Spark DataFrame processing on EC2 using pyspark

Spark is a technology for fast processing of big data. This page documents a use case of Spark via the pyspark library and Parquet data format on a single node EC2 instance. This use case accomodates big data processing as a single node EC2 instance can scale up to 96 CPUs and 64 TB of Elastic Block Storage (EBS). See this page regarding the use case of Spark on a multi-node cluster.

The working example will be to count events by vehicle-id for a set of vehicles. The example will demonstrate the following generic concepts in Spark data processing:

- setup Spark and pyspark on an EC2 instance
- create a Parquet dataset on EBS from individual pandas DataFrames
- create a SparkSession object and read a Parquet dataset into a Spark DataFrame object
- create a Spark DataFrame view and run queries via Spark SQL interface
- aggregate and enrich a Spark DataFrame

Setup Spark and pyspark on an EC2 instance

The following is based on a Linux Ubuntu version 20.04 EC2 instance. Spark requires Java and hadoop. Java may be installed via `sudo apt install openjdk-8-jdk` from the Linux console. hadoop may be downloaded via `wget https://dlcdn.apache.org/hadoop/common/hadoop-3.2.2/hadoop-3.2.2.tar.gz` (or latest online version) and then `tar xzf hadoop-3.2.2.tar.gz`. Spark needs environment variables set following install and download of Java and hadoop; eg `.bashrc` including (your config will not be exactly the same):

```
export JAVA_HOME="/usr/lib/jvm/java-1.8.0-openjdk-amd64"
export HADOOP_HOME="/mnt/home/russell.burdt/hadoop-3.2.2"
export LD_LIBRARY_PATH="/mnt/home/russell.burdt/hadoop-3.2.2/lib/native"
```

Following install/download/configuration of Java and hadoop, `pyspark` may be installed in a conda environment via `conda install -c conda-forge pyspark pyarrow`. At this point a `SparkSession` object may be created in a Python session:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()
```

```
In [1]: from pyspark.sql import SparkSession

In [2]: spark = SparkSession.builder.getOrCreate()
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
22/05/11 16:56:16 WARN Utils: Service 'SparkUI' could not bind on port 4040. Attempting port 4041.

In [3]: spark
Out[3]: <pyspark.sql.session.SparkSession at 0x7f802c60dfd0>

In [4]:
```

Note that Spark binds by default to port 4040. At time of writing I had another Spark Session in use so Spark automatically went to port 4041 instead. Note that SparkUI is a built-in service to provide details regarding Spark Session configuration and processing, which is in my case available in a local browser at http://10.144.240.35:4041/ where the first component references the IPv4 address of the EC2 instance.

Create a Parquet dataset on EBS from individual pandas DataFrames

This page assumes the EC2 instance already has EBS mounted at a known location, eg my instance has EBS mounted at `/mnt/home` which may be configured via the AWS console under EC2 and Volumes. Generally, Spark will run faster when working from data on EBS with respect to data on S3, though both are possible.

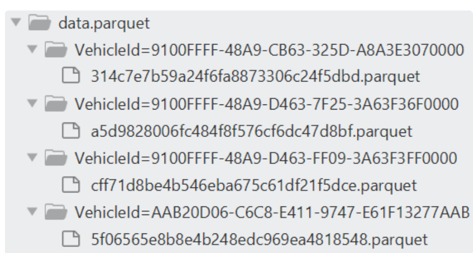
To reiterate, the working example is to count events by vehicle-id for a set of vehicles. So a set of vehicles is first defined as a list of strings each representing a vehicle-id:

```
vids = [
    '9100FFFF-48A9-D463-7F25-3A63F36F0000',
    '9100FFFF-48A9-D463-FF09-3A63F3FF0000',
    '9100FFFF-48A9-CB63-325D-A8A3E3070000',
    'AAB20D06-C6C8-E411-9747-E61F13277AAB']
```

The concept is to iteratively create a Parquet dataset on EBS from individual pandas DataFrames, as in the code below. `path` is the location on EBS of the Parquet dataset. `df` is a pandas DataFrame representing raw events data for each vehicle-id. Arguments in the `to_parquet` method of `df` are recommended, except for `partition_cols` that will be application-specific.

```
path = r'/mnt/home/russell.burdt/data.parquet'
edw = pyodbc.connect('DSN=EDW')
for vid in vids:
    query = f"""
        SELECT VehicleId, RecordDate, Latitude, Longitude, EventTriggerTypeId AS Id
        FROM flat.Events
        WHERE VehicleId = '{vid}'
        AND RecordDate BETWEEN '2021-10-01' AND '2021-12-31'
        """
    df = pd.read_sql_query(query, edw)
    df.to_parquet(
        path=path, engine='pyarrow', compression='snappy', index=False,
        partition_cols=['VehicleId'], flavor='spark')
```

The above code is holding data for one vehicle-id at a time in local memory. The parquet dataset with all data for all vehicles is on EBS which may scale up to 64TB on a single node EC2 instance. In this manner a Parquet dataset of arbitrary size may be created from a loop over queries returning data that fit in local memory. Following above code execution, data at the `path` location will appear as:



Create a SparkSession object and read a Parquet dataset into a Spark DataFrame object

At this point a Parquet dataset persists at `/mnt/home/russell.burdt/data.parquet`, limited in size by EBS space and created from individual pandas DataFrames (in memory).

A `SparkSession` object is created in the code below, where configuration options I have found useful in my own work are included (see descriptions as comments in the code). Regarding the choice for `spark.sql.shuffle.partitions` I have used `2000` when data volume is 10s of GB and `20000` when data volume is 100s of GB. Tuning this parameter can be critical for fast Spark performance.

```
from pyspark import SparkConf
from pyspark.sql import SparkSession

conf = SparkConf()
# memory available for objects returned by Spark
conf.set('spark.driver.memory', '2g')
# enable Apache Arrow
conf.set('spark.sql.execution.arrow.pyspark.enabled', 'true')
# no implicit timezone conversions
```

```
conf.set('spark.sql.session.timeZone', 'UTC')
# directory for temporary files
conf.set('spark.local.dir', r'/mnt/home/russell.burdt/rbin')
# 200 is default, critical to increase proportionally to data volume
conf.set('spark.sql.shuffle.partitions', 200)
spark = SparkSession.builder.config(conf=conf).getOrCreate()
```

A Spark DataFrame object can then be created as `sdf = spark.read.parquet(path)`.

```
In [10]: type(sdf)
Out[10]: pyspark.sql.dataframe.DataFrame
```

Create a Spark DataFrame view and run queries via Spark SQL interface

At this point data can be queried directly from methods of the Spark DataFrame object, or by creating `df` as a view of the Spark DataFrame and running standard SQL directly, as in the code below. Note that `spark.sql(f'SELECT COUNT(*) FROM df')` by itself will return another Spark DataFrame object, and the `toPandas` (also `show` or `collect`) method is needed to return actual data. The key idea is to only return data that can fit in memory, whereas the entire Parquet dataset does not need to fit in memory.

```
In [13]: sdf.createOrReplaceTempView('df')

In [14]: spark.sql(f'SELECT COUNT(*) FROM df').toPandas()
Out[14]:
   count(1)
0       219

In [15]: sdf.count()
Out[15]: 219
```

More complex SQL queries can be executed, such as the query to count events by vehicle-id.

```
spark.sql(f"""
    SELECT VehicleId, COUNT(*) AS records
    FROM df
    GROUP BY VehicleId
    ORDER BY VehicleId""").toPandas()
```

```
Out[37]:
                               VehicleId  records
0  9100FFFF-48A9-CB63-325D-A8A3E3070000       51
1  9100FFFF-48A9-D463-7F25-3A63F36F0000       71
2  9100FFFF-48A9-D463-FF09-3A63F3FF0000       87
3  AAB20D06-C6C8-E411-9747-E61F13277AAB       10
```

Aggregate and enrich a Spark DataFrame

Previous sections demonstrated the full working example - to count events by vehicle-id for a set of vehicles. Another Spark DataFrame object is created here to demonstrate common patterns I have used in my own work to aggregate and enrich a Spark DataFrame; see the code below.

```
import pandas as pd
from pyspark import SparkConf
from pyspark.sql import SparkSession

conf = SparkConf()
conf.set('spark.sql.execution.arrow.pyspark.enabled', 'true')
spark = SparkSession.builder.config(conf=conf).getOrCreate()

pdf = pd.DataFrame({'x': [1, 1, 2, 2], 'y': [3, 4, 5, 6]})
sdf = spark.createDataFrame(pdf)
sdf.createOrReplaceTempView('df')
```

```
In [2]: pdf
Out[2]:
   x  y
0  1  3
1  1  4
2  2  5
3  2  6

In [3]: sdf
Out[3]: DataFrame[x: bigint, y: bigint]

In [4]:
```

The code pattern to implement aggregation or enrichment will depend on the data conversion taking place. More formal references as compared to the following sections may be online, eg here and here.

Series to series

The code below demonstrates enrichment of the Spark DataFrame with an additional column `product`. The column definition is implemented via a `pandas_udf` which is a high-performance user-defined function object enabled by Apache Arrow. Note that type declarations are required when using this feature of Spark, and will specify argument types, return type, and return data type. In this example, the arguments are pandas `Series` objects and the returned object is a pandas `Series` object of integers. There would not be any restrictions to code that can go inside of the user-defined function, other than the use of interactive debuggers (eg `ipdb`) will create issues.

```
from pyspark.sql.functions import pandas_udf
from pyspark.sql.types import IntegerType, DoubleType

@pandas_udf(returnType=IntegerType())
def func(a: pd.Series, b: pd.Series) -> pd.Series:
    return a * b
spark.udf.register('func', func)
dx = spark.sql(f'SELECT x, y, func(x, y) AS product FROM df').toPandas()
```

```
In [4]: dx
Out[4]:
   x  y  product
0  1  3        3
1  1  4        4
2  2  5       10
3  2  6       12
```

Series to float

The series to float data conversion via `pandas_udf` implements an aggregation operation, eg:

```
@pandas_udf(DoubleType())
def func(x: pd.Series) -> float:
    return x.mean()
spark.udf.register('func', func)
dx = spark.sql(f'SELECT x, func(y) AS ymean FROM df GROUP BY x').toPandas()
```

Series to series (window function)

The `applyInPandas` method can be used to implement the equivalent of a SQL window function. For example, the code below implements a cumulative sum for each group of `x` via Spark SQL.

```
In [14]: spark.sql(f'SELECT x, y, SUM(y) OVER (PARTITION BY x ORDER BY y) AS ysum FROM df').toPandas()
Out[14]:
   x  y  ysum
0  1  3     3
1  1  4     7
2  2  5     5
3  2  6    11
```

The equivalent using `applyInPandas` is below. Here the type declarations are implemented as a string argument, and there is no requisite decoration of the distributed function.

```
def func(pdf):
    pdf['ysum'] = pdf['y'].cumsum()
    return pdf
dx = sdf.groupby('x').applyInPandas(func, schema='x long, y long, ysum long').toPandas()
```

```
In [16]: dx
Out[16]:
   x  y  ysum
0  1  3     3
1  1  4     7
2  2  5     5
3  2  6    11
```

DataFrame to DataFrame

The key idea of `mapInPandas` is that an object consistently transforming a DataFrame to another DataFrame is distributed to all Spark executors which process batches of a Spark DataFrame in parallel. In some scenarios it may be necessary to define `func` within a separate module with respect to where `mapInPandas` is executed (eg see here).

```
from typing import Iterator

def func(iterator: Iterator[pd.DataFrame]) -> Iterator[pd.DataFrame]:
    constant = 77.3
    for df in iterator:
        df['operation'] = df['x'] + df['y'] + constant
        yield df
dx = sdf.mapInPandas(func, schema='x long, y long, operation double').toPandas()
```

```
In [23]: dx
Out[23]:
   x  y  operation
0  1  3       81.3
1  1  4       82.3
2  2  5       84.3
3  2  6       85.3
```

Summary

One use case of Spark for distributed data processing via pyspark and Parquet has been described. The same code patterns in this page scale to big data processing. My own work uses Spark and the code patterns in this page to extract consistent metrics for 100s of thousands of Lytx vehicles over variable time windows, which is based on multiple Parquet datasets up to 1TB and has used up to 96 CPUs.

Other useful Spark references are here:

Spark latest official documentation

Spark optimization guidelines

Spark broadcast optimization

Appendix - Spark in Action

The screenshot below is generated by the `htop` Linux utility while Spark consumes appx 100% of 96 CPUs on a single node EC2 instance.

Mem[62.4G/374G]
Swp[0K/0K]
Tasks: 195, 907 thr, 848 kthr; 96 running
Load average: 84.34 84.33 82.14
Uptime: 01:32:15