Solving Business Needs with Delta Lake

A practical guide including code samples and notebooks by vertical

databricks

Introduction

Data professionals across a wide range of industries are looking to help their organizations innovate for competitive advantage by making the most of their data. Good quality, reliable data forms the foundation for success for such analytics and machine learning initiatives.

Delta Lake is an open source storage layer that brings reliability to data lakes a key component of modern data architectures. It provides:

- ACID transactions
- Scalable metadata handling
- Unified streaming and batch data processing.

Delta Lake runs on top of your existing data lake and is fully compatible with Apache Spark™ APIs.

Seeing how Delta Lake can be used to address various business needs can help you get a good sense of how you can deploy it to help you enable downstream users with reliable data.

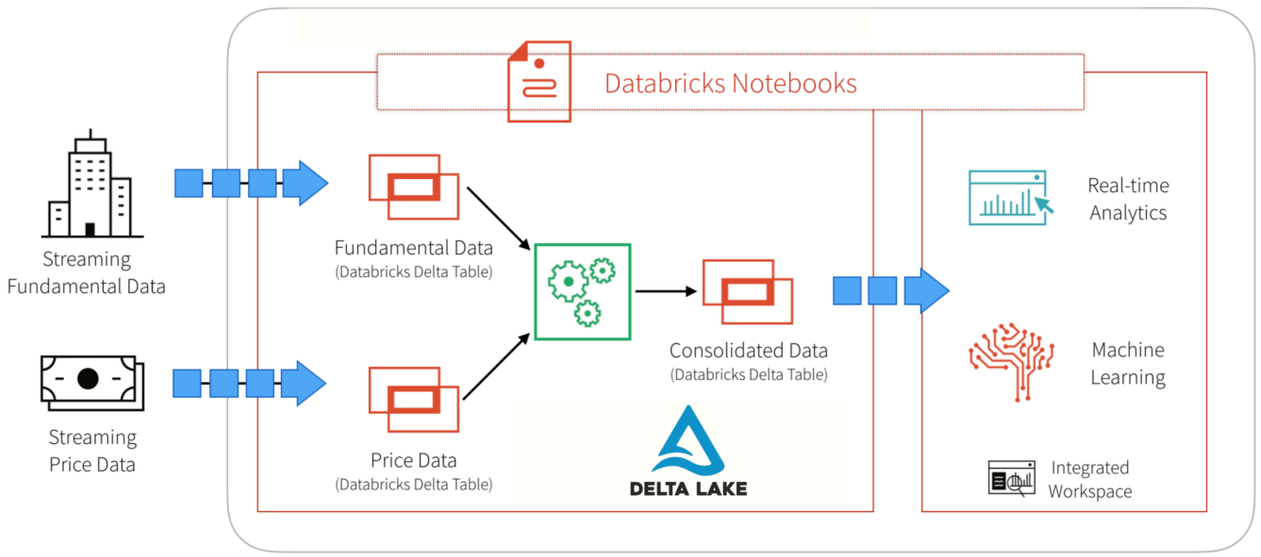
In this guide we will walk you through four examples of applying Delta Lake to solve business needs. Specifically, they cover building data pipelines for:

- Streaming financial stock data analysis that delivers transactional consistency of legacy and streaming data concurrently
- Genomic data analytics used for analyzing population-scale genomic data
- Real-time display advertising attribution for delivering information on advertising spend effectiveness
- Mobile gaming data event processing to enable fast metric calculations and responsive scaling

Streaming Financial Stock Data Analysis

Traditionally, real-time analysis of stock data was a complicated endeavor due to the complexities of maintaining a streaming system and ensuring transactional consistency of legacy and streaming data concurrently. Delta Lake helps solve many of the pain points of building a streaming system to analyze stock data in real-time.

In the following diagram, we provide a high-level architecture to simplify this problem. We start by ingesting two different sets of data into two Delta Lake tables. The two datasets are stocks prices and fundamentals. After ingesting the data into their respective tables, we then join the data in an ETL process and write the data out into a third Delta Lake table for downstream analysis.



In this section we will review:

- The typical problems of running real-time stock analysis
- How Delta Lake addresses these problems
- How to implement the system using Databricks (note: the system can be implemented with just open source software)

TYPICAL PAIN POINTS (BEFORE DELTA LAKE)

The pain points of a traditional streaming and data warehousing solution can be broken into two groups: data lake pains and data warehouse pains.

DATA LAKE PAIN POINTS

While data lakes allow you to flexibly store an immense amount of data in a file system, there are many pain points including (but not limited to):

- Consolidation of streaming data from many disparate systems is difficult.
- Updating data in a data lake is nearly impossible and much of the streaming data needs to be updated as changes are made. This is especially important in scenarios involving financial reconciliation and subsequent adjustments.
- Query speeds for a data lake are typically very slow.

DATA WAREHOUSE PAIN POINTS

The power of a data warehouse is that you have a persistent performant store of your data. But the pain points for building modern continuous applications include (but not limited to):

- Constrained to SQL queries; i.e. no machine learning or advanced analytics.
- Accessing streaming data and stored data together is very difficult if at all possible.
- Data warehouses do not scale very well.
- Tying compute and storage together makes using a warehouse very expensive.

HOW DELTA LAKE SOLVES THESE ISSUES

Delta Lake (Delta Lake Guide) is an open source storage layer that brings reliability to data lakes. It has numerous reliability features including ACID transactions, scalable metadata handling, and unified streaming and batch data processing. Delta Lake runs on top of your existing data lake and is fully compatible with Apache Spark™ APIs. There are several points to note:

- Your streaming/data lake/warehousing solution has ACID compliance.
- Delta Lake, along with Structured Streaming, makes it possible to analyze streaming and historical data together at data warehouse speeds.
- Using Delta Lake tables as sources and destinations of streaming big data makes it easy to consolidate disparate data sources.
- Upserts are supported on Delta Lake tables so changes are simpler to manage.
- You can easily include machine learning scoring and advanced analytics into ETL and queries.
- Compute and storage are decoupled resulting in a completely scalable solution.

IMPLEMENT YOUR STREAMING STOCK ANALYSIS SOLUTION WITH DELTA LAKE

Delta Lake and Apache Spark do most of the work for our solution; you can try out the full notebook and follow along with the code samples below. Let's start by enabling Delta Lake.

As noted in the preceding diagram, we have two datasets to process – one for fundamentals and one for price data. To create our two Delta Lake tables, we specify the `.format("delta")` against our DBFS locations.

```
# Create Fundamental Data (Databricks Delta table)
dfBaseFund = spark \
.read \
.format('delta') \
.load('/delta/stocksFundamentals')

# Create Price Data (Databricks Delta table)
dfBasePrice = spark \
.read \
.format('delta') \
.load('/delta/stocksDailyPrices')
```

While we're updating the `stockFundamentals` and `stocksDailyPrices`, we will consolidate this data through a series of ETL jobs into a consolidated view (`stocksDailyPricesWFund`). With the following code snippet, we can determine the start and end date of available data and then combine the price and fundamentals data for that date range into DBFS.

```
# Determine start and end date of available data
row = dfBasePrice.agg(
 func.max(dfBasePrice.price_date).alias("maxDate"),
 func.min(dfBasePrice.price_date).alias("minDate")
).collect()[0]
startDate = row["minDate"]
endDate = row["maxDate"]

# Define our date range function
def daterange(start_date, end_date):
    for n in range(
        int((end_date - start_date).days)):
        yield start_date + datetime.timedelta(n)

# Define combinePriceAndFund information by date
def combinePriceAndFund(theDate):
  dfFund = dfBaseFund.where(
      dfBaseFund.price_date == theDate)
  dfPrice = dfBasePrice.where(
      dfBasePrice.price_date == theDate) \
                .drop('price_date')
  # Drop the updated column
  dfPriceWFund = dfPrice.join(dfFund,["ticker"]) \
                        .drop("updated")

# Save data to DBFS
dfPriceWFund.write.format("delta").mode("append") \
.save("/delta/stocksDailyPricesWFund")

# Loop through dates to complete fundamentals
# + price ETL process
for single_date in daterange(startDate,
    (endDate + datetime.timedelta(days=1))):
  start = datetime.datetime.now()
  combinePriceAndFund(single_date)
  end = datetime.datetime.now()
```

Now we have a stream of consolidated fundamentals and price data that is being pushed into DBFS in the `/delta/stocksDailyPricesWFund` location. We can build a Delta Lake table by specifying `.format("delta")` against that DBFS location.

```
dfPriceWithFundamentals = spark
.readStream
.format("delta")
.load("/delta/stocksDailyPricesWFund")
# Create temporary view of the data
dfPriceWithFundamentals.
createOrReplaceTempView("priceWithFundamentals")
```

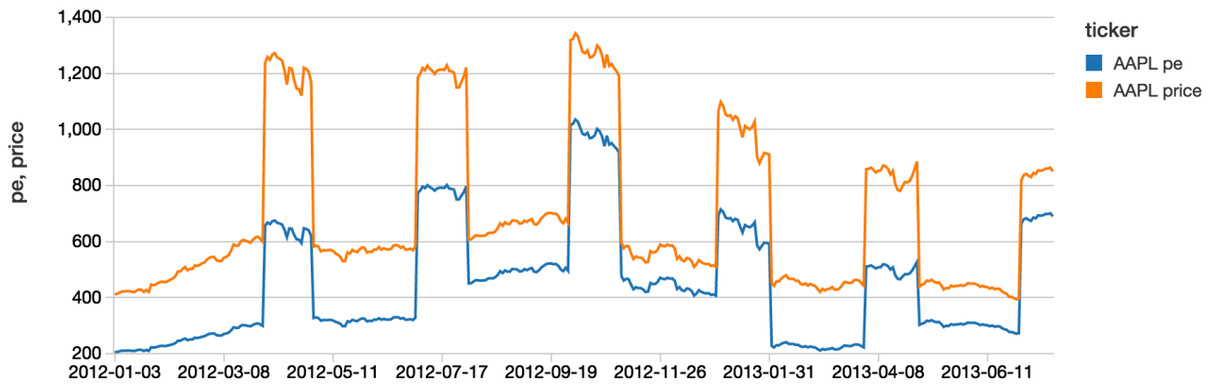
Now that we have created our initial Delta Lake table, let's create a view that will allow us to calculate the price/earnings ratio in real time (because of the underlying streaming data updating our Delta Lake table).

```
%sql
CREATE OR REPLACE TEMPORARY VIEW viewPE AS
select ticker,
       price_date,
       first(close) as price,
       (close/eps_basic_net) as pe
  from priceWithFundamentals
 where eps_basic_net > 0
```

ANALYZE STREAMING STOCK DATA IN REAL TIME

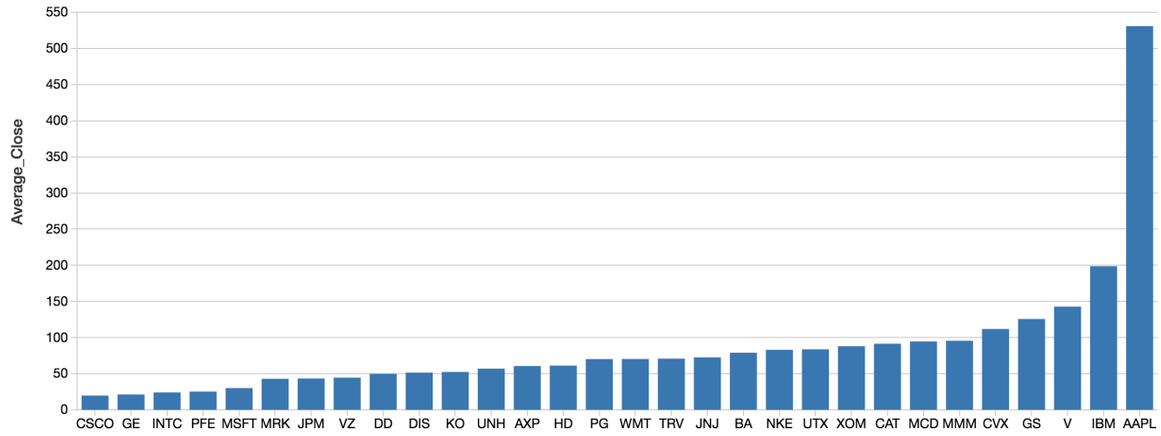
With our view in place, we can quickly analyze our data using Spark SQL.

```
%sql
select *
from viewPE
where ticker == "AAPL"
order by price_date
```



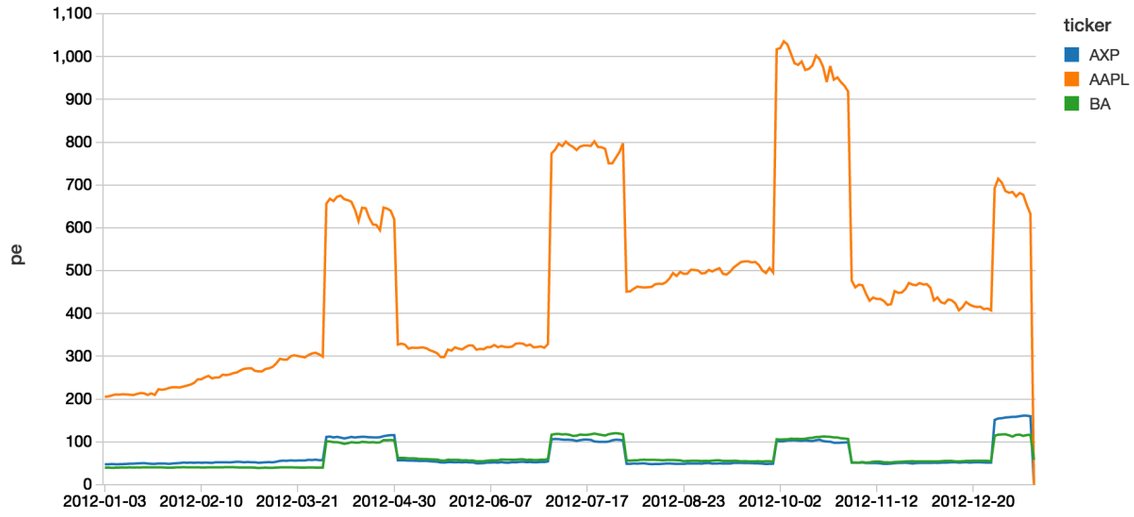
As the underlying source of this consolidated dataset is a Delta Lake table, this view isn't just showing the batch data but also any new streams of data that are coming in as per the following streaming dashboard.

Underneath the covers, Structured Streaming isn't just writing the data to Delta Lake tables but also keeping the state of the distinct number of keys (in this case ticker symbols) that need to be tracked.



Because you are using Spark SQL, you can execute aggregate queries at scale and in real-time.

```
%sql
SELECT ticker, AVG(close) as Average_Close
FROM priceWithFundamentals
GROUP BY ticker
ORDER BY Average_Close
```



SIMPLIFYING STREAMING STOCK ANALYSIS: ON-DEMAND WEBINAR AND FAQ NOW AVAILABLE

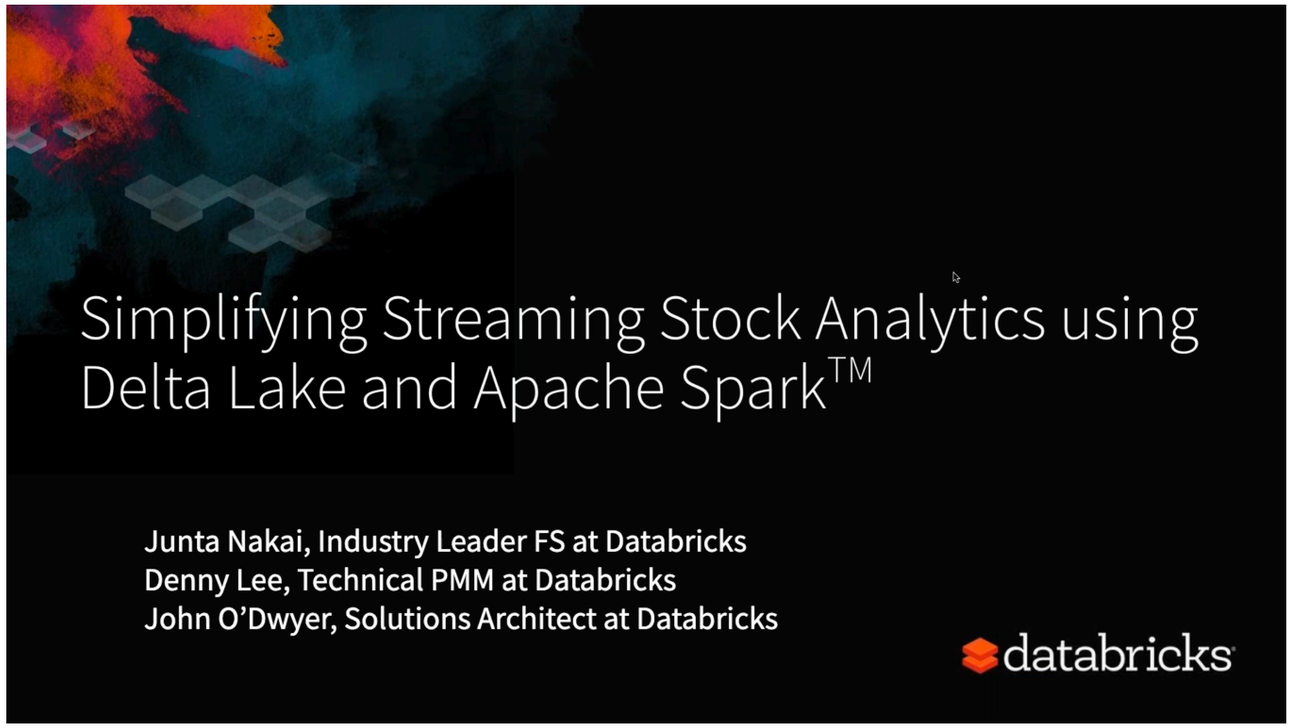
On June 13th, we hosted a live webinar — Simplifying Streaming Stock Analysis using Delta Lake and Apache Spark — with Junta Nakai, Industry Leader – Financial Services at Databricks, John O'Dwyer, Solution Architect at Databricks, and Denny Lee, Technical Product Marketing Manager at Databricks. This is the first webinar in a series of financial services webinars.

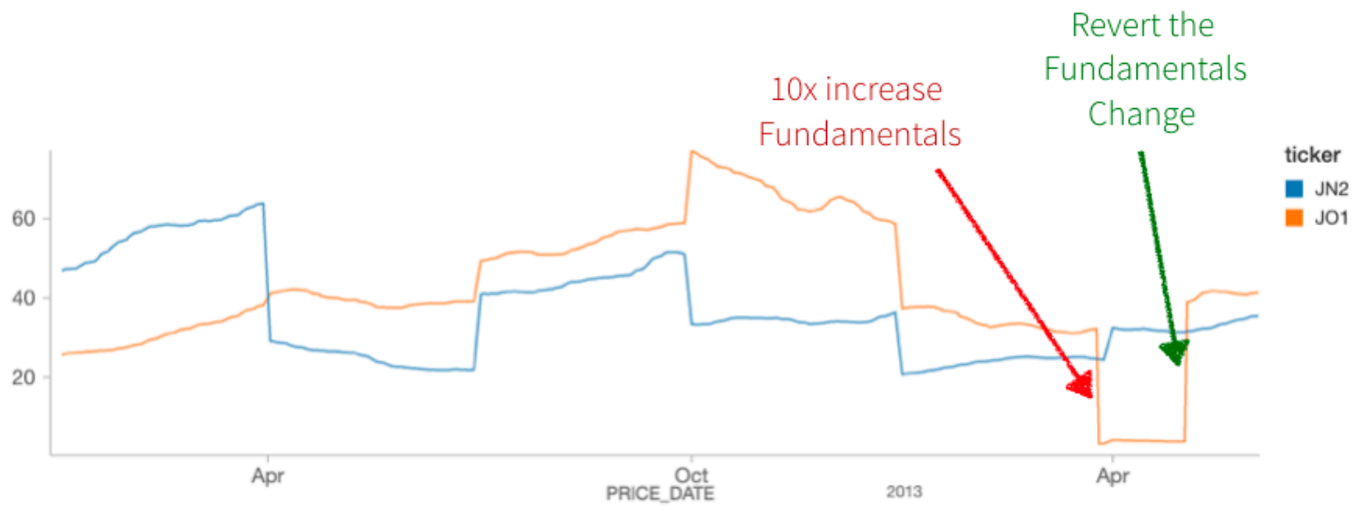
During the webinar, we showcased Streaming Stock Analysis with a Delta Lake notebook. To run it yourself, please download the following notebooks:

- Streaming Stock Analysis with Delta Lake: Setup – First run this notebook so it can automatically download the generated source data and start loading data into a file location.
- Streaming Stock Analysis with Delta Lake – This is the main notebook that showcases Delta Lake within the context of streaming stock analysis including *unified streaming*, *batch sync* and *time travel*.

We also showcase the update of data in real-time with streaming and batch stock analysis data joined together as noted in the image (next panel).

Toward the end, we also held a Q&A, and the questions and their answers follow this slide.





Q: WHAT IS THE DIFFERENCE BETWEEN DELTA LAKE AND APACHE PARQUET?

Delta Lake is an open-source storage layer that brings ACID transactions to Apache Spark™ and big data workloads. While the Delta Lake stores data in Apache Parquet format, it includes features that allow data lakes to be reliable at scale. These features include:

- **ACID Transactions:** Delta Lake ensures data integrity and provides serializability.

- **Scalable Metadata Handling:** For Big Data systems, the metadata itself is often "big" enough to slow down any system that tries to make sense of it, let alone making sense of the actual underlying data. Delta Lake treats metadata like regular data and leverages Apache Spark's distributed processing power. As a result, Delta Lake can handle petabyte-scale tables with billions of partitions and files at ease.

- **Time Travel (data versioning):** Creates snapshots of data, allowing you to access and revert to earlier versions of data for audits, rollbacks or to reproduce experiments.

- **Open Format:** All data in Delta Lake is stored in Apache Parquet format enabling Delta Lake to leverage the efficient compression and encoding schemes that are native to Parquet.

- **Unified Batch and Streaming Source and Sink:** A table in Delta Lake is both a batch table, as well as a streaming source and sink. Streaming data ingest, batch historic backfill, and interactive queries all just work out of the box.

- **Schema Enforcement:** Delta Lake provides the ability to specify your schema and enforce it. This helps ensure that the data types are correct and required columns are present, preventing bad data from causing data corruption.

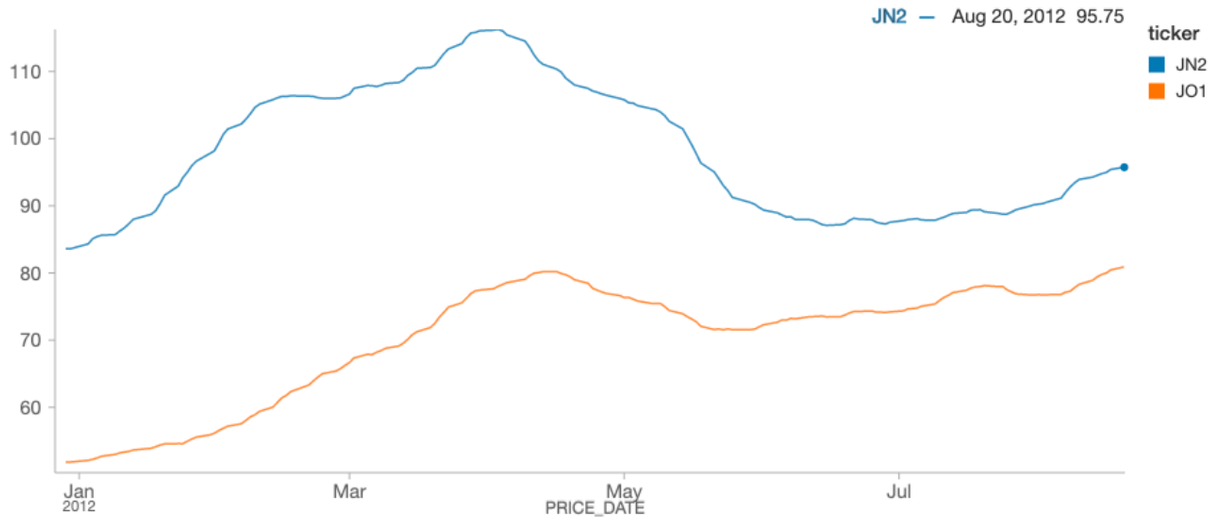
- **Schema Evolution:** Big data is continuously changing. Delta Lake enables you to make changes to a table schema that can be applied automatically, without the need for cumbersome DDL.

- **100% Compatible with Apache Spark API:** Developers can use Delta Lake with their existing data pipelines with minimal change as it is fully compatible with Spark, the commonly used big data processing engine.

Q: HOW CAN YOU VIEW THE DELTA LAKE TABLE FOR BOTH STREAMING AND BATCH NEAR THE BEGINNING OF THE NOTEBOOK?

As noted in the Streaming Stock Analysis with Delta Lake notebook, in cell 8 we ran the following batch query:

```
dfPrice = \
spark.read.format("delta").load(deltaPricePath)
display(dfPrice.where(
  dfPrice.ticker.isin({'JO1', 'JN2'})))
```

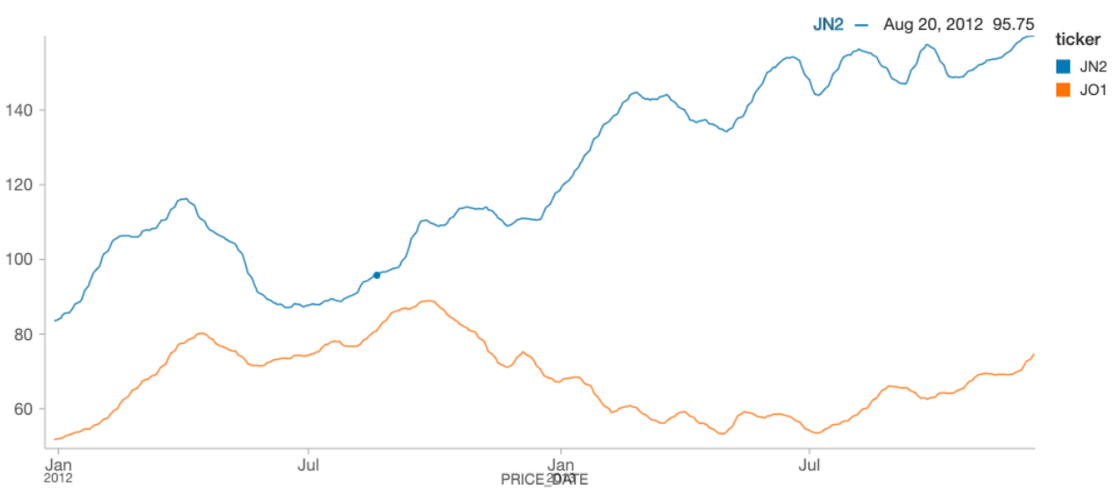


Notice that we ran this query earlier in the cycle with data up until August 20th, 2012. Using the same folder path (`deltaPricePath`), we also created a structured streaming DataFrame via the following code snippet in cell 4:

```
# Create Stream and Temp View for Price
dfPriceStream =
spark.readStream.format("delta").load(deltaPricePath)
dfPriceStream.createOrReplaceTempView("priceStream")
```

We can then run the following real-time Spark SQL query that will continuously refresh.

```
%sql
SELECT *
FROM priceStream
where ticker in ('JO1', 'JN2')
```



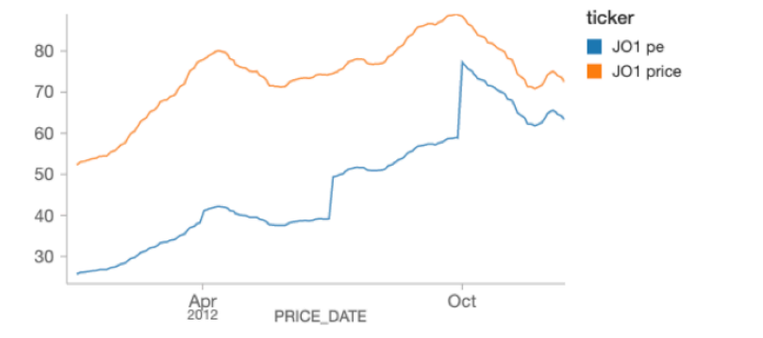
Notice that, even though the batch query executed earlier (and ended at August 20th, 2012), the structured streaming query continued to process data long past that date (the small blue dot denotes where August 20th, 2012 is on the streaming line chart). As you can see from the preceding code snippets, both the batch and structured streaming DataFrames query off of the same folder path of `deltaPricePath`.

Q: WITH THE "MISTAKE" THAT YOU HAD ENTERED INTO THE DATA, CAN I GO BACK AND FIND IT AND POSSIBLY CORRECT IT FOR AUDITING PURPOSES?

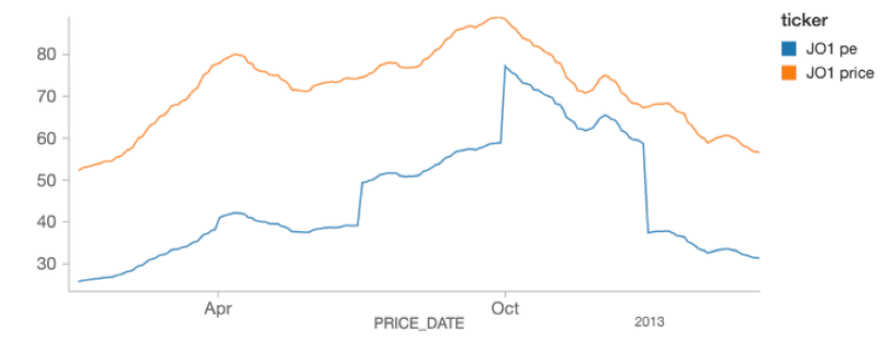
Delta Lake has a data versioning feature called *Time Travel*. It provides snapshots of data, allowing you to access and revert to earlier versions of data for audits, rollbacks or to reproduce experiments. To visualize this, note cells 36 onwards in the Streaming Stock Analysis with Delta Lake notebook. The screenshot (opposite panel) shows three different queries using the `VERSION AS OF` syntax allowing you to view your data by version (or by timestamp using the `TIMESTAMP` syntax).

With this capability, you can know what changes to your data were made and when those transactions had occurred.

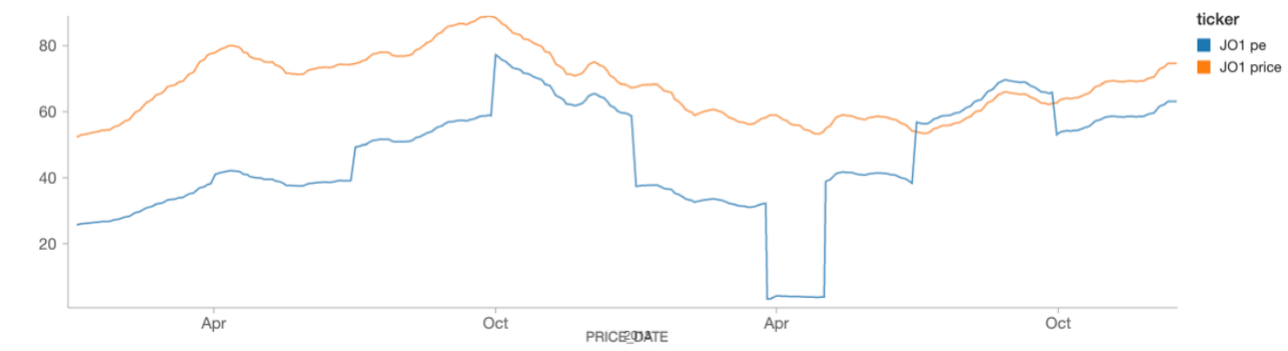
```
%sql
SELECT ticker, price_date, adj_close as price, (adj_close/eps_basic_net) as pe
FROM priceWithFundamentalsHistory VERSION AS OF 1
WHERE ticker == "JO1"
```




```
%sql
SELECT ticker, price_date, adj_close as price, (adj_close/eps_basic_net) as pe
FROM priceWithFundamentalsHistory VERSION AS OF 20
WHERE ticker == "JO1"
```




```
%sql
SELECT ticker, price_date, adj_close as price, (adj_close/eps_basic_net) as pe
FROM priceWithFundamentalsHistory
WHERE ticker == "JO1"
```



Q: I SAW THAT THE STOCK STREAMING DATA UPDATE WAS ACCOMPLISHED VIA A VIEW; I WONDER IF UPDATES CAN BE DONE ON ACTUAL DATA FILES THEMSELVES. FOR INSTANCE, DO WE NEED TO REFRESH THE WHOLE PARTITION PARQUET FILES TO ACHIEVE UPDATES? WHAT IS THE SOLUTION UNDER DELTA LAKE?

While the changes were done to a Spark SQL view, the changes are actually happening to the underlying files on storage. Delta Lake itself determines which Parquet files need to be updated to reflect the new changes.

Q: CAN WE QUERY DELTA LAKE TABLES IN APACHE HIVE?

Currently (as of version 0.1.0) it is not possible to query Delta Lake tables with Apache Hive nor is the Hive metastore supported (though this feature is on the roadmap). For the latest on this particular issue, please refer to the GitHub issue #18.

Q: IS THERE ANY GUIDE THAT COVERS DETAILED USAGE OF DELTA LAKE?

For the latest guidance on Delta Lake, please refer to the delta.io as well as the Delta Lake documentation. Join the Delta Lake Community to communicate with fellow Delta Lake users and contributors through our Slack channel or Google Groups.

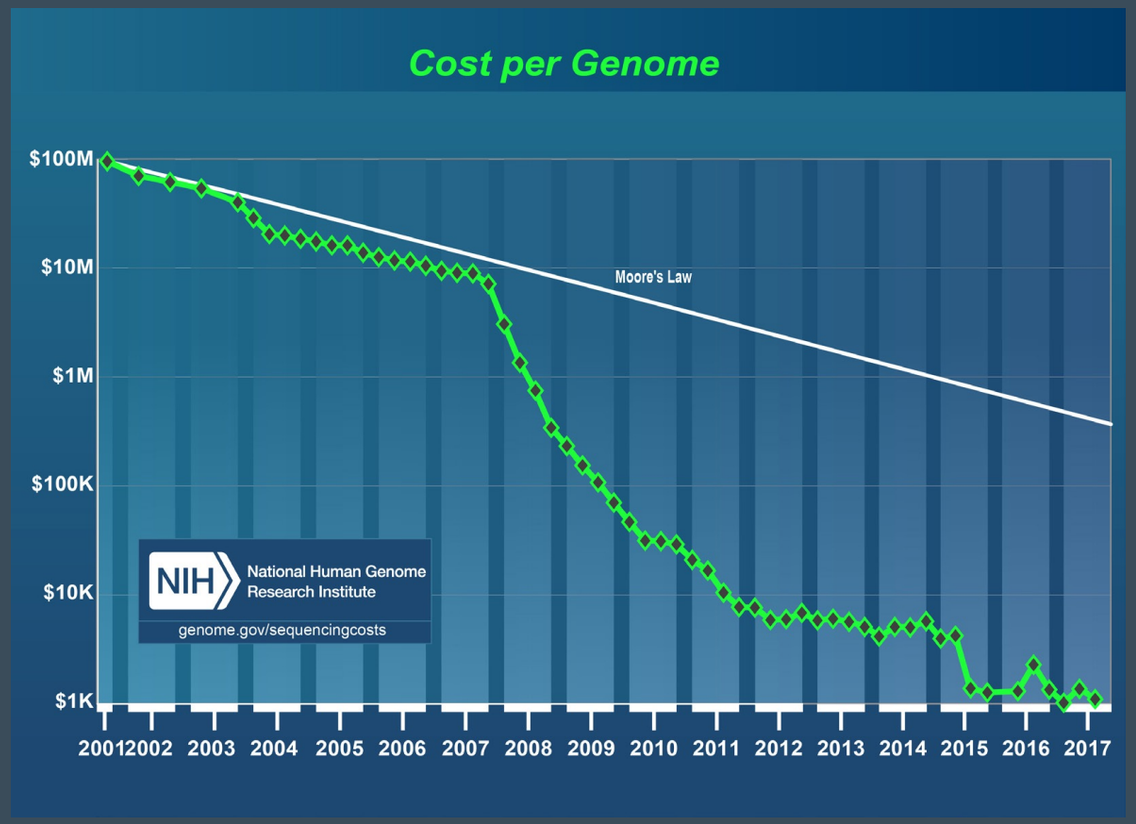
SUMMARY

In closing, we demonstrated how to simplify streaming stock data analysis using Delta Lake. By combining Spark Structured Streaming and Delta Lake, we have a scalable solution that has the advantages of both data lakes and data warehouses. This removes the data engineering complexities commonly associated with streaming and transactional consistency enabling data engineering and data science teams to focus on understanding the trends in their stock data.

Genomic Data Analytics

DOWNLOAD THE NOTEBOOK AND TRY IT OUT

Since the completion of the Human Genome Project in 2003, there has been an explosion in data fueled by a dramatic drop in the cost of DNA sequencing, from $3B for the first genome to under $1,000 today.



Source: DNA Sequencing Costs: Data

[1] The Human Genome Project was a $3B project led by the Department of Energy and the National Institutes of Health began in 1990 and completed in 2003.

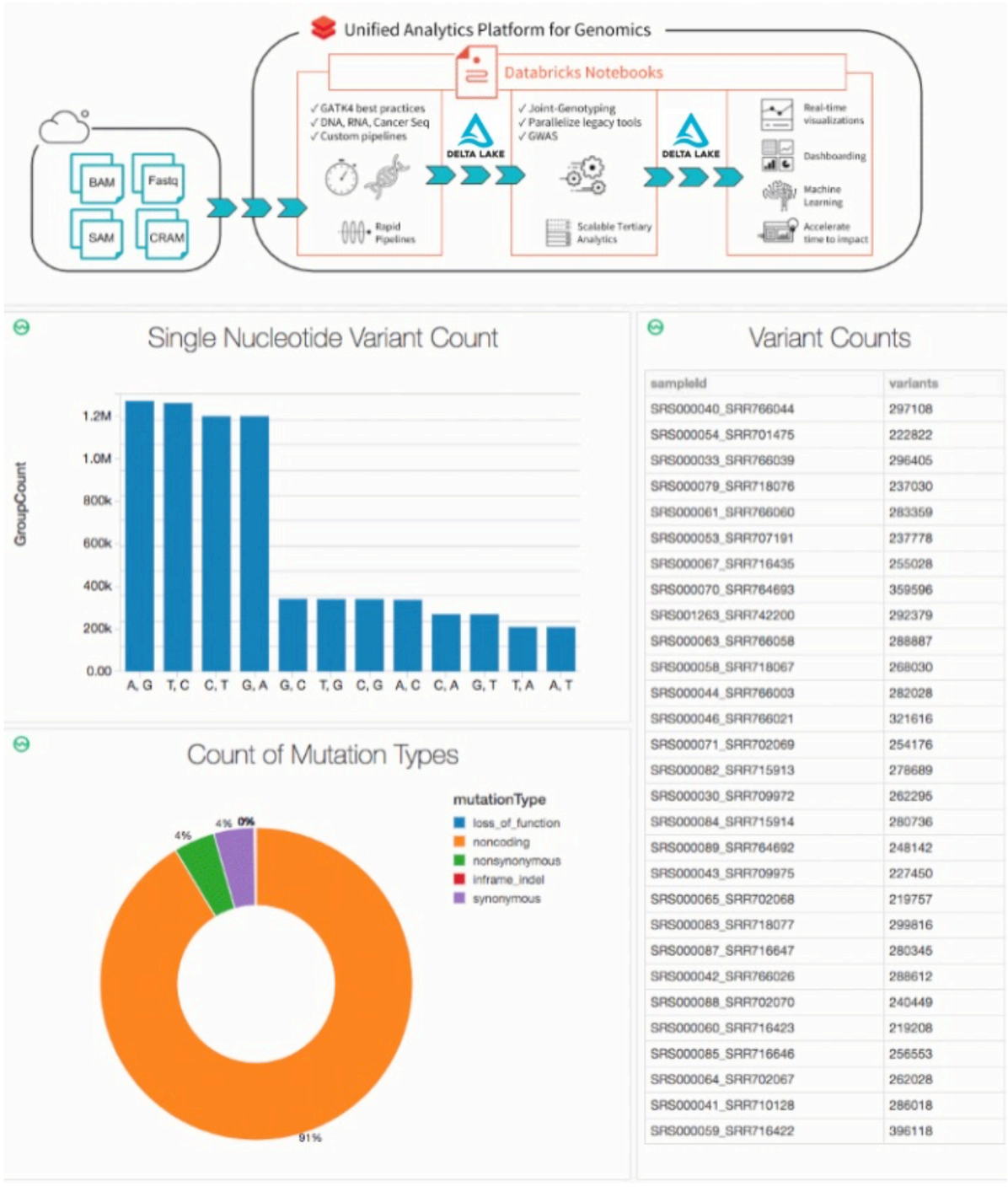
Consequently, the field of genomics has now matured to a stage where companies have started to do DNA sequencing at population-scale. However, sequencing the DNA code is only the first step, the raw data then needs to be transformed into a format suitable for analysis. Typically this is done by gluing together a series of bioinformatics tools with custom scripts and processing the data on a single node, one sample at a time, until we wind up with a collection of genomic variants. Bioinformatics scientists today spend the majority of their time building out and maintaining these pipelines. As genomic data sets have expanded into the petabyte scale, it has become challenging to answer even the following simple questions in a timely manner:

- How many samples have we sequenced this month?
- What is the total number of unique variants detected?
- How many variants did we see across different classes of variation?

Further compounding this problem, data from thousands of individuals cannot be stored, tracked nor versioned while also remaining accessible and queryable. Consequently, researchers often duplicate subsets of their genomic data when performing their analyses, causing the overall storage footprint and costs to escalate. In an attempt to alleviate this problem, today researchers employ a strategy of "data freezes", typically between six months to two years, where they halt work on new data and instead focus on a frozen copy of existing data. There is no solution to incrementally build up analyses over shorter time frames, causing research progress to slow down.

ARCHITECTURE FOR END-TO-END GENOMICS ANALYSIS WITH DATABRICKS

Note that this architecture can also be achieved with open source software.



There is a compelling need for robust software that can consume genomic data at industrial scale, while also retaining the flexibility for scientists to explore the data, iterate on their analytical pipelines, and derive new insights.

With Delta Lake, you can store all your genomic data in one place, and create analyses that update in real-time as new data is ingested. As an example, we've highlighted the following dashboard showing quality control metrics and visualizations that can be calculated and presented in an automated fashion and customized to suit your specific requirements.

In the rest of this section, we will walk through the steps we took to build the quality control dashboard above, which updates in real time as samples finish processing. By using a Delta Lake-based pipeline for processing genomic data, data teams can now operate their pipelines in a way that provides real-time, sample-by-sample visibility. With Databricks notebooks (and integrations such as GitHub and MLflow) they can track and version analyses in a way that will ensure their results are reproducible. Their bioinformaticians can devote less time to maintaining pipelines and spend more time making discoveries. The net effect is to help drive the transformation from ad-hoc analyses to production genomics on an industrial scale, enabling better insights into the link between genetics and disease.

READ SAMPLE DATA

Let's start by reading variation data from a small cohort of samples; the following statement reads in data for a specific sampleId and saves it using the Delta Lake format (in the `delta_stream_output` folder).

```
# Read the initial sample annotation file
spark.read \
     .format("parquet"). \
     .load("dbfs:/annotations_etl_parquet/sampleId=" +
          "SRS000030_SRR709972") \
     .write \
     .format("delta") \
     .save(delta_stream_outpath)
```

Note, the annotations_etl_parquet folder contains annotations generated from the 1000 genomes dataset stored in parquet format. The ETL and processing of these annotations were performed using Databricks' Unified Analytics Platform for Genomics.

START STREAMING THE DELTA LAKE TABLE

In the following statement, we are creating the `exomes` Apache Spark DataFrame which is reading a stream (via readStream) of data using the Delta Lake format. This is a continuously running or dynamic DataFrame, i.e. the `exomes` DataFrame will load new data as data is written into the `delta_stream_output` folder. To view the `exomes` DataFrame, we can run a DataFrame query to find the count of variants grouped by the `sampleId`.

```
# Read the stream of data
exomes = spark.readStream \
         .format(“delta").load(delta_stream_outpath)

# Display the data via DataFrame query
display(exomes.groupBy("sampleId").count() \
          .withColumnRenamed("count", "variants"))
```

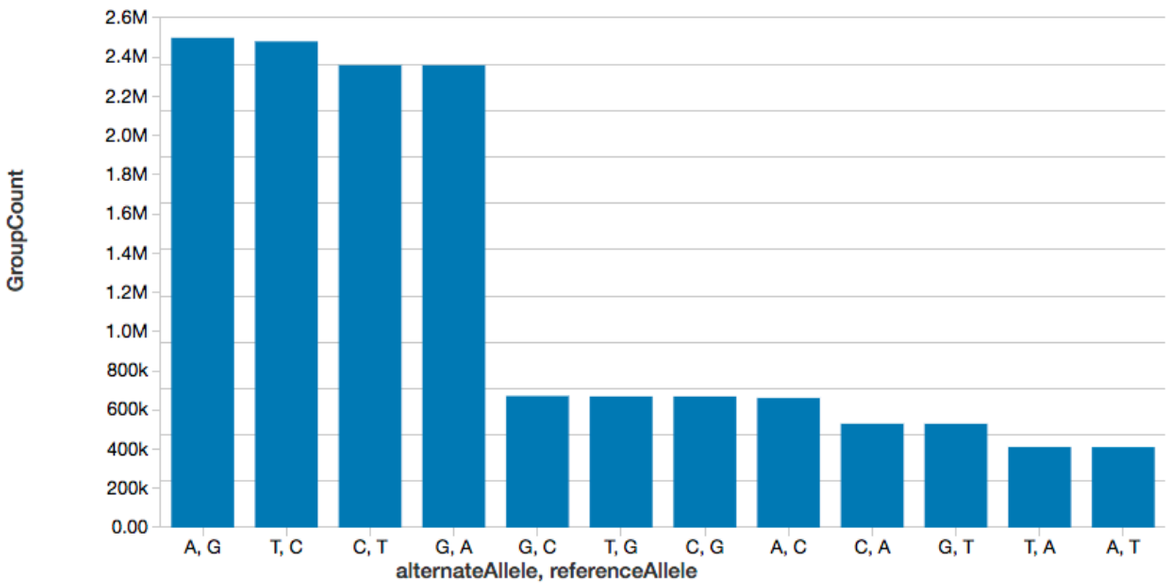
When executing the `display` statement, the Databricks notebook provides a streaming dashboard to monitor the streaming jobs (as noted in the preceding figure). Immediately below the streaming job are the results of the display statement (i.e. the count of variants by `sample_id`).

Let's continue answering our initial set of questions by running other DataFrame queries based on our `exomes` DataFrame.

SINGLE NUCLEOTIDE VARIANT COUNT

To continue the example, we can quickly calculate the number of single nucleotide variants (SNVs), as displayed in the following graph.

```
%sql
select referenceAllele, alternateAllele, count(1) as
GroupCount
  from snvs
 group by referenceAllele, alternateAllele
 order by GroupCount desc
```

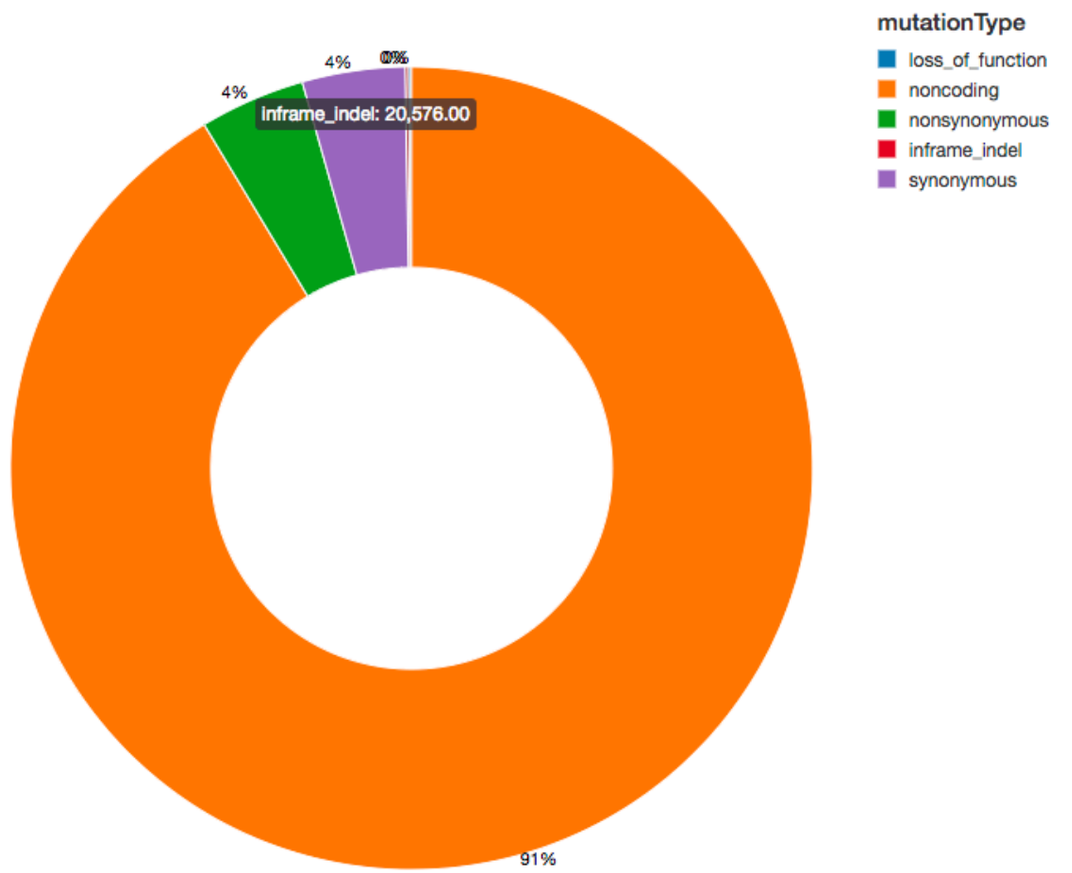


VARIANT COUNT

Since we have annotated our variants with functional effects, we can continue our analysis by looking at the spread of variant effects we see. The majority of the variants detected flank regions that code for proteins, these are known as noncoding variants.

```
display(exomes.groupBy("mutationType").count())
```

Note, the `display` command is part of the Databricks workspace that allows you to view your DataFrame using Databricks visualizations (i.e. no coding required).



AMINO ACID SUBSTITUTION HEATMAP

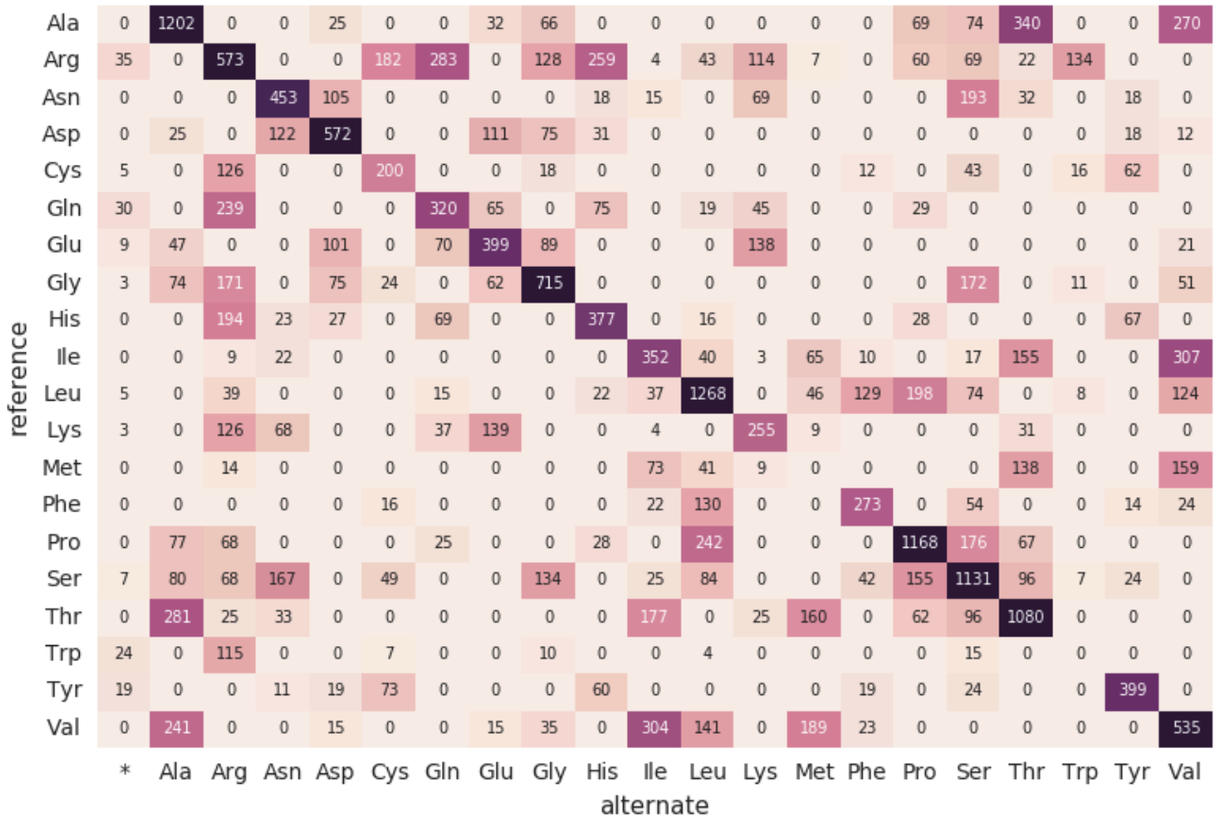
Continuing with our `exomes` DataFrame, let's calculate the amino acid substitution counts with the following code snippet. Similar to the previous DataFrames, we will create another dynamic DataFrame (`aa_counts`) so that as new data is processed by the `exomes` DataFrame, it will subsequently be reflected in the amino acid substitution counts as well. We are also writing the data into memory (i.e. `.format("memory")`) and processed batches every 60s (i.e. `trigger(processingTime='60 seconds')`) so the downstream Pandas heatmap code can process and visualize the heatmap.

```
# Calculate amino acid substitution counts
coding = get_coding_mutations(exomes)
aa_substitutions =
get_amino_acid_substitutions(coding.select("proteinHgvs"), "proteinHgvs")
aa_counts =
count_amino_acid_substitution_combinations(aa_substitutions)
aa_counts. \
  writeStream. \
  format("memory"). \
  queryName("amino_acid_substitutions"). \
  outputMode("complete"). \
  trigger(processingTime='60 seconds'). \
  start()
```

The following code snippet reads the preceding `amino_acid_substitutions` Spark table, determines the max count, creates a new Pandas pivot table from the Spark table, and then plots out the heatmap.

```
# Use pandas and matplotlib to build heatmap
amino_acid_substitutions =
spark.read.table("amino_acid_substitutions")
max_count =
amino_acid_substitutions.agg(fx.max("substitutions")).
collect()[0][0]
aa_counts_pd = amino_acid_substitutions.toPandas()
aa_counts_pd = pd.pivot_table(aa_counts_pd,
values='substitutions', index=['reference'],
columns=['alternate'], fill_value=0)

fig, ax = plt.subplots()
with sns.axes_style("white"):
  ax = sns.heatmap(aa_counts_pd, vmax=max_count*0.4,
cbar=False, annot=True, annot_kws={"size": 7},
fmt="d")
plt.tight_layout()
display(fig)
```



MIGRATING TO A CONTINUOUS PIPELINE

Up to this point, the preceding code snippets and visualizations represent a single run for a single `sampleId`. But because we're using Structured Streaming and Delta Lake, this code can be used (without any changes) to construct a production data pipeline that computes quality control statistics continuously as samples roll through our pipeline. To demonstrate this, we can run the following code snippet that will load our entire dataset.

```
import time
parquets = "dbfs:/databricks-datasets/genomics/
annotations_etl_parquet/"
files = dbutils.fs.ls(parquets)
counter = 0
for sample in files:
  counter += 1
  annotation_path = sample.path
  sampleId = annotation_path.split("/")[4].split("=")
[1]
  variants = spark.read.format("parquet"). \
load(str(annotation_path))
  print("running " + sampleId)
  if(sampleId != "SRS000030_SRR709972"):
    variants.write.format("delta"). \
             mode("append"). \
             save(delta_stream_outpath)
  time.sleep(10)
```

As described in the earlier code snippets, the source of the `exomes` DataFrame are the files loaded into the `delta_stream_output` folder. Initially, we had loaded a set of files for a single `sampleId` (i.e., `sampleId="SRS000030_SRR709972"`). The preceding code snippet now takes all of the generated parquet samples (i.e. `parquets`) and incrementally loads those files by `sampleId` into the same `delta_stream_outputfolder`.

VISUALIZING YOUR GENOMICS PIPELINE

When you scroll back to the top of your notebook, you will notice that the `exomes` DataFrame is now automatically loading the new `sampleIds`. Because the structured streaming component of our genomics pipeline runs continuously, it processes data as soon as new files are loaded into the `delta_stream_output` path folder. By using the Delta format, we can ensure the transactional consistency of the data streaming into the `exomes` DataFrame.

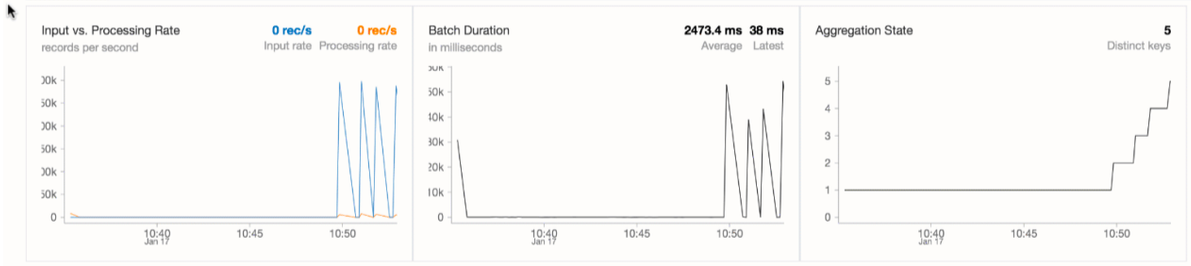
```
exomes = spark.readStream.format("delta").load(delta_stream_outpath)
display(exomes.groupBy("sampleId").count().withColumnRenamed("count", "variants"))
```

Cancel

(1) Spark Jobs

display_query_1 (Id: a7ff3628-dc67-42f1-b5f5-dbfe2a66d08d) *Last updated: 35 seconds ago*

Dashboard | Raw Data



sampleId	variants
SRS000040_SRR766044	297108
SRS000033_SRR766039	296405
SRS000030_SRR709972	262295
SRS000041_SRR710128	286018

As opposed to the initial creation of our `exomes` DataFrame, notice how the structured streaming monitoring dashboard is now loading data (i.e., the fluctuating "input vs. processing rate", fluctuating "batch duration", and an increase of distinct keys in the "aggregations state"). As the `exomes` DataFrame is processing, notice the new rows of `sampleIds` (and variant counts). This same action can also be seen for the associated group by mutation type query.

```
1  import time
2  parquets = "dbfs:/wbrandler/dnaseq/annotations_etl_parquet/"
3  files = dbutils.fs.ls(parquets)
4  counter=0
5  for sample in files:
6    counter+=1
7    annotation_path = sample.path
8    sampleId = annotation_path.split("/")[4].split("=")[1]
9    variants = spark.read.format("parquet").load(str(annotation_path))
10   print("running " + sampleId)
11   if(sampleId != "SRS000030_SRR709972"):
12     variants.write.format("delta"). \
13              mode("append"). \
14              save(delta_stream_outpath)
15   time.sleep(10)
```

Cancel • Running command...

▸ (6) Spark Jobs

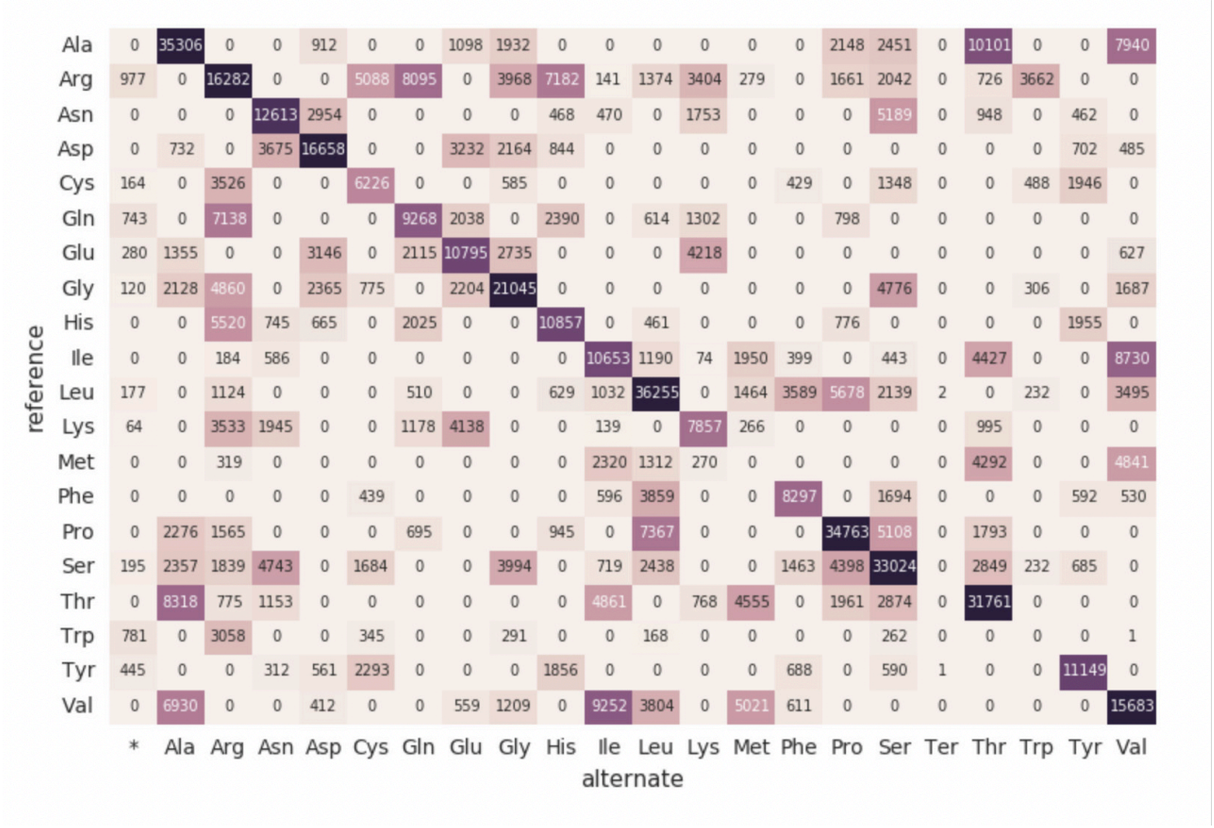
▸ variants: pyspark.sql.dataframe.DataFrame = [contigName: string, start: long ... 12 more fields]

```
running SRS000030_SRR709972
running SRS000033_SRR766039
running SRS000040_SRR766044
```

Cmd 26

```
1
```

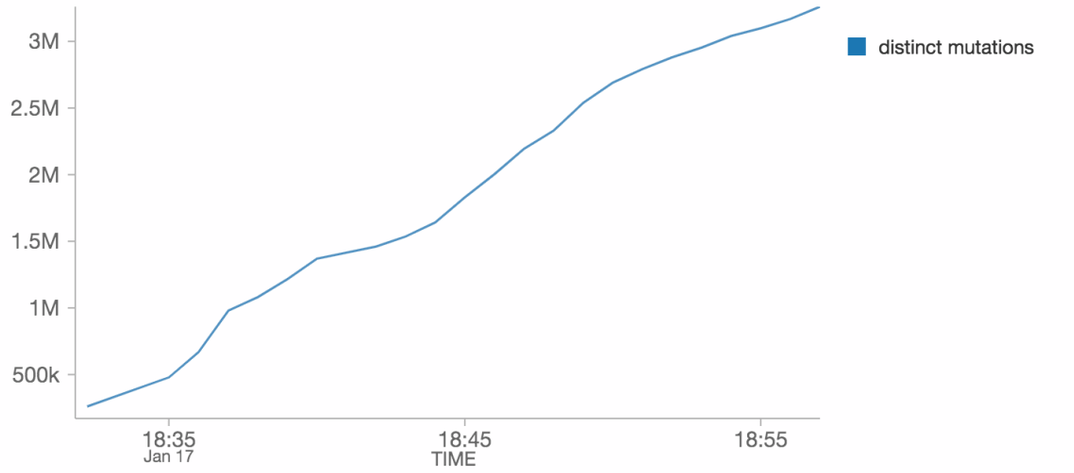
With Delta Lake, any new data is transactionally consistent in each and every step of our genomics pipeline. This is important because it ensures your pipeline is consistent (maintains consistency of your data, i.e. ensures all of the data is "correct"), reliable (either the transaction succeeds or fails completely), and can handle real-time updates (the ability to handle many transactions concurrently and any outage of failure will not impact the data). Thus even the data in our downstream amino acid substitution map (which had a number of additional ETL steps) is refreshed seamlessly.



As the last step of our genomics pipeline, we are also monitoring the distinct mutations by reviewing the Delta Lake parquet files within DBFS (i.e. increase of distinct mutations over time).

```
1 display(spark.read.parquet("dbfs:/wbrandler/dnaseq/demo/data/variant_count_test_parquet"). \
2           withColumnRenamed("count", "distinct mutations")
3         )
```

▸ (3) Spark Jobs



PART 1: SUMMARY

Using Delta Lake, bioinformaticians and researchers can apply distributed analytics with transactional consistency. These abstractions allow data practitioners to simplify genomics pipelines. Here we have created a genomic sample quality control pipeline that continuously processes data as new samples are processed, without manual intervention. Whether you are performing ETL or performing sophisticated analytics, your data will flow through your genomics pipeline rapidly and without disruption. Try it yourself today by downloading the Simplifying Genomics Pipelines at Scale with Delta Lake notebook.

PART 2: ACCURATELY BUILDING GENOMICS COHORTS AT SCALE

At Databricks we have leveraged innovations in distributed computation, storage, and cloud infrastructure and applied them to genomics to help solve problems that have hindered the ability for organizations to perform joint-genotyping, the "N + 1" problem, and the challenge of scaling to population-level cohorts.

In this blog, we explore how to apply those innovations to joint genotyping.

Before we dive into joint genotyping, first let's discuss why people do large scale sequencing. Most people are familiar with the genetic data produced by 23andMe or AncestryDNA. These tests use genotyping arrays, which read a fixed number of variants in the genome, typically ~1,000,000 well-known variants which occur commonly in the normal human population. With sequencing, we get an unbiased picture of all the variants an individual has, whether they are variants we've seen many times before in healthy humans or variants that we've never seen before that contribute to or protect against diseases. Figure 1 demonstrates the difference between these two approaches.

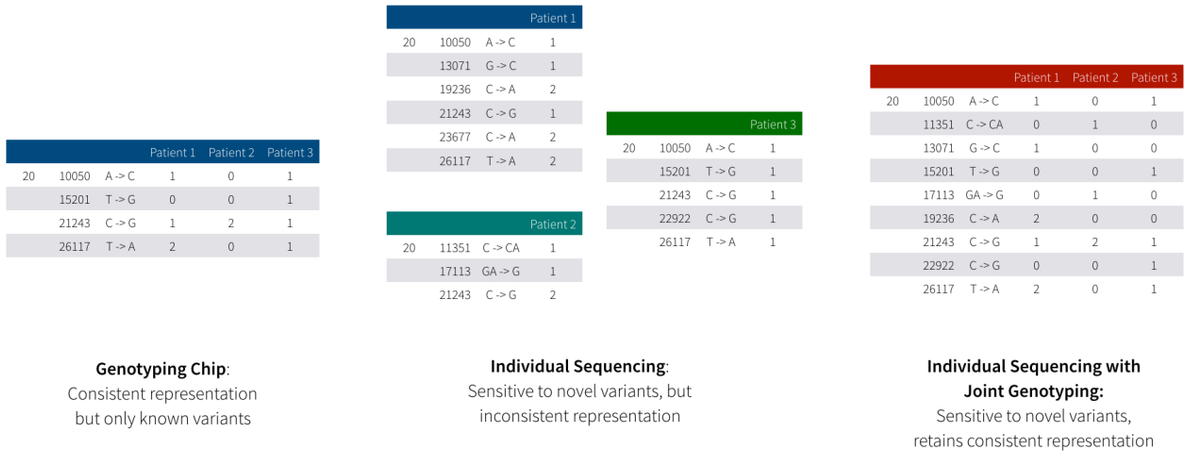


Figure 1: *This diagram illustrates the difference between variation data produced by genotype arrays (left) and by sequencing (middle) followed by joint genotyping (right). Genotyping arrays are restricted to "read" a fixed number of known variants, but guarantee a genotype for every sample at every variant. In sequencing, we are able to discover variants that are so rare that they only exist in a single individual, but determining if a novel variant is truly unique in this person or just hard to detect with current technology is a non-trivial problem.*

While sequencing provides much higher resolution, we encounter a problem when trying to examine the effect of a genetic variant across many patients. Since a genotyping array measures the same variants across all samples, looking across many individuals is a straightforward proposition: all variants have been measured across all individuals. When working with sequencing data, we have a trickier proposition: if we saw a variant in patient 1, but didn't see that variant in patient 2, what does that tell us? Did patient 2 not have an allele of that variant? Alternatively, when we sequenced patient 2, did an error occur that caused the sequencer to not read the variant we are interested in?

Joint genotyping addresses this problem in three separate ways:

1. Combining evidence from multiple samples enables us to rescue variants which do not meet strict statistical significance to be detected accurately in a single sample
2. As the accuracy of your predictions at each site in the human genome increases, you are better able to model sequencing errors and filter spurious variants
3. Joint genotyping provides a common variant representation across all samples that simplifies asking whether a variant in individual X is also present in individual Y

ACCURATELY IDENTIFYING GENETIC VARIANTS AT SCALE WITH JOINT GENOTYPING

Joint genotyping works by pooling data together from all of the individuals in our study when computing the likelihood for each individual's genotype. This provides us a uniform representation of how many copies of each variant are present in each individual, a key stepping stone for looking at the link between a genetic variant and disease. When we compute these new likelihoods, we are also able to compute a prior probability distribution for a given variant appearing in a population, which we can use to disambiguate borderline variant calls.

For a more concrete example, table 1 shows the precision and recall statistics for indel (insertions/deletions) and single-nucleotide variants (SNVs) for the sample HG002 called via the GATK variant calling pipeline compared to the Genome-in-a-Bottle (GIAB) high-confidence variant calls in high-confidence regions.

	Recall	Precision
Indel	**96.25%**	**98.32%**
SNV	**99.72%**	**99.40%**

Table 1: *Variant calling accuracy for HG002, processed as a single sample*

As a contrast, table 2 shows improvement for indel precision and recall and SNV recall when we jointly call variants across HG002 and two relatives (HG003 and HG004). The halving of this error rate is significant, especially for clinical applications.

	Recall	Precision
Indel	**98.21%**	**98.98%**
SNV	**99.78%**	**99.34%**

Table 2: Variant calling accuracy for HG002 following joint genotyping with HG003 and HG004

Originally, joint genotyping was performed directly from the raw sequencing data for all individuals, but as studies have grown to petabytes in size, this approach has become impractical. Modern approaches start from a genome variant call file (gVCF), a tab-delimited file containing all variants seen in a single sample, and information about the quality of the sequencing data at every position where no variant was seen. While the gVCF-based approach touches less data than looking at the sequences, a moderately sized project can still have tens of terabytes of gVCF data. This gVCF-based approach eliminates the need to go back to the raw reads but still requires all N+1 gVCFs to be reprocessed when jointly calling variants with a new sample added to the cohort. Our approach uses Delta Lakes to enable incrementally squaring-off the N+1 samples in the cohort, while parallelizing regenotyping using Apache Spark™.

CHALLENGES CALLING VARIANTS AT A POPULATION LEVEL

Despite the importance of joint variant calling, bioinformatics teams often defer this step because the existing infrastructure around GATK4 makes these workloads hard to run and even harder to scale. The default implementation of the GATK4's joint genotyping algorithm is single threaded, and scaling this implementation relies on manually parallelizing the joint genotyping kernel using a workflow language and runners like WDL and Cromwell. While GATK4 has support for a Spark-based HaplotypeCaller, it does not support running GenotypeGVCFs parallelized using Spark. Additionally, for scalability, the GATK4 best practice joint genotyping workflow relies on storing data in GenomicsDB. Unfortunately, GenomicsDB has limited support for cloud storage systems like AWS S3 or Azure Blob Storage, and studies have demonstrated that the new GenomicsDB workflow is slower than the old CombineGVCFs/GenotypeGVCFs workflow on some large datasets.

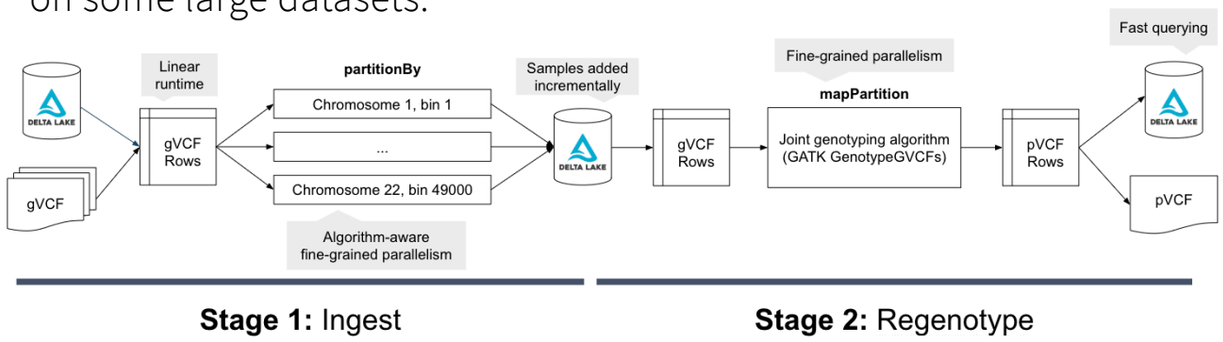


Figure 2: *The computational flow of the Databricks joint genotyping pipeline. In stage 1, the gVCF data is ingested into a Delta Lake columnar store in a scheme partitioned by a genomic bin. This Delta Lake table can be incrementally updated as new samples arrive. In stage 2, we then load the variants and reference models from the Delta Lake tables and directly run the core re-genotyping algorithm from the GATK4's GenotypeGVCFs tool. The final squared off genotype matrix is saved to Delta Lake by default, but can also be written out as VCF.on some large datasets.*

OUR SOLUTION

The Unified Analytics Platform for Genomics' Joint Genotyping Pipeline (Azure | AWS) provides a solution for these common needs. Figure 2 shows the computational architecture of the joint genotyping pipeline. This pipeline is provided as a notebook (Azure | AWS) that can be called as a Databricks job, the joint variant calling pipeline is simple to run: the user simply needs to provide their input files and output directory. When the pipeline runs, it starts by appending the input gVCF data via our VCF reader (Azure | AWS) to Delta Lake. Delta Lake provides inexpensive incremental updates, which makes it cheap to add an N+1th sample into an existing cohort. When the pipeline runs, it uses Spark SQL to bin the variant calls. The joint variant calling algorithm then runs in parallel over each bin, scaling linearly with the number of variants.

The parallel joint variant calling step is implemented through Spark SQL, using a similar architecture to our DNASeq and TNSeq pipelines. Specifically, we bin all of the input genotypes/reference models from the gVCF files into contiguous regions of the reference genome. Within each bin, we then sort the data by reference position and sample ID. We then directly invoke the joint genotyping algorithm from the GATK4's GenotypeGVCFs tool over the sorted iterator for the genomic bin. We then save this data out to a Delta table, and optionally as a VCF file. For more detail on the technical implementation, see our Spark Summit 2019 talk.

BENCHMARKING

To benchmark our approach, we used the low-coverage WGS data from the 1000 Genomes project for scale testing, and data from the Genome-in-a-Bottle consortium for accuracy benchmarking. To generate input gVCF files, we aligned and called variants using our DNASeq pipeline. Figure 3 demonstrates that our approach is efficiently scalable with both dataset and cluster size. With this architecture, we are able to jointly call variants across the 2,504 sample whole genome sequencing data from the 1000 Genomes Project in 79 hours on 13 c5.9xlarge machines. To date, we have worked with customers to scale this pipeline across projects with more than 3,000 whole genome samples.

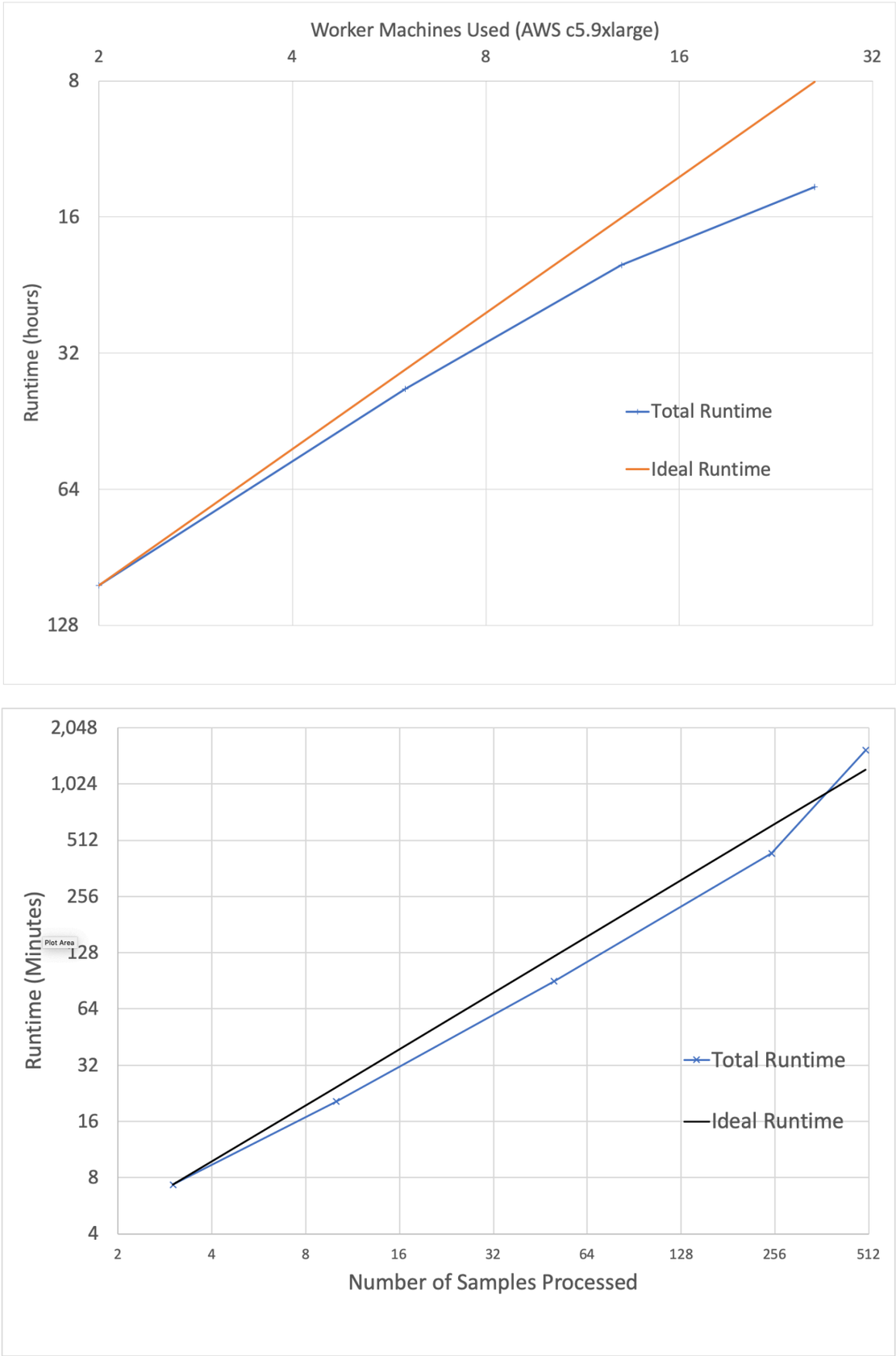


Figure 3 (right panel): *Strong scaling (top) was evaluated by holding the input data constant at 10 samples and increasing the number of executors. Weak scaling (bottom) was evaluated by holding the cluster size fixed at 13 i3.8xlarge workers and increasing the number of gVCFs processed.*

Running with default settings, the pipeline is highly concordant with the GATK4 joint variant calling pipeline on the HG002, HG003 and HG004 trio. Table 3 describes concordance at the variant and genotype level when comparing our pipeline against the "ground truth" of the GATK4 WDL workflow for joint genotyping. Variant concordance implies that the same variant was called across both tools; a variant called by our joint genotyper only is a false positive, while a variant called only by the GATK GenotypeGVCFs workflow is a false negative. Genotype concordance is computed across all variants that were called by both tools. A genotype call is treated as a false positive relative to the GATK if the count of called alternate alleles increased in our pipeline, and as a false negative if the count of called alternate alleles decreased.

	Precision	Recall
Variant	**99.9985%**	**99.9982%**
Genotype	**99.9988%**	**99.9992%**

Table 3: *Concordance Statistics Comparing Open-source GATK4 vs. Databricks for Joint Genotyping Workflows*

The discordant calls generally occur at locations in the genome that have a high number of observed alleles in the input gVCF files. At these sites, the GATK discards some alternate alleles to reduce the cost of re-genotyping. However, the alleles eliminated in this process depends on the sequence that variants are read. Our approach is to list the alternate alleles in the lexicographical order of the sample names prior to pruning. This approach ensures that the output variant calls are consistent given the same sample names and variant sites, regardless of how the computation is parallelized.

Of additional note, our joint genotyping implementation exposes a configuration option (Azure | AWS) which "squares off" and re-genotypes the input data without adjusting the genotypes by a prior probability which is computed from the data. This feature was requested by customers who are concerned about the deflation of rare variants caused by the prior probability model used in the GATK4.

SUMMARY

- Using Delta Lake and Spark SQL, we have been able to develop an easy-to-use, fast, and scalable joint variant calling framework on Databricks. Read our technical docs to learn how to get started (Azure | AWS).
- Learn more about our Databricks Unified Analytics Platform for Genomics and try out a preview today.

Real-Time Display Advertising Attribution

DOWNLOAD THE NOTEBOOK AND TRY IT OUT

In digital advertising, one of the most important things to be able to deliver to clients is information about how their advertising spend drove results. The more quickly we can provide this, the better.

To tie conversions or engagements to the impressions served in an advertising campaign, companies must perform attribution. Attribution can be a fairly expensive process, and running attribution against constantly updating datasets is challenging without the right technology. Traditionally, this has not been an easy problem to solve as there are lots of things to reason about:

- How do we make sure the data can be written at low latency to a read location without corrupting records?
- How can we continuously append to a large, query-able dataset without exploding costs or loss of performance?
- And where and how should I introduce a join for attribution?

Fortunately, this is easy with Structured Streaming and Delta Lake. In this section (and associated notebook), we are going to take a quick look at how to use the DataFrame API to build Structured Streaming applications on top of Kinesis (for those using Azure Databricks, you can use Azure EventHubs, Apache Kafka on HDInsight, or Azure Cosmos DB integration), and use Delta Lake to query the streams in near-real-time. We will also show how you can use the BI tool of your choice to review your attribution data in real-time.

DEFINE STREAMS

The first thing we will need to do is to establish the impression and conversion data streams. The impression data stream provides us a real- time view of the attributes associated with those customers who were served the digital ad (impression) while the conversion stream denotes customers who have performed an action (e.g. click the ad, purchased an item, etc.) based on that ad.
You can quickly plug into the stream as Databricks supports direct connectivity to Kafka (Apache Kafka, Apache Kafka on AWS, Apache Kafka on HDInsight) and Kinesis as noted in the following code snippet (this is for impressions, repeat this step for conversions).

```
// Read impressions stream
val kinesis = spark.readStream
  .format("kinesis")
  .option("streamName", kinesisStreamName)
  .option("region", kinesisRegion)
  .option("initialPosition", "latest")
  .option("awsAccessKey", $awsAccessKeyId$)
  .option("awsSecretKey", $awsSecretKey$)
  .load()
```

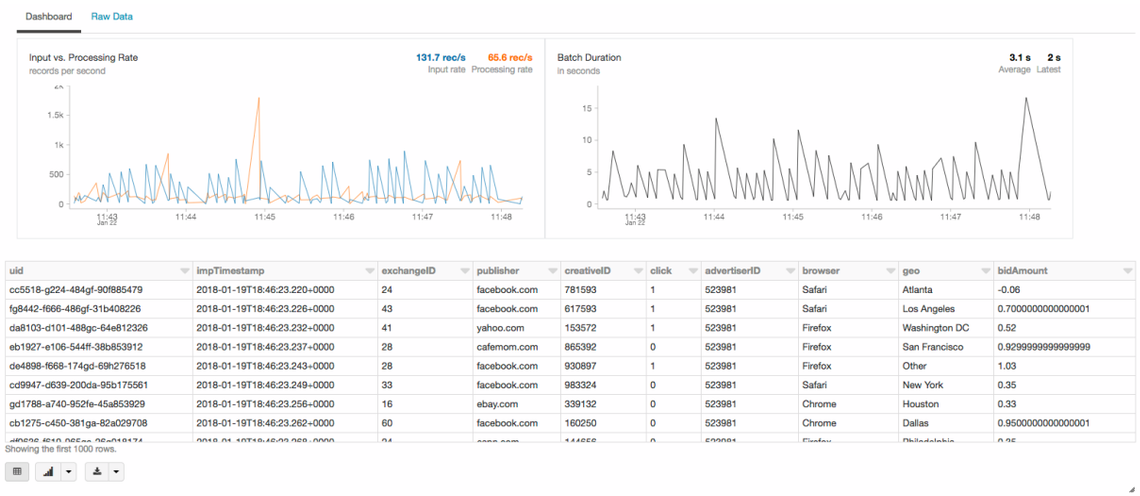
Next, create data streams schema as noted in the following snippet.

```
// Define impressions stream schema
val schema = StructType(Seq(
    StructField("uid", StringType, true),
    StructField("impTimestamp", TimestampType, true),
    StructField("exchangeID", IntegerType, true),
    StructField("publisher", StringType, true),
    StructField("creativeID", IntegerType, true),
    StructField("click", StringType, true),
    StructField("advertiserID", IntegerType, true),
    StructField("browser", StringType, true),
    StructField("geo", StringType, true),
    StructField("bidAmount", DoubleType, true)
))
```

Finally, we will want to create our streaming impressions DataFrame. With the Databricks display command, we will see both the data and the input/ processing rate in real-time alongside our data.

```
// Define streaming impressions DataFrame
val imp =
kinesis.select(from_json('data.cast("string"), schema)
as "fields"').select($"fields.*")

// View impressions real-time data
display(imp)
```



uid	impTimestamp	exchangeID	publisher	creativeID	click	advertiserID	browser	geo	bidAmount
cc5518-g224-484gf-90f885479	2018-01-19T18:46:23.220+0000	24	facebook.com	781593	1	523981	Safari	Atlanta	-0.06
fg8442-f666-486gf-31b408226	2018-01-19T18:46:23.226+0000	43	facebook.com	617593	1	523981	Safari	Los Angeles	0.7000000000000001
da8103-d101-488gc-64e812326	2018-01-19T18:46:23.232+0000	41	yahoo.com	153572	1	523981	Firefox	Washington DC	0.52
eb1927-e106-544ff-38b853912	2018-01-19T18:46:23.237+0000	28	cafemom.com	865392	0	523981	Firefox	San Francisco	0.9299999999999999
de4898-f668-174gd-69h276518	2018-01-19T18:46:23.243+0000	28	facebook.com	930897	1	523981	Firefox	Other	1.03
cd9947-d639-200da-95b175561	2018-01-19T18:46:23.249+0000	33	facebook.com	983324	0	523981	Safari	New York	0.35
gd1788-a740-952fe-45a853929	2018-01-19T18:46:23.256+0000	16	ebay.com	339132	0	523981	Chrome	Houston	0.33
cb1275-c450-381ga-82a029708	2018-01-19T18:46:23.262+0000	60	facebook.com	160250	0	523981	Chrome	Dallas	0.9500000000000001

SYNC STREAMS TO DELTA LAKE

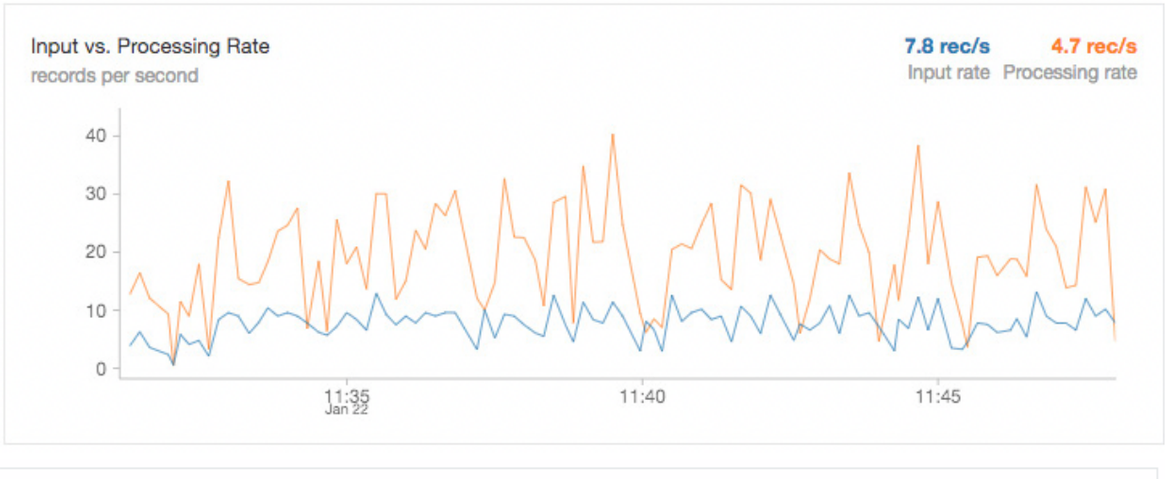
The impression (`imp`) and conversion (`conv`) streams can be synced directly to Delta Lake allowing us a greater degree of flexibility and scalability for this real-time attribution use-case. It allows you to quickly write these real-time data streams into Parquet format on S3 / Blob Storage while allowing users to read from the same directory simultaneously without the overhead of managing consistency, transactionality and performance yourself. As noted in the following code snippet, we're capturing the raw records from a single source and writing it into its own Delta Lake table.

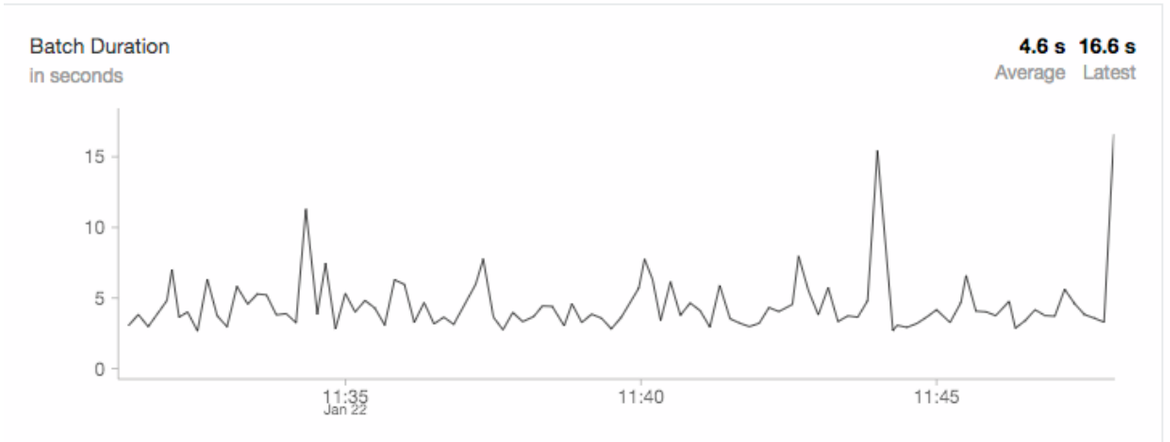
```
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.expressions.Window

// Persist Impression `imp` data to Databricks Delta
imp.withWatermark("impTimestamp", "1 minute")
  .repartition(1)
  .writeStream
  .format("delta")
  .option("path","/tmp/adtech/impressions")
  .option("checkpointLocation","/tmp/adtech/
impressions-checkpoints")
  .trigger(org.apache.spark.sql.streaming.Trigger.Proc
essingTime("10 seconds")).start()
```

It is important to note that with Delta Lake, you can also:

- Apply additional ETL, analytics, and/or enrichment steps at this point
- Write data from different streams or batch process and different sources into the same table





DELTA LAKE VIEWS FOR AD HOC REPORTING

Now that we have created our impression and conversion Delta Lake tables, we will create named views so we can easily execute our joins in Spark SQL as well as make this data available to query from your favorite BI tool. Let's first start with creating our Delta Lake views.

```
%sql
use adtech;
create or replace view impressionsDelta
as select *
from delta.`/tmp/adtech/impressions`;
create or replace view conversionsDelta
as select *
from delta.`/tmp/adtech/conversions`;
```

Now that we have established our Delta Lake views, we can calculate our last touch attribution on the view and then calculate *weighted attribution* on the view.

CALCULATE LAST TOUCH ATTRIBUTION ON VIEW

To calculate the real-time window attribution, as noted in the preceding sections, we will need to join two different Delta Lake streams of data: *impressions* and *conversions*. As noted in the following code snippet, we will first define our *Delta Lake-based impressions* and *conversions*. We will also define window specification which will be used by the following `dense_rank()` statement. The window and rank define our attribution logic.

```
# Define needed Impression data
val imps = spark.sql("select uid as impUid,
advertiserID as impAdv, * fromsparksummit.imps")
  .drop("advertiserID")

// Define needed Conversions Data
val convs = spark.sql("select * from
sparksummit.convs")

// Define Spark SQL window ordered by Impression
// Timestamp partitioned by Impression Uid and
// Impression Advertisor
val windowSpec = Window.partitionBy("impUid","impAdv")
  .orderBy(desc("impTimestamp"))

// Calculate real-time attribution by joining
// impression.impUid == conversion.uid to ensure
// impression time happened before conversion time
// filtering via dense_rank
val windowedAttribution = convs.join(imps,
imps.col("impUid") === convs.col("uid") &&
imps.col("impTimestamp") < convs.col("convTimestamp")
&& imps.col("impAdv") === convs.col("advertiserID"))
  .withColumn("dense_rank", dense_rank()
    .over(windowSpec))
  .filter($"dense_rank"===1)

// Create global temporary view
windowedAttribution.createGlobalTempView(
  "realTimeAttribution")
```

Now we can calculate our real-time window attribution by joining the impression and conversion data together but filtering for the most recent impression user id (as defined by `impUid`). Finally, we can create a global temporary view (`.createGlobalTempView`) so this real-time view is accessible by downstream systems.

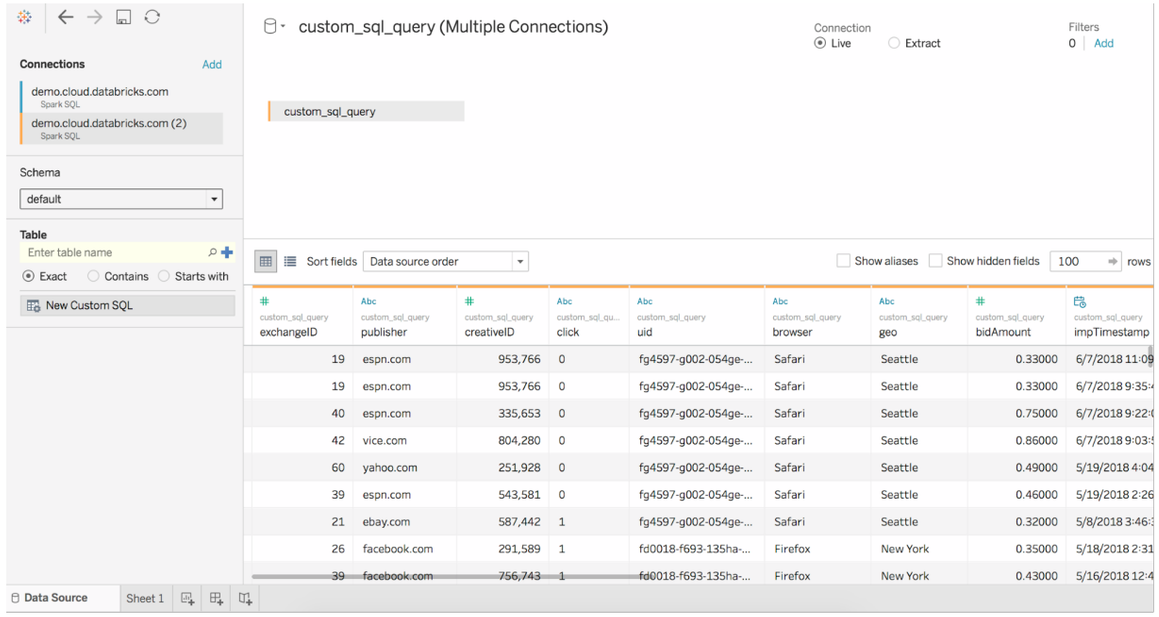
For example, you can view your real-time data using Spark SQL in the following code snippet.

```
%sql
select * from global_temp.realTimeAttribution
```

▸ (7) Spark Jobs

uid	convTimestamp	conversionID	advertiserID	pixelID	conversionValue	impUid	impAdv	uid	impTimestamp	exchangeID	publisher	creativeID	click	advertiserID	browser
gc1932-e367-422af-78a153015	2018-01-19T18:55:13.645+0000	89169	523981	103952	57.54	gc1932-e367-422af-78a153015	523981	gc1932-e367-422af-78a153015	2018-01-19T18:56:59.297+0000	24	facebook.com	442957	1	523981	Internet Explorer
fe3576-e384-785ec-92g750761	2018-01-19T18:55:08.565+0000	50320	523981	103952	22.080000000000002	fe3576-e384-785ec-92g750761	523981	fe3576-e384-785ec-92g750761	2018-01-19T18:55:14.349+0000	35	ebay.com	481457	0	523981	Chrome
fe3576-e384-785ec-	2018-01-19T18:55:03.565+0000	93331	523981	103952	37.269999999999996	fe3576-e384-785ec-	523981	fe3576-e384-785ec-	2018-01-19T18:55:14.349+0000	35	ebay.com	481457	0	523981	Chrome

As well, you can plug in your favorite BI tool such as Tableau to perform ad-hoc analysis of your data.



CALCULATE WEIGHTED ATTRIBUTION ON VIEW

In the preceding case, we demonstrate a very naive model — isolating all of a user's impressions prior to conversion, selecting the most recent, and attributing only the most recent impression prior to conversion. A more sophisticated model might apply attribution windows or weight the impressions by time such as the code snippet below.

```
// Define attribution window for impressions
val attrWindow = Window \
  .partitionBy("uid")
  .orderBy($"impTimestamp".desc)
val attrRank = dense_rank().over(attrWindow)

// Define ranked window by taking attribution window
// and partition by conversionID
val rankedWindow = Window.partitionBy("conversionID")
val numAttrImps = \
  max(col("attrRank")).over(rankedWindow)

// Reference impression table
val imps = spark \
  .sql("select * from sparksummit.imps") \
  .withColumn("date", $"impTimestamp" \
    .cast(DateType)) \
  .drop("advertiserID")

// Reference conversion table
val convs = spark \
  .sql("select * from sparksummit.convs")
  .withColumnRenamed("uid", "cuid")

// Join impression and conversions
val joined = imps.join(convs,
  imps.col("uid") === convs.col("cuid") &&
  imps.col("impTimestamp") <
  convs.col("convTimestamp")).drop("cuid")
```

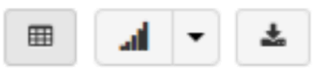
(continued on the next panel)

```
// Calculate weighted attribution
val attributed =joined \
  .withColumn("attrRank",attrRank) \
  .withColumn("numImps",numAttrImps) \
  .withColumn(
    "weightedRevAttr", $"conversionValue"/$"numImps")

// Create attributed temporary view
attributed.createOrReplaceTempView("attributed")

// Display data
display(spark.sql("select sum(weightedRevAttr),
  advertiserID from attributed group by advertiserID"))
```

sum(weightedRevAttr)	advertiserID
7846749.570946295	702394
4431214.068630882	489251
2200343.4445629856	295381
10097961.92226806	523981



SUMMARY

In this section, we have reviewed how Delta Lake provides a simplified solution for a real-time attribution pipeline. The advantages of using Delta Lake to sync and save your data streams include (but and not limited to) the ability to:

- Save and persist your real-time streaming data like a data warehouse because Delta Lake maintains a transaction log that efficiently tracks changes to the table.
- Yet still, have the ability to run your queries and perform your calculations in real-time and completing in seconds.
- With Delta Lake, you can have multiple writers simultaneously modify a dataset and still see consistent views.
- Writers can modify a dataset without interfering with jobs reading the dataset.

Together with your streaming framework and the Databricks Unified Analytics Platform, you can quickly build and use your real-time attribution pipeline with Delta Lake to solve your complex display advertising problems in real-time.

Mobile Gaming Data Event Processing

DOWNLOAD THE NOTEBOOK AND TRY IT OUT

The world of mobile gaming is fast paced and requires the ability to scale quickly. With millions of users around the world generating millions of events per second by means of game play, you will need to calculate key metrics (score adjustments, in-game purchases, in-game actions, etc.) in real-time. Just as important, a popular game launch or feature will increase event traffic by orders of magnitude and you will need infrastructure to handle this rapid scale.

With complexities of low-latency insights and rapidly scalable infrastructure, building data pipelines for high volume streaming use cases like mobile game analytics can be complex and
confusing. Developers who are tasked with this endeavor will encounter a number of architectural questions.

- First, what set of technologies they should consider that will reduce their learning curve and that integrate well?
- Second, how scalable will the architecture be when built?
- And finally, how will different personas in an organization collaborate?

Ultimately, they will need to build an end-to-end data pipeline comprised of these three functional components:

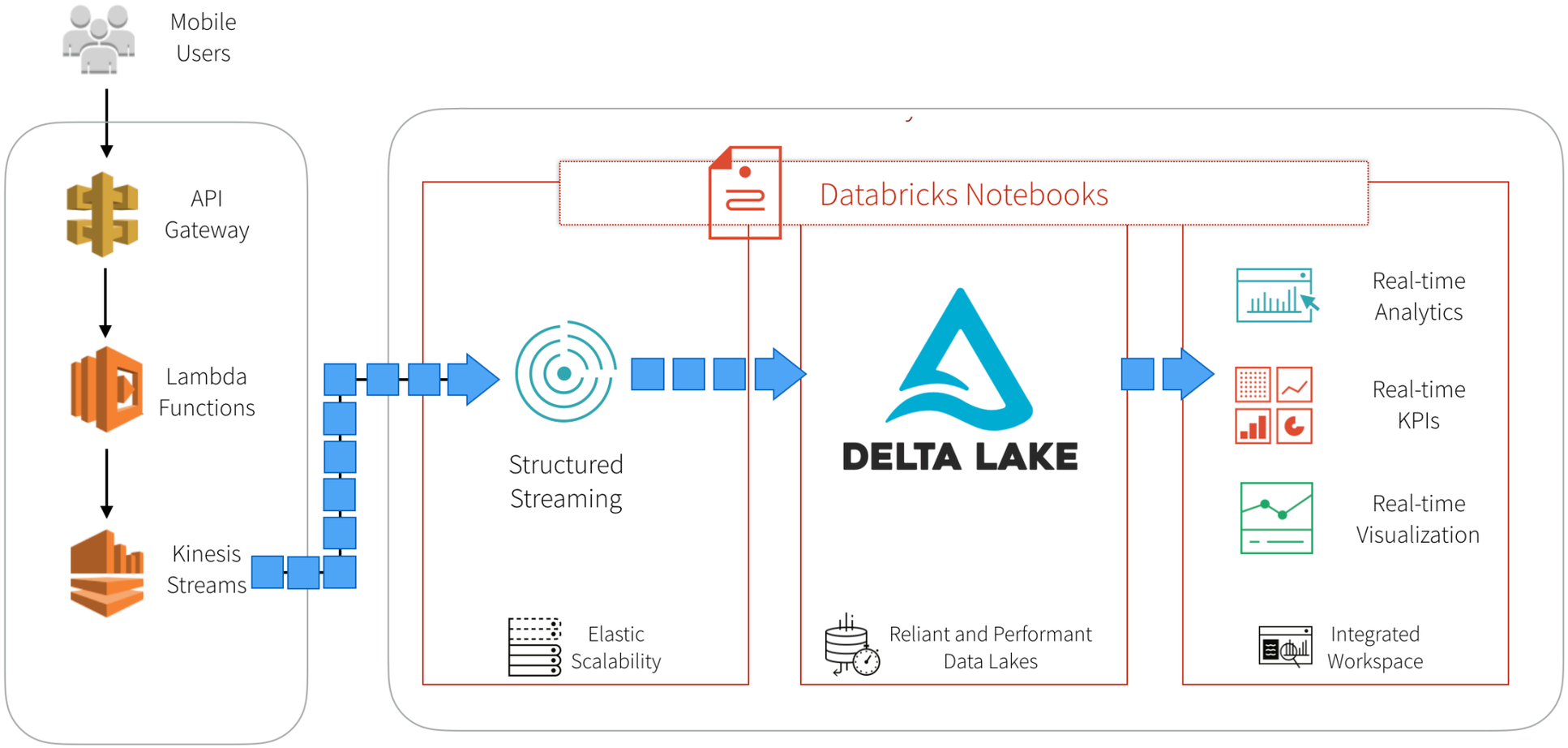
- data ingestion/streaming
- data transformation (ETL)
- and data analytics and visualization.

In this section, we will explore how to:

- Build a mobile gaming data pipeline using AWS services such as API Gateway, Lambda, and Kinesis Streams
- Build a stream ingestion service using Spark Structured Streaming
- Use Delta Lake as a sink for our streaming operations
- Explore how analytics can be performed directly on this table, minimizing data latency
- Illustrate how Delta Lake solves traditional issues with streaming data

HIGH LEVEL INFRASTRUCTURE COMPONENTS

Building mobile gaming data pipelines is complicated by the fact that you need rapidly scalable infrastructure to handle millions of events by millions of users and gain actionable insights in real-time. Thankfully, Kinesis shards can be dynamically re-provisioned to handle increased loads, and Databricks automatically scales out your cluster to handle the increase in data.

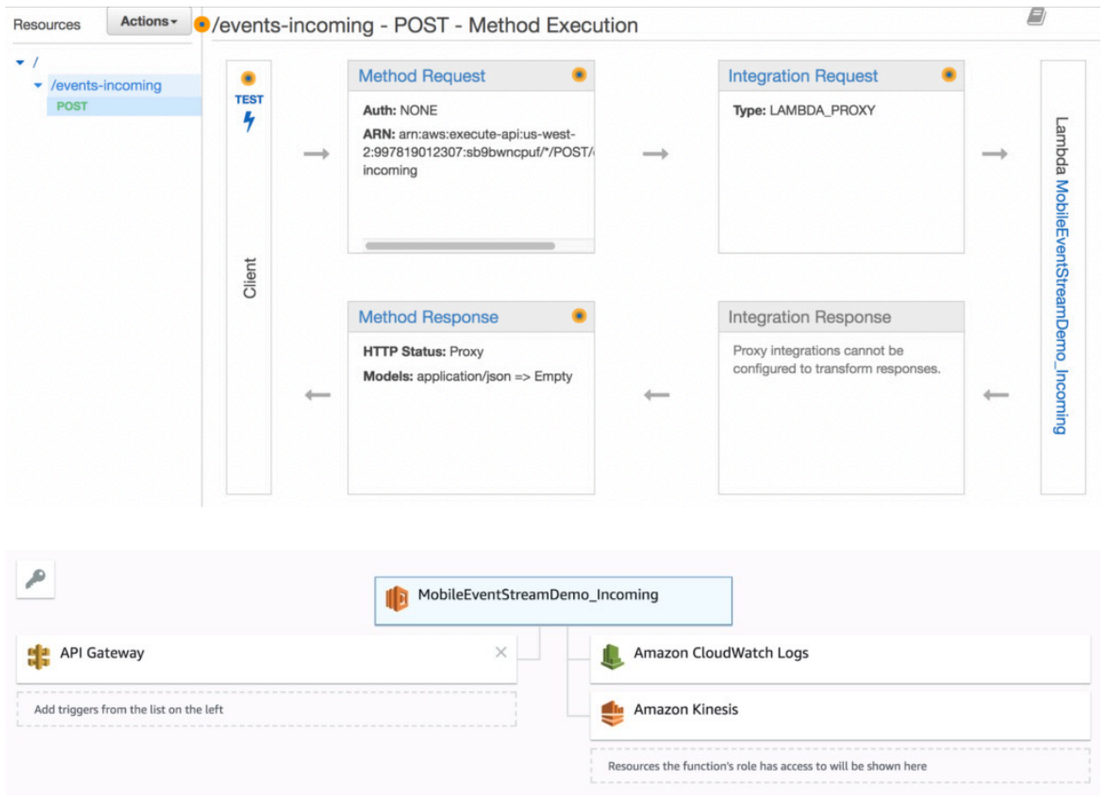


NOTE: The above architecture can also be achieved with open source software.

In our example, we simulate game play events from mobile users with an event generator. These events are pushed to a REST endpoint and follow our data pipeline through ingestion into our Delta Lake table. The code for this event generator can be found here.

AMAZON API GATEWAY, LAMBDA, AND KINESIS STREAMS

For this example, we build a REST endpoint using Amazon API Gateway. Events that arrive at this endpoint automatically trigger a serverless lambda function, which pipes these events into a Kinesis stream for our consumption. You will want to setup lambda integration with your endpoint to automatically trigger, and invoke a function that will write these events to Kinesis.

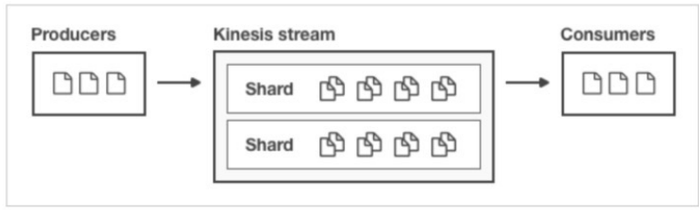


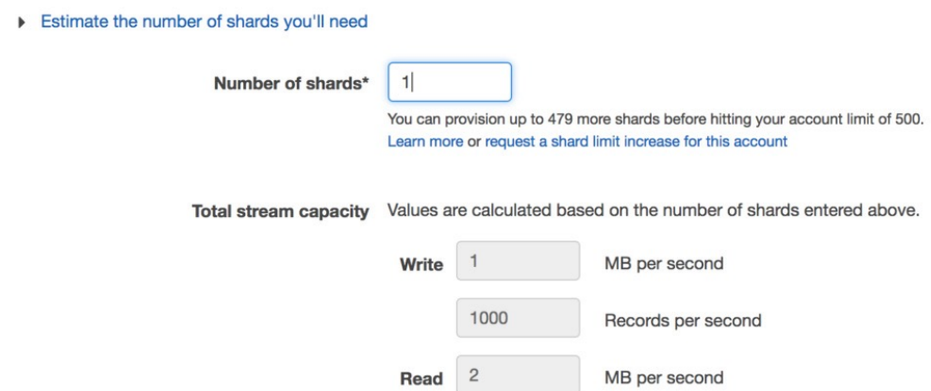
Setup a Python lambda function like so:

```python
import json
import boto3
import random
import base64
import time

def lambda_handler(event, context):
    stream_name = 'streamdemo_incoming'
    record = json.loads(event['body'])
    record['eventTime'] = int(time.time())
    event['body'] = record
    client = boto3.client('kinesis')
    client.put_record(StreamName = stream_name, \
      Data = json.dumps(event), \
      PartitionKey = str(random.randint(1,100)))
    return None
```

Kinesis streams are provisioned by throughput, so you can provision as many shards as necessary to handle your expected data throughput. Each shard provides a throughput of 1 MB/sec for writes and 2MB/sec for reads, or up to 1000 records per second. For more information regarding Kinesis streams throughput, check out the documentation. Random PartitionKeys are important for even distribution if you have more than one shard.





INGESTING FROM KINESIS USING STRUCTURED STREAMING

Ingesting data from a Kinesis stream is straightforward. In a production environment, you will want to setup the appropriate IAM role policies to make sure your cluster has access to your Kinesis Stream. The minimum permissions for this look like this:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "kinesis:DescribeStream",
                "kinesis:GetRecords",
                "kinesis:GetShardIterator"
            ],
            "Resource": "ARN_FOR_YOUR_STREAM"
        }
] }
```

Alternatively, you can also use AWS access keys and pass them in as options, however, IAM roles are best practice method for production use cases. In this example, let's assume the cluster has the appropriate IAM role setup.

Start by creating a DataFrame like this:

```
kinesisDataFrame = spark.readStream.format('kinesis') \
  .option('streamName','MY_KINESIS_STREAM_NAME') \
  .option('initialPosition','STREAM_POSITION') \
  .option('region','KINESIS_REGION') \
  .load()
```

You'll want to also define the schema of your incoming data. Kinesis data gets wrapped like so:

```
kinesisSchema = StructType() \
  .add('body', StringType()) \
  .add('resource', StringType()) \
  .add('requestContext',StringType()) \
  .add('queryStringParameters', StringType()) \
  .add('httpMethod', StringType()) \
  .add('pathParameters', StringType()) \
  .add('headers', StringType()) \
  .add('stageVariables', StringType()) \
  .add('path', StringType()) \
  .add('isBase64Encoded', StringType())

eventSchema = StructType() \
  .add('eventName', StringType()) \
  .add('eventTime', TimestampType()) \
    .add('eventParams', StructType() \
      .add('game_keyword', StringType()) \
      .add('app_name', StringType()) \
      .add('scoreAdjustment', IntegerType()) \
      .add('platform', StringType()) \
      .add('app_version', StringType()) \
      .add('device_id', StringType()) \
      .add('client_event_time', TimestampType()) \
      .add('amount', DoubleType()))
```

For this demo, we're really only interested in the body of the kinesisSchema, which will contain data that we describe in our `eventSchema`.

```
someEventDF = kinesisDataFrame.selectExpr(
    "cast (data as STRING) jsonData") \
  .select(from_json('jsonData',kinesisSchema) \
    .alias('requestBody')) \
  .select(from_json('requestBody.body', eventSchema) \
    .alias('body')) \
  .select('body.attr1', 'body.attr2', 'body.etc')
```

REAL-TIME DATA PIPELINES USING DELTA LAKE

Now that we have our streaming DataFrame defined, let's go ahead and do some simple transformations. Event data is usually time-series based, so it's best to partition on something like an event date. Our incoming stream does not have an event date parameter, however, so we'll make our own by transforming the `eventTime` column. We'll also throw in a check to make sure the eventTime is not null:

```
base_path = '/path/to/mobile_events_stream/'
eventsStream = gamingEventDF \
  .filter(gamingEventDF.eventTime.isNotNull()) \
  .withColumn("eventDate", \
    to_date(gamingEventDF.eventTime)) \
  .writeStream \
  .partitionBy('eventDate') \
  .format('delta') \
  .option('checkpointLocation', base_path + \
    '/_checkpoint') \
  .start(base_path)
```

Let's also take this opportunity to define our table location.

```
CREATE TABLE
IF NOT EXISTS mobile_events_delta_raw
USING DELTA
location '/path/to/mobile_events_stream/';
```

NOTE: DDL is part of the roadmap and will be available in Delta Lake in the near future.

REAL-TIME ANALYTICS, KPIS, AND VISUALIZATION

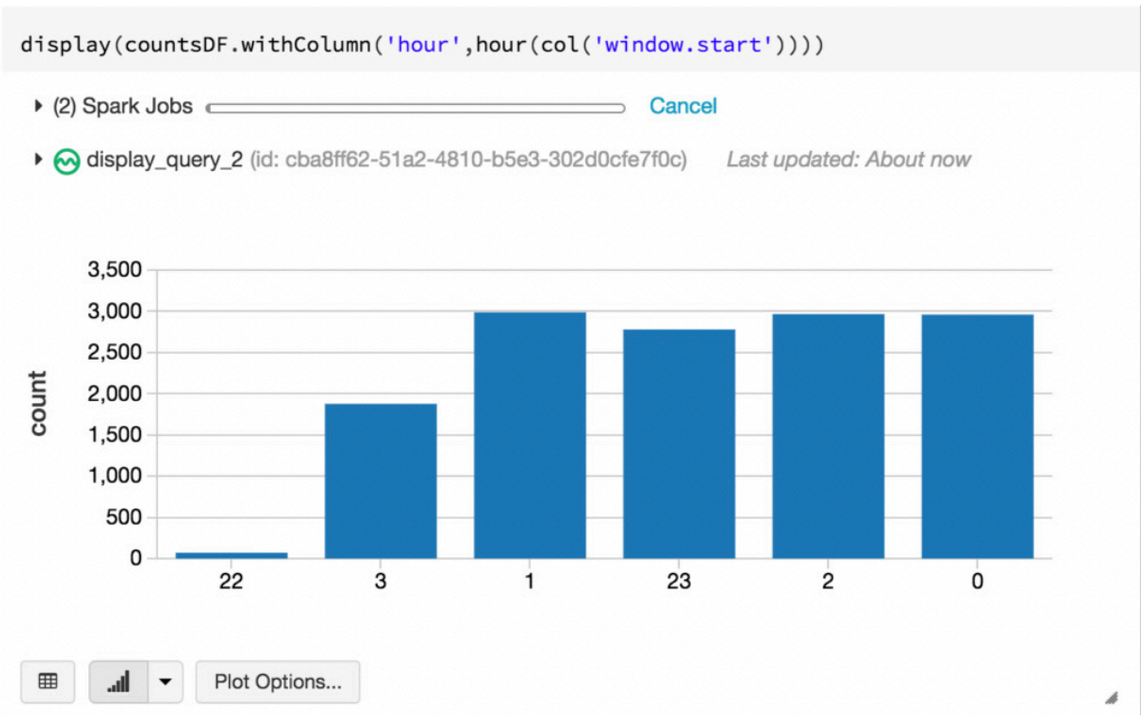
Now that we have data streaming live into our Delta Lake table, we can go ahead and look at some KPIs. Traditionally, companies would only look at these on a daily basis, but with Structured Streaming and Delta Lake, you have the capability to visualize these in real time all within your Databricks notebooks.

Let's start with a simple one. How many events have I seen in the last hour?

```
countsDF = gamingEventDF \
  .withWatermark("eventTime", "180 minutes") \
  .groupBy(window("eventTime", "60 minute")) \
  .count()
countsQuery = countsDF \
  .writeStream \
  .format('memory') \
  .queryName('incoming_events_counts') \
  .start()
```

We can then visualize this in our notebook as say, a bar graph:

```
display(countsDF.withColumn('hour',hour(col('window.start'))))
```



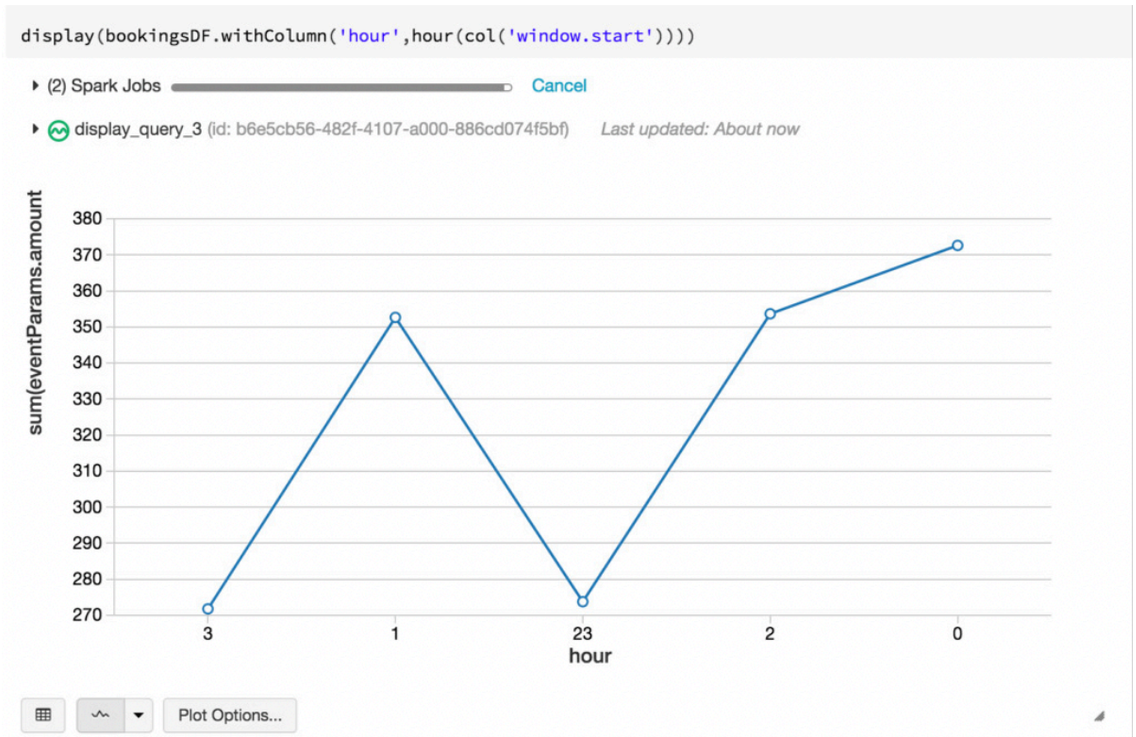
Maybe we can make things a little more interesting. How much money have I made in the last hour? Let's examine bookings. Understanding bookings per hour is an important metric because it can be indicative of how our application/production systems are doing. If there was a sudden drop in bookings right after a new game patch was deployed, for example, we immediately know something is wrong.

We can take the same DataFrame, but filter on all purchaseEvents, grouping by a window of 60 minutes.

```
bookingsDF = gamingEventDF \
  .withWatermark("eventTime", "180 minutes") \
  .filter(gamingEventDF.eventName == 'purchaseEvent') \
  .groupBy(window("eventTime", "60 minute")) \
  .sum("eventParams.amount")

bookingsQuery = bookingsDF \
  .writeStream
  .format('memory') \
  .queryName('incoming_events_bookings') \
  .start()
```

Let's pick a line graph to visualize this one:



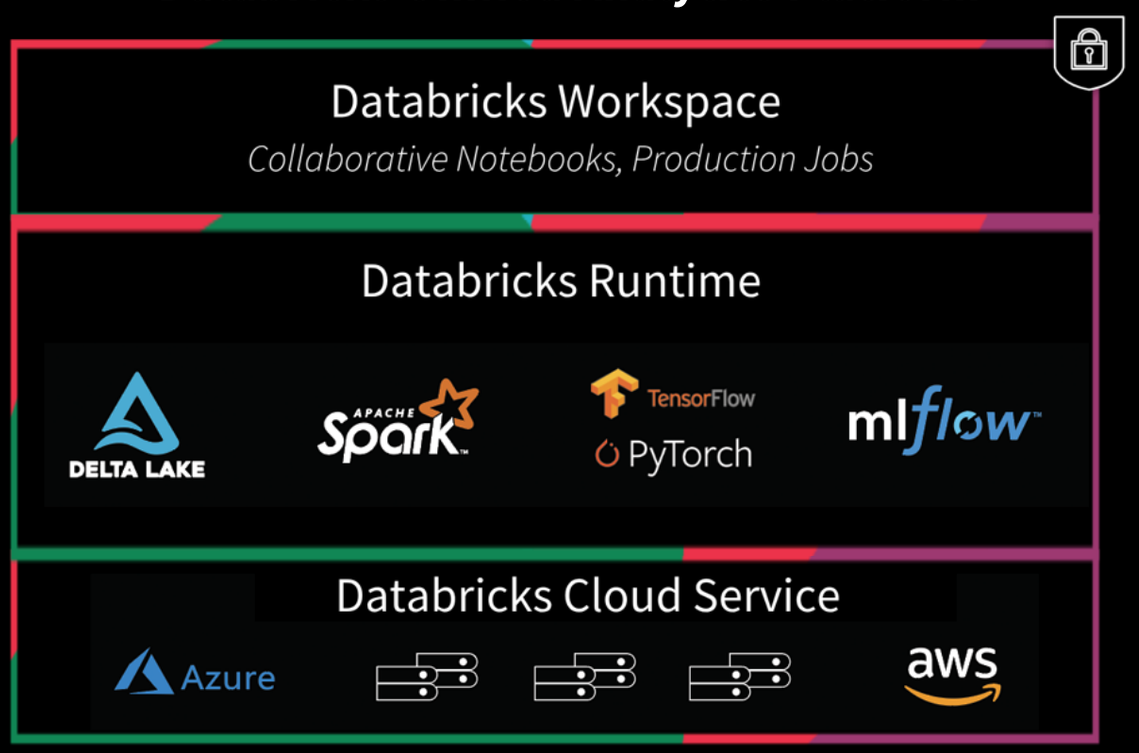
For the SQL enthusiasts, you can query the Delta Lake table directly. Let's take a look at a simple query to show the current daily active users (DAU). We know we're actually looking at device id because our sample set doesn't contain a user id, so for the sake of example, let's assume that there is a 1-1 mapping between users and devices (although, in the real world, this is not always the case).

```
select count (distinct eventParams.device_id)
as DAU
from mobile_events_delta_raw
where to_date(eventTime) = current_date;
```

SUMMARY

In closing, we demonstrated how to build a data pipeline's three functional components: data ingestion/streaming, data transformation, and data analytics and visualization. We've illustrated different ways that you can extrapolate key performance metrics from this real-time streaming data, as well as solve issues that are traditionally associated with streaming. The combination of Spark Structured Streaming and Delta Lake reduces the overall end-to-end latency and availability of data, enabling data engineering, data analytics, and data science teams to respond quickly to events like a sudden drop in bookings, or an increased error-message events, that have direct impact on revenue. Additionally, by removing the data engineering complexities commonly associated with such pipelines, this enables data engineering teams to focus on higher-value projects.

Learn More



Databricks, founded by the original creators of Apache Spark™, is on a mission to accelerate innovation for our customers by unifying data science, engineering and business teams. The Databricks Unified Analytics Platform powered by Apache Spark enables data professionals to manage the entire data analytics lifecycle from data ingestion and preparation through to model development, serving and analytics.

Innovations such as Delta Lake, for data lake reliability, and MLflow, for managing the complete machine learning lifecycle, are key enablers of the platform.

To learn how you can benefit from Delta Lake for building reliable data lakes:

SCHEDULE A PERSONALIZED DEMO

SIGN UP FOR A FREE TRIAL