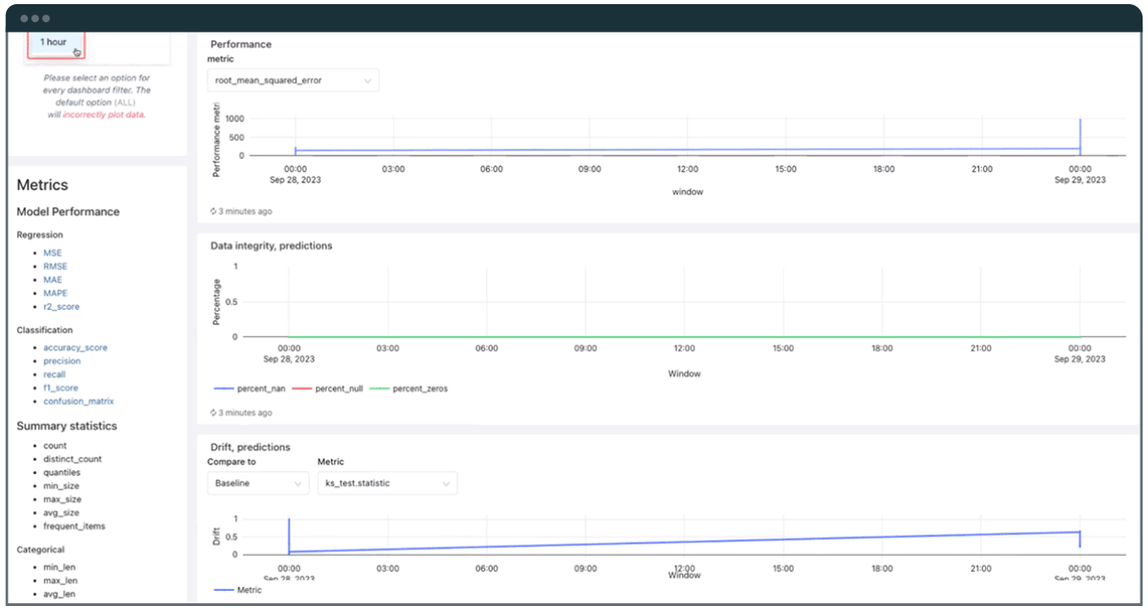
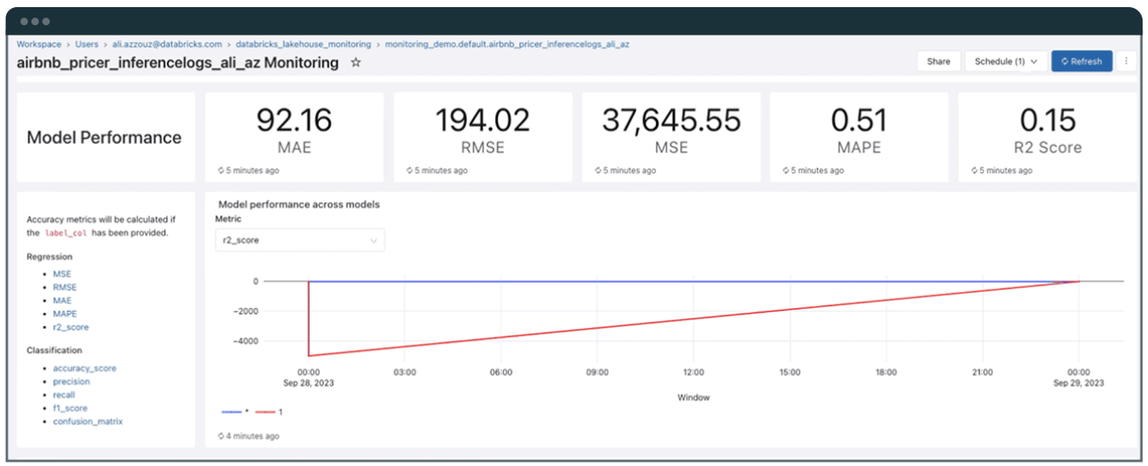
It further shows data integrity of predictions and data drift over time:

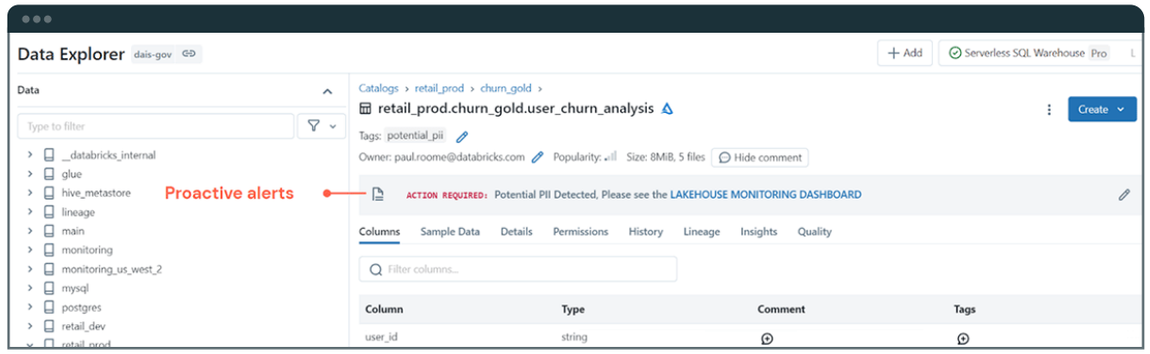


And model performance over time, according to a variety of ML metrics such as R2, RMSE, and MAPE:



Lakehouse Monitoring dashboards show data and AI assets quality

It's one thing to intentionally seek out ML model information when you are looking for answers, but it is a whole other level to get automated proactive alerts on errors, data drift, model failures, or quality issues. Below is an example alert for a potential PII (Personal Identifiable Information) data breach:



Example proactive alert of potential unmasked private data

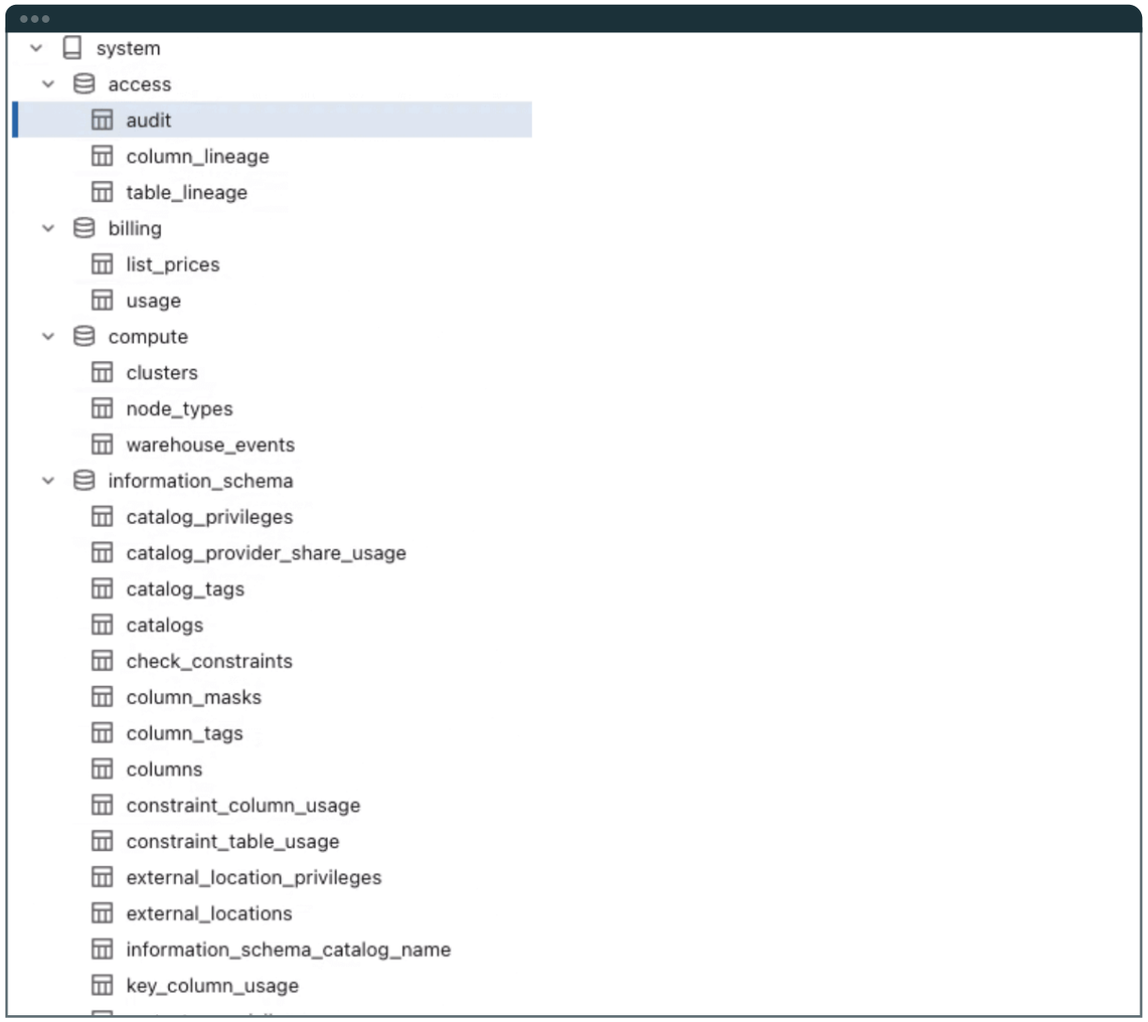
One more thing — you can assess the impact of issues, do a root cause analysis, and assess the downstream impact by Databrick's powerful lineage capabilities — from table-level to column-level.

System tables: metadata information for lakehouse observability and ensuring compliance

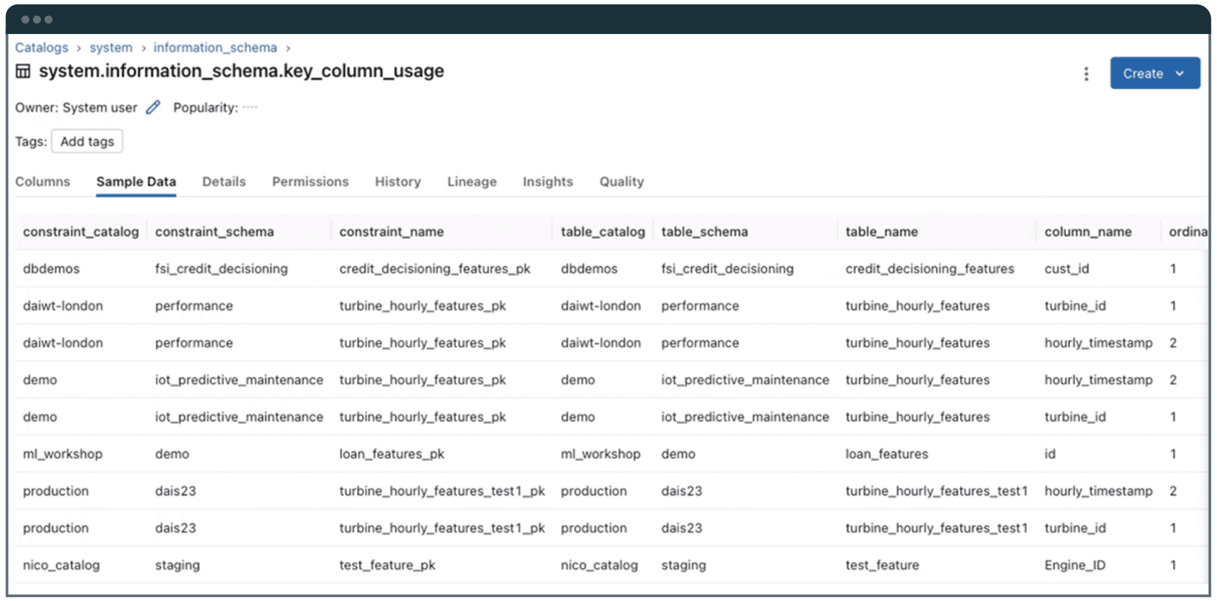
These underlying tables can be queried through SQL or activity dashboards to provide observability about every asset within the Databricks Intelligence Platform. Examples include which users have access to which data objects; billing tables that provide pricing and usage; compute tables that take cluster usage and warehouse events into consideration; and lineage information between columns and tables:

- Audit tables include information on a wide variety of UC events. UC captures an audit log of actions performed against the metastore giving administrators access to details about who accessed a given dataset and the actions that they performed.
- Billing and historical pricing tables will include records for all billable usage across the entire account; therefore you can view your account's global usage from whichever region your workspace is in.
- Table lineage and column lineage tables are great because they allow you to programmatically query lineage data to fuel decision making and reports. Table lineage records each read-and-write event on a UC table or path that might include job runs, notebook runs and dashboards associated with the table. For column lineage, data is captured by reading the column.
- Node types tables capture the currently available node types with their basic hardware information outlining the node type name, the number of vCPUs for the instance, and the number of GPUs and memory for the instance. Also in private preview are node_utilization metrics on how much usage each node is leveraging.
- Query history holds information on all SQL commands, i/o performance, and number of rows returned.
- Clusters table contains the full history of cluster configurations over time for all-purpose and job clusters.
- Predictive optimization tables are great because they optimize your data layout for peak performance and cost efficiency. The tables track the operation history of optimized tables by providing the catalog name, schema name, table name, and operation metrics about compaction and vacuuming.

From the catalog explorer, here are just a few of the system tables any of which can be viewed for more details:

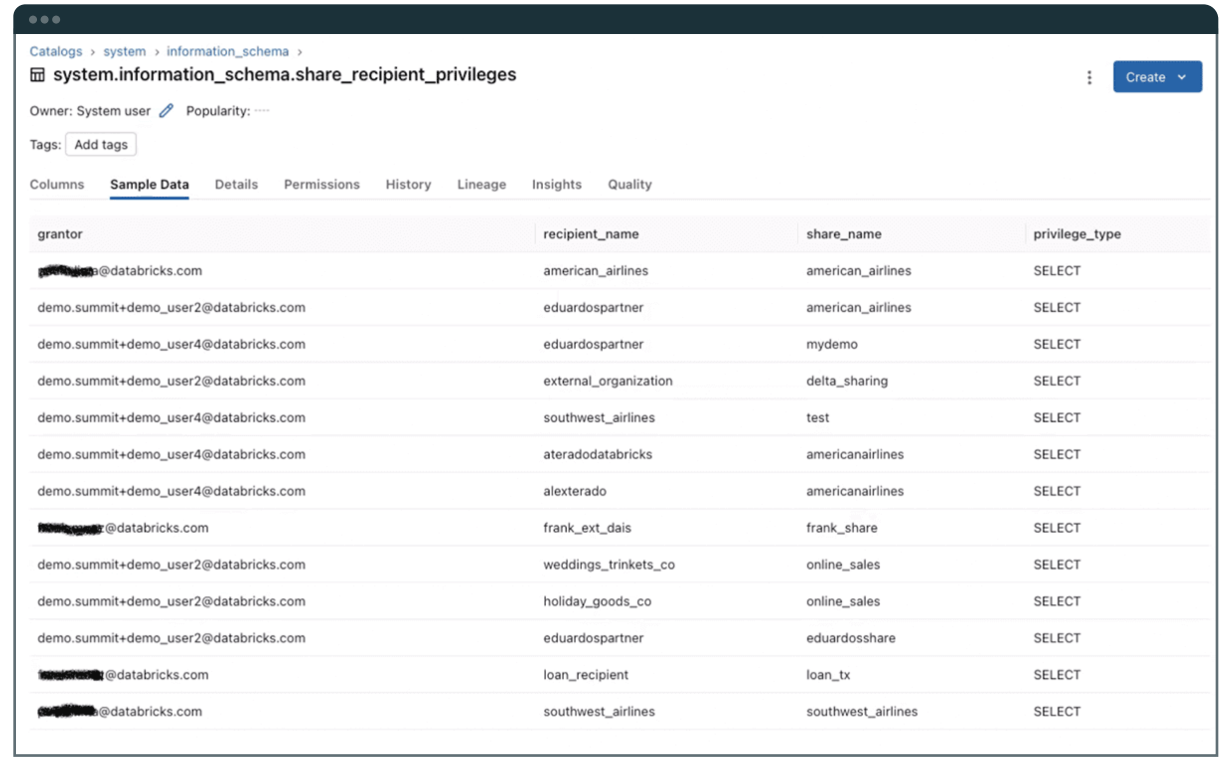


As an example, drilling down on the "key_column_usage" table, you can see precisely how tables relate to each other via their primary key:



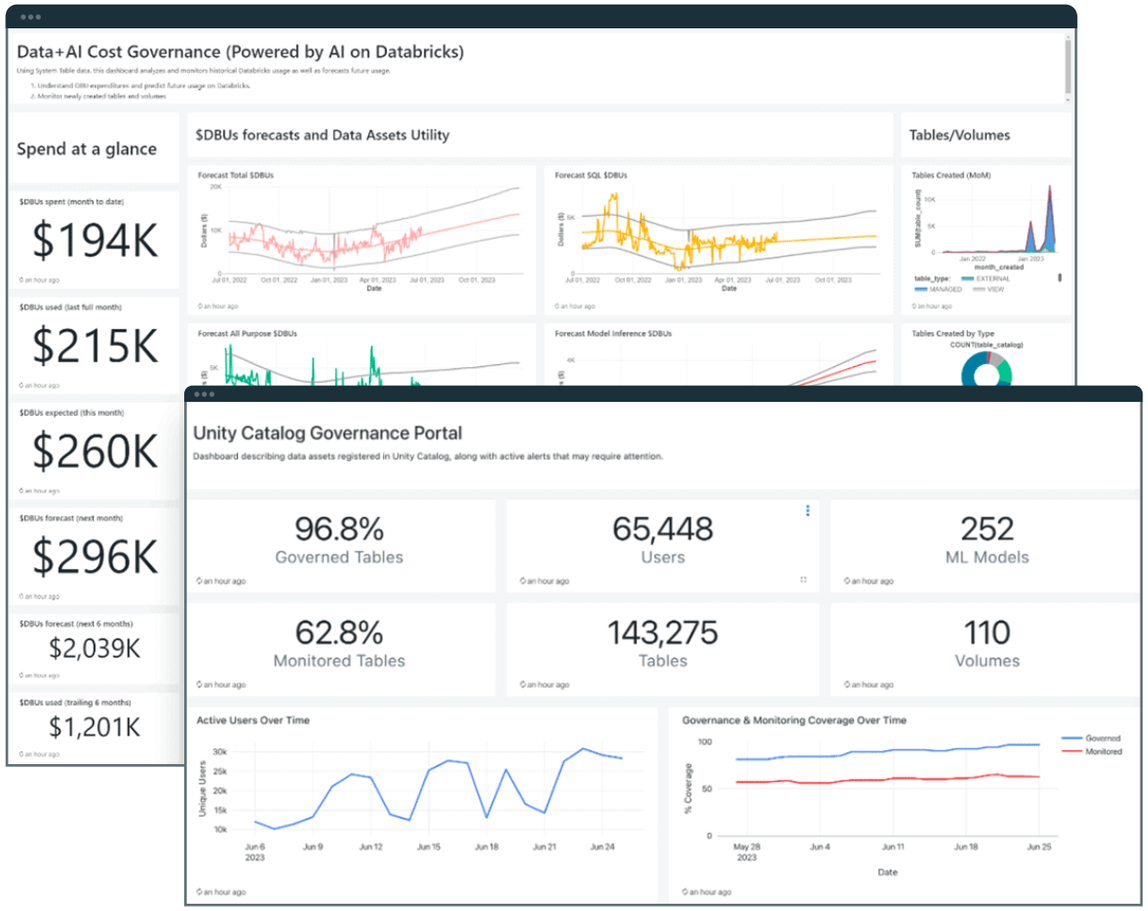
constraint_catalog	constraint_schema	constraint_name	table_catalog	table_schema	table_name	column_name	ordina
dbdemos	fsi_credit_decisioning	credit_decisioning_features_pk	dbdemos	fsi_credit_decisioning	credit_decisioning_features	cust_id	1
daiwt-london	performance	turbine_hourly_features_pk	daiwt-london	performance	turbine_hourly_features	turbine_id	1
daiwt-london	performance	turbine_hourly_features_pk	daiwt-london	performance	turbine_hourly_features	hourly_timestamp	2
demo	iot_predictive_maintenance	turbine_hourly_features_pk	demo	iot_predictive_maintenance	turbine_hourly_features	hourly_timestamp	2
demo	iot_predictive_maintenance	turbine_hourly_features_pk	demo	iot_predictive_maintenance	turbine_hourly_features	turbine_id	1
ml_workshop	demo	loan_features_pk	ml_workshop	demo	loan_features	id	1
production	dais23	turbine_hourly_features_test1_pk	production	dais23	turbine_hourly_features_test1	hourly_timestamp	2
production	dais23	turbine_hourly_features_test1_pk	production	dais23	turbine_hourly_features_test1	turbine_id	1
nico_catalog	staging	test_feature_pk	nico_catalog	staging	test_feature	Engine_ID	1

Another example is the "share_recipient_privileges" table, to see who granted which shares to whom:



grantor	recipient_name	share_name	privilege_type
[redacted]@databricks.com	american_airlines	american_airlines	SELECT
demo.summit+demo_user2@databricks.com	eduardospartner	american_airlines	SELECT
demo.summit+demo_user4@databricks.com	eduardospartner	mydemo	SELECT
demo.summit+demo_user2@databricks.com	external_organization	delta_sharing	SELECT
demo.summit+demo_user4@databricks.com	southwest_airlines	test	SELECT
demo.summit+demo_user4@databricks.com	ateradodatabricks	americanairlines	SELECT
demo.summit+demo_user4@databricks.com	alexterado	americanairlines	SELECT
[redacted]@databricks.com	frank_ext_dais	frank_share	SELECT
demo.summit+demo_user2@databricks.com	weddings_trinkets_co	online_sales	SELECT
demo.summit+demo_user2@databricks.com	holiday_goods_co	online_sales	SELECT
demo.summit+demo_user2@databricks.com	eduardospartner	eduardosshare	SELECT
[redacted]@databricks.com	loan_recipient	loan_tx	SELECT
[redacted]@databricks.com	southwest_airlines	southwest_airlines	SELECT

The example dashboard below shows the number of users, tables, ML models, percent of tables that are monitored or not, dollars spent on Databricks DBUs over time, and so much more:



Governance dashboard showing billing trends, usage, activity and more

What does having a comprehensive data and AI monitoring and reporting tool result in?

- Reduced risk of non-compliance with better monitoring of internal policies and security breach potential results in safeguarded reputation and improved data and AI trust from employees and partners.
- Improved integrity and trustworthiness of data and AI with "one source of truth", anomaly detection, and reliability metrics.

Value Levers with Databricks Unity Catalog

If you are looking to learn more about the values Unity Catalog brings to businesses, the prior Unity Catalog Governance Value Levers blog went into detail: mitigating risk around compliance; reducing platform complexity and costs; accelerating innovation; facilitating better internal and external collaboration; and monetizing the value of data.

CONCLUSION

Governance is key to mitigating risks, ensuring compliance, accelerating innovation, and reducing costs. Databricks Unity Catalog is unique in the market, providing a single unified governance solution for all of a company's data and AI across clouds and data platforms.

UC Databricks architecture makes governance seamless: a unified view and discovery of all data assets, one tool for access management, one tool for auditing for enhanced data and AI security, and ultimately enabling platform-independent collaboration that unlocks new business values.

Getting started is easy - UC comes enabled by default with Databricks if you are a new customer! Also if you are on premium or enterprise workspaces, there are no additional costs.

Scalable Spark Structured Streaming for REST API Destinations

by Art Rask and Jay Palaniappan

Spark Structured Streaming is the widely-used open source engine at the foundation of data streaming on the Data Intelligence Platform. It can elegantly handle diverse logical processing at volumes ranging from small-scale ETL to the largest Internet services. This power has led to adoption in many use cases across industries.

Another strength of Structured Streaming is its ability to handle a variety of both sources and sinks (or destinations). In addition to numerous sink types supported natively (incl. Delta, AWS S3, Google GCS, Azure ADLS, Kafka topics, Kinesis streams, and more), Structured Streaming supports a specialized sink that has the ability to perform arbitrary logic on the output of a streaming query: the foreachBatch extension method. With foreachBatch, any output target addressable through Python or Scala code can be the destination for a stream.

In this chapter we will share best practice guidance we've given customers who have asked how they can scalably turn streaming data into calls against a REST API. Routing an incoming stream of data to calls on a REST API is a requirement seen in many integration and data engineering scenarios.

Some practical examples that we often come across are in Operational and Security Analytics workloads. Customers want to ingest and enrich real-time streaming data from sources like kafka, eventhub, and Kinesis and publish it into operational search engines like Elasticsearch, Opensearch, and Splunk. A key advantage of Spark Streaming is that it allows us to enrich, perform data quality checks, and aggregate (if needed) before data is streamed out into the search engines. This provides customers a high quality real-time data pipeline for operational and security analytics.

The most basic representation of this scenario is shown in Figure 1. Here we have an incoming stream of data - it could be a Kafka topic, AWS Kinesis, Azure Event Hub, or any other streaming query source. As messages flow off the stream we need to make calls to a REST API with some or all of the message data.

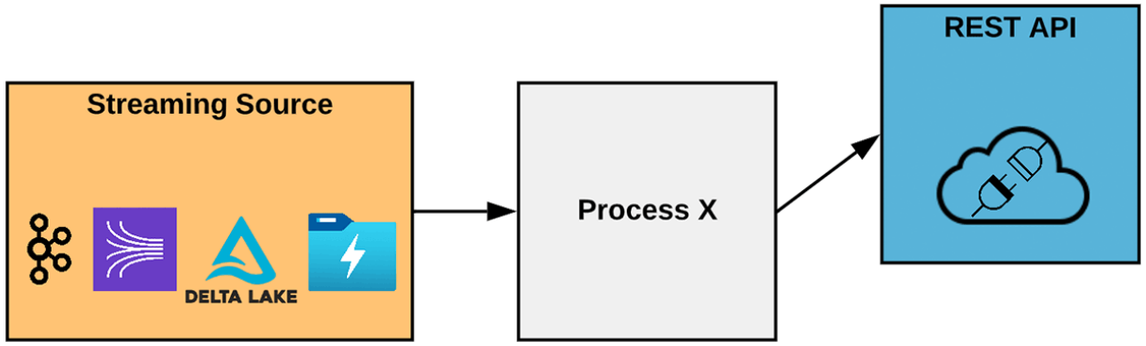


Figure 1

In a greenfield environment, there are many technical options to implement this. Our focus here is on teams that already have streaming pipelines in Spark for preparing data for machine learning, data warehousing, or other analytics-focused uses. In this case, the team will already have skills, tooling and DevOps processes for Spark. Assume the team now has a requirement to route some data to REST API calls. If they wish to leverage existing skills or avoid re-working their tool chains, they can use Structured Streaming to get it done.

KEY IMPLEMENTATION TECHNIQUES, AND SOME CODE

A basic code sample is included as Exhibit 1. Before looking at it in detail, we will call out some key techniques for effective implementation.

For a start, you will read the incoming stream as you would any other streaming job. All the interesting parts here are on the output side of the stream. If your data must be transformed in flight before posting to the REST API, do that as you would in any other case. This code snippet reads from a Delta table; as mentioned, there are many other possible sources.

```
dfSource = (spark.readStream
                .format("delta")
                .table("samples.nyctaxi.trips"))
```

For directing streamed data to the REST API, take the following approach:

1. Use the foreachBatch extension method to pass incoming micro-batches to a handler method (callRestAPIBatch) which will handle calls to the REST API.

```
streamHandle = (dfSource.writeStream
                        .foreachBatch(callRestAPIBatch)
                        .start())
```

2. Whenever possible, group multiple rows from the input on each outgoing REST API call. In relative terms, making the API call over HTTP will be a slow part of the process. Your ability to reach high throughput will be dramatically improved if you include multiple messages/records on the body of each API call. Of course, what you can do will be dictated by the target REST API. Some APIs allow a POST body to include many items up to a maximum body size in bytes. Some APIs have a max count of items on the POST body. Determine the max you can fit on a single call for the target API. In your method invoked by foreachBatch, you will have a prep step to transform the micro-batch dataframe into a pre-batched dataframe where each row has the grouped records for one call to the API. This step is also a chance for any last transform of the records to the format expected by the target API. An example is shown in the code sample in Exhibit 1 with the call to a helper function named preBatchRecordsForRestCall.
3. In most cases, to achieve a desired level of throughput, you will want to make calls to the API from parallel tasks. You can control the degree of parallelism by calling repartition on the dataframe of pre-batched data. Call repartition with the number of parallel tasks you want calling the API. This is actually just one line of code.

```
### Repartition pre-batched df for parallelism of API calls
new_df = pre_batched_df.repartition(8)
```

It is worth mentioning (or admitting) that using repartition here is a bit of an anti-pattern. Explicit repartitioning with large datasets can have performance implications, especially if it causes a shuffle between nodes on the cluster. In most cases of calling a REST API, the data size of any micro-batch is not massive. So, in practical terms, this technique is unlikely to cause a problem. And, it has a big positive effect on throughput to the API.

4. Execute a dataframe transformation that calls a nested function dedicated to making a single call to the REST API. The input to this function will be one row of pre-batched data. In the sample, the payload column has the data to include on a single call. Call a dataframe action method to invoke execution of the transformation.

```
submitted_df = new_df.withColumn("RestAPIResponseCode",\
                           callRestApiOnce(new_df["payload"])).\
                           collect()
```

5. Inside the nested function which will make one API call, use your libraries of choice to issue an HTTP POST against the REST API. This is commonly done with the Requests library but any library suitable for making the call can be considered. See the callRestApiOnce method in Exhibit 1 for an example.
6. Handle potential errors from the REST API call by using a try..except block or checking the HTTP response code. If the call is unsuccessful, the overall job can be failed by throwing an exception (for job retry or troubleshooting) or individual records can be diverted to a dead letter queue for remediation or later retry.

```
if not (response.status_code==200 or response.status_code==201) :
 raise Exception("Response status : {} .Response message : {}".\
                 format(str(response.status_code),response.text))
```

The six elements above should prepare your code for sending streaming data to a REST API, with the ability to scale for throughput and to handle error conditions cleanly. The sample code in Exhibit 1 is an example implementation. Each point stated above is reflected in the full example.

```
from pyspark.sql.functions import *
from pyspark.sql.window import Window
import math
import requests
from requests.adapters import HTTPAdapter

def preBatchRecordsForRestCall(microBatchDf, batchSize):
    batch_count = math.ceil(microBatchDf.count() / batchSize)
    microBatchDf = microBatchDf.withColumn("content", to_json(struct(col("*"))))
    microBatchDf = microBatchDf.withColumn("row_number",\
                                           row_number().over(Window().
orderBy(lit('A'))))
    microBatchDf = microBatchDf.withColumn("batch_id", col("row_number") % batch_
count)
    return microBatchDf.groupBy("batch_id").\
                                        agg(concat_ws(",|", collect_
list("content")).\
                                        alias("payload"))


def callRestAPIBatch(df, batchId):
  restapi_uri = "<REST API URL>"

  @udf("string")
  def callRestApiOnce(x):
    session = requests.Session()
    adapter = HTTPAdapter(max_retries=3)
    session.mount('http://', adapter)
    session.mount('https://', adapter)

    #this code sample calls an unauthenticated REST endpoint; add headers
necessary for auth
    headers = {'Authorization':'abcd'}
    response = session.post(restapi_uri, headers=headers, data=x, verify=False)
    if not (response.status_code==200 or response.status_code==201) :
      raise Exception("Response status : {} .Response message : {}".\
                      format(str(response.status_code),response.text))

    return str(response.status_code)
```

```
  ### Call helper method to transform df to pre-batched df with one row per REST
API call
  ### The POST body size and formatting is dictated by the target API; this is an
example
  pre_batched_df = preBatchRecordsForRestCall(df, 10)

  ### Repartition pre-batched df for target parallelism of API calls
  new_df = pre_batched_df.repartition(8)

  ### Invoke helper method to call REST API once per row in the pre-batched df
  submitted_df = new_df.withColumn("RestAPIResponseCode",\
                                   callRestApiOnce(new_df["payload"])).collect()


dfSource = (spark.readStream
                .format("delta")
                .table("samples.nyctaxi.trips"))

streamHandle = (dfSource.writeStream
                        .foreachBatch(callRestAPIBatch)
                        .trigger(availableNow=True)
                        .start())
```

Exhibit 1

DESIGN AND OPERATIONAL CONSIDERATIONS

Exactly Once vs At Least Once Guarantees

As a general rule in Structured Streaming, using foreachBatch only provides at-least-once delivery guarantees. This is in contrast to the exactly-once delivery guarantee provided when writing to sinks like a Delta table or file sinks. Consider, for example, a case where 1,000 records arrive on a micro-batch and your code in foreachBatch begins calling the REST API with the batch. In a hypothetical failure scenario, let's say that 900 calls succeed before an error occurs and fails the job. When the stream restarts, processing will resume by re-processing the failed batch. Without additional logic in your code, the 900 already-processed calls will be repeated. It is important that you determine in your design whether this is acceptable, or whether you need to take additional steps to protect against duplicate processing.

The general rule when using foreachBatch is that your target sink (REST API in this case) should be idempotent or that you must do additional tracking to account for multiple calls with the same data.

Estimating Cluster Core Count for a Target Throughput

Given these techniques to call a REST API with streaming data, it will quickly become necessary to estimate how many parallel executors/tasks are necessary to achieve your required throughput. And you will need to select a cluster size. The following table shows an example calculation for estimating the number of worker cores to provision in the cluster that will run the stream.

A	Target Throughput (records/sec)	3200	←Required rate of records processed by stream and submitted to REST API
B	# records batched per REST API call	10	←Average number of records included on POST body for each API call
C	Total required calls per second	320	A / B, rounded up
D	API Latency (ms to make one call)	50	← Plug in estimated or observed latency
E	Max calls per task per second	20	← 1000ms / D, rounded up
F	Min required parallelism	16	C / E, rounded up
G	Shared use factor	2	Rule-of-thumb multiplier to account for other work being done on cluster
H	Estimated core count	32	F * G, rounded up.

Line H in the table shows the estimated number of worker cores necessary to sustain the target throughput. In the example shown here, you could provision a cluster with two 16-core workers or 4 8-core workers, for example. For this type of workload, fewer nodes with more cores per node is preferred.

Line H is also the number that would be put in the repartition call in foreachBatch, as described in item 3 above.

Line G is a rule of thumb to account for other activity on the cluster. Even if your stream is the only job on the cluster, it will not be calling the API 100% of the time. Some time will be spent reading data from the source stream, for example. The value shown here is a good starting point for this factor - you may be able to fine tune it based on observations of your workload.

Obviously, this calculation only provides an estimated starting point for tuning the size of your cluster. We recommend you start from here and adjust up or down to balance cost and throughput.