CS-GY 6513 Big Data Project Report

RouteSavvy Team Members

Krapa Karthik [NET ID: kk5754, NYU ID: N12039854]

Sourik Dutta [NET ID: sd5913, NYU ID: N19304628]

Shreyansh Bhardwaj [NET ID: sb10261, NYU ID: N17664537]

Dev Thakkar [NET ID: djt8795, NYU ID: N19070379]

Project Slides - RouteSavvy Slide

RouteSavvy: A Real-Time Subway Ridership Analysis System

Introduction

NYC subway commuters face significant challenges with delays and overcrowding, with minimal visibility into real-time station conditions. While the MTA publishes detailed ridership data, this valuable information isn't effectively utilized for live route optimization. Existing apps don't account for station-level congestion, creating an opportunity to build smarter, data-driven transit recommendations

Project Vision and Objectives

RouteSavvy aims to revolutionize urban commuting through data-driven insights, making transit more intelligent, adaptive, and comfortable

. Our specific objectives are:

1. To transform urban commuting through intelligent, adaptive insights into transit congestion.

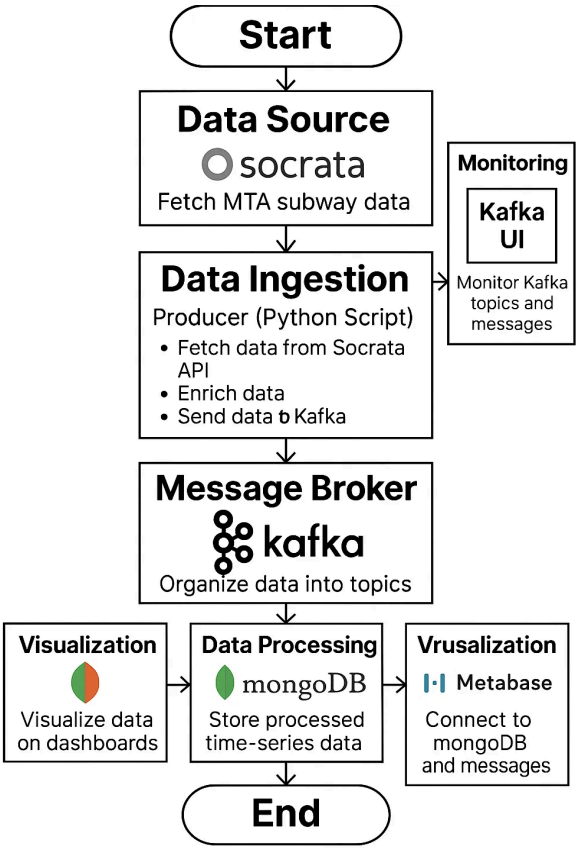
2. To enhance commuter satisfaction by minimizing time in crowded stations using predictive analytics.
3. To develop a scalable big data pipeline that processes hourly MTA ridership data and delivers optimized routes via APIs.

Big Data Justification

Key Characteristics

1. Volume: Billions of records spanning multiple years.
2. Velocity: Real-time data updates for ridership and service alerts.
3. Variety: Structured (e.g., ridership counts) and unstructured (e.g., customer feedback).
4. Veracity: Ensuring data accuracy and consistency across sources.
5. Value: Actionable insights to improve service reliability and rider experience.

System Architecture



Dataset Overview

The dataset is published by the Metropolitan Transportation Authority (MTA) and hosted on data.ny.gov, providing comprehensive subway ridership estimates. We accessed this data programmatically through the Socrata Open Data API using the Sodapy Python client, which allowed us to efficiently query and stream the data into our pipeline.

The temporal coverage spans from July 2020 through July 2024, encompassing:

- Pre-pandemic ridership patterns (early 2020)
- Dramatic ridership changes during COVID-19 lockdowns
- Recovery and new travel patterns in post-pandemic years

This longitudinal perspective provided us with rich insights into how ridership evolves under different societal conditions, which strengthens our prediction models.

Data Granularity and Enrichment

The dataset offers hourly records for each subway station complex across the MTA system:

- Each data point represents one hour of ridership at a specific station complex
- Station complexes often serve multiple train lines (e.g., "Times Sq-42 St (N,Q,R,W,S,1,2,3,7)/42 St (A,C,E)")
- This high temporal resolution allows us to capture precise commuter flow patterns throughout the day

With approximately 60,000 new entries every day across all stations, the complete dataset contains over 100 million records. This volume necessitated our streaming-based architecture using Kafka and PySpark to process data efficiently.

Data Attributes and Enrichment

Our pipeline works with these key data attributes from the MTA dataset:

- `transit_timestamp`: The exact hour when ridership was recorded, which we parse to derive time-based features
- `station_complex`: Station identifiers that we map to subway lines and transfer points
- `borough`: Location data (Manhattan, Brooklyn, Queens, Bronx, or Staten Island)
- `ridership`: Total entries recorded at the station during that hour
- `transfers`: Movement between lines, helping us understand connection patterns

- `latitude/longitude`: Geographic coordinates for spatial analysis and visualization
- `payment_method`: Method used (MetroCard, OMNY, etc.)
- `fare_class_category`: Type of fare (Full Fare, Student, Senior, etc.)

Our data producer script enhances this raw data with calculated fields such as:

- Year, month, day, and hour components extracted from timestamps
- Peak-hour flags (1 for rush hours, 0 otherwise)
- Transfer ratios to identify high-connectivity stations
- Cumulative ridership metrics for trend analysis

As demonstrated in our Python producer script, we've implemented robust data enrichment to transform the raw MTA data into actionable insights for commuter route optimization.

Data Quality and Preprocessing

To ensure reliable analysis, our pipeline addresses several data quality challenges:

- Handling timestamp inconsistencies across records
- Normalizing station names for consistent identification
- Converting string representations to appropriate numeric types
- Implementing safeguards against missing or zero values

This preprocessing creates a clean, structured dataset that feeds directly into our PySpark streaming jobs for real-time crowd score computation.

Data Processing

PySpark Structured Streaming reads from Kafka and transforms data into crowd-level insights. Aggregations are performed over sliding windows to simulate live congestion detection for timely route recommendations.

Data Storage and Visualization

Processed data is stored in MongoDB for flexible document-based storage. We use Metabase for visualization, which provides intuitive dashboards and free MongoDB integration to analyze key metrics like station congestion and ridership patterns.

Deployment

Our entire streaming stack is containerized using Docker, ensuring consistency across environments and simplifying deployment. We created a custom Spark image with all necessary dependencies to handle our specific processing needs.

Implementation Details

Data Producer Implementation

Our Python producer script connects to the MTA API and streams data to Kafka:

```
#!/usr/bin/env python3
import json
from datetime import datetime
from sodapy import Socrata
from confluent_kafka import Producer

# API & Kafka setup
client = Socrata("data.ny.gov", "XXXXXXXXXX")
producer_config = {
      'bootstrap.servers': 'localhost:9092',
      'client.id': 'csv-json-producer',
      'compression.type': 'gzip',
      'batch.num.messages': 100_000,
      # Additional configuration...
}
producer = Producer(producer_config)

# Helper function to enrich data
def enrich(row: dict) -> dict:
      # Parse timestamp into components
      ts = datetime.fromisoformat(row["transit_timestamp"].replace("Z",
"+00:00"))
      row["year"] = ts.year
      row["month"] = ts.month
      row["day"] = ts.day
      row["hour"] = ts.hour

      # Add peak-hour flag
      row["peak_hour"] = 1 if (7 <= ts.hour <= 9) or (16 <= ts.hour <= 19)
else 0

      # Calculate metrics
      ridership = float(row.get("ridership", 0) or 0)
      transfers = float(row.get("transfers", 0) or 0)
      row["transfer_ratio"] = 0.0 if ridership <= 0 or transfers <= 0 else
transfers / ridership
```

```
        # Track cumulative metrics
        global cumulative_ridership
        cumulative_ridership += ridership
        row["cumulative_ridership"] = cumulative_ridership

        return row

# Main processing loop with pagination
try:
        while True:
        page = client.get(
                dataset_id,
                where=f"transit_timestamp >= '{start_date}' AND
transit_timestamp <= '{end_date}'",
                limit=batch_size,
                offset=offset
        )

        if not page:  # no more rows
                break

        for raw_row in page:
                enriched = enrich(raw_row)
                producer.produce(
                        topic=topic,
                        value=json.dumps(enriched).encode("utf-8")
                )

        producer.flush()  # force delivery for each batch
        offset += batch_size
finally:
        producer.flush()
```

Scalable Spark Architecture

We built a scalable processing architecture using Docker Compose with the following key services:

Kafka Ecosystem:

- Zookeeper for coordination

- Kafka broker for message streaming
- Kafka UI for monitoring topic health and consumer lag

Spark Cluster:

- Master node for job coordination
- Multiple worker nodes for distributed processing
- History server for job tracking and performance monitoring

Storage and Visualization:

- MongoDB for flexible document storage
- Metabase for analytics dashboards and visualization

Our custom Spark Dockerfile ensures all dependencies are properly installed

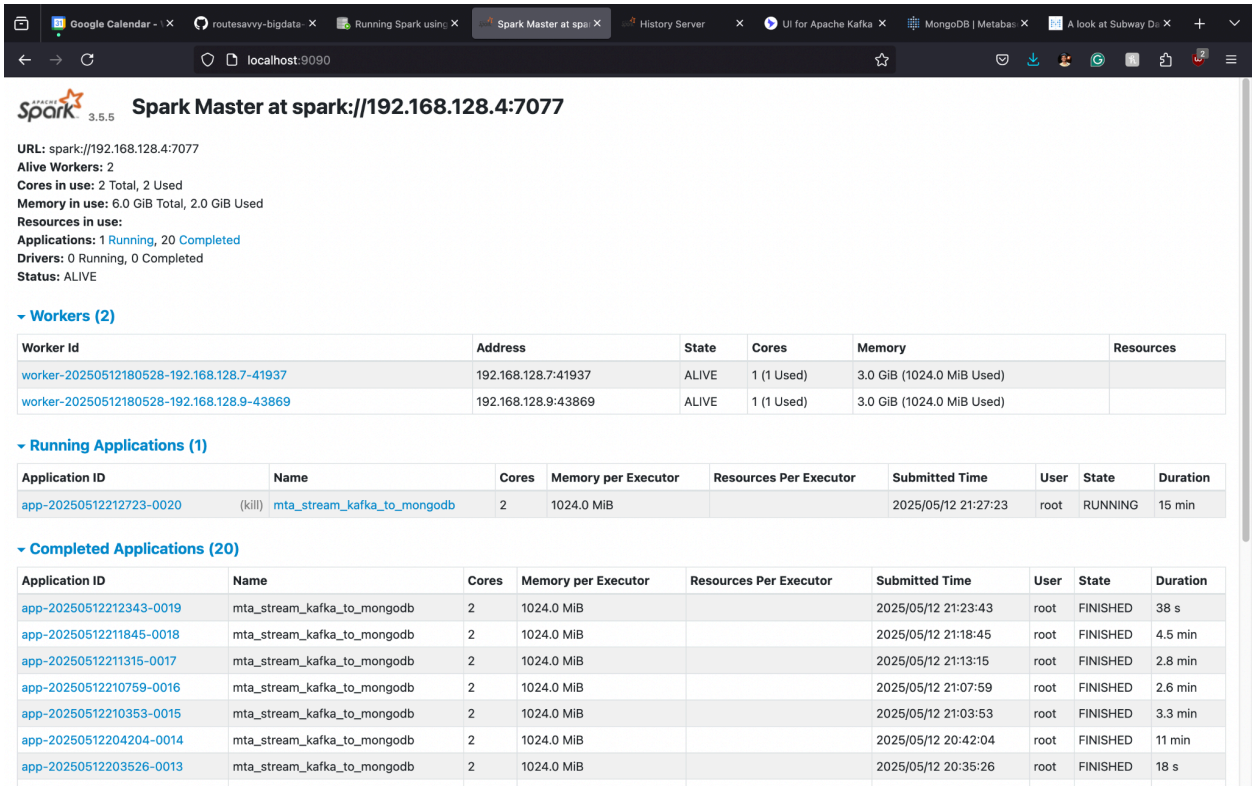
```
FROM bitnami/spark:latest

COPY requirements.txt .
COPY ./conf/ /opt/bitnami/spark/conf/

USER root

RUN apt-get clean && \
      apt-get update && \
      apt-get install -y python3-pip && \
      pip3 install -r ./requirements.txt

ENV SPARK_CONF_DIR=/opt/bitnami/spark/conf
ENV SPARK_RPC_AUTHENTICATION_ENABLED=no
ENV SPARK_RPC_ENCRYPTION_ENABLED=no
ENV SPARK_LOCAL_STORAGE_ENCRYPTION_ENABLED=no
ENV SPARK_SSL_ENABLED=no
ENV SPARK_USER=spark
```

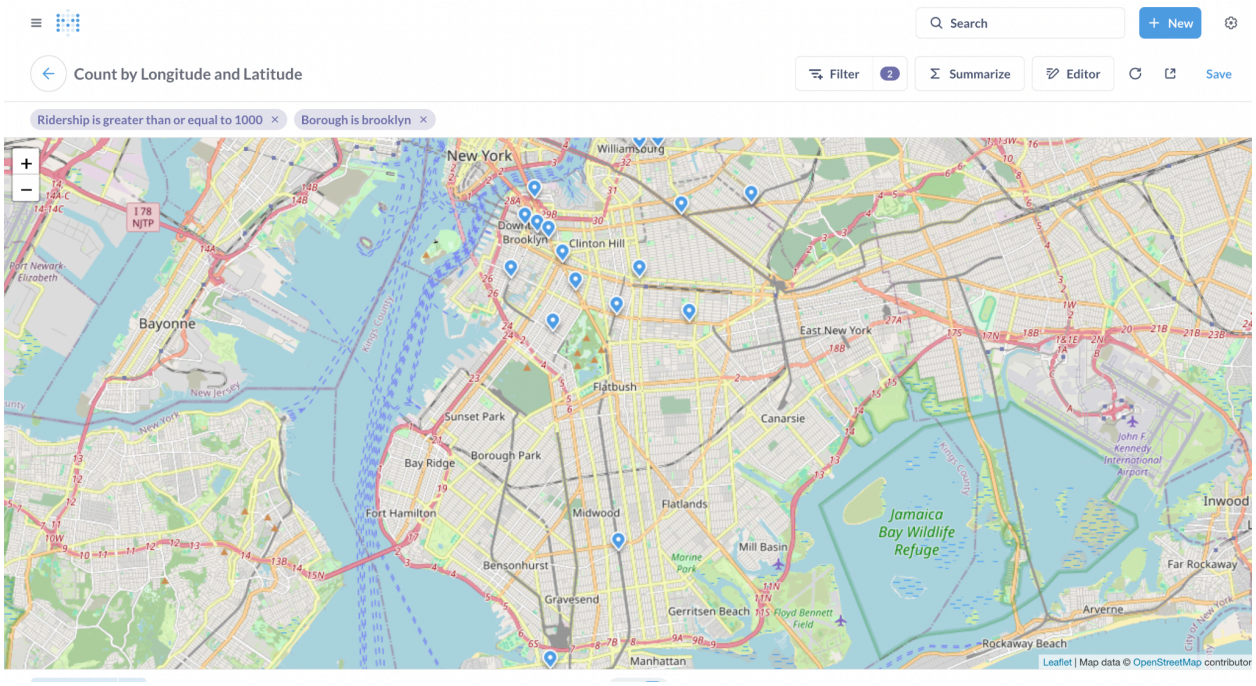


Results and Analysis

Our analysis revealed several key insights about NYC subway ridership patterns:

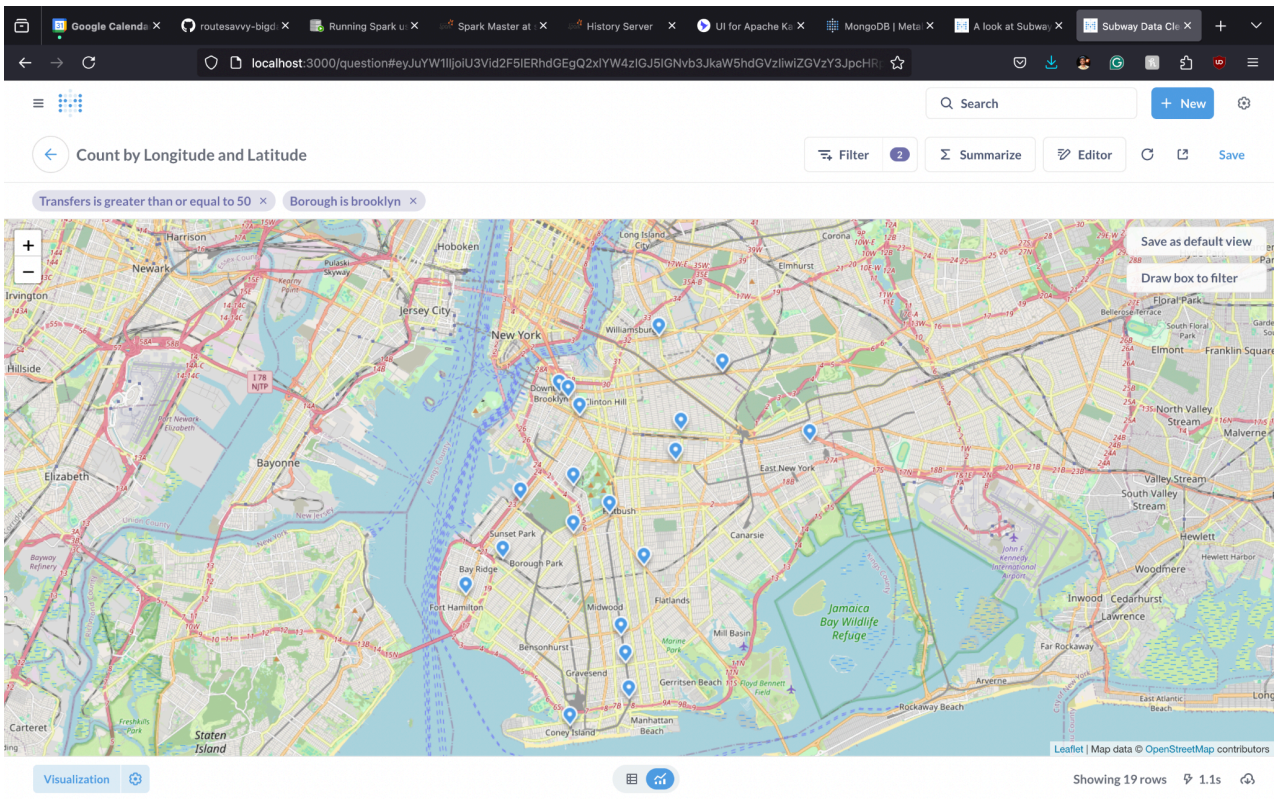
Station Congestion Patterns

We identified high-transfer subway stations in Brooklyn, with Atlantic Av-Barclays Center standing out as the busiest, serving as a major hub for multiple lines and heavy commuter flow.



Exploring Transfers:

Brooklyn subway stations with ridership exceeding 1,000 entries per hour, demonstrating RouteSavvy's ability to identify high-traffic locations.



The ridership data processed through our PySpark streaming jobs reveals distinct commuting patterns that are critical for RouteSavvy's congestion avoidance strategies:

- **Morning and Evening Rush Hours**: Our enriched data clearly shows two distinct spikes during typical commuting times:
    - Morning Peak (7 AM - 9 AM): Reflects the inbound commute to workplaces
    - Evening Peak (5 PM - 7 PM): Represents the outbound commute back home
- **Off-Peak Hours**: There is a noticeable dip in ridership during midday and late-night hours, indicating reduced subway utilization during these times.

Our system captures these patterns through the "peak_hour" flag that we calculate in our data enrichment process:

```
row["peak_hour"] = 1 if (7 <= ts.hour <= 9) or (16 <= ts.hour <= 19) else 0
```

This flag enables RouteSavvy to provide more intelligent route recommendations, potentially suggesting alternate paths during identified peak hours when certain stations may be overcrowded.

Exploring Ridership: Delays and Alerts

RouteSavvy can adopt a approach focused on how delays affect crowd levels:

- **Impact on Commuters**: Service alerts create noticeable impacts on ridership patterns, as visualized in our Metabase dashboards comparing ridership levels "With Alerts" and "Without Alerts."

- **Ridership Impact**: The presence of alerts correlates with reduced ridership, indicating that service reliability directly influences commuter behavior.

Exploring Where Riders Are Concentrated

Our analysis of the MTA dataset reveals the most heavily trafficked subway stations in New York City:

- **Manhattan Hubs Dominate**: Stations in Manhattan like Times Square-42nd St and Grand Central-42nd St are major centers of commuter activity.
- **Peak Hour Patterns**: These key stations see ridership peaks between 8 AM and 6-7 PM, coinciding with typical workday commuting hours.

Our system identifies high-traffic stations through aggregation operations in PySpark:

```
station_activity = df.groupBy("station_complex") \
    .agg(sum("ridership").alias("total_entries")) \
```

```
    .orderBy(col("total_entries").desc())
```

Challenges We Faced

Implementing RouteSavvy presented several challenges:

1. **Tool Learning Curve**: Configuring Kafka and Spark on Docker was not plug-and-play and required significant learning.
2. **Connectivity Issues**: Ensuring reliable communication between containerized services.
3. **Data Quality**: We encountered timestamp inconsistencies and missing values that required robust cleaning strategies.
4. **Access & Scalability**: Managing access to high-volume data streams while maintaining performance.
5. **Resource Constraints**: Balancing processing needs with available system resources.

What We Learned

This project provided valuable hands-on experience with big data technologies:

Data Integration & Streaming

- Connected to MTA subway data using Sodapy + Socrata.
- Learned how to enrich, validate, and route data within pipelines.
- Streamed data through Apache Kafka for real-time processing.

Architecture & Deployment

- Designed a robust system with Kafka, Spark, MongoDB, and Metabase.
- Used Docker and Docker Compose for containerized deployment.
- Improved understanding of service orchestration and fault tolerance.

Future Steps

Looking ahead, we plan to enhance RouteSavvy in several ways:

1. **Contextual Data Integration**: Incorporate more contextual data such as bike-share availability and weather conditions for more nuanced route suggestions.
2. **Improved Modularity**: Make our codebase more flexible to easily plug in new data sources or swap out components like Spark or Kafka if needed.

3. **Advanced Visualization**: Explore Grafana and InfluxDB as alternatives for more sophisticated real-time dashboards.
4. **Cross-City Adaptability**: Adapt the system architecture to work with transit data from other cities beyond NYC.

Conclusion

RouteSavvy demonstrates the power of applying big data technologies to urban transportation challenges. By leveraging real-time MTA ridership data, we've created a scalable system that can help commuters make informed decisions about their routes, ultimately making urban travel smarter and more comfortable.

The successful integration of multiple technologies-Sodapy for API access, Kafka for streaming, PySpark for processing, MongoDB for storage, and Metabase for visualization-showcases how modern data pipelines can address real-world problems. As cities continue to grow, data-driven solutions like RouteSavvy will become increasingly important for optimizing transit systems and improving the commuter experience.