Introduction

Driving is one of the most common yet dangerous tasks that people perform every day. According to the NHTSA, there is an average of 6 million accidents in the U.S. per year, resulting in over 2.35 million injuries and 37,000 deaths. [1] Additionally, road crashes cost the U.S. $230.6 billion per year. There are various factors that can contribute to accidents such as distracted driving, speeding, poor weather conditions, and alcohol involvement. However, using these factors and additional statistics, we can better predict the cause of accidents and put laws and procedures in place to help minimize the number of accidents. For this assignment we will use Apache Spark to analyze the motor vehicle collisions in New York City (NYC). Our goal is to gain additional insights into the causes of accidents in the Big Apple and how we can help to prevent them.

Datasets

The following datasets were used for our analysis, with the NYC Motor Vehicle Collisions - Crashes being used for a majority of the analysis. All of the datasets were obtained from NYC OpenData in CSV format and contain the most up to date motor vehicle collision information available to the public.

NYC Motor Vehicle Collisions - Crashes: [2]

Contains information about the motor vehicle collisions in NYC from July 2012 to February 2020. The data was extracted from police reports (form MV104-AN) that were filed at the time of the crash. A form MV104-AN is only filed in the case where an individual is injured or fatally injured, or when the damage caused by the accident is $1,000 or greater. This dataset has 29 columns, 1.65 Million rows, and a size of 369 MB. Each row includes details about a specific motor vehicle collision.

NYC Motor Vehicle Crashes - Vehicle: [3]

Contains information about each vehicle that was involved in a crash in NYC from September 2012 to May 2020 and a police report MV104-AN was filed. This dataset has 3.35M rows, 25 columns, and a size of 566.3 MB. Each row represents the vehicle information for a specific crash, which can also be tied back to the NYC Motor Vehicle Collisions - Crashes dataset. Multiple vehicles can be involved in a single crash.

2015 NYC Tree Census: [4]

Contains detailed information on the trees living throughout NYC collected by the NYC parks and recreation board in 2015. This datset has 684K rows, 45 columns, and has a size of 220.4 MB. Each row represents the information for a tree living in NYC.

Chosen Spark API for Answering Analytical Questions

I chose to use the Spark DataFrames API for my analysis. The Spark DataFrames API is similar to relational tables in SQL, however it also provides a programmatic API allowing for more flexibility in query expressiveness, query testing, and is extensible. Additionally, the datasets that we will be using for our analysis are in a format that can be easily worked with using DataFrames. For example our data is in a tabular format with columns and rows, which is how data is represented in DataFrames. Additionally,

DataFrames inherits all of the properties of RDDs such as read only, fault tolerance, caching, and lazy execution but with the additional data storage optimizations, code generation, query planner, and abstraction.

Loading Datasets

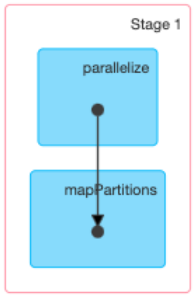
In the beforeAll() function, all three CSV files are read in as as DataFrames, then compressed to Parquet format and persisted to external storage. A check is made to determine if the Parquet files exist on disk before running a test. If the files already exist then they will be reused for all subsequent tests otherwise they will be regenerated. Through the use of Parquet, our data will be stored in a compressed columnar format, which will allow for faster read and write performance as compared to reading from CSV. The file size of the datasets when converted to parquet are significantly smaller as compared to the original CSV files. For example, the CSV file for the NYC Motor Vehicle - Crashes was 369 MB and was reduced to 75.7 MB when converted to Parquet. Additionally, the output Parquet files for the The NYC Motor Vehicle - Vehicles dataset were partitioned by ZIP_CODE. By doing this, the data is physically laid out on the filesystem in an order that will be the most efficient for performing our queries.

Some data preparation was needed before converting to Parquet which include the following.

- For all DataFrames, the whitespace between columns names needed to be replaced with underscores to avoid the invalid character errors when converting to Parquet. E.g. from CRASH TIME to CRASH_TIME.
- The data types of several columns in the NYC Motor Vehicle - Crashes dataset needed to casted from a string to an integer type because they will be used for numerical calculations in our query.

The below Directed acyclic graph (DAG) show the operations performed when reading a parquet file. This operation is performed each time a test is ran. Spark performs a parallelize operation followed by a mapPartions operation.

▾ DAG Visualization

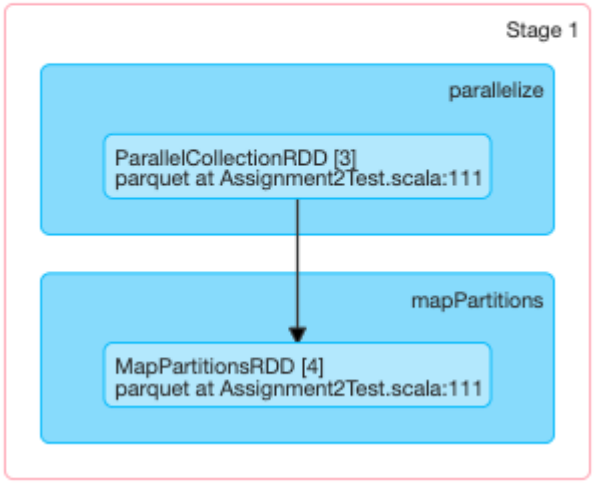


▾ Completed Stages (1)

Stage Id ▾	Description	Submitted	Duration
1	parquet at Assignment2Test.scala:111 +details	2020/05/09 19:12:59	0.6 s

Total Time Across All Tasks: 0.6 s
Locality Level Summary: Process local: 1

▾ DAG Visualization



Analytic Questions

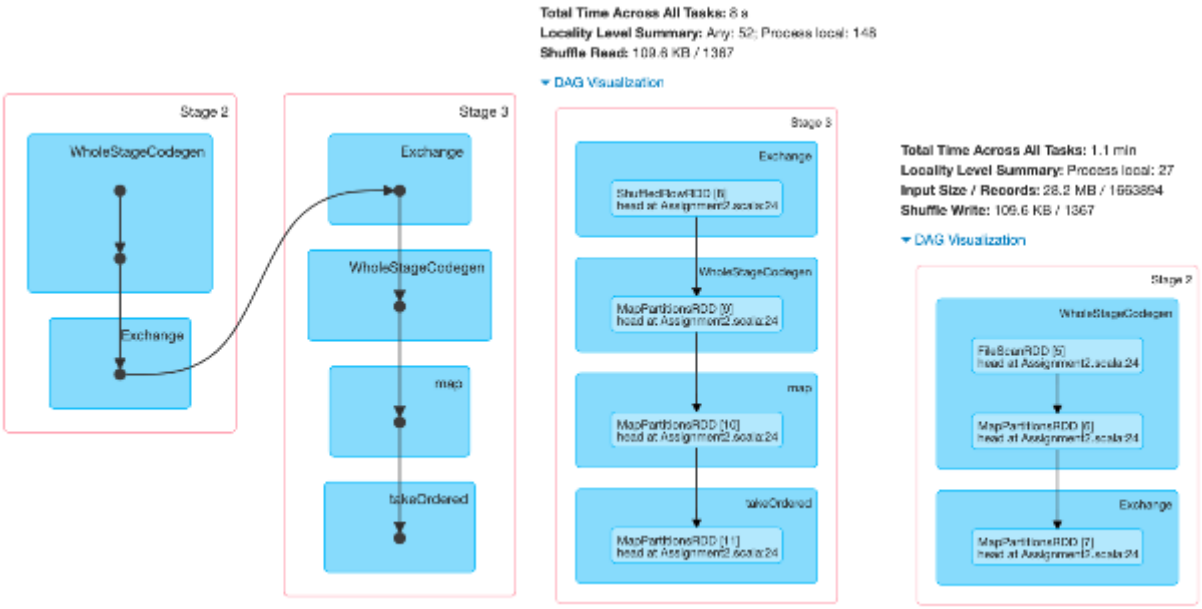
1. What is the top five most frequent contributing factors for accidents in NYC?

For this test, we first filtered out rows with an unspecified contributing factor. Next, we perform a groupBy CONTRIBUTING_FACTOR_VEHICLE_1, then a count the number of occurrence of each contributing factor. Finally, we order by count and get the top 5 contributing factors.

Spark Internal Analysis

There are three stages that occur for this problem. In stage 1 we read the Parquet file from disk or memory if it is cached (as seen in the previous section). The RDDs created in stage 1 are parrallelCollectionRDD followed by a mapPartitionsRDD. Next, in stage 2 a FileScanRDD is created, followed by a MapPartitionRDD, then another MapParitionsRDD. In stage 3 a shuffledRowRDD is performed because we are performing a group by which is a wide transformation, and thus data is

shuffled across the cluster of nodes. Next, a MapPartsRDD is created, then another mapPartitionsRDD is created in the map step. Map is a narrow transformation, so the computations do not need to be shuffled across the cluster.



2. What percentage of accidents had alcohol as a contributing factor?

For this test, we first got the count of total accidents and stored in a val numTotalAccidents. Next, we filtered out accidents where alcohol involvement was a contributing factor for any of the vehicles involved in the accident ans stored the result in numAlcoholRelatedAccidents. Finally, we performed the following calculation to get the percentage.

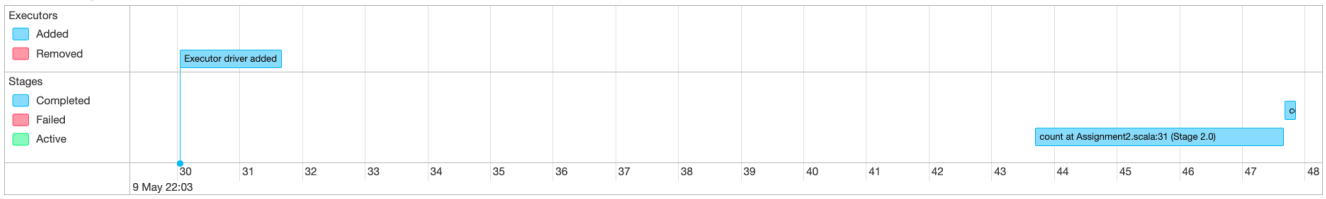
```
(numAlcoholRelatedAccidents * 100) / numTotalAccidents.toDouble
```

Spark Internal Analysis

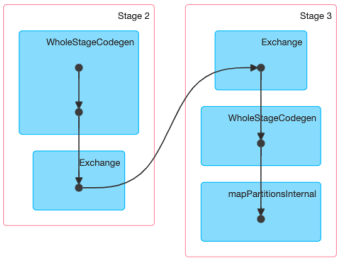
The data in stage 2 was split into 27 partitions, each executing a task. However, the data in stage 3 is only executing on a single partition.

▾ Event Timeline

Enable zooming

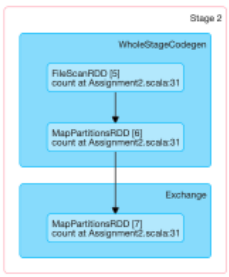


▾ DAG Visualization



▾ Completed Stages (2)

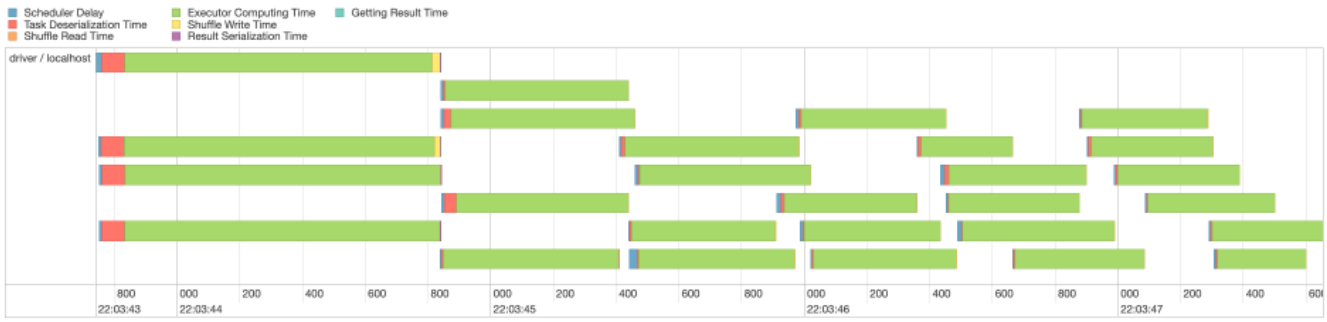
Stage Id ▾	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
3	count at Assignment2.scala:31 +details	2020/05/09 22:03:47	0.2 s	1/1			1592.0 B	
2	count at Assignment2.scala:31 +details	2020/05/09 22:03:43	4 s	27/27	12.4 MB			1592.0 B



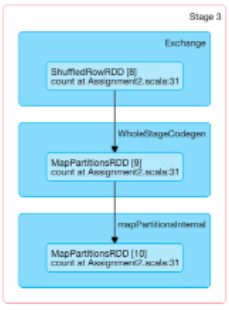
▸ Show Additional Metrics

▾ Event Timeline

Enable zooming



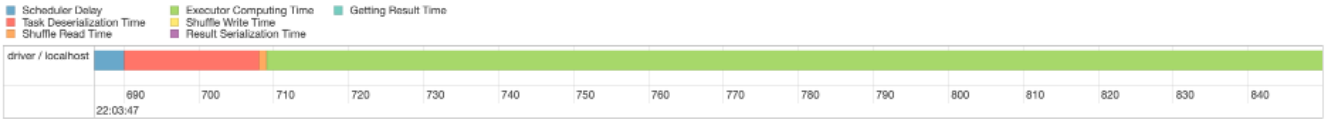
Summary Metrics for 27 Completed Tasks



▸ Show Additional Metrics

▾ Event Timeline

Enable zooming



Summary Metrics for 1 Completed Tasks

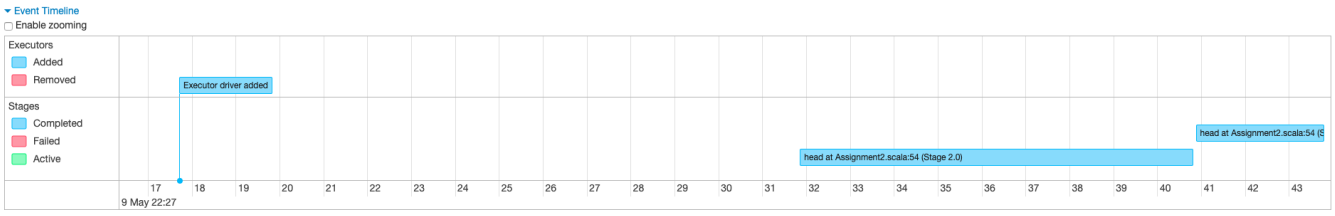
Metric	Min	25th percentile	Median	75th percentile	Max
Duration	0.1 s	0.1 s	0.1 s	0.1 s	0.1 s
GC Time	0 ms	0 ms	0 ms	0 ms	0 ms
Shuffle Read Size / Records	1592.0 B / 27	1592.0 B / 27	1592.0 B / 27	1592.0 B / 27	1592.0 B / 27

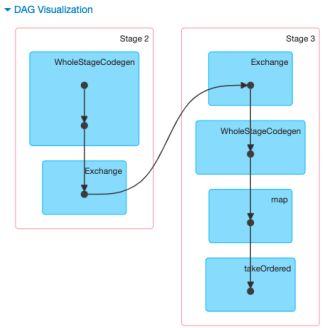
3. What time of day sees the most cyclist injures or deaths caused by a motor vehicle collision?

For this test, we first filter out accidents where there was at least one cyclist injury or fatality. Next, we perform a groupBy("CRASH_TIME") and counted the number of crashes for the various times throughout a 24 hour period. Finally, we order by count in descending order and get the top 3 times.

Spark Internal Analysis

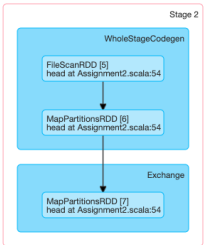
As we can see from stage 2, the tasks each took various amount of time, some longer and some shorter. Additionally, towards the end of each task, a shuffle write was performed. This is due to a wide transformation being performed, and thus the tasks will need to spend some time writing the results across the cluster. In total stage 2 took 1.1 minute to complete all of its 27 tasks. On the other hand, stage 3 executed over 200 tasks with a total time across all tasks of only 13 seconds. By this we can tell that shuffling is costly when it comes to execution time, especially if we are processing large amounts of data.

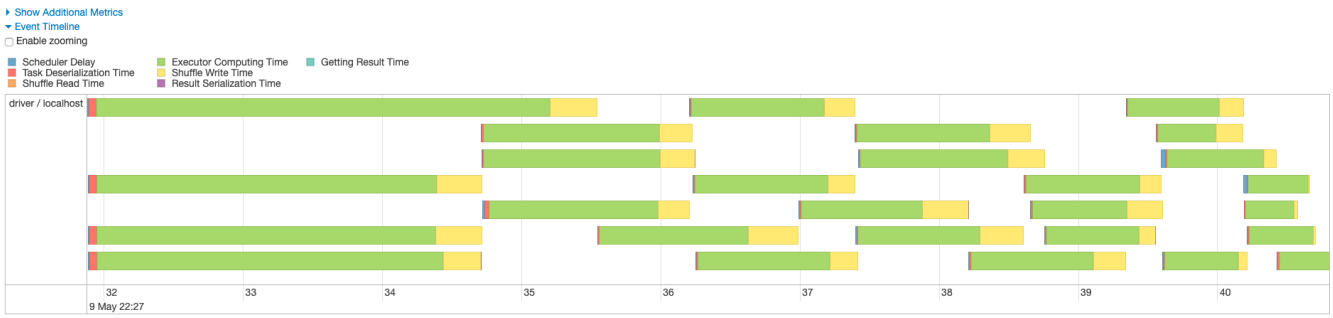




Completed Stages (2)

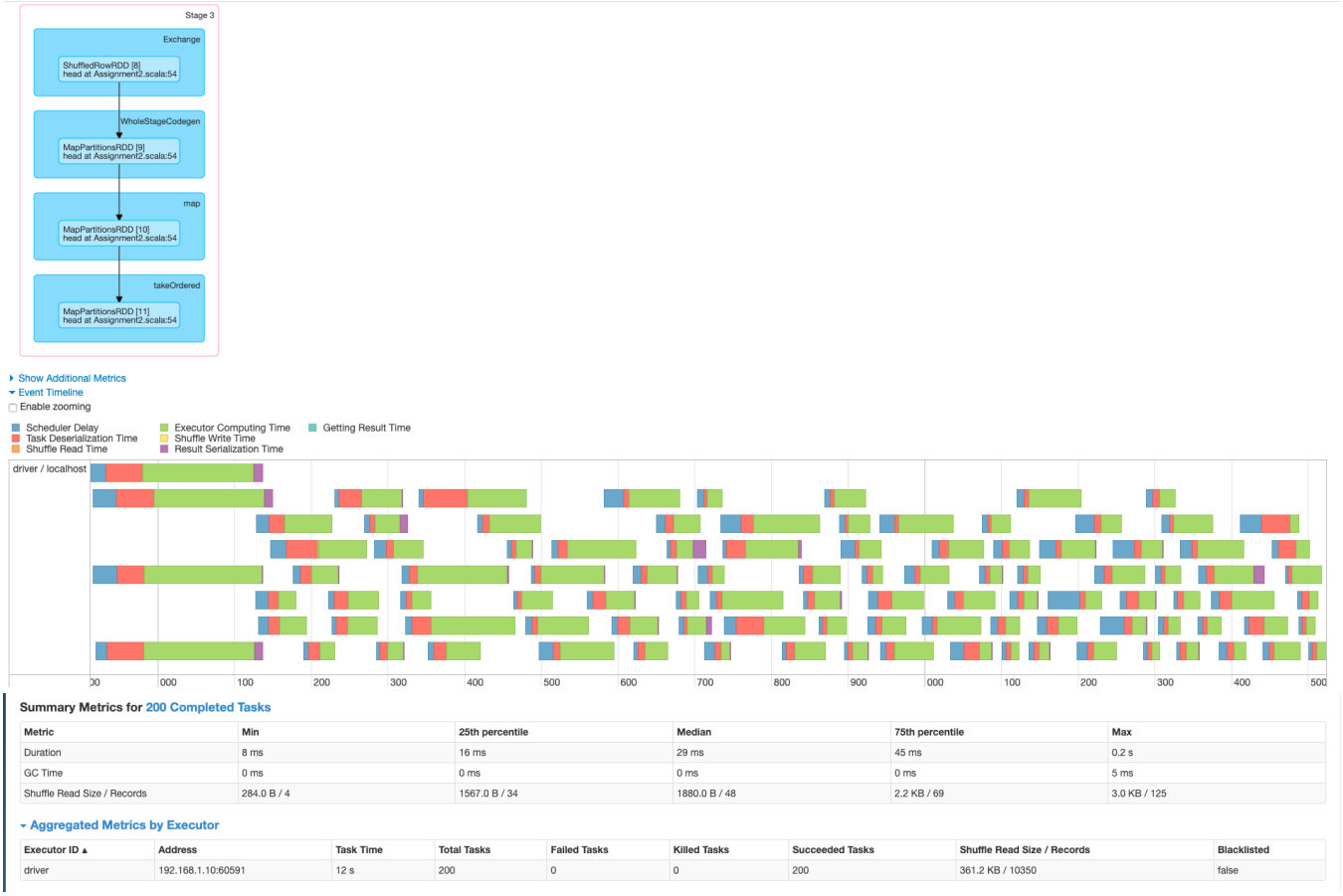
Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
3	head at Assignment2.scala:54 +details	2020/05/09 22:27:40	3 s	200/200			361.2 KB	
2	head at Assignment2.scala:54 +details	2020/05/09 22:27:31	9 s	27/27	57.0 MB			361.2 KB





Summary Metrics for 27 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	0.4 s	0.8 s	1 s	1 s	4 s
GC Time	0 ms	0 ms	13 ms	34 ms	0.2 s
Input Size / Records	580.7 KB / 44	1346.6 KB / 15036	2.2 MB / 42884	2.9 MB / 70449	4.0 MB / 590849

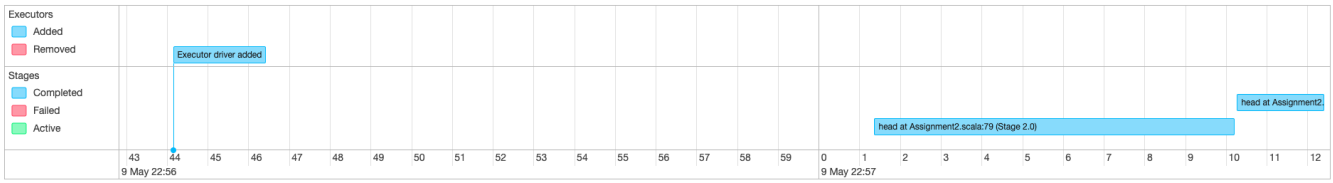


4. Which zip code had the largest number of nonfatal and fatal accidents?

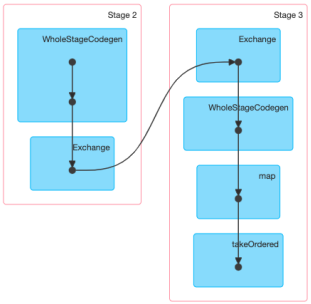
Find the zip code with the largest number of nonfatal and fatal accidents required several steps. First remove rows with null zip codes. Next create a new column "TOTAL_INJURED_OR_KILLED", which is the sum of all nonfatal and fatal injuries for each accident. Next, get the total number of nonfatal and fatal injuries per zip code by first performing a group by zip code. then use the agg and sum functions to sum up the values in the TOTAL_INJURED_OR_KILLED column per zip code. Afterwards, store the computed output per zip code in a column aliased as TOTAL_INJURIES_AND_FATALITIES. Finally, order the results by TOTAL_INJURIES_AND_FATALITIES in descending order and get the top 3.

Spark Internal Analysis

Since we are calling the filter function right after loading the dataset, Spark will perform a predicate pushdown, where it will push the filter down to the data source and perform the filter query before returning the result. This improves query performance as it is having the data source do the work before returning the result.

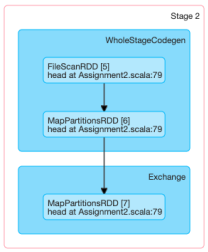


DAG Visualization



Completed Stages (2)

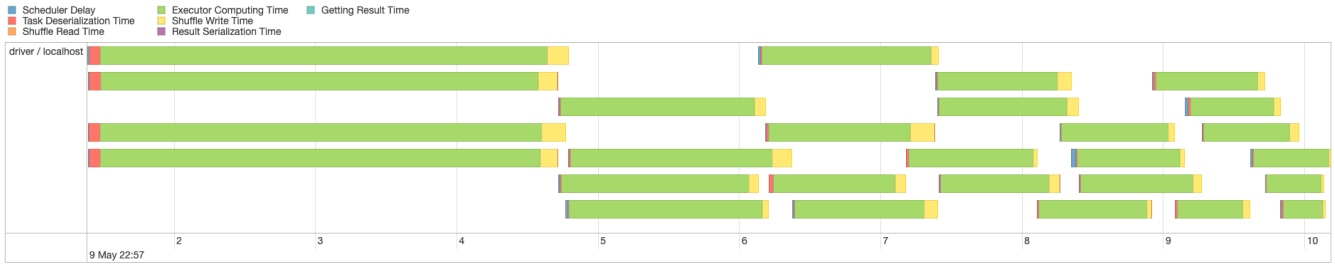
Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
3	head at Assignment2.scala:79 +details	2020/05/09 22:57:10	2 s	200/200			43.7 KB	
2	head at Assignment2.scala:79 +details	2020/05/09 22:57:01	9 s	27/27	29.1 MB			43.7 KB



Show Additional Metrics

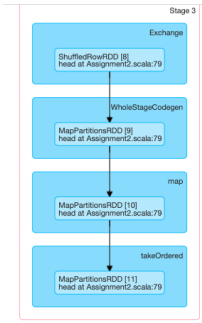
Event Timeline

Enable zooming



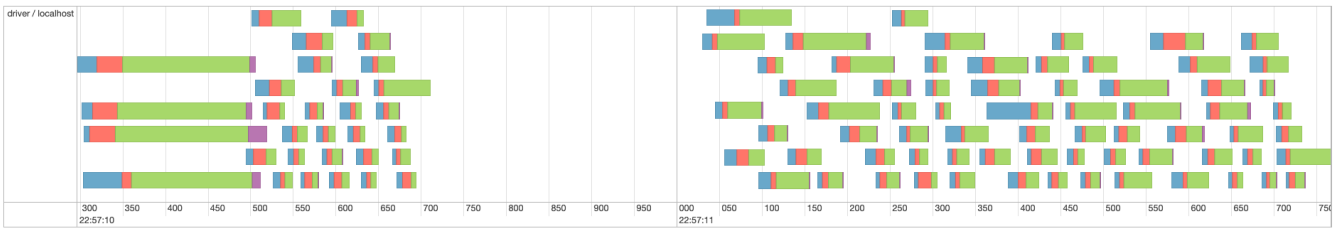
Summary Metrics for 27 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	0.3 s	0.8 s	0.9 s	1 s	3 s
GC Time	3 ms	9 ms	14 ms	24 ms	94 ms
Input Size / Records	427.8 KB / 29	845.6 KB / 13653	1114.4 KB / 43149	1365.0 KB / 70152	1748.8 KB / 105570



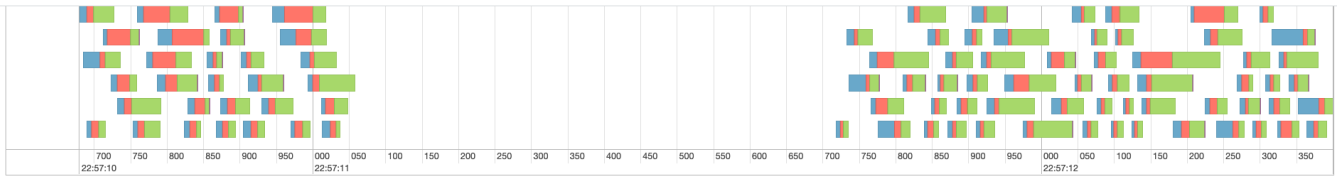
Show Additional Metrics
Event Timeline
Enable zooming

Scheduler Delay
Task Deserialization Time
Shuffle Read Time
Executor Computing Time
Shuffle Write Time
Result Serialization Time
Getting Result Time



Summary Metrics for 200 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	6 ms	10 ms	17 ms	28 ms	0.2 s



Summary Metrics for 200 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	6 ms	10 ms	17 ms	28 ms	0.2 s
GC Time	0 ms	0 ms	0 ms	0 ms	20 ms
Shuffle Read Size / Records	0.0 B / 0	0.0 B / 0	213.0 B / 3	355.0 B / 6	860.0 B / 15

Aggregated Metrics by Executor

Executor ID ▲	Address	Task Time	Total Tasks	Failed Tasks	Killed Tasks	Succeeded Tasks	Shuffle Read Size / Records	Blacklisted
driver	192.168.1.10:64089	9 s	200	0	0	200	43.7 KB / 667	false

Tasks (200)

Page: < 1 2 — 2 Pages. Jump to 2 . Show 100 items in a page. Go

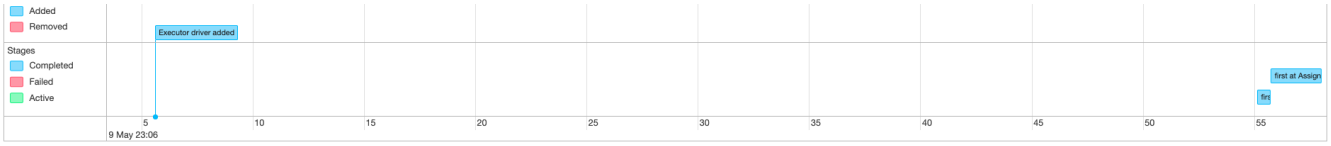
Index ▲	ID	Attempt	Status	Locality Level	Executor ID	Host	Launch Time	Duration	GC Time	Shuffle Read Size / Records	Errors
100	397	0	SUCCESS	ANY	driver	localhost	2020/05/09 22:57:11	7 ms		213.0 B / 3	
101	293	0	SUCCESS	PROCESS_LOCAL	driver	localhost	2020/05/09 22:57:10	28 ms		0.0 B / 0	
102	398	0	SUCCESS	ANY	driver	localhost	2020/05/09 22:57:11	77 ms		426.0 B / 6	
103	399	0	SUCCESS	ANY	driver	localhost	2020/05/09 22:57:11	21 ms		497.0 B / 7	
104	400	0	SUCCESS	ANY	driver	localhost	2020/05/09 22:57:11	13 ms		213.0 B / 3	
105	401	0	SUCCESS	ANY	driver	localhost	2020/05/09 22:57:11	48 ms		284.0 B / 4	
106	294	0	SUCCESS	PROCESS_LOCAL	driver	localhost	2020/05/09 22:57:10	21 ms		0.0 B / 0	
107	295	0	SUCCESS	PROCESS_LOCAL	driver	localhost	2020/05/09 22:57:10	10 ms		0.0 B / 0	
108	402	0	SUCCESS	ANY	driver	localhost	2020/05/09 22:57:11	22 ms		140.0 B / 2	
109	296	0	SUCCESS	PROCESS_LOCAL	driver	localhost	2020/05/09 22:57:10	18 ms		0.0 B / 0	
110	403	0	SUCCESS	ANY	driver	localhost	2020/05/09 22:57:11	13 ms		284.0 B / 4	
111	404	0	SUCCESS	ANY	driver	localhost	2020/05/09 22:57:11	18 ms		284.0 B / 4	
112	405	0	SUCCESS	ANY	driver	localhost	2020/05/09 22:57:11	36 ms		142.0 B / 2	
113	406	0	SUCCESS	ANY	driver	localhost	2020/05/09 22:57:11	7 ms		142.0 B / 2	
114	407	0	SUCCESS	ANY	driver	localhost	2020/05/09 22:57:11	12 ms		314.0 B / 6	

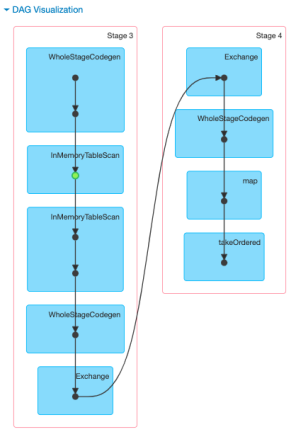
5. Which vehicle make, model, and year was involved in the most accidents?

In order to determine which vehicle make, model, and year was involved in the most accidents, we first remove any rows where the vehicle make, model, and year is null. Next, we did a groupBy ("VEHICLE_MAKE", "VEHICLE_MODEL", "VEHICLE_YEAR") then count and sort by descending order and get the first value, which is the most accidents. Likewise, to get the least accidents, we did the same previous steps but sort by ascending order and got the first value.

Spark Internal Analysis

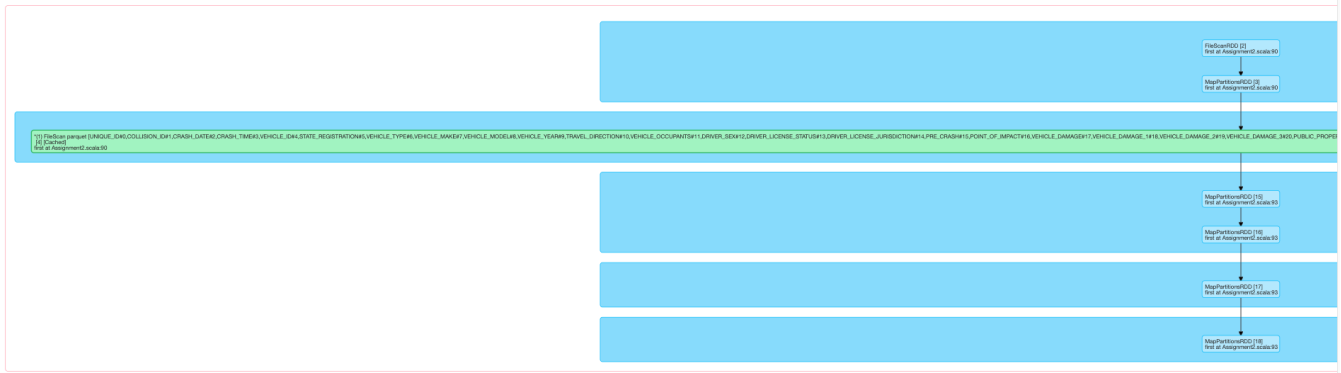
Simillar to test 4, test 5 is also performing a filter as it's first function in the test, so Spark will also perform a predicate pushdown as well.

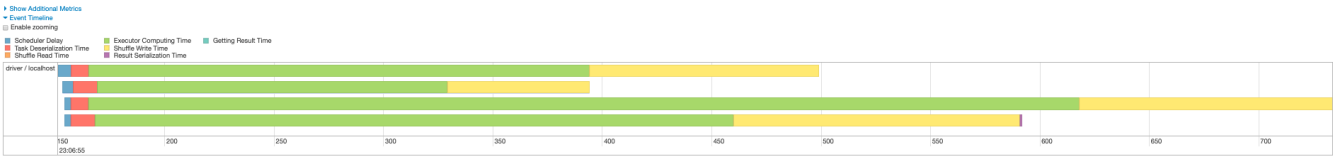




Completed Stages (2)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
4	first at Assignment2.scala:93 +details	2020/05/09 23:06:55	2 s	200/200			400.8 KB	
3	first at Assignment2.scala:93 +details	2020/05/09 23:06:55	0.6 s	4/4	304.0 MB			400.8 KB





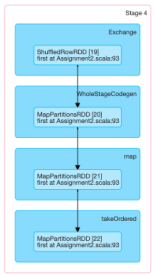
Summary Metrics for 4 Completed Tasks

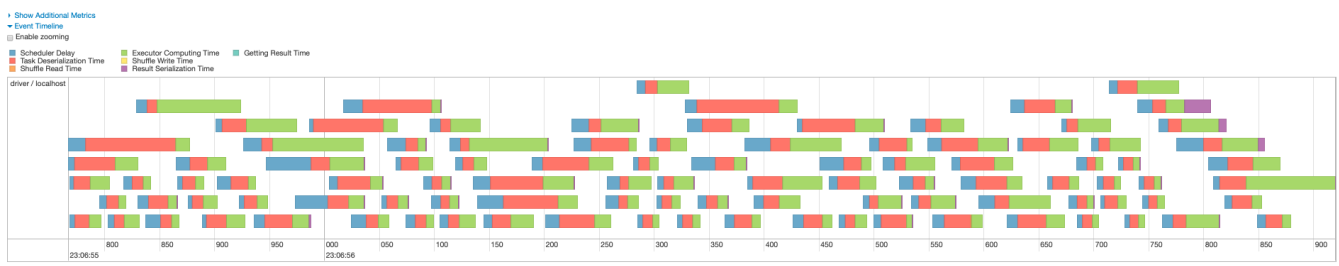
Metric	Min	25th percentile	Median	75th percentile	Max
Duration	0.2 s	0.3 s	0.4 s	0.6 s	0.6 s
GC Time	8 ms	8 ms	8 ms	8 ms	8 ms
Input Size / Records	56.0 MB / 57	56.1 MB / 57	83.4 MB / 96	108.5 MB / 127	108.5 MB / 127
Shuffle Write Size / Records	4.0 KB / 43	5.4 KB / 57	174.9 KB / 6015	216.6 KB / 7551	216.6 KB / 7551

Aggregated Metrics by Executor

Executor ID	Address	Task Time	Total Tasks	Failed Tasks	Killed Tasks	Succeeded Tasks	Input Size / Records	Shuffle Write Size / Records	Blacklisted
driver	192.168.1.10:65173	2 s	4	0	0	4	304.0 MB / 337	400.8 KB / 13666	false

Analysis





Summary Metrics for 200 Completed Tasks

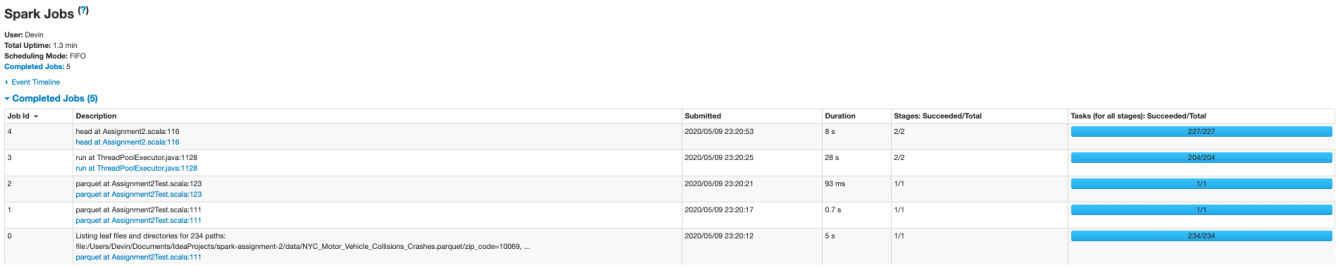
Metric	Min	25th percentile	Median	75th percentile	Max
Duration	5 ms	8 ms	13 ms	21 ms	0.1 s
GC Time	0 ms	0 ms	0 ms	0 ms	7 ms
Shuffle Read Size / Records	1481.0 B / 48	1856.0 B / 60	2.0 KB / 68	2.2 KB / 75	2.7 KB / 96

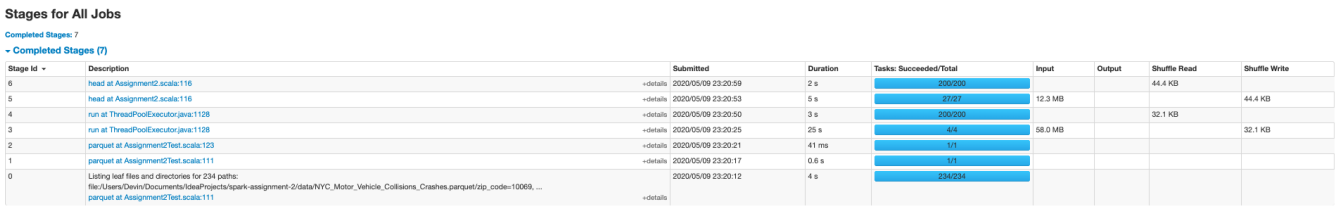
6. How do the number of collisions in an area of NYC correlate to the number of trees in the area?

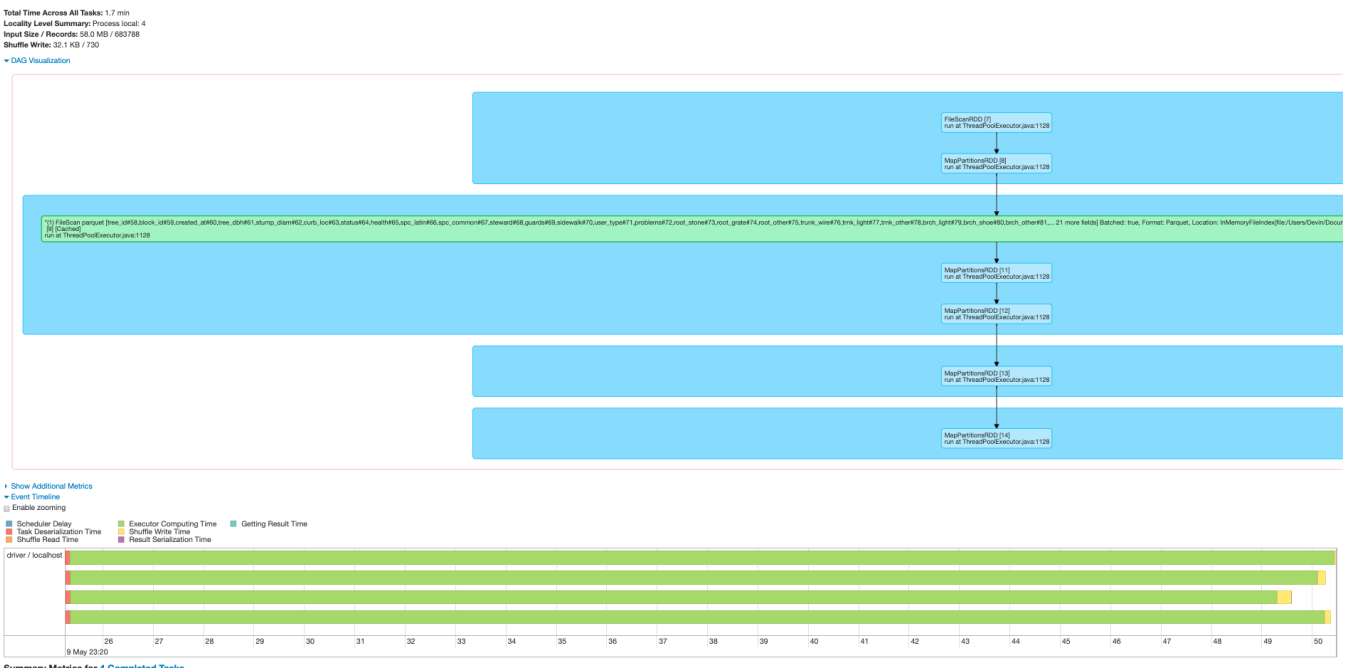
The NYC motor vehicle collisions - crashes and 2015 NYC tree census datasets were used for this test. We first rename the postcode column to ZIP_CODE on the treeCensus DataFrame. Next, we perform a groupBy on the treeCensus DataFrame to get the number of trees per zip code. We also, performed a groupBy on the collisions DataFrame to get the number of accidents per zip code. Finally, we did an inner equi-join of the collisions DataFrame with the treeCensus DataFrame using the "ZIP_CODE" collumn. Then finally, ordered by "TOTAL_CRASHES" in descending order.

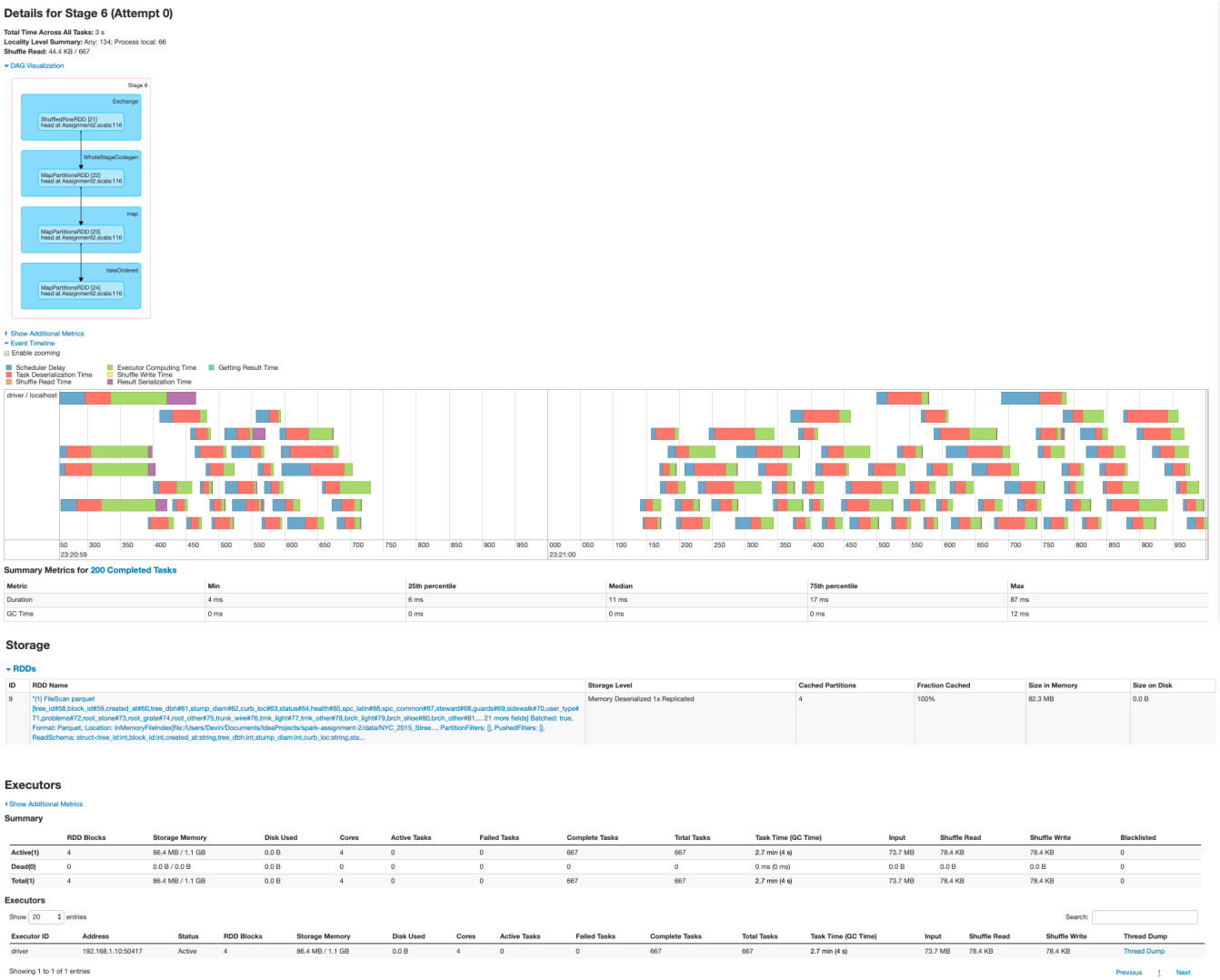
Spark Internal Analysis

There were only two parquet jobs for the other tests. However, this test uses two DataFrames instead of one, and thus required another job to read the additional data. Additionally, since a join is a wide transformation, shuffling needed to be performed. The code was ran on a single executor with 4 RDD Blocks, 4 cores CPU, and 1.1 GB RAM.









Project Overview

- Language: Scala
- Framework: Apache Spark
- Build tool: SBT
- Testing Framework: Scalatest

Running Tests

From Intellij

Right click on `ExampleDriverTest` and choose `Run 'ExampleDriverTest'`

From the command line

On Unix systems, test can be run:

```
$ ./sbt test
```

or on Windows systems:

```
C:\> ./sbt.bat test
```

Configuring Logging

Spark uses log4j 1.2 for logging. Logging levels can be configured in the file `src/test/resources/log4j.properties`

Spark logging can be verbose, for example, it will tell you when each task starts and finishes as well as resource cleanup messages. This isn't always useful or desired during regular development. To reduce the verbosity of logs, change the line `log4j.logger.org.apache.spark=INFO` to `log4j.logger.org.apache.spark=WARN`

Scala Worksheets

The worksheet `src/test/scala/com/spark/example/playground.sc` is a good place to try out Scala code. Add your code to the left pane of the worksheet, click the 'play' button, and the result will display in the right pane.

Note: The worksheet will not work for Spark code.

Documentation

- RDD: https://spark.apache.org/docs/latest/rdd-programming-guide.html
- Batch Structured APIs: https://spark.apache.org/docs/latest/sql-programming-guide.html

References

[1] National Highway Traffic Safety Administration. “NCSA Publications & Data Requests.” Early Estimate of Motor Vehicle Traffic Fatalities for the First Quarter of 2019, 2019, https://crashstats.nhtsa.dot.gov/Api/Public/ViewPublication/812783.

[2] (NYPD), Police Department. “Motor Vehicle Collisions - Crashes: NYC Open Data.” Motor Vehicle Collisions - Crashes | NYC Open Data, 8 May 2020, data.cityofnewyork.us/Public-Safety/Motor-Vehicle-Collisions-Crashes/h9gi-nx95.

[3] (NYPD), Police Department. “Motor Vehicle Collisions - Vehicles: NYC Open Data.” Motor Vehicle Collisions - Vehicles | NYC Open Data, 8 May 2020, data.cityofnewyork.us/Public-Safety/Motor-Vehicle-Collisions-Vehicles/bm4k-52h4.

[4] Department of Parks and Recreation. “2015 Street Tree Census - Tree Data: NYC Open Data.” 2015 Street Tree Census - Tree Data | NYC Open Data, 4 Oct. 2017, data.cityofnewyork.us/Environment/2015-Street-Tree-Census-Tree-Data/uvpi-gqnh.