Project Report

TOPIC: NYC Subway Ridership
DATE: May 15, 2020
BY: Michael Nip

Introduction

The New York City subway houses the highest number of stations of all metro systems in the world. On average, the system sees several million riders on a daily basis. Behind the scenes, the Metropolitan Transportation Authority (MTA) tracks and publishes a data feed of this activity. Every time a New Yorker goes through a turnstile of a station, that turnstile increments its entry and exit counters. All turnstiles in all stations are built to record this information on a 24-hour basis and in 4-hour intervals, submit their records to a centralized data feed maintained by the MTA. This data feed is known as Turnstile Data and it can be found under the developer section on MTA's website.

To better understand subway ridership at a more granular level, my project focused on constructing a unified database of hourly rider activity across all stations for the past five years. Then, using this database, building data visualizations and summaries that showed and compared rider activity trends across all stations in the subway system, varying both seasons and times of day. With regards to implementation, all source code was written in Python and the big data technologies utilized were Apache Kafka, Spark, and Zeppelin.

Overview of Data Pipeline

The Turnstile Data is split into 270 text files, each containing approximately 200,000 data records and representing one week for every week in the past five years. At the start of the pipeline, I used Kafka to extract the data records from every file and append them to a database in the Hadoop File System (HDFS). Next, I used Spark to clean, transform, and aggregate the database into an ingestible size for analysis. Lastly, I used Zeppelin to explore, visualize, and summarize the database as specified above.

Apache Kafka: ETL Process

In order to make use of Kafka, I used PyKafka, which is a Kafka client for Python that provides implementations of Kafka producers and consumers. To this end, there are two major client variants available on the current open-source market, PyKafka and KafkaPython. I chose PyKafka because it strives to keep its API as pythonic as possible, making use of Python-specific features where applicable for simplicity in client code. By contrast, KafkaPython seeks to imitate the functionality of the Java client.

Since Kafka requires a ZooKeeper server, I first installed and set up a ZooKeeper server on the default port 2181, then started a Kafka server, pointing to said ZooKeeper server. Once set up, I created a topic with a replication factor of 3 and 10 partitions for a producer and set of consumers to produce to and consume from, respectfully. The replication factor is the number of replications, or copies, of a partition stored in other partitions. These replications ensure that partitions are not lost in the case of node failure in the Kafka cluster. A partition represents a piece of a topic, and it enables the multiple consumers to read messages from a topic in parallel. I specified this replication factor and partition combination in order to allow for sufficient parallelism with balanced replication latency, which is the additional time and work required to maintain partition replicas.

To extract every data record from the Turnstile Data, I created a producer, configured as follows:

```
producer = topic.get_producer(
  max_queued_messages = 1000,
  min_queued_messages = 1000,
  linger_ms = 5000,
  queue_empty_timeout_ms = 0,
  block_on_queue_full = True,
  sync = False,
  delivery_reports = True,
  pending_timeout_ms = 5000,
  auto_start = True
)
```

A producer maintains a message queue, which acts like a holding pen for messages waiting to be sent to the topic. Both *max_queued_messages* and *min_queued_messages* were set to 1,000, meaning that the producer produces messages from the queue in batches of at most 1,000 at a time. At the same time, *linger_ms* and *queue_empty_timeout_ms* requires the producer to "linger" for 5 seconds when the queue does not have at least 1,000 messages before producing the current batch. The *block_on_queue_full* indicates that the producer should be blocked from adding messages to the queue when it is full and unblocked when space becomes available. Finally, *sync* and *delivery_reports* ensure that the producer produces messages asynchronously without waiting for the response from the topic broker and an internal queue of delivery reports for each message produced is maintained. All these parameters made for a higher throughput of messages. The producer is implemented in this way to produce messages in reasonably sized batches and with periodic checks of delivery reports in order to reproduce messages that previously failed to deliver. Under this configuration and implementation, approximately 200,000 messages were produced every 3 seconds. Given the Turnstile Data, which is comprised of approximately 54 million messages (270 source files x 200,000 data records each), the producer produced the entire data feed to the topic in about 22.5 minutes.

To transform and load every data record, I created a consumer, configured as follows:

```
consumer = topic.get_balanced_consumer(
  consumer_group = b'consumers',
  auto_commit_enable = True,
  auto_commit_interval_ms = 60000,
  auto_offset_reset = OffsetType.EARLIEST,
  consumer_timeout_ms = 5000,
  zookeeper_connect = 'localhost:2181',
  auto_start = True,
  reset_offset_on_start = False
)
```

As previously mentioned, messages in the topic were organized into 10 partitions and again, in order to maximize throughput, 10 consumers with the above configuration were used in a multithreaded program to consume these messages. Note that no more than 10 consumers were used because there cannot be more consumers than partitions, otherwise the excess consumers would not be assigned a partition. In this configuration, *consumer_group* and *zookeeper_connect* in every consumer were set to the same values to enable each consumer to coordinate their consumption with each other. This ensures that no message is wastefully consumed more than once. The parameters *auto_commit_enable* and *auto_commit_interval_ms* requires that each consumer saves its offset into the topic, or how far into the topic it has already consumed, every minute. Lastly, *auto_offset_reset* and *reset_offset_on_start* ensures the consumers always consume from the last place they left off in the topic. For each message consumed, a consumer transforms the message into a row of a Spark data frame and appends that onto a larger data frame that represents that consumer's current collection of

messages. When the data frame reaches a row count of 50,000, the consumer writes this data frame into HDFS as a text file in append mode and the data frame is reset for this process to repeat for every subsequent group of 50,000 messages. The transformed collection of messages is written into HDFS in batches because writes to the HDFS are expensive operations, so this helps minimize the overall load time. Under this configuration and implementation, 200,000 messages were processed (consumed, transformed, and loaded) every 3 minutes, which made for a total running time of 13.5 hours to fully process all Turnstile Data messages.

Apache Spark: Data Cleaning, Augmentation, and Aggregation

In order to make use of Spark, I used PySpark, which is the Python API for Spark. With the Turnstile Data now loaded into a database in HDFS, I created a Spark application that takes this database, transforms and reduces it to manageable size, and writes the newly created database back into HDFS for visualization and analysis. The process of converting the database into an analyzable format can be broken down into three steps: cleaning, augmentation, and aggregation. To provide context for these steps, below is a preview of the database:

```
C/A,UNIT,SCP,STATION,LINENAME,DIVISION,DATE,TIME,DESC,ENTRIES,EXITS
A002,R051,02-00-00,LEXINGTON AVE,NQR456,BMT,02/28/2015,03:00:00,REGULAR,0005023331,0001701900
A002,R051,02-00-00,LEXINGTON AVE,NQR456,BMT,02/28/2015,07:00:00,REGULAR,0005023340,0001701906
A002,R051,02-00-00,LEXINGTON AVE,NQR456,BMT,02/28/2015,11:00:00,REGULAR,0005023421,0001701986
A002,R051,02-00-00,LEXINGTON AVE,NQR456,BMT,02/28/2015,15:00:00,REGULAR,0005023619,0001702052
A002,R051,02-00-00,LEXINGTON AVE,NQR456,BMT,02/28/2015,19:00:00,REGULAR,0005024000,0001702109
A002,R051,02-00-00,LEXINGTON AVE,NQR456,BMT,02/28/2015,23:00:00,REGULAR,0005024211,0001702130
A002,R051,02-00-00,LEXINGTON AVE,NQR456,BMT,03/01/2015,03:00:00,REGULAR,0005024269,0001702142
A002,R051,02-00-00,LEXINGTON AVE,NQR456,BMT,03/01/2015,07:00:00,REGULAR,0005024275,0001702150
A002,R051,02-00-00,LEXINGTON AVE,NQR456,BMT,03/01/2015,11:00:00,REGULAR,0005024316,0001702205
```

C/A and SPC stand for control area and subunit channel position, both of which, along with the UNIT, uniquely identify a turnstile. Other fields of interest are STATION, DATE, TIME, ENTRIES, and EXITS. Note that both the ENTRIES and EXITS fields represent the cumulative count of entries and exits up until the specified DATE and TIME, not the incremental count since the last feed. For example, the first and second data rows indicate that there were 9 entries and 6 exits between 3:00 AM and 7:00 AM on February 28, 2015 at Lexington Avenue station. Further, a turnstile has limited memory and its entry and exit counters reset to 0 when the maximum count is reached, and that reset is periodically reflected in these fields.

In the first step, the DATE, TIME, ENTRIES, and EXITS fields are casted into date, integer, and long integer variables. All fields in the database are of string type because the database had to be saved as a text file during the ETL process. To calculate the incremental count of entries and exits, the ENTRIES and EXITS fields are first lagged by 1 row such that each row contains the current and last cumulative counts. Then, the difference between the current and last cumulative count is taken as the incremental count if the current count is greater than the last, otherwise the current count is taken as the incremental count. This conditional calculation accounts for the counter reset highlighted earlier. Next, the database is filtered to the exact and relevant date range of March 1, 2015 to April 30, 2020. It is then further filtered to remove outliers, which are caused by feed records like the below:

```
N601,R319,00-00-00,LEXINGTON AV/63,F,IND,03/11/2018,04:00:00,REGULAR,0003113633,0009883240
N601,R319,00-00-00,LEXINGTON AV/63,F,IND,03/11/2018,06:00:00,RECOVR AUD,0000000007,0000000018
N601,R319,00-00-00,LEXINGTON AV/63,F,IND,03/11/2018,08:00:00,REGULAR,0003113674,0009883403
```

There are primarily two values in the DESC field, indicating either regular or recover audit feeds. Recover audit feeds are feeds that are produced after a communication outage in a given turnstile. However, as shown above, sometimes turnstiles arbitrarily reset the regular feed counter in their recover audit feeds and resume from the last regular feed counter in the next regular feed. This can cause erroneous entry and exit calculations. In the above example, the incremental entries and exits would be correctly calculated in the first and second row, but incorrectly in the third, since the cumulative entries and exits in the second row would cause an erroneous calculation of 3,113,674 entries and 9,883,403 exits between 6:00 AM to 8:00 AM on March 11, 2018 on Lexington Avenue – 63 Street Station. Unfortunately, since many recover audit feeds are valid feeds that do not always reset the regular feed counter, recover audit feeds cannot be simply excluded altogether. In other words, these arbitrary counter resets are not uniquely identifiable in the database. As such, these specific outliers are identified as rows where the incremental entry or exit is greater than or equal to 88,000. This threshold is the estimated maximum entry and exit count of a given turnstile in a 4-hour interval, calculated based on the following: (at most, 100 people in a subway car) x (at most, 11 subway cars in a train) x (on average, 20 trains per hour given a 3-minute wait time per train) x (4 hours).

In the second step, the database is augmented with fields for the season, time of day, as well as latitude and longitude coordinates. The season is calculated using the DATE field and contains values for five categories: spring, summer, fall, winter, and coronavirus. The four seasons are based on meteorological seasons in the northern hemisphere, where spring runs from March to May, summer from June to August, fall from September to November, and winter from December to February. The coronavirus period is a special case that runs from March to April 2020. The time of day is calculated using the TIME field and contains values for three categories: early-day, day, and night, each of which corresponds, in the order they are listed, to 8-hour intervals in a 24-hour timespan starting from 12:00 AM. The latitude and longitude coordinates are merged onto the database from a lookup of station locations, which is also maintained by the MTA.

In the third step, the database is aggregated twice, first to calculate the sum of entries and exits by station, season, time of day, and date, then to calculate the daily average entries and exits by station, season, and time of day. Within each grouping of station, season, and time of day, the first aggregation increases the level of granularity in the database from turnstile to station level, while preserving the variance across days. The second aggregation then calculates the within-group average of this variance. Finally, the cleaned database is written back into HDFS for further analysis.

Apache Zeppelin: Data Visualization and Analysis

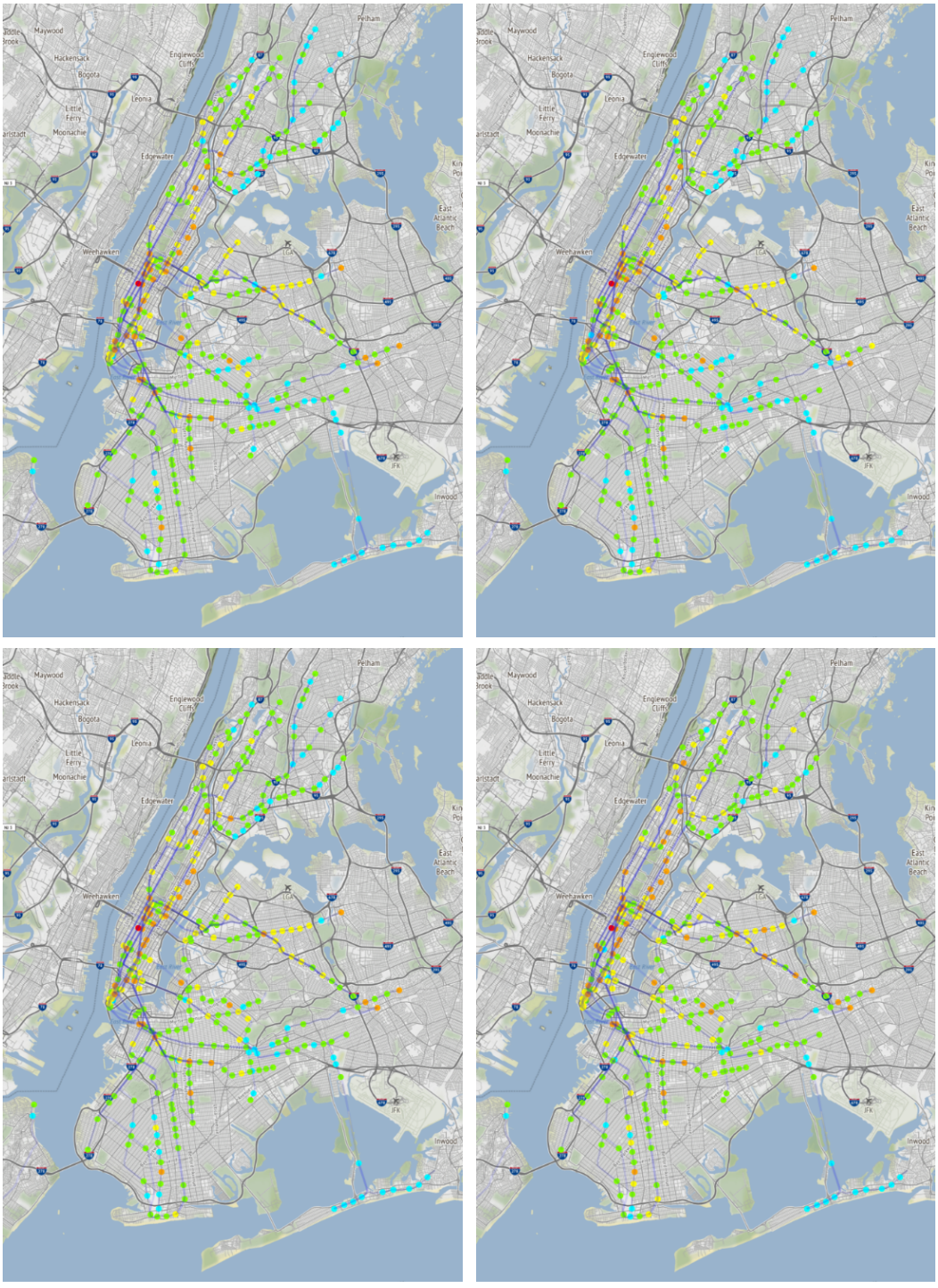
Within the Zeppelin interface, I created a notebook with both Python and Spark interpreters. The goal here was to use the notebook to generate several geospatial data visualizations that displayed ridership activity as points on the subway system lines on a map of New York City for every combination of season and time of day, separately for each pattern (entries and exits) and where every station would be the proxy for ridership activity and would be plotted as a point of a certain color that corresponded to that station's daily average entries or exits for said combination. The season and time of day categories included spring, summer, fall, winter, and coronavirus as well as early-day for 12:00 AM – 8:00 AM, day for 8:00 AM – 4:00 PM, and night for 4:00 PM – 12:00 AM, respectfully. In total, there were 15 combinations of season and time of day categories, which made for 30 total maps since each combination was repeated separately for entries and exits. I decided to use GeoPandas and Matplotlib to accomplish this, as both are powerful Python libraries for data visualization and PySpark provides a simple method to convert from a Spark to Pandas data frame.

To start, I first used Spark to import the cleaned database stored in HDFS and label each data row with entry and exit color categories to inform the visualization. The categories for both entries and exits are red if the daily average falls into the range (50000, ∞), orange if (10000, 50000], yellow if (5000, 10000], chartreuse if (1000, 5000], otherwise cyan. Note that the underlying thresholds of these color labels are homogenous across season, time of day, and pattern in order to be able to compare the color variance between seasons as well as times of

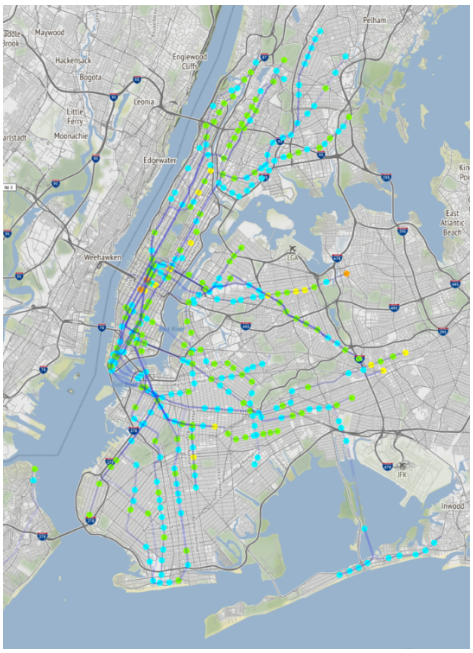
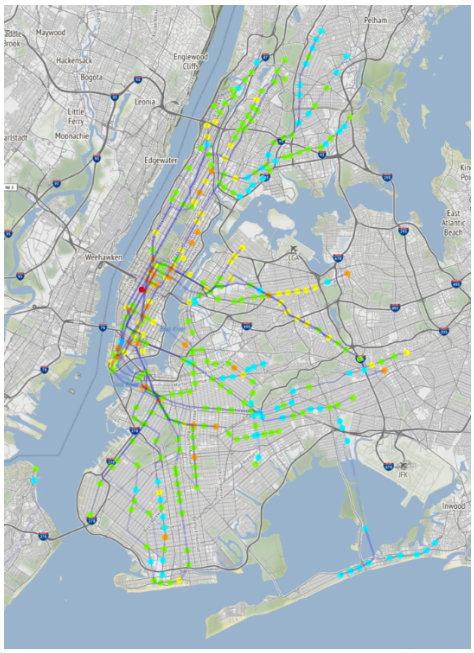
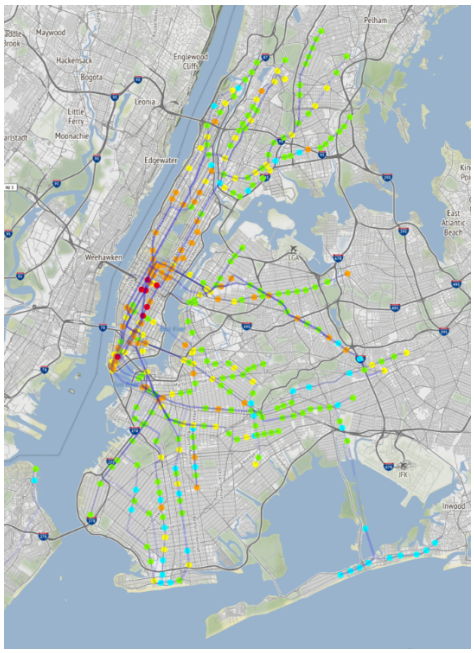
day. The labeled database was then converted from a Spark to Pandas to GeoPandas data frame and this was used to generate each of the maps in a loop that filtered and plotted the stations in the data frame as colored points according to their label for every combination of season, time of day, and pattern.

Upon review of the 30 generated maps, the most interesting findings were that there is little variation in entry and exit patterns between seasons for all times of day, entries are more concentrated in Midtown and Uptown Manhattan at night yet sparsely scattered across New York City during the early-day and day, and exits are relatively more concentrated in Midtown and Uptown Manhattan during all times of day. Further, similar entry and exit patterns were observed during the coronavirus period, but at an unsurprisingly smaller scale.

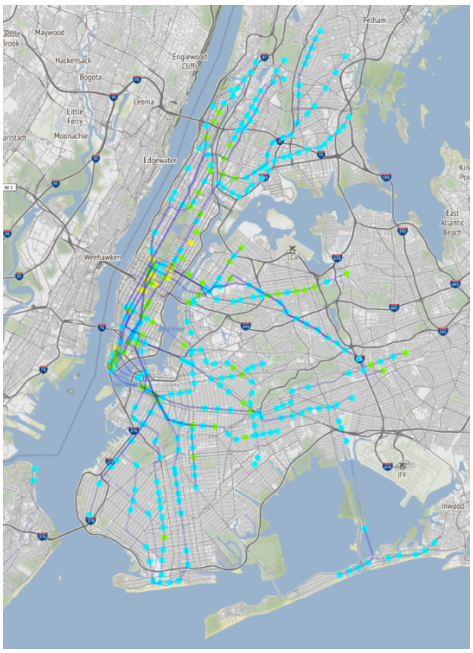
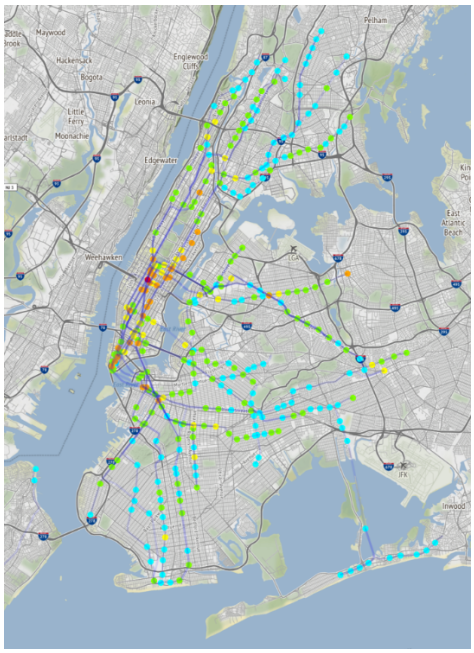
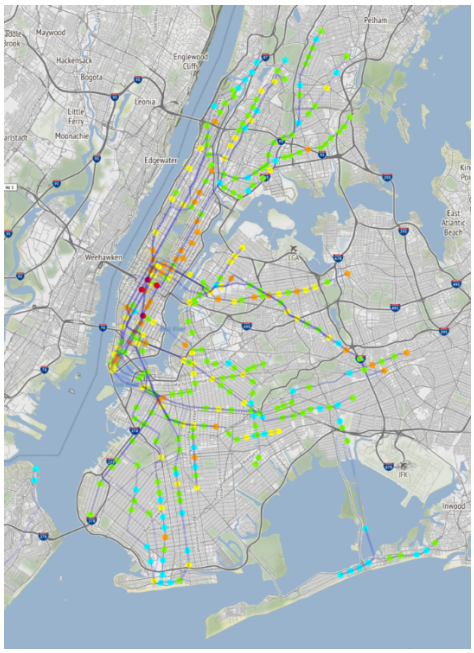
To the first finding, below are maps of entries during the day, for the spring (top-left), summer (top-right), fall (bottom-left), and winter (bottom-right) seasons:



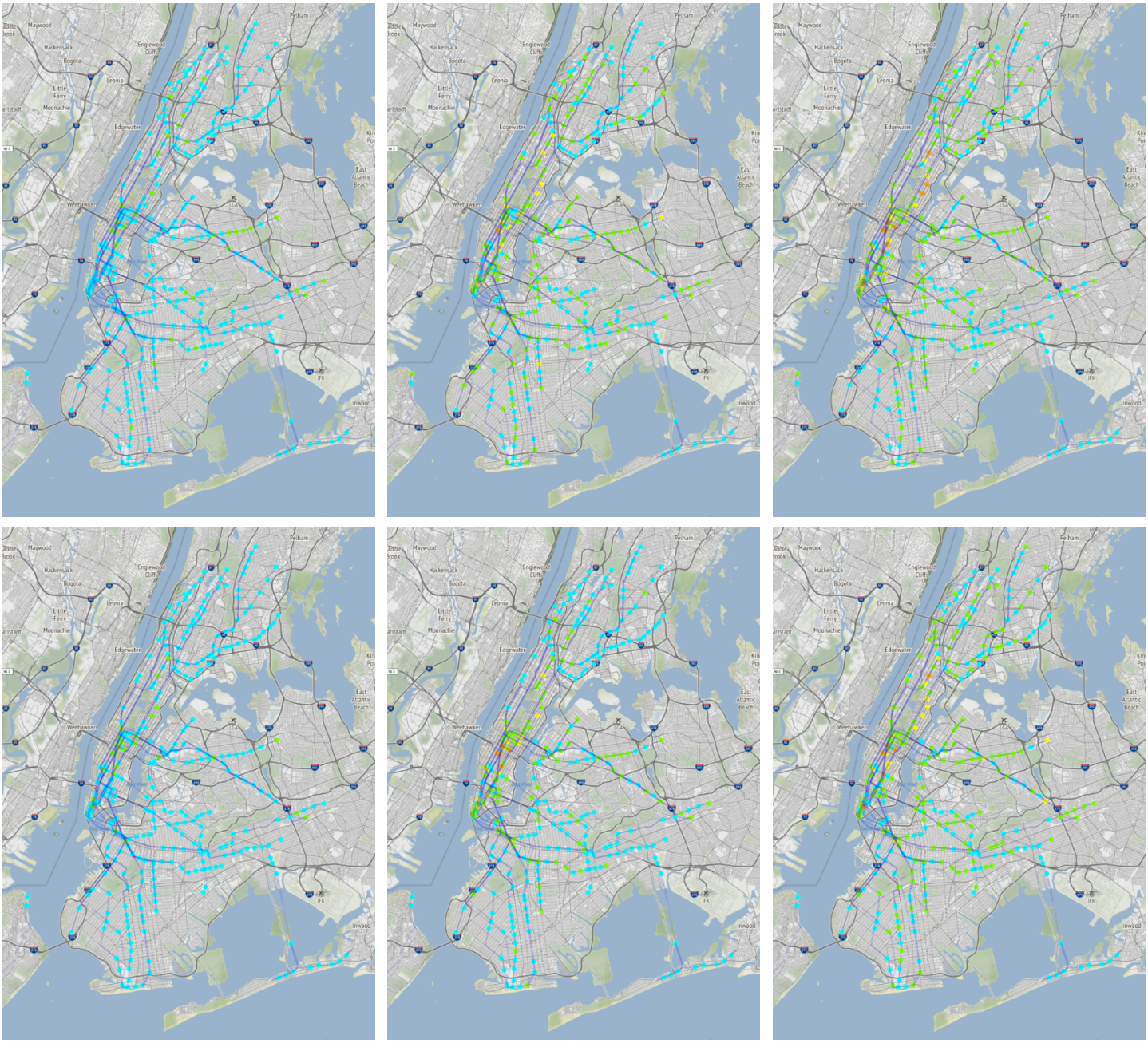
To the second finding, below are maps of entries during the spring season, for the early-day, day, and night times of day from left to right:

To the third finding, below are maps of exits during the spring season, for the early-day, day, and night times of day from left to right:

To the fourth finding, below are maps of entries (first row) and exits (second row) during the coronavirus period, for the early-day, day, and night times of day from left to right:



To further substantiate the findings from the maps, below are summary tables of descriptive statistics of the daily average entries (first table) and exits (second table) for all seasons and times of day. Both tables show clear variance amongst times of day within seasons, but little variance amongst seasons within times of day. A few statistics worth detailing are the stations with 0 daily average exits, which are comprised of Staten Island stations such as St. George and Tompkinsville as well as stations with daily average entries and exits that well exceed the means of the distributions, which are consistently comprised of central Manhattan stations such as Times Sq – 42 St, Grand Central – 42 St, 34 St – Herald Sq, 34 St – Penn Station, and 14 St – Union Sq irrespective of season and time of day.

Time of Day	Season	Mean	Std Dev	Min	Q1	Med	Q3	Max
Early-day	Spring	1359	1830	20	292	763	1695	15099
Early-day	Summer	1248	1819	8	176	627	1577	15002
Early-day	Fall	1425	1729	26	446	892	1708	14062
Early-day	Winter	1471	1461	19	597	1026	1853	12087
Early-day	Coronavirus	470	543	4	151	291	586	3518
Day	Spring	4896	6397	34	1387	2809	5924	53747
Day	Summer	4523	6083	33	1226	2537	5392	51652
Day	Fall	5133	6486	44	1529	3169	6130	54041
Day	Winter	5669	6695	75	1886	3586	6913	54005
Day	Coronavirus	1488	1664	15	482	965	1783	12155
Night	Spring	7878	12283	136	1810	3811	8978	94221
Night	Summer	7736	12044	144	1758	3658	8716	93886
Night	Fall	7554	11976	143	1731	3594	8585	92406
Night	Winter	5933	10416	22	1085	2319	6212	75509
Night	Coronavirus	2029	2767	35	568	1144	2271	17694

Time of Day	Season	Mean	Std Dev	Min	Q1	Med	Q3	Max
Early-day	Spring	878	1409	0	156	394	840	9058
Early-day	Summer	847	1430	0	139	349	817	9632
Early-day	Fall	932	1391	0	212	468	961	9552
Early-day	Winter	980	1332	0	262	541	1142	11708
Early-day	Coronavirus	364	512	0	86	195	378	3256
Day	Spring	3939	7467	0	564	1499	3591	58276
Day	Summer	3720	7158	0	526	1400	3404	56898
Day	Fall	4102	7631	0	637	1613	3753	59617
Day	Winter	4430	7967	0	817	1699	4012	58458
Day	Coronavirus	1332	2117	0	272	627	1310	14592
Night	Spring	5909	8672	0	1580	3200	6680	72805
Night	Summer	5880	8684	0	1553	3191	6784	69509
Night	Fall	5743	8443	0	1515	3277	6498	74872
Night	Winter	4653	7083	0	1163	2529	5022	74928
Night	Coronavirus	1804	2286	0	556	1134	2163	19645

For a comprehensive view of all 30 generated maps, please see the Appendix at the end of this report.

Conclusion

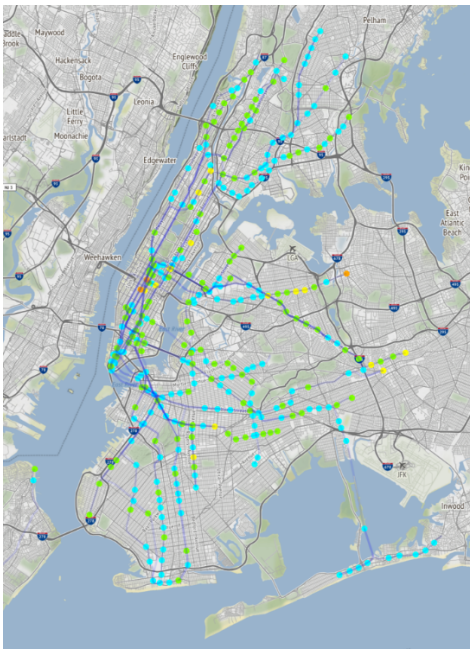
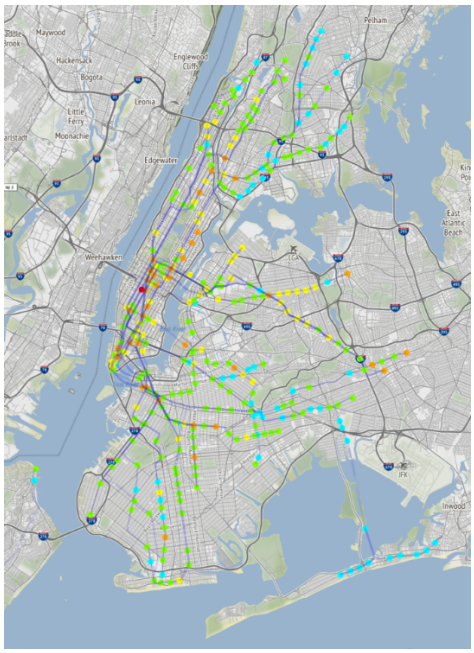
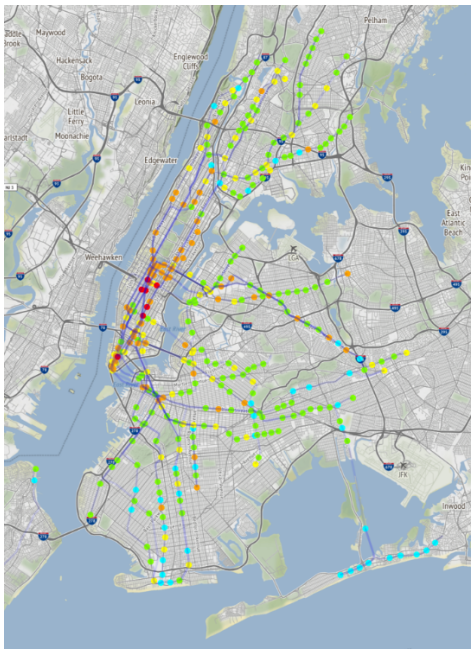
The Turnstile Data posed interesting challenges in the design and implementation of the data pipeline. With regard to the ETL process, Kafka provided an efficient solution for moving the data feed from raw data files into HDFS. The producer could produce an average of 200,000 messages in a matter of seconds, whereas the consumer could consume this amount in minutes. Given the size of the Turnstile Data, the producer and consumer performed satisfactorily and within reasonable time frames. If they were put to the test with more than 54 million messages, they could be scaled by increasing the number of partitions in the topic as well as inflating the parameters to handle longer and larger intakes of data. For example, the minimum and maximum queued messages in the producer could be increased in order to further consolidate throughput. In the case of an extreme increase in demand of message intake, an alternative option for further optimizing the consumption part of this ETL process would be to replace the consumer with Kafka Connect. Kafka Connect is a framework that enables Kafka to directly integrate with other systems, including HDFS. The idea here would be to export data in a Kafka cluster to HDFS, or sink in this case, which would bypass the need for a consumer.

As for the data cleaning, augmentation, and aggregation, Spark and its distributed framework for processing data was an appropriate and scalable fit for the task. Since data rows are spread across nodes in a cluster, committing data cleaning and augmentation were straightforward and fast, as the code for these actions were delivered to each node and performed in parallel. Spark employs a similar method for data aggregation except with the addition of shuffling in order to allow for partial aggregation as it maps the aggregation over the entire data. Again, data was filtered where possible and as early as possible to eliminate unnecessary work. The biggest challenge in working with the Turnstile Data was that entry and exit observations were in a cumulative format. As a result, the calculation of incremental entries and exits was prioritized before any filtering or aggregation since this lag-based calculation was based on the order of raw data rows as well as dependency of observations between raw data rows. Note that this is why there was no preemptive data cleaning during the transformation stage in the ETL process.

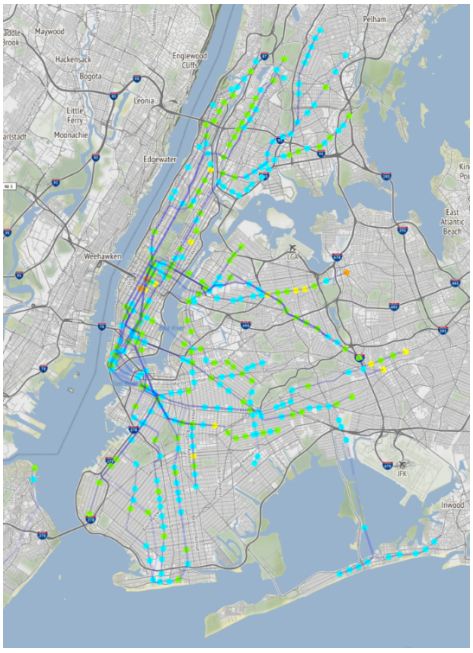
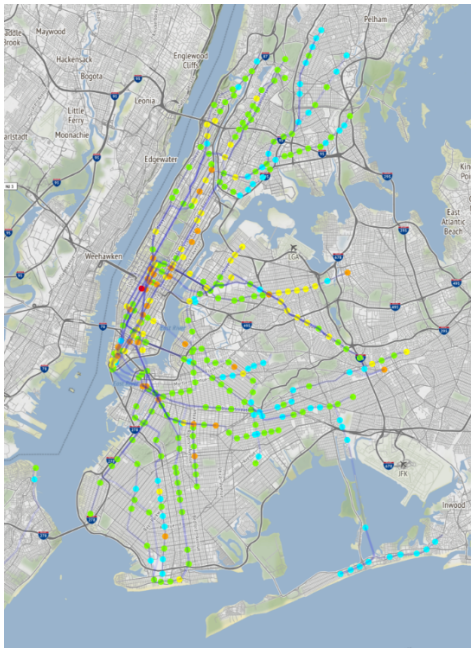
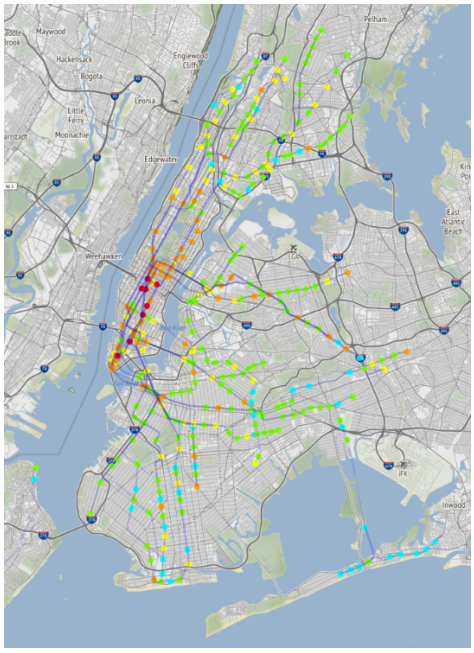
Lastly, Zeppelin was used to create the map and summary table visualizations. Since Zeppelin can be configured with a Spark interpreter and, by way of Spark, stream data from HDFS, it served as a seamless tool for quickly testing and producing visualizations for varying cuts of the cleaned database stored on HDFS. The visualizations produced via Zeppelin revealed both expected and unexpected findings about the entry and exit patterns of subway ridership. To recap, there was little variation in entry and exit patterns between seasons for all times of day, entries were more concentrated in Midtown and Uptown Manhattan at night yet sparsely scattered across New York City during the early-day and day, and exits were relatively more concentrated in Midtown and Uptown Manhattan during all times of day. Further, similar entry and exit patterns were observed during the coronavirus period, but at an unsurprisingly smaller scale. These findings provide deeper insight to the daily and seasonal movement of New Yorkers insofar as public transportation is concerned. This kind of data can prove to be particularly useful for public transportation design as well as urban planning. It is also interesting to view these findings from a sociological point of view and helps inform our understanding of behavior of New Yorkers across time.

Appendix

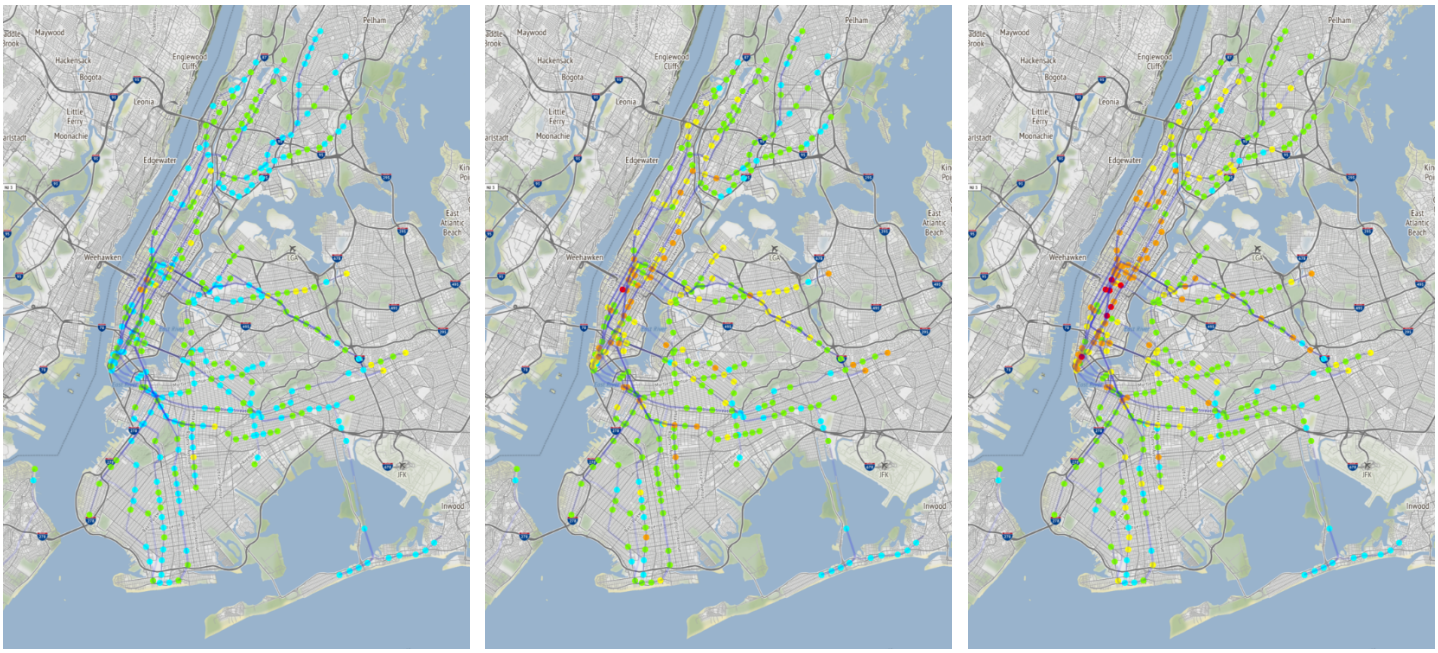
Entries during the spring season, for the early-day, day, and night times of day from left to right:

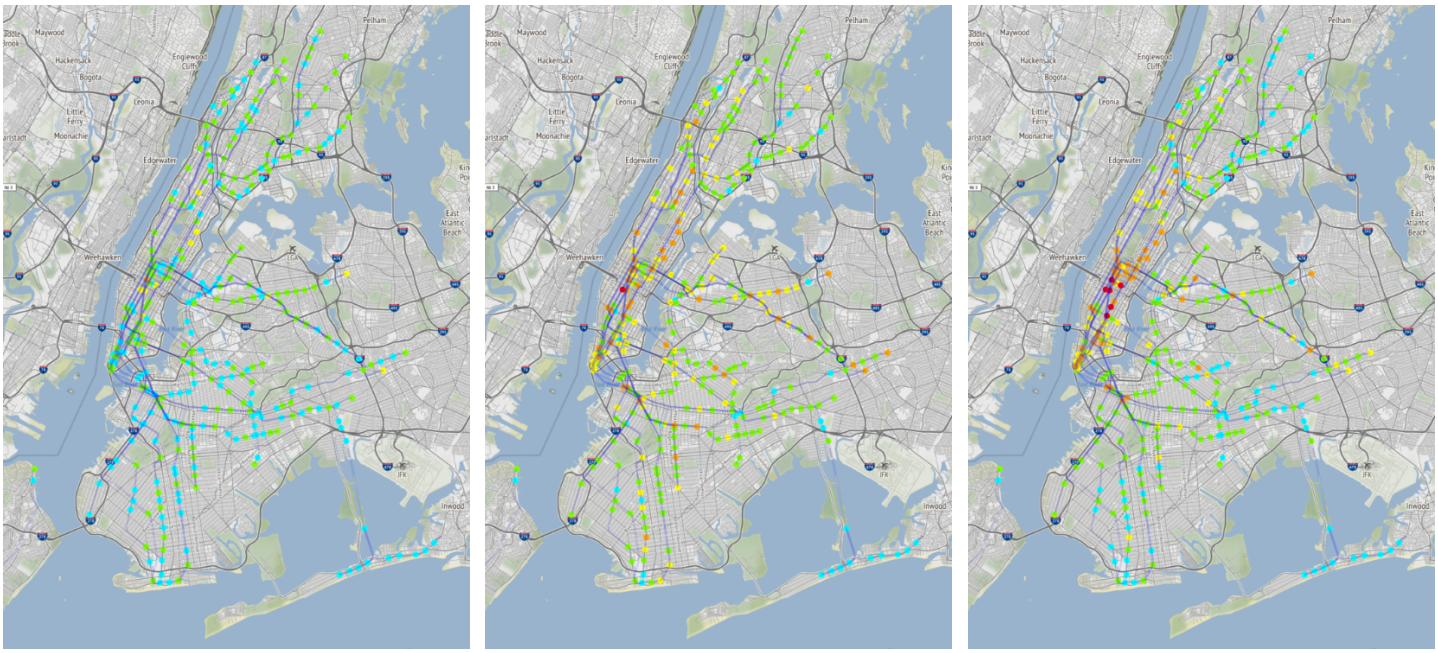
Entries during the summer season, for the early-day, day, and night times of day from left to right:

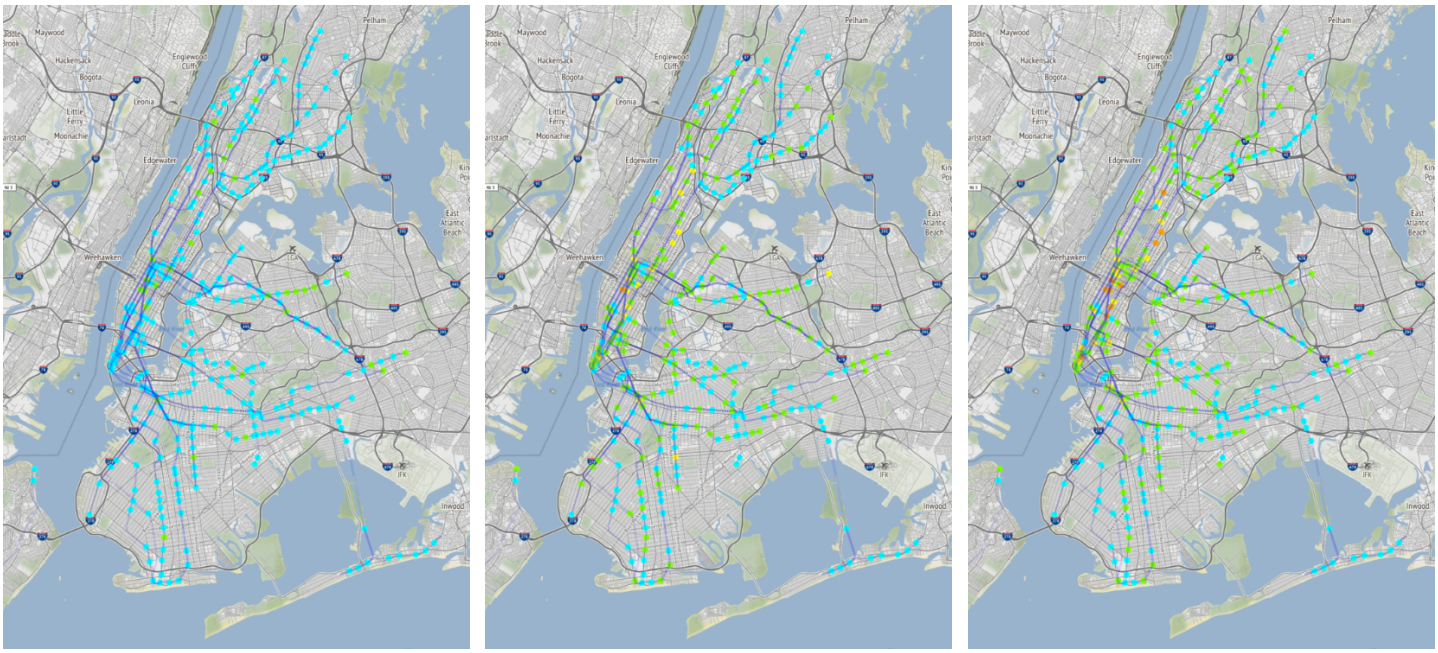
Entries during the fall season, for the early-day, day, and night times of day from left to right:



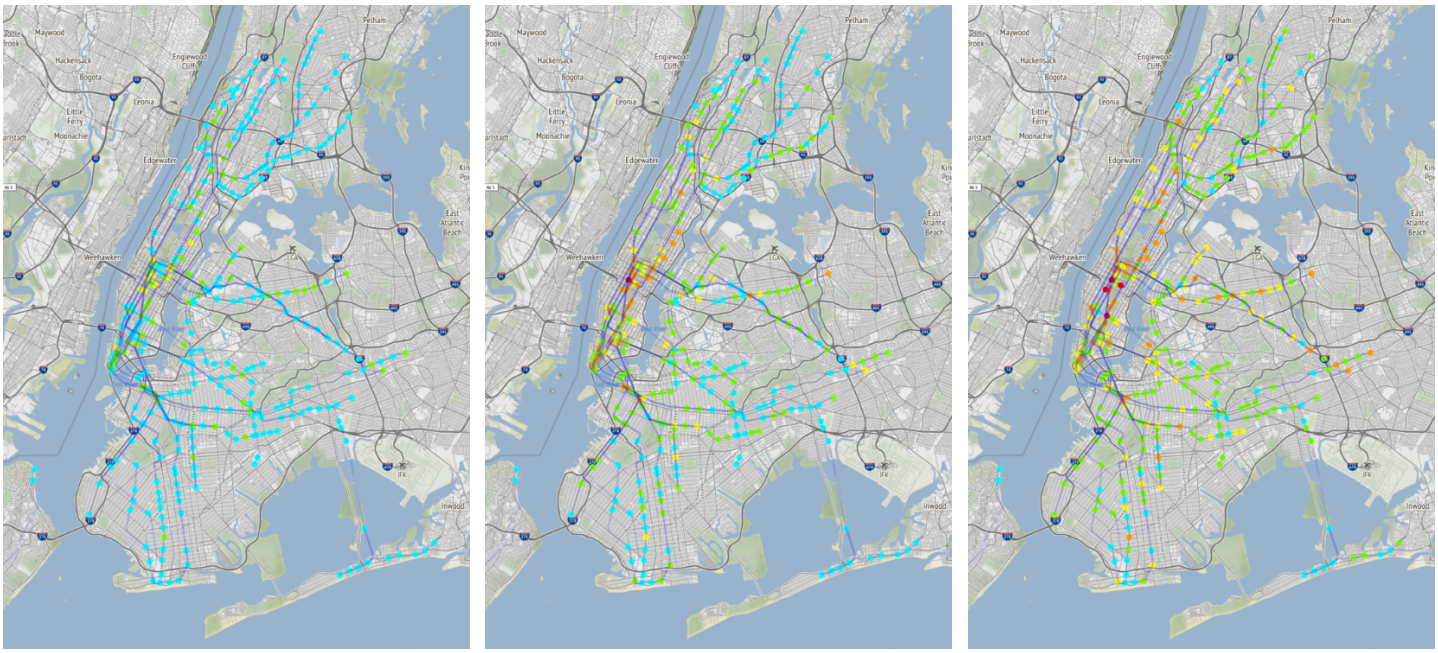
Entries during the winter season, for the early-day, day, and night times of day from left to right:



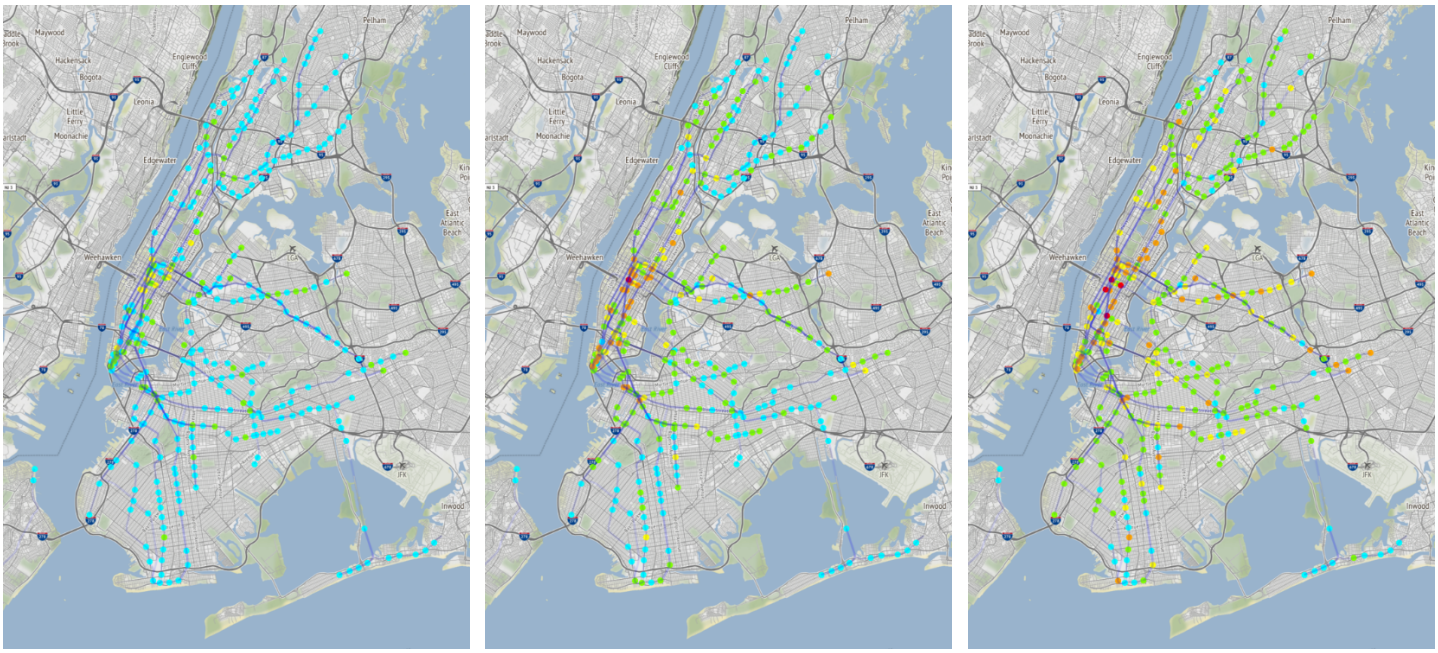
Entries during the coronavirus period, for the early-day, day, and night times of day from left to right:



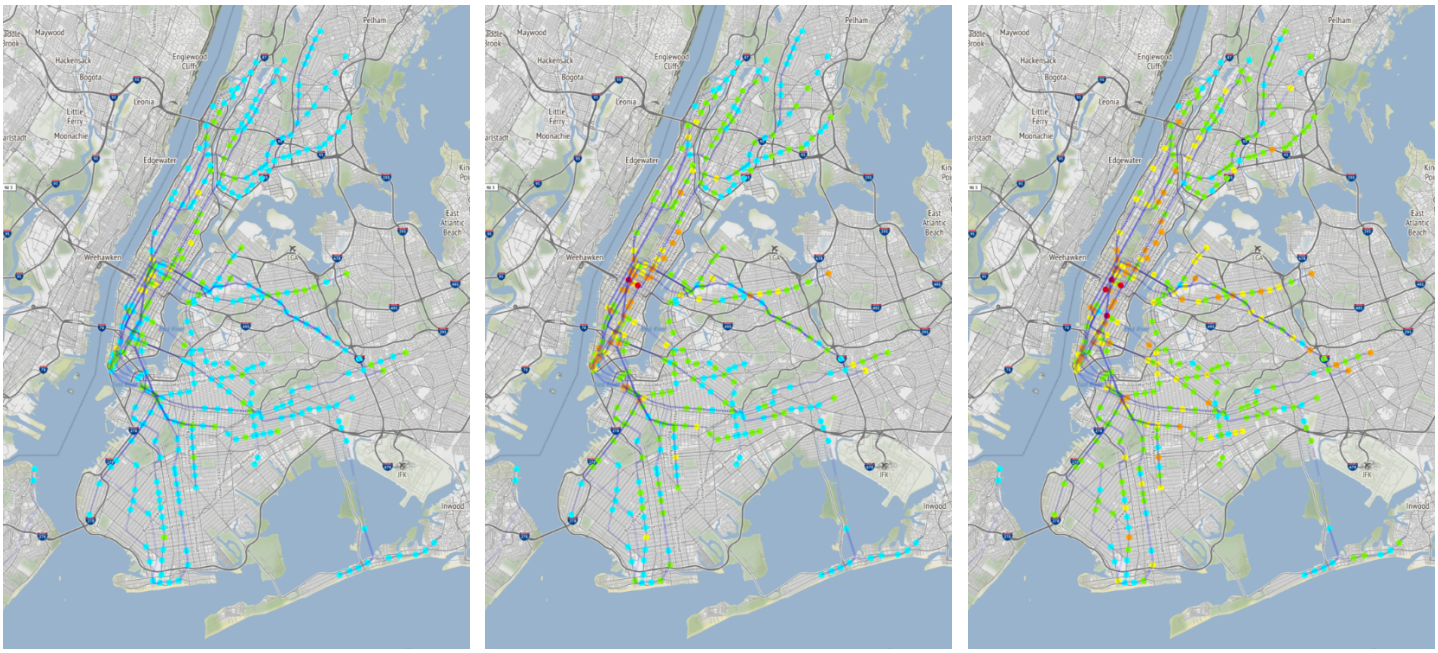
Exits during the spring season, for the early-day, day, and night times of day from left to right:



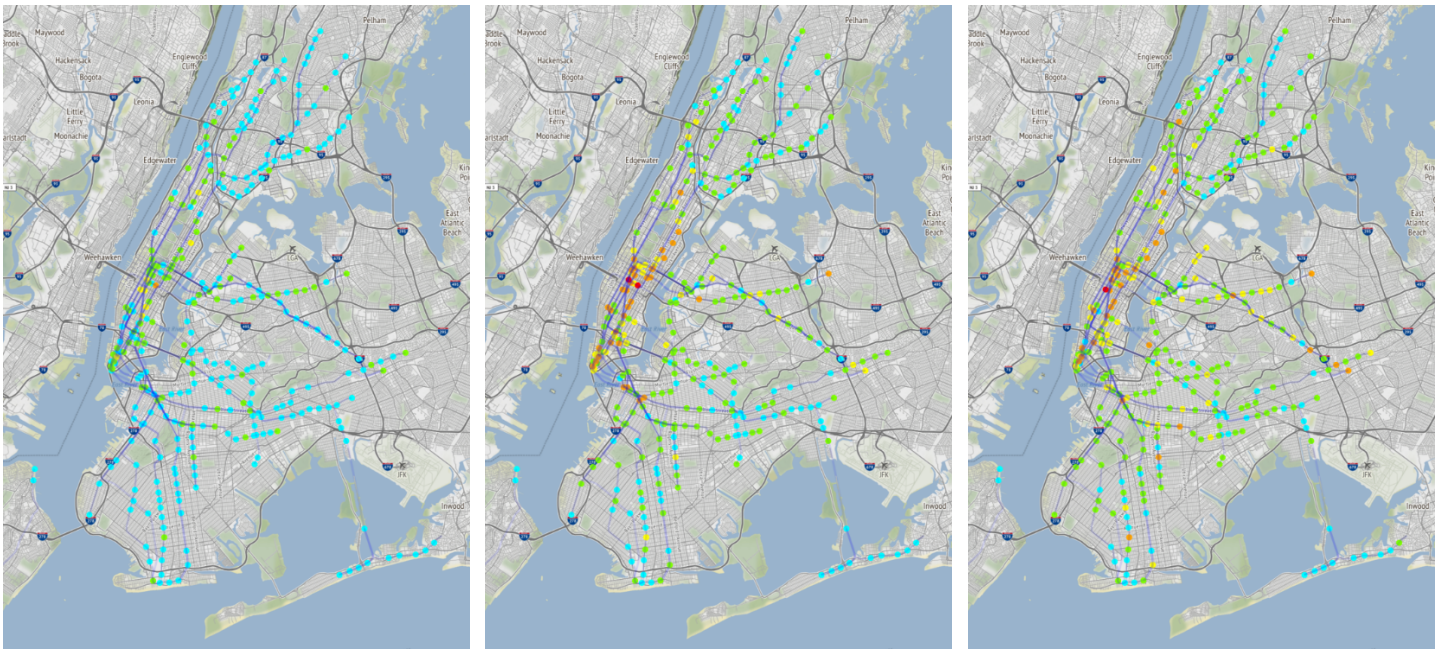
Exits during the summer season, for the early-day, day, and night times of day from left to right:



Exits during the fall season, for the early-day, day, and night times of day from left to right:



Exits during the winter season, for the early-day, day, and night times of day from left to right:



Exits during the coronavirus period, for the early-day, day, and night times of day from left to right: