’s load up our data from the disk into Spark’s traditional abstracted data structure, the [Resilient Distributed Dataset (RDD)](https://spark.apache.org/docs/latest/rdd-programming-guide.html#resilient-distributed-datasets-rdds)

https://miro.medium.com/max/60/1*Xzf1mPLwzAMA-LAr3CPc0Q.png?q=20



You can also verify the type of data structure of our data (RDD) using the following code.

**type(raw\_rdd)**

https://miro.medium.com/max/60/1*8uEM9T5F4UYW_OPPudomlw.png?q=20

https://miro.medium.com/max/196/1*8uEM9T5F4UYW_OPPudomlw.png

## Building a Spark DataFrame on our Data

A Spark DataFrame is an interesting data structure representing a distributed collecion of data. Typically the entry point into all SQL functionality in Spark is the **SQLContext** class. To create a basic instance of this call, all we need is a **SparkContext** reference. In Databricks, this global context object is available as **sc** for this purpose.

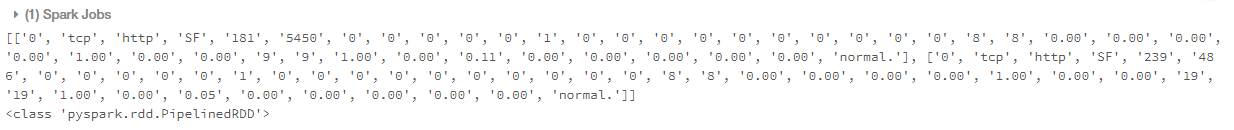
https://miro.medium.com/max/60/1*NmlSRpGRPWc9Z6VTVLbl5w.png?q=20

https://miro.medium.com/max/465/1*NmlSRpGRPWc9Z6VTVLbl5w.png

## Splitting the CSV data

Each entry in our RDD is a comma-separated line of data which we first need to split before we can parse and build our dataframe.

https://miro.medium.com/max/60/1*2w6d5tOsYj-_gx7V0RjXeA.png?q=20



## Check the total number of features (columns)

We can use the following code to check the total number of potential columns in our dataset.

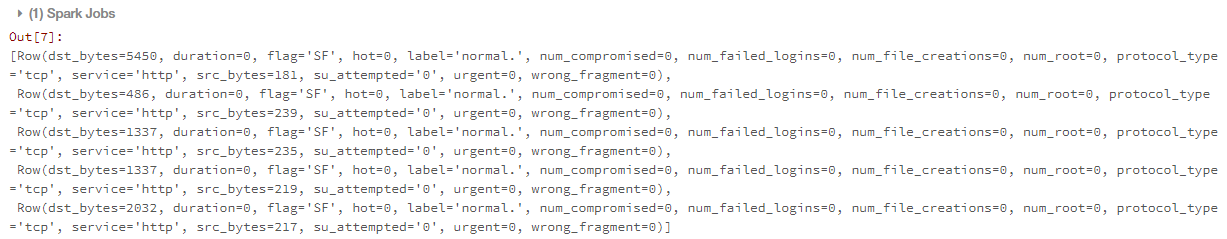
**len(csv\_rdd.take(1)[0])Out[57]: 42**

## Data Understanding and Parsing

The KDD 99 Cup data consists of different attributes captured from connection data. The full list of attributes in the data can be obtained [**here**](http://kdd.ics.uci.edu/databases/kddcup99/kddcup.names) and further details pertaining to the description for each attribute\column can be found [**here**](http://kdd.ics.uci.edu/databases/kddcup99/task.html). We will just be using some specific columns from the dataset, the details of which are specified as follows.

We will be extracting the following columns based on their positions in each datapoint (row) and build a new RDD as follows.

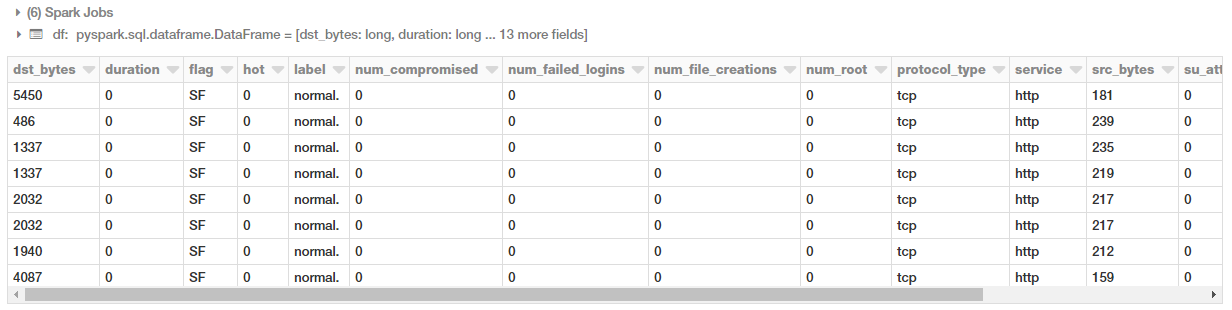
https://miro.medium.com/max/60/1*z4ahp-EkGDxouqRUQUvxEw.png?q=20



## Constructing the DataFrame

Now that our data is neatly parsed and formatted, let’s build our DataFrame!

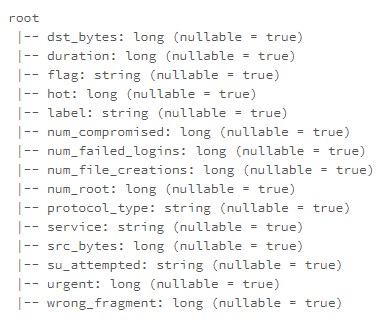
https://miro.medium.com/max/60/1*duH2ONp9GTM6VXz0h6ttJg.png?q=20



You can also now check out the schema of our dataframe using the following code.

**df.printSchema()**

https://miro.medium.com/max/60/1*xUq7CKHtT-NDPCxdIh4U-w.png?q=20



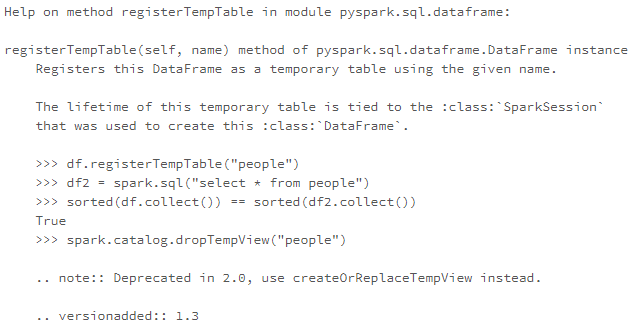
## Building a temporary table

We can leverage the registerTempTable() function to build a temporaty table to run SQL commands on our DataFrame at scale! A point to remember is that the lifetime of this temp table is tied to the session. It creates an in-memory table that is scoped to the cluster in which it was created. The data is stored using Hive's highly-optimized, in-memory columnar format.

You can also check out saveAsTable() which creates a permanent, physical table stored in S3 using the Parquet format. This table is accessible to all clusters. The table metadata including the location of the file(s) is stored within the Hive metastore.

**help(df.registerTempTable)**

https://miro.medium.com/max/60/1*YdcU1hRWBR_pvnvu-mdScQ.png?q=20



**df.registerTempTable("connections")**

# Executing SQL at Scale

Let’s look at a few examples of how we can run SQL queries on our table based off our dataframe. We will start with some simple queries and then look at aggregations, filters, sorting, sub-queries and pivots in this tutorial.

## Connections based on the protocol type

Let’s look at how we can get the total number of connections based on the type of connectivity protocol. First, we will get this information using normal DataFrame DSL syntax to perform aggregations.

https://miro.medium.com/max/60/1*ngrz3HQHLWNrFFcjKIBfzw.png?q=20



Can we also use SQL to perform the same aggregation? Yes, we can leverage the table we built earlier for this!

https://miro.medium.com/max/60/1*-vAaaUzMlsC5ew74oQd2NQ.png?q=20

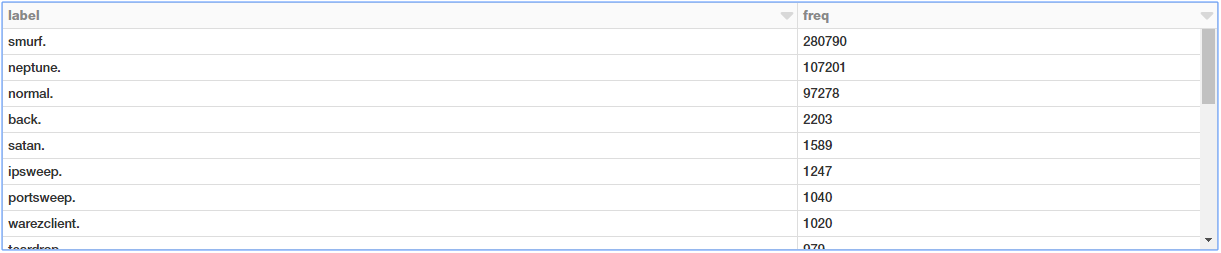


You can clearly see, that you get the same results and you do not need to worry about your background infrastructure or how the code is executed. Just write simple SQL!

## Connections based on good or bad (attack types) signatures

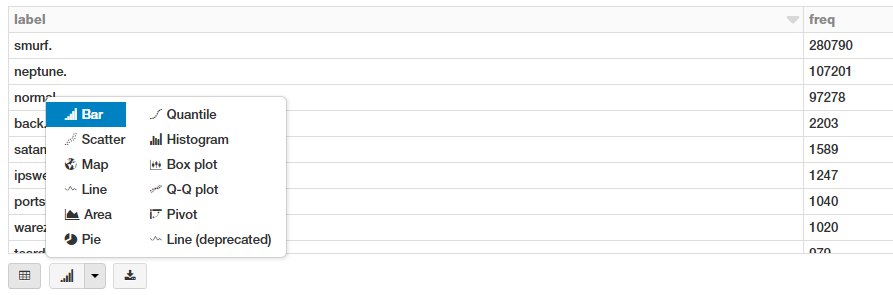
We will now run a simple aggregation to check the total number of connections based on good (normal) or bad (intrusion attacks) types.

https://miro.medium.com/max/60/1*Fz-4P-V8DosZm2h3a_p1lw.png?q=20



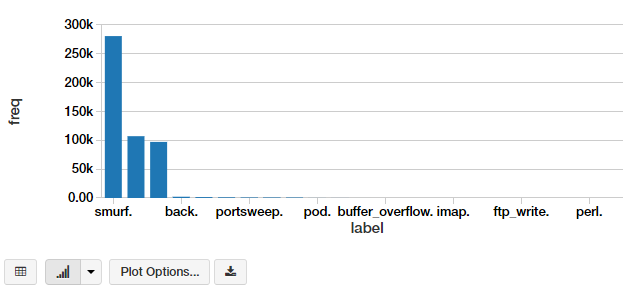
We have a lot of different attack types. We can visualize this in the form of a bar chart. The simplest way is to use the excellent interface options in the Databricks notebook itself!

https://miro.medium.com/max/60/1*MWtgLR6H4siUB1Ugc8sqog.png?q=20



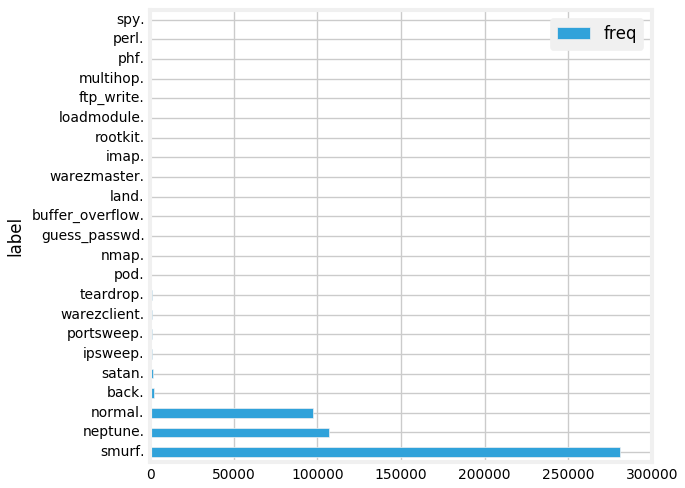
This gives us the following nice looking bar chart! Which you can customize further by clicking on **Plot Options** as needed.

https://miro.medium.com/max/60/1*vL5_tmej7dfh9o5ZDDtHqA.png?q=20



Another way is to write the code yourself to do it. You can extract the aggregated data as a pandas DataFrame and then plot it as a regular bar chart.

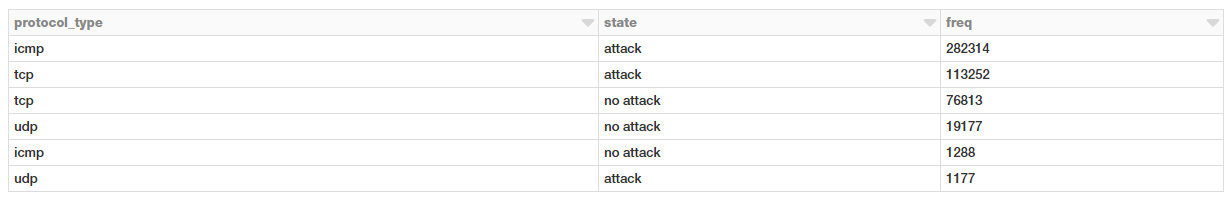
https://miro.medium.com/max/60/1*mmlDBC5oLk4KBjshr6Bt7Q.png?q=20



# Connections based on protocols and attacks

Let’s look at which protocols are most vulnerable to attacks now based on the following SQL query.

https://miro.medium.com/max/60/1*C3aovKsvavE4gXLAcLRXiw.png?q=20

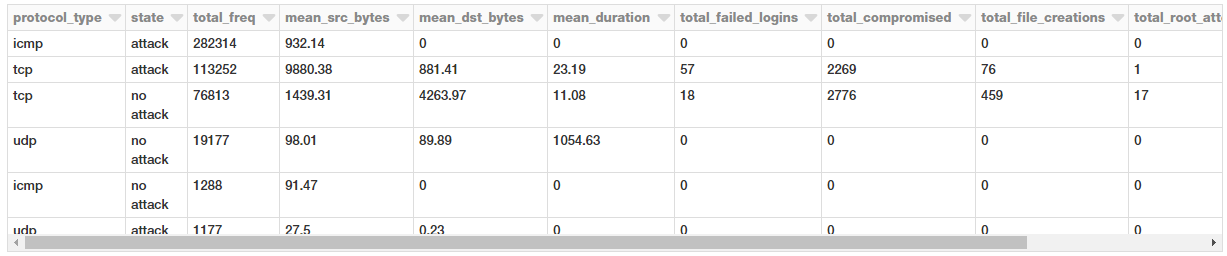


Well, looks like ICMP connections followed by TCP connections have had the maximum attacks!

## Connection stats based on protocols and attacks

Let’s take a look at some statistical measures pertaining to these protocols and attacks for our connection requests.

https://miro.medium.com/max/60/1*Py3sXd9XHLa_0cqG1tz2jg.png?q=20

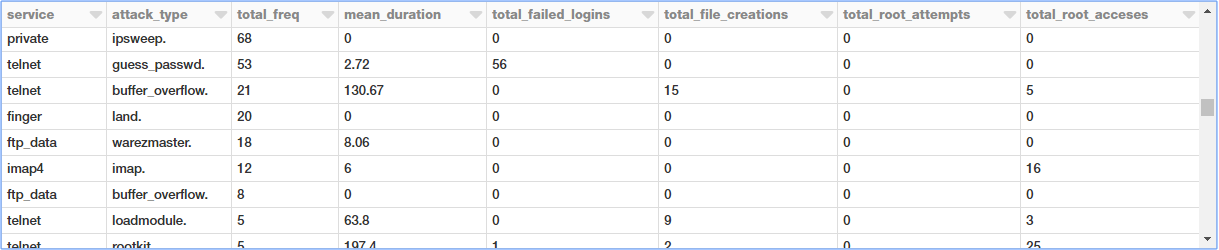


Looks like average amount of data being transmitted in TCP requests are much higher which is not surprising. Interestingly, attacks have a much higher average payload of data being transmitted from the source to the destination.

## Filtering connection stats based on the TCP protocol by service and attack type

Let’s take a closer look at TCP attacks given that we have more relevant data and statistics for the same. We will now aggregate different types of TCP attacks based on service, attack type and observe different metrics.

https://miro.medium.com/max/60/1*IRX3qOE4Llla2MYtVnmk2A.png?q=20

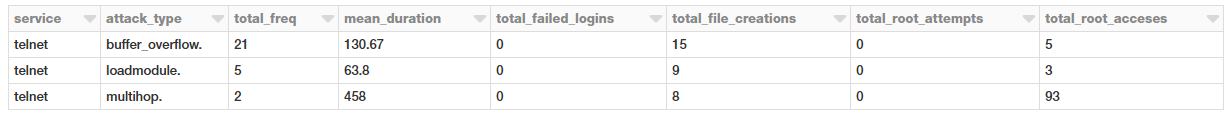


There are a lot of attack types and the preceding output shows a specific section of the same.

## Filtering connection stats based on the TCP protocol by service and attack type

We will now filter some of these attack types by imposing some constraints based on duration, file creations, root accesses in our query.

https://miro.medium.com/max/60/1*wc0ExfXKG60LJKmvQxawfA.png?q=20

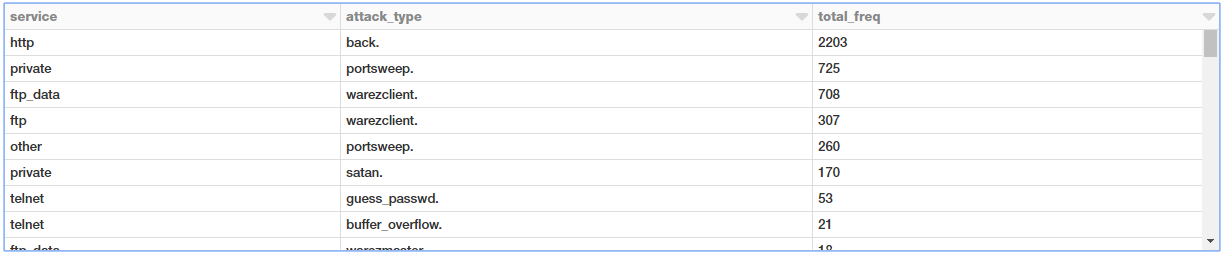


Interesting to see [multihop attacks](https://attack.mitre.org/techniques/T1188/" \t "_blank) being able to get root accesses to the destination hosts!

## Subqueries to filter TCP attack types based on service

Let’s try to get all the TCP attacks based on service and attack type such that the overall mean duration of these attacks is greater than zero (**> 0**). For this, we can do an inner query with all aggregation statistics and then extract the relevant queries and apply a mean duration filter in the outer query as shown below.

https://miro.medium.com/max/60/1*3T5av-peogELACxu_d8Qww.png?q=20

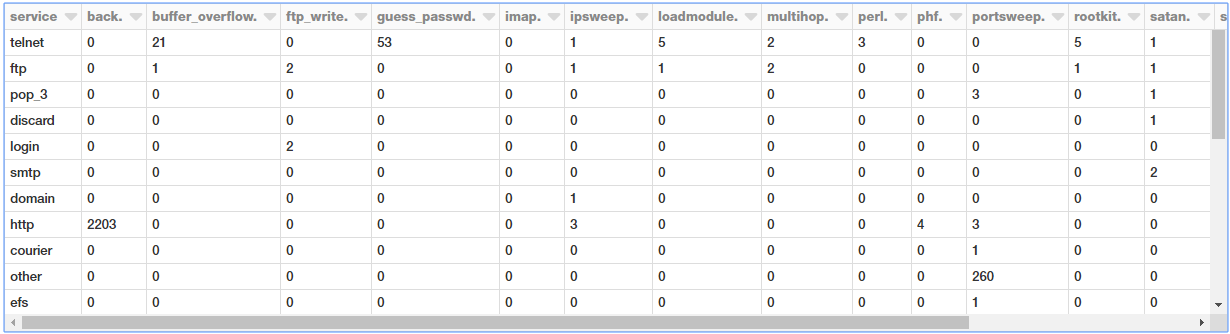


This is nice! Now an interesting way to also view this data is to use a pivot table where one attribute represents rows, and another one represents columns. Let’s see if we can leverage Spark DataFrames to do this!

## Building a Pivot Table from Aggregated Data

Here, we will build upon the previous DataFrame object we obtained where we aggregated attacks based on type and service. For this, we can leverage the power of Spark DataFrames and the DataFrame DSL.

https://miro.medium.com/max/60/1*4U4EcnuMy5G6znSYJqcAyw.png?q=20



We get a nice neat pivot table showing all the occurrences based on service and attack type!

# Next Steps

I would encourage you to go out and play with Spark SQL and DataFrames, you can even [**import my notebook**](https://databricks-prod-cloudfront.cloud.databricks.com/public/4027ec902e239c93eaaa8714f173bcfc/3137082781873852/3704545280501166/1264763342038607/latest.html) and play with it yourself in your own account.

Feel free to refer to [**my GitHub repository**](https://github.com/dipanjanS/data_science_for_all/tree/master/tds_spark_sql_intro) also for all the code and notebooks used in this article. What we didn’t cover here includes the following.

* Joins
* Window Functions
* Detailed operations and Transformations of Spark DataFrames

There are plenty of articles\tutorials available online so I would recommend you to check them out. Some useful resources for you to check out include, the complete guide to [**Spark SQL from Databricks**](https://docs.databricks.com/spark/latest/spark-sql/index.html).

## [SQL Guide - Databricks Documentation](https://docs.databricks.com/spark/latest/spark-sql/index.html?source=post_page-----c567853a702f----------------------)

### [View Azure Databricks documentation Azure docs](https://docs.databricks.com/spark/latest/spark-sql/index.html?source=post_page-----c567853a702f----------------------)

#### [docs.databricks.com](https://docs.databricks.com/spark/latest/spark-sql/index.html?source=post_page-----c567853a702f----------------------)

Thinking of working with JSON data but unsure of using Spark SQL. They support it! Check out this excellent guide to [**JSON support in Spark SQL**](https://databricks.com/blog/2015/02/02/an-introduction-to-json-support-in-spark-sql.html).

## [An introduction to JSON support in Spark SQL](https://databricks.com/blog/2015/02/02/an-introduction-to-json-support-in-spark-sql.html?source=post_page-----c567853a702f----------------------)

### [In this blog post, we introduce Spark SQL's JSON support, a feature we have been working on at Databricks to make it…](https://databricks.com/blog/2015/02/02/an-introduction-to-json-support-in-spark-sql.html?source=post_page-----c567853a702f----------------------)

#### [databricks.com](https://databricks.com/blog/2015/02/02/an-introduction-to-json-support-in-spark-sql.html?source=post_page-----c567853a702f----------------------)

Interested in advanced concepts like window functions and ranks in SQL? Check out this excellent article on [**Window Functions in Spark SQL**](https://databricks.com/blog/2015/07/15/introducing-window-functions-in-spark-sql.html)

## [Introducing Window Functions in Spark SQL](https://databricks.com/blog/2015/07/15/introducing-window-functions-in-spark-sql.html?source=post_page-----c567853a702f----------------------)

### [In this blog post, we introduce the new window function feature that was added in Apache Spark 1.4. Window functions…](https://databricks.com/blog/2015/07/15/introducing-window-functions-in-spark-sql.html?source=post_page-----c567853a702f----------------------)

#### [databricks.com](https://databricks.com/blog/2015/07/15/introducing-window-functions-in-spark-sql.html?source=post_page-----c567853a702f----------------------)

I will also write a followup article to this covering some of these concepts in an intuitive way which should be easy for you to understand. Stay tuned!

You can directly access my notebook on Databricks by going to [**this link**](https://databricks-prod-cloudfront.cloud.databricks.com/public/4027ec902e239c93eaaa8714f173bcfc/3137082781873852/3704545280501166/1264763342038607/latest.html) and import it directly and play around with it

## [Working with SQL at Scale - Spark SQL Tutorial - Databricks](https://databricks-prod-cloudfront.cloud.databricks.com/public/4027ec902e239c93eaaa8714f173bcfc/3137082781873852/3704545280501166/1264763342038607/latest.html?source=post_page-----c567853a702f----------------------)

### [By Dipanjan (DJ) Sarkar](https://databricks-prod-cloudfront.cloud.databricks.com/public/4027ec902e239c93eaaa8714f173bcfc/3137082781873852/3704545280501166/1264763342038607/latest.html?source=post_page-----c567853a702f----------------------)

#### [databricks-prod-cloudfront.cloud.databricks.com](https://databricks-prod-cloudfront.cloud.databricks.com/public/4027ec902e239c93eaaa8714f173bcfc/3137082781873852/3704545280501166/1264763342038607/latest.html?source=post_page-----c567853a702f----------------------)

All the code and resources for this tutorial is available on [**my GitHub**](https://github.com/dipanjanS/data_science_for_all/tree/master/tds_spark_sql_intro)

## [dipanjanS/data\_science\_for\_all](https://github.com/dipanjanS/data_science_for_all/tree/master/tds_spark_sql_intro?source=post_page-----c567853a702f----------------------)

### [Code and resources for my blog and articles to share Data Science and AI knowledge and learnings with everyone …](https://github.com/dipanjanS/data_science_for_all/tree/master/tds_spark_sql_intro?source=post_page-----c567853a702f----------------------)

#### [github.com](https://github.com/dipanjanS/data_science_for_all/tree/master/tds_spark_sql_intro?source=post_page-----c567853a702f----------------------)

You can also access my tutorial as a **[Jupyter Notebook](http://nbviewer.jupyter.org/github/dipanjanS/data_science_for_all/blob/master/tds_spark_sql_intro/Working%20with%20SQL%20at%20Scale%20-%20Spark%20SQL%20Tutorial.ipynb" \t "_blank)** in case you want to use it offline.

## [Jupyter Notebook Viewer](http://nbviewer.jupyter.org/github/dipanjanS/data_science_for_all/blob/master/tds_spark_sql_intro/Working%20with%20SQL%20at%20Scale%20-%20Spark%20SQL%20Tutorial.ipynb?source=post_page-----c567853a702f----------------------)

### [Spark SQL brings native support for SQL to Spark and streamlines the process of querying data stored both in RDDs…](http://nbviewer.jupyter.org/github/dipanjanS/data_science_for_all/blob/master/tds_spark_sql_intro/Working%20with%20SQL%20at%20Scale%20-%20Spark%20SQL%20Tutorial.ipynb?source=post_page-----c567853a702f----------------------)

#### [nbviewer.jupyter.org](http://nbviewer.jupyter.org/github/dipanjanS/data_science_for_all/blob/master/tds_spark_sql_intro/Working%20with%20SQL%20at%20Scale%20-%20Spark%20SQL%20Tutorial.ipynb?source=post_page-----c567853a702f----------------------)

Have feedback for me? Or interested in working with me on research, data science, artificial intelligence or even publishing an article on [**TDS**](https://towardsdatascience.com/)? You can reach out to me on [**LinkedIn**](https://www.linkedin.com/in/dipanzan/)**.**

Comparison between Spark RDD vs DataFrame  
  
1. Release of DataSets  
RDD – Basically, Spark 1.0 release introduced an RDD API.           
  
DataFrame-  Basically, Spark 1.3 release introduced a preview of the new dataset, that is dataFrame.  
  
2. Data Formats  
RDD- Through RDD, we can process structured as well as unstructured data. But, in RDD user need to specify the schema of ingested data, RDD cannot infer its own.  
  
DataFrame- In data frame data is organized into named columns. Through dataframe, we can process structured and unstructured data efficiently. It also allows Spark to manage schema.  
  
3. Data Representations  
RDD- It is a distributed collection of data elements. That is spread across many machines over the cluster, they are a set of Scala or Java objects representing data.  
  
DataFrame-  As we discussed above, in a data frame data is organized into named columns. Basically, it is as same as a table in a relational database.  
  
4. Compile- Time Type Safety  
RDD-  RDD Supports object-oriented programming style with compile-time type safety.  
  
DataFrame- If we try to access any column which is not present in the table, then an attribute error may occur at runtime. Dataframe will not support compile-time type safety in such case.  
  
5. Immutability and Interoperability  
RDD- RDDs are immutable in nature. That means we can not change anything about RDDs. We can create it through some transformation on existing partitions. Due to immutability, all the computations performed are consistent in nature. If RDD is in tabular format, we can move from RDD to dataframe by to() method. We can also do the reverse by the .rdd method.  
  
DataFrame-  One cannot regenerate a domain object, after transforming into dataframe. By using the example, if we generate one test data frame from tested then, we can not recover the original RDD again of the test class.  
  
6. Data Sources API  
RDD- From any data source, e.g. text files, a database via JDBC, etc. , an  RDD can come. Also, can easily handle data with no predefined structure.  
  
DataFrame- In different formats, data source API allows data processing, such as AVRO, CSV, JSON, and storage system HDFS, HIVE tables, MySQL.  
  
7. Optimization  
  
RDD-  There was no provision for optimization engine in RDD. On the basis of its attributes, developers optimise each RDD.  
  
DataFrame- By using Catalyst Optimizer, optimization takes place in dataframes. In 4 phases, dataframes use catalyst tree transformation framework  
  
By Analysis  
With logical plan optimization  
By physical planning  
With code generation to compile parts of the query to java bytecode.  
  
8. Serialization  
RDD-  Spark uses java serialization, whenever it needs to distribute data over a cluster. Serializing individual Scala and Java objects are expensive. It also requires sending both data and structure between nodes.  
  
DataFrame- In dataframe, we can serialize data into off-heap storage in binary format. Afterwards, it performs transformations on this off-heap memory, as spark understands schema. Moreover, to encode the data, there is no need to use java serialization.  
  
9. Efficiency/Memory use  
RDD-  When serialization executes individually on a java and scala object, efficiency decreases. It also takes lots of time.  
  
DataFrame- Use of off-heap memory for serialization reduces the overhead also generates, bytecode. So that, many operations can be performed on that serialized data. Basically, there is no need of deserialization for small operations.  
  
10. Lazy Evaluation  
RDD-  Spark does not compute their result right away, it evaluates RDDs lazily. Apart from it, Spark memorizes the transformation applied to some base data set. Moreover, When an action needs, a result sent to driver program for computation.  
  
DataFrame- Similarly, computation happens only when action appears as Spark evaluates dataframe lazily.  
  
11. Language Support  
RDD- APIs for RDD is available in 4 languages, such as Java, Scala, Python, and R. As a result, this feature provides flexibility to the developers.  
  
DataFrame- As similar as RDD, it also has APIs in same 4 languages, such as Java, Scala, Python, and R.  
  
12. Schema Projection  
RDD- Since RDD APIs, use schema projection explicitly. Therefore, a user needs to define the schema manually.  
  
DataFrame- In dataframe, there is no need to specify a schema. Generally, it discovers schema automatically.  
  
13. Aggregation  
RDD- While performing simple grouping and aggregation operations RDD API is slower.  
  
DataFrame- In performing exploratory analysis, creating aggregated statistics on data, dataframes are faster.  
  
14. Usage  
RDD-  When you want low-level transformation and actions, we use RDDs. Also, when we need high-level abstractions we use RDDs.  
  
DataFrame-  We use dataframe when we need a high level of abstraction and for unstructured data, such as media streams or streams of text.

<https://databricks.com/blog/2015/04/13/deep-dive-into-spark-sqls-catalyst-optimizer.html>