Real-Time Point-of-Sale Analytics using Apache Kafka and PySpark

Gabriel McReynolds & Nathan Jarvis

Kansas State University

CIS 531/731: Programming Techniques for Data Analytics and Science

12/12/2024

Real-Time Point-of-Sale Analytics using Apache Kafka and PySpark

Point-of-sale (POS) analytics involves the collection, interpretation, and utilization of data generated from transactions processed at retail locations. This process serves as an important component of modern retail operations, providing valuable insights into customer behavior, sales trends, and inventory movement.

In today's retail landscape, characterized by market volatility, increasing competition, and narrowing profit margins, POS analytics has become indispensable. Retailers need reliable data to optimize their inventory management systems, streamline operations, and make informed decisions to stay competitive. By leveraging POS analytics, businesses can not only reduce costs associated with overstocking or understocking but also respond more effectively to shifts in consumer demand.

POS analytics empowers retailers to transform raw transactional data into meaningful insights, creating smarter decisions to properly forecast demand while making sure items are moving off the shelves and not staying stagnant.

Background and Related Work

Origin and Modifications

This project is inspired by the Databricks Solution Accelerator for Real-Time Point-of-Sale Analytics (*Real-time point-of-sale analytics*), which provides a framework for creating Delta Live Tables that automate the ingestion and transformation the of transactional data in a real-time streaming environment. However, rather than using Delta Live Tables, which is central to the original solution, this project captures transaction data through Kafka, enabling real-time streaming and ensuring low-latency processing.

A key departure from the original framework lies in the replacement of managed services with a custom Python server setup. This modification was undertaken to prioritize cost efficiency and provide greater control over event handling. By using Python to build the server infrastructure, the project achieves a balance between scalability and flexibility, ensuring the solution can be tailored to specific business needs without reliance on proprietary tools.

Data Introduction

The dataset used in this project is from the original solution Databricks Solution Accelerator. It provides 7 files that offer a comprehensive view of inventory and sales activities, combining both historical snapshots and real-time changes. The two primary types of files—Inventory Snapshot Files and Inventory Change Files—along with supporting metadata to enrich the data and make it actionable. The Inventory Snapshot Files provide a baseline view of stock levels for items at specific timestamps, capturing the state of inventory at both online and in-store locations. The Inventory Change Files, on the other hand, record every update to inventory levels. These changes include activities such as sales, restocks, shrinkage, and buy-online-pickup-in-store (BOPIS) transactions that are marked with a corresponding value for the change type. Each entry is timestamped and contains details like the quantity changed and a unique transaction ID, making it possible to trace the history of individual items across different stores or online.

Supporting these core datasets are three Metadata Files, which add context and depth to the analysis. The Change Type file maps the type of change with its corresponding value along with other items. The Store metadata includes store names and IDs, enabling differentiation between online and in-store transactions. The Item file includes the item name, supplier, and safety stock levels. Together, these datasets form a cohesive and detailed record of inventory activity, enabling precise and dynamic analysis. By integrating these data sources, the project creates a solid foundation for real-time insights and demand forecasting.

Methodology

We departed from the Databricks accelerator notebook after extensively debugging issues with Delta Live Tables on the Community version of Databricks as well as several issues with connecting

Azure Devices to our notebooks. The solution was to create a custom system using two python servers – producer.py and consumer.py. The producer would read the csv files and send them to the consumer using Kafka. The consumer would read the Kafka batches using Apache Spark and process them accordingly.

Latency Analysis

To evaluate the latency from sending the message to receiving and processing the message, the consumer server would save the data sent to MongoDB and PostgreSQL to observe differences in latency. Another change to observe was whether each batch was aggregated or not. The consumer would receive thousands of transactions each batch, and we wanted to evaluate to see if aggregating the results using an Apache Spark RDD would speed up the process by grouping by “Item_ID” and “Store_ID” to sum up the “quantity” of the items that were sold. Therefore, there were four main variations: aggregating batches and storing single transactions with MongoDB and PostgreSQL.

To increase the number of transactions for each batch more producer servers were spawned. The consumer server would finish processing one batch and then start on the next batch. This means that by running multiple producer servers simultaneously the number of transactions in a batch would increase simulating a higher request count.

Data Analysis

In order to do analysis on the data, first Kafka topics had to be made for the four main data files in a separate producer. The two inventory snapshots and the two inventory change files were streamed into Kafka to later be used in a consumer. For the metadata, a python file was made and functioned as a static reference for mapping to the main files creating “Item_Lookup”, “Store_Lookup”, and “Change_Type_Lookup”. These dictionaries were then combined with the Kafka topics to create our consumer. The consumer joined the topic together by the date and the “Item_Lookup” to match our snapshot and the change files. The “Change_Type_Lookup” matched the “Change_ID” from the change files to give it the proper value for the corresponding change type. The “Store_Lookup” matched where the sale was at either online or in store.

Other fields were created as well like “Action” and “Final_Quantity”. Action was just the “Change_Type” but added in the value of “inventory report” for the snapshot files for analysis, so when trying to look at quantity before and after changes by “Change_Type” there wouldn’t be nulls. “Final_Quantity” calculated the quantity from the snapshots after the change. If there was an original quantity of 100 for an item, and the change type was a restock of 50 for that item on the same day, the final quantity would be 150. Now that the data was integrated into a single data set, it was ready for analysis in Tableau.

Evaluation

Latency

Accurate latency calculations are important for evaluating different approaches. Calculating latency was the same for each variation. The producer server added a column to each transaction called “sent_time”. Then the consumer server would calculate the average sent time in each batch and subtract it from the ending time of processing.

To evaluate how resilient the consumer was with higher throughput in each batch, more producers were run. For each variation of consumer there were 6 test runs, each increasing the number of producers that were run.

Once the transactions are added to the database, a query was then run to determine how long it would take to aggregate all the rows grouped by “Item_ID” and “Store_ID”. For SQL this used a simple group by table function, and for MongoDB the “.aggregate” function was used on the collection with the “$sum” operator applied to the “quantity” key.

Demand Forecasting

Now that the data was all joined together into one file, analysis could be done. What exactly were we looking for though? From the data, we wanted to be able to look at a specific item, or group of items and see what the trends for the next days, weeks, or months were. Since there are also so many fields in the data, you can filter by whatever you would like to see more fine-grained detail on the forecasting. Fields such as “Store_Name”, “Employee_ID”, and “Date_Time” that go down to the minute can provide a more in depth understanding of what is happening to items and inventory.

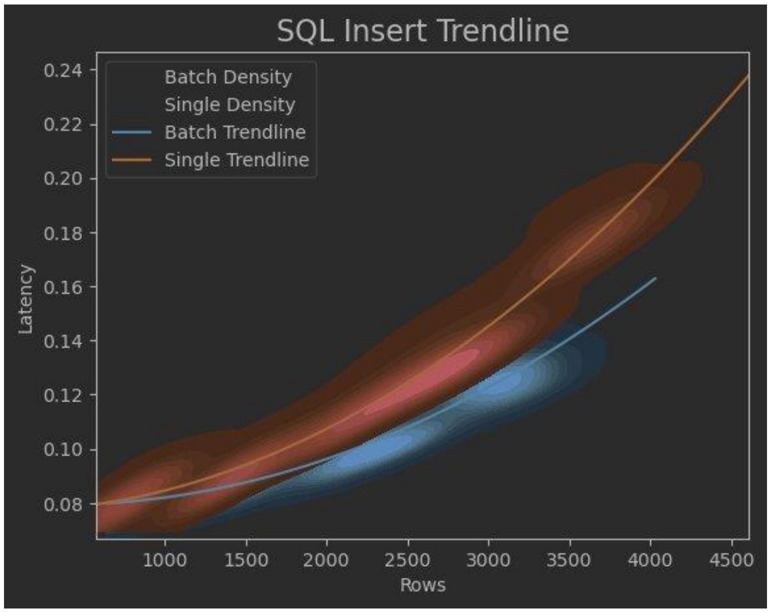
Results

Latency

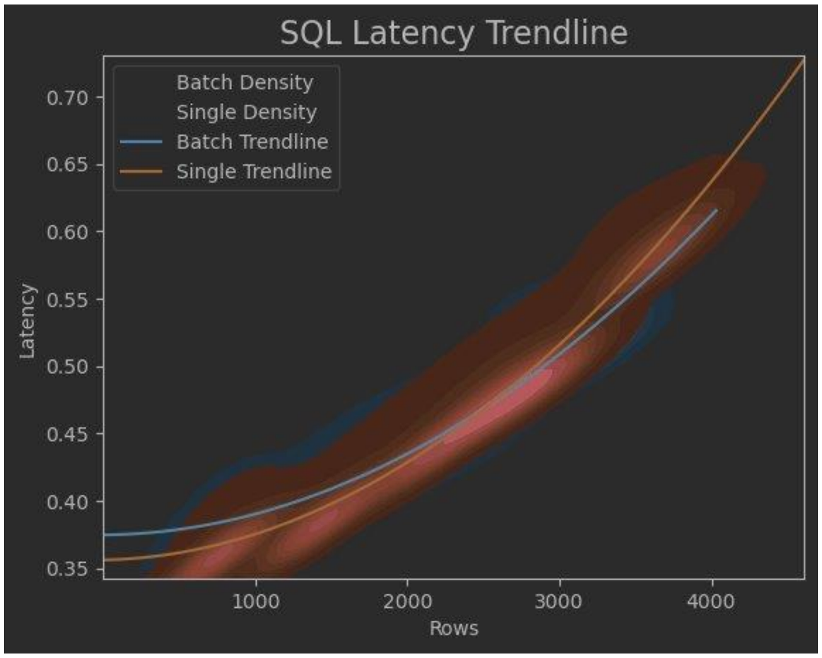
Both MongoDB and PostgreSQL had similar graphs for batch vs single insert and latency. As seen in Figure 1, the insert time of the batch is significantly lower than the insert time of inserting every single transaction. This makes logical sense as there will be fewer rows to be inserted as Apache Spark has aggregated many rows down. The latency trendline, however, converges around 2300 rows for overall request time. This is due to the overhead of the aggregation with the Spark RDD becoming more efficient with more rows as more time is saved inserting into the database. Overall, batch processing is generally more performant than just inserting every row into the database.

SQL insert times are significantly slower than PostgreSQL insert times. When comparing varieties, there is a strong relationship between the number of rows and the latency time as seen in Figure 3. For processing batches in MongoDB, there is a correlation of 0.8515, and for SQL there is a correlation of .5507. To determine if MongoDB really is faster than PostgreSQL a t-test was done with null hypothesis being they are the same. The result is a T-statistic of 91.67 and p-value of effectively 0.0 < 0.05 therefore MongoDB is faster than PostgreSQL.

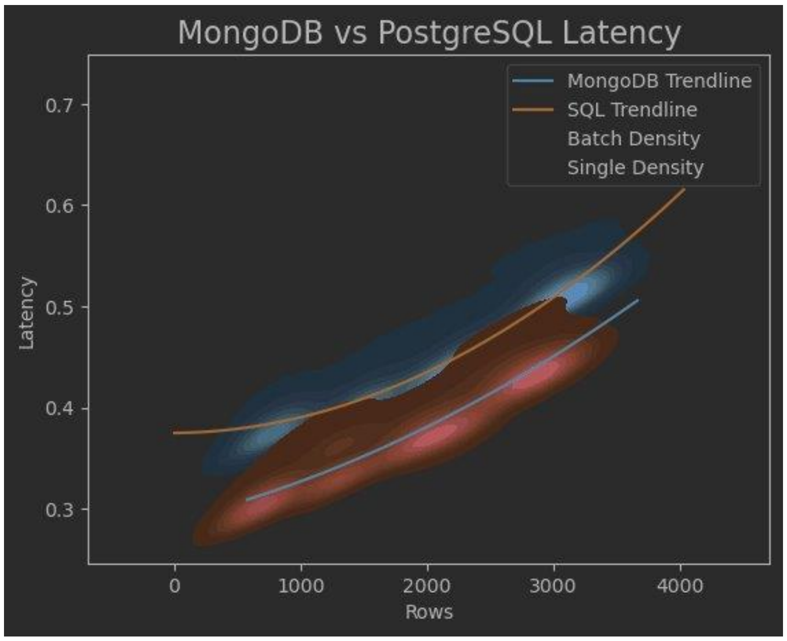
Comparing query times is a different story, with MongoDB taking 2.4 seconds to aggregate transactions for the single collection and 1.6 seconds for the batch collection. PostgreSQL takes 1.5 seconds for the single table and .8 seconds for the batch table.



(Figure 1: SQL insert time trendline with density)



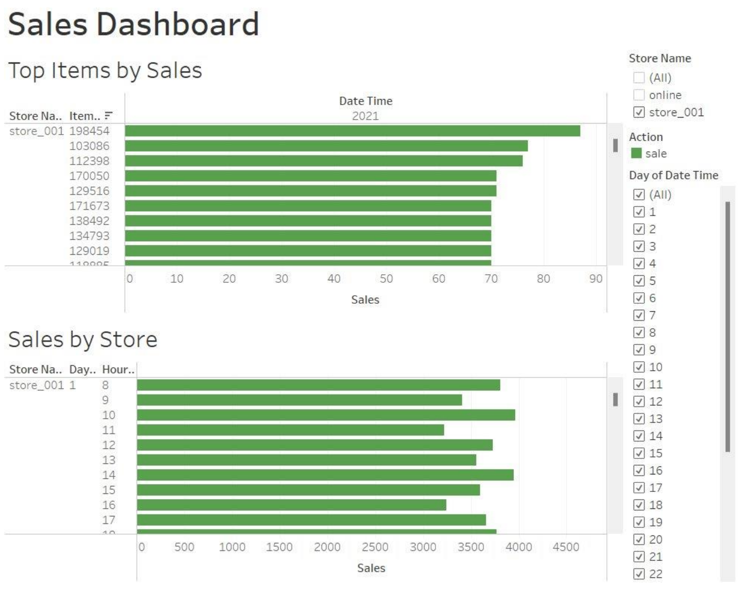
(Figure 2: SQL Total Batch Latency Trendline with Density)

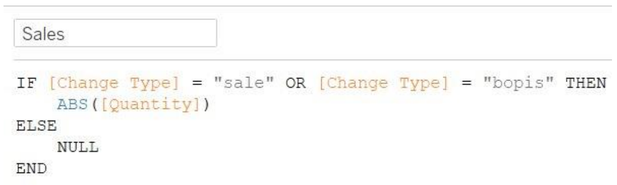


(Figure 3: MongoDB vs PostgreSQL Latency)

Tableau Analysis

After loading the data into Tableau, it is now time for analysis of our combined CSV. We created a dashboard of sales graphics that included top items by sales and sales by store. By filtering Action to include only "Sale" transactions and filtering by “Store_Name” to only include Store_001, the analysis highlighted items that performed well in the physical store. These visualizations can reveal which store type contributed most to overall revenue, aiding in decision-making about inventory allocation.





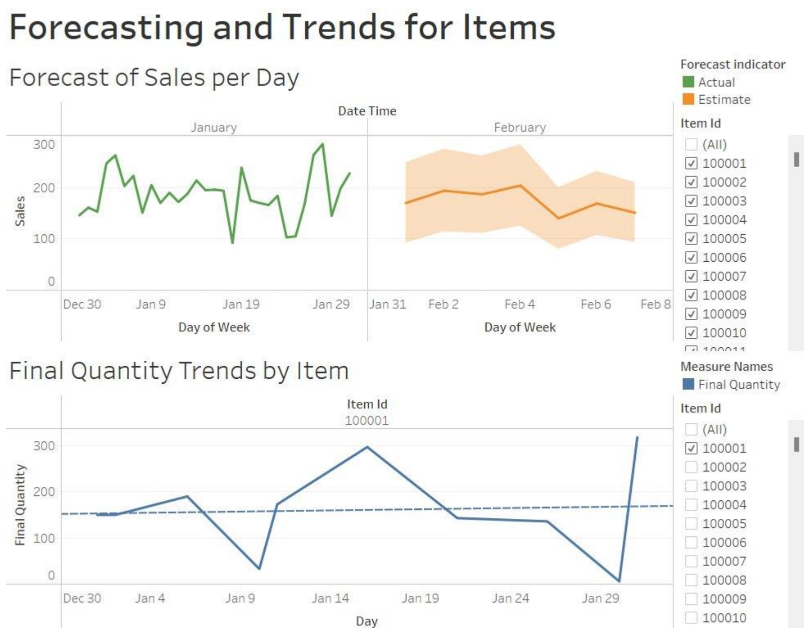
Sales

```
IF [Change Type] = "sale" OR [Change Type] = "bopis" THEN
    ABS([Quantity])
ELSE
    NULL
END
```

(Figure 4: Calculated Field to Make Sales)

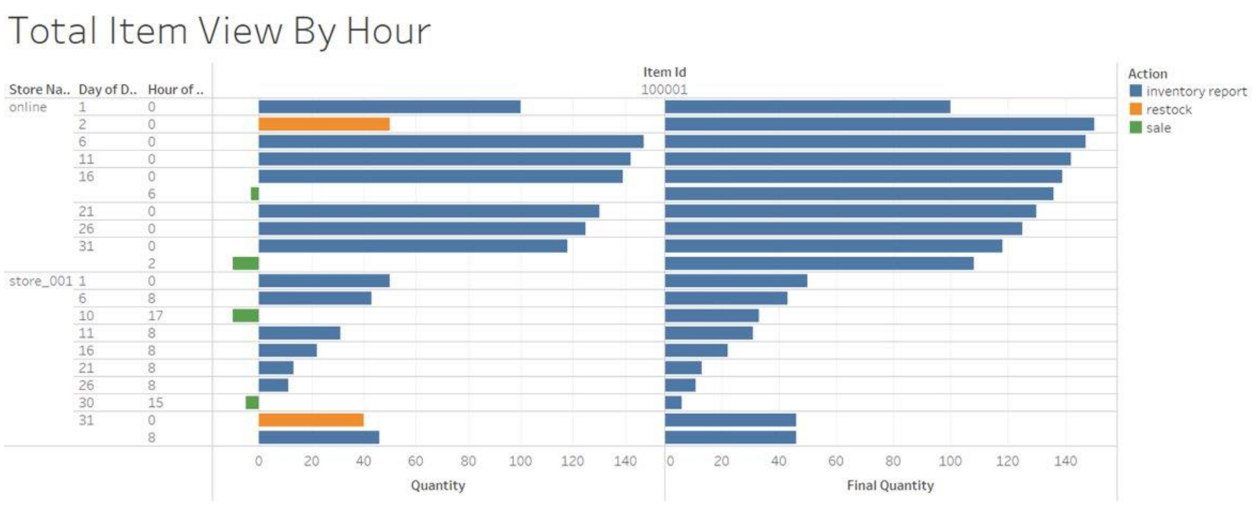
(Figure 5: Sales Dashboard)

The second piece of analysis was done to forecast and look at the trends of sales and items. The first visualization on this dashboard created a forecast for the next 2 weeks of sales for a group of items that could all be related. This can give accurate forecast for sales so you can look at where sales have been moving for certain items and prepare for this by having more or less stock of certain items. The second visual shows the final quantity for an item and the trend that it is following. You can filter both visualizations by one item or many to get a better understand of what is happening to each



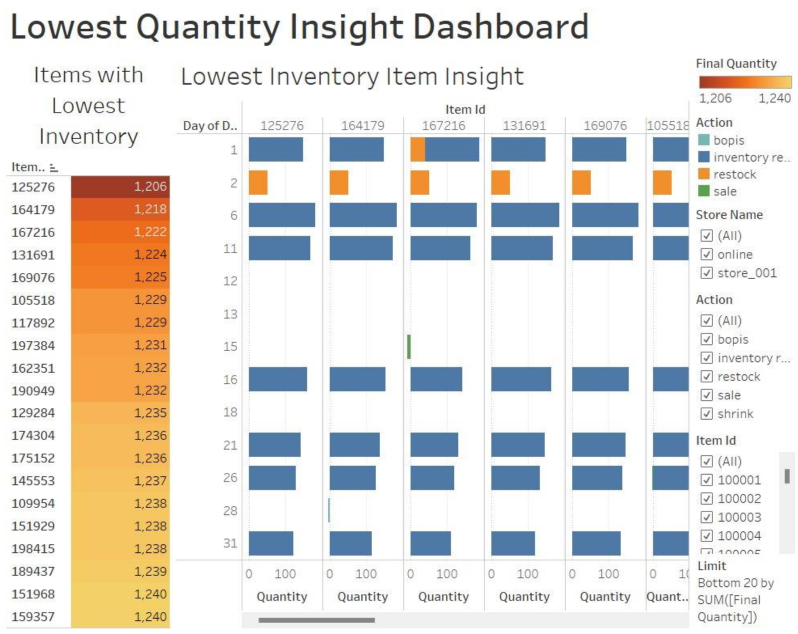
(Figure 6: Forecasting and Trends for Items)

For a deeper understanding of individual item performance, Tableau dashboards allow filtering down to specific item ID. This view provided detailed insights into daily and hourly activity for a single item, including quantity, action, and final quantity. Comparing these metrics helped identify trends like declining sales or consistent restocking needs.



(Figure 7: Total Item View by Hour)

The last visualization that can be used is another dashboard showing the current 20 items that have the lowest final quantity. For these at-risk items, there is another visual that shows what has been happening to these items over the past days. You can see the sales and restocks for the items to note if they aren't selling or have been restocked too much.


Lowest Quantity Insight Dashboard
Items with Lowest Inventory
Lowest Inventory Item Insight
Item Id
Day of D..
Quantity
Final Quantity
1,206
1,240
Action
bopis
inventory re..
restock
sale

Item..	
125276	1,206
164179	1,218
167216	1,222
131691	1,224
169076	1,225
105518	1,229
117892	1,229
197384	1,231
162351	1,232
190949	1,232
129284	1,235
174304	1,236
175152	1,236
145553	1,237
109954	1,238
151929	1,238
198415	1,238
189437	1,239
151968	1,240
159357	1,240

Store Name
☑ (All)
☑ online
☑ store_001

Action
☑ (All)
☑ bopis
☑ inventory r...
☑ restock
☑ sale
☑ shrink

Item Id
☑ (All)
☑ 100001
☑ 100002
☑ 100003
☑ 100004

Limit
Bottom 20 by SUM([Final Quantity])

(Figure 8: Lowest Quantity Insight Dashboard)

The Tableau analysis provided a comprehensive framework for exploring sales, inventory, and demand forecasting. By combining filters and predictive analytics, the visualizations support strategic decisions on inventory management and demand planning. The dashboards ensure adaptability to specific questions or needs, enhancing the utility of the data for real-time insights.

Summary

Real World Application and Improvements

While the current implementation demonstrates how transactional data can be processed in real time, adapting this project for real-world use would involve creating an automated and continuous workflow. A practical approach would be to implement a trigger-based system that runs on a scheduled interval, such as every hour or day. This trigger would initiate the producer script to ingest new transaction data from the POS system and update Kafka topics with minimal latency. Once the producer completes its task, another trigger could activate the consumer script to process the newly ingested data which will make it into the consolidated file. The enriched file would then feed into a visualization pipeline, such as Tableau, where real-time dashboards could provide analytics. This pipeline would allow for seamless updates and ensure decision-makers have access to the most current data.

While the current implementation demonstrates a robust framework for processing and analyzing point-of-sale data, there are several areas where improvements could enhance its scalability and effectiveness. One potential enhancement involves query optimization. By investigating advanced indexing strategies in SQL and MongoDB, query times could be significantly reduced, especially for large datasets, leading to more efficient data retrieval and processing. Additionally, as the dataset expands to include several months of data, better forecasting models could be developed in Tableau. These models would enable better predictions for demand and inventory management.

Another area for improvement is item clustering. By grouping similar items into clusters, the analysis could reveal deeper insights into product performance and shared behaviors across product categories. This clustering could support more targeted marketing and inventory decisions. Finally, incorporating seasonality analysis would provide a clearer understanding of how items and clusters perform during different times of the year. Tracking these seasonal trends could enhance strategic planning for inventory and sales, particularly during peak and off-peak periods. Together, these enhancements represent practical next steps in evolving the system into a more comprehensive and scalable solution capable of addressing complex retail challenges.

Conclusion

This project displays the efficiency and scalability of batch processing for Kafka streams in handling point-of-sale data. By combining the strengths of MongoDB and SQL, the system strikes a balance between fast data insertion and rapid querying for aggregated insights. The findings emphasize that data preparation and aggregation are crucial steps before meaningful analysis can be performed. Although the current setup supports forecasting and real-time analytics, achieving more descriptive and actionable insights will require a larger dataset and additional time for trend development. This system lays a strong foundation for advanced analytics in retail environments, demonstrating the potential for real-time data processing to drive strategic decision-making. With ongoing improvements, such as enhanced forecasting and clustering, the solution will continue to evolve.

References

How to run Kafka locally with Docker. Confluent. (n.d.). https://developer.confluent.io/confluent-tutorials/kafka-on-docker/

Real-time point-of-sale analytics. Databricks. (n.d.). https://www.databricks.com/solutions/accelerators/real-time-point-of-sale-analytics

Smith, B., & Saker, R. (2021, September 9). Real-time point-of-sale analytics with a Data Lakehouse. Databricks. https://www.databricks.com/blog/2021/09/09/real-time-point-of-sale-analytics-with-a-data-lakehouse.html