SQL at Scale with Spark SQL and DataFrames

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 also includes a cost-based optimizer, columnar storage, and code generation to make queries fast. At the same time, it scales to thousands of nodes and multi-hour queries using the Spark engine, which provides full mid-query fault tolerance, without having to worry about using a different engine for historical data.

For getting a deeper perspective into the background, concepts, architecture of Spark SQL and DataFrames you can check out the original article, _'SQL at Scale with Apache Spark SQL and DataFrames-Concepts, Architecture and Examples'__

This tutorial will familiarize you with essential Spark capabilities to deal with structured data typically 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 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 backend which makes our queries scalable and as efficient as possible.

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

```
import pyspark
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("nik").getOrCreate()

from pyspark import SparkContext
sc = SparkContext.getOrCreate()
```

Data Retrieval

We will use data from the KDD Cup 1999, 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 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`

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 RDD

```
data_file = "kddcup_data.gz"
raw_rdd = sc.textFile(data_file)
```

```
print(raw_rdd)
```

```
kddcup_data.gz MapPartitionsRDD[1] at textFile at NativeMethodAccessorImpl.java:0
```

```
raw_rdd.collect()
```

```
'0,tcp,http,SF,246,1718,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,10,12,0.00,0.00,0.00,0.00,1.00,0.00,0.25,255,255,1.00,0.00,0.00,0.00,0.00
'0,tcp,http,SF,218,1484,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,7,7,0.00,0.00,0.00,0.00,1.00,0.00,0.00,255,255,1.00,0.00,0.00,0.00,0.00,0
'0,tcp,http,SF,222,1651,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,4,4,0.00,0.00,0.00,0.00,1.00,0.00,0.00,255,255,1.00,0.00,0.00,0.00,0.00,0
'0,tcp,http,SF,215,1108,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,1,1,0.00,0.00,0.00,0.00,1.00,0.00,0.00,255,255,1.00,0.00,0.00,0.00,0.00,0
'0,tcp,http,SF,216,1645,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,3,8,0.00,0.00,0.00,0.00,1.00,0.00,0.50,255,255,1.00,0.00,0.00,0.00,0.00,0
'0,tcp,http,SF,213,1694,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,13,18,0.00,0.00,0.00,0.00,1.00,0.00,0.22,255,255,1.00,0.00,0.00,0.00,0.00
'0,tcp,http,SF,243,1719,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,6,24,0.00,0.00,0.00,0.00,1.00,0.00,0.08,255,255,1.00,0.00,0.00,0.00,0.00,
'0,tcp,http,SF,147,3253,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,3,3,0.00,0.00,0.00,0.00,1.00,0.00,0.00,255,255,1.00,0.00,0.00,0.00,0.00,0
'0,tcp,http,SF,182,4023,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,9,9,0.00,0.00,0.00,0.00,1.00,0.00,0.00,255,255,1.00,0.00,0.00,0.00,0.00,0
'0,tcp,http,SF,195,24572,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,2,2,0.00,0.00,0.00,0.00,1.00,0.00,0.00,255,255,1.00,0.00,0.00,0.00,0.00,
'0,tcp,smtp,SF,3366,329,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,1,1,0.00,0.00,0.00,0.00,1.00,0.00,0.00,8,7,0.88,0.25,0.12,0.00,0.00,0.00,
'0,tcp,smtp,SF,1666,328,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,1,1,0.00,0.00,0.00,0.00,1.00,0.00,0.00,18,12,0.67,0.17,0.06,0.00,0.00,0.0
'0,tcp,smtp,SF,751,279,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,1,1,0.00,0.00,0.00,0.00,1.00,0.00,0.00,28,19,0.68,0.14,0.04,0.00,0.00,0.00
'0,tcp,finger,SF,9,140,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0.00,0.00,0.00,0.00,1.00,0.00,0.00,38,5,0.13,0.11,0.03,0.00,0.00,0.00,
'0,tcp,smtp,SF,620,329,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,1,2,0.00,0.00,0.00,0.00,1.00,0.00,1.00,48,26,0.54,0.10,0.02,0.00,0.00,0.00
'0,tcp,smtp,SF,805,327,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,2,2,0.00,0.00,0.00,0.00,1.00,0.00,0.00,58,35,0.60,0.09,0.02,0.00,0.00,0.00
'0,tcp,smtp,SF,842,279,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,2,1,0.00,0.00,0.00,0.00,0.50,1.00,0.00,68,41,0.60,0.07,0.01,0.00,0.00,0.00
'0,udp,domain_u,SF,33,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,2,0.00,0.00,0.00,0.00,1.00,0.00,1.00,78,14,0.18,0.06,0.18,0.00,0.00,0.0
'0,udp,domain_u,SF,30,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0.00,0.00,0.00,0.00,1.00,0.00,0.00,88,15,0.17,0.06,0.17,0.00,0.00,0.0
'0,tcp,smtp,SF,1375,328,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,1,1,0.00,0.00,0.00,0.00,1.00,0.00,0.00,98,64,0.65,0.05,0.01,0.00,0.00,0.0
'1,tcp,smtp,SF,1550,326,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,1,3,0.00,0.00,0.00,0.00,1.00,0.00,1.00,108,72,0.67,0.05,0.01,0.00,0.00,0.
'0,udp,domain_u,SF,30,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,4,0.00,0.00,0.00,0.00,1.00,0.00,0.75,118,21,0.18,0.04,0.18,0.00,0.00,0.
'0,tcp,smtp,SF,1254,323,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,1,1,0.00,0.00,0.00,0.00,1.00,0.00,0.00,5,82,0.80,0.40,0.20,0.02,0.00,0.00
'0,udp,domain_u,SF,31,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,2,0.00,0.00,0.00,0.00,1.00,0.00,1.00,15,26,0.27,0.20,0.27,0.08,0.00,0.0
'0,tcp,smtp,SF,2753,280,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,1,1,0.00,0.00,0.00,0.00,1.00,0.00,0.00,25,95,0.68,0.16,0.04,0.02,0.00,0.0
'0,tcp,auth,SF,9,35,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,3,1,0.00,0.00,0.00,0.00,0.33,0.67,0.00,35,12,0.11,0.11,0.03,0.17,0.00,0.00,0.
```

```
raw_rdd.count()
```

```
494021
```

```
type(raw_rdd)
```

```
pyspark.rdd.RDD
```

Building a Spark DataFrame on our Data

A Spark DataFrame is an interesting data structure representing a distributed collecion of data. A DataFrame is a Dataset organized into named columns. It is conceptually equivalent to a table in a relational database or a dataframe 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 in our case.

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.

```
from pyspark.sql import SQLContext
sqlContext = SQLContext(sc)
sqlContext
```

```
C:\spark\python\pyspark\sql\context.py:113: FutureWarning: Deprecated in 3.0.0. Use SparkSession.builder.getOrCreate() instead.
  warnings.warn(
<pyspark.sql.context.SQLContext at 0x1c50a45a810>
```

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

```
csv_rdd = raw_rdd.map(lambda row: row.split(","))
```

```
csv_rdd.collect()
```

```
  '0',
  '0',
  '0',
  '0',
  '8',
  '8',
  '0.00',
  '0.00',
  '0.00',
  '0.00',
  '1.00',
  '0.00',
  '0.00',
  '29',
  '29',
  '1.00',
  '0.00',
  '0.03',
  '0.00',
  '0.00',
  '0.00',
  '0.00',
  '0.00',
  'normal.'],
 ['0',
  'tcp',
  'http',
  'SF',
  '219',
  '1337',
  '0',
  '0',
  '0',
  '0',
  '0',
  '1',
  '0',
  '0',
  '0',
  '0',
  '0',
  '0',
  '0',
  '0',
  '0',
  '0',
  '6',
  '6',
  '0.00',
  '0.00',
  '0.00',
  '0.00',
  '1.00',
  '0.00',
  '0.00',
  '39',
  '39',
  '1.00',
  '0.00',
```

```
print(csv_rdd.take(2))
```

```
[['0', 'tcp', 'http', 'SF', '181', '5450', '0', '0', '0', '0', '0', '1', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '8', '8',
```

```
print(type(csv_rdd))
```

```
<class 'pyspark.rdd.PipelinedRDD'>
```

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])
```

```
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** and further details pertaining to the description for each attribute\column can be found **here**. We will just be using some specific columns from the dataset, the details of which are specified below.

feature num	feature name	description	type
1	duration	length (number of seconds) of the connection	continuous
2	protocol_type	type of the protocol, e.g. tcp, udp, etc.	discrete
3	service	network service on the destination, e.g., http, telnet, etc.	discrete
4	src_bytes	number of data bytes from source to destination	continuous
5	dst_bytes	number of data bytes from destination to source	continuous
6	flag	normal or error status of the connection	discrete
7	wrong_fragment	number of ``wrong'' fragments	continuous
8	urgent	number of urgent packets	continuous
9	hot	number of ``hot'' indicators	continuous
10	num_failed_logins	number of failed login attempts	continuous
11	num_compromised	number of ``compromised'' conditions	continuous
12	su_attempted	1 if ``su root'' command attempted; 0 otherwise	discrete
13	num_root	number of ``root'' accesses	continuous
14	num_file_creations	number of file creation operations	continuous

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

```
from pyspark.sql import Row

parsed_rdd = csv_rdd.map(lambda r: Row(
    duration=int(r[0]),
    protocol_type=r[1],
    service=r[2],
    flag=r[3],
    src_bytes=int(r[4]),
    dst_bytes=int(r[5]),
    wrong_fragment=int(r[7]),
    urgent=int(r[8]),
    hot=int(r[9]),
    num_failed_logins=int(r[10]),
    num_compromised=int(r[12]),
    su_attempted=r[14],
    num_root=int(r[15]),
    num_file_creations=int(r[16]),
    label=r[-1]
    )
)

```

```
parsed_rdd.collect()
```

```
 Row(duration=0, protocol_type='tcp', service='http', flag='SF', src_bytes=305, dst_bytes=19509, wrong_fragment=0, urgent=0,
hot=0, num_failed_logins=0, num_compromised=0, su_attempted='0', num_root=0, num_file_creations=0, label='normal.'),
 Row(duration=0, protocol_type='tcp', service='http', flag='SF', src_bytes=231, dst_bytes=1869, wrong_fragment=0, urgent=0,
hot=0, num_failed_logins=0, num_compromised=0, su_attempted='0', num_root=0, num_file_creations=0, label='normal.'),
 Row(duration=0, protocol_type='tcp', service='http', flag='SF', src_bytes=217, dst_bytes=363, wrong_fragment=0, urgent=0,
hot=0, num_failed_logins=0, num_compromised=0, su_attempted='0', num_root=0, num_file_creations=0, label='normal.'),
 Row(duration=0, protocol_type='tcp', service='http', flag='SF', src_bytes=297, dst_bytes=2769, wrong_fragment=0, urgent=0,
hot=0, num_failed_logins=0, num_compromised=0, su_attempted='0', num_root=0, num_file_creations=0, label='normal.'),
 Row(duration=0, protocol_type='tcp', service='http', flag='SF', src_bytes=308, dst_bytes=1278, wrong_fragment=0, urgent=0,
hot=0, num_failed_logins=0, num_compromised=0, su_attempted='0', num_root=0, num_file_creations=0, label='normal.'),
 Row(duration=0, protocol_type='tcp', service='http', flag='SF', src_bytes=274, dst_bytes=196, wrong_fragment=0, urgent=0,
hot=0, num_failed_logins=0, num_compromised=0, su_attempted='0', num_root=0, num_file_creations=0, label='normal.'),
 Row(duration=0, protocol_type='tcp', service='http', flag='SF', src_bytes=291, dst_bytes=5657, wrong_fragment=0, urgent=0,
hot=0, num_failed_logins=0, num_compromised=0, su_attempted='0', num_root=0, num_file_creations=0, label='normal.'),
 Row(duration=0, protocol_type='tcp', service='http', flag='SF', src_bytes=328, dst_bytes=1433, wrong_fragment=0, urgent=0,
hot=0, num_failed_logins=0, num_compromised=0, su_attempted='0', num_root=0, num_file_creations=0, label='normal.'),
 Row(duration=0, protocol_type='tcp', service='http', flag='SF', src_bytes=211, dst_bytes=2101, wrong_fragment=0, urgent=0,
hot=0, num_failed_logins=0, num_compromised=0, su_attempted='0', num_root=0, num_file_creations=0, label='normal.'),
 Row(duration=0, protocol_type='tcp', service='http', flag='SF', src_bytes=325, dst_bytes=2452, wrong_fragment=0, urgent=0,
hot=0, num_failed_logins=0, num_compromised=0, su_attempted='0', num_root=0, num_file_creations=0, label='normal.'),
 Row(duration=0, protocol_type='tcp', service='http', flag='SF', src_bytes=340, dst_bytes=22965, wrong_fragment=0, urgent=0,
hot=0, num_failed_logins=0, num_compromised=0, su_attempted='0', num_root=0, num_file_creations=0, label='normal.'),
 Row(duration=0, protocol_type='tcp', service='http', flag='SF', src_bytes=235, dst_bytes=1028, wrong_fragment=0, urgent=0,
hot=0, num_failed_logins=0, num_compromised=0, su_attempted='0', num_root=0, num_file_creations=0, label='normal.'),
 Row(duration=0, protocol_type='tcp', service='http', flag='SF', src_bytes=223, dst_bytes=1275, wrong_fragment=0, urgent=0,
hot=0, num_failed_logins=0, num_compromised=0, su_attempted='0', num_root=0, num_file_creations=0, label='normal.'),
 Row(duration=0, protocol_type='tcp', service='http', flag='SF', src_bytes=226, dst_bytes=4006, wrong_fragment=0, urgent=0,
hot=0, num_failed_logins=0, num_compromised=0, su_attempted='0', num_root=0, num_file_creations=0, label='normal.'),
 Row(duration=0, protocol_type='tcp', service='http', flag='SF', src_bytes=231, dst_bytes=5283, wrong_fragment=0, urgent=0,
hot=1, num_failed_logins=0, num_compromised=0, su_attempted='0', num_root=0, num_file_creations=0, label='normal.'),
 Row(duration=0, protocol_type='tcp', service='http', flag='SF', src_bytes=230, dst_bytes=5558, wrong_fragment=0, urgent=0,
hot=0, num_failed_logins=0, num_compromised=0, su_attempted='0', num_root=0, num_file_creations=0, label='normal.'),
 ...]
```

```
parsed_rdd.take(5)
```

```
---------------------------------------------------------------------------
Py4JJavaError                             Traceback (most recent call last)
Cell In[20], line 1
----> 1 parsed_rdd.take(5)

File C:\spark\python\pyspark\rdd.py:2855, in RDD.take(self, num)
   2852         taken += 1
   2854 p = range(partsScanned, min(partsScanned + numPartsToTry, totalParts))
-> 2855 res = self.context.runJob(self, takeUpToNumLeft, p)
   2857 items += res
   2858 partsScanned += numPartsToTry

File C:\spark\python\pyspark\context.py:2510, in SparkContext.runJob(self, rdd, partitionFunc, partitions, allowLocal)
   2508 mappedRDD = rdd.mapPartitions(partitionFunc)
   2509 assert self._jvm is not None
-> 2510 sock_info = self._jvm.PythonRDD.runJob(self._jsc.sc(), mappedRDD._jrdd, partitions)
   2511 return list(_load_from_socket(sock_info, mappedRDD._jrdd_deserializer))

File C:\python\Lib\site-packages\py4j\java_gateway.py:1322, in JavaMember.__call__(self, *args)
   1316 command = proto.CALL_COMMAND_NAME +\
   1317     self.command_header +\
   1318     args_command +\
   1319     proto.END_COMMAND_PART
   1321 answer = self.gateway_client.send_command(command)
-> 1322 return_value = get_return_value(
   1323     answer, self.gateway_client, self.target_id, self.name)
   1325 for temp_arg in temp_args:
   1326     if hasattr(temp_arg, "_detach"):

File C:\spark\python\pyspark\errors\exceptions\captured.py:179, in capture_sql_exception.<locals>.deco(*a, **kw)
    177 def deco(*a: Any, **kw: Any) -> Any:
    178     try:
--> 179         return f(*a, **kw)
    180     except Py4JJavaError as e:
    181         converted = convert_exception(e.java_exception)

File C:\python\Lib\site-packages\py4j\protocol.py:326, in get_return_value(answer, gateway_client, target_id, name)
    324 value = OUTPUT_CONVERTER[type](answer[2:], gateway_client)
    325 if answer[1] == REFERENCE_TYPE:
--> 326     raise Py4JJavaError(
    327         "An error occurred while calling {0}{1}{2}.\n".
    328         format(target_id, ".", name), value)
    329 else:
    330     raise Py4JError(
    331         "An error occurred while calling {0}{1}{2}. Trace:\n{3}\n".
    332         format(target_id, ".", name, value))

Py4JJavaError: An error occurred while calling z:org.apache.spark.api.python.PythonRDD.runJob.
: org.apache.spark.SparkException: Job aborted due to stage failure: Task 0 in stage 10.0 failed 1 times, most recent failure: 
Lost task 0.0 in stage 10.0 (TID 10) (konoha executor driver): java.net.SocketException: Connection reset by peer: socket write 
error
	at java.net.SocketOutputStream.socketWrite0(Native Method)
	at java.net.SocketOutputStream.socketWrite(SocketOutputStream.java:111)
	at java.net.SocketOutputStream.write(SocketOutputStream.java:155)
	at java.io.BufferedOutputStream.flushBuffer(BufferedOutputStream.java:82)
	at java.io.BufferedOutputStream.write(BufferedOutputStream.java:126)
	at java.io.DataOutputStream.write(DataOutputStream.java:107)
	at java.io.FilterOutputStream.write(FilterOutputStream.java:97)
	at org.apache.spark.api.python.PythonRDD$.writeUTF(PythonRDD.scala:492)
	at org.apache.spark.api.python.PythonRDD$.write$1(PythonRDD.scala:312)
	at org.apache.spark.api.python.PythonRDD$.$anonfun$writeIteratorToStream$1(PythonRDD.scala:322)
	at org.apache.spark.api.python.PythonRDD$.$anonfun$writeIteratorToStream$1$adapted(PythonRDD.scala:322)
	at scala.collection.IterableOnceOps.foreach(IterableOnce.scala:563)
	at scala.collection.IterableOnceOps.foreach$(IterableOnce.scala:561)
	at scala.collection.AbstractIterator.foreach(Iterator.scala:1293)
	at org.apache.spark.api.python.PythonRDD$.writeIteratorToStream(PythonRDD.scala:322)
	at org.apache.spark.api.python.PythonRunner$$anon$2.writeIteratorToStream(PythonRunner.scala:751)
	at org.apache.spark.api.python.BasePythonRunner$WriterThread.$anonfun$run$1(PythonRunner.scala:451)
	at org.apache.spark.util.Utils$.logUncaughtExceptions(Utils.scala:1928)
	at org.apache.spark.api.python.BasePythonRunner$WriterThread.run(PythonRunner.scala:282)

Driver stacktrace:
	at org.apache.spark.scheduler.DAGScheduler.failJobAndIndependentStages(DAGScheduler.scala:2844)
	at org.apache.spark.scheduler.DAGScheduler.$anonfun$abortStage$2(DAGScheduler.scala:2780)
	at org.apache.spark.scheduler.DAGScheduler.$anonfun$abortStage$2$adapted(DAGScheduler.scala:2779)
	at scala.collection.immutable.List.foreach(List.scala:333)
	at org.apache.spark.scheduler.DAGScheduler.abortStage(DAGScheduler.scala:2779)
	at org.apache.spark.scheduler.DAGScheduler.$anonfun$handleTaskSetFailed$1(DAGScheduler.scala:1242)
	at org.apache.spark.scheduler.DAGScheduler.$anonfun$handleTaskSetFailed$1$adapted(DAGScheduler.scala:1242)
	at scala.Option.foreach(Option.scala:437)
	at org.apache.spark.scheduler.DAGScheduler.handleTaskSetFailed(DAGScheduler.scala:1242)
	at org.apache.spark.scheduler.DAGSchedulerEventProcessLoop.doOnReceive(DAGScheduler.scala:3048)
	at org.apache.spark.scheduler.DAGSchedulerEventProcessLoop.onReceive(DAGScheduler.scala:2982)
	at org.apache.spark.scheduler.DAGSchedulerEventProcessLoop.onReceive(DAGScheduler.scala:2971)
	at org.apache.spark.util.EventLoop$$anon$1.run(EventLoop.scala:49)
	at org.apache.spark.scheduler.DAGScheduler.runJob(DAGScheduler.scala:984)
	at org.apache.spark.SparkContext.runJob(SparkContext.scala:2398)
	at org.apache.spark.SparkContext.runJob(SparkContext.scala:2419)
	at org.apache.spark.SparkContext.runJob(SparkContext.scala:2438)
	at org.apache.spark.api.python.PythonRDD$.runJob(PythonRDD.scala:181)
```

```
        at org.apache.spark.api.python.PythonRDD.runJob(PythonRDD.scala)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
        at java.lang.reflect.Method.invoke(Method.java:498)
        at py4j.reflection.MethodInvoker.invoke(MethodInvoker.java:244)
        at py4j.reflection.ReflectionEngine.invoke(ReflectionEngine.java:374)
        at py4j.Gateway.invoke(Gateway.java:282)
        at py4j.commands.AbstractCommand.invokeMethod(AbstractCommand.java:132)
        at py4j.commands.CallCommand.execute(CallCommand.java:79)
        at py4j.ClientServerConnection.waitForCommands(ClientServerConnection.java:182)
        at py4j.ClientServerConnection.run(ClientServerConnection.java:106)
        at java.lang.Thread.run(Thread.java:748)
Caused by: java.net.SocketException: Connection reset by peer: socket write error
        at java.net.SocketOutputStream.socketWrite0(Native Method)
        at java.net.SocketOutputStream.socketWrite(SocketOutputStream.java:111)
        at java.net.SocketOutputStream.write(SocketOutputStream.java:155)
        at java.io.BufferedOutputStream.flushBuffer(BufferedOutputStream.java:82)
        at java.io.BufferedOutputStream.write(BufferedOutputStream.java:126)
        at java.io.DataOutputStream.write(DataOutputStream.java:107)
        at java.io.FilterOutputStream.write(FilterOutputStream.java:97)
        at org.apache.spark.api.python.PythonRDD$.writeUTF(PythonRDD.scala:492)
        at org.apache.spark.api.python.PythonRDD$.write$1(PythonRDD.scala:312)
        at org.apache.spark.api.python.PythonRDD$.$anonfun$writeIteratorToStream$1(PythonRDD.scala:322)
        at org.apache.spark.api.python.PythonRDD$.$anonfun$writeIteratorToStream$1$adapted(PythonRDD.scala:322)
        at scala.collection.IterableOnceOps.foreach(IterableOnce.scala:563)
        at scala.collection.IterableOnceOps.foreach$(IterableOnce.scala:561)
        at scala.collection.AbstractIterator.foreach(Iterator.scala:1293)
        at org.apache.spark.api.python.PythonRDD$.writeIteratorToStream(PythonRDD.scala:322)
        at org.apache.spark.api.python.PythonRunner$$anon$2.writeIteratorToStream(PythonRunner.scala:751)
        at org.apache.spark.api.python.BasePythonRunner$WriterThread.$anonfun$run$1(PythonRunner.scala:451)
        at org.apache.spark.util.Utils$.logUncaughtExceptions(Utils.scala:1928)
        at org.apache.spark.api.python.BasePythonRunner$WriterThread.run(PythonRunner.scala:282)
```

Constructing the DataFrame

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

```
df = sqlContext.createDataFrame(parsed_rdd)
df.show(10)
```

```
+--------+-------------+-------+----+---------+---------+--------------+------+---+-----------------+--------------+------------+-
|duration|protocol_type|service|flag|src_bytes|dst_bytes|wrong_fragment|urgent|hot|num_failed_logins|num_compromised|su_attempted|nu
+--------+-------------+-------+----+---------+---------+--------------+------+---+-----------------+--------------+------------+-
|       0|          tcp|   http|  SF|      181|     5450|             0|     0|  0|                0|             0|           0|
|       0|          tcp|   http|  SF|      239|      486|             0|     0|  0|                0|             0|           0|
|       0|          tcp|   http|  SF|      235|     1337|             0|     0|  0|                0|             0|           0|
|       0|          tcp|   http|  SF|      219|     1337|             0|     0|  0|                0|             0|           0|
|       0|          tcp|   http|  SF|      217|     2032|             0|     0|  0|                0|             0|           0|
|       0|          tcp|   http|  SF|      217|     2032|             0|     0|  0|                0|             0|           0|
|       0|          tcp|   http|  SF|      212|     1940|             0|     0|  0|                0|             0|           0|
|       0|          tcp|   http|  SF|      159|     4087|             0|     0|  0|                0|             0|           0|
|       0|          tcp|   http|  SF|      210|      151|             0|     0|  0|                0|             0|           0|
|       0|          tcp|   http|  SF|      212|      786|             0|     0|  1|                0|             0|           0|
+--------+-------------+-------+----+---------+---------+--------------+------+---+-----------------+--------------+------------+-
only showing top 10 rows
```

Now, we can easily have a look at our dataframe's schema using tne `printSchema(...)` function.

```
df.printSchema()
```

```
root
 |-- duration: long (nullable = true)
 |-- protocol_type: string (nullable = true)
 |-- service: string (nullable = true)
 |-- flag: string (nullable = true)
 |-- src_bytes: long (nullable = true)
 |-- dst_bytes: long (nullable = true)
 |-- wrong_fragment: long (nullable = true)
 |-- urgent: long (nullable = true)
 |-- hot: long (nullable = true)
 |-- num_failed_logins: long (nullable = true)
 |-- num_compromised: long (nullable = true)
 |-- su_attempted: string (nullable = true)
 |-- num_root: long (nullable = true)
 |-- num_file_creations: long (nullable = true)
 |-- label: string (nullable = true)
```

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)
```

```
Help on method registerTempTable in module pyspark.sql.dataframe:

registerTempTable(name: str) -> None method of pyspark.sql.dataframe.DataFrame instance
    Registers this :class:`DataFrame` as a temporary table using the given name.

    The lifetime of this temporary table is tied to the :class:`SparkSession`
    that was used to create this :class:`DataFrame`.

    .. versionadded:: 1.3.0

    .. versionchanged:: 3.4.0
        Supports Spark Connect.

    .. deprecated:: 2.0.0
        Use :meth:`DataFrame.createOrReplaceTempView` instead.

    Parameters
    ----------
    name : str
        Name of the temporary table to register.

    Examples
    --------
    >>> df = spark.createDataFrame([(2, "Alice"), (5, "Bob")], schema=["age", "name"])
    >>> df.registerTempTable("people")
    >>> df2 = spark.sql("SELECT * FROM people")
    >>> sorted(df.collect()) == sorted(df2.collect())
    True
    >>> spark.catalog.dropTempView("people")
    True
```

```
df.registerTempTable("connections")
```

```
C:\spark\python\pyspark\sql\dataframe.py:329: FutureWarning: Deprecated in 2.0, use createOrReplaceTempView instead.
  warnings.warn("Deprecated in 2.0, use createOrReplaceTempView instead.", FutureWarning)
```

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, subqueries and pivots

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.

```
df.groupBy('protocol_type').count().orderBy('count', ascending=False).show()
```

```
+-------------+------+
|protocol_type| count|
+-------------+------+
|         icmp|283602|
|          tcp|190065|
|          udp| 20354|
+-------------+------+
```

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

```
protocols = sqlContext.sql("""
                           SELECT protocol_type, count(*) as freq
                           FROM connections
                           GROUP BY protocol_type
                           ORDER BY 2 DESC
                           """)
protocols.show()
```

```
+-------------+------+
|protocol_type|  freq|
+-------------+------+
|         icmp|283602|
|          tcp|190065|
|          udp| 20354|
+-------------+------+
```

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.

```
labels = sqlContext.sql("""
                           SELECT label, count(*) as freq
                           FROM connections
                           GROUP BY label
                           ORDER BY 2 DESC
                           """)
labels.show()
```

```
+----------------+------+
|           label|  freq|
+----------------+------+
|          smurf.|280790|
|        neptune.|107201|
|         normal.| 97278|
|           back.|  2203|
|          satan.|  1589|
|        ipsweep.|  1247|
|      portsweep.|  1040|
|     warezclient.|  1020|
|       teardrop.|   979|
|            pod.|   264|
|           nmap.|   231|
|    guess_passwd.|    53|
|buffer_overflow.|    30|
|           land.|    21|
|     warezmaster.|    20|
|           imap.|    12|
|        rootkit.|    10|
|      loadmodule.|     9|
|       ftp_write.|     8|
|        multihop.|     7|
+----------------+------+
only showing top 20 rows
```


```
labels = sqlContext.sql("""
                           SELECT label, count(*) as freq
                           FROM connections
                           GROUP BY label
                           ORDER BY 2 DESC
                           """)
labels.show()
```

```
+----------------+------+
|           label|  freq|
+----------------+------+
|          smurf.|280790|
|        neptune.|107201|
|         normal.| 97278|
|           back.|  2203|
|          satan.|  1589|
|        ipsweep.|  1247|
|      portsweep.|  1040|
|     warezclient.|  1020|
|       teardrop.|   979|
|            pod.|   264|
|           nmap.|   231|
|    guess_passwd.|    53|
|buffer_overflow.|    30|
|           land.|    21|
|     warezmaster.|    20|
|           imap.|    12|
|        rootkit.|    10|
|      loadmodule.|     9|
|       ftp_write.|     8|
|        multihop.|     7|
+----------------+------+
```

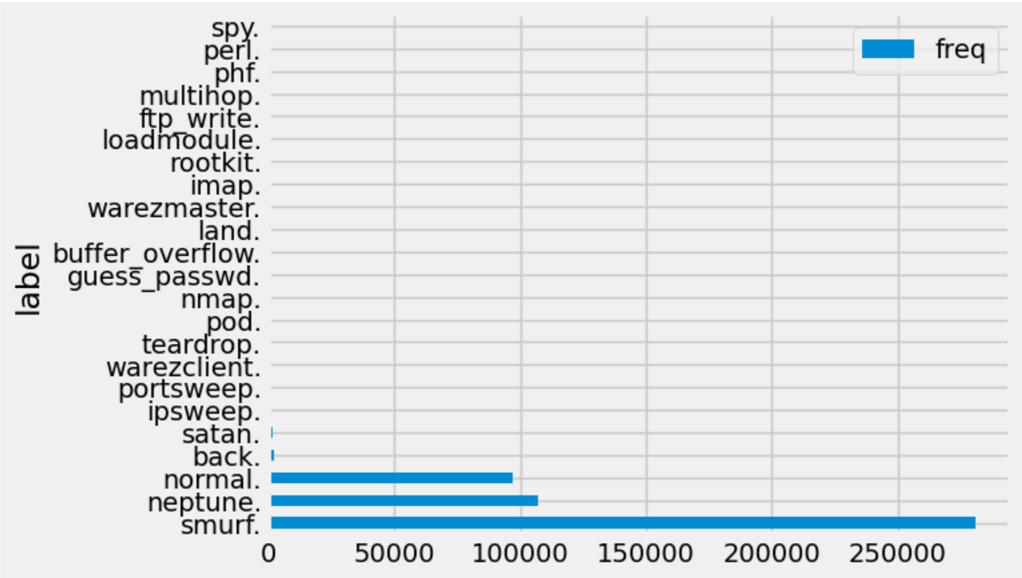
```
only showing top 20 rows
```

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.

```
labels_df = pd.DataFrame(labels.toPandas())
labels_df.set_index("label", drop=True,inplace=True)
labels_fig = labels_df.plot(kind='barh')

plt.rcParams["figure.figsize"] = (7, 5)
plt.rcParams.update({'font.size': 10})


```



Connections based on protocols and attacks

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

```
attack_protocol = sqlContext.sql("""
                           SELECT
                             protocol_type,
                             CASE label
                               WHEN 'normal.' THEN 'no attack'
                               ELSE 'attack'
                             END AS state,
                             COUNT(*) as freq
                           FROM connections
                           GROUP BY protocol_type, state
                           ORDER BY 3 DESC
                           """)
attack_protocol.show()
```

```
+-------------+---------+------+
|protocol_type|    state|  freq|
+-------------+---------+------+
|         icmp|   attack|282314|
|          tcp|   attack|113252|
|          tcp|no attack| 76813|
|          udp|no attack| 19177|
|         icmp|no attack|  1288|
|          udp|   attack|  1177|
+-------------+---------+------+
```

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.

```
attack_stats = sqlContext.sql("""
                           SELECT
```

```
                              protocol_type,
                              CASE label
                                WHEN 'normal.' THEN 'no attack'
                                ELSE 'attack'
                              END AS state,
                              COUNT(*) as total_freq,
                              ROUND(AVG(src_bytes), 2) as mean_src_bytes,
                              ROUND(AVG(dst_bytes), 2) as mean_dst_bytes,
                              ROUND(AVG(duration), 2) as mean_duration,
                              SUM(num_failed_logins) as total_failed_logins,
                              SUM(num_compromised) as total_compromised,
                              SUM(num_file_creations) as total_file_creations,
                              SUM(su_attempted) as total_root_attempts,
                              SUM(num_root) as total_root_acceses
                            FROM connections
                            GROUP BY protocol_type, state
                            ORDER BY 3 DESC
                            """)
attack_stats.show()
```

```
+------------+---------+----------+--------------+--------------+-------------+-------------------+-----------------+------------
|protocol_type|    state|total_freq|mean_src_bytes|mean_dst_bytes|mean_duration|total_failed_logins|total_compromised|total_file_cre
+------------+---------+----------+--------------+--------------+-------------+-------------------+-----------------+------------
|        icmp|   attack|    282314|        932.14|           0.0|          0.0|                  0|                0|
|         tcp|   attack|    113252|       9880.38|        881.41|        23.19|                 57|             2269|
|         tcp|no attack|     76813|       1439.31|       4263.97|        11.08|                 18|             2776|
|         udp|no attack|     19177|         98.01|         89.89|      1054.63|                  0|                0|
|        icmp|no attack|      1288|         91.47|           0.0|          0.0|                  0|                0|
|         udp|   attack|      1177|          27.5|          0.23|          0.0|                  0|                0|
+------------+---------+----------+--------------+--------------+-------------+-------------------+-----------------+------------
```

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.

```
tcp_attack_stats = sqlContext.sql("""
                                   SELECT
                                     service,
                                     label as attack_type,
                                     COUNT(*) as total_freq,
                                     ROUND(AVG(duration), 2) as mean_duration,
                                     SUM(num_failed_logins) as total_failed_logins,
                                     SUM(num_file_creations) as total_file_creations,
                                     SUM(su_attempted) as total_root_attempts,
                                     SUM(num_root) as total_root_acceses
                                   FROM connections
                                   WHERE protocol_type = 'tcp'
                                   AND label != 'normal.'
                                   GROUP BY service, attack_type
                                   ORDER BY total_freq DESC
                                   """)
tcp_attack_stats.show()
```

```
+----------+-----------+----------+-------------+-------------------+--------------------+-------------------+------------------+
|   service|attack_type|total_freq|mean_duration|total_failed_logins|total_file_creations|total_root_attempts|total_root_acceses|
+----------+-----------+----------+-------------+-------------------+--------------------+-------------------+------------------+
|   private|   neptune.|    101317|          0.0|                  0|                   0|                0.0|                 0|
|      http|      back.|      2203|         0.13|                  0|                   0|                0.0|                 0|
|     other|     satan.|      1221|          0.0|                  0|                   0|                0.0|                 0|
|   private|  portsweep.|       725|      1915.81|                  0|                   0|                0.0|                 0|
|  ftp_data|warezclient.|       708|       403.71|                  0|                   0|                0.0|                 0|
|       ftp|warezclient.|       307|      1063.79|                  0|                   0|                0.0|                 0|
|     other|  portsweep.|       260|      1058.22|                  0|                   0|                0.0|                 0|
|    telnet|   neptune.|       197|          0.0|                  0|                   0|                0.0|                 0|
|      http|   neptune.|       192|          0.0|                  0|                   0|                0.0|                 0|
|    finger|   neptune.|       177|          0.0|                  0|                   0|                0.0|                 0|
|  ftp_data|   neptune.|       170|          0.0|                  0|                   0|                0.0|                 0|
|   private|     satan.|       170|         0.05|                  0|                   0|                0.0|                 0|
|   csnet_ns|   neptune.|       123|          0.0|                  0|                   0|                0.0|                 0|
|      smtp|   neptune.|       120|          0.0|                  0|                   0|                0.0|                 0|
|remote_job|   neptune.|       118|          0.0|                  0|                   0|                0.0|                 0|
|     pop_3|   neptune.|       118|          0.0|                  0|                   0|                0.0|                 0|
|   discard|   neptune.|       115|          0.0|                  0|                   0|                0.0|                 0|
|   iso_tsap|   neptune.|       115|          0.0|                  0|                   0|                0.0|                 0|
```

```
|    systat|    neptune.|       113|         0.0|                 0|                  0|               0.0|                 0|
|    domain|    neptune.|       112|         0.0|                 0|                  0|               0.0|                 0|
+----------+------------+----------+------------+------------------+-------------------+------------------+------------------+
only showing top 20 rows
```

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.

```
tcp_attack_stats = sqlContext.sql("""
                                   SELECT
                                     service,
                                     label as attack_type,
                                     COUNT(*) as total_freq,
                                     ROUND(AVG(duration), 2) as mean_duration,
                                     SUM(num_failed_logins) as total_failed_logins,
                                     SUM(num_file_creations) as total_file_creations,
                                     SUM(su_attempted) as total_root_attempts,
                                     SUM(num_root) as total_root_acceses
                                   FROM connections
                                   WHERE (protocol_type = 'tcp'
                                          AND label != 'normal.')
                                   GROUP BY service, attack_type
                                   HAVING (mean_duration >= 50
                                           AND total_file_creations >= 5
                                           AND total_root_acceses >= 1)
                                   ORDER BY total_freq DESC
                                   """)
tcp_attack_stats.show()
```

```
+-------+---------------+----------+-------------+-------------------+--------------------+-------------------+------------------+
|service|    attack_type|total_freq|mean_duration|total_failed_logins|total_file_creations|total_root_attempts|total_root_acceses|
+-------+---------------+----------+-------------+-------------------+--------------------+-------------------+------------------+
| telnet|buffer_overflow.|        21|       130.67|                  0|                  15|                0.0|                 5|
| telnet|    loadmodule.|         5|         63.8|                  0|                   9|                0.0|                 3|
| telnet|      multihop.|         2|        458.0|                  0|                   8|                0.0|                93|
+-------+---------------+----------+-------------+-------------------+--------------------+-------------------+------------------+
```

Interesting to see multihop attacks 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.