2ND EDITION

eBook

The Big Book of Data Science Use Cases

databricks

Contents

Introduction

The world of data science is evolving so fast that it's not easy to find real-world use cases that are relevant to what you're working on. That's why we've collected together these blogs from industry thought leaders with practical use cases you can put to work right now. This how-to reference guide provides everything you need — including code samples — so you can get your hands dirty working with the Databricks platform.

CHAPTER 1:

Solution Accelerator: Multi-factory Overall Equipment Effectiveness (OEE) and KPI Monitoring

By **Jeffery Annor, Tarik Boukherissa** and **Bala Amavasai**

November 29, 2022

The need to monitor and measure manufacturing equipment performance is critical for operational teams within manufacturing. The advancements in Industry 4.0 and smart manufacturing have enabled manufacturers to collect vast volumes of sensor and equipment data. Making sense of this data for productivity measures provides a critical competitive advantage.

To this end, Overall Equipment Effectiveness (OEE) has become the standard for measuring manufacturing equipment productivity. OEE is determined by the equation:

OEE = Machine Availability x Process Efficiency x Product Quality

According to Engineering USA, an OEE value of 85% and above is considered world-leading. However, many manufacturers achieve a typical range of between 40-60%. Anything below 40% is considered low.

In manufacturing, the OEE metric is used by different teams for a variety of objectives. On the manufacturing shop floor, for instance, OEE is used to identify lagging processes. Business executives, on the other hand, may use aggregated values to monitor the overall performance of their manufacturing business. In addition, OEE is used as an indicator to evaluate the need to modernize equipment, e.g., to justify CAPEX investments and to monitor the return on invested capital (ROIC). In this instance, OEE is used as the metric for sweating assets.

From an operational standpoint, it is important to work on the freshest information, and critical that information flows continuously to all stakeholders with minimum latency. From the scalability standpoint, it is important that all stakeholders work on the same information and be provided the ability to drill down to identify the behavior of an OEE drift. Both are very difficult to achieve with legacy systems.

The computation of OEE has traditionally been a manual exercise. In this Solution Accelerator, we demonstrate how the computation of OEE may be achieved in a multi-factory environment and in near real-time on Databricks.

The Databricks Lakehouse Platform

Databricks offers a complete platform to build forecasting solutions (from small scale to large) to help manufacturers maneuver through the challenges of ingesting and converging operational technology (OT) data with traditional data from IT systems, such as ERPs. These include:

- A structure to ingest, store and govern different types of data (structured, semi-structured, and unstructured) at scale based on open source format
- A managed solution to deploy distributed compute to ingest at any velocity, transform, orchestrate and query data without the need to copy data to/from multiple systems
- Collaborative notebooks (in Python, R, SQL, and Scala) that can be used to explore, enrich, and visualize data from multiple sources while accommodating business knowledge and domain expertise
- Fine-grained modeling and forecasting per item (e.g., product, SKU, or part) that can be parallelized, scaling to thousands, if not hundreds of thousands of items
- Integrated machine learning features, for example using MLflow to track experiments ensuring model reproducibility, traceability of performance metrics, and ease of deployment

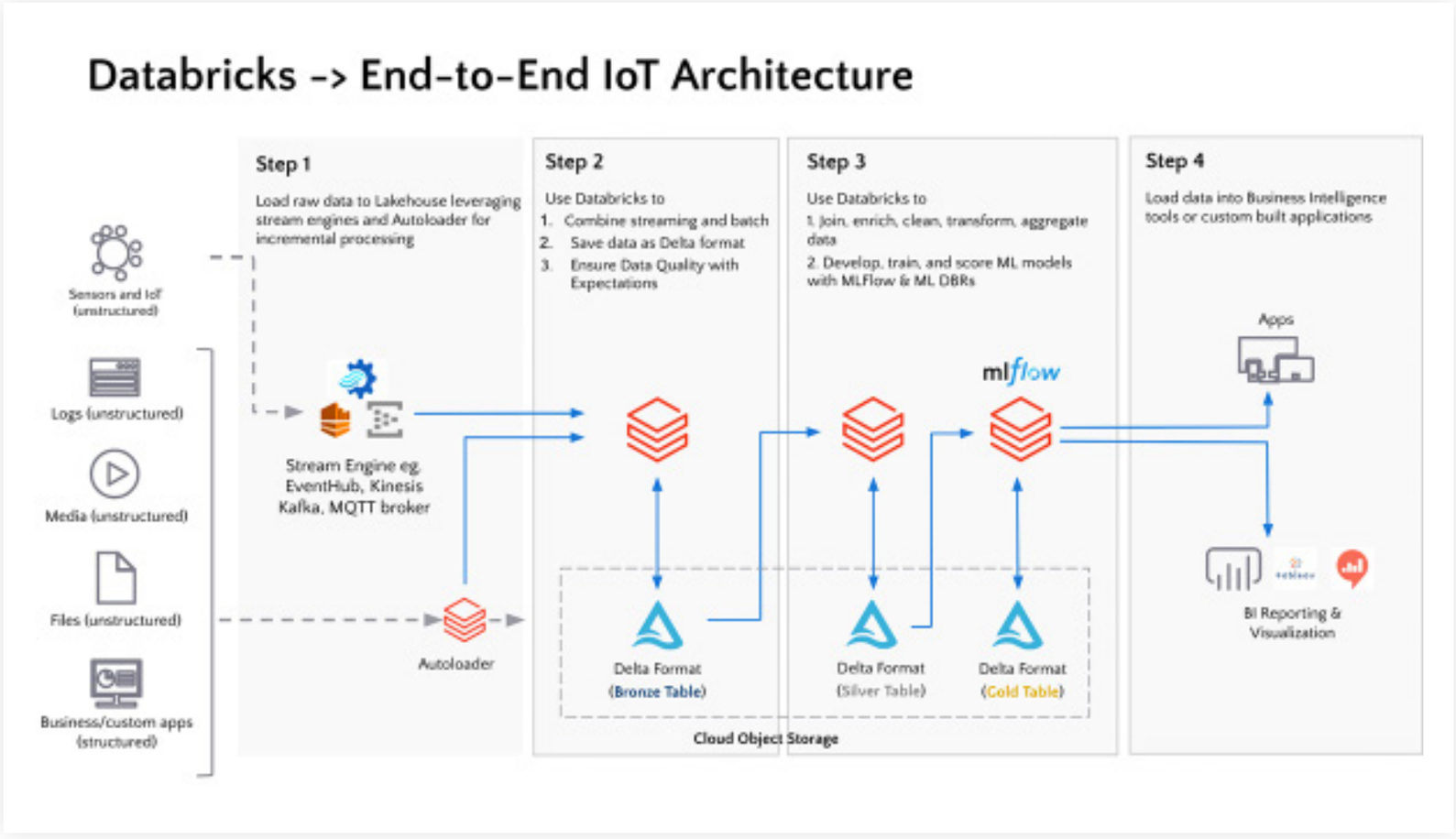
As a manufacturer, getting the latest data ensures that decisions are made using the latest information and that this unique source can feed the rest of the data pipeline in real time. As a data engineer, you would need a connector to the IoT source, a means to facilitate the process of sensor data (exactly once ingestion, late arrival management, data quality processing, and pushing the data from raw to aggregated layer with minimal latency).

The Databricks Lakehouse provides an end-to-end data engineering, serving, ETL, and machine learning platform that enables organizations to accelerate their analytics workloads by automating the complexity of building and maintaining analytics pipelines through open architecture and formats. This facilitates the connection to high-velocity Industrial IoT data using standard protocols like MQTT, Kafka, Event Hubs, or Kinesis to external data sets, like ERP systems, allowing manufacturers to converge their IT/OT data infrastructure for advanced analytics.

Our Solution Accelerator for OEE and KPI monitoring provides prebuilt notebooks and best practices to enable performant and scalable end-to-end monitoring.

The flow implemented in this Solution Accelerator is as follows:

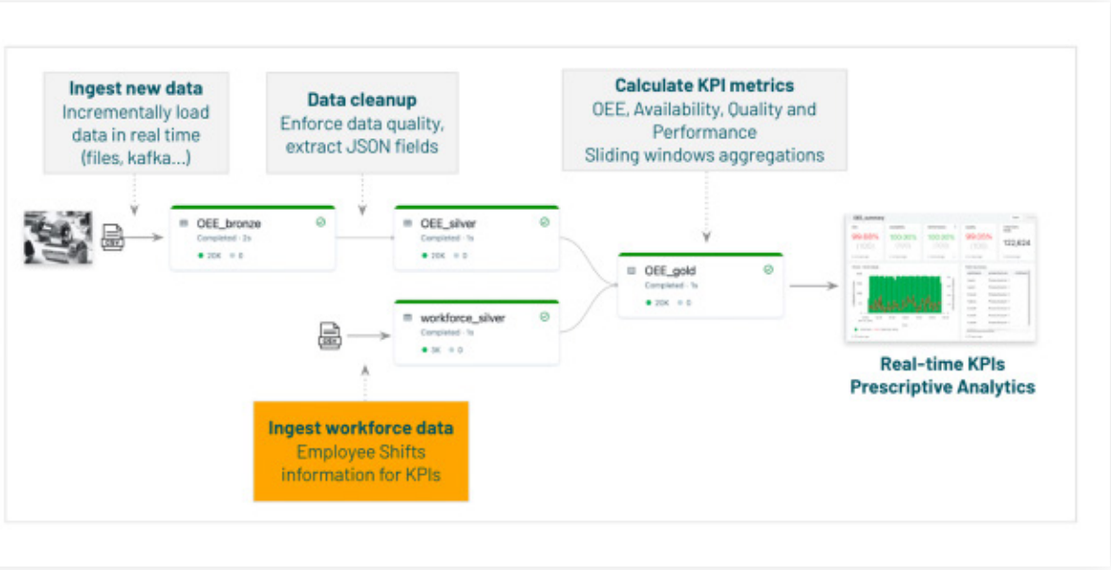
1. Incremental ingestion of data from the sensor/IoT devices
2. Cleanup of the data and extraction of the required information
3. Integration of workforce data set coming from our ERP systems
4. Merging of both data sets and real-time aggregation based on a temporal window
5. Computation and surfacing of KPIs and metrics to drive valuable insights



Fundamental to the lakehouse view of ETL/ELT is the usage of a multi-hop data architecture known as the medallion architecture.

Implementation

Data from devices are typically in non-standardized formats such as JSON and binary formats, hence the ability to parse and structure the payloads of data near real-time is important to implement such use cases.



Data architecture for OEE KPI monitoring.

Using a Delta Live Tables pipeline, we leverage the medallion architecture to ingest data from multiple sensors in a semi-structured format (JSON) into our Bronze layer where data is replicated in its natural format. The Silver layer transformations include parsing of key fields from sensor data that are needed to be extracted/ structured for subsequent analysis, and the ingestion of preprocessed workforce data from ERP systems needed to complete the analysis.

Finally, the Gold layer aggregates sensor data using Structured Streaming stateful aggregations, calculates OT metrics, e.g., OEE, TA (technical availability), and finally combines the aggregated metrics with workforce data based on shifts allowing for IT-OT convergence.

Surfacing outcomes

The computation of OEE itself is made up of three variables:

Availability: accounts for planned and unplanned stoppages, percentage of scheduled time that the operation is/was available to operate. This is given by (healthy_time - error_time)/(total_time)

Performance: a measure of the speed at which the work happens, percentage of its designed speed. This is given by healthy_time/ total_time

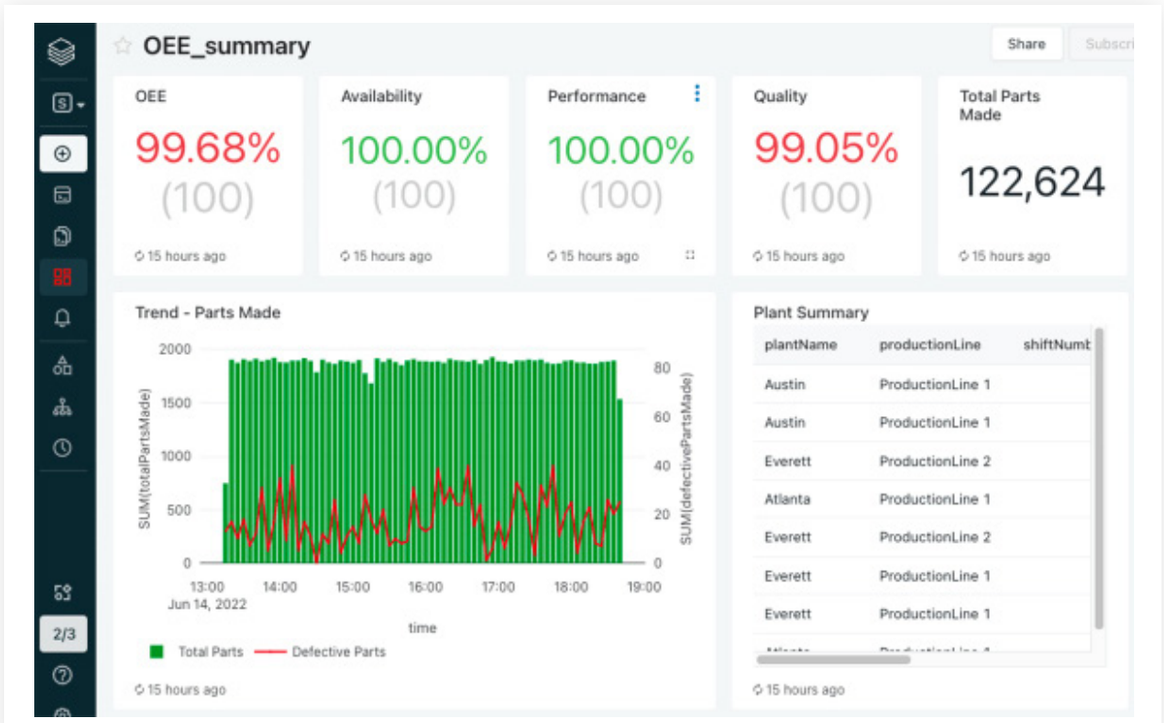
Quality: percentage of good units produced compared to the total units planned/produced. This is given by (Total Parts Made - Defective Parts Made)/ Total Parts Made

Recall that OEE = Availability x Performance x Quality.

This computation is relevant because performance improvement on any of these three KPIs will lead to better OEE. In addition, the three indicators above measure the division of actual vs ideal. OEE can be improved by the reduction of:

- Planned downtime
- Failures and breakdowns
- Micro stops
- Speed and throughput loss
- Production rejects
- Rejects on startup

Leveraging the Databricks SQL Workbench, users can leverage last mile queries. An example dashboard is shown in the figure below.



Dashboard for multi-factory monitoring.

Try our OEE Accelerator

Interested in seeing how this works or implementing it yourself? Check out the Databricks Solution Accelerator for OEE and KPI monitoring in manufacturing. By following the step-by-step instructions provided, users can learn how building blocks provided by Databricks can be assembled to enable performant and scalable end-to-end monitoring.

CHAPTER 2:

New Methods for Improving Supply Chain Demand Forecasting

Fine-grained demand forecasting with causal factors

Organizations are rapidly embracing fine-grained demand forecasting

Retailers and consumer goods manufacturers are increasingly seeking improvements to their supply chain management in order to reduce costs, free up working capital and create a foundation for omnichannel innovation. Changes in consumer purchasing behavior are placing new strains on the supply chain. Developing a better understanding of consumer demand via a demand forecast is considered a good starting point for most of these efforts as the demand for products and services drives decisions about the labor, inventory management, supply and production planning, freight and logistics, and many other areas.

In *Notes from the AI Frontier*, McKinsey & Company highlight that a 10% to 20% improvement in retail supply chain forecasting accuracy is likely to produce a 5% reduction in inventory costs and a 2% to 3% increase in revenues. Traditional supply chain forecasting tools have failed to deliver the desired results. With claims of industry-average inaccuracies of 32% in retailer supply chain demand forecasting, the potential impact of even modest forecasting improvements is immense for most retailers. As a result, many organizations are moving away from prepackaged forecasting solutions, exploring ways to bring demand forecasting skills in-house and revisiting past practices which compromised forecast accuracy for computational efficiency.

A key focus of these efforts is the generation of forecasts at a finer level of temporal and (location/product) hierarchical granularity. Fine-grained demand forecasts have the potential to capture the patterns that influence demand closer to the level at which that demand must be met. Whereas in the past a retailer might have predicted short-term demand for a class of products at a market level or distribution level, for a month or week period, and then used the forecasted values to allocate how many units of a specific product in that class should be placed in a given store and day, fine-grained demand forecasting allows forecasters to build more localized models that reflect the dynamics of that specific product in a particular location.

Fine-grained demand forecasting comes with challenges

As exciting as fine-grained demand forecasting sounds, it comes with many challenges. First, by moving away from aggregate forecasts, the number of forecasting models and predictions which must be generated explodes. The level of processing required is either unattainable by existing forecasting tools, or it greatly exceeds the service windows for this information to be useful. This limitation leads to companies making trade-offs in the number of categories being processed, or the level of grain in the analysis.

As examined in a prior blog post, Apache Spark™ can be employed to overcome this challenge, allowing modelers to parallelize the work for timely, efficient execution. When deployed on cloud-native platforms such as Databricks, computational resources can be quickly allocated and then released, keeping the cost of this work within budget.

The second and more difficult challenge to overcome is understanding that demand patterns that exist in aggregate may not be present when examining data at a finer level of granularity. To paraphrase Aristotle, the whole may often be greater than the sum of its parts. As we move to lower levels of detail in our analysis, patterns more easily modeled at higher levels of granularity may no longer be reliably present, making the generation of forecasts with techniques applicable at higher levels more challenging. This problem within the context of forecasting is noted by many practitioners going all the way back to Henri Theil in the 1950s.

As we move closer to the transaction level of granularity, we also need to consider the external causal factors that influence individual customer demand and purchase decisions. In aggregate, these may be reflected in the averages, trends and seasonality that make up a time series, but at finer levels of granularity, we may need to incorporate these directly into our forecasting models.

Finally, moving to a finer level of granularity increases the likelihood the structure of our data will not allow for the use of traditional forecasting techniques. The closer we move to the transaction grain, the higher the likelihood we will need to address periods of inactivity in our data. At this level of granularity, our dependent variables, especially when dealing with count data such as units sold, may take on a skewed distribution that's not amenable to simple transformations and which may require the use of forecasting techniques outside the comfort zone of many data scientists.

Accessing the historical data

See the Data Preparation notebook for details →

In order to examine these challenges, we will leverage public trip history data from the New York City Bike Share program, also known as Citi Bike NYC. Citi Bike NYC is a company that promises to help people "Unlock a Bike. Unlock New York." Their service allows people to go to any of over 850 various rental locations throughout the NYC area and rent bikes. The company has an inventory of over 13,000 bikes with plans to increase the number to 40,000. Citi Bike has well over 100,000 subscribers who make nearly 14,000 rides per day.

Citi Bike NYC reallocates bikes from where they were left to where they anticipate future demand. Citi Bike NYC has a challenge that is similar to what retailers and consumer goods companies deal with on a daily basis. How do we best predict demand to allocate resources to the right areas? If we underestimate demand, we miss revenue opportunities and potentially hurt customer sentiment. If we overestimate demand, we have excess bike inventory being unused.

This publicly available data set provides information on each bicycle rental from the end of the prior month all the way back to the inception of the program in mid-2013. The trip history data identifies the exact time a bicycle is rented from a specific rental station and the time that bicycle is returned to another rental station. If we treat stations in the Citi Bike NYC program as store locations and consider the initiation of a rental as a transaction, we have something closely approximating a long and detailed transaction history with which we can produce forecasts.

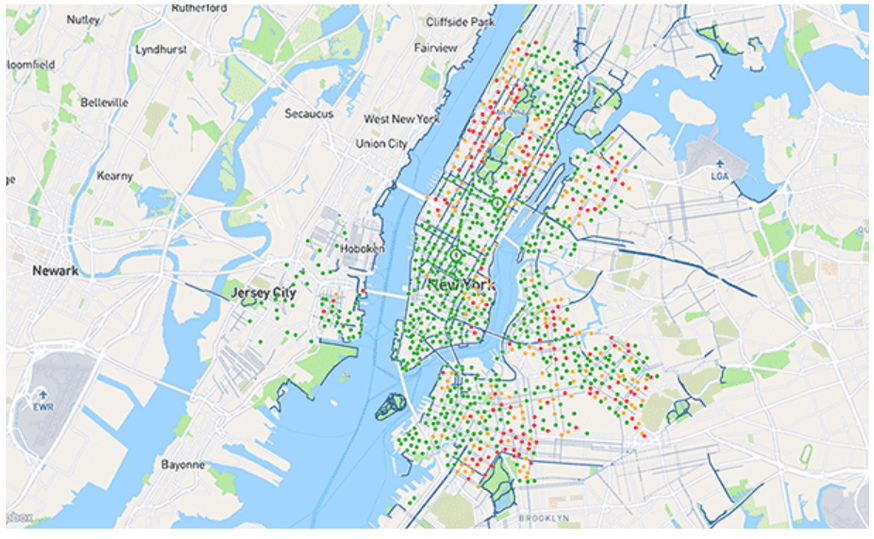
As part of this exercise, we will need to identify external factors to incorporate into our modeling efforts. We will leverage both holiday events as well as historical (and predicted) weather data as external influencers. For the holiday data set, we will simply identify standard holidays from 2013 to present using the holidays library in Python. For the weather data, we will employ hourly extracts from Visual Crossing, a popular weather data aggregator.

Citi Bike NYC and Visual Crossing data sets have terms and conditions that prohibit our directly sharing their data. Those wishing to recreate our results should visit the data providers' websites, review their Terms & Conditions, and download their data sets to their environments in an appropriate manner.
We will provide the data preparation logic required to transform these raw data assets into the data objects used in our analysis.

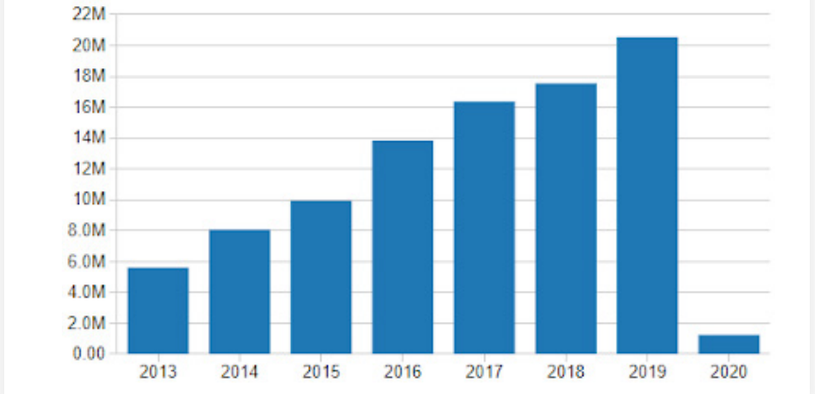
Examining the transactional data

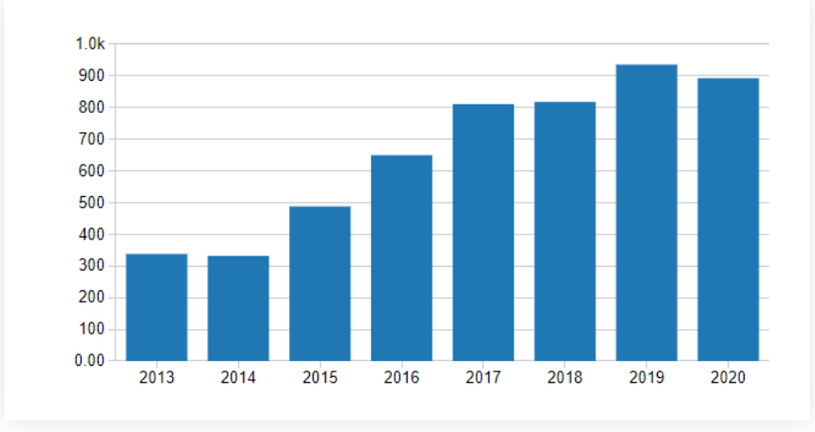
See the Exploratory Analysis notebook for details →

As of January 2020, the Citi Bike NYC bike share program consisted of 864 active stations operating in the New York City metropolitan area, primarily in Manhattan. In 2019 alone, a little over 4 million unique rentals were initiated by customers with as many as nearly 14,000 rentals taking place on peak days.

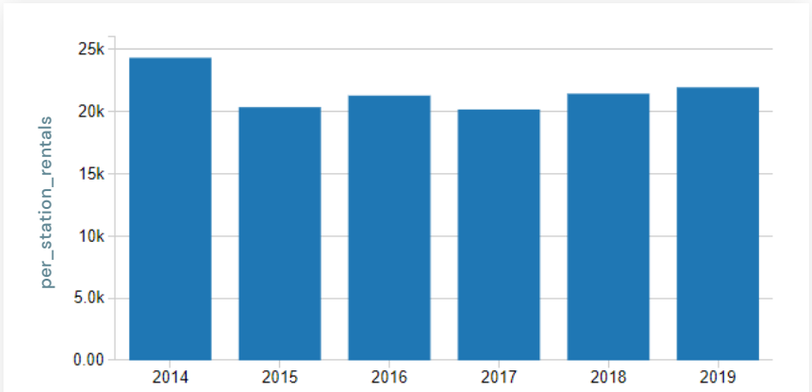


Since the start of the program, we can see the number of rentals has increased year over year. Some of this growth is likely due to the increased utilization of the bicycles, but much of it seems to be aligned with the expansion of the overall station network.

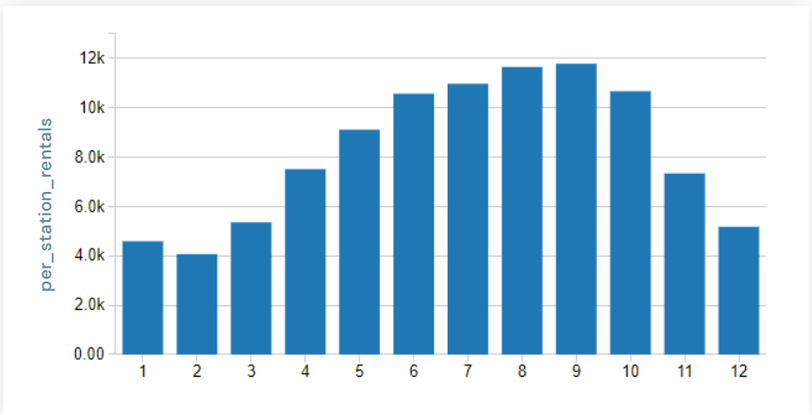




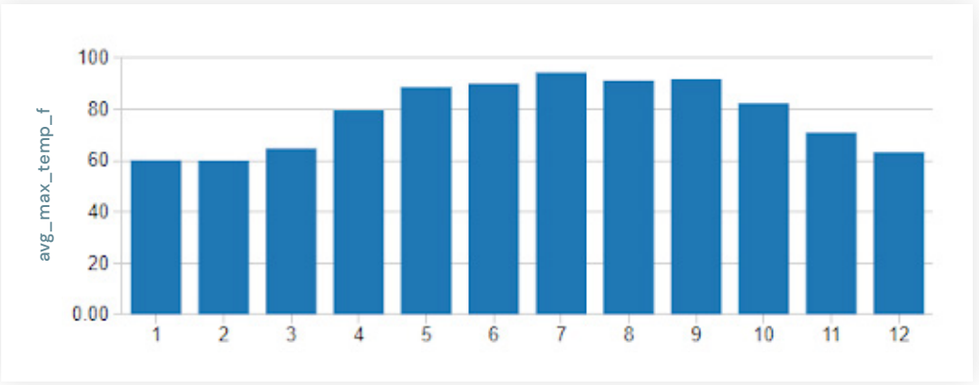
Normalizing rentals by the number of active stations in the network shows that growth in ridership on a per-station basis has been slowly ticking up for the last few years in what we might consider to be a slight linear upward trend.



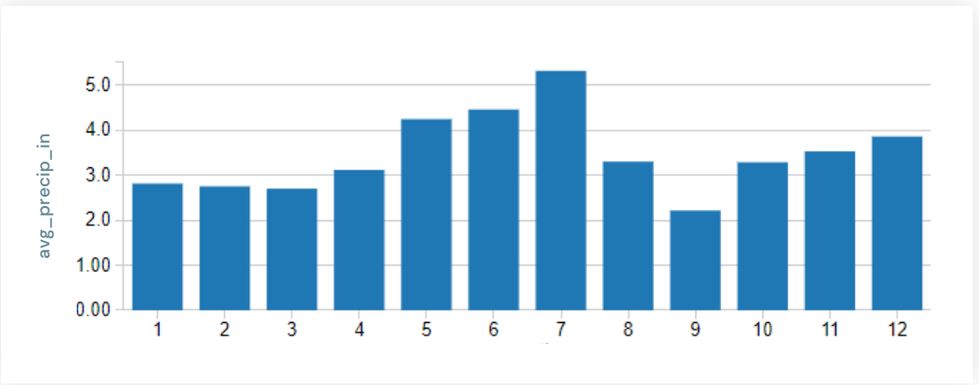
Using this normalized value for rentals, ridership seems to follow a distinctly seasonal pattern, rising in the spring, summer and fall and then dropping in winter as the weather outside becomes less conducive to bike riding.



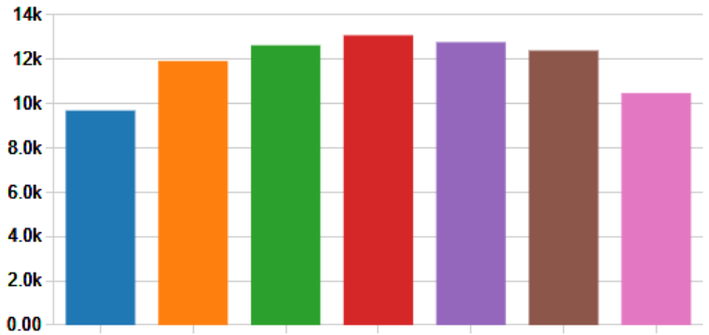
This pattern appears to closely follow patterns in the maximum temperatures (in degrees Fahrenheit) for the city.



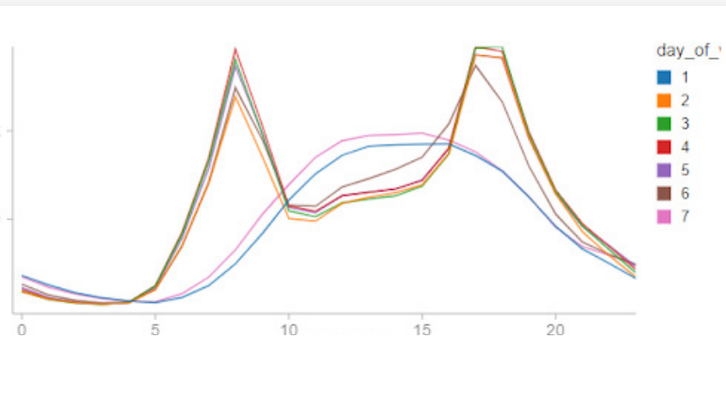
While it can be hard to separate monthly ridership from patterns in temperatures, rainfall (in average monthly inches) does not mirror these patterns quite so readily.



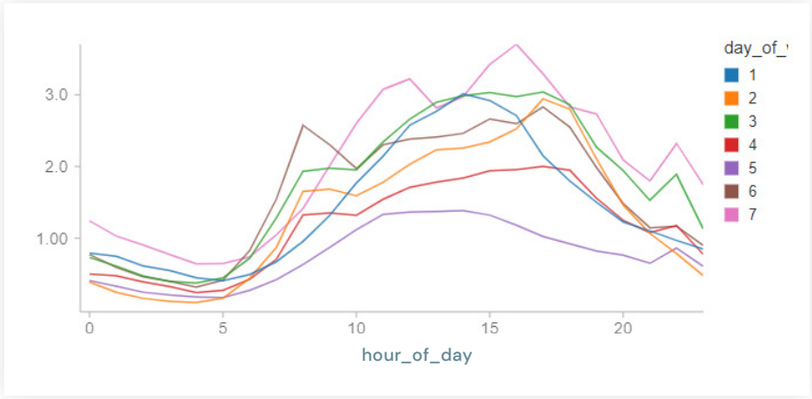
Examining weekly patterns of ridership with Sunday identified as 1 and Saturday identified as 7, it would appear that New Yorkers are using the bicycles as commuter devices, a pattern seen in many other bike share programs.



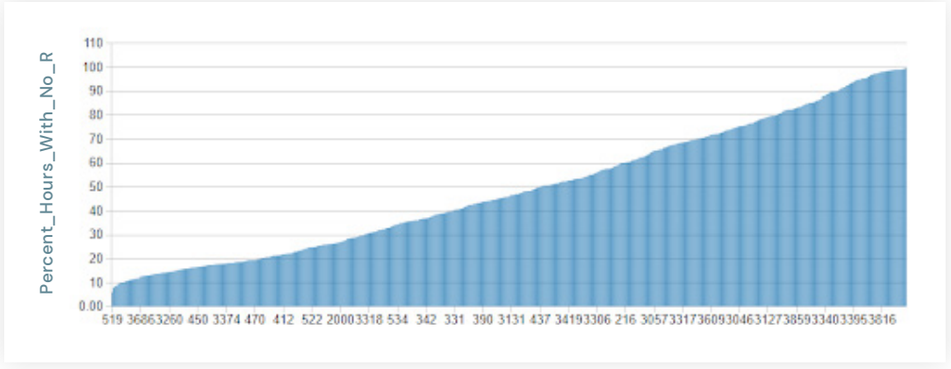
Breaking down these ridership patterns by hour of the day, we see distinct weekday patterns where ridership spikes during standard commute hours. On the weekends, patterns indicate more leisurely utilization of the program, supporting our earlier hypothesis.



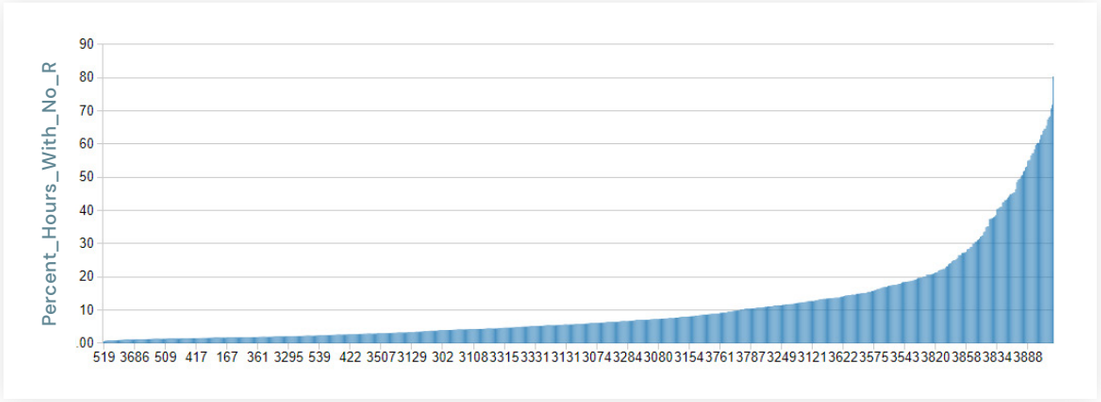
An interesting pattern is that holidays, regardless of their day of week, show consumption patterns that roughly mimic weekend usage patterns. The infrequent occurrence of holidays may be the cause of erraticism of these trends. Still, the chart seems to support that the identification of holidays is important to producing a reliable forecast.



In aggregate, the hourly data appear to show that New York City is truly the city that never sleeps. In reality, there are many stations for which there are a large proportion of hours during which no bicycles are rented.



These gaps in activity can be problematic when attempting to generate a forecast. By moving from 1-hour to 4-hour intervals, the number of periods within which individual stations experience no rental activity drops considerably, though there are still many stations that are inactive across this time frame.



Instead of ducking the problem of inactive periods by moving toward even higher levels of granularity, we will attempt to make a forecast at the hourly level, exploring how an alternative forecasting technique may help us deal with this data set. As forecasting for stations that are largely inactive isn't terribly interesting, we'll limit our analysis to the top 200 most active stations.

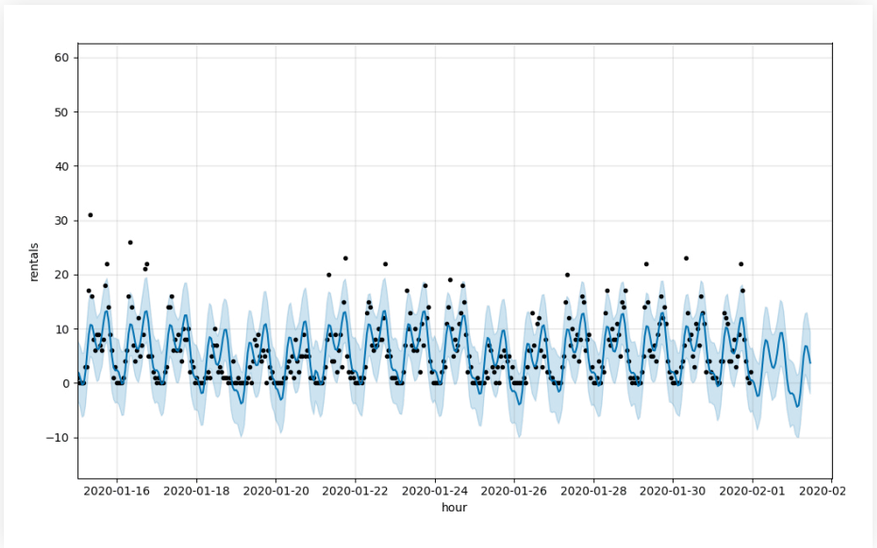
Forecasting bike share rentals with Facebook Prophet

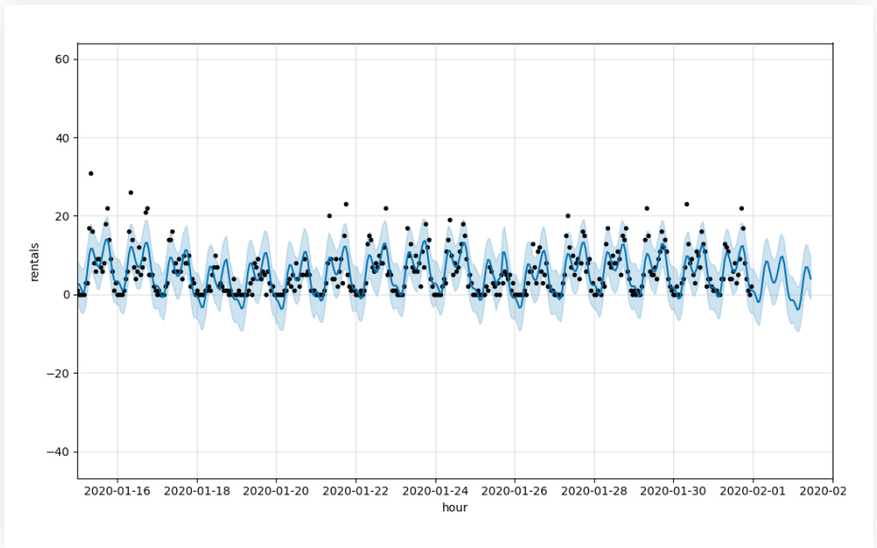
In an initial attempt to forecast bike rentals at the per-station level, we made use of Facebook Prophet, a popular Python library for time series forecasting. The model was configured to explore a linear growth pattern with daily, weekly and yearly seasonal patterns. Periods in the data set associated with holidays were also identified so that anomalous behavior on these dates would not affect the average, trend and seasonal patterns detected by the algorithm.

Using the scale-out pattern documented in the previously referenced blog post, models were trained for the most active 200 stations and 36-hour forecasts were generated for each. Collectively, the models had a Root Mean Squared Error (RMSE) of 5.44 with a Mean Average Proportional Error (MAPE) of 0.73. (Zero-value actuals were adjusted to 1 for the MAPE calculation.)

These metrics indicate that the models do a reasonably good job of predicting rentals but are missing when hourly rental rates move higher. Visualizing sales data for individual stations, you can see this graphically such as in this chart for Station 518, E 39 St and 2 Ave, which has an RMSE of 4.58 and a MAPE of 0.69:

See the Time Series notebook for details →





The model was then adjusted to incorporate temperature and precipitation as regressors. Collectively, the resulting forecasts had a RMSE of 5.35 and a MAPE of 0.72. While a very slight improvement, the models are still having difficulty picking up on the large swings in ridership found at the station level, as demonstrated again by Station 518, which had an RMSE of 4.51 and a MAPE of 0.68:

See the Time Series With Regressors notebook for details →

This pattern of difficulty modeling the higher values in both the time series models is typical of working with data having a Poisson distribution. In such a distribution, we will have a large number of values around an average with a long-tail of values above it. On the other side of the average, a floor of zero leaves the data skewed. Today, Facebook Prophet expects data to have a normal (Gaussian) distribution but plans for the incorporation of Poisson regression have been discussed.

Alternative approaches to forecasting supply chain demand

How might we then proceed with generating a forecast for these data? One solution, as the caretakers of Facebook Prophet are considering, is to leverage Poisson regression capabilities in the context of a traditional time series model. While this may be an excellent approach, it is not widely documented, so tackling this on our own before considering other techniques may not be the best approach for our needs.

Another potential solution is to model the scale of non-zero values and the frequency of the occurrence of the zero-valued periods. The output of each model can then be combined to assemble a forecast. This method, known as Croston's method, is supported by the recently released croston Python library while another data scientist has implemented his own function for it. Still, this is not a widely adopted method (despite the technique dating back to the 1970s) and our preference is to explore something a bit more *out-of-the-box*.

Given this preference, a random forest regressor would seem to make quite a bit of sense. Decision trees, in general, do not impose the same constraints on data distribution as many statistical methods. The range of values for the predicted variable is such that it may make sense to transform rentals using something like a square root transformation before training the model, but even then, we might see how well the algorithm performs without it.

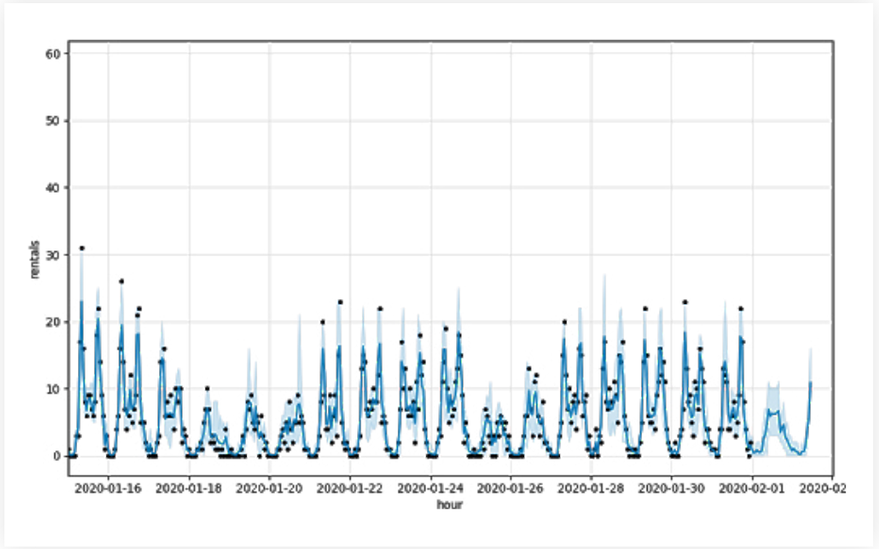
To leverage this model, we'll need to engineer a few features. It's clear from the exploratory analysis that there are strong seasonal patterns in the data, both at the annual, weekly and daily levels. This leads us to extract year, month, day of week and hour of the day as features. We may also include a flag for holidays.

Using a random forest regressor and nothing but time-derived features, we arrive at an overall RMSE of 3.4 and MAPE of 0.39. For Station 518, the RMSE and MAPE values are 3.09 and 0.38, respectively:

See Temporal notebook for details →

By leveraging precipitation and temperature data in combination with some of these same temporal features, we are able to better (though not perfectly) address some of the higher rental values. The RMSE for Station 518 drops to 2.14 and the MAPE to 0.26. Overall, the RMSE drops to 2.37 and MAPE to 0.26, indicating weather data is valuable in forecasting demand for bicycles.

See the Random Forest With Temporal & Weather Features notebook for details →



Implications of the results

Demand forecasting at finer levels of granularity may require us to think differently about our approach to modeling. External influencers which may be safely considered summarized in high-level time series patterns may need to be more explicitly incorporated into our models. Patterns in data distribution hidden at the aggregate level may become more readily exposed and necessitate changes in modeling approaches. In this data set, these challenges were best addressed by the inclusion of hourly weather data and a shift away from traditional time series techniques toward an algorithm which makes fewer assumptions about our input data.

There may be many other external influencers and algorithms worth exploring, and as we go down this path, we may find that some of these work better for some subset of our data than for others. We may also find that as new data arrives, techniques that previously worked well may need to be abandoned and new techniques considered.

A common pattern we are seeing with customers exploring fine-grained demand forecasting is the evaluation of multiple techniques with each training and forecasting cycle, something we might describe as an automated model bake-off. In a bake-off round, the model producing the best results for a given subset of the data wins the round with each subset able to decide its own winning model type. In the end, we want to ensure we are performing good data science where our data is properly aligned with the algorithms we employ, but as is noted in article after article, there isn't always just one solution to a problem and some may work better at one time than at others. The power of what we have available today with platforms like Apache Spark and Databricks is that we have access to the computational capacity to explore all these paths and deliver the best solution to our business.

Additional retail/CPG and demand forecasting resources

1. Sign up for a free trial and download these notebooks to start experimenting:
   - Data Preparation notebook
   - Exploratory Analysis notebook
   - Time Series notebook
   - Time Series With Regressors notebook
   - Temporal Notebook
   - Random Forest With Temporal & Weather Features notebook
2. Download our Guide to Data Analytics and AI at Scale for Retail and CPG
3. Visit our Retail and CPG page to learn how Dollar Shave Club and Zalando are innovating with Databricks
4. Read our recent blog Fine-Grained Time Series Forecasting At Scale With Facebook Prophet and Apache Spark to learn how Databricks Unified Data Analytics Platform addresses challenges in a timely manner and at a level of granularity that allows the business to make precise adjustments to product inventories

CHAPTER 3:

How to Build a Quality of Service (QoS) Analytics Solution for Streaming Video Services

By **Andrei Avramescu** and **Hector Leano**
May 6, 2020

Try this notebook in Databricks →

Click on the following link to view and download the QoS notebooks discussed below in this article.

CONTENTS

The importance of quality to streaming video services

As traditional pay TV continues to stagnate, content owners have embraced direct-to-consumer (D2C) subscription and ad-supported streaming for monetizing their libraries of content. For companies whose entire business model revolved around producing great content which they then licensed to distributors, the shift to now owning the entire glass-to-glass experience has required new capabilities such as building media supply chains for content delivery to consumers, supporting apps for a myriad of devices and operating systems, and performing customer relationship functions like billing and customer service.

With most vMVPD (virtual multichannel video programming distributor) and SVOD (streaming video on demand) services renewing on a monthly basis, subscription service operators need to prove value to their subscribers every month/week/day (the barriers to a viewer for leaving AVOD (ad-supported video on demand) are even lower — simply opening a different app or channel). General quality of streaming video issues (encompassing buffering, latency, pixelation, jitter, packet loss, and the blank screen) have significant business impacts, whether it's increased subscriber churn or decreased video engagement.

When you start streaming you realize there are so many places where breaks can happen and the viewer experience can suffer, whether it be an issue at the source in the servers on-prem or in the cloud; in transit at either the CDN level or ISP level or the viewer's home network; or at the playout level with player/client issues. What breaks at n x 104 concurrent streamers is different from what breaks at n x 105 or n x 106. There is no prerelease testing that can quite replicate real-world users and their ability to push even the most redundant systems to their breaking point as they channel surf, click in and out of the app, sign on from different devices simultaneously, and so on. And because of the nature of TV, things will go wrong during the most important, high-profile events drawing the largest audiences. If you start receiving complaints on social media, how can you tell if they are unique to that one user or rather a regional or a national issue? If national, is it across all devices or only certain types (e.g., possibly the OEM updated the OS on an older device type which ended up causing compatibility issues with the client)?

Identifying, remediating, and preventing viewer quality of experience issues becomes a big data problem when you consider the number of users, the number of actions they are taking, and the number of handoffs in the experience (servers to CDN to ISP to home network to client). Quality of Service (QoS) helps make sense of these streams of data so you can understand what is going wrong, where, and why. Eventually you can get into predictive analytics around what could go wrong and how to remediate it before anything breaks.

Databricks QoS solution overview

The aim of this solution is to provide the core for any streaming video platform that wants to improve their QoS system. It is based on the AWS Streaming Media Analytics Solution provided by AWS Labs, which we then built on top of to add Databricks as a Unified Data Analytics Platform for both the real-time insights and the advanced analytics capabilities.

By using Databricks, streaming platforms can get faster insights leveraging always the most complete and recent data sets powered by robust and reliable data pipelines, decreased time to market for new features by accelerating data science using a collaborative environment with support for managing the end-to-end machine learning lifecycle, and reduced operational costs across all cycles of software development by having a unified platform for both data engineering and data science.

Video QoS solution architecture

With complexities like low-latency monitoring alerts and highly scalable infrastructure required for peak video traffic hours, the straightforward architectural choice was the Delta architecture — both standard big data architectures like Lambda and Kappa architectures having disadvantages around operational effort required to maintain multiple types of pipelines (streaming and batch) and lack of support for unified data engineering and data science approach.

The Delta architecture is the next-generation paradigm that enables all the types of data personas in your organization to be more productive:

- Data engineers can develop data pipelines in a cost-efficient manner continuously without having to choose between batch and streaming
- Data analysts can get near real-time insights and faster answers to their BI queries
- Data scientists can develop better machine learning models using more reliable data sets with support for time travel that facilitates reproducible experiments and reports

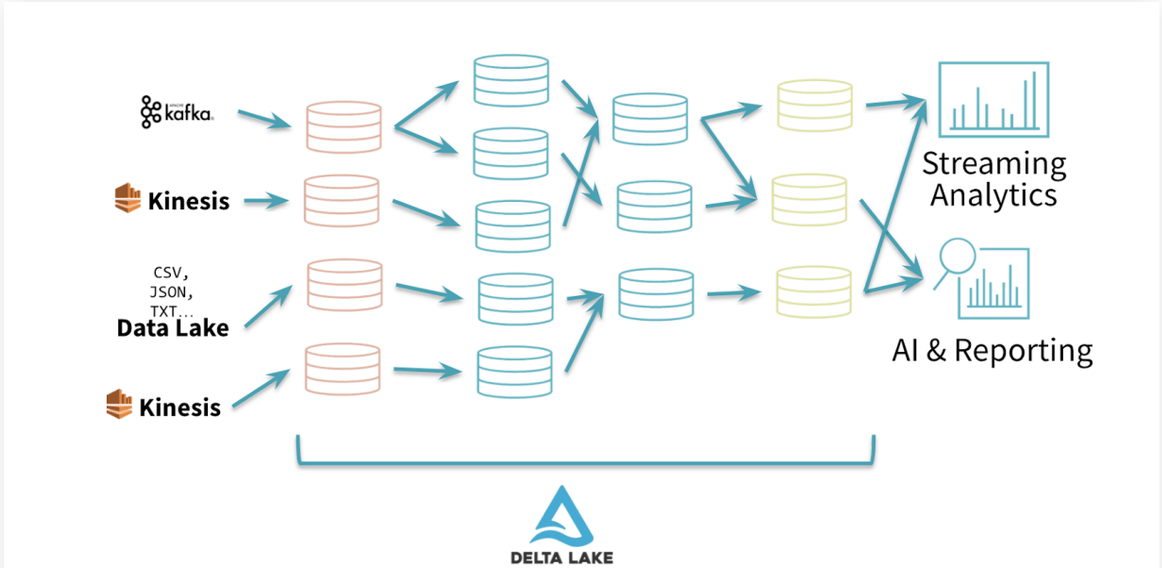


Figure 1. Delta architecture using the "multi-hop" approach for data pipelines.

Writing data pipelines using the Delta architecture follows the best practices of having a multi-layer "multi-hop" approach where we progressively add structure to data: "Bronze" tables, or ingestion tables, are usually raw data sets in the native format (JSON, CSV or txt), "Silver" tables represent cleaned/transformed data sets ready for reporting or data science, and "Gold" tables are the final presentation layer.

For the pure streaming use cases, the option of materializing the DataFrames in intermediate Delta tables is basically just a trade-off between latency/SLAs and cost (an example being real-time monitoring alerts vs updates of the recommender system based on new content).

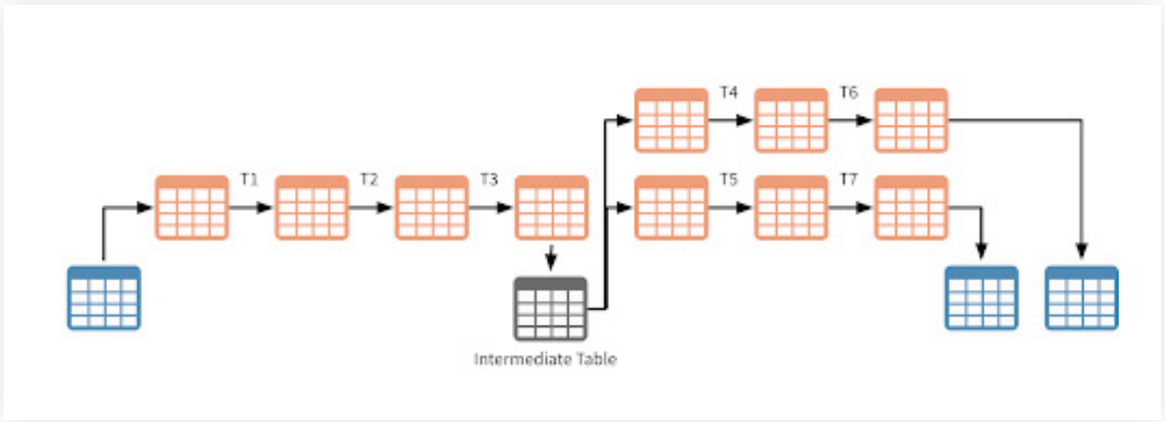


Figure 2. A streaming architecture can still be achieved while materializing DataFrames in Delta tables.

The number of "hops" in this approach is directly impacted by the number of consumers downstream, complexity of the aggregations (e.g., structured streaming enforces certain limitations around chaining multiple aggregations) and the maximization of operational efficiency.

The QoS solution architecture is focused around best practices for data processing and is not a full VOD (video-on-demand) solution — some standard components like the "front door" service Amazon API Gateway being avoided from the high-level architecture in order to keep the focus on data and analytics.

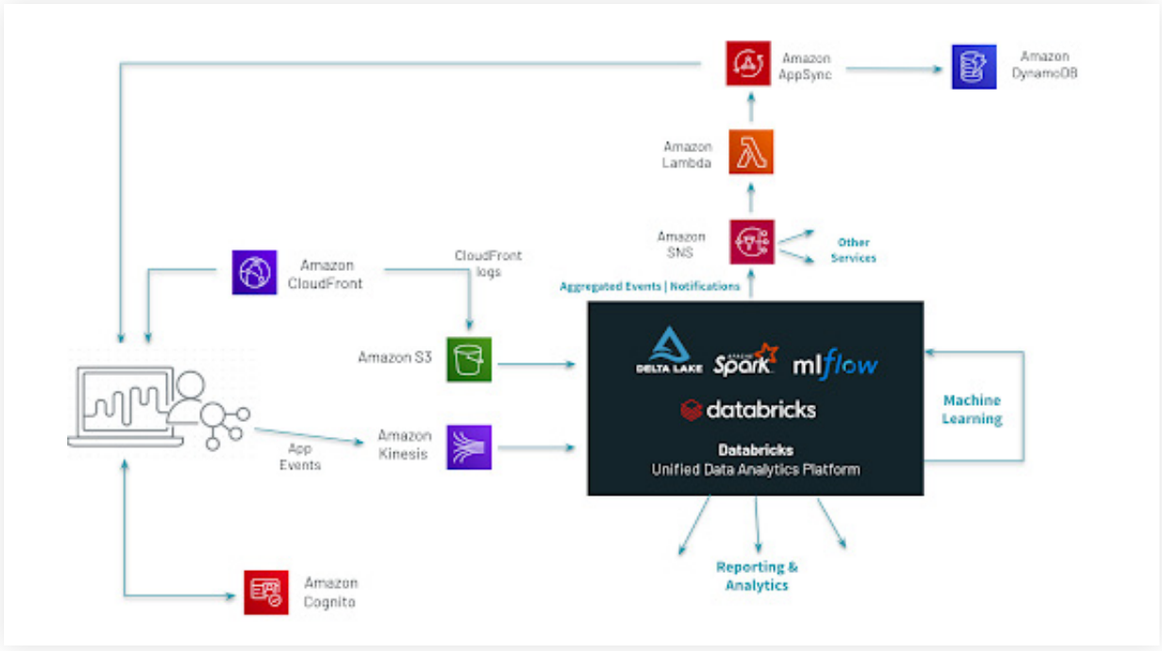


Figure 3. High-level architecture for the QoS platform.

Making your data ready for analytics

Both sources of data included in the QoS solution (application events and CDN logs) are using the JSON format, great for data exchange — allowing you to represent complex nested structures — but not scalable and difficult to maintain as a storage format for your data lake/analytics system.

In order to make the data directly queryable across the entire organization, the Bronze to Silver pipeline (the "make your data available to everyone" pipeline) should transform any raw formats into Delta and include all the quality checks or data masking required by any regulatory agencies.

Video applications events

Based on the architecture, the video application events are pushed directly to Kinesis Streams and then just ingested to a Delta append-only table without any changes to the schema.

type	ts	jsonData
stream	2020-04-08T09:53:58.864+0000	{"metrictype":"stream","at":105.077308,"rtt":350,"connection_type":"4g","package":"hls","resolution":"640x360","fps":"29.970","avg_bitrate":1330203,"duration":597,"cdn_tracking_id":"pxiujwunmzflqxa8soetfjrjvznfwsovfnh8epu0-6szzy38svl1vg==","user_id":"us-west-2:610f9e88-55e8-4a80-9b63-936446e6df2d","video_id":"bigbuckbunny","playlist_type":"live","timestamp":1586339638864}
stream	2020-04-08T09:54:00.396+0000	{"metrictype":"stream","at":415.111827,"rtt":528,"connection_type":"3g","package":"hls","resolution":"640x360","fps":"29.970","avg_bitrate":1350743,"duration":735,"cdn_tracking_id":"jahzaxapw4y-xrhrjoescatcytahbjvduwfgcvawqyihu4pagnrsya==","user_id":"us-west-2:cc7af3ef-6cf9-4da1-8274-3212db46be48","video_id":"tearsofsteel","playlist_type":"live","timestamp":1586339640396}
stream	2020-04-08T09:54:01.332+0000	{"metrictype":"stream","at":410.095889,"rtt":591,"connection_type":"4g","package":"hls","resolution":"640x360","fps":"29.970","avg_bitrate":1330203,"duration":597,"cdn_tracking_id":"svynftjqsrpmuy-zewwi4u3pbdn2sfgcaodm-2cgdpfiihhy4zlhka==","user_id":"us-west-2:83cb8ebe-4f29-4822-80a2-feb10095a9a2","video_id":"bigbuckbunny","playlist_type":"live","timestamp":1586339641332}
stream	2020-04-08T09:53:53.498+0000	{"metrictype":"stream","at":385.057842741,"rtt":523,"connection_type":"not available","package":"hls","resolution":"640x360","fps":"29.970","avg_bitrate":1350743,"duration":735,"cdn_tracking_id":"h3dpltndfpfy9acagudbie3kwck9gknh98fjjdu7uqhikigwagc8ga==","user_id":"us-west-2:2f7000b3-1e01-4cba-a0ef-d49b8952633c","video_id":"tearsofsteel","playlist_type":"live","timestamp":1586339633498}
buffer	2020-04-	{"metrictype":"buffer","buffer_type":"firstbuffer","at":385.057842741,"rtt":523,"connection_type":"not available","time_millisecond":386983,"cdn_tracking_id":"h3dpltndfpfy9acagudbie3kwck9gknh98fjjdu7uqhikigwagc8ga==","user_id":"us-west-2:2f7000b3-1e01-4cba-

Showing the first 1000 rows.

Figure 4. Raw format of the app events.

Using this pattern allows a high number of consumers downstream to process the data in a streaming paradigm without having to scale the throughput of the Kinesis stream. As a side effect of using a Delta table as a sink (which supports optimize!), we don't have to worry about the way the size of the processing window will impact the number of files in your target table — known as the "small files" issue in the big data world.

Both the timestamp and the type of message are being extracted from the JSON event in order to be able to partition the data and allow consumers to choose the type of events they want to process. Again combining a single Kinesis stream for the events with a Delta "Events" table reduces the operational complexity while making things easier for scaling during peak hours.

Schema:

col_name	data_type
browserfamily	string
bytes	string
cdn_source	string
isbot	boolean
origin	string
location	string
logdate	date
logtime	string
osfamily	string
requestid	string
ip	string
resulttype	string
year	int
month	int
day	int
hour	int

Figure 5. All the details are extracted from JSON for the Silver table.

CDN Logs

The CDN Logs are delivered to S3, so the easiest way to process them is the Databricks Auto Loader, which incrementally and efficiently processes new data files as they arrive in S3 without any additional setup.

```
    auto_loader_df = spark.readStream.format("cloudFiles") \
        .option("cloudFiles.format", "json") \
        .option("cloudFiles.region", region) \
        .load(input_location)

    anonymized_df = auto_loader_df.select('*', ip_
anonymizer('requestip').alias('ip'))\
        .drop('requestip')\
        .withColumn("origin", map_ip_to_location(col('ip')))
```

```
anonymized_df.writeStream \
    .option('checkpointLocation', checkpoint_location)\
    .format('delta') \
    .table(silver_database + '.cdn_logs')
```

As the logs contain IPs — considered personal data under the GDPR regulations — the "make your data available to everyone" pipeline has to include an anonymization step. Different techniques can be used, but we decided to just strip the last octet from IPv4 and the last 80 bits from IPv6. On top, the data set is also enriched with information around the origin country and the ISP provider which will be used later in the network operation centers for localization.

Creating the dashboard/virtual network operation centers

Streaming companies need to monitor network performance and the user experience as near real-time as possible, tracking down to the individual level with the ability to abstract at the segment level, easily defining new segments such as those defined by geos, devices, networks, and/or current and historical viewing behavior. For streaming companies that has meant adopting the concept of network operation centers (NOC) from telco networks for monitoring the health of the streaming experience for their users at a macro level, flagging and responding to any issues early on. At their most basic, NOCs should have dashboards that compare the current experience for users against a performance baseline so that the product teams can quickly and easily identify and attend to any service anomalies.

In the QoS solution we have incorporated a Databricks dashboard. BI Tools can also be effortlessly connected in order to build more complex visualizations, but based on customer feedback, built-in dashboards are most of the time the fastest way to present the insights to business users.

The aggregated tables for the NoC will basically be the Gold layer of our Delta architecture — a combination of CDN logs and the application events.

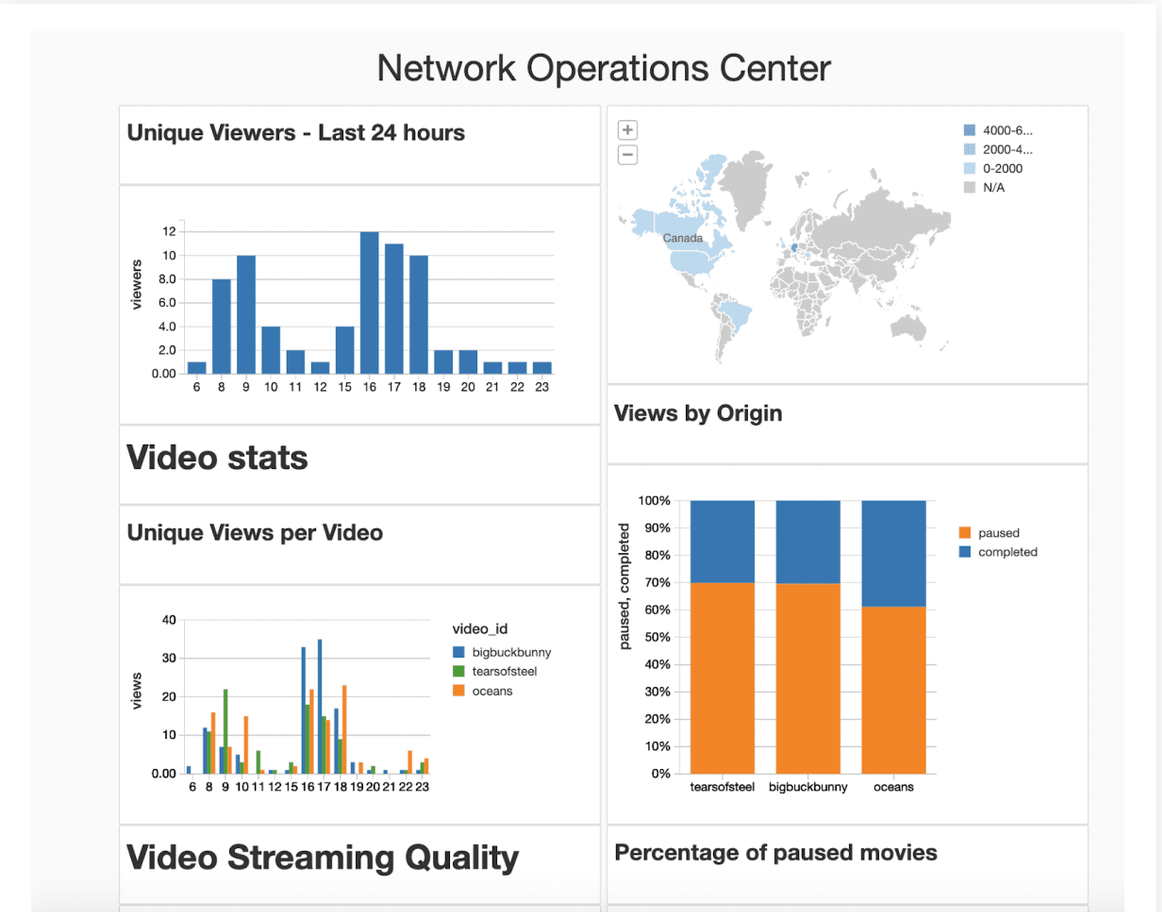


Figure 6. Example of Network Operations Center Dashboard.

The dashboard is just a way to visually package the results of SQL queries or Python/R transformation — each notebook supports multiple dashboards so in case of multiple end users with different requirements we don't have to duplicate the code — and as a bonus the refresh can also be scheduled as a Databricks job.

```
%sql

select origin, count(*)
from media_demo.user_activity
group by origin
```

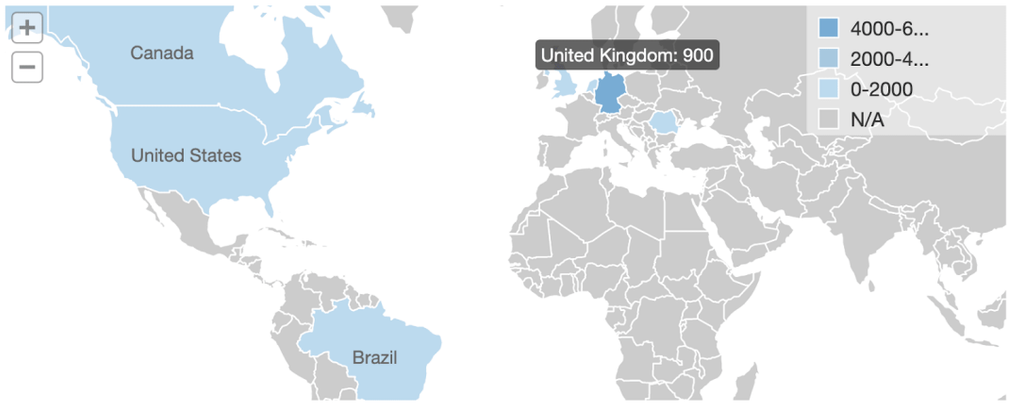
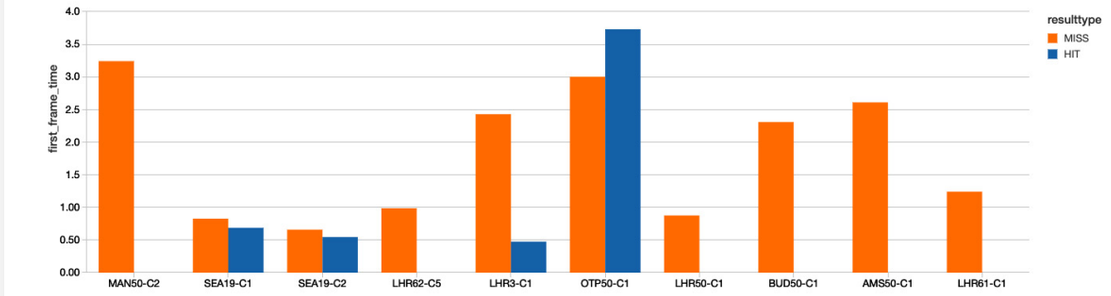
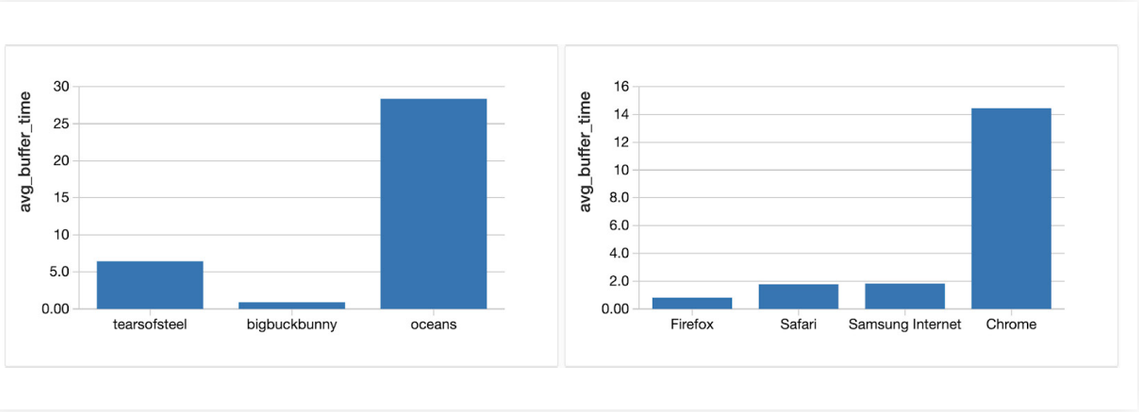


Figure 7. Visualization of the results of a SQL query.

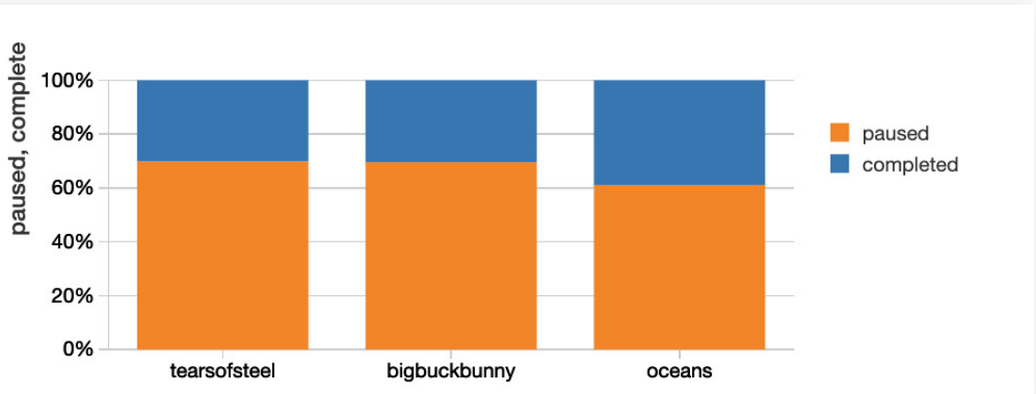
Loading time for videos (time to first frame) allows better understanding of the performance for individual locations of your CDN — in this case the AWS CloudFront Edge nodes — which has a direct impact in your strategy for improving this KPI, either by spreading the user traffic over multi-CDNs or maybe just implementing a dynamic origin selection in case of AWS CloudFront using Lambda@Edge.



Failure to understand the reasons for high levels of buffering — and the poor video quality experience that it brings — has a significant impact on subscriber churn rate. On top of that, advertisers are not willing to spend money on ads responsible for reducing the viewer engagement — as they add extra buffering on top, so the profits on the advertising business usually are impacted too. In this context, collecting as much information as possible from the application side is crucial to allow the analysis to be done not only at video level but also browser or even type/version of application.



On the content side, events for the application can provide useful information about user behavior and overall quality of experience. How many people that paused a video have actually finished watching that episode/video? Is the cause for stopping the quality of the content or are there delivery issues ? Of course further analyses can be done by linking all the sources together (user behavior, performance of CDNs/ISPs) to not only create a user profile but also to forecast churn.



Creating (near) real-time alerts

When dealing with the velocity, volume, and variety of data generated in video streaming from millions of concurrent users, dashboard complexity can make it harder for human operators in the NOC to focus on the most important data at the moment and zero in on root cause issues. With this solution, you can easily set up automated alerts when performance crosses certain thresholds that can help the human operators of the network as well as set off automatic remediation protocols via a Lambda function. For example:

- If a CDN is having latency much higher than baseline (e.g., if it's more than 10% latency versus baseline average), initiate automatic CDN traffic shifts
- If more than [some threshold, e.g., 5%] of clients report playback errors, alert the product team that there is likely a client issue for a specific device
- If viewers on a certain ISP are having higher than average buffering and pixelation issues, alert frontline customer representatives on responses and ways to decrease issues (e.g., set stream quality lower)

From a technical perspective generating real-time alerts requires a streaming engine capable of processing data real time and publish-subscribe service to push notifications.

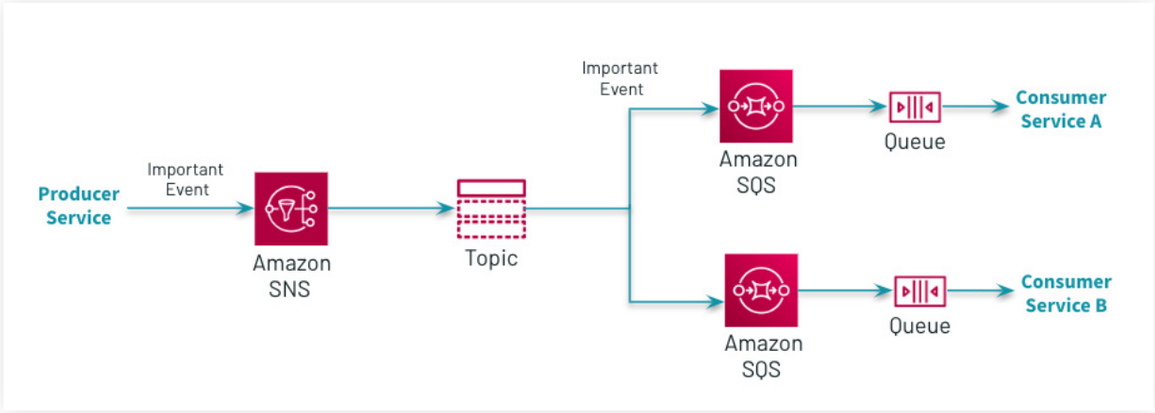


Figure 8. Integrating microservices using Amazon SNS and Amazon SQS.

The QoS solution implements the AWS best practices for integrating microservices by using Amazon SNS and its integrations with Amazon Lambda (see below the updates of web applications) or Amazon SQS for other consumers. The custom for each writer option makes the writing of a pipeline to send email notifications based on a rule-based engine (e.g., validating the percentage of errors for each individual type of app over a period of time) really straightforward.

```
    def send_error_notification(row):

        sns_client = boto3.client('sns', region)

        error_message = 'Number of errors for the App has exceeded the
threshold {}'.format(row['percentage'])

        response = sns_client.publish(
            TopicArn=<topicarn>,
            Message= error_message,
            Subject=<message>,
            MessageStructure='string')

        # Structured Streaming Job

        getKinesisStream("player_events")\
            .selectExpr("type", "app_type")\
            .groupBy("app_type")\
            .apply(calculate_error_percentage)\
            .where("percentage > {}".format(threshold)) \
            .writeStream\
            .foreach(send_error_notification)\
            .start()
</message></topicarn>
```

Figure 9. Sending email notifications using AWS SNS.

On top of the basic email use case, the Demo Player includes three widgets updated in real time using AWS AppSync: number of active users, most popular videos, number of users watching concurrently a video.

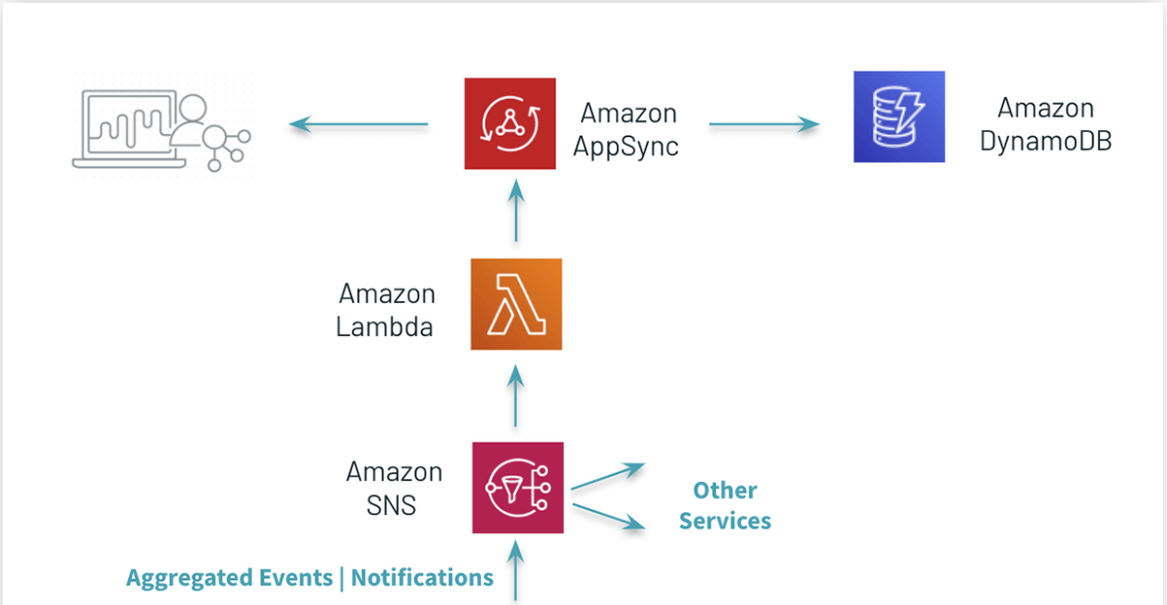


Figure 10. Updating the application with the results of real-time aggregations.

The QoS solution is applying a similar approach — Structured Streaming and Amazon SNS — to update all the values allowing for extra consumers to be plugged in using AWS SQS — a common pattern when huge volumes of events have to be enhanced and analyzed — pre-aggregate data once and allow each service (consumer) to make its own decision downstream.

Next steps: machine learning

Manually making sense of the historical data is important but is also very slow — if we want to be able to make automated decisions in the future, we have to integrate machine learning algorithms.

As a Unified Data Analytics Platform, Databricks empowers data scientists to build better data science products using features like the ML Runtime with the built-in support for Hyperopt / Horvod / AutoML or the integration with MLflow, the end-to-end machine learning lifecycle management tool.

We have already explored a few important use cases across our customer base while focusing on the possible extensions to the QoS solution.

Point-of-failure prediction and remediation

As D2C streamers reach more users, the costs of even momentary loss of service increases. ML can help operators move from reporting to prevention by forecasting where issues could come up and remediating before anything goes wrong (e.g., a spike in concurrent viewers leads to switching CDNs to one with more capacity automatically).

Customer churn

Critical to growing subscription services is keeping the subscribers you have. By understanding the quality of service at the individual level, you can add QoS as a variable in churn and customer lifetime value models. Additionally, you can create customer cohorts for those who have had video quality issues in order to test proactive messaging and save offers.

Getting started with the Databricks Streaming Video QoS Solution

Providing consistent quality in the streaming video experience is table stakes at this point to keep fickle audiences with ample entertainment options to stay on your platform. With this solution we have sought to create a quickstart for most streaming video platform environments to embed this QoS real-time streaming analytics solution in a way that:

- Scales to any audience size
- Quickly flags quality performance issues at key parts of the distribution workflow
- Is flexible and modular enough to easily customize for your audience and your needs such as creating new automated alerts or enabling data scientists to test and roll out predictive analytics and machine learning

> To get started, download the notebooks for the Databricks Streaming Video QoS Solution. For more guidance on how to unify batch and streaming data into a single system, view the Delta Architecture webinar.

CHAPTER 4:

Mitigating Bias in Machine Learning With SHAP and Fairlearn

Understand, find, and try to fix bias in machine learning applications with open source tools

By **Sean Owen**
September 16, 2022

Try this notebook in Databricks →

With good reason, data science teams increasingly grapple with questions of ethics, bias and unfairness in machine learning applications. Important decisions are now commonly made by models rather than people. Do you qualify for this loan? Are you allowed on this plane? Does this sensor recognize your face as, well, your face? New technology has, in some ways, removed human error, intended or otherwise. Far from solving the problem, however, technology merely reveals a new layer of questions.

It's not even obvious what bias means in a given case. Is it fair if a model recommends higher auto insurance premiums for men, on average? Does it depend on why the model does this? Even this simple question has plausible yet contradictory answers.

If we can't define bias well, we can't remedy it. This blog will examine sources of bias in the machine learning lifecycle, definitions of bias and how to quantify it. These topics are relatively well covered already, but not so much the follow-on question: what can be done about bias once found? So, this will also explore how to apply open source tools like SHAP and Fairlearn to try to mitigate bias in applications.

What is bias?

Most of us may struggle to define bias, but intuitively we feel that "we know it when we see it." Bias has an ethical dimension and is an idea we typically apply to people acting on people. It's not getting a job because of your long hair; it's missing a promotion for not spending time on the golf course.

This idea translates to machine learning models when models substitute for human decision-makers, and typically when these decisions affect people. It feels sensible to describe a model that consistently approves fewer loans for older people as "biased," but it's odd to call a flawed temperature forecasting model unfair, for example. (Machine learning uses "bias" as a term of art to mean "consistent inaccuracy," but that is not its sense here.)

Tackling this problem in machine learning, which is built on formalisms, math, code, and clear definitions, quickly runs into a sea of gray areas. Consider a model that recommends insurance premiums. Which of the following, if true, would suggest the model is fair?

- The model does not use gender as an input
- The model's input includes gender, but analysis shows that the effect of gender on the model's recommended premium is nearly zero
- The average premium recommended for men is the same as that for women
- Among those who caused an accident in the past year, the average premium recommended for men is the same as that for women

Individually, each statement sounds like a plausible criterion, and could match many people's intuition of fairness. They are, however, materially different statements, and that's the problem. Even if one believes that only one definition is reasonable in this scenario, just imagine a slightly different one. Does the analogous answer suffice if the model were recommending offering a coupon at a grocery store? Or recommending release from prison?

Where does bias come from?

Consider a predictive model that decides whether a person in prison will reoffend, perhaps as part of considering whether they qualify for early release. This is the example that this blog will actually consider. Let's say that the model consistently recommends release less frequently for African-American defendants, and that one agrees it is biased. How would this be remediated? Well, where did the bias come from?

Unfortunately, for data science teams, bias does not spring from the machine learning process. For all the talk of "model bias," it's not the models that are unfair. Shannon Vallor (among others) rightly pointed this out when Amazon made headlines for its AI-powered recruiting tool that was reported to be biased against women. In fact, the AI had merely learned to mimic a historically biased hiring process.

The ethical phenomenon of bias comes from humans in the real world. This blog post is not about changing the world, unfortunately.

It's not always humans or society. Sometimes data is biased. Data may be collected less accurately and less completely on some groups of people. All too frequently, headlines report that facial recognition technology underperforms when recognizing faces from racial minority groups. In no small part, the underperformance is due to lack of ethnic or minority representation in the training data. The real faces are fine, but the data collected on some faces is not. This blog will not address bias in data collection, though this is an important topic within the power of technologists to get right.

This leaves the models, who may turn out to be the heroes, rather than villains, of bias in machine learning. Models act as useful summaries of data, and data summarizes the real world, so models can be a lens for detecting bias, and a tool for compensating for it.

How can models help with bias?

Machine learning models are functions that take inputs (perhaps age, gender, income, prior history) and return a prediction (perhaps whether you will commit another crime). If you know that much, you may be asking, "Can't we just not include inputs like age, gender, race?" Wouldn't that stop a model from returning predictions that differ along those dimensions?

Unfortunately, no. It's possible for a model's predictions to be systematically different for men versus women even if gender is not an input into the model's training. Demographic factors correlate, sometimes surprisingly, with other inputs. In the insurance premium example, it's possible that "number of previous accidents," which sounds like a fair input for such a model, differs by gender, and thus learning from this value will result in a model whose results look different for men and women.

Maybe that's OK! The model does not explicitly connect gender to the outcome, and if one gender just has more accidents, then maybe it's fine that their recommended premiums are consistently higher.

If bias can be quantified, then model training processes stand a chance of trying to minimize it, just as modeling processes try to minimize errors in their predictions during training. Researchers have adopted many fairness metrics — equalized odds, equalized opportunity, demographic parity, predictive equality, and so on (see https://fairmlbook.org/classification.html). These definitions generally build on well-understood classification metrics like false positive rate, precision, and recall. Fairness metrics assert that classifier performance, as measured by some metric, is the same across subsets of the data, sliced by sensitive features that "shouldn't" affect the output.

This example considers two commonly used fairness metrics (later, using the tool Fairlearn): *equalized opportunity and equalized odds*. **Equalized opportunity** asserts that recall (true positive rate, TPR) is the same across groups. **Equalized odds** further asserts that the false positive rate (FPR) is also equal between groups. Further, these conditions assert these equalities conditional on the label. That is to say, in the insurance example above, equalized odds would consider TPR/FPR for men vs women, considering those that had high rates of accidents, separately from those that had none.

These metrics assert "equality of outcome," but do allow for inequality across subsets that were known to have, or not have, the attribute being predicted (perhaps getting in a car accident, or as we'll see, committing a crime again). That's less drastic than trying to achieve equality regardless of the ground truth.

There's another interesting way in which models can help with bias. They can help quantify the effect of features like age and race on the model's output, using the SHAP library. Once quantified, it's possible to simply subtract those effects. This is an intriguing middle-ground approach that tries to surgically remove unwanted effects from the model's prediction, by explicitly learning them first.

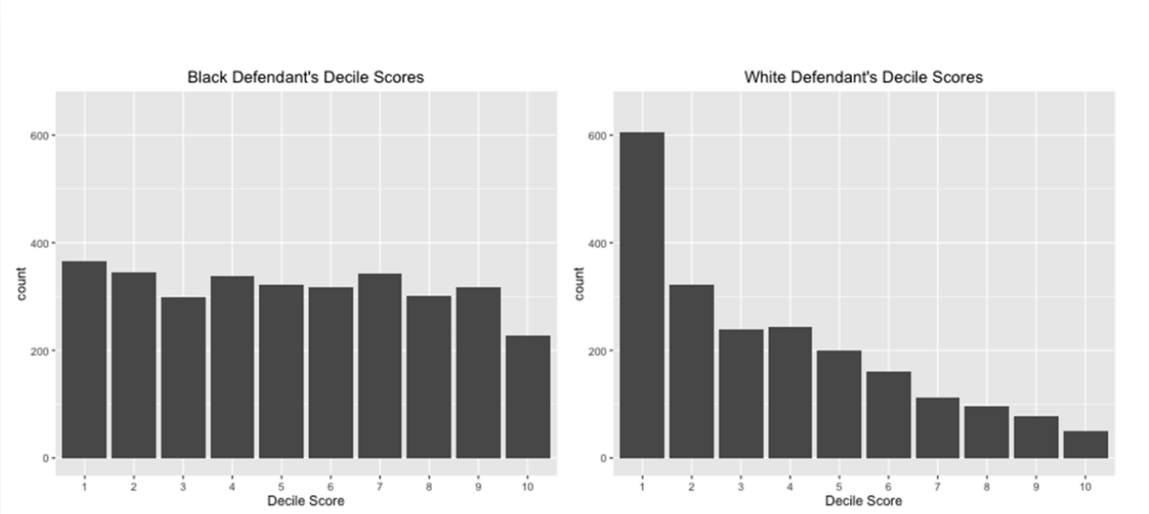
The remainder of this blog will explore how to apply these strategies for remediating bias with open source tools like SHAP, Fairlearn and XGBoost, to a well-publicized problem and data set: predicting recidivism, and the COMPAS data set.

Recidivism, and the COMPAS data set

In 2016, ProPublica analyzed a system called COMPAS. It is used in Broward County, Florida, to help decide whether persons in prison should be released or not on parole, based on whether they are predicted to commit another crime (recidivate?). ProPublica claimed to find that the system was unfair. Quoting their report:

- … black defendants who did not recidivate over a two-year period were nearly twice as likely to be misclassified as higher risk compared to their white counterparts (45% vs 23%)
- … white defendants who re-offended within the next two years were mistakenly labeled low risk almost twice as often as black re-offenders (48% vs 28%)

COMPAS uses a predictive model to assign risk decile scores, and there seemed to be clear evidence it was biased against African-Americans. See, for instance, the simple distribution of decile scores that COMPAS assigns to different groups of defendants:



COMPAS recidivism score by decile, for black vs white defendants. Lower deciles mean "less risk."

However, whether one judges this system "fair" again depends on how "fairness" is defined. For example, Abe Gong reached a different conclusion using the same data. He argues that there is virtually no difference in the models' outputs between races if one accounts for the observed probability of reoffending. Put more bluntly, historical data collected on released prisoners suggested that African-American defendants actually reoffended at a higher rate compared to other groups, which explains a lot of the racial disparity in model output.

The purpose here is not to analyze COMPAS, the arguments for or against it, or delve into the important question of why the observed difference in rate of reoffending might be so. Instead, imagine that we have built a predictive model like COMPAS to predict recidivism, and now we want to analyze its bias — and potentially mitigate it.

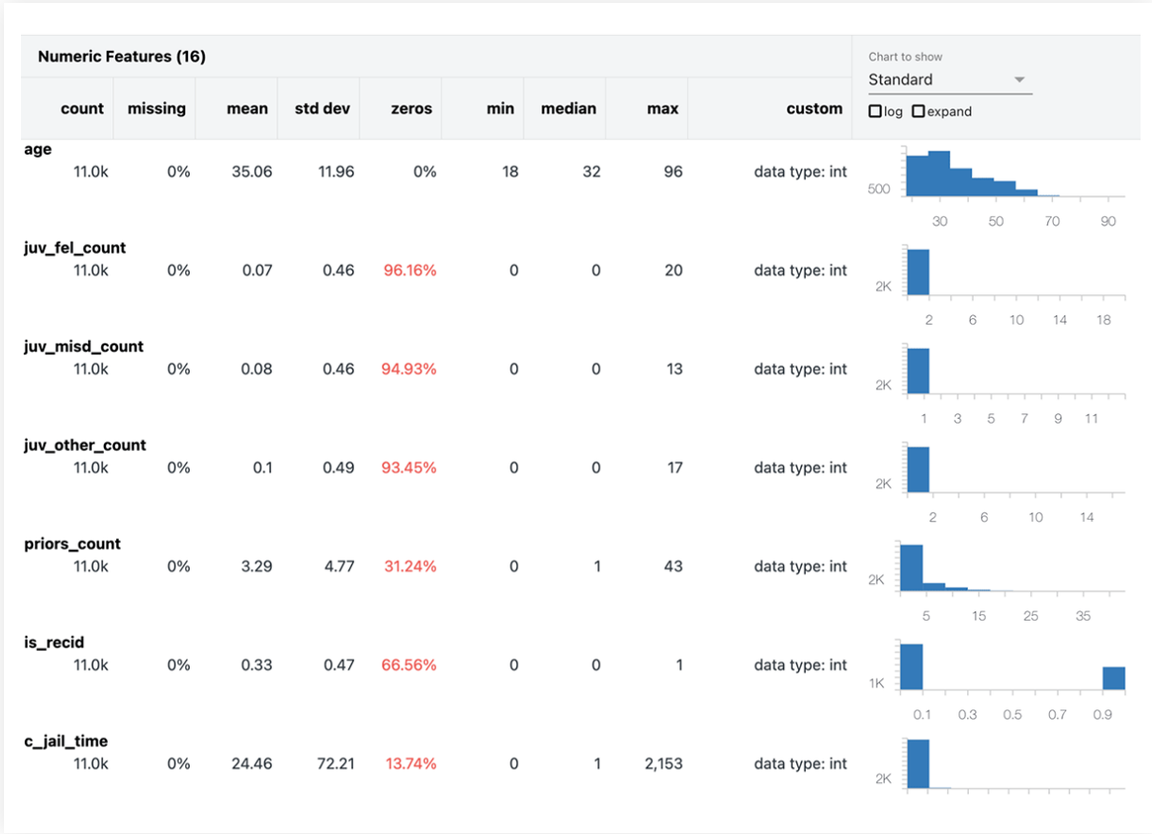
Thankfully, ProPublica did the hard work of compiling the data set and cleaning it. The data set contains information on 11,757 defendants, including their demographics, name and some basic facts about their prior history of arrests. It also indicates whether the person ultimately was arrested again within 2 years for a violent or nonviolent crime. From this data, it's straightforward to build a model.

A first model

It's not necessary to build the best possible model here, but merely to build a reasonably good one, in order to analyze the results of the model, and then experiment with modifications to the training process. A simple pass at this might include:

- Turning ProPublica's sqlite DB into CSV files
- Some feature engineering — ignore defendant name, columns with the same data encoded differently
- Ignore some qualitative responses from the defendant questionnaire in favor of core quantitative facts
- Drop rows missing key data like the label (reoffended or not)
- Fit a simple XGBoost model to it, tuning it with Hyperopt and Spark
- Observe and retrieve the hyperparameter tuning results with MLflow

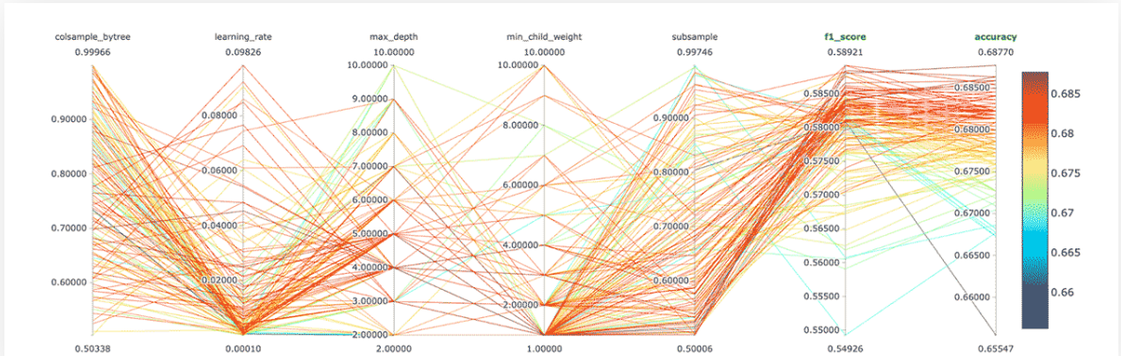
These choices are debatable, and certainly not the only possible approach. These are not the point of this post, and the details are left to the accompanying notebook. For interest, here is a data profile of this data set, which is available when reading any data in a Databricks notebook:



Profile of training data set, from Databricks.

The data set includes demographic information like age, gender and race, in addition to the number of prior convictions, charge degree, and so on. is_recid is the label to predict — did the person recidivate?

This yields a model with 68.7% accuracy, after a brief hyperparameter search over hundreds of tunings of an XGBoost model with Hyperopt and Spark. The accompanying notebook further uses MLflow to automatically track the models and results; practitioners can rerun and review the tuning results (figure below).



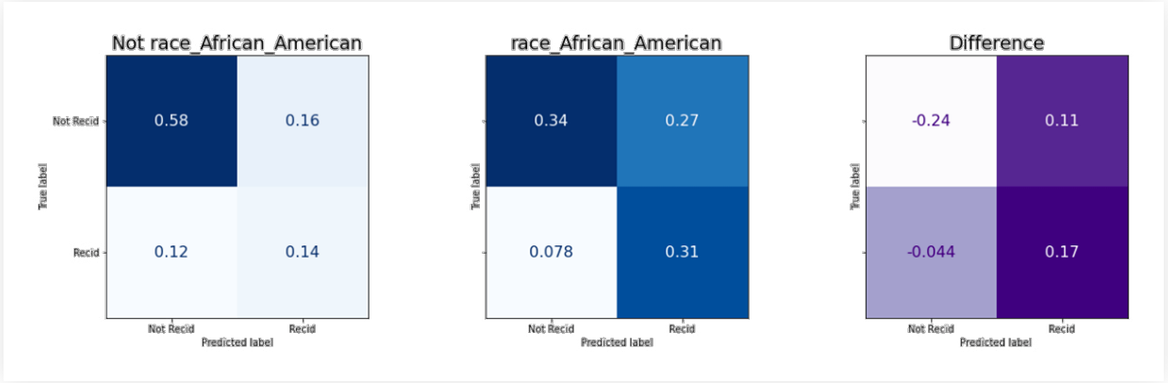
Parallel coordinates plot showing hyperparameters (left) and metrics (right) of models evaluated during tuning.

With a trained model in hand, we can begin to assess the training process for evidence of bias in its results. Following the ProPublica study, this example will specifically examine bias that affects African-American defendants. This is not to say that this is the only, or most important, form of bias that could exist.

Attempt 1: Do nothing

It's useful to first assess fairness metrics on this base model, without any mitigation strategies. The model includes inputs like race, gender and age. This example will focus on equalized odds, which considers true and false positive rates across classes. Standard open source tools can compute these values for us. Microsoft's Fairlearn has utility methods to do so, and with a little Matplotlib plotting, we get some useful information out:

```
def show_confusion_matrix_diff(data_experiment_tag,
                               drop_cols=[],
                               race="African_American"):
  recid_df = get_recid_data(data_experiment_tag, drop_cols)
  def get_cm(label):
    data_pd = recid_df[recid_df[f"race_{race}"] == label][["is_recid",
"prediction"]]
    return confusion_matrix(data_pd['is_recid'], data_pd['prediction'],
                            normalize='all')

  not_race_cm = get_cm(0)
  race_cm = get_cm(1)

  _, axes = plt.subplots(1, 3, figsize=(20,5), sharey='row')
  plt.rcParams.update({'font.size': 16})
  labels = ["Not Recid", "Recid"]
  ConfusionMatrixDisplay(not_race_cm, display_labels=labels).\
    plot(ax=axes[0], cmap='Blues', colorbar=False).\
    ax_.set_title(f"Not race_{race}")
  ConfusionMatrixDisplay(race_cm, display_labels=labels).\
    plot(ax=axes[1], cmap='Blues', colorbar=False).\
    ax_.set_title(f"race_{race}")
  ConfusionMatrixDisplay(race_cm - not_race_cm, display_labels=labels).\
    plot(ax=axes[2], cmap='Purples', colorbar=False).\
    ax_.set_title("Difference")
  plt.show()
```



Confusion matrices for African-American defendants vs rest, and difference, for baseline model.

The two blue matrices are confusion matrices, showing classifier performance on non-African-American defendants, and African-American ones. The four values represent the fraction of instances where the classifier correctly predicted that the person would not reoffend (top left) or would reoffend (bottom right). The top right value is the fraction of false positives, and bottom left are false negatives. The purple matrix at the right is simply the difference — rate for African-Americans, minus rate for others.

It's plain that the model makes a positive (will offend) prediction more often for African-American defendants (right columns). That doesn't feel fair, but so far this is just a qualitative assessment. What does equalized odds tell us, if we examine actual TPR/FPR?

```
def show_fairness_metrics(data_experiment_tag, drop_cols=[]):
  recid_df = get_recid_data(data_experiment_tag, drop_cols)
  metrics = {
    "accuracy": accuracy_score,
    "recall": recall_score,
    "false positive rate": false_positive_rate,
    "false negative rate": false_negative_rate,
    "selection rate": selection_rate
  }
  mf = MetricFrame(metrics=metrics, y_true=y_val,
                   y_pred=recid_df["prediction"],
                   sensitive_features=X_val["race_African_American"],
                   control_features=y_val).by_group
# Update the run with new metrics
(fpr_not_af_am, fpr_af_am, _, _) = mf['false positive rate']
(_, _, fnr_not_af_am, fnr_af_am) = mf['false negative rate']
run_id = find_best_run_id(data_experiment_tag)
with mlflow.start_run(run_id=run_id):
  mlflow.log_metrics({
    "Not Af-Am FPR": fpr_not_af_am,
    "Af-Am FPR": fpr_af_am,
    "Not Af-Am FNR": fnr_not_af_am,
    "Af-Am FNR": fnr_af_am
  })
return mf
```

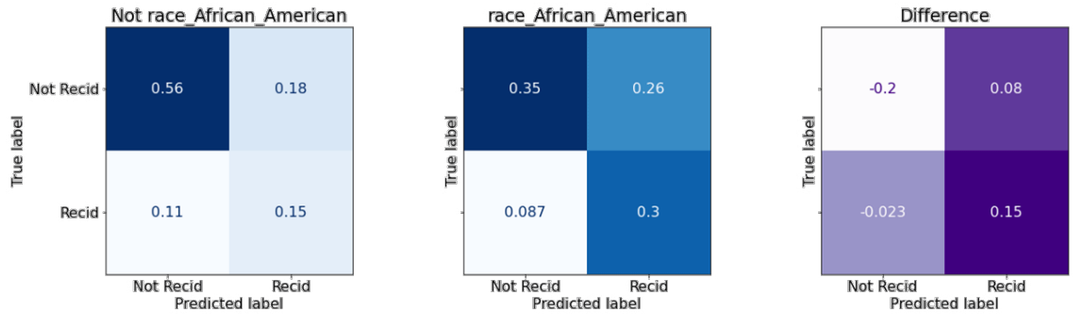
is_recid	race_African_American	accuracy	recall	false positive rate	false negative rate	selection rate
0	0	0.786335	0.0	0.213665	0.0	0.213665
	1	0.558824	0.0	0.441176	0.0	0.441176
1	0	0.533101	0.533101	0.0	0.466899	0.533101
	1	0.798144	0.798144	0.0	0.201856	0.798144

Classifier metrics, broken down by race and ground truth, for baseline model.

This table has the values that matter for equalized odds. Recall (true positive rate) and false positive rate are broken down by label (did the person actually reoffend later?) and status as African-American or not. There is a difference of 26.5% and 22.8%, respectively, between African-American and non-African-American defendants. If a measure like equalized odds is the measure of fairness, then this would be "unfair." Let's try to do something about it.

Attempt 2: Ignore demographics

What if we just don't tell the model the age, gender or race of the defendants? Is it sufficient? Repeat everything above, merely removing these inputs to the model.



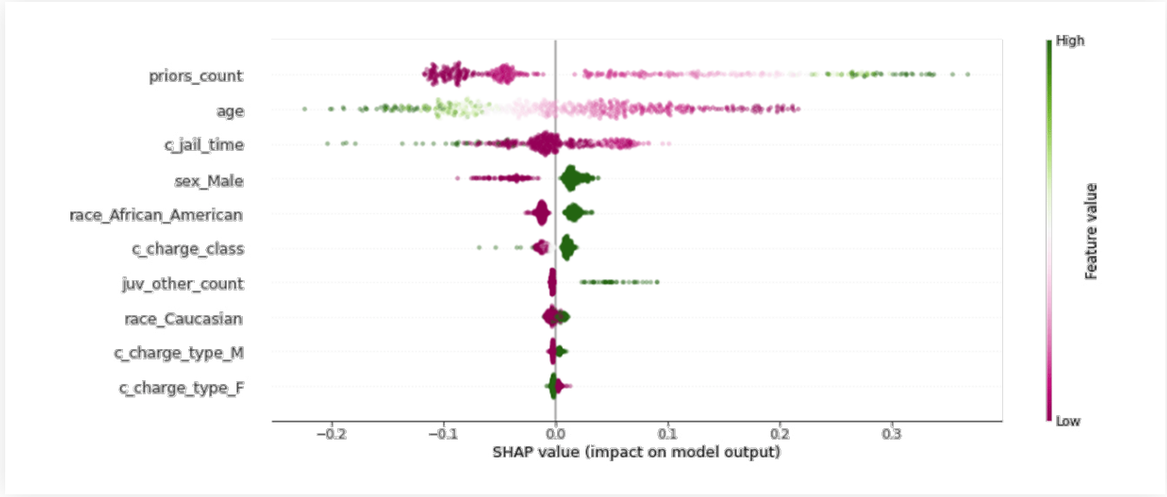
Confusion matrices for African-American defendants vs rest, and difference, when demographic inputs are omitted.

is_recid	race_African_American	accuracy	recall	false positive rate	false negative rate	selection rate
0	0	0.756522	0.0	0.243478	0.0	0.243478
	1	0.576471	0.0	0.423529	0.0	0.423529
1	0	0.581882	0.581882	0.0	0.418118	0.581882
	1	0.774942	0.774942	0.0	0.225058	0.774942

Classifier metrics, broken down by race and ground truth, when demographics are omitted.

The result is, maybe surprisingly, similar. The gap in TPR and FPR has shrunk somewhat to 19.3% and 18%. This may be counterintuitive. What is the model reasoning about that results in this difference, if it doesn't know race? To answer this, it's useful to turn to SHAP (SHapley Additive Explanations) to explain what the model is doing with all of these inputs:

```
def draw_summary_plot(data_experiment_tag, drop_cols=[]):
  run_id = find_best_run_id(data_experiment_tag)
  model = mlflow.xgboost.load_model(f"runs:/{run_id}/model")

  train_sample = X_train.drop(drop_cols, axis=1)
  example_samples = np.random.choice(np.arange(len(X_val)), 500,
replace=False)
  example = X_val.drop(drop_cols, axis=1).iloc[example_samples]

  explainer = TreeExplainer(model, train_sample, model_
output="probability")
  shap_values = explainer.shap_values(example, y=y_val.iloc[example_
samples])
  summary_plot(shap_values, example, max_display=10, alpha=0.4,
               cmap="PiYG", plot_size=(14,6))
```



SHAP plot explaining model predictions and most-important features for baseline model.

In the above graphic dots represent individual defendants. Rows are features, with more important features shown at the top of the plot. Color represents the value of that feature for that defendant. Green is "high," so in the first row, green dots are defendants with a high priors_count (number of prior arrests). The horizontal position is SHAP value, or the contribution of that feature for that defendant to the model's prediction. It is in units of the model's output, which is here probability. The right end of the first row shows that the green defendants' high priors_count caused the model to increase its predicted probability of recidivism by about 25-33%. SHAP 'allocates' that much of the final prediction to priors_count.

Being African-American, it seems, has only a modest effect on the model's predictions — plus a few percent (green dots). Incidentally, so does being Caucasian, to a smaller degree, according to the model. This seems at odds with the observed "unfairness" according to equalized odds. One possible interpretation is that the model predicts recidivism significantly more often for African-American defendants not because of their race per se, but because on average this group has a higher priors count.

This could quickly spin into a conversation on social justice, so a word of caution here. First, this figure does not claim causal links. It is not correct to read that being a certain race causes more recidivism. It also has nothing to say about why priors count and race may or may not be correlated.

Perhaps more importantly, note that this model purports to predict recidivism, or committing a crime again. The data it learns on really tells us whether the person was arrested again. These are not the same things. We might wonder whether race has a confounding effect on the chance of being arrested as well. It doesn't even require a direct link; are neighborhoods where one race is overrepresented policed more than others?

In this case or others, some might view this SHAP plot as reasonable evidence of fairness. SHAP quantified the effects of demographics and they were quite small. What if that weren't sufficient, and equalized odds is deemed important to achieve? For that, it's necessary to force the model to optimize for it.

Attempt 3: Equalized odds with Fairlearn

Model fitting processes try to optimize a chosen metric. Above, Hyperopt and XGBoost were used to choose a model with the highest accuracy. If equalized odds is also important, the modeling process will have to optimize for that too. Fortunately, Fairlearn has several functions that bolt onto standard modeling tools, changing how they fit, to attempt to steer a model's fairness metrics toward the desired direction.

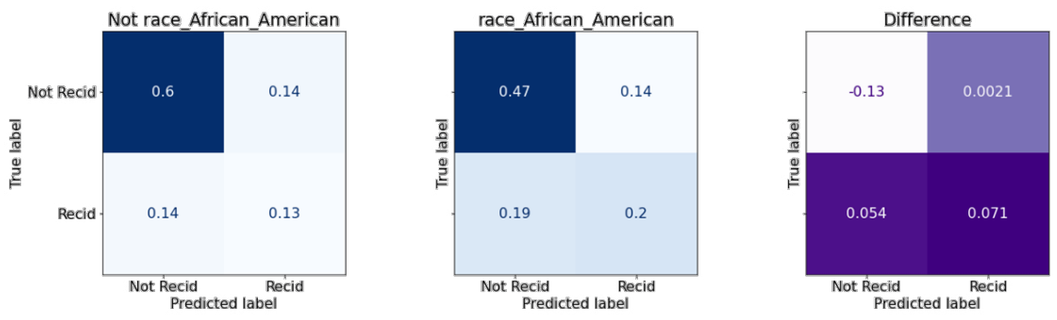
For the practitioner, Fairlearn has a few options. One set of approaches learns to reweight the inputs to achieve this goal. However, this example will try ThresholdOptimizer, which learns optimal thresholds for declaring a prediction "positive" or "negative." That is, when a model produces probabilities as its output, it's common to declare the prediction "positive" when the probability exceeds 0.5. The threshold need not be 0.5; in fact, it need not be the same threshold for all inputs. ThresholdOptimizer picks different thresholds for different inputs, and in this example would learn different thresholds for the group of African-American defendants versus other groups.

```
def predict_xgboost_fairlearn(X_train_xgb, y_train_xgb, X_val_xgb, y_
val_xgb, params):
  mlflow.xgboost.autolog(disable=True)
  pos_weight = (len(y_train_xgb) / y_train_xgb.sum()) - 1
  # Can't use early stopping
  model = XGBClassifier(use_label_encoder=False, n_jobs=4, n_
estimators=200,
                        random_state=0, scale_pos_weight=pos_weight,
                        max_depth=int(params['max_depth']),
                        learning_rate=params['learning_rate'],
                        min_child_weight=params['min_child_weight'],
                        subsample=params['subsample'],
                        colsample_bytree=params['colsample_bytree'])
  # Wrap in an optimizer that prioritizes equalized odds while
  # trying to maintain accuracy
  optimizer = ThresholdOptimizer(estimator=model,
constraints="equalized_odds",
    objective="accuracy_score", predict_method="predict_proba")
  sensitive_cols = ["race_African_American"]
  optimizer.fit(X_train_xgb.drop(sensitive_cols, axis=1), y_train_xgb,
    sensitive_features=X_train_xgb[sensitive_cols])
  wrapper = FairlearnThresholdWrapper(optimizer, sensitive_cols)
  mlflow.pyfunc.log_model("model", python_model=wrapper)
  return wrapper.predict(None, X_val_xgb)
```

All along, the accompanying code has been using MLflow to track models and compare them. Note that MLflow is flexible and can log "custom" models. That's important here, as our model is not simply an XGBoost booster, but a custom combination of several libraries. We can log, evaluate and deploy the model all the same:

```
class FairlearnThresholdWrapper(mlflow.pyfunc.PythonModel):
  def __init__(self, threshold_optimizer, sensitive_cols):
    self.threshold_optimizer = threshold_optimizer
    self.sensitive_cols = sensitive_cols

  def predict(self, context, data):
    return self.threshold_optimizer.predict(
      data.drop(self.sensitive_cols, axis=1),
      sensitive_features=data[self.sensitive_cols], random_state=42)
```

Trying to optimize for equalized odds comes at a cost. It's not entirely possible to optimize for it and accuracy at the same time. They will trade off. Fairlearn lets us specify that, if all else is equal, more accuracy is better where equalized odds is achieved. The result?



Confusion matrices for African-American defendants vs rest, and difference, for Fairlearn-adjusted model

is_recid	race_African_American	accuracy	recall	false positive rate	false negative rate	selection rate
0	0	0.808696	0.0	0.191304	0.0	0.191304
	1	0.766176	0.0	0.233824	0.0	0.233824
1	0	0.480836	0.480836	0.0	0.519164	0.480836
	1	0.508121	0.508121	0.0	0.491879	0.508121

Classifier metrics, broken down by race and ground truth, for Fairlearn-adjusted model.

Not surprisingly, the TPR and FPR are much closer now. They're only about 3-4% different. In fact, Fairlearn has a configurable tolerance for mismatch in equalized odds that defaults to 5% — it's not necessary to demand perfect parity, especially when it may cost accuracy.

Shrinking the TPR/FPR gap did cost something. Generally speaking, the model is more reluctant to classify defendants as "positive" (will recidivate). That means fewer false positives, but more false negatives. Accuracy is significantly lower for those that did reoffend. Whether this is worth the trade-off is a subjective and fraught question, but tools like this can help achieve a desired point in the trade-off. Ultimately, the trade-offs and their tolerance are highly dependent on the application and the ML practitioner thinking about the problem.

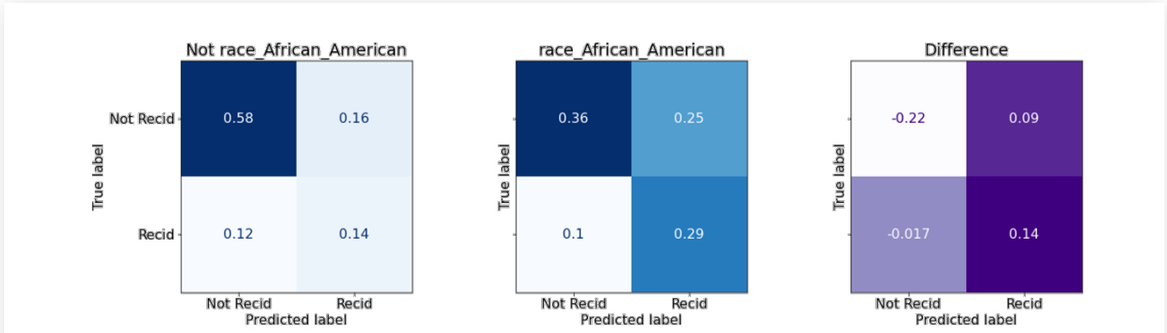
This ThresholdOptimizer approach has a problem that returns again to the question of what fairness is. Quite literally, this model has a different bar for people of different races. Some might say that is objectively unfair. Others may argue this is merely counteracting other systematic unfairness, and worth it. Whatever one thinks in this case, the answer could be entirely different in another context!

In cases where this approach may seem unprincipled, there is another possibly more moderate option. Why not simply quantify and then subtract out the demographic effects that were isolated with SHAP?

Attempt 4: Mitigate with SHAP values

Above, SHAP tried to allocate the prediction to each individual input for a prediction. For African-American defendants, SHAP indicated that the model increased its predicted probability of recidivism by 1-2%. Can we leverage this information to remove the effect? The SHAP Explainer can calculate the effect of features like age, race and gender, and sum them, and simply subtract them from the prediction. This can be bottled up in a simple MLflow custom model, so that this idea can be managed and deployed like any other model:

```
class SHAPCorrectedXGBoostModel(PythonModel):

  def __init__(self, booster, explainer, model_columns):
    self.booster = booster
    self.explainer = explainer
    self.sum_shap_indices =
      [model_columns.tolist().index(c) for c in model_columns if \
       c.startswith("sex") or c.startswith("race")]

  def predict(self, context, model_input):
    predicted_probs = self.booster.predict_proba(model_input)[:,1]
    shap_values = self.explainer.shap_values(model_input)
    corrected_probs = predicted_probs -
      shap_values[:,self.sum_shap_indices].sum(axis=1)
    return pd.DataFrame((corrected_probs >= 0.5).astype('int32'))
```



Confusion matrices for African-American defendants vs rest, and difference, for SHAP-adjusted model.

is_recid	race_African_American	accuracy	recall	false positive rate	false negative rate	selection rate
0	0	0.783851	0.0	0.216149	0.0	0.216149
	1	0.592647	0.0	0.407353	0.0	0.407353
1	0	0.547038	0.547038	0.0	0.452962	0.547038
	1	0.737819	0.737819	0.0	0.262181	0.737819

Classifier metrics, broken down by race and ground truth, for SHAP-adjusted model.

It's not so different from the first two results. The TPR and FPR gap remains around 19%. The estimated effect of factors like race has been explicitly removed from the model. There is virtually no direct influence of demographics on the result now. Differences in other factors, like priors_count, across demographic groups might explain the persistence of the disparity.

Is that OK? If it were true that African-American defendants on average had higher priors count, and this legitimately predicted higher recidivism, is it reasonable to just accept that this means African-American defendants are less likely to be released? This is not a data science question, but a question of policy and even ethics.

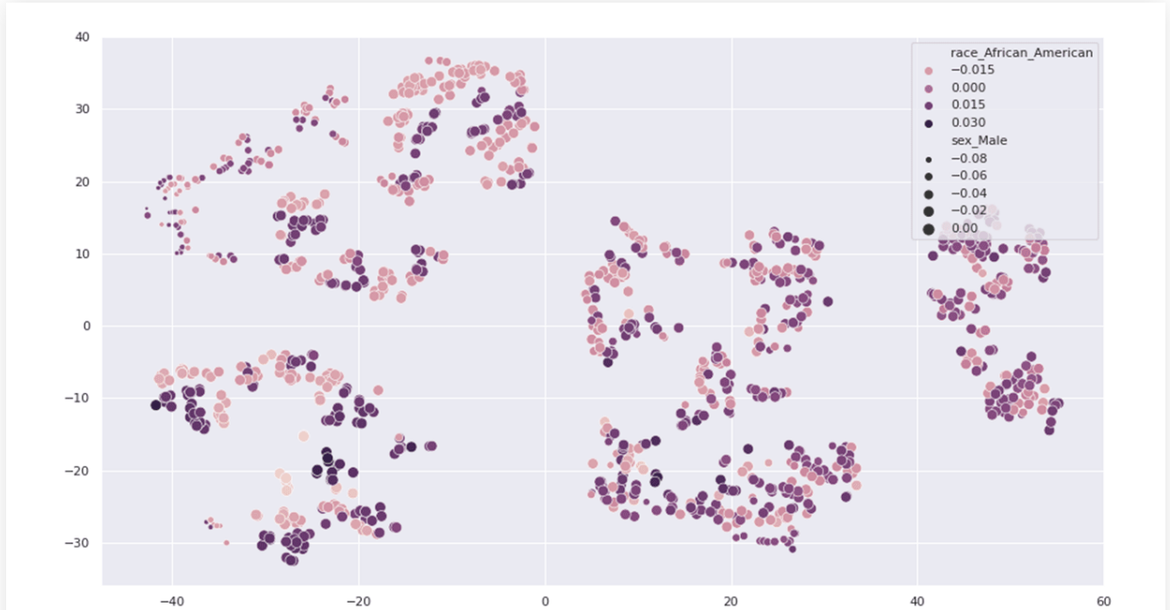
Finding anomalous cases with SHAP

Time for a bonus round. SHAP's output and plots are interesting by themselves, but they also transform the model inputs into a different space, in a way. In this example, demographics and arrest history turn into units of "contribution predicted probability of recidivism," an alternate lens on the defendants in light of the model's prediction.

Because the SHAP values are in the same units (and useful units at that), they lead to a representation of defendants that is naturally clusterable. This approach differs from clustering defendants based on their attributes, and instead groups defendants according to how the model sees them in the context of recidivism. Clustering is a useful tool for anomaly detection and visualization. It's possible to cluster defendants by SHAP values to get a sense of whether a few defendants are being treated unusually by the model. After all, it is one thing to conclude that on average the model's predictions are unbiased. It's not the same as concluding that every prediction is fair!

It's possible that, in some instances, the model is forced to lean heavily on race and gender in order to correctly predict the outcome for a defendant. If an investigator were looking for individual instances of unfairness, it would be reasonable to examine cases where the model tried to predict "will recidivate" for a defendant but could only come up with that answer by allocating a lot of probability to demographic features. These might be instances where the defendant was treated unfairly, possibly on account of race and other factors.

A fuller treatment of this idea deserves its own post, but as an example, here is a clustering of the defendants by SHAP values, using t-SNE to generate a clustering that prioritizes a clear separation in groups:



t-SNE "clustering" of defendants by their SHAP values.

There is some kind of group at the bottom left for which race_African_American caused a different predicted probability of recidivism , either notably on the high or low side (dark or light points). These are not African-American defendants or not; these are defendants where being or not-being African-American had a wider range of effects. There is a group near the top left for whom gender (sex_Male) had a noticeable negative effect on probability; these are not males, these almost certainly represent female defendants.

This plot takes a moment to even understand, and does not necessarily reveal anything significant in this particular case. It is an example of the kind of exploration that is possible when looking for patterns of bias by looking for patterns of significant effects attributed to demographics.

Conclusion

Fairness, ethics and bias in machine learning are an important topic. What "bias" means and where it comes from depends heavily on the machine learning problem. Machine learning models can help detect and remediate bias. Open source tools like Fairlearn and SHAP not only turn models into tools for data analysis, but offer means of counteracting bias in model predictions. These techniques can easily be applied to standard open source model training and management frameworks like XGBoost and MLflow. Try them on your model!

CHAPTER 5:

Real-Time Point-of-Sale Analytics

By **Bryan Smith** and **Rob Saker**
September 9, 2021

Disruptions in the supply chain — from reduced product supply and diminished warehouse capacity — coupled with rapidly shifting consumer expectations for seamless omnichannel experiences are driving retailers to rethink how they use data to manage their operations. Prior to the pandemic, 71% of retailers named lack of real-time visibility into inventory as a top obstacle to achieving their omnichannel goals. The pandemic only increased demand for integrated online and in-store experiences, placing even more pressure on retailers to present accurate product availability and manage order changes on the fly. Better access to real-time information is the key to meeting consumer demands in the new normal.

In this blog, we'll address the need for real-time data in retail, and how to overcome the challenges of moving real-time streaming of point-of-sale data at scale with a data lakehouse. To learn more, check out our Solution Accelerator for Real-Time Point-of-Sale Analytics.

The point-of-sale system

The point-of-sale (POS) system has long been the central piece of in-store infrastructure, recording the exchange of goods and services between retailer and customer. To sustain this exchange, the POS typically tracks product inventories and facilitates replenishment as unit counts dip below critical levels. The importance of the POS to in-store operations cannot be overstated, and as the system of record for sales and inventory operations, access to its data is of key interest to business analysts.

Historically, limited connectivity between individual stores and corporate offices meant the POS system (not just its terminal interfaces) physically resided within the store. During off-peak hours, these systems might phone home to transmit summary data, which when consolidated in a data warehouse, provide a day-old view of retail operations performance that grows increasingly stale until the start of the next night's cycle.

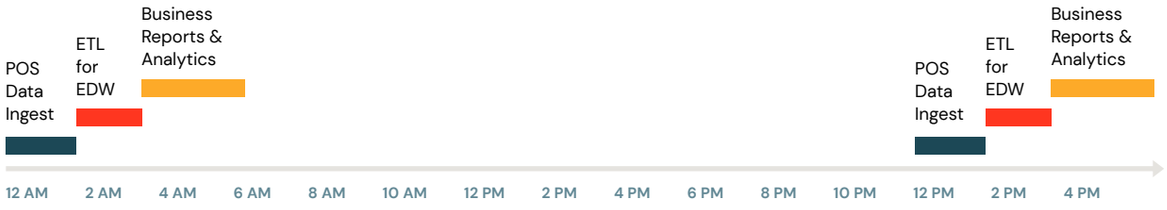


Figure 1. Inventory availability with traditional, batch-oriented ETL patterns.

Modern connectivity improvements have enabled more retailers to move to a centralized, cloud-based POS system, while many others are developing near real-time integrations between in-store systems and the corporate back office. Near real-time availability of information means that retailers can continuously update their estimates of item availability. No longer is the business managing operations against their knowledge of inventory states as they were a day prior but instead is taking actions based on their knowledge of inventory states as they are now.

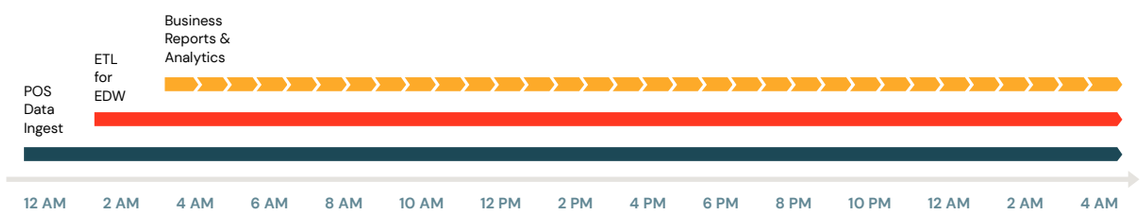


Figure 2. Inventory availability with streaming ETL patterns.

Near real-time insights

As impactful as near real-time insights into store activity are, the transition from nightly processes to continuous streaming of information brings particular challenges, not only for the data engineer who must design a different kind of data processing workflow, but also for the information consumer. In this post, we share some lessons learned from customers who've recently embarked on this journey and examine how key patterns and capabilities available through the lakehouse pattern can enable success.

LESSON 1:

Carefully consider scope

POS systems are not often limited to just sales and inventory management. Instead, they can provide a sprawling range of functionality from payment processing, store credit management, billing and order placement, to loyalty program management, employee scheduling, time-tracking and even payroll, making them a veritable Swiss Army knife of in-store functionality.

As a result, the data housed within the POS is typically spread across a large and complex database structure. If lucky, the POS solution makes a data access layer available, which makes this data accessible through more easily interpreted structures. But if not, the data engineer must sort through what can be an opaque set of tables to determine what is valuable and what is not.

Regardless of how the data is exposed, the classic guidance holds true: identify a compelling business justification for your solution and use that to limit the scope of the information assets you initially consume. Such a justification often comes from a strong business sponsor, who is tasked with addressing a specific business challenge and sees the availability of more timely information as critical to their success.

To illustrate this, consider a key challenge for many retail organizations today: the enablement of omnichannel solutions. Such solutions, which enable buy-online, pickup in-store (BOPIS) and cross-store transactions, depend on reasonably accurate information about store inventory. If we were to limit our initial scope to this one need, the information requirements for our monitoring and analytics system become dramatically reduced. Once a real-time inventory solution is delivered and value recognized by the business, we can expand our scope to consider other needs, such as promotions monitoring and fraud detection, expanding the breadth of information assets leveraged with each iteration.

LESSON 2:

Align transmission with patterns of data generation and time sensitivities

Different processes generate data differently within the POS. Sales transactions are likely to leave a trail of new records appended to relevant tables. Returns may follow multiple paths triggering updates to past sales records, the insertion of new, reversing sales records and/or the insertion of new information in returns-specific structures. Vendor documentation, tribal knowledge and even some independent investigative work may be required to uncover exactly how and where event-specific information lands within the POS.

Understanding these patterns can help build a data transmission strategy for specific kinds of information. Higher frequency, finer-grained, insert-oriented patterns may be ideally suited for continuous streaming. Less frequent, larger-scale events may best align with batch-oriented, bulk data styles of transmission. But if these modes of data transmission represent two ends of a spectrum, you are likely to find most events captured by the POS fall somewhere in between.

The beauty of the data lakehouse approach to data architecture is that multiple modes of data transmission can be employed in parallel. For data naturally aligned with the continuous transmission, streaming may be employed. For data better aligned with bulk transmission, batch processes may be used. And for those data falling in the middle, you can focus on the timeliness of the data required for decision-making and allow that to dictate the path forward. All of these modes can be tackled with a consistent approach to ETL implementation, a challenge that thwarted many earlier implementations of what were frequently referred to as Lambda architectures.

LESSON 3:

Land the data in stages

Data arrives from the in-store POS systems with different frequencies, formats, and expectations for timely availability. Leveraging the Bronze, Silver and Gold design pattern popular within lakehouses, you can separate initial cleansing, reformatting, and persistence of the data from the more complex transformations required for specific business-aligned deliverables.

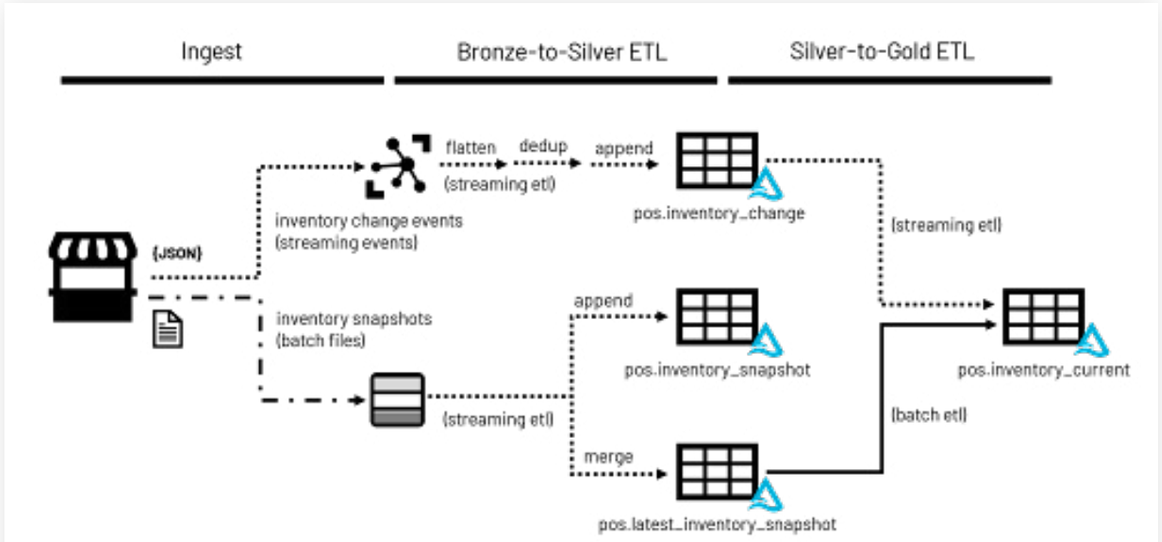


Figure 3. A data lakehouse architecture for the calculation of current inventory leveraging the Bronze, Silver and Gold pattern of data persistence.

LESSON 4:

Manage expectations

The move to near real-time analytics requires an organizational shift. Gartner describes through their Streaming Analytics Maturity model within which analysis of streaming data becomes integrated into the fabric of day-to-day operations. This does not happen overnight.

Instead, data engineers need time to recognize the challenges inherent to streaming delivery from physical store locations into a centralized, cloud-based back office. Improvements in connectivity and system reliability coupled with increasingly more robust ETL workflows land data with greater timeliness, reliability and consistency. This often entails enhancing partnerships with systems engineers and application developers to support a level of integration not typically present in the days of batch-only ETL workflows.

Business analysts will need to become familiar with the inherent noisiness of data being updated continuously. They will need to relearn how to perform diagnostic and validation work on a data set, such as when a query that ran seconds prior now returns a slightly different result. They must gain a deeper awareness of the problems in the data which are often hidden when presented in daily aggregates. All of this will require adjustments both to their analysis and their response to detected signals in their results.

All of this takes place in just the first few stages of maturation. In later stages, the organization's ability to detect meaningful signals within the stream may lead to more automated sense and response capabilities. Here, the highest levels of value in the data streams are unlocked. But monitoring and governance must be put into place and proven before the business will entrust its operations to these technologies.

Implementing POS streaming

To illustrate how the lakehouse architecture can be applied to POS data, we've developed a demonstration workflow within which we calculate a near real-time inventory. In it, we envision two separate POS systems transmitting inventory-relevant information associated with sales, restocks and shrinkage data along with buy-online, pickup in-store (BOPIS) transactions (initiated in one system and fulfilled in the other) as part of a streaming inventory change feed. Periodic (snapshot) counts of product units on-shelf are captured by the POS and transmitted in bulk. These data are simulated for a one-month period and played back at 10x speed for greater visibility into inventory changes.

The ETL processes (as pictured in figure 3 above) represent a mixture of streaming and batch techniques. A two-staged approach with minimally transformed data captured in Delta tables representing our Silver layer separates our initial, more technically aligned ETL approach with the more business-aligned approach required for current inventory calculations. The second stage has been implemented using traditional Structured Streaming capabilities, something we may revisit with the new Delta Live Tables functionality as it makes its way toward general availability.

The demonstration makes use of Azure IOT Hubs and Azure Storage for data ingestion but would work similarly on the AWS and GCP clouds with appropriate technology substitutions. Further details about the setup of the environment along with the replayable ETL logic can be found in the following notebooks:

Get the Solution Accelerator

CHAPTER 6:

Design Patterns for Real-Time Insights in Financial Services

Streaming foundations for personalization

By **Ricardo Portilla**
May 20, 2022

Personalization is a competitive differentiator for most every financial services institution (FSIs, for short), from banking to insurance and now investment management platforms. While every FSI wants to offer intelligent and real-time personalization to customers, the foundations are often glossed over or implemented with incomplete platforms, leading to stale insights, long time-to-market, and loss of productivity due to the need to glue streaming, AI, and reporting services together.

This blog will demonstrate how to lay a robust foundation for real-time insights for financial services use cases with the Databricks Lakehouse Platform, from OLTP database change data capture (CDC) data to reporting dashboard. Databricks has long supported streaming, which is native to the platform. The recent release of Delta Live Tables (DLT) has made streaming even simpler and more powerful with new CDC capabilities. We have covered a guide to CDC using DLT in a recent comprehensive blog. In this blog, we focus on streaming for FSIs and show how these capabilities help streamline new product differentiators and internal insights for FSIs.

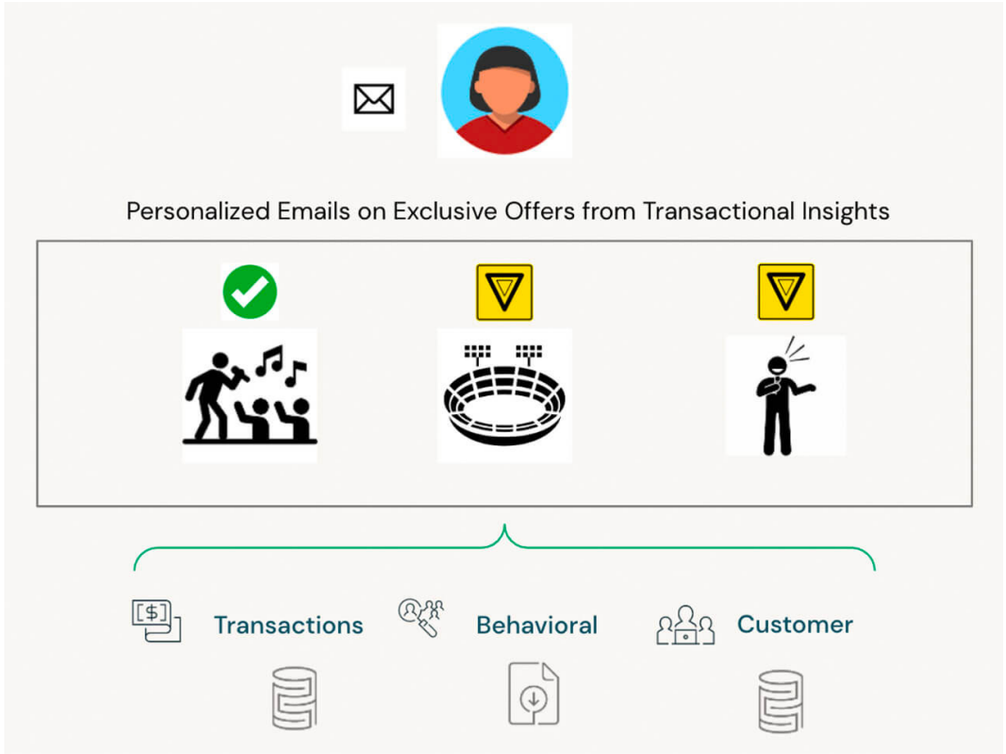
Why streaming ingestion is critical

Before getting into technical details, let's discuss why Databricks is best for personalization use cases, and specifically why implementing streaming should be the first step. Many Databricks customers who are implementing customer 360 projects or full-funnel marketing strategies typically have the base requirements below. Note the temporal (time-related) data flow.

FSI data flow and requirements

1. User app saves and updates data such as clickstream, user updates, and geolocation data — requires operational databases
2. Third-party behavioral data is delivered incrementally via object storage or is available in a database in a cloud account — requires streaming capabilities to incrementally add/update/delete new data in single source of truth for analytics
3. FSI has an automated process to export all database data including user updates, clickstream, and user behavioral data into data lake — requires change data capture (CDC) ingestion and processing tool, as well as support for semi-structured and unstructured data
4. Data engineering teams run automated data quality checks and ensure the data is fresh — requires data quality tool and native streaming
5. Data science teams use data for next best action or other predictive analytics — requires native ML capabilities
6. Analytics engineers and data analysts will materialize data models and use data for reporting — requires dashboard integration and native visualization

The core requirements here are data freshness for reporting, data quality to maintain integrity, CDC ingestion, and ML-ready data stores. In Databricks, these map directly to Delta Live Tables (notably Auto Loader, Expectations, and DLT's SCD Type I API), Databricks SQL, and Feature Store. Since reporting and AI-driven insights depend upon a steady flow of high-quality data, streaming is the logical first step to master.



Consider, for example, a retail bank wanting to use digital marketing to attract more customers and improve brand loyalty. It is possible to identify key trends in customer buying patterns and send personalized communication with exclusive product offers in real time tailored to the exact customer needs and wants. This is a simple, but an invaluable use case that's only possible with streaming and change data capture (CDC) — both capabilities required to capture changes in consumer behavior and risk profiles.

For a sneak peak at the types of data we handle in our reference DLT pipeline, see the samples below. Notice the temporal nature of the data — all banking or lending systems have time-ordered transactional data, and a trusted data source means having to incorporate late-arriving and out-of-order data. The core data sets shown include transactions from, say, a checking account (figure 2), customer updates, but also behavioral data (figure 3) which may be tracked from transactions or upstream third-party data.

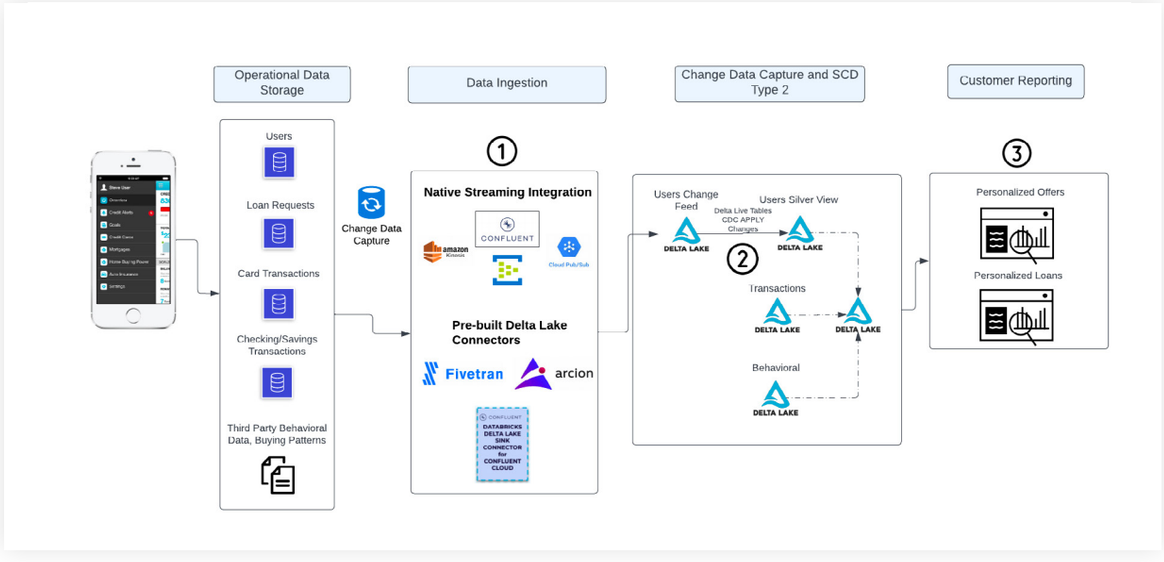
	customer_id	scheduled_payment	txn_amount	debit_or_credit	updt_ts	initial_balance
1	3	1	3	0	2022-05-02T03:29:54.337+0000	10000
2	3	1	3	1	2022-05-02T03:29:56.011+0000	10000
3	3	1	13	1	2022-05-03T03:29:57.201+0000	10000
4	3	1	23	1	2022-05-04T03:29:58.370+0000	10000
5	3	1	33	1	2022-05-05T03:29:59.532+0000	10000
6	3	1	43	1	2022-05-06T03:30:00.692+0000	10000
7	3	1	53	1	2022-05-07T03:30:01.854+0000	10000

	customer_id	event_ts	home_mortgage_purchased_ts	number_of_comedy_shows	number_of_concert_tickets	number_of_sport_events
1	3	2022-05-05 03:11:27	null	1	6	2
2	3	2022-05-15 03:11:38	null	2	7	3
3	3	2022-05-25 03:11:49	null	3	8	3
4	3	2022-06-04 03:11:59	null	3	8	3
5	3	2022-06-14 03:12:09	null	3	8	3
6	3	2022-06-24 03:12:20	null	3	8	4
7	3	2022-07-04 03:12:30	null	4	9	4
8	3	2022-07-14 03:12:41	null	4	9	4
9	3	2022-07-24 03:12:51	1658632371000	4	9	4

Getting started with streaming

In this section, we will demonstrate a simple end-to-end data flow so that it is clear how to capture continuous changes from transactional databases and store them in a lakehouse using Databricks streaming capabilities.

Our starting point are records mocked up from standard formats from transactional databases. The diagram below provides an end-to-end picture of how data might flow through an FSI's infrastructure, including the many varieties of data which ultimately land in Delta Lake, are cleaned, and summarized and served in a dashboard. There are three main processes mentioned in this diagram, and in the next section we'll break down some prescriptive options for each one.



End-to-end architecture of how data might flow through an FSI's infrastructure, illustrating the myriad data which ultimately land in Delta Lake, is cleaned, and summarized and served in a dashboard.

PROCESS #1

Data ingestion

Native Structured Streaming ingestion option

With the proliferation of data that customers provide via banking and insurance apps, FSIs have been forced to devise strategies around collecting this data for downstream teams to consume for various use cases. One of the most basic decisions these companies face is how to capture all changes from app services which customers have in production: from users, to policies, to loan apps and credit card transactions. Fundamentally, these apps are backed by transactional data stores, whether it's MySQL databases or more unstructured data residing in NoSQL databases such as MongoDB.

Luckily, there are many open source tools, like Debezium, that can ingest data out of these systems. Alternatively, we see many customers writing their own stateful clients to read in data from transactional stores, and write to a distributed message queue like a managed Kafka cluster. Databricks has tight integration with Kafka, and a direct connection along with a streaming job is the recommended pattern when the data needs to be as fresh as possible. This setup enables near real-time insights to businesses, such as real-time cross-sell recommendations or real-time views of loss (cash rewards effect on balance sheets). The pattern is as follows:

1. Set up CDC tool to write change records to Kafka
2. Set up Kafka sink for Debezium or other CDC tool
3. Parse and process change data capture (CDC) records in Databricks using Delta Live Tables, first landing data directly from Kafka into Bronze tables

CONSIDERATIONS

Pros

- Data arrives continuously with lower latencies, so consumers get results in near real-time without relying on batch updates
- Full control of the streaming logic
- Delta Live Tables abstracts cluster management away for the Bronze layer, while enabling users to efficiently manage resources by offering auto-scaling
- Delta Live Tables provides full data lineage, and seamless data quality monitoring for the landing into Bronze layer

Cons

- Directly reading from Kafka requires some parsing code when landing into the Bronze staging layer
- This relies on extra third-party CDC tools to extract data from databases and feed into a message store rather than using a tool that establishes a direct connection

Partner ingestion option

The second option for getting data into a dashboard for continuous insights is Databricks Partner Connect, the broad network of data ingestion partners that simplify data ingestion into Databricks. For this example, we'll ingest data via a Delta connector created by Confluent, a robust managed Kafka offering which integrates closely with Databricks. Other popular tools like Fivetran and Arcion have hundreds of connectors to core transactional systems.

Both options abstract away much of the core logic for reading raw data and landing it in Delta Lake through the use of COPY INTO commands. In this pattern, the following steps are performed:

1. Set up CDC tool to write change records to Kafka (same as before)
2. Set up the Databricks Delta Lake Sink Connector for Confluent Cloud and hook this up to the relevant topic

The main difference between this option and the native streaming option is the use of Confluent's Delta Lake Sink Connector. See the trade-offs for understanding which pattern to select.

CONSIDERATIONS

Pros

- Low-code CDC through partner tools supports high-speed replication of data from on-prem legacy sources, databases, and mainframes (e.g., Fivetran, Arcion, and others with direct connection to databases)
- Low-code data ingestion for data platform teams familiar with streaming partners (such as Confluent Kafka) and preferences to land data into Delta Lake without the use of Apache Spark™
- Centralized management of topics and sink connectors in Confluent Cloud (similarly with Fivetran)

Cons

- Less control over data transformation and payload parsing with Spark and third-party libraries in the initial ETL stages
- Databricks cluster configuration required for the connector

File-based ingestion

Many data vendors — including mobile telematics providers, tick data providers, and internal data producers — may deliver files to clients. To best handle incremental file ingestion, Databricks introduced Auto Loader, a simple, automated streaming tool which tracks state for incremental data such as intraday feeds for trip data, trade-and-quote (TAQ) data, or even alternative data sets such as sales receipts to predict earnings forecasts.

Auto Loader is now available to be used in the Delta Live Tables pipelines, enabling you to easily consume hundreds of data feeds without having to configure lower level details. Auto Loader can scale massively, handling millions of files per day with ease. Moreover, it is simple to use within the context of Delta Live Tables APIs (see SQL example below):

```
CREATE INCREMENTAL LIVE TABLE customers
AS SELECT * FROM cloud_files("/databricks-datasets/retail-org/
customers/", "csv", map("delimiter", "\t"))
```

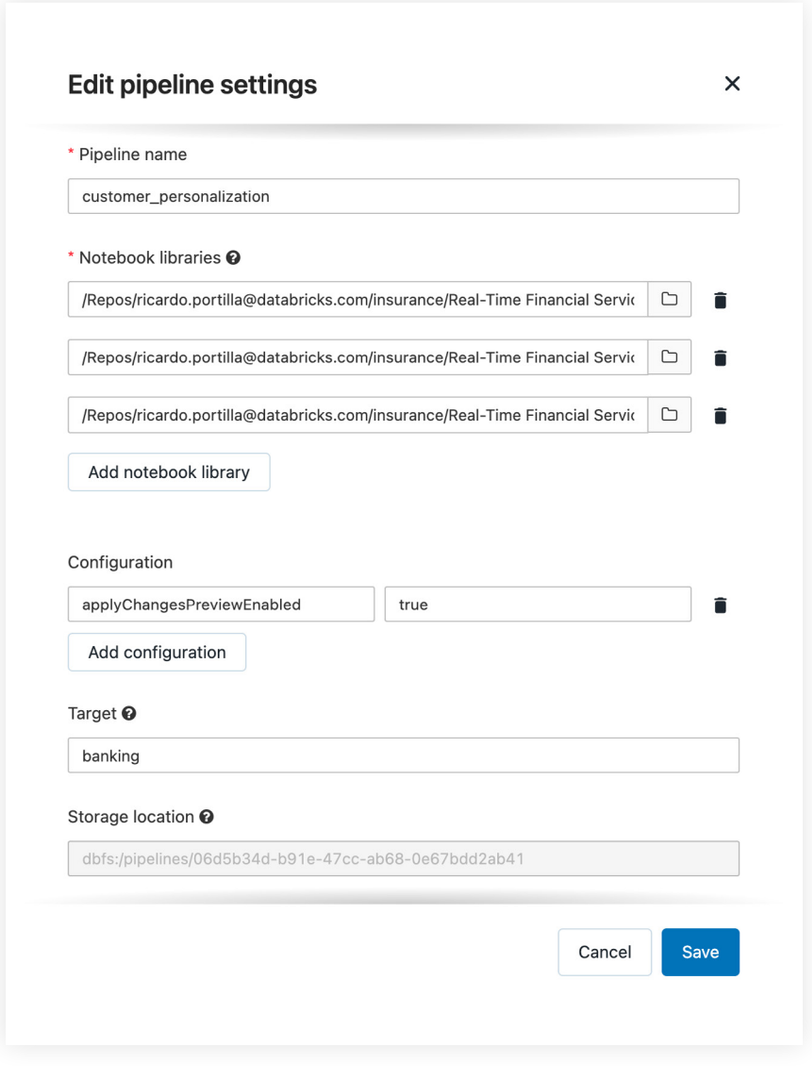
PROCESS #2

Change data capture

Change data capture solutions are necessary since they ultimately save changes from core systems to a centralized data store without imposing additional stress on transactional databases. With abundant streams of digital data, capturing changes to customers' behavior is paramount to personalizing the banking or claims experience.

From a technical perspective, we are using Debezium as our highlighted CDC tool. Of importance to note is the sequence key, which is Debezium's `datetime_updated` epoch time, which Delta Live Tables (DLT) uses to sort through records to find the latest change and apply to the target table in real time. Again, because a user journey has an important temporal component, the APPLY CHANGES INTO functionality from DLT is an elegant solution since it abstracts the complexity of having to update the user state — DLT simply updates the state in near real-time with a one-line command in SQL or Python (say, updating customer preferences in real time from 3 concert events attended to 5, signifying an opportunity for a personalized offer).

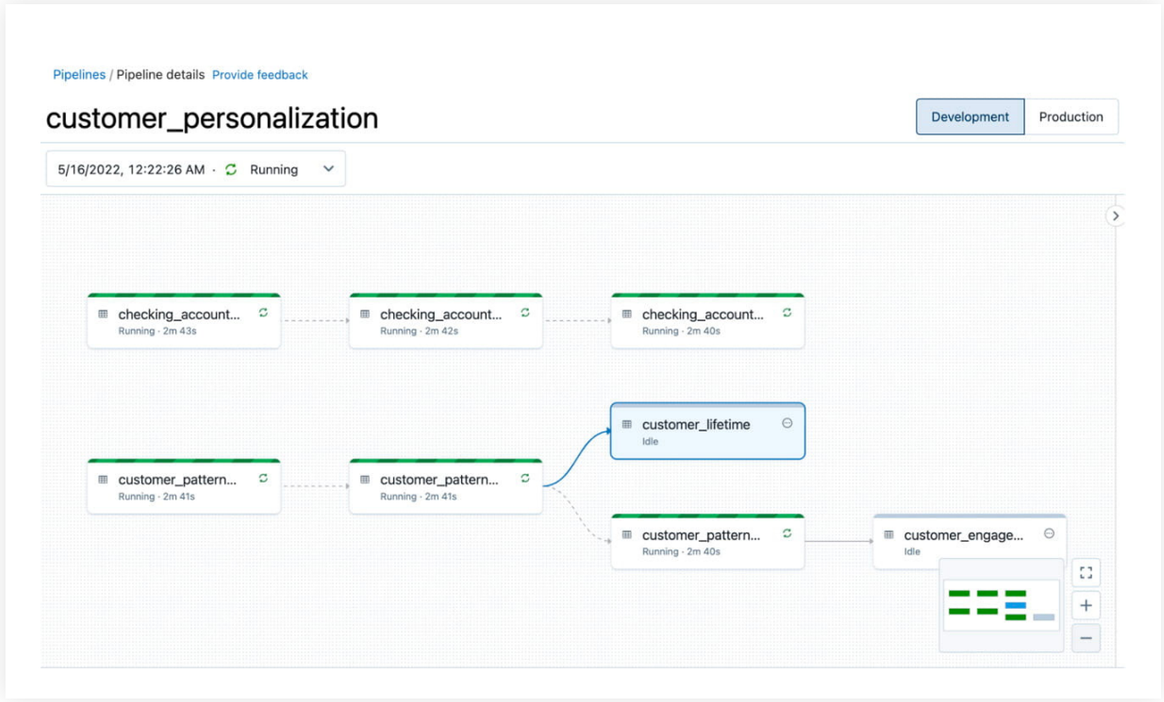
In the code below, we are using SQL streaming functionality, which allows us to specify a continuous stream landing into a table to which we apply changes to see the latest customer or aggregate update. See the full pipeline configuration below. The full code is available here.



Here are some basic terms to note:

- The `STREAMING` keyword indicates a table (like customer transactions) which accepts incremental insert/updates/deletes from a streaming source (e.g., Kafka)
- The `LIVE` keyword indicates the data set is internal, meaning it has already been saved using the DLT APIs and comes with all the auto-managed capabilities (including auto-compaction, cluster management, and pipeline configurations) that DLT offers
- `APPLY CHANGES INTO` is the elegant CDC API that DLT offers, handling out-of-order and late-arriving data by maintaining state internally — without users having to write extra code or SQL commands.

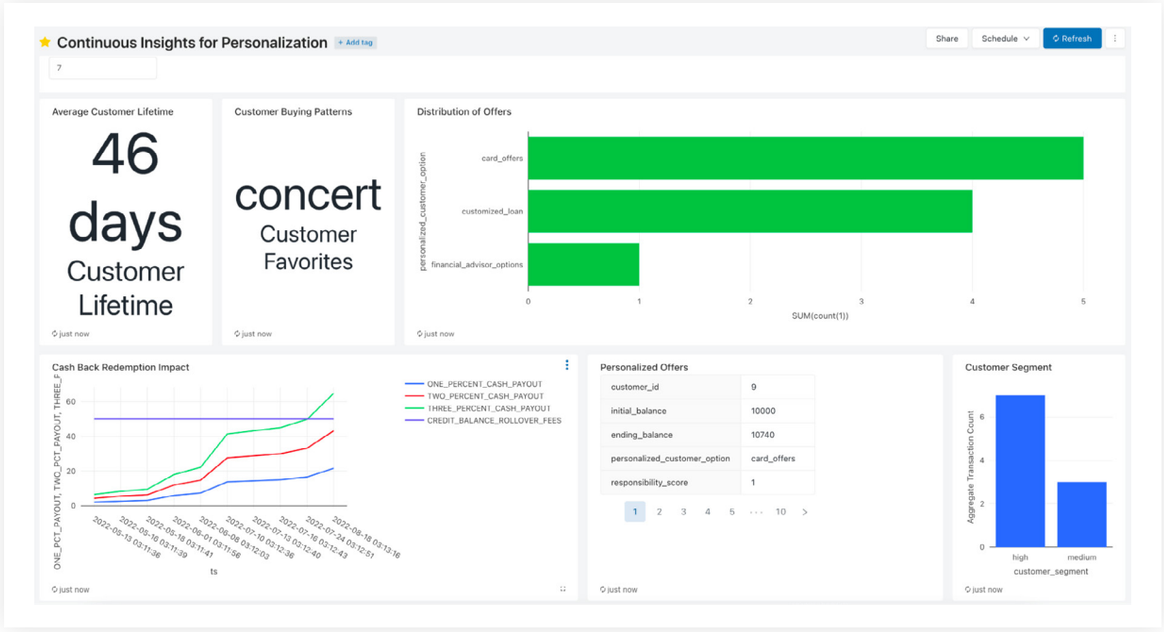
```
CREATE STREAMING LIVE TABLE customer_patterns_silver_copy
(
 CONSTRAINT customer_id EXPECT (customer_id IS NOT NULL) ON VIOLATION
DROP ROW
)
TBLPROPERTIES ("quality" = "silver")
COMMENT "Cleansed Bronze customer view (i.e. what will become Silver)"
AS SELECT json.payload.after.* , json.payload.op
FROM stream(live.customer_patterns_bronze);

APPLY CHANGES INTO live.customer_patterns_silver
FROM stream(live.customer_patterns_silver_copy)
 KEYS (customer_id)
 APPLY AS DELETE WHEN op = "d"
 SEQUENCE BY datetime_updated;
```



PROCESS #3

Summarizing customer preferences and simple offers

To cap off the simple ingestion pipeline above, we now highlight a Databricks SQL dashboard to show what types of features and insights are possible with the lakehouse. All of the metrics, segments, and offers seen below are produced from the real-time data feeds mocked up for this insights pipeline. These can be scheduled to refresh every minute, and more importantly, the data is fresh and ML-ready. Metrics to note are customer lifetime, prescriptive offers based on a customer's account history and purchasing patterns, and cash-back losses and break-even thresholds. Simple reporting on real-time data can highlight key metrics that will inform how to release a specific product, such as cash-back offers. Finally, reporting dashboards (Databricks or BI partners such as Power BI or Tableau) can surface these insights; when AI insights are available, they can easily be added to such a dashboard since the underlying data is centralized in one lakehouse.



Databricks SQL dashboard showing how streaming hydrates the lakehouse and produces actionable guidance on offer losses, opportunities for personalized offers to customers, and customer favorites for new products.

Conclusion

This blog highlights multiple facets of the data ingestion process, which is important to support various personalization use cases in financial services. More importantly, Databricks supports near real-time use cases natively, offering fresh insights and abstracted APIs (Delta Live Tables) for handling change data, supporting both Python and SQL out-of-the-box.

With more banking and insurance providers incorporating more personalized customer experiences, it will be critical to support the model development but more importantly, create a robust foundation for incremental data ingestion. Ultimately, Databricks Lakehouse Platform is second-to-none in that it delivers both streaming and AI-driven personalization at scale to deliver higher CSAT/NPS, lower CAC/churn, and happier and more profitable customers.

To learn more about the Delta Live Tables methods applied in this blog, find all the sample data and code in this GitHub repository.

CHAPTER 7:

Building Patient Cohorts With NLP and Knowledge Graphs

Streaming foundations for personalization

Check out the Solution Accelerator to download the notebooks referred throughout this blog.

By **Amir Kermany, Moritz Steller, David Talby** and **Michael Sanky**

September 30, 2022

Cohort building is an essential part of patient analytics. Defining which patients belong to a cohort, testing the sensitivity of various inclusion and exclusion criteria on sample size, building a control cohort with propensity score matching techniques: These are just some of the processes that healthcare and life sciences researchers live day in and day out, and that's unlikely to change anytime soon. What is changing is the underlying data, the complexity of clinical criteria, and the dynamism demanded by the industry.

While tools exist for building patient cohorts based on structured data from EHR data or claims, their practical utility is limited. More and more, cohort building in healthcare and life sciences requires criteria extracted from unstructured and semi-structured clinical documentation with natural language processing (NLP) pipelines. Making this a reality requires a seamless combination of three technologies:

1. a platform that scales for computationally intensive calculations of massive real-world data sets,
2. an accurate NLP library and healthcare-specific models to extract and relate entities from medical documents, and
3. a knowledge graph toolset, able to represent the relationships between a network of entities.

The latest solution from John Snow Labs and Databricks brings all of this together in the lakehouse.

Optimizing clinical trial protocols

Let's consider one high-impact application of dynamic cohort building.

Recruiting and retaining patients for clinical trials is a long-standing problem that the pandemic has exacerbated. 80% of trials are delayed due to recruitment problems[1], with many sites under-enrolling. Delays in recruitment have huge financial implications in terms of both the cash burn to manage extended trials and the opportunity cost of patent life, not to mention the implications of delaying potentially lifesaving medications.

One of the challenges is that as medications become more specialized, clinical trial protocols are increasingly complex. It is not uncommon to see upwards of 40 different criteria for inclusion and exclusion. The age-old "measure twice, cut once" is exceedingly important here. Let's look at a relatively straightforward example of a protocol for a Phase 3 trial estimated to run for six years: Effect of Evolocumab in Patients at High Cardiovascular Risk Without Prior Myocardial Infarction or Stroke (VESALIUS-CV)[2]:

Ages Eligible for Study: 50 Years to 79 Years (Adult, Older Adult)
Sexes Eligible for Study: All
Accepts Healthy Volunteers: No

Criteria

Inclusion criteria:

1. Age: Adult subjects ≥ 50 (men) or ≥ 55 (women) to < 80 years of age (either sex) and meeting lipid criteria
2. Low-density lipoprotein cholesterol (LDL-C) ≥ 90 mg/dL (≥ 2.3 mmol/L) or non-high-density lipoprotein cholesterol (non-HDL) ≥ 120 mg/dL (≥ 3.1 mmol/L), or apolipoprotein B ≥ 80 mg/dL (≥ 1.56 µmol/L)
3. Evidence of at least one of the following at screening (without prior myocardial infarction or stroke):

   A. Significant coronary artery disease (CAD) B. Significant atherosclerotic cerebrovascular disease C. Significant peripheral arterial disease D. Diabetes mellitus
4. At least 1 high-risk feature

Exclusion criteria

- MI or stroke prior to randomization
- Coronary artery bypass grafting (CABG) < 3 months prior to screening
- Estimated glomerular filtration rate (eGFR) < 15 mL/min/1.73 m²
- Uncontrolled or recurrent ventricular tachycardia in the absence of an implantable-cardioverter defibrillator.
- Atrial fibrillation or atrial flutter not on anticoagulation therapy (vitamin K antagonist, heparin, low molecular weight heparin, fondaparinux,or non-Vitamin K antagonist oral anticoagulant)
- Triglycerides ≥ 500 mg/dL (5.7 mmol/L) measured up to 3 months prior to screening. The most recent results must be used.
- Last measured left-ventricular ejection fraction < 30% or New York Heart Association (NYHA) Functional Class III/IV
- Planned arterial revascularization

In terms of protocol design, the inclusion and exclusion criteria must be targeted enough to have the appropriate clinical sensitivity, and broad enough to facilitate recruitment. Real-world data can provide the guideposts to help forecast patient eligibility and understand the relative impact of various criteria. In the example above, does left-ventricular ejection fraction > 30% limit the population by 10%, 20%? How about eGFR < 15? Does clinical documentation include mentions of atrial flutter that are not diagnosed, which would impact screen failure rates?

Fortunately, these questions can be answered with real-world data and AI.

[1] https://www.biopharmadive.com/spons/decentralized-clinical-trials-are-we-ready-to-make-the-leap/546591
[2] https://clinicaltrials.gov/ct2/show/NCT0387240

Site selection and patient recruitment

Similar challenges exist once a clinical trial protocol has been defined. One of the next decisions for a pharmaceutical company is where to set up sites for the trial. Setting up a site is time consuming, expensive, and often wasteful: Over two-thirds of sites fail to meet their original patient enrollment goals and up to 50% of sites enroll one or no patients in their studies.[3]

This challenge is amplified in newer clinical trials — especially those focusing on rare diseases, or on cancer patients with specific genomic biomarkers. In those cases, a hospital may see only a handful of relevant patients per year, so estimating in advance how many patients are candidates for a trial, and then actually recruiting them when they appear, are both critical to timely success.

The advent of precision health leads to many more clinical trials that target a very small population.[4] This requires the automation scale to find candidate patients to these trials automatically, as well as state-of-the-art NLP capabilities since trial inclusion and exclusion criteria call out more facts that are only available in unstructured text. These facts include genomic variants, social determinants of health, family history, and specific tumor characteristics.

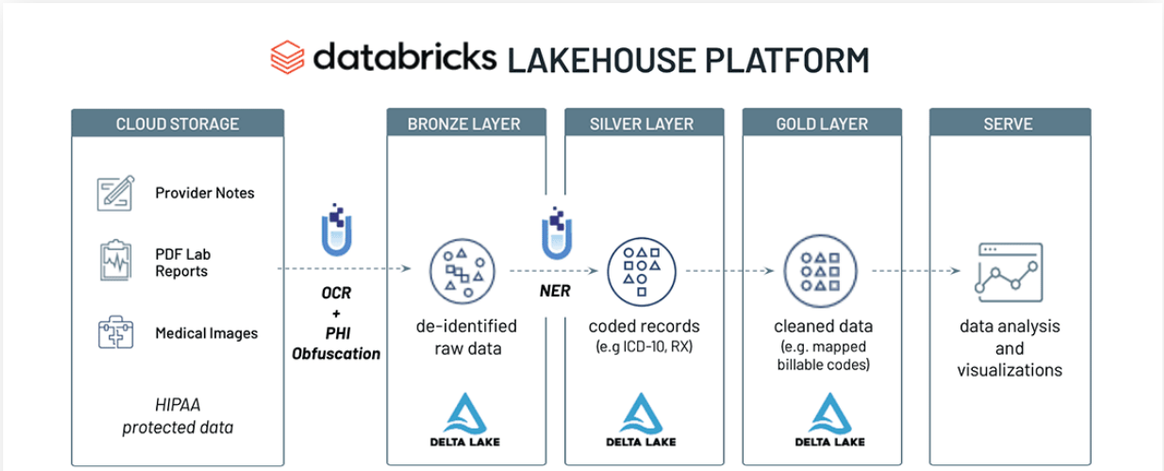
Fortunately, new AI technology is now ready to meet these challenges.

[3] https://www.clinicalleader.com/doc/considerations-for-improving-patient-0001

[4] https://www.webmd.com/cancer/precision-medicine-clinical-trials

Design and run better clinical trials with John Snow Labs and Databricks

First, let's understand the end-to-end solution architecture for patient cohort building with NLP and knowledge graphs:



An end-to-end workflow for automating PHI removal from documents and images

We will build a knowledge graph (KG) using Spark NLP relation extraction models and a graph API. The main point of this solution is to show creating a clinical knowledge graph using Spark NLP pretrained models. For this purpose, we will use pretrained relation extraction and NER models. After creating the knowledge graph, we will query the KG to get some insightful results.

As Building Patient Cohorts With NLP and Knowledge Graphs was part of DAIS 2022, please view its session here: demo.

NLP preprocessing

Overall, there are 965 clinical records in our example data set stored in Delta tables. We read the data and write the records into Bronze Delta tables.

	subject_id	date	text	gender	dateOfBirth
0	19823	2167-02-25	Admission Date: [**2167-2-16**] Discharge Date: [**2167-2-24**]\n\nDate of Birth: [**2...	F	2099-05-05
1	19823	2167-11-27	Admission Date: [**2167-11-27**] Discharge Date: [**2167-12-9**]\n\nDate of Birth: [**...	F	2099-05-05
2	19823	2170-10-12	Admission Date: [**2170-9-19**] Discharge Date: [**2170-10-12**]\n\nDate of Birt...	F	2099-05-05
3	19823	2172-06-22	Admission Date: [**2172-6-13**] Discharge Date: [**2172-6-22**]\n\nDate of Birth...	F	2099-05-05
4	19823	2167-12-07	PATIENT/TEST INFORMATION:\nIndication: Aortic valve disease. Shortness of breath.\nHeight: (in) ...	F	2099-05-05
...	...	...	...	...	...
960	70004	2182-06-14	[**2182-6-14**] 10:45 AM\n MR HEAD W & W/O CONTRAST Clip ...	M	2127-12-06
961	70004	2182-06-25	FDG TUMOR IMAGING (PET-CT) Clip # [**Clip Number (Radiology...	M	2127-12-06
962	70004	2182-08-05	[**2182-8-5**] 11:46 AM\n MR HEAD W & W/O CONTRAST Clip #...	M	2127-12-06
963	70004	2182-08-23	FDG TUMOR IMAGING (PET-CT) Clip # [**Clip Number (Radiology...	M	2127-12-06
964	70004	2182-08-16	[**2182-8-16**] 9:41 AM\n MR [**Name13 (STitle) 279**] W& W/O CONTRAST ...	M	2127-12-06

An example data set of clinical health records stored in a Delta table within Delta Lake.

Extracting from relationships from the text in this DataFrame, Spark NLP for Healthcare applies a posology relation extraction pretrained model that supports the following relations:

DRUG-DOSAGE, DRUG-FREQUENCY, DRUG-ADE (Adverse Drug Events), DRUG-FORM, DRUG-ROUTE, DRUG-DURATION, DRUG-REASON, DRUG=STRENGTH

The model has been validated against the posology data set described in (Magge, Scotch, & Gonzalez-Hernandez, 2018)

http://proceedings.mlr.press/v90/magge18a/magge18a.pdf.

Relation	Recall	Precision	F1	F1 (Magge, Scotch, & Gonzalez-Hernandez, 2018)
DRUG-ADE	0.66	1.00	**0.80**	0.76
DRUG-DOSAGE	0.89	1.00	**0.94**	0.91
DRUG-DURATION	0.75	1.00	**0.85**	0.92
DRUG-FORM	0.88	1.00	**0.94**	0.95*
DRUG-FREQUENCY	0.79	1.00	**0.88**	0.90
DRUG-REASON	0.60	1.00	**0.75**	0.70
DRUG-ROUTE	0.79	1.00	**0.88**	0.95*
DRUG-STRENGTH	0.95	1.00	**0.98**	0.97

*Magge, Scotch, Gonzalez-Hernandez (2018) collapsed DRUG-FORM and DRUG-ROUTE into a single relation.

Within our NLP pipeline, Spark NLP for Healthcare is following the standardized steps of preprocessing (documenter, sentencer, tokenizer), word embeddings, part-of-speech tagger, NER, dependency parsing, and relation extraction. Relation extraction in particular is the most important step in this pipeline as it establishes the connection by bringing relationships to the extracted NER chunks.

The resulting DataFrame includes all relationships accordingly:

	subject_id	date	relation	entity1	entity1_begin	entity1_end	chunk1	entity2	entity2_begin	entity2_end	chunk2	confidence
0	19823	2167-02-25	DRUG-FORM	DRUG	1391	1399	Albuterol	FORM	1414	1423	nebulizers	1.0
1	19823	2167-02-25	DRUG-FORM	DRUG	1405	1412	Atrovent	FORM	1414	1423	nebulizers	1.0
2	19823	2167-02-25	STRENGTH-DRUG	STRENGTH	1539	1543	40 mg	DRUG	1551	1555	Lasix	1.0
3	19823	2167-02-25	ROUTE-DRUG	ROUTE	1548	1549	IV	DRUG	1551	1555	Lasix	1.0
4	19823	2167-02-25	DRUG-STRENGTH	DRUG	2336	2341	Amaryl	STRENGTH	2343	2348	2.0 mg	1.0
...	...	...	...	...	...	...	...	...	...	...	...	...
2777	70004	2182-08-05	ROUTE-DRUG	ROUTE	545	546	IV	DRUG	548	555	contrast	1.0
2778	70004	2182-08-05	DOSAGE-DRUG	DOSAGE	942	946	20 cc	DRUG	951	959	Magnevist	1.0
2779	70004	2182-08-05	DRUG-ROUTE	DRUG	951	959	Magnevist	ROUTE	961	971	intravenous	1.0
2780	70004	2182-08-16	ROUTE-DRUG	ROUTE	475	476	IV	DRUG	478	485	CONTRAST	1.0
2781	70004	2182-08-16	ROUTE-DRUG	ROUTE	832	842	intravenous	DRUG	844	851	contrast	1.0

Spark NLP for Healthcare maps the relationships within the data for analysis.

Within our Lakehouse for Healthcare, this final DataFrame will be written to the Silver layer.

Next, the RxNorm codes are extracted from the prior established data set. Firstly, we use a basic rules-based logic to define and clean up 'entity1' and 'entity2', followed by an SBERT (Sentence BERT) based embedder and BioBERT based resolver to support the transformation to RxNorm codes.

See below for the first three records of the Silver layer data set, the extracted Rx related text, its NER chunks, the applicable RxNorm code, all related codes, RxNorm resolutions and final drug resolution.

rx_text	sent_id	ner_chunk	entity	rxnorm_code	all_codes	resolutions	res	resolution	drug_resolution
Albuterol nebulizers	0	Albuterol nebulizers	NaN	2108226	['2108226' '1154602' '370790' '1154603' '2108233' '2108255' '2108276'\n '745678' '1163444' '2108...	['albuterol Inhalation Solution' 'albuterol Inhalant Product'\n 'albuterol Injectable Solution' ...	['albuterol	albuterol	albuterol
Atrovent nebulizers	0	Atrovent nebulizers	NaN	2108451	['2108451' '1173573' '379767' '1173576' '2463732' '1945043' '1172634'\n '1171309' '363357' '1184...	['ipratropium Inhalation Solution [Atrovent]' 'Atrovent Inhalant Product'\n 'Atrovent Autohaler'...	['ipratropium	ipratropium	ipratropium
Lasix 40 mg	0	Lasix 40 mg	NaN	200809	['200809' '617319' '103919' '1871459' '201286' '2556796' '1927858'\n '1648194' '977916'	['furosemide 40 MG Oral Tablet [Lasix]' 'atorvastatin 40 MG [Lipitor]'\n 'fluvastatin 40 MG Oral...	['furosemide	furosemide	furosemide

The results of transformed data within the Silver layer of Delta Lake.

This result DataFrame is written to the Gold layer.

Lastly, a pretrained named entity recognition deep learning model for clinical terminology (https://nlp.johnsnowlabs.com/2021/08/13/ner_jsl_slim_en.html) is applied to our initial data set to extract generalized entities from our medical text.

The result DataFrame includes the NER chunk and NER label from the unstructured text:

	subject_id	date	sentence_id	chunk	begin	end	ner_label
0	19823	2167-02-25	0	Shortness of breath	178	196	Symptom
1	19823	2167-02-25	0	cough	199	203	Symptom
2	19823	2167-02-25	1	diabetes type II	345	360	Disease_Syndrome_Disorder
3	19823	2167-02-25	1	congestive heart failure	363	386	Disease_Syndrome_Disorder
4	19823	2167-02-25	1	hypertension	413	424	Disease_Syndrome_Disorder
...	...	...	...	...	...	...	...
18456	70004	2182-08-16	13	Multilevel degenerative changes	1860	1890	Symptom
18457	70004	2182-08-16	13	uncovertebral joint hypertrophy	1897	1927	Disease_Syndrome_Disorder
18458	70004	2182-08-16	14	metastatic disease	1966	1983	Oncological
18459	70004	2182-08-16	14	cervical spine	1992	2005	Body_Part
18460	70004	2182-08-16	14	cord	2015	2018	Body_Part

Using deep learning, generalized entities can be recognized and extracted for the Gold layer within Delta Lake.

This result DataFrame is written to the Gold layer.

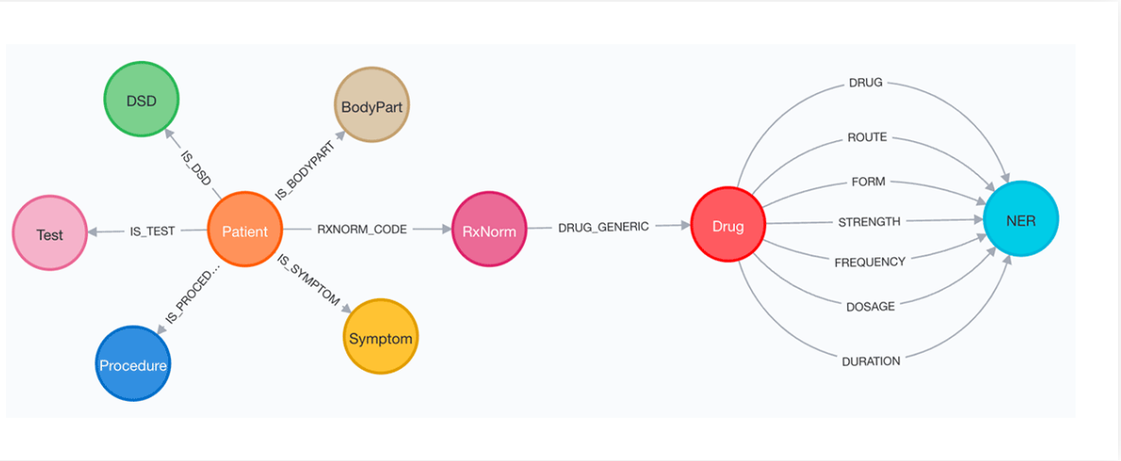
Creating and querying of the knowledge graph

For the creation of the knowledge graph (KG), the prior result DataFrames in the Gold layer are required as well as additional tabular de-identified demographic information of patients. See:

	subject_id	gender	dateOfBirth
0	19823	F	2099-05-05
1	22015	M	2085-10-04
2	17494	F	2193-10-01
3	21153	M	2057-03-03
4	12200	M	2136-06-17
5	23266	M	2122-03-16
6	75632	M	2118-09-04
7	27386	M	2114-08-20
8	28552	F	2044-10-22
9	70004	M	2127-12-06

For building the KG, best practices are to use your main cloud provider's graph capabilities. Two agnostic options to build a sufficient graph are: 1. Write your DataFrame to a NoSql database and use its graph API. 2. Use a native graph database.

The goal of both options is to get to a graph schema for the extracted entities that look like the following:



A visual representation of a graph schema to retrieve information based on underlying relationships for querying.

This can be achieved by splitting DataFrame into multiple DataFrames by ner_label and creating nodes and relationships. Examples for establishing relationship are (Examples are written in Cypher https://neo4j.com/developer/cypher/):

```
def add_symptoms(rows, batch_size=500):
    query = '''
    UNWIND $rows as row
    MATCH(p:Patient{name:row.subject_id})
    MERGE(n:Symptom {name:row.chunk})
    MERGE (p)-[:IS_SYMPTOM{date:date(row.date)}]->(n)

    WITH n
    MATCH (n)
    RETURN count(*) as total
    '''
    return update_data(query, rows, batch_size)

def add_tests(rows, batch_size=500):
    query = '''
    UNWIND $rows as row
    MATCH(p:Patient{name:row.subject_id})
    MERGE(n:Test {name:row.chunk})
    MERGE (p)-[:IS_TEST{date:date(row.date)}]->(n)

    WITH n
    MATCH (n)
    RETURN count(*) as total
    '''
    return update_data(query, rows, batch_size)
```

Once the KG is properly established, within any of the two options (in this example a graph database), a schema check will validate the count of records in each node and relationship:

	type	size
0	RXNORM_CODE	1665
1	DRUG_GENERIC	1665
2	DRUG	2782
3	FORM	241
4	STRENGTH	791
5	ROUTE	930
6	FREQUENCY	487
7	DOSAGE	320
8	DURATION	13
9	IS_SYMPTOM	4080
10	IS_DSD	2008
11	IS_TEST	2338
12	IS_BODYPART	3211
13	IS_PROCEDURE	972

Running a schema check ensures that the format and data relationships are as expected.

The KG is now ready to be intelligently queried to retrieve information based on the underlying established relationships within our NLP RE steps prior. The following shows a set of queries answering clinical questions:

1. Patient 21153's journey in medical records: symptoms, procedures, disease-syndrome-disorders, test, drugs and RxNorms:

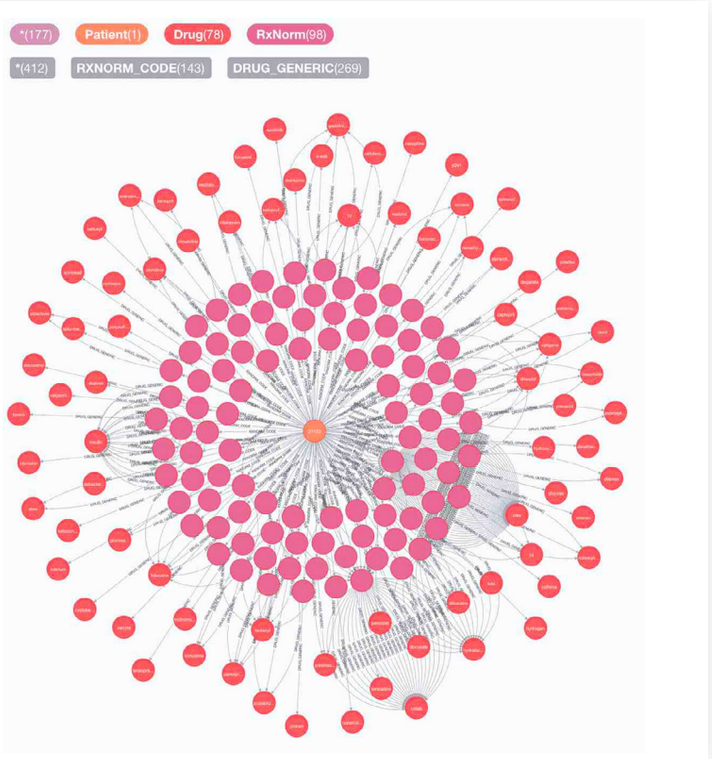
QUERY:

```
patient_name = '21153'

query_part1 = f'MATCH (p:Patient)-[r1:IS_SYMPTOM]->(s:Symptom) WHERE p.name = {patient_name} '

query_part2 = '''
WITH DISTINCT p.name as patients, r1.date as dates, COLLECT(DISTINCT s.name) as symptoms, COUNT(DISTINCT s.name) as num_symptoms

MATCH (p:Patient)-[r2:IS_PROCEDURE]->(pr:Procedure)
WHERE p.name=patients AND r2.date = dates

WITH DISTINCT p.name as patients, r2.date as dates, COLLECT(DISTINCT pr.name) as procedures, COUNT(DISTINCT pr.name) as num_procedures, symptoms, num_symptoms
MATCH (p:Patient)-[r3:IS_DSD]->(_d:DSD)
WHERE p.name=patients AND r3.date = dates

WITH DISTINCT p.name as patients, r3.date as dates, symptoms, num_symptoms, procedures, num_procedures,  COLLECT(DISTINCT _d.name) as dsds, COUNT(DISTINCT _d.name) as num_dsds
MATCH (p:Patient)-[r4:IS_TEST]->(_t:Test)
WHERE p.name=patients AND r4.date = dates

WITH DISTINCT p.name as patients, r4.date as dates, symptoms, num_symptoms, procedures, num_procedures, dsds, num_dsds, COLLECT(_t.name) as tests, COUNT(_t.name) as num_tests
MATCH (p:Patient)-[r5:RXNORM_CODE]->(rx:RxNorm)-[r6]->(_d:Drug)
WHERE p.name=patients AND r5.date = dates
RETURN DISTINCT p.name as patients, r5.date as dates, symptoms, num_symptoms, procedures, num_procedures, dsds, num_dsds, tests, num_tests, COLLECT(DISTINCT toLower(_d.name)) as drugs, COUNT(DISTINCT toLower(_d.name)) as num_drugs, COLLECT(DISTINCT rx.code) as rxnorms, COUNT(DISTINCT rx.code) as num_rxnorm
ORDER BY dates;
'''
```

DATAFRAME:

	patients	dates	symptoms	num_symptoms	procedures	num_procedures	dsds	num_dsds	tests	num_tests	drugs	num_drugs	rxnorms	num_rxnorm
0	21153	2109-12-17	[enviromental - rash, unresponsive, flatus, pa...	18	[thyroglossal cyst removal, serosagluos drai...	9	[esophgeal varices, cirrhosis, METABOLIC ALKAL...	7	[MAINTAIN FSBG, SuO2, PA, POST PLT CT, INR REM...	18	[everolimus, dex4]	2	[1310138, 607868]	2
1	21153	2109-12-18	[pain, Weak gag, sob, Pain, slight abd pain, s...	19	[Suction, removal of swan ganz catheter, Liver...	4	[ERYTHEMA, DSD, FORGETFUL, AT TIMES, PH DOWN, D...	10	[ABG, glucose, magnesium, ABG'S CLOSELY, Repla...	12	[dex4, insulin, clorsulon, cidaflex, crofom]	5	[607868, 1740938, 1006411, 1013644, 216281]	5
2	21153	2109-12-19	[aspiration voice clear after swallowing, slig...	25	[Anticipate extubation, Tolerating extubation,...	4	[metabolic alkalosis, Assit with pulmonary tio...	6	[IVP, 7p-7a, ABG - metabolic alkolotic, LS CTA...	9	[tobi, dimethicone, clonidine, diamox, insulin...	6	[220329, 1310821, 1360120, 151619, 139825, 211...	6
3	21153	2109-12-20	[C/O SOB, SOB, N/V]	3	[INCISION - STAPLES INTACT, LUNGS CTA BILAT]	2	[MIN C/O PAIN, DIMINISHED AT BASES, ABG'S ACCE...	3	[LS CTA, INCLUDING PLT CT, MIN ASSIST, HCL GTT...	6	[x-seb]	1	[220956]	1
4	21153	2109-12-21	[flatus, Flat affect, nonproductive cough, brn,...	17	[bile drainage]	1	[Hypertensive]	1	[cyclosporin levels, HRR, Plt, Phos, Hct, RA, ...	9	[insulin]	1	[378841]	1
5	21153	2109-12-28	[estimatedntblood loss,	15	[tonsillectomy, extubation, liver	7	[numerous spider nevi, chronic	9	[T tube study, platelet count, liver	12	[percocet, p2e1, citalopram,	36	[42844, 219017, 329444,	46

GRAPH:



A visual graph that uses NLP to show established relationships between data records.

2. Patients who are prescribed Lasix between May 2060 and May 2125:

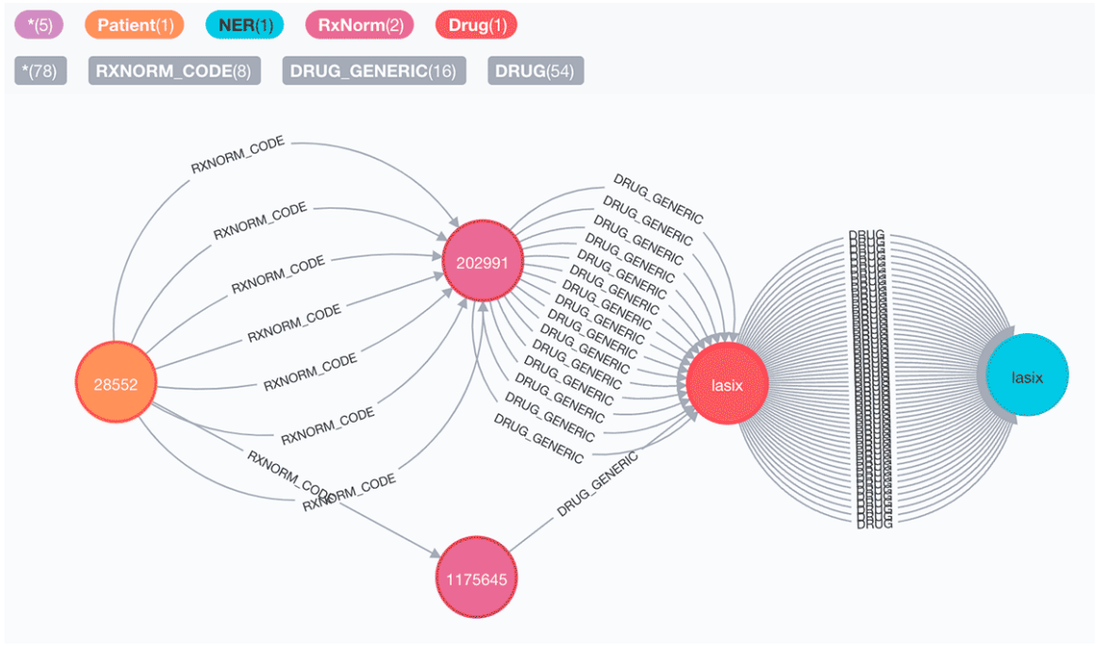
QUERY:

```
query_string ='''
MATCH (p:Patient)-[rel_rx]->(rx:RxNorm)-[rel_d]->(d:Drug)-[rel_n:DRUG]-
>(n:NER)
WHERE d.name IN ['lasix']
    AND rel_n.patient_name=p.name
    AND rel_n.date=rel_rx.date
    AND rel_rx.date >= date("2060-05-01")
    AND rel_n.date >= date("2060-05-01")
    AND rel_rx.date < date("2125-05-01")
    AND rel_n.date < date("2125-05-01")
RETURN DISTINCT
    d.name as drug_generic_name,
    p.name as patient_name,
    rel_rx.date as date
ORDER BY date ASC
'''
```

DATAFRAME:

	drug_generic_name	patient_name	date
0	lasix	28552	2122-07-17
1	lasix	28552	2122-07-18
2	lasix	28552	2122-07-20
3	lasix	28552	2122-07-21
4	lasix	28552	2122-07-22
5	lasix	28552	2122-07-24
6	lasix	28552	2122-07-29

GRAPH:



A visual graph that uses NLP to show established relationships between patient records and medication.

3. Dangerous drug combinations:

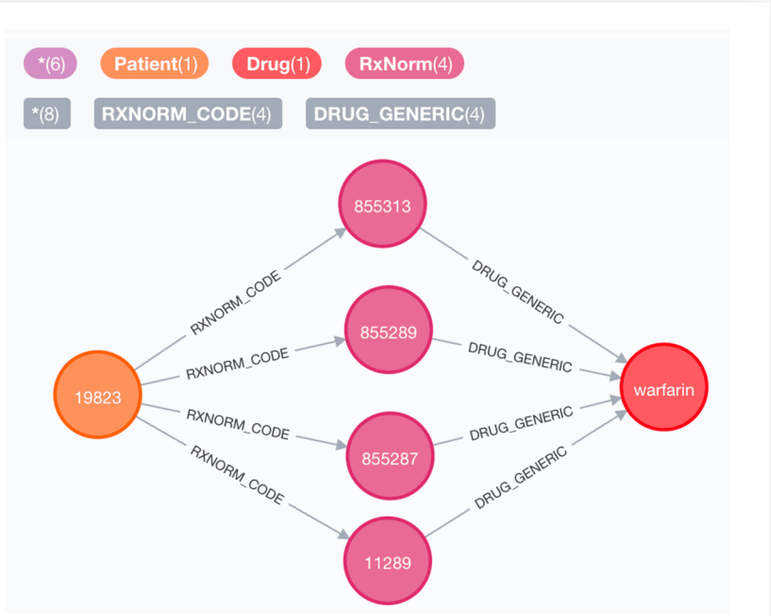
QUERY:

```
query_string ='''
WITH ["ibuprofen", "naproxen", "diclofenac", "indometacin", "ketorolac",
"aspirin", "ketoprofen", "dexketoprofen", "meloxicam"] AS nsaids
MATCH (p:Patient)-[r1:RXNORM_CODE]->(rx:RxNorm)-[r2]->(d:Drug)
WHERE any(word IN nsaids WHERE d.name CONTAINS word)
WITH DISTINCT p.name as patients, COLLECT(DISTINCT d.name) as nsaid_
drugs, COUNT(DISTINCT d.name) as num_nsaids
MATCH (p:Patient)-[r1:RXNORM_CODE]->(rx:RxNorm)-[r2]->(d:Drug)
WHERE p.name=patients AND d.name='warfarin'
RETURN DISTINCT patients,
                nsaid_drugs,
                num_nsaids,
                d.name as warfarin_drug,
                r1.date as date
'''
```

DATAFRAME:

	patients	nsaid_drugs	num_nsaids	warfarin_drug	date
0	19823	[diclofenac, aspirin, ketoprofen]	3	warfarin	2172-06-22
1	19823	[diclofenac, aspirin, ketoprofen]	3	warfarin	2167-11-27
2	19823	[diclofenac, aspirin, ketoprofen]	3	warfarin	2170-10-12

GRAPH:



A visual graph that uses NLP to show established relationships between prescription codes and medication.

4. Patients with hypertension or diabetes with chest pain:

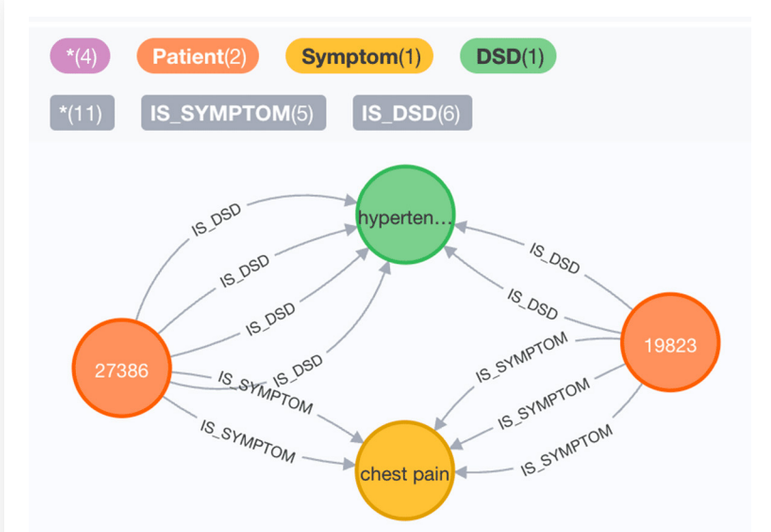
QUERY:

```
query_string = """
MATCH (p:Patient)-[r:IS_SYMPTOM]->(s:Symptom),
(p1:Patient)-[r2:IS_DSD]->(_dsd:DSD)
WHERE s.name CONTAINS "chest pain" AND p1.name=p.name AND _dsd.name IN
['hypertension', 'diabetes'] AND r2.date=r.date
RETURN DISTINCT p.name as patient, r.date as date, _dsd.name as dsd,
s.name as symptom
"""
```

DATAFRAME:

	patient	date	dsd	symptom
0	27386	2189-07-06	hypertension	chest pain
1	19823	2167-02-25	hypertension	chest pain

GRAPH:



A visual graph that uses NLP to show established relationships between patient records and medical symptoms.

Spark NLP and your preferred native KG database or KG API work well together for building knowledge graphs from extracted entities and established relationships. In many scenarios, federal agencies and industry enterprises require retrieving cohorts fast to gain population health or adverse event insights. As most data is available as unstructured text from clinical documents, as demonstrated, we can create a scalable and automated production solution to extract entities, build their relationships, establish a KG, and ask intelligent queries where the lakehouse supports the end-to-end.

Start building your cohorts with knowledge graphs using NLP

With this Solution Accelerator, Databricks and John Snow Labs make it easy to enable building clinical cohorts using KGs.

To use this Solution Accelerator, you can preview the notebooks online and import them directly into your Databricks account. The notebooks include guidance for installing the related John Snow Labs NLP libraries and license keys.

You can also visit our Lakehouse for Healthcare and Life Sciences page to learn about all of our solutions.

CHAPTER 8:

Solution Accelerator: Scalable Route Generation With Databricks and OSRM

Check out our new Scalable Route Generation Solution Accelerator for more details and to download the notebooks.

By **Rohit Nijhawan, Bryan Smith** and **Timo Roest**

September 2, 2022

Delivery has become an integral part of the modern retail experience. Customers increasingly expect access to goods delivered to their front doors within narrowing windows of time. Organizations that can meet this demand are attracting and retaining customers, forcing more and more retailers into the delivery game.

But even the largest retailers struggle with the cost of providing last mile services. Amazon has played a large part in driving the expansion of rapid home delivery, but even it has had difficulty recouping the costs. For smaller retailers, delivery is often provided through partner networks, which have had their own profitability challenges given customers' unwillingness to pay for delivery, especially for fast turnaround items such as food and groceries.

While many conversations about the future of delivery focus on autonomous vehicles, in the short term retailers and partner providers are spending considerable effort on improving the efficiency of driver-enabled solutions in an attempt to bring down costs and return margin to delivery orders. Given constraints like order-specific pickup and narrow delivery windows, differences in item perishability, which limits the amount of time some items may sit in a delivery vehicle, and the variable (but generally rising) cost of fuel and labor, easy solutions remain elusive.

Accurate route information is critical

Software options intended to help solve these challenges are dependent on quality information about travel times between pickup and delivery points. Simple solutions such as calculating straight-line distances and applying general rates of traversal may be appropriate in hypothetical examinations of routing problems, but for real-world evaluations of new and novel ways to move drivers through a complex network of roads and pathways, access to far more sophisticated approaches for time and distance estimations are required.

It's for this reason that more and more organizations are adopting routing software in both their operational and analytics infrastructures. One of the more popular software solutions for this is Project OSRM (Open Source Routing Machine). Project OSRM is an open source platform for the calculation of routes using open source map data provided by the OpenStreetMap project. It provides a fast, low-cost way for organizations to deploy a routing API within their environment for use by a wide range of applications.

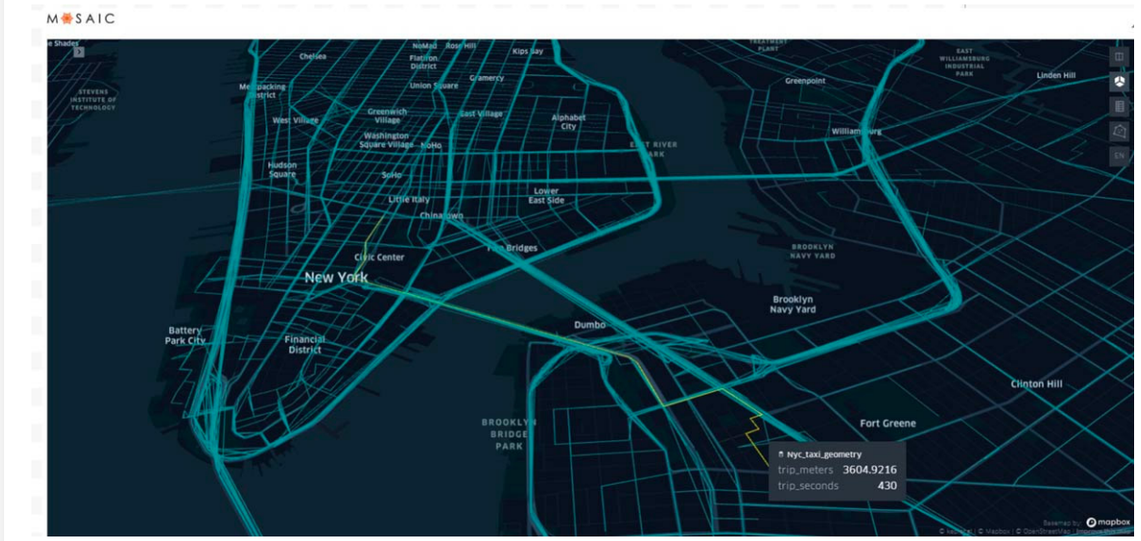


Figure 1. Routes associated with taxi rides in New York City generated by the OSRM Backend Server and visualized using Databricks Mosaic.

Scalability needed for route analytics

Typically deployed as a containerized service, the OSRM server is often employed as a shared resource within an IT infrastructure. This works fine for most application integration scenarios, but as teams of data scientists push large volumes of simulated and historical order data to such solutions for route generation, they can often become overwhelmed. As a result, we are hearing an increasing number of complaints from data scientists about bottlenecks in their route optimization and analysis efforts and from applications teams frustrated with applications being taken offline by hard-to-predict analytics workloads.

To address this problem, we wish to demonstrate how the OSRM server can be deployed within a Databricks cluster. Databricks, a unified platform for machine learning, data analysis and data processing, provides scalable and elastic access to cloud computing resources. Equally as important, it has support for complex data structures such as those generated by the OSRM software and geospatial analysis through a variety of native capabilities and open source libraries. By deploying the OSRM server into a Databricks cluster, data science teams can access the routing capacity they need without interfering with other workloads in their environment.

The key to such a solution is in how Databricks leverages the combined computational capacity of multiple servers within a cluster. When the OSRM server software is deployed to these servers, the capacity of the environment grows with the cluster. The cluster can be spun up and shut down on demand as needed, helping the organization avoid wasting budget on idle resources. Once configured, the OSRM route generation capabilities are presented as easy-to-consume functions that fit nicely into the familiar data processing and analysis work the data science teams perform today.

By deploying the OSRM server in the Databricks cluster, data science teams can now evaluate new algorithms against large volumes of data without capacity constraints or fear of interfering with other workloads. Faster evaluation cycles mean these teams can more rapidly iterate their algorithms, fine tuning their work to drive innovation. We may not yet know the perfect solution to getting drivers from stores to customers' homes in the most cost-effective manner, but we can eliminate one more of the impediments to finding it.

> **To examine more closely how the OSRM server can be deployed within a Databricks cluster, check out our Databricks Solution Accelerator for Scalable Route Generation.**

CHAPTER 9:

Fine-Grained Forecasting With R

By **Divya Saini** and **Bryan Smith**

October 17, 2022

Demand forecasting is foundational to most organizations' planning functions. By anticipating consumer demand for goods and services, activities across marketing and up and down the supply chain can be aligned to ensure the right resources are in the right place at the right time and price, maximizing revenue and minimizing cost.

While the progression from forecast to plan and ultimately to execution is often far less orderly than the previous statement would imply, the process begins with the adoption of an accurate forecast. The expectation is never that the forecast will perfectly predict future demand but instead will provide a manageable range of likely outcomes around which the organization can plan its activities (figure 1).

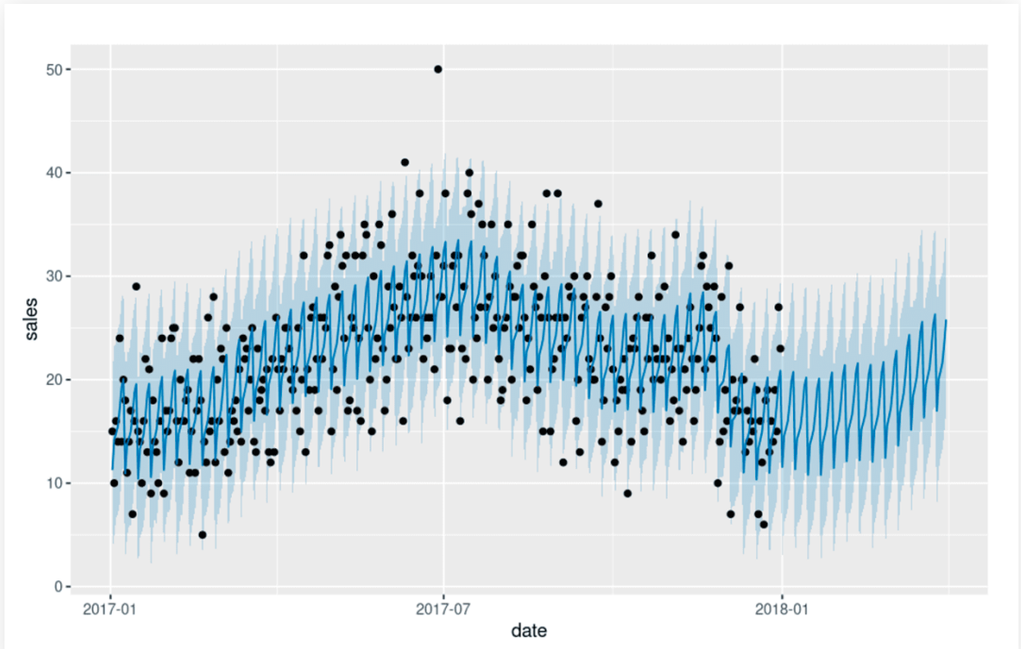
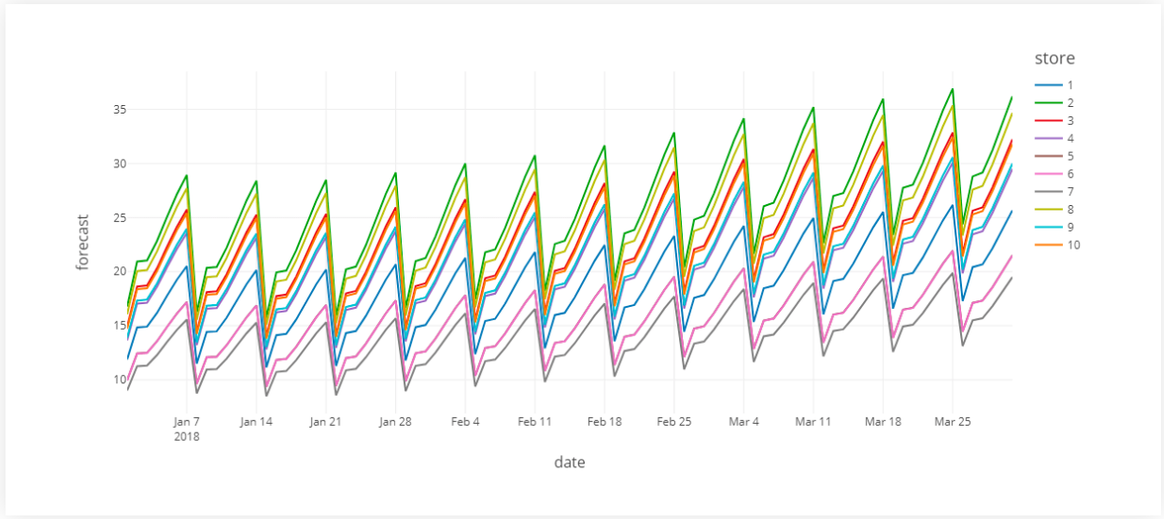


Figure 1. A typical forecast showing mean prediction (blue line), 95% confidence interval (light blue band), and observed values relative to forecast for the historical period (black dots).

That said, many organizations struggle to produce forecasts sufficiently reliable for this purpose. One survey revealed inaccuracies of 20 to 40% across a range of industries. Other surveys report similar results with the consequence being an erosion of organizational trust and the continuation of planning activities based on expert opinion and gut feel.

The reasons behind this high level of inaccuracy vary, but often they come down to a lack of access to more sophisticated forecasting technology or the computational resources needed to apply them at scale. For decades, organizations have found themselves bound to the limitations of prebuilt software solutions and forced to produce aggregate-level forecasts in order to deliver timely results. The forecasts produced often do a good job of presenting the macro-level patterns of demand but are unable to capture localized variations where that demand is met (figure 2).



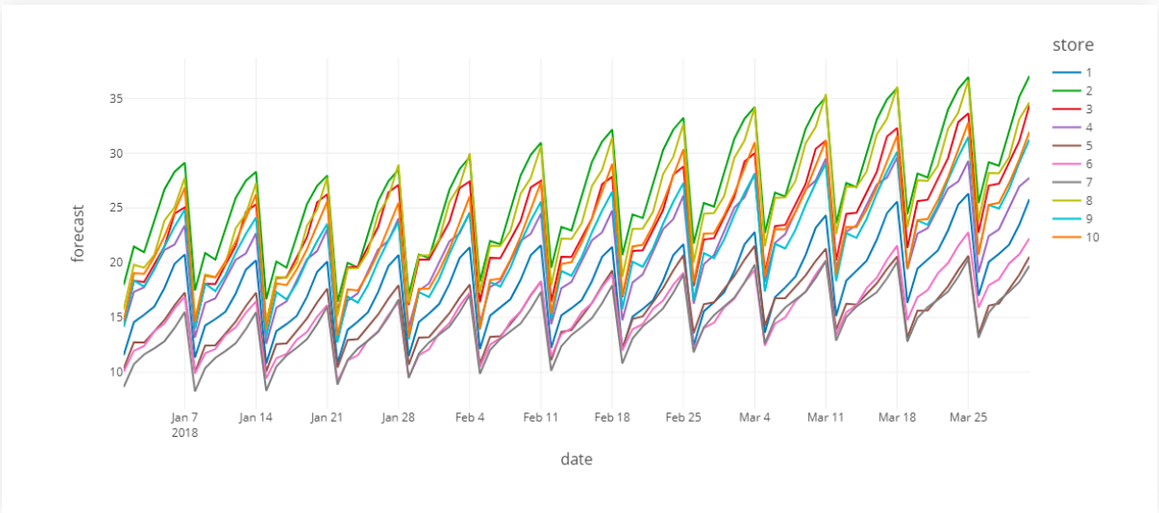


Figure 2. A comparison of an aggregated forecast allocated to the store-item level vs a fine-grained forecast performed at the individual store-item level.

But with the emergence of data science as an enterprise capability, there is renewed interest across many organizations to revisit legacy forecasting practices. As noted in a far-reaching examination of data and AI-driven innovation by McKinsey, "a 10 to 20% increase in forecasting accuracy translates into a potential 5% reduction of inventory costs and revenue increases of 2 to 3%." In an increasingly competitive economic climate, such benefits are providing ample incentive for organizations to explore new approaches.

The R language provides access to new forecasting techniques

Much of the interest in new and novel forecasting techniques is expressed through the community of R developers. Created in the mid-1990s as an open source language for statistical computing, R remains highly popular with statisticians and academics pushing the boundaries of a broad range of analytic techniques. With broad adoption across both graduate and undergraduate data science programs, many organizations are finding themselves flush with R development skills, making the language and the sophisticated packages it supports increasingly accessible.

The key to R's sustained success is its extensibility. While the core R language provides access to fundamental data management and analytic capabilities, most data scientists make heavy use of the broad portfolio of packages contributed by the practitioner community. The availability of forecasting-related packages in the R language is particularly notable, reflecting the widespread interest in innovation within this space.

The cloud provides access to the computational resources

But the availability of statistical software is just part of the challenge. To drive better forecasting outcomes, organizations need the ability to predict exactly how a product will behave in a given location, and this can often mean needing to produce multiple-millions of forecasts on a regular basis.

In prior periods, organizations struggled to provide the resources needed to turn around this volume of forecasts in a timely manner. To do so, organizations would need to secure large volumes of servers that would largely sit idle between forecasting cycles, and few could justify the expense.

However, with the emergence of the cloud, organizations now find themselves able to rent these resources for just those periods within which they are needed. Taking advantage of low-cost storage options, forecast output can be persisted for consumption by analysts between forecasting cycles, making fine-grained forecasting at the store-item level suddenly viable for most organizations.

The challenge then is how best to acquire these resources when needed, distribute the work across them, persist the output and deprovision the computing resources in a timely manner. Past frameworks for such work were often complex and inflexible, but today we have Databricks.

Databricks brings together R functionality with cloud scalability

Databricks is a highly accessible platform for scalable data processing within the cloud. Through Databricks clusters, large volumes of virtual servers can be rapidly provisioned and deprovisioned to affect workloads big and small. Leveraging widely adopted languages such as Python, SQL and R, developers can work with data using simple constructs, allowing the underlying engine to abstract much of the complexity of the distributed work. That said, if we peer a bit behind the covers, we can find even greater opportunities to exploit Databricks, including ways that help us tackle the problem of producing very large volumes of forecasts leveraging R.

To get us started with this, consider that Databricks collects data within a Spark DataFrame. To the developer, this DataFrame appears as a singular object. However, the individual records accessible within the DataFrame are in fact divided into smaller collections referred to as partitions. A given partition resides on a single worker node (server) within the Databricks cluster, so while the DataFrame is accessed as a singular object (on the primary/driver node), it is in fact a collection of non-overlapping partitions residing on the servers that make up the cluster (figure 3).

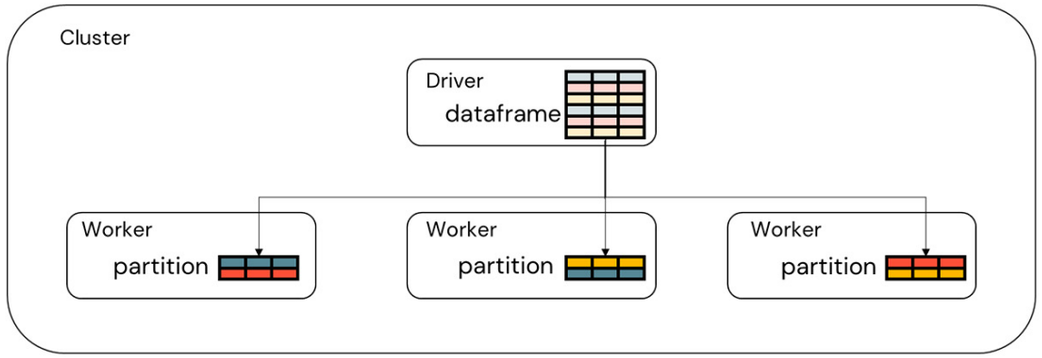


Figure 3. The relationship between the Spark DataFrame, its partitions and individual records.

Against the DataFrame, a developer applies various data transformations. Under the covers, the Spark engine decides how these transformations are applied to the partitions. Some transformations such as the grouping of data ahead of an aggregation trigger the engine to rearrange the records in the DataFrame into a new set of partitions within which all the records associated with a given key value reside. If you were, for example, to read historical sales data into a Spark DataFrame and then group that data on a store and item key, then you would be left with one partition for each store-item combination within the data set. And that's the key to tackling our fine-grained forecasting challenge (figure 4).

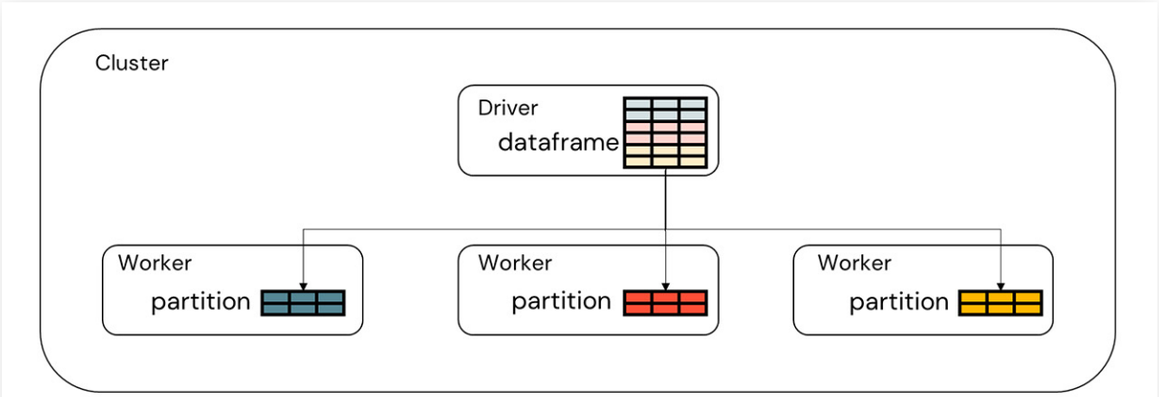


Figure 4. The reordering of records within a DataFrame as part of a grouping operation.

By grouping all the records for a given store and item within a partition, we've isolated all the data we need to produce a store-item level forecast in one place. We can then write a function which treats the partition as an R DataFrame, train a model against it and return predictions back as a new Spark DataFrame object (figure 5).

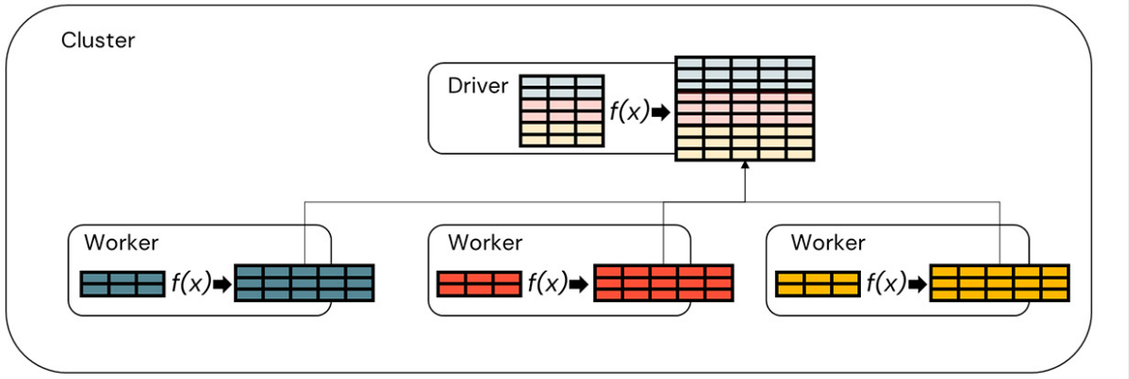


Figure 5. The application of a custom function (f(x)) against a grouped DataFrame to produce a new DataFrame.

The R language provides us two options for doing this. Using the SparkR package, we can use the *gapply* function to group the data and apply a user-defined function to each partition. Using the SparklyR package, we can do the same thing using the spark_apply function. The choice of R packages, i.e., SparkR or SparklyR, to facilitate this work is really up to developer preference, with SparkR providing more direct access to Spark functionality and SparklyR wrapping that functionality in an API more familiar to developers comfortable with the R *dpylr* package.

Want to see these two approaches in action? Then please check out the Solution Accelerators.

CHAPTER 10:

GPU-Accelerated Sentiment Analysis Using PyTorch and Hugging Face on Databricks

By **Divya Saini** and **Bryan Smith**

October 17, 2022

Sentiment analysis is commonly used to analyze the sentiment present within a body of text, which could range from a review to an email or a tweet. Deep learning-based techniques are one of the most popular ways to perform such an analysis. However, these techniques tend to be very computationally intensive and often require the use of GPUs, depending on the architecture and the embeddings used. Hugging Face (https://huggingface.co) has put together a framework with the transformers package that makes accessing these embeddings seamless and reproducible. In this work, I illustrate how to perform scalable sentiment analysis by using the Hugging Face package within PyTorch and leveraging the ML runtimes and infrastructure on Databricks.

Sentiment analysis

Sentiment analysis is the process of estimating the polarity in a user's sentiment, (i.e., whether a user feels positively or negatively from a document or piece of text). The sentiment can also have a third category of neutral to account for the possibility that one may not have expressed a strong positive or negative sentiment regarding a topic. Sentiment analysis is a form of opinion mining but differs from stance or aspect detection, where a user's stance regarding a particular aspect or feature is extracted.

For example, the sentiment in the sentence below is overwhelmingly positive:

"The restaurant was great"

However, consider the sentence below:

"The restaurant was great but the location could be better"

It is harder to estimate the sentiment, but the user's stance regarding the restaurant can be seen as generally positive in spite of the fact that their stance regarding the location was negative. To summarize, sentiment analysis provides coarse-grained information, while stance detection provides more information regarding certain aspects.

Sentiment analysis can be used to ascertain a customer's sentiment regarding a particular product, the public's reaction to an event, etc.

Types of sentiment analysis

Sentiment analysis can be performed using lexicon-based techniques or machine learning-based techniques. Lexicon-based techniques use pre-labeled vocabulary to estimate the sentiment from text. A variety of techniques are used to aggregate the sentiment from the individual sentiment assigned to the tokenized words. Some of the popular frameworks in this category are SentiNet and AFINN . VADER, an open source package with the NLTK, is another example that is used specifically for analyzing social media posts. Machine learning-based sentiment analysis uses pretrained embeddings along with a deep learning (DL) architecture to infer the sentiment in a body of text. In this blog, we will only cover ML-based techniques through the embeddings available from Hugging Face. The sentiment analysis model, composed of the architecture and the embeddings, can then be optionally fine-tuned if domain-specific labels are available for the data. It is often the case that such supervised training can improve the performance even when only a small amount of labeled data is available. Embeddings such as Elmo, BERT and Roberta are some of the popularly available language embeddings for this purpose.

```
from transformers import AutoModelForSequenceClassification
from transformers import AutoTokenize
MODEL =  "distilbert-base-uncased-finetuned-sst-2-english"
tokenizer = AutoTokenizer.from_pretrained(MODEL)
model = AutoModelForSequenceClassification.from_pretrained(MODEL)
tokenized_text = tokenizer([“Hello world”], padding=True, return_
tensors='pt')
output = model(tokenized_text[‘input_ids’])
pt_predictions = nn.functional.softmax(output.logits, dim=1)
```

Looking at the example above, we notice two imports for a tokenizer and a model class. We can instantiate these by specifying a certain pretrained model such as BERT. You can search for a model here. You then pass a sequence of strings to the tokenizer to tokenize it and specify that the result should be padded and returned as PyTorch tensors. The tokenized results are an object from which we extract the encoded text and pass it to the model. The results of the model are then passed through a softmax layer in the case of sentiment analysis to normalize the results as a sentiment score.

(Multi) GPU-enabled inference

The process of inference consists of the following components:

1. Dataloader for serving batches of tokenized data
2. Model class that performs the inference
3. Parallelization of the model on the GPU devices
4. Iterating through the data for inference and extracting the results

Dataloader

PyTorch uses the Dataloader abstraction for extracting batches of data to be used either for training or inference purposes. It takes as input an object of a class that extends the 'Dataset' class. Here we call that class 'TextLoader'. It is necessary to have at least two methods in this class :

(a) __len__() : returns the length of the entire data

(b) __getitem__(): extracts and returns a single element of the data

```
MODEL = "distilbert-base-uncased-finetuned-sst-2-english"
class TextLoader(Dataset):
    def __init__(self, file=None, transform=None, target_transform=None,
tokenizer=None):
        self.file = pd.read_json(file, lines=True)
        self.file = self.file
        self.file = tokenizer(list(self.file['full_text']), padding=True,
truncation=True, max_length=512, return_tensors='pt')
        self.file = self.file['input_ids']
        self.transform = transform
        self.target_transform = target_transform

    def __len__(self):
        return len(self.file)

    def __getitem__(self, idx):
        data = self.file[idx]
        return(data)
```

Now Dataloader accepts the object instance of this class named 'data' here, along with the batch size of the data to be loaded in a single iteration. Note that I have set the 'shuffle' flag to False here, since we want to preserve the order of the data.

Dataloader automatically handles the division of the data that it receives to be utilized by each of the GPU devices. If the data is not evenly divisible, it offers the option to either drop elements or pad a batch with duplicate data points. This is something you might want to keep in mind, especially during the inference or prediction process.

```
tokenizer = AutoTokenizer.from_pretrained(MODEL)
data = TextLoader(file=file = '/PATH_TO/FILE.txt', tokenizer=tokenizer)
train_dataloader = DataLoader(data, batch_size=120, shuffle=False) #
Shuffle should be set to False
```

Model class

The model class is fairly similar to the code that we saw above, with the only difference being that it is now wrapped in a nn.module class. The model definition is initialized within __init__ and the forward method applies the model that is loaded from Hugging Face.

```
class SentimentModel(nn.Module):
   def __init__(self):
        super(SentimentModel, self).__init__()
        self.model = AutoModelForSequenceClassification.from_
pretrained(MODEL)

    def forward(self, input):
        output = self.model(input)
        pt_predictions = nn.functional.softmax(output.logits, dim=1)
        return(pt_predictions)

model3 = SentimentModel()
```

Model parallelization and GPU dispatch

In PyTorch, a model or variable that is created needs to be explicitly dispatched to the GPU. This can be done by using the '.to('cuda') method. If you have multiple GPUs, you can even specify a device id as '.to(cuda:0)'. Additionally, in order to benefit from data parallelism and run the training or inference across all the GPU devices on your cluster, one has to wrap the model within 'DataParallel'.

While this code assumes that you have more than one GPU on your cluster, if that is not the case, the only change required is 'device_ids' to [0] or simply not specifying that parameter (the default gpu device will be automatically selected).

```
dev = 'cuda'
if dev == 'cpu':
  device = torch.device('cpu')
  device_staging = 'cpu:0'
else:
  device = torch.device('cuda')
  device_staging = 'cuda:0'

try:
  model3 = nn.DataParallel(model3, device_ids=[0,1,2,3])
  model3.to(device_staging)
except:
  torch.set_printoptions(threshold=10000)
```

Iteration loop

The following loop iterates over the batches of data, transfers the data to the GPU device before passing the data through the model. The results are then concatenated so that they can be exported to a data store.

```
out = torch.empty(0,0)
for data in train_dataloader:
    input = data.to(device_staging)
    if(len(out) == 0):
      out = model3(input)
    else:
      output = model3(input)
      with torch.no_grad():
        out = torch.cat((out, output), 0)
```

```
file = '/PATH_TO/FILE.txt'
df = pd.read_json(file, lines=True)['full_text']
res = out.cpu().numpy()
df_res = pd.DataFrame({ "text": df, "negative": res[:,0], "positive":
res[:,1]})
display(df_res)
```

	text	negative	positive
1	RT @PeteButtigieg: I am not just here to end the era of Donald Trump. I am here to launch the era that must come next—an era of hope and be…	0.01036986	0.98963016
2	“The president has again dishonored our armed services.” —@PeteButtigieg https://t.co/E7ju4w7vki	0.9844322	0.015567774
3	Win with hope. https://t.co/JI1ifDCg5Z	0.18291013	0.81708986
4	RT @UniNoticias: El turno de @PeteButtigieg: al llegar al escenario del foro #ReaAmérica fue ovacionado. https://t.co/swHQQ6zFL2 #Destino20…	0.9839091	0.016090982
5	Don’t miss @PeteButtigieg speak about his vision for the future at the @Univision and @CA_Dem Presidential Forum 📺 https://t.co/4EWYxJ4MPG	0.9264734	0.073526636
6	87 days until the New Hampshire primary. Are you on board? https://t.co/7xmc5lpms9 https://t.co/xpWSK70vTU	0.9577003	0.04229966
7	The American Dream should be accessible to everyone: https://t.co/2oXvwc7PJv https://t.co/MPmEy14fW1	0.04887515	0.95112485

Scalable inference for lots of files

In the example above, the data was read in from a single file; however, when dealing with large amounts of data, it is unlikely that all of this data is available in a single file. The following shows the entire code with the changes highlighted for using the Dataloader with multiple files.

```
MODEL = “distilbert-base-uncased-finetuned-sst-2-english”
def get_all_files():
  file_list = [‘/PATH/FILE1’,
               ‘/PATH/FILE2’,
               ‘/PATH/FILE3’]
  return(file_list)

class TextLoader(Dataset):
    def __init__(self, file=None, transform=None, target_
transform=None, tokenizer=None):
        self.file = pd.read_json(file, lines=True)
        self.file = self.file
        self.file = tokenizer(list(self.file[‘full_text’]),
padding=True, truncation=True, max_length=512, return_tensors=’pt’)
        self.file = self.file[‘input_ids’]
        self.transform = transform
        self.target_transform = target_transform

    def __len__(self):
        return len(self.file)

    def __getitem__(self, idx):
        data = self.file[idx]
        return(data)

class SentimentModel(nn.Module):
      def __init__(self):
        super(SentimentModel, self).__init__()
        self.model = AutoModelForSequenceClassification.from_
pretrained(MODEL)

    def forward(self, input):
        output = self.model(input)
        pt_predictions = nn.functional.softmax(output.logits, dim=1)
        return(pt_predictions)
```

```
dev = 'cuda'
if dev == 'cpu':
  device = torch.device('cpu')
  device_staging = 'cpu:0'
else:
  device = torch.device('cuda')
  device_staging = 'cuda:0'
tokenizer = AutoTokenizer.from_pretrained(MODEL)
all_files = get_all_files()
model3 = SentimentModel()
try:
      model3 = nn.DataParallel(model3, device_ids=[0,1,2,3])
      model3.to(device_staging)
except:
      torch.set_printoptions(threshold=10000)

for file in all_files:
    data = TextLoader(file=file, tokenizer=tokenizer)
    train_dataloader = DataLoader(data, batch_size=120,
shuffle=False) # Shuffle should be set to False


    out = torch.empty(0,0)
    for data in train_dataloader:
        input = data.to(device_staging)
        if(len(out) == 0):
          out = model3(input)
        else:
          output = model3(input)
          with torch.no_grad():
            out = torch.cat((out, output), 0)

    df = pd.read_json(file, lines=True)['full_text']
    res = out.cpu().numpy()
    df_res = pd.DataFrame({ "text": df, "negative": res[:,0],
"positive": res[:,1]})
```

Conclusion

We discussed how the Hugging Face framework can be used for performing sentiment analysis using PyTorch. Additionally, it was shown how GPUs could be used to accelerate this inference process. The use of the Databricks platform with the easily available ML runtimes and availability of the state-of-the-art GPUs make it easy to experiment with and deploy these solutions.

> **For more details, please check out the attached notebook!**

CHAPTER 11:

Jumbo Transforms How They Delight Customers With Data-Driven Personalized Experiences

This is a collaborative post between Databricks and Jumbo. We thank Wendell Kuling, Manager Data Science and Analytics at Jumbo Supermarkten, for his contribution.

By **Wendell Kuling**
June 7, 2022

At Jumbo, a leading supermarket chain in the Netherlands and Belgium with more than 100 years of history, we pride ourselves on a "customers first, data second" approach to business. However, this doesn't mean that data isn't central to our mission. Rather, our organization and data teams are completely built around customer satisfaction and loyalty. Supermarkets operate in an extremely competitive and complex space, in which all components of the customer experience require optimization, including inventory, assortment selection, pricing strategies, and product importance per segment.

When we rolled out our loyalty program, the amount of new customer data points that started coming in got our data teams to rethink about how we optimize the customer experience at scale. At its core, Jumbo seeks to delight customers and deliver optimal grocery shopping experiences. Running differentiated store formats on the one hand, and personalizing messages and offers to customers on the other, made it impossible to continue working in the traditional way. That's where data analytics and AI come in: they help us to make decisions at the scale required for personalization and differentiation.

With the rollout of our revamped customer loyalty program, we were suddenly able to better understand a wide range of our individual customer preferences, such as which products are most important and which are often forgotten, as well as the best time of day to communicate with customers and on which channel. However, as data volumes grew exponentially, our analytics and ML capabilities began to slow as we weren't equipped to handle such scale. Increased data volumes meant increased complexity and resources required to try to handle it from an infrastructure perspective, which impacted our ability to deliver insights in a timely manner. Long processing and querying times were unacceptable. After years on a traditional statistical software package connecting to a traditional RDBMS and analytics in Jupyter notebooks, we knew that if we wanted to best use this data and deliver shopping experiences that make a difference, it was time for us to take steps to modernize our approach and the underlying technologies that enable it. We needed a platform that was able to crunch through customer-level data and train models at a scale much more than we could handle on our individual machines.

In addition to needing to modernize our infrastructure to help us thrive with big data analytics, we also needed better ways to increase the speed from concept to production, decrease onboarding time for new people, collaborate, and deliver self-service access to data insights for our analysts and business users to help serve insights around pricing, inventory, merchandising, and customer preferences. After looking into a variety of options, we selected the Databricks Lakehouse Platform as the right solution for our needs.

From foundational customer loyalty to exceptional customer experiences

With Databricks Lakehouse implemented, we now run a substantial number of data science and data engineering initiatives in parallel to turn our millions of customers into even more loyal fans.

As an example of data products exposed directly to customers, we're now able to combine, on a customer level, information about purchases made online and offline, which was very challenging before. This omnichannel view allows us to create a more comprehensive recommendation engine online, which has seen tremendous engagement. Now, based on past purchasing history as well as first-party data collected with consent, we can serve product-relevant recommendations that pique the interests of the consumer. Obviously, this is great from a business perspective, but the real benefit is how happy it's made our customers. Now, they're less likely to forget important items or purchase more than they need. This balance has significantly increased customer loyalty.

In one example of data products that have helped improve the customer experience, we continually run an algorithm that proactively suggests assortment optimizations to assortment managers. This algorithm needs to run at scale at acceptable costs, as it optimizes using large quantities of data on physical store and online customers, the overall market, financials and store-level geographical characteristics. Once opportunities are identified, it is presented in combination with the same breadth and depth of data upon which it was based.

From a technical perspective, the Databricks Lakehouse architecture is able to drive these improved experiences with the help of Microsoft Azure Synapse. Together, this combo has allowed us to manage, explore and prepare data for analysis for automated (proposed) decision-making, and make that analysis easy to digest via BI tools such as Power BI. With deeper insights, we have helped spread a more meaningful understanding of customer behavior and empowered our data teams to more effectively predict the products and services they would want.

Databricks is now fully integrated into our end-to-end workflow. The process starts with a unified lakehouse architecture, which leverages Delta Lake to standardize access to all relevant data sources (both historical and real-time). For example, Delta Lake also helps us to build data pipelines that enable scalable, real-time analytics to reduce in-store stockouts for customers and at the same time reduce unnecessary food waste due to over-ordering perishables, such as fresh produce, that won't sell. At the same time, Databricks SQL provides our data analysts with the ability to easily query our data to better understand customer service issues, processed with NLP in the background, and relate these issues to the operational performance of different departments involved. This helps us to make improvements faster that enhance the customer experience most.

We would not have been able to accelerate our modernization efforts without the expert training and technical guidance from the Databricks Academy and Customer Success Engineer, which acts as a direct injection of knowledge for our data science department. This deeper understanding of how to leverage all our data has led to significant improvements in how we manage our assortment and supply chain, make strategic decisions, and better support our customers' evolving needs.

Excellence has no ceiling when driven by data and AI

By focusing on improving the customer experience through Databricks Lakehouse, we were enabled beyond our initial expectations. The steps we've taken to modernize our approach to how we can use data have really set us up as we continue to transform our business in a way that pushes our industry forward.

It's remarkable to see how quickly data and AI capabilities are becoming the new normal and we are now well positioned to realize the direct impact of our efforts to be data-driven. The output of sophisticated machine learning models is considered "common practice" within 4 weeks after introduction. And the speed of idea to production is counted in weeks nowadays, not months.

Going forward, we'll continue to see the adoption level increase, not just at Jumbo, but across all of commerce. And for those who are also embarking on a data transformation, I would strongly recommend that they take a closer look at improvement possibilities in the experience they're providing customers. Feeding back your analytics data products into operational processes at scale is key to transforming all areas of the business forward and successfully into the future.

CHAPTER 12:

Ordnance Survey Explores Spatial Partitioning Using the British National Grid

This is a collaborative post by Ordnance Survey, Microsoft and Databricks. We thank Charis Doidge, Senior Data Engineer, and Steve Kingston, Senior Data Scientist, Ordnance Survey, and Linda Sheard, Cloud Solution Architect for Advanced Analytics and AI at Microsoft, for their contributions.

By **Milos Colic, Robert Whiffin, Pritesh Patel, Charis Doidge, Steve Kingston** and **Linda Sheard**

October 11, 2021

This blog presents a collaboration between Ordnance Survey (OS), Databricks and Microsoft that explores spatial partitioning using the British National Grid (BNG).

OS is responsible for the design and development of a new National Geographic Database (NGD) data delivery for Great Britain (GB) under the Public Sector Geospatial Agreement.

OS has been working closely with Databricks and Microsoft on the architectural strategy and data engineering capabilities that underpin the NGD as part of a Core Data Services Platform. This platform enables OS to migrate geospatial data processing that has traditionally been carried out on on-prem machines in single-threaded processes and applications, such as FME to cloud compute, that are available and scalable on-demand — thus, achieving the processing and analysis of geospatial data at scale. OS is using Azure Databricks to add Apache Spark™ capability to the cloud platform, and this brings the opportunity to rethink how to optimize both data and approach to perform geospatial joins at scale using parallelized processing.

Indexing spatial data appropriately is one aspect of such optimization work, and it doesn't just stop at selecting an index. The focus of this blog is on how we designed a process that makes maximal use of the index to allow the optimizers provided by Azure Databricks to tune the way that data is loaded from disk during scaled geospatial joins.

There are various grid indexes such as BNG, Geohash, Uber's H3, and Google's S2 that divide the spatial world into bins with identifiers. While some of these have been developed specifically in the context of modern geoanalytics, and therefore tend to be well supported with associated libraries and practical examples of use in that context, the British National Grid indexing system was defined in 1936 and is deeply embedded in the Great Britain geospatial data ecosystem, but not yet exploited and made accessible for geoanalytics at scale. Our secondary motivation here, therefore, was to show that it can be used directly for optimizing spatial joins, avoiding the need to convert Great Britain's geospatial data sets to other indexing systems first. Our team implemented a mosaic technique that decomposed polygons into simplified geometries bounded by their presence in a given BNG index. By effectively limiting index space comparisons and spatial predicate evaluations, the approach yielded notable query performance gains.

The point-in-polygon: how hard can it be?

How hard is it to determine whether a point is inside a polygon (PIP)? The question of how to determine whether a point is contained within a polygon has already been answered years ago. This fact can introduce bias, making us jump to conclusions like, "it is easy; it has already been solved." However, with the advancement of technology and the introduction of parallel systems, we have found ourselves asking this same question but in a new context. That context is using a PIP as a join relation over big (geospatial) data. The new problem is ensuring that we have high levels of parallelism in our approach. Unfortunately, the old answers no longer apply in this new context.

We can think of the join relationship as a pairing problem. We can observe it as having two data sets that contain rows that match with a set of rows from the other data set while satisfying the join condition. The complexity of join relation is O(n*m), or what is commonly known as the Cartesian Product (complexity). This is the worst-case complexity for a join relation and, in simple terms, means that we need to compare each record from one data set with each record of the other data set to resolve all matches. Many systems implement techniques and heuristics to push this complexity to a lower level. However, this is the baseline, and we will start our considerations from this baseline.

In the context of OS's geospatial data processing, one of the most common PIP joins routinely undertaken is between all address point geometries (approximately 37 million) and all large-scale building polygon geometries (approximately 46 million) in GB.

Diagram A

The (not so) hidden cost?

While discussing join relation complexity, we have made an oversight. The traditional complexity assumes a fixed cost for each pair resolution, that is, the cost of arriving at a conclusion of match or no match for each pair of records during the join operation, which we will call `O(join)`. The true cost of the join is `O(n*m)*O(join)`. In the traditional equivalence relationship class, where we are just looking whether a join key on the left matches a join key on the right, we assume `O(join)` is `O(1)`, or to put it simply, the cost of comparison is one arithmetic operation, and it is constant. This is not always the case; for example, joining on a string comparison is more expensive than an equivalence between two integers.

But what of PIP — how costly is it relatively? The most widely used algorithm to answer PIP is the ray-tracing method. The complexity of this algorithm is `O(v)`, where v is the number of vertices of the polygon in question. The algorithm is applicable to both convex and non-convex shapes, and it maintains the same complexity in both cases.

Adding the comparison cost to our cost model brings our total complexity to cubic form. If we replace `O(join)` with `O(v)`, where v is the average number of vertices, we have the total complexity of `O(n*m)*O(v)`. And this is both expensive and time-consuming!

Work smarter, not harder!

We can do better than `O(n*m)*O(v)`. We can use Spark to help us beat the Cartesian Product complexity. Spark leverages hash joins under the hood. Depending on the join predicate, Spark can execute one of the following join strategies:

- broadcast hash join with complexity of `O(max(n,m))*O(join)`
- shuffle hash join (similar to Grace Hash Join) with complexity of `O(n+m)*O(join)`
- shuffle sort-merge join with complexity of `O(n*log(n)+m*log(m))*O(join)`
- broadcast nested loop join (Cartesian join) with complexity `O(n*m)*O(join)`

Amazing! We can just use Spark, and we will avoid the most costly outcome, can't we? No! Unfortunately, Spark will default to Cartesian join for PIP joins. Why? Where PIP differs from traditional equi-based joins is that it is based on a general relation. These joins are commonly known as a Theta Join class. These are usually much harder to execute and require the end user to help the system. While we are starting from a disadvantageous position, we can still achieve the desired performance.

Spatial indices (PIP as a pseudo-equivalence)

Is there a way to make PIP an equivalence relationship? Strictly speaking, no. However, in practice, we can make PIP approach the efficiency of an equivalence relation if we employ spatial indexing techniques.

Spatial indices help us index coordinate space in an efficient way by logically grouping geometries that are close to one another in said space. We achieve this by uniquely associating a point in the coordinate system to an index ID. These systems allow us to represent reference space at different levels of detail, or simply, a different resolution. In addition, geospatial index systems are hierarchical systems; this means that there is a well-defined parent-child relationship between indices on different levels of representation.

How does this help us? If we assign to each geometry an index to which it belongs, we can use index ID to index ID equivalence as an equivalence relation proxy. We will perform PIP (or any other geometry-based relation) only on geometries that belong to the same indices.

It is important to note that while POINT geometries belong to one and only one index, all other geometry types, including LINESTRINGs and POLYGONs, may span over a set of indices. This implies that the cost of resolving a PIP relation via index space is `O(k)*O(v)`, where k is the number of indices used to represent the geometry and v is the number of vertices of such geometry. This indicates that we are increasing the price of each comparison by exploding records of complex geometries into multiple index records carrying the same geometry.

Why is this a wise choice? While we are increasing the price of comparing a single pair of geometries, we are avoiding a full Cartesian Product, our archnemesis in large-scale geospatial joins. As we will show in more detail later, index ID to index ID join will allow us to skip large amounts of unnecessary comparisons.

Lastly, data sources that contain complex geometries do not evolve as fast as do point-wise data sources. Complex geometries usually represent regions, areas of interest, buildings, etc., and these concepts have a fairly stable timeline. Objects that change over time change rarely, and objects that change are relatively few. This means that while we do spend extra time to preprocess complex geometries, for the majority of them, this preprocessing is a one-off event. This approach is still applicable even for frequently updated data; the amount of data we can skip when joining via index ID to index ID relationship outweighs the increased number of rows used to represent a single geometry.

The BNG Index System

The BNG is a local coordinate reference (CRS) system (EPSG:27700) established in 1936 and designed for national mapping that covers Great Britain. Unlike global CRS', BNG has been fitted and shaped to the landmass of Great Britain, projecting coordinates onto a flat, regular square grid with an origin (0, 0) to the southwest of the Isles of Scilly.

Within the grid bounds, geographic grid references (or indices) are used to identify grid squares at different resolutions expressed in meters which can be translated from and to BNG easting (x) and northing (y) coordinates. Given the location of the grid origin, easting and northing values are always positive. BNG serves as the primary reference system for all OS location data captured under their national mapping public task and, therefore, has been widely adopted by public and private users of OS data operating within Great Britain.

Each grid square can be represented as a polygon geometry where the length of each side is equal to the resolution of the grid reference. This makes BNG a much easier starting point for geospatial data partitioning strategies. We are starting with a square as a building block, and it will make a lot of the starting considerations simple while not losing on the generalization of the approach.

By convention, BNG grid references are expressed as strings, using the letters and coordinates of the southwest corner of a given grid square quoted to a particular resolution. The first two characters of any reference are letters (prefixes) (e.g., TQ) identifying one of the 91 grid squares measuring 100.000m (100km) across. Only 55 of the 91 100km grid squares cover some landmass within Great Britain. The remainder of these squares falls into British waters.

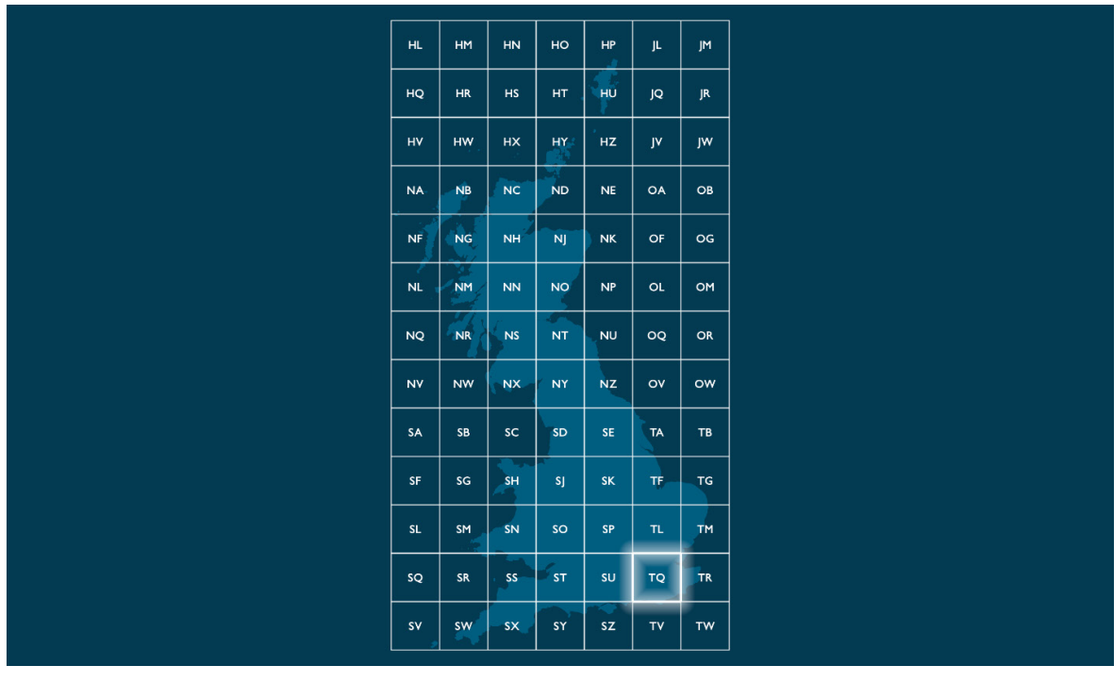


Diagram B

References identifying more granular grid resolutions below 100km will have additional x and y integer values appended after the two letters locating a child grid square within the parent grid square hierarchy. Child squares are numbered from 0 to 9 from the lower-left (southwest) corner, in an easterly (x) and northerly (y) direction.

Diagram C

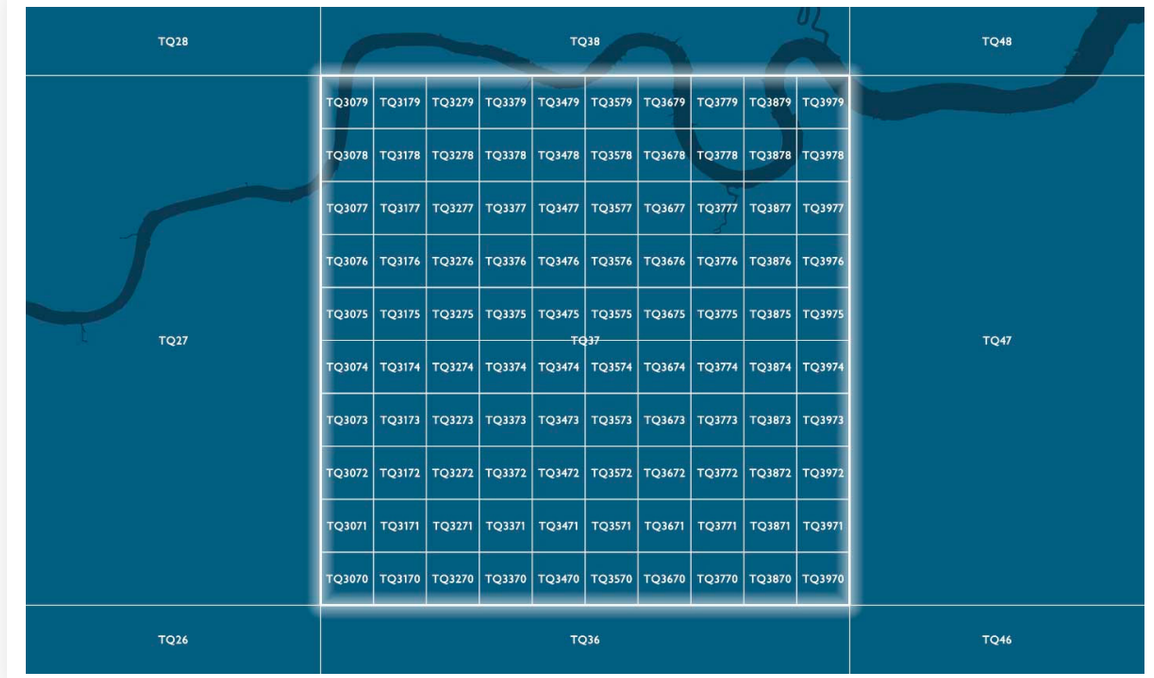


Diagram D

Why BNG?

Whilst there are alternative global index systems that we could have adopted for this work, we chose to use BNG because:

- The BNG system is native to OS's geospatial data collection, with almost all OS data referenced against the BNG CRS (EPSG:27700). This includes OS aerial imagery tiles and other raster data sets, such as Digital Terrain Models (DTMs) and Digital Surface Models (DSMs).
- The use of BNG enables the efficient retrieval and colocation of vector and raster data for analysis, including the clipping or masking of raster data for deriving training patches for deep learning applications, as an example
- Using BNG avoids the costly transformation to the World Geodetic System 1984 (WGS-84) (EPSG:4326) or European Terrestrial Reference System 1989 (ETRS89) (EPSG:4258) CRSs via the OSTN15 transformation grid. Different CRSs realize their model of the Earth using different parameters, and a global system (e.g., WGS84) will show an offset when compared to a local system (e.g., BNG). The true cost of this conversion is reflected in the fact that OS published OSTN15, a 15MB corrections file containing approximately 1.75 million parameters to transform accurately between satellite-derived coordinates and BNG coordinates.

Due to the GB-local nature of the problems OS is trying to solve, BNG is a natural choice. In the case of a more global context, we should switch our focus on H3 or S2 as more suitable global alternatives.

BNG as a spatial partitioning strategy

A spatial partitioning strategy defines an approach to segmenting geospatial data into non-overlapping regions. BNG grid squares at different resolutions provide the non-overlapping regions across Great Britain in this context. By retrieving the BNG indices, which cover geometries, we can use the indices attribute as a join key to collocate rows and then only test a spatial predicate within those collocated rows (e.g., does geometry A intersect geometry B or does geometry A contain geometry B?).

This is very important! Splitting the original data into geospatially collocated portions of data makes our problem "embarrassingly parallel," and, therefore, very suitable for Spark/PySpark. We can send different chunks of data to different machines and only compare local portions of the data that are likely to join one to another. There is little point in checking if a building in London contains an address in Manchester. Geospatial indices are our way to convey this intuition to the machine.

The baseline

We used Python and PySpark to bring our solution to life. OS provided the logic for converting the pair of coordinates provided as eastings and northings to a unique BNG index ID. Lastly, to ensure an unbiased output, we used a randomized data set of points and a randomized data set of polygons; 10 million points were scattered all over the territory of GB, 1 million polygons were scattered in the same manner. To generate such a set of polygonal data, we have loaded a GeoJSON set into a Spark DataFrame, and we have used a random function in conjunction with a generator function (*explode*) to generate an unbiased data set. Due to randomness introduced in the data, one should expect that the relationship between points and polygons is many-to-many.

The baseline algorithm we used for our considerations is the naive join that would result in the unoptimized theta join. This approach will, at the execution time, be evaluated as a Broadcasted Nested Loop Join.

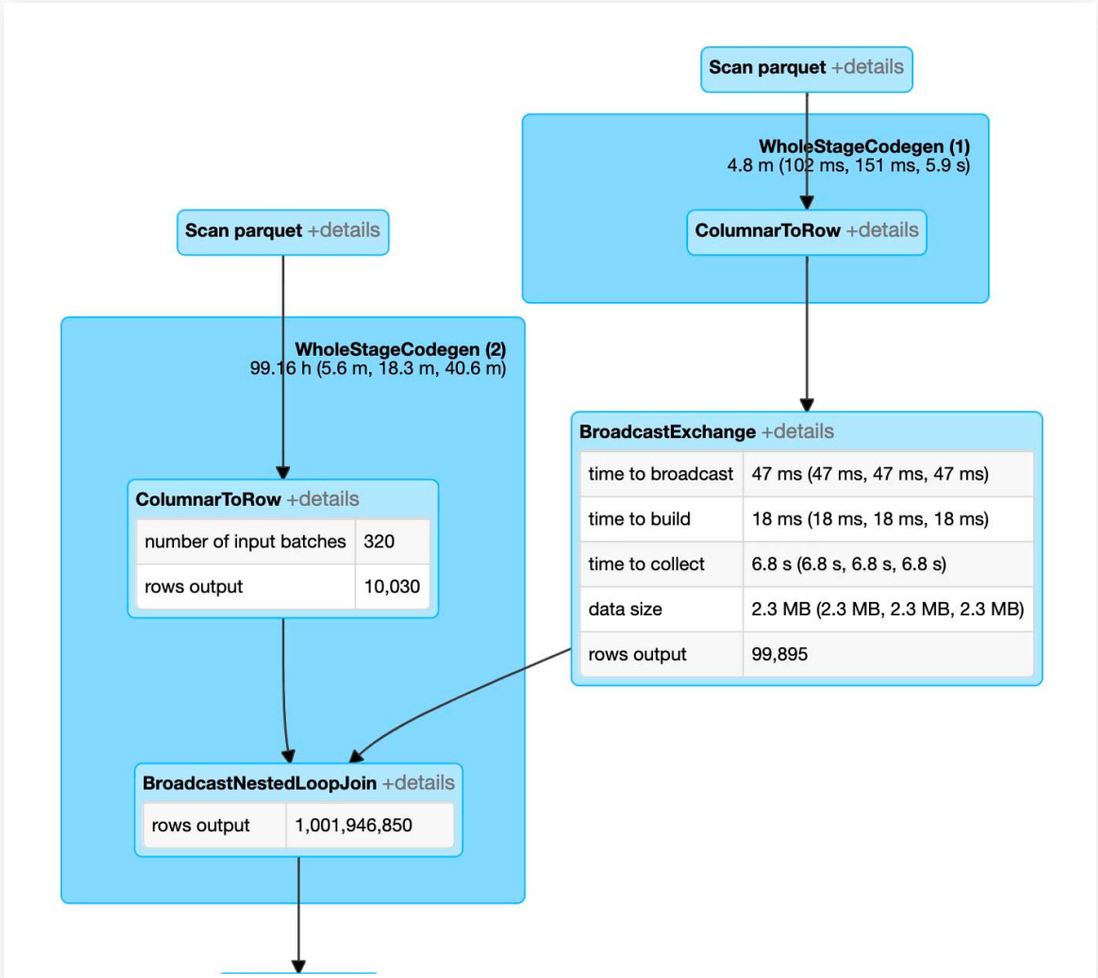


Diagram E

The broadcast nested loop join runs very slowly. And the reason for this is the fact it is evaluated similarly to a Cartesian join. Each of the point-polygon pairs is evaluated against a PIP relation before the join is resolved. The outcome is that we require 1 billion comparisons for 100 thousand points to be joined to 10 thousand polygons. Note that neither of these data sets is large enough to be called big data.

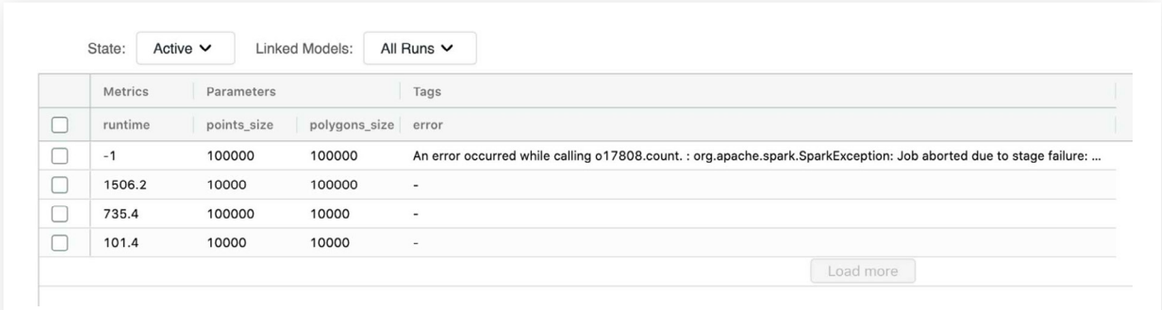


Diagram F

We used MLflow to conduct a series of naive joins to evaluate the baseline performance we are trying to outperform. For the naive approach, the largest join we were able to successfully execute was 10 thousand points to 100 thousand polygons. Any further increase in data volume resulted in our Spark jobs failing without producing the desired outputs. These failures were caused by the unoptimized nature of the workloads we were trying to run.

Let's frame our problem

What if we represented all of our geometries, no matter their shape, with a corresponding BNG-aligned bounding box? A bounding box is a rectangular polygon that can fit the entirety of the original geometry within. And what if we represented said bounding box as a set of BNG indices at a given resolution that together covers the same area?

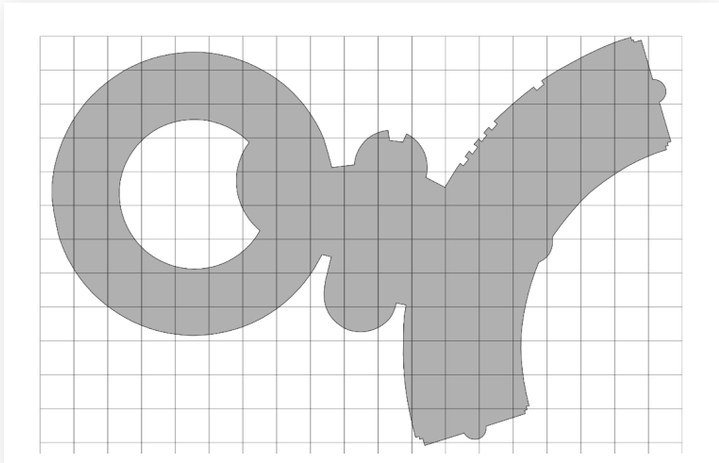


Diagram G

Now we can execute our joins via a more optimized theta join. We will only check whether a point is inside the polygon via PIP relation if a point falls into one of the BNG indices that are used to represent the polygon. This reduces our join effort by multiple orders of magnitude.

In order to produce the said set of BNG indices, we have used the following code; note that the `bng_to_geom, coords_to_bng and bng_get_resolution` functions are not provided with this blog.

```
from shapely.geometry import box

#auxiliary function to retrieve the first neighbours
#of a BNG index cell to the right
def next_horizontal(bng_index, resolution):
  x, y = bng_to_geom(bng_index)
  return coords_to_bng(x+resolution, y, resolution)

#auxiliary function to retrieve the first neighbours
#of a BNG index cell to the bottom
def next_vertical(bng_index, resolution):
  x, y = bng_to_geom(bng_index)
  return coords_to_bng(x, y-resolution, resolution)

#filling function that represents the input geometry as set of
indices
#corresponding to the area of the bounding box of said geometry
def bng_polyfil(polygon, resolution):
  (x1,y1,x2,y2) = polygon.bounds
  bounding_box = box(*polygon.bounds)
  lower_left = coords_to_bng(x1, y2, resolution)
  queue = [lower_left]
  result = set()
  visited = set()
  while queue:
    index = queue.pop()
    index_geom = shapely.wkt.loads(bng_to_geom_grid(index, "WKT"))
    intersection = bounding_box.intersects(index_geom)
    if intersection:
      result.add(index)
      n_h = next_horizontal(index, resolution)
      if n_h not in visited:
        queue.append(n_h)
      n_v = next_vertical(index, resolution)
      if n_v not in visited:
        queue.append(n_v)
    visited.add(index)
  visited = []

  return  result
```

This code ensures that we can represent any shape in a lossless manner. We are using intersects relation between a BNG index candidate and the original geometry to avoid blind spots in representation. Note that a more efficient implementation is possible by using contains relation and a centroid point; that approach is only viable if false positives and false negatives are acceptable. We assume the existence of the *bng_to_geom* function that given a BNG index ID can produce a geometry representation, the *bng_get_resolution* function that given a BNG index ID determines the selected resolution and *coords_to_bng* function that given the coordinates returns a BNG index ID.

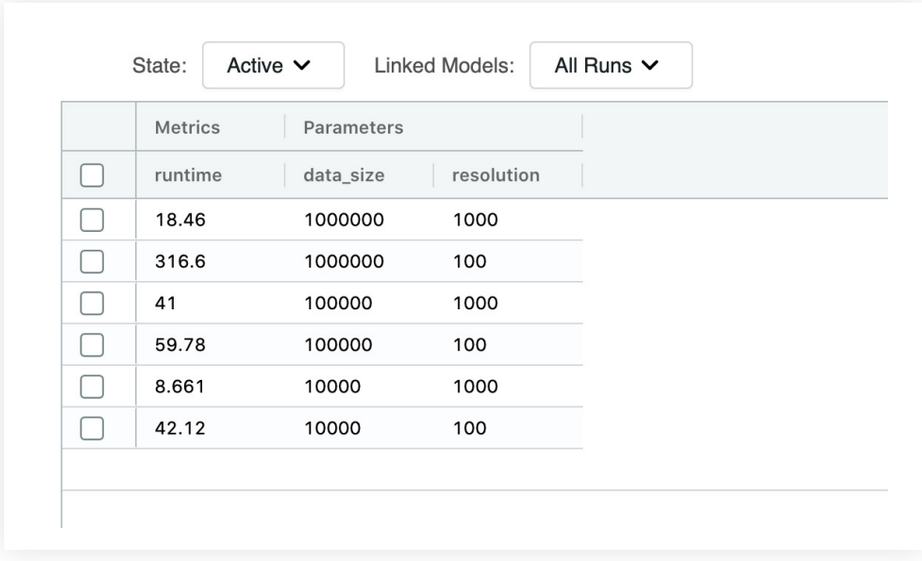


Diagram H

We have run our polygon bounding box representation for different resolutions of the BNG index system and for different data set sizes. Note that running this process was failing consistently for resolutions below 100. Resolutions are represented in meters in these outputs. The reason for consistent failures at resolutions below 100m can be found in overrepresentation; some polygons (due to random nature) are much larger than others, and while some polygons would be represented by a set of a dozen indices, other polygons can be represented by thousands of indices, and this can result in a big disparity in compute and memory requirements between partitions in a Spark job that is generating this data.

We have omitted the benchmarks for points data set transformations since this is a relatively simple operation that does not yield any new rows; only a single column is added, and the different resolutions do not affect execution times.

With both sides of the join being represented with their corresponding BNG representations, all we have to do is to execute the adjusted join logic:

```
@udf("boolean")
def pip_filter(poly_wkt, point_x, point_y):
  from shapely import wkt
  from shapely import geometry
  polygon = wkt.loads(poly_wkt)
  point = geometry.Point(point_x, point_y)
  return polygon.contains(point)

def run_bounding_box_join(polygons_path, points_path):
  polygons = spark.read.format("delta").load(polygons_path)
  polygons = polygons.select(
    F.col("id"),
    F.col("wkt_polygon"),
    F.explode(F.col("bng_set")).alias("bng")
  )
  points = spark.read.format("delta").load(points_path)

  return polygons.join(
    points,
    on=["bng"],
    how="inner"
  ).where(pip_filter("wkt_polygon", "eastings", "northings"))
#run an action on the join dataset to evaluate join runtime
run_bounding_box_join(polygons_path, points_path).count()
```

These modifications in our code have resulted in a different Spark execution plan. Spark is now able to first run a sort merge join based on the BNG index ID and vastly reduce the total number of comparisons. In addition, each pair comparison is a string-to-string comparison, which is much shorter than a PIP relationship. This first stage will generate all the join set candidates. We will then perform a PIP relationship test on this set of candidates to resolve the final output. This approach ensures that we limit the number of times we have to run the PIP operation.

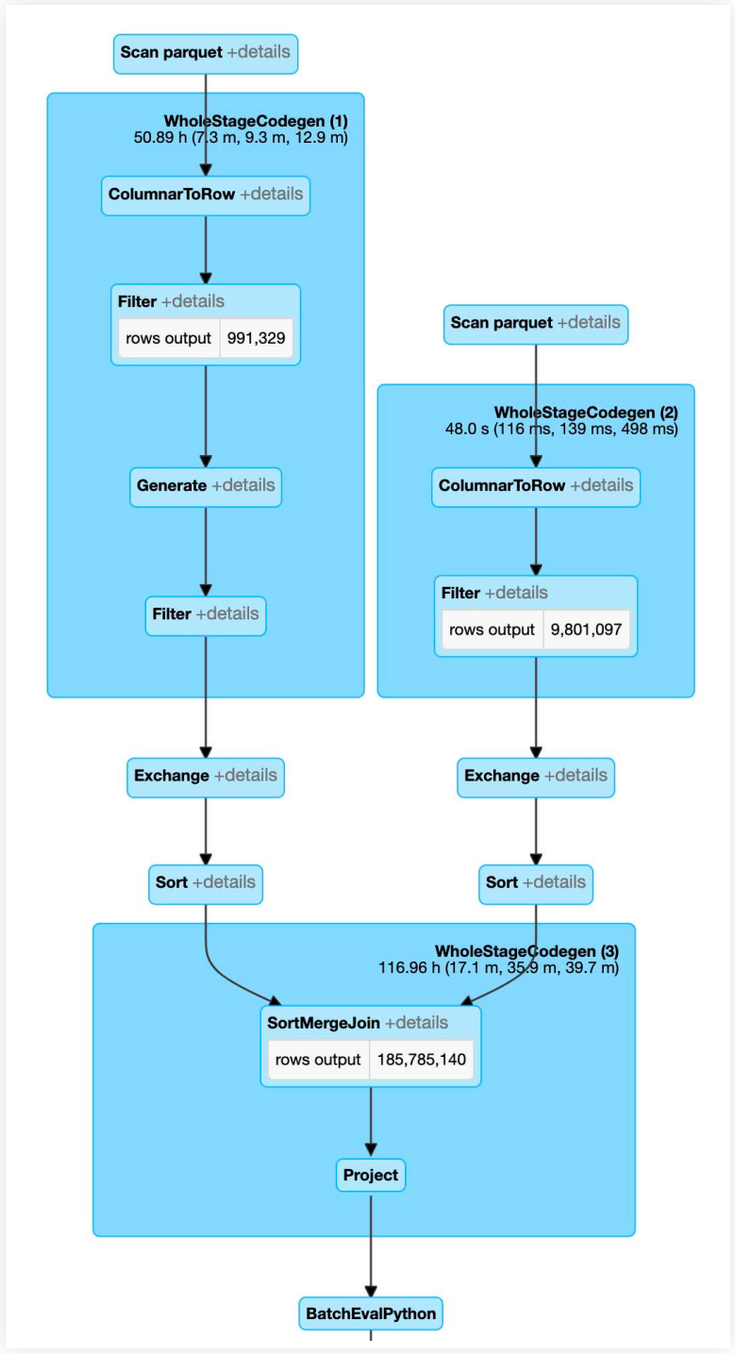


Diagram I

From the execution plan, we can see that Spark is performing a very different set of operations in comparison to the naive approach. Most notably, Spark is now executing Sort Merge Join instead of Broadcast Nested Loop Join, which is bringing a lot of efficiencies. We are now performing about 186 million PIP operations instead of a billion. This alone is allowing us to run much larger joins with better response time whilst avoiding any breaking failures that we have experienced in the naive approach.

State: Active Linked Models: All Runs

	Metrics	Parameters		
	↓ runtime	points_size	polygons_size	resolution
	2548.7	10000000	1000000	100
	2023.9	1000000	1000000	100
	202.7	10000000	100000	100
	134.6	1000000	100000	100
	8.518	100000	1000000	100
	7.344	10000000	10000	100
	5.895	10000	1000000	100
	5.063	10000	10000	100
	4.933	1000000	10000	100
	3.367	100000	100000	100
	1.429	100000	10000	100
	1.393	10000	100000	100

Diagram J

This simple yet effective optimization has enabled us to run a PIP join between 10 million points and 1 million polygons in about 2,500 seconds. If we compare that to the baseline execution times, the largest join we were able to successfully execute was 10 thousand points to 100 thousand polygons, and even that join required about 1,500 seconds on the same hardware.

Divide and conquer

Being able to run joins between data sets in the million rows domain is great; however, our largest benchmark join took almost 45 minutes (2,500 seconds). And in the world where we want to run ad hoc analytics over large volumes of geospatial data, these execution times are simply too slow.

We need to further optimize our approach. The first candidate for optimization is our bounding box representation. If we are representing polygons via bounding boxes, we include too many false positive indices, i.e., indices that do not overlap at all with the original geometry.

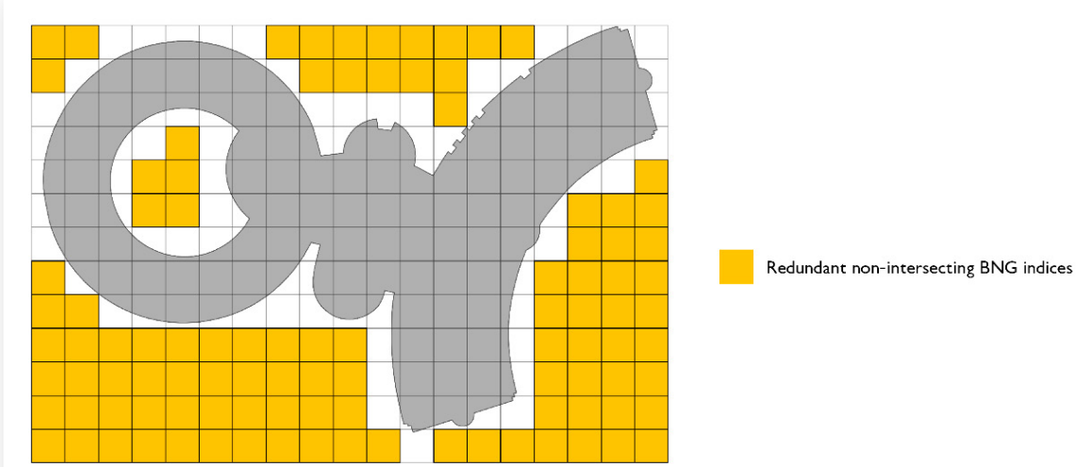


Diagram K

The way to optimize that portion of the code is to simply use intersects function call in our polyfill method on the original geometry.

```
def k_ring(bng_index):
  x, y = bng_to_geom(bng_index)
  increment = bng_get_resolution(bng_index)
  neighbours = [
    [x-increment, y+increment], [x, y+increment], [x+increment,
y+increment],
    [x-increment, y], [x+increment, y],
    [x-increment, y-increment], [x, y-increment], [x+increment,
y-increment]
  ]
  neighbours = [coords_to_bng(i[0], i[1], increment) for i in
neighbours]
  return neighbours

def bng_polyfil(polygon, resolution):
  from shapely.geometry import box
  start = get_starting_point(polygon, resolution)
  queue = k_ring(start)
  result = set()
  visited = set()
  while queue:
    index = queue.pop()
    if polygon.intersects(shapely.wkt.loads(bng_to_geom_grid(index,
"WKT"))):
      result.add(index)
      for n in k_ring(index):
        if n not in visited and n not in queue:
          queue.append(n)
    visited.add(index)
  visited = []

  return  result
```

This optimization, while increasing the cost by utilizing intersects call, will result in smaller resulting index sets and will make our joins run faster due to the smaller join surface.

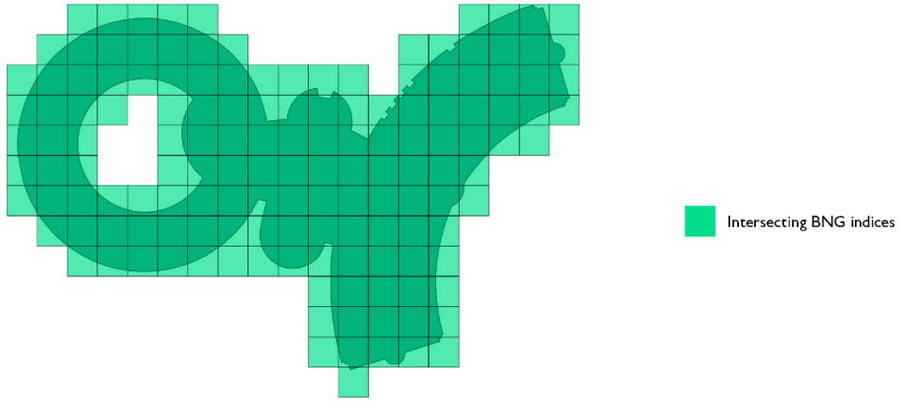


Diagram L

The second optimization we can employ is splitting the representation into two sets of indices. Not all indices are equal in our representation. Indices that touch the border of the polygon require a PIP filtering after an index to index join. Indices that do not touch the border and belong to the representation of the polygon do not require any additional filtering. Any point that falls into such an index definitely belongs to the polygon and, in such cases, we can skip the PIP operation.

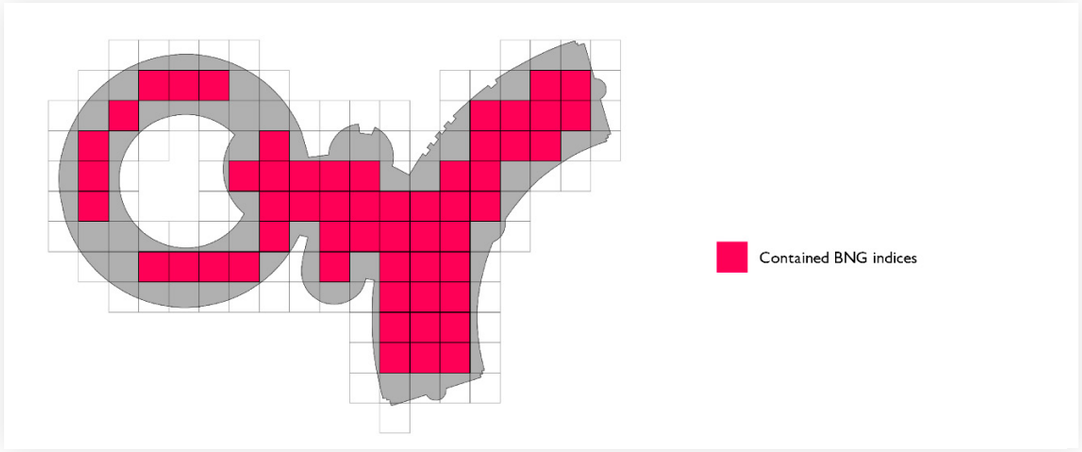


Diagram M

The third and final optimization we can implement is the mosaic approach. Instead of associating the complete original geometry with each index that belongs to the set of indices that touch the polygon border (border set), we can only keep track of the section of interest. If we intersect the geometry that represents the index in question and the polygon, we get the local representation of the polygon; only that portion of the original polygon is relevant over the area of the index in question. We refer to these pieces as polygon chips.

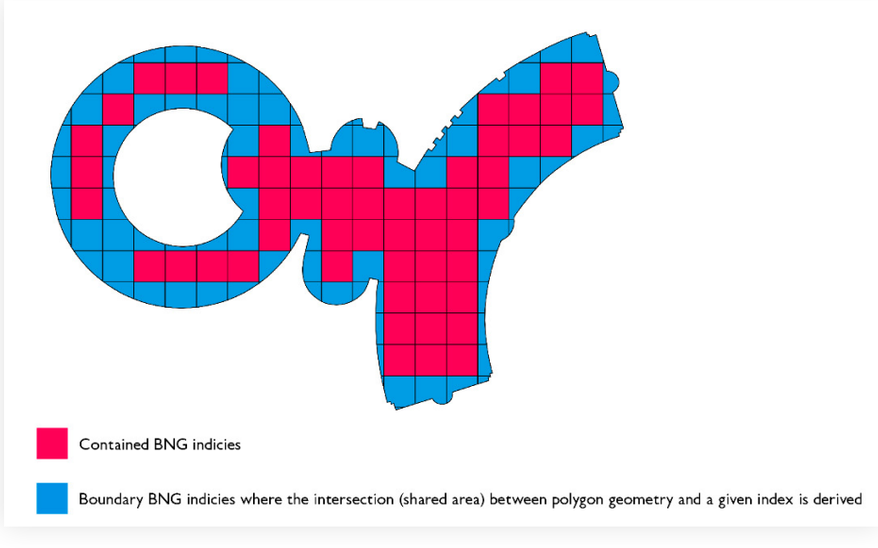


Diagram N

Polygon chips serve two purposes from the optimization perspective. Firstly, they vastly improve the efficiency of the PIP filter that occurs after the index-to-index join is executed. This is due to the fact that the ray tracing algorithm runs in O(v) complexity and individual chips on average have an order of magnitude fewer vertices than the original geometry. Secondly, the representation of chips is much smaller than the original geometry, and as a result of this, we are shuffling much less data as part of the shuffle stage in our sort merge join stage.

Putting all of these together yields the following code:

```
def add_children(queue, visited, index):
  for n in k_ring(index):
    if n not in visited and n not in queue:
      queue.append(n)
  return queue

def bng_polyfil(polygon, resolution):
  start = get_starting_point(polygon, resolution)
  queue = k_ring(start)
  result = set()
  visited = set()
  while queue:
    index = queue.pop()
    index_geom = shapely.wkt.loads(bng_to_geom_grid(index, "WKT"))
    intersection = polygon.intersection(index_geom)
    if intersection.equals(index_geom):
      result.add((index, False, "POLYGON EMPTY"))
      queue = add_children(queue, visited, index)
    elif "EMPTY" not in intersection.to_wkt():
      result.add((index, True, intersection.to_wkt()))
      queue = add_children(queue, visited, index)
    visited.add(index)
  visited = []

  return  result
```

This code is very similar to the original bounding box methods, and we have only done a few minor changes to make sure we are not duplicating some portions of the code; hence, we have isolated the *add_children* helper method.

State: Active Linked Models: All Runs

Metrics			Parameters	Tags
runtime	data_size	resolution	error	
-	-	-	-	
42.25	1000000	1000	-	
600.2	1000000	100	-	
-1	1000000	10	An error occurred while calling o43502.save. : org.apache.spark.SparkException: Job aborted. a...	
8.771	100000	1000	-	
75.66	100000	100	-	
16661.6	100000	10	-	
25.99	10000	1000	-	
15.56	10000	100	-	
2579.8	10000	10	-	

Load more

Diagram O

We have performed the same data generation benchmarking as we have done for our bounding box polygon representation. One thing we found in common with the original approach is that resolutions below 100m were causing over-representation of the polygons. In this case, we were, however, able to generate data up to 100 thousand polygons on a resolution of 10m — although, granted, the runtime of such data generation process was too slow to be considered for production workloads.

At the resolution of 100m, we have got some very promising results; it took about 600 seconds to generate and write out the data set of 1 million polygons. For reference, it took about 300 seconds to do the same for the bounding box approach. Bounding box was a simpler procedure, and we are adding some processing time in the data preparation stage. Can we justify this investment?

Mosaics are pretty (fast!)

We have run the same benchmark for PIP joins using our mosaic data. We have adapted our join logic slightly in order to make sure our border set and core set of indices are both utilized correctly and in the most efficient way.

```
def run_polyfill_chipping_join(polygons_path, points_path):
  polygons = spark.read.format("delta").load(polygons_path)
  polygons = polygons.select(
    F.col("id"),
    F.explode(F.col("bng_set")).alias("bng")
  ).select(
    F.col("id"),
    F.col("bng.*")
  )
  right = spark.read.format("delta").load(right_path)

  return polygons.join(
    right,
    on=["bng"],
    how="inner"
  ).where(
    ~F.col("is_dirty") |
    pip_filter("wkt_chip", "eastings", "northings")
  )

#run an action to execute the join
run_polyfill_chipping_join(polygons_path, points_path).count()
```

is_dirty column is introduced by our polyfill method. Any index that touches the border of the original geometry will be marked as dirty (*i.e., is_dirty=True*). These indices will require post-filtering in order to correctly determine if any point that falls into said index is contained within the comparing geometry. It is crucial that *is_dirty* filtering happens first before the *pip_fiter* call because the logical operators in Spark have a short-circuiting capability; if the first part of the logical expression is true, the second part won't execute.

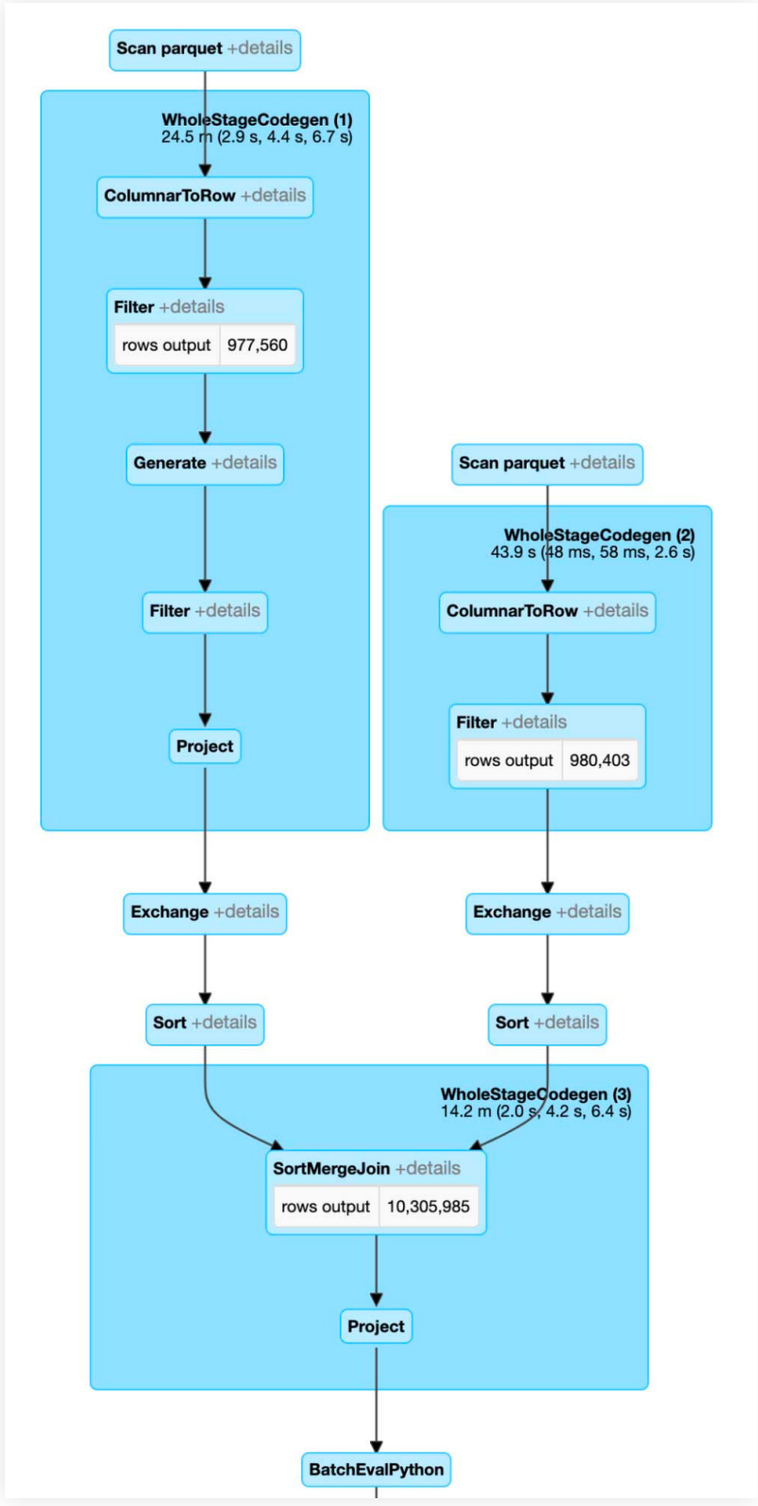


Diagram P

This code will yield a much more efficient execution plan in Spark. Due to better representation in the index space, our join surfaces are much smaller. In addition, our post-filters benefit from 2 set representation and mosaic splitting of the geometries.

State: Active Linked Models: All Runs

	Metrics	Parameters	
	↓ runtime	points_size	polygons_size
	36.44	10000000	1000000
	12.44	1000000	1000000
	11.43	10000	1000000
	8.779	100000	1000000
	7.911	100000	100000
	7.063	10000	100000
	4.599	10000000	100000
	3.446	100000	10000
	3.085	10000	10000
	2.356	1000000	100000
	2.177	10000000	10000
	1.781	1000000	10000

Diagram Q

We can finally quantify our efforts. A PIP type join between 10 million points and 1 million polygons via our new mosaic approach has been executed in 37 seconds. To bring this into context, the bounding box equivalent join at the same index resolution was executed in 2,549 seconds. This results in a 69x improvement in runtime.

This improvement purely focuses on the serving runtime. If we include the preparation times, which were 600 seconds for the mosaic approach and 317 seconds for the bounding box approach, we have the total adjusted performance improvement of 4.5x.

The total potential of these improvements largely depends on how often you are updating your geometrical data versus how often you query it.

A general approach

In this post, we have focused on point in polygon (PIP) joins using the British National Grid (BNG) as the reference index system. However, the approach is more general than that. The same optimizations can be adapted to any hierarchical geospatial system. The difference is that of the chip shapes and available resolutions. Furthermore, the same optimizations can help you scale up theta joins between two complex geometries, such as large volume polygon intersection joins.

Our focus remained on a PySpark first approach, and we have consciously avoided introducing any third-party frameworks. We believe that ensures a low barrier to consume our solution, and it is custom-tailored primarily to Python users.

The solution has proved that with a few creative optimizations, we can achieve up to 70 times the performance improvements of the bounding box approach with a minimal increase in the preprocessing investment.

We have brought large-scale PIP joins into the execution time domain of seconds, and we have unlocked the ad hoc analytical capabilities against such data.

CHAPTER 13:

HSBC Augments SIEM for Cybersecurity at Cloud Scale

By **Michael Ortega** and **Monzy Merza**
October 11, 2021

Over the last decade, security incident and event management tools (SIEMs) have become a standard in enterprise security operations. SIEMs have always had their detractors. But the explosion of cloud footprints is prompting the question, are SIEMs the right strategy in the cloud-scale world? Security leaders from HSBC don't think so. In a recent talk, *Empower Splunk and Other SIEMs with the Databricks Lakehouse for Cybersecurity*, HSBC highlighted the limitations of legacy SIEMs and how the Databricks Lakehouse Platform is transforming cyberdefense. With $3 trillion in assets, HSBC's talk warrants some exploration.

In this blog post, we will discuss the changing IT and cyber-attack threat landscape, the benefits of SIEMs, the merits of the Databricks Lakehouse and why SIEM + Lakehouse is becoming the new strategy for security operations teams. Of course, we will talk about my favorite SIEM! But I warn you, this isn't a post about critiquing "legacy technologies built for an on-prem world." This post is about how security operations teams can arm themselves to best defend their enterprises against advanced persistent threats.

The enterprise tech footprint

Some call it cloud-first and others call it cloud-smart. Either way, it is generally accepted that every organization is involved in some sort of cloud transformation or evaluation — even in the public sector, where onboarding technology isn't a light decision. As a result, the main U.S. cloud service providers all rank within the top 5 largest market cap companies in the world. As tech footprints are migrating to the cloud, so are the requirements for cybersecurity teams. Detection, investigation and threat hunting practices are all challenged by the complexity of the new footprints, as well as the massive volumes of data. According to IBM, it takes 280 days on average to detect and contain a security breach. According to HSBC's talk at Data + AI Summit, 280 days would mean over a petabyte of data — just for network and EDR (endpoint threat detection and response) data sources.

When an organization needs this much data for detection and response, what are they to do? Many enterprises want to keep the cloud data in the cloud. But what about from one cloud to the other? I spoke to one large financial services institution this week who said, "We pay over a $1 million in egress cost to our cloud provider." Why? Because their current SIEM tool is on one cloud service and their largest data producers are on another. Their SIEM isn't multicloud. And over the years, they have built complicated transport pipelines to get data from one cloud provider to the other. Complications like this have warped their expectations from technology. For example, they consider 5-minute delays in data to be real-time. I present this here as a reality of what modern enterprises are confronted with — I am sure the group I spoke with is not the only one with this complication.

Security analytics in the cloud world

The cloud terrain is really messing with every security operations team's m.o. What was called big data 10 years ago is puny data by today's cloud standards. With the scale of today's network traffic, gigabytes are now petabytes, and what used to take months to generate now happens in hours. The stacks are new and security teams are having to learn them. Mundane tasks like, "have we seen these IPs before" are turning into hours or days-long searches in SIEM and logging tools. Slightly more sophisticated contextualization tasks, like adding the user's name to network events, are turning into near impossible ordeals. And if one wants to do streaming enrichments of external threat intelligence at terabytes of data per day — good luck — hope you have a small army and a deep pocket. And we haven't even gotten to anomaly detection or threat hunting use cases. This is by no means a jab at SEIMs. In reality, the terrain has changed and it's time to adapt. Security teams need the best tools for the job.

What capabilities do security teams need in the cloud world? First and foremost, an open platform that can be integrated with the IT and security tool chains and does not require you to provide your data to a proprietary data store. Another critical factor is a multicloud platform, so it can run on the clouds (plural) of your choice. Additionally, a scalable and highly performant analytics platform, where compute and storage are decoupled that can support end-to-end streaming AND batch processing. And finally, a unified platform to empower data scientists, data engineers, SOC analysts and business analysts — all data people. These are the capabilities of the Databricks Lakehouse Platform.

The SaaS and auto-scaling capabilities of Databricks simplify the use of these sophisticated capabilities. Databricks security customers are crunching across petabytes of data in sub ten minutes. One customer is able to collect from 15 million+ endpoints and analyze the threat indicators in under an hour. A global oil and gas producer, paranoid about ransomware, runs multiple analytics and contextualizes every single powershell execution in their environment — analysts only see high confidence alerts.

Lakehouse + SIEM : The pattern for cloud-scale security operations

George Webster, Head of Cybersecurity Sciences and Analytics at HSBC, describes the lakehouse + SIEM is THE pattern for security operations. It leverages the strengths of the two components: a lakehouse architecture for multicloud-native storage and analytics, and SIEM for security operations workflows. For Databricks customers, there are two general patterns for this integration. But they are both underpinned by what Webster calls *The Cybersecurity Data Lake With Lakehouse*.

The first pattern: The lakehouse stores all the data for the maximum retention period. A subset of the data is sent to the SIEM and stored for a fraction of the time. This pattern has the advantage that analysts can query near-term data using the SIEM while having the ability to do historical analysis and more sophisticated analytics in Databricks. And manage any licensing or storage costs for the SIEM deployment.

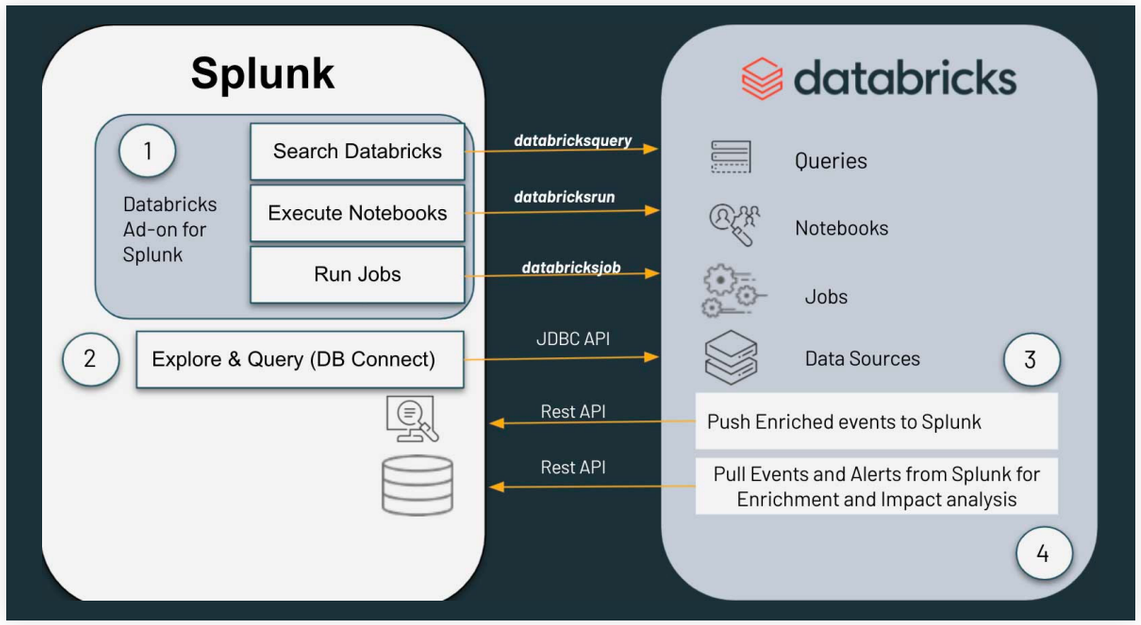
The second pattern is to send the highest volume data sources to Databricks, (e.g., cloud native logs, endpoint threat detection and response logs, DNS data and network events). Comparatively low volume data sources go to the SIEM, (e.g., alerts, email logs and vulnerability scan data). This pattern enables Tier 1 analysts to quickly handle high-priority alerts in the SIEM. Threat hunt teams and investigators can leverage the advanced analytical capabilities of Databricks. This pattern has a cost benefit of offloading processing, ingestion and storage off of the SIEM.

Integrating the lakehouse with Splunk

What would a working example look like? Because of customer demand, the Databricks Cybersecurity SME team created the Databricks add-on for Splunk. The add-on allows security analysts to run Databricks queries and notebooks from Splunk and receive the results back into Splunk. A companion Databricks notebook enables Databricks to query Splunk, get Splunk results and forward events and results to Splunk from Databricks.

With these two capabilities, analysts on the Splunk search bar can interact with Databricks without leaving the Splunk UI. And Splunk search builders or dashboards can include Databricks as part of their searches. But what's most exciting is that security teams can create bidirectional, analytical automation pipelines between Splunk and Databricks. For example, if there is an alert in Splunk, Splunk can automatically search Databricks for related events, and then add the results to an alerts index or a dashboard or a subsequent search. Or conversely, a Databricks notebook code block can query Splunk and use the results as inputs to subsequent code blocks.

With this reference architecture, organizations can maintain their current processes and procedures, while modernizing their infrastructure, and become multicloud native to meet the cybersecurity risks of their expanding digital footprints.



Achieving scale, speed, security and collaboration

Since partnering with Databricks, HSBC has reduced costs, accelerated threat detection and response, and improved their security posture. Not only can the financial institution process all of their required data, but they've increased online query retention from just days to many months at the PB scale. The gap between an attacker's speed and HSBC's ability to detect malicious activity and conduct an investigation is closing. By performing advanced analytics at the pace and speed of adversaries, HSBC is closer to their goal of moving faster than bad actors.

As a result of data retention capabilities, the scope of HSBC threat hunts has expanded considerably. HSBC is now able to execute 2-3x more threat hunts per analyst, without the limitations of hardware. Through Databricks notebooks, hunts are reusable and self-documenting, which keeps historical data intact for future hunts. This information, as well as investigation and threat hunting lifecycles, can now be shared between HSBC teams to iterate and automate threat detection. With efficiency, speed and machine learning/artificial intelligence innovation now available, HSBC is able to streamline costs, reallocate resources, and better protect their business-critical data.

What's next

Watch *Empower Splunk and Other SIEMs With the Databricks Lakehouse for Cybersecurity* to hear directly from HSBC and Databricks about how they are addressing their cybersecurity requirements.

> **Learn more about the Databricks add-on for Splunk.**

CHAPTER 14:

How the City of Spokane Improved Data Quality While Lowering Costs

By **Clinton Ford**
January 21, 2021

The challenges of data quality

One of the most common issues our customers face is maintaining high data quality standards, especially as they rapidly increase the volume of data they process, analyze and publish. Data validation, data transformation and de-identification can be complex and time-consuming. As data volumes grow, new downstream use cases and applications emerge, and expectations of timely delivery of high-quality data increase the importance of fast and reliable data transformation, validation, de-duplication and error correction. Over time, a wide variety of data sources and types add processing overhead and increase the risk of an error being introduced into the growing number of data pipelines as both streaming and batch data are merged, validated and analyzed.

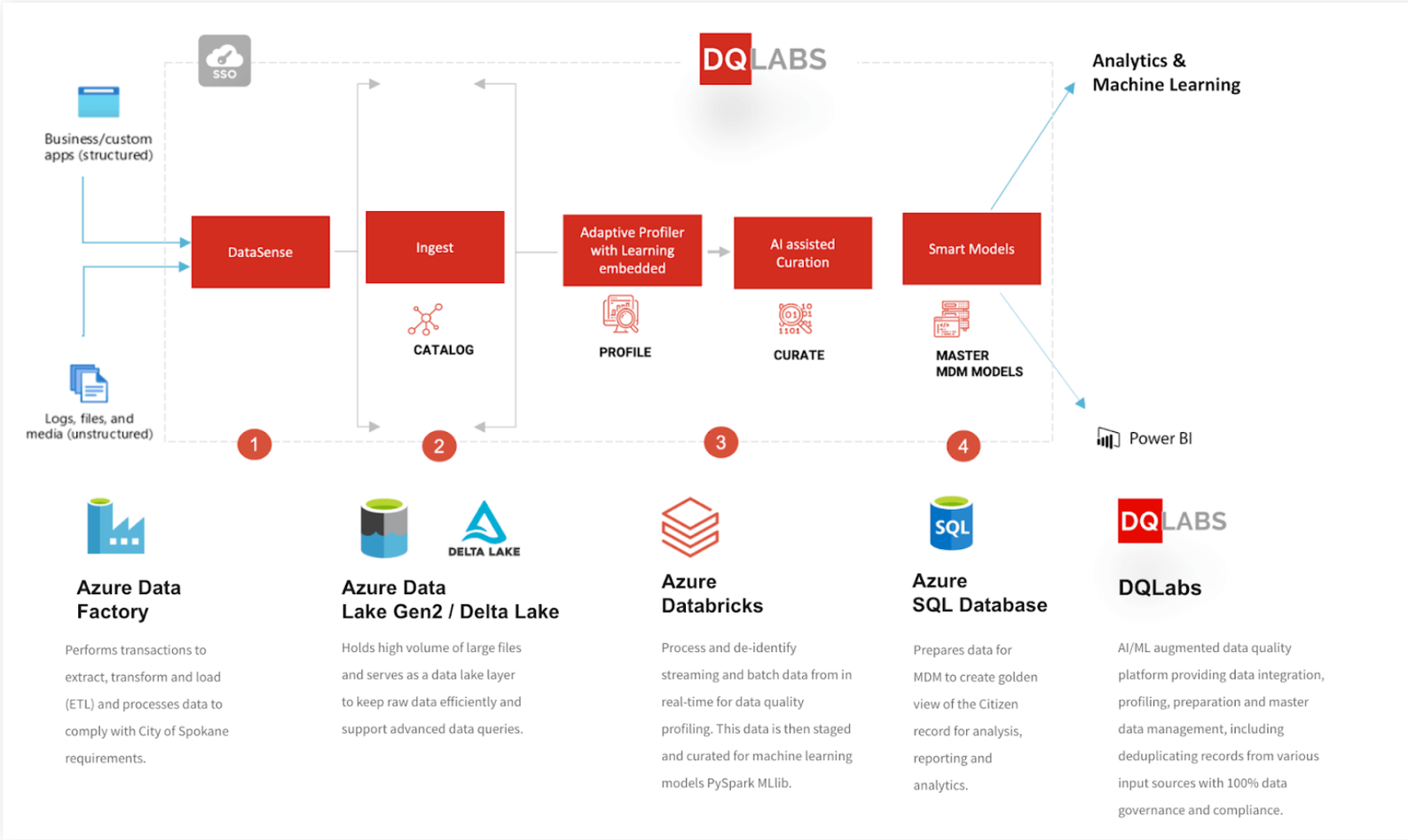
City-scale data processing

The City of Spokane, located in Washington state, is committed to providing information that promotes government transparency and accountability and understands firsthand the challenges of data quality. The City of Spokane deals with an enormous amount of critical data that is required for many of its operations, including financial reports, city council meeting agendas and minutes, issued and pending permits, as well as map and geographic information system (GIS) data for road construction, crime reporting and snow removal. With their legacy architecture, it was nearly impossible to obtain operational analytics and real-time reports. They needed a method of publishing and disseminating city data sets from various sources for analytics and reporting purposes through a central location that could efficiently process data to ensure data consistency and quality.

How the City of Spokane improved data quality while lowering costs

To abstract their entire ETL process and achieve consistent data through data quality and master data management services, the City of Spokane leveraged DQLabs and Azure Databricks. They merged a variety of data sources, removed duplicate data and curated the data in Azure Data Lake Storage (ADLS).

> "Transparency and accountability are high priorities for the City of Spokane," said Eric Finch, Chief Innovation and Technology Officer, City of Spokane. "DQLabs and Azure Databricks enable us to deliver a consistent source of cleansed data to address concerns for high-risk populations and to improve public safety and community planning."



City of Spokane ETL/ELT process with DQLabs and Azure Databricks.

Using this joint solution, the City of Spokane increased government transparency and accountability and can provide citizens with information that encourages and invites public participation and feedback. Using the integrated golden record view, data sets became easily accessible to improve reporting and analytics. The result was an 80% reduction in duplicates, significantly improving data quality. With DQLabs and Azure Databricks, the City of Spokane also achieved a 50% lower total cost of ownership (TCO) by reducing the amount of manual labor required to classify, organize, de-identify, de-duplicate and correct incoming data as well as lower costs to maintain and operate their information systems as data volumes increase.

How DQLabs leverages Azure Databricks to improve data quality

"DQLabs is an augmented data quality platform, helping organizations manage data smarter," said Raj Joseph, CEO, DQLabs. "With over two decades of experience in data and data science solutions and products, what I find is that organizations struggle a lot in terms of consolidating data from different locations. Data is commonly stored in different forms and locations, such as PDFs, databases, and other file types scattered across a variety of locations such as on-premises systems, cloud APIs, and third-party systems."

To help customers make sense of their data and answer even simple questions such as, "Is it good?" or "Is it bad?" are far more complicated than organizations ever anticipated. To solve these challenges, DQLabs built an augmented data quality platform. DQLabs helped the City of Spokane to create an automated cloud data architecture using Azure Databricks to process a wide variety of data formats, including JSON and relational databases. They first leveraged Azure Data Factory (ADF) with DQLabs' built-in data integration tools to connect the various data sources and orchestrate the data ingestion at different velocities, for both full and incremental updates.

DQLabs uses Azure Databricks to process and de-identify both streaming and batch data in real time for data quality profiling. This data is then staged and curated for machine learning models PySpark MLlib.

Incoming data is evaluated to understand its semantic type using DQLabs' artificial intelligence (AI) module, DataSense. This helps organizations classify, catalog, and govern their data, including sensitive data, such as personally identifiable information (PII) that includes contact information and social security numbers.

Based on the DataSense classifications, additional checks and custom rules can be applied to ensure data is managed and shared according to the city's guidelines. Data quality scores can be monitored to catch errors quickly. Master data models (MDM) are defined at different levels. For example, contact information can include name, address and phone number.

Refined data are published as golden views for downstream analysis, reporting and analytics. Thanks to DQLabs and Azure Databricks, this process is fast and efficient, putting organizations like the City of Spokane in a leading position to leverage their data for operations, decision-making and future planning.

Get started with DQLabs and Azure Databricks to improve data quality

Learn more about DQLabs by registering for a live event with Databricks, Microsoft, and DQLabs. Get started with Azure Databricks with a Quickstart Lab and this 3-part webinar training series.

CHAPTER 15:

How Thasos Optimized and Scaled Geospatial Workloads With Mosaic on Databricks

This is a collaborative post from Databricks and Thasos. We thank Zachary Warren (Lead Data Engineer) of Thasos for his contributions.

By **Krishanu Nandy** and **Zachary Warren**
October 12, 2022

Customer insights from geospatial data

Thasos is an alternative data intelligence firm that transforms real-time location data from cell phones into actionable business performance insights for customers, which include institutional investors and commercial real-estate firms. Investors, for example, might want to understand metrics like aggregate foot traffic for properties owned by companies within their portfolios or to make new data-driven investment decisions. With billions of phones broadcasting their location throughout the world, Thasos gives their customers a competitive edge with powerful real-time tools for measuring and forecasting economic activity anywhere.

Challenges of scaling geospatial workloads

To derive actionable insights from cell phone ping data (a time series of points defined by a latitude and longitude pair), Thasos created, maintains and manages a vast collection of verified geofences — a virtual boundary or perimeter around an area of interest with metadata about each particular boundary. In geospatial terms, a simple geofence can be defined as a polygon, a sequence of latitude and longitude pairs that define the vertices of the polygon. The image below (figure 1) is a visualization of example geofences, and the ticker symbols a piece of metadata that is critical for the intelligence products that Thasos develops.

Figure 1. Aerial photograph of an open-air mall overlayed with geofence polygons (red) corresponding to the store location and parking lots of various companies of interest. Ticker symbols (text boxes) are representative of metadata associated with geofence polygons.

Generating insights requires figuring out which cell phone location points fall within which geofence polygon and is called a point in polygon (PIP) join. While the challenge of scaling PIP joins has been discussed in detail in a previous blog post, we will reiterate some key points:

- A naive approach to the PIP problem is to directly compare each of n points to m polygons (resulting in a Cartesian product)
- The individual comparisons of a point and polygon pair are a function of the complexity of the polygon; the more vertices v the polygon has, the more expensive it is to determine whether or not an arbitrary point is contained within the polygon
- Putting these together, the complexity is $O(n*m)*O(v)$, which scales very poorly. What this complexity means is that the naive approach can get very expensive as the number of the points and polygons, and the complexity of polygons, increases. Furthermore, there may be variance in the number of polygon vertices across rows of data (e.g., figure 1). Consequently the $O(v)$ multiplier across partitions of data may vary, which can cause bottlenecks in distributed processing systems.

To overcome this scaling challenge, we leverage spatial index systems to reframe the PIP join as an equivalence relationship, which is computationally much less expensive than the naive approach. Geospatial index systems such as H3 and S2 create hierarchical tilings or mosaics of Earth's surface in which each tile (or *cell*) is assigned a unique index ID that logically groups geometries close to each other. A point is associated with a single cell (a single index ID), whereas a polygon may span over a set of cells (multiple index IDs) that are either wholly or partially contained within the polygon (figure 2). For cells that are partially contained within a polygon, a local representation of the polygon is derived by intersecting the cell geometry with the polygon geometry yielding what is called a "chip."

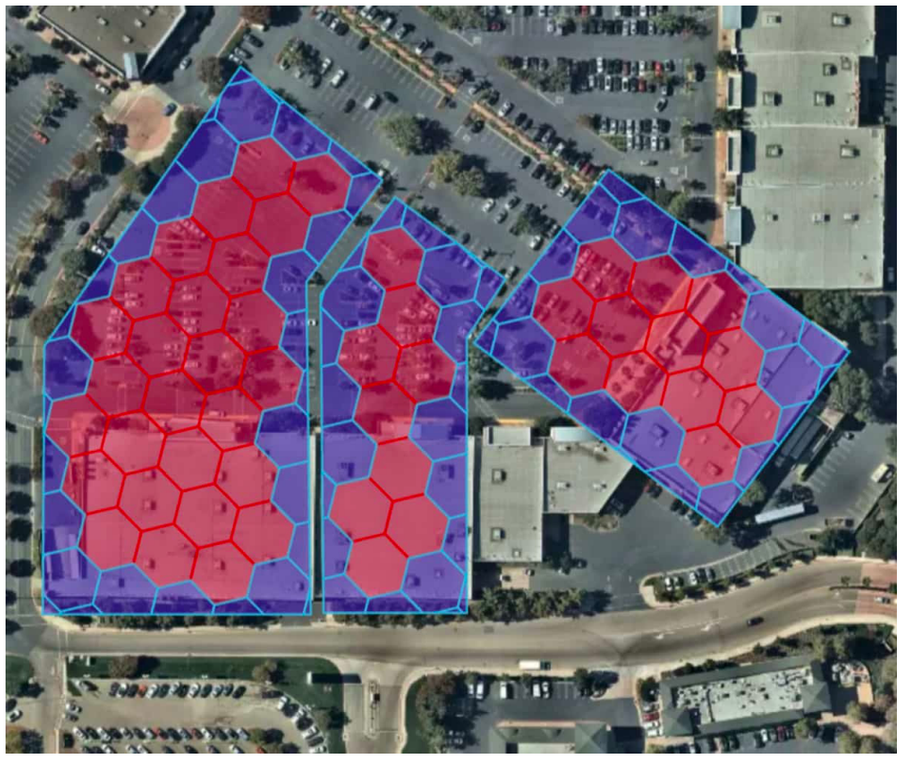


Figure 2. Geofence polygons from figure 1 showing contained H3 cells (in red) and the derived boundary chips (in blue).

When comparing the single index ID corresponding to a point to the set of index IDs associated with the polygon, there are three possible outcomes:

CASE	CONCLUSION
The point index ID is not contained within the set of polygon index IDs	The polygon does not contain the point
The point index ID is contained within the set of polygon index IDs and the cell corresponding to that index ID is completely contained within the polygon	The polygon contains the point
The point index ID is contained within the set of polygon index IDs and the cell corresponding to that index ID is partially contained within the polygon	We will need to compare the point with the chip (intersection of the cell geometry and the polygon) directly using contains relation — an O(v) operation

It is important to note that the indexing of polygons is a more expensive operation than indexing point data and is a one-off cost to this approach. As geofence polygons generally do not evolve as quickly as new cell phone ping points are received, the one-time preprocessing cost is a viable compromise for avoiding expensive Cartesian products.

Scalability and performance with Databricks and Mosaic

At Thasos, the team was working with hundreds of thousands of polygons and receiving billions of new points every day. While not all problems leveraged their full data set, the poor scaling of the naive comparison approach was apparent with out-of-memory issues, causing failed pipelines on their warehouse.
To remedy these issues, the team spatially partitioned their data, yielding stable pipelines. However, the issues of exponentially increasing processing cost and time still remained.

To address the issue of processing time, the team considered a framework that could elastically scale with their workloads, and Apache Spark was a natural choice. By leveraging more instances, the time taken to complete the pipelines could be managed. Despite the stability in time, total costs to execute the pipeline stayed relatively constant and the Thasos team was looking for a solution with better scaling, as they knew that the size and complexity of their data sources would continue to grow.

By collaborating with the Databricks team and leveraging Mosaic, the engineers at Thasos were able to reap the benefits of the optimized approach to PIP joins described in the previous section without having to implement the approach from scratch. Mosaic provided an easy to install geospatial framework with built-in functions for some of the more complicated operations (such as calculating an appropriate resolution for H3 indexing) required for the optimized PIP joins. The Thasos team identified two large pipelines appropriate for benchmarking. The first pipeline PIP joined new geofences received on a daily basis with the entire history of their cell phone ping data, while the second pipeline joined newly received cell phone ping data with Census Block polygons for the purpose of data partitioning and population estimates.

PIPELINE	SPARK PERFORMANCE	DATABRICKS + MOSAIC PERFORMANCE	IMPACT
#1 - Geofence PIP Pipeline	$4.33*	$1.74*	2.5x better price/ performance
#2 - Census Block PIP Pipeline	$130 35-40 mins	$13.08 25 mins	10x cheaper 29-38% faster

(*$ values are normalized)

The sizable improvements in performance provided the Thasos team confidence in the scalability of the approach and Mosaic is now an integral part of their production pipelines. Building these pipelines on the Databricks Lakehouse Platform has not only saved Thasos on cloud computing costs, but has also unlocked the opportunity to onboard data scientists on the team to develop new intelligence products and also to integrate with the broader Databricks partner ecosystem.

Getting started

Try Mosaic on Databricks to accelerate your geospatial analytics on lakehouse today and contact us to learn more about how we assist customers with similar use cases.

- **Mosaic is available as a Databricks Labs repository here.**
- **Detailed Mosaic documentation is available here.**
- **You can access the latest code examples here.**

> Read more about our built-in functionality for H3 indexing here.

About Databricks

Databricks is the lakehouse company. More than 7,000 organizations worldwide — including Comcast, Condé Nast and over 50% of the Fortune 500 — rely on the Databricks Lakehouse Platform to unify their data, analytics and AI. Databricks is headquartered in San Francisco, with offices around the globe. Founded by the original creators of Apache Spark™, Delta Lake and MLflow, Databricks is on a mission to help data teams solve the world's toughest problems. To learn more, follow Databricks on Twitter, LinkedIn and Facebook.

Schedule a personalized demo

Sign up for a free trial

databricks