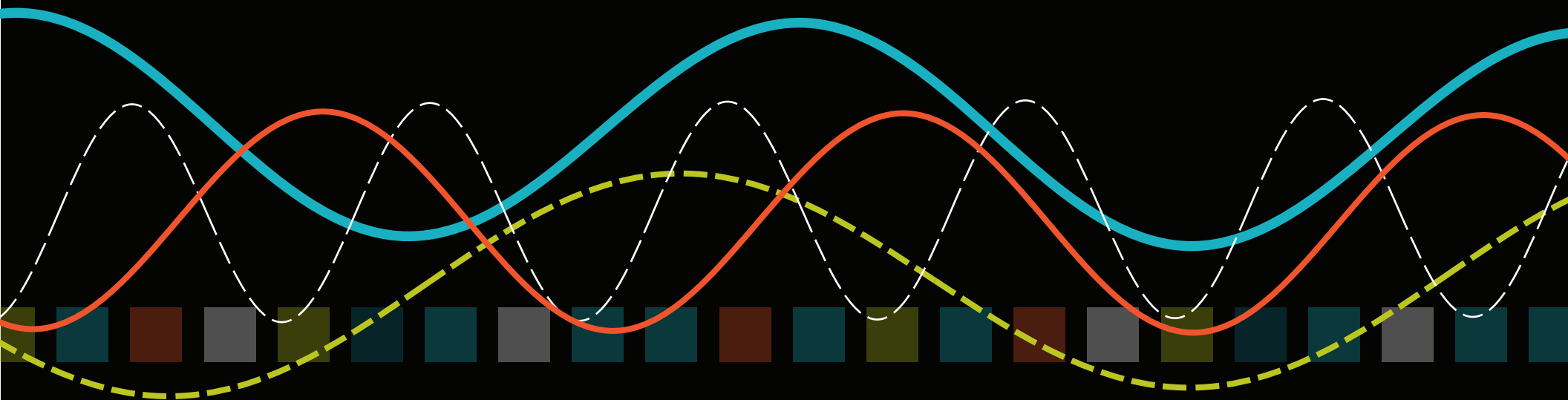


Building Streaming Applications with Apache Spark™

How to Use Structured Streaming to Build Complex Continuous Applications



Building Streaming Applications with Apache Spark™

How to Use Structured Streaming to Build Complex Continuous Applications

Special thanks to the contributions of Michael Armbrust, Bill Chambers, Tyson Condie, Jules Damji, Tathagata Das, Kunal Khumar, Reynold Xin, Burak Yavuz, and Matei Zaharia to this ebook.

About Databricks

Databricks' mission is to accelerate innovation for its customers by unifying Data Science, Engineering and Business. Founded by the team who started the Apache Spark™ project, Databricks provides a Unified Analytics Platform for data science teams to collaborate with data engineering and lines of business to build data products. Users achieve faster time-to-value with Databricks by creating analytic workflows that go from ETL and interactive exploration to production. The company also makes it easier for its users to focus on their data by providing a fully managed, scalable, and secure cloud infrastructure that reduces operational complexity and total cost of ownership. Databricks, venture-backed by Andreessen Horowitz and NEA, has a global customer base that includes Salesforce, Viacom, Amgen, Shell and HP. For more information, visit www.databricks.com.

Databricks

160 Spear Street, 13th Floor

San Francisco, CA 94105

[Contact Us](#)

© Databricks 2018. All rights reserved. Apache, Apache Spark, Spark and the Spark logo are trademarks of the [Apache Software Foundation](#).

Table of Contents

Introduction	4
Part 1: An Introduction to Structured Streaming	5
Structured Streaming In Apache Spark	6
Part 2: Real-time ETL Pipelines	15
Real-time Streaming ETL with Structured Streaming in Apache Spark 2.1	16
Part 3: Fixing Data Transformation Challenges	24
Working with Complex Data Formats with Structured Streaming in Apache Spark 2.1	25
Part 4: Processing Data in Apache Kafka	36
Processing Data in Apache Kafka with Structured Streaming in Apache Spark 2.2	37
Part 5: Event-time Aggregations and Watermarking	47
Event-time Aggregation and Watermarking in Apache Spark's Structured Streaming	48
Part 6: Vital Steps to Ensure Production Readiness	53
Taking Apache Spark's Structured Streaming to Production	54
Part 7: Better Cost Management through APIs	59
Running Streaming Jobs Once a Day For 10x Cost Savings	60
Part 8: Customized and Arbitrary Stateful Processing	64
Arbitrary Stateful Processing in Apache Spark's Structured Streaming	65
Conclusion	72

Introduction

Since its release, Apache Spark Streaming has become one of the most widely used distributed streaming engines, thanks to its high-level API and exactly-once semantics. Most streaming engines focus on performing computations on a stream. Instead, stream processing happens as part of a larger application, which we'll call a continuous application.

We define a continuous application as an end-to-end application that reacts to data in real-time. Structured Streaming is a high-level API originally contributed to Apache Spark 2.0 to support continuous applications and was recently improved upon in the release of Apache Spark 2.3. Structured Streaming incorporates the idea of continuous applications to provide a number of features that no other streaming engines offer strong guarantees about consistency with batch jobs, transactional integration with storage systems, and tight integration with the rest of Spark.

At Databricks, we've worked with thousands of users to understand how to simplify real-time applications. This ebook provides an overview of Structured Streaming and explores how we are using the new features of Apache Spark 2.1 and 2.2 to overcome the primary challenges of building continuous applications and building our own production pipelines. Highlights include how to use Structured Streaming to:

- Easily build an end-to-end streaming ETL pipeline;
- Solve complex data transformation challenges;
- Perform monitoring and alerting;
- Consume and transform complex data streams with Spark and Kafka;
- Easily process streaming aggregations; and
- Better manage resources for incremental processing of data.



Part 1:
An Introduction to
Structured Streaming

Structured Streaming In Apache Spark

A new high-level API for streaming

July 28, 2016 | by Matei Zaharia, Tathagata Das, Michael Armbrust and Reynold Xin

 Try this notebook in Databricks: [Scala Notebook](#), [Python Notebook](#)

Apache Spark 2.0 adds the first version of a new higher-level API, Structured Streaming, for building [continuous applications](#). The main goal is to make it easier to build *end-to-end* streaming applications, which integrate with storage, serving systems, and batch jobs in a consistent and fault-tolerant way. In this post, we explain why this is hard to do with current distributed streaming engines, and introduce Structured Streaming.

Why Streaming is Difficult

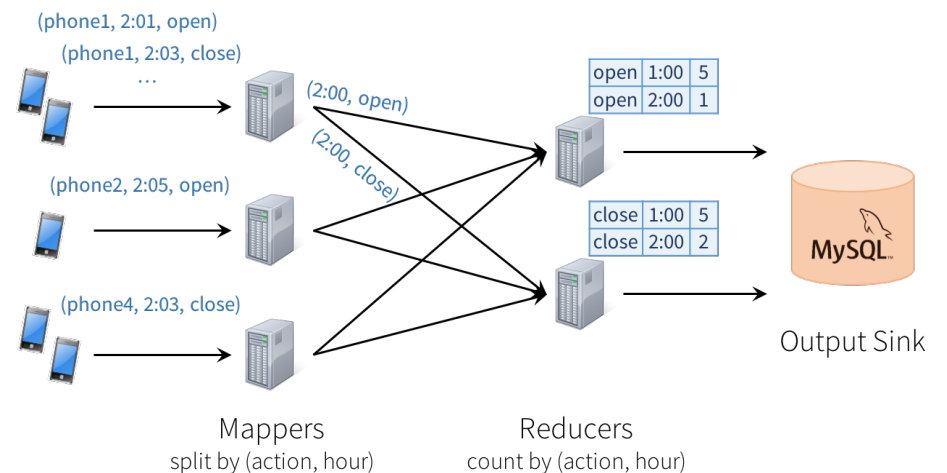
At first glance, building a distributed streaming engine might seem as simple as launching a set of servers and pushing data between them. Unfortunately, distributed stream processing runs into multiple complications that don't affect simpler computations like batch jobs.

To start, consider a simple application: we receive (phone_id, time, action) events from a mobile app, and want to count how many

actions of each type happened each hour, then store the result in MySQL. If we were running this application as a batch job and had a table with all the input events, we could express it as the following SQL query:

```
SELECT action, WINDOW(time, "1 hour"), COUNT(*)
FROM events
GROUP BY action, WINDOW(time, "1 hour")
```

In a distributed streaming engine, we might set up nodes to process the data in a “map-reduce” pattern, as shown below. Each node in the first layer reads a partition of the input data (say, the stream from one set of phones), then hashes the events by (action, hour) to send them to a reducer node, which tracks that group's count and periodically updates MySQL.



Unfortunately, this type of design can introduce quite a few challenges:

1. **Consistency:** This distributed design can cause records to be processed in one part of the system before they're processed in another, leading to nonsensical results. For example, suppose our app sends an “open” event when users open it, and a “close” event when closed. If the reducer node responsible for “open” is slower than the one for “close”, we might see a *higher total count of “closes” than “opens” in MySQL*, which would not make sense. The image above actually shows one such example.
2. **Fault tolerance:** What happens if one of the mappers or reducers fails? A reducer should not count an action in MySQL twice, but should somehow know how to request old data from the mappers when it comes up. Streaming engines go through a great deal of trouble to provide strong semantics here, at least *within* the engine. In many engines, however, keeping the result consistent in external storage is left to the user.
3. **Out-of-order data:** In the real world, data from different sources can come out of order: for example, a phone might upload its data hours late if it's out of coverage. Just writing the reducer operators to assume data arrives in order of time fields will not work—they need to be prepared to receive out-of-order data, and to update the results in MySQL accordingly.

In most current streaming systems, some or all of these concerns are left to the user. This is unfortunate because these issues—how the application interacts with the outside world—are some of the hardest to reason about and get right. In particular, there is no easy way to get semantics as simple as the SQL query above.

Structured Streaming Model

In Structured Streaming, we tackle the issue of semantics head-on by making a strong guarantee about the system: at any time, *the output of the application is equivalent to executing a batch job on a prefix of the data*. For example, in our monitoring application, the result table in MySQL will always be equivalent to taking a prefix of each phone's update stream (whatever data made it to the system so far) and running the SQL query we showed above. There will never be “open” events counted faster than “close” events, duplicate updates on failure, etc. Structured Streaming automatically handles consistency and reliability both within the engine and in interactions with external systems (e.g. updating MySQL transactionally).

This *prefix integrity* guarantee makes it easy to reason about the three challenges we identified. In particular:

1. Output tables are **always consistent** with all the records in a prefix of the data. For example, as long as each phone uploads its data as a sequential stream (e.g., to the same partition in Apache Kafka), we will always process and count its events in order.

2. **Fault tolerance** is handled holistically by Structured Streaming, including in interactions with output sinks. This was a major goal in supporting [continuous applications](#).
3. The effect of **out-of-order data** is clear. We know that the job outputs counts grouped by action and time for a prefix of the stream. If we later receive more data, we might see a time field for an hour in the past, and we will simply update its respective row in MySQL. Structured Streaming also supports APIs for filtering out overly old data if the user wants. But fundamentally, out-of-order data is not a “special case”: the query says to group by time field, and seeing an old time is no different than seeing a repeated action.

The last benefit of Structured Streaming is that the API is very easy to use: it is simply Spark’s [DataFrame and Dataset API](#). Users just describe the query they want to run, the input and output locations, and optionally a few more details. The system then runs their query incrementally, maintaining enough state to recover from failure, keep the results consistent in external storage, etc. For example, here is how to write our streaming monitoring application:

```
// Read data continuously from an S3 location
val inputDF = spark.readStream.json("s3://logs")

// Do operations using the standard DataFrame API and write to MySQL
inputDF.groupBy($"action", window($"time", "1 hour")).count()
        .writeStream.format("jdbc")
        .start("jdbc:mysql://...")
```

This code is nearly identical to the batch version below—only the “read” and “write” changed:

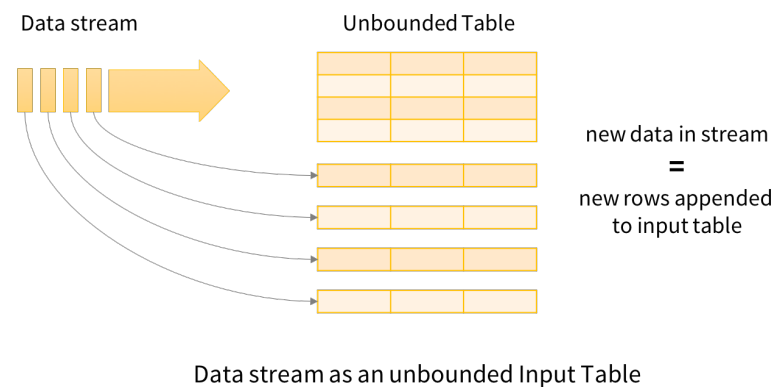
```
// Read data once from an S3 location
val inputDF = spark.read.json("s3://logs")

// Do operations using the standard DataFrame API and write to MySQL
inputDF.groupBy($"action", window($"time", "1 hour")).count()
        .writeStream.format("jdbc")
        .save("jdbc:mysql://...")
```

The next sections explain the model in more detail, as well as the API.

Model Details

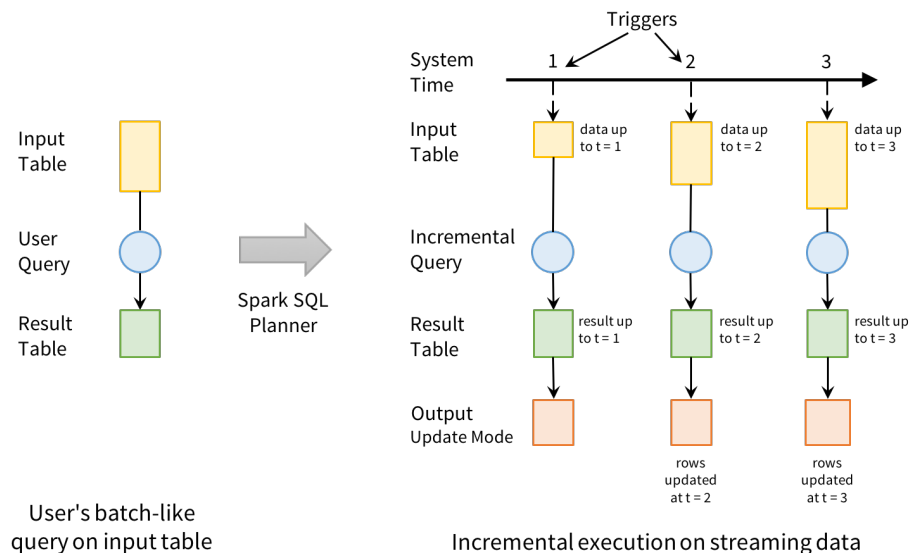
Conceptually, Structured Streaming treats all the data arriving as an unbounded **input table**. Each new item in the stream is like a row appended to the input table. We won’t actually retain all the input, but our results will be equivalent to having all of it and running a batch job.



The developer then defines a **query** on this input table, as if it were a static table, to compute a final **result table** that will be written to an output sink. Spark automatically converts this batch-like query to a streaming execution plan. This is called *incrementalization*: Spark figures out what state needs to be maintained to update the result each time a record arrives. Finally, developers specify **triggers** to control when to update the results. Each time a trigger fires, Spark checks for new data (new row in the input table), and incrementally updates the result.

The last part of the model is **output modes**. Each time the result table is updated, the developer wants to write the changes to an external system, such as S3, HDFS, or a database. We usually want to write output incrementally. For this purpose, Structured Streaming provides three output modes:

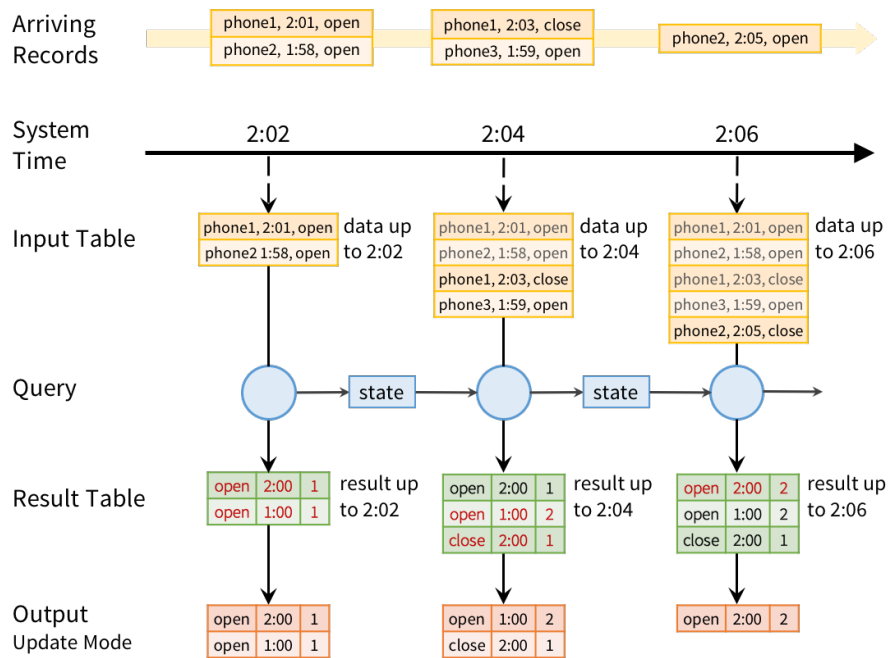
- **Append:** Only the new rows appended to the result table since the last trigger will be written to the external storage. This is applicable only on queries where existing rows in the result table cannot change (e.g. a map on an input stream).
- **Complete:** The entire updated result table will be written to external storage.
- **Update:** Only the rows that were updated in the result table since the last trigger will be changed in the external storage. This mode works for output sinks that can be updated in place, such as a MySQL table.



Structured Streaming Processing Model

Users express queries using a batch API; Spark incrementalizes them to run on streams

Let's see how we can run our mobile monitoring application in this model. Our batch query is to compute a count of actions grouped by (action, hour). To run this query incrementally, Spark will maintain some state with the counts for each pair so far, and update when new records arrive. For each record changed, it will then output data according to its output mode. The figure below shows this execution using the Update output mode:



At every trigger point, we take the previous grouped counts and update them with new data that arrived since the last trigger to get a new result table. We then emit only the changes required by our output mode to the sink—here, we update the records for (action, hour) pairs that changed during that trigger in MySQL (shown in red).

Note that the system also automatically handles late data. In the figure above, the “open” event for phone3, which happened at 1:58 on the phone, only gets to the system at 2:02. Nonetheless, even though it’s past 2:00, we update the record for 1:00 in MySQL. However, the prefix integrity guarantee in Structured Streaming ensures that we process

the records from each source *in the order they arrive*. For example, because phone1’s “close” event arrives after its “open” event, we will always update the “open” count before we update the “close” count.

Fault Recovery and Storage System Requirements

Structured Streaming keeps its results valid even if machines fail. To do this, it places two requirements on the input sources and output sinks:

1. Input sources must be *replayable*, so that recent data can be re-read if the job crashes. For example, message buses like Amazon Kinesis and Apache Kafka are replayable, as is the file system input source. Only a few minutes’ worth of data needs to be retained; Structured Streaming will maintain its own internal state after that.
2. Output sinks must support *transactional updates*, so that the system can make a set of records appear atomically. The current version of Structured Streaming implements this for file sinks, and we also plan to add it for common databases and key-value stores.
3. We found that most Spark applications already use sinks and sources with these properties, because users want their jobs to be reliable.

Apart from these requirements, Structured Streaming will manage its internal state in a reliable storage system, such as S3 or HDFS, to store data such as the running counts in our example. Given these properties, Structured Streaming will enforce prefix integrity end-to-end.

Structured Streaming API

Structured Streaming is integrated into Spark's [Dataset and DataFrame APIs](#); in most cases, you only need to add a few method calls to run a streaming computation. It also adds new operators for windowed aggregation and for setting parameters of the execution model (e.g. output modes). In [Apache Spark 2.0](#), we've built an alpha version of the system with the core APIs. More operators, such as sessionization, will come in future releases.

API Basics

Streams in Structured Streaming are represented as DataFrames or Datasets with the `isStreaming` property set to true. You can create them using special read methods from various sources. For example, suppose we wanted to read data in our monitoring application from JSON files uploaded to Amazon S3. The code below shows how to do this in Scala:

```
val inputDF = spark.readStream.json("s3://logs")
```

Our resulting DataFrame, `inputDF`, is our input table, which will be continuously extended with new rows as new files are added to the directory. The table has two columns—time and action. Now you can use the usual DataFrame/Dataset operations to transform the data. In our example, we want to count action types each hour. To do that we have to group the data by action and 1 hours windows of time.

```
val countsDF = inputDF.groupBy($"action", window($"time", "1 hour"))  
    .count()
```

The new DataFrame `countsDF` is our result table, which has the columns `action`, `window`, and `count`, and will be continuously updated when the query is started. Note that this transformation would give hourly counts even if `inputDF` was a static table. This allows developers to test their business logic on static datasets and seamlessly apply them on streaming data without changing the logic.

Finally, we tell the engine to write this table to a sink and start the streaming computation.

```
val query = countsDF.writeStream.format("jdbc").start("jdbc://...")
```

The returned `query` is a `StreamingQuery`, a handle to the active streaming execution and can be used to manage and monitor the execution. You can run this complete example by importing the following notebooks into [Databricks Community edition](#): [Scala Notebook](#), [Python Notebook](#).

Beyond these basics, there are many more operations that can be done in Structured Streaming.

Mapping, Filtering and Running Aggregations

Structured Streaming programs can use DataFrame and Dataset's existing methods to transform data, including map, filter, select, and others. In addition, running (or infinite) aggregations, such as a `count` from the beginning of time, are available through the existing APIs. This is what we used in our monitoring application above.

Windowed Aggregations on Event Time

Streaming applications often need to compute data on various types of *windows*, including *sliding windows*, which overlap with each other (e.g. a 1-hour window that advances every 5 minutes), and *tumbling windows*, which do not (e.g. just every hour). In Structured Streaming, *windowing is simply represented as a group-by*. Each input event can be mapped to one or more windows, and simply results in updating one or more result table rows.

Windows can be specified using the window function in DataFrames. For example, we could change our monitoring job to count actions by sliding windows as follows:

```
inputDF.groupBy($"action", window($"time", "1 hour", "5
minutes"))
    .count()
```

Whereas our previous application outputted results of the form (hour, action, count), this new one will output results of the form (window, action, count), such as ("1:10-2:10", "open", 17). If a late record arrives, we will update all the corresponding windows in MySQL. And unlike in many other systems, windowing is not just a special operator for streaming computations; we can run the same code in a batch job to group data in the same way.

Windowed aggregation is one area where we will continue to expand Structured Streaming. In particular, in Spark 2.1, we plan to add watermarks, a feature for dropping overly old data when sufficient time has passed. Without this type of feature, the system might have to track state for all old windows, which would not scale as the application runs. In addition, we plan to add support for session-based windows, i.e. grouping the events from one source into variable-length sessions according to business logic.

Joining Streams with Static Data

Because Structured Streaming simply uses the DataFrame API, it is straightforward to join a stream against a static DataFrame, such as an Apache Hive table:

```
// Bring in data about each customer from a static "customers"
table,
// then join it with a streaming DataFrame
val customersDF = spark.table("customers")
inputDF.join(customersDF, "customer_id")
    .groupBy($"customer_name", hour($"time"))
    .count()
```

Moreover, the static DataFrame could itself be computed using a Spark query, allowing us to mix batch and streaming computations.

Interactive Queries

Structured Streaming can expose results directly to interactive queries through Spark's JDBC server. In Spark 2.0, there is a rudimentary "memory" output sink for this purpose that is not designed for large data volumes. However, in future releases, this will let you write query results to an in-memory Spark SQL table, and run queries directly against it.

```
// Save our previous counts query to an in-memory table
countsDF.writeStream.format("memory")
    .queryName("counts")
    .outputMode("complete")
    .start()

// Then any thread can query the table using SQL
sql("select sum(count) from counts where action='login'")
```

Comparison With Other Engines

To show what's unique about Structured Streaming, the next table compares it with several other systems. As we discussed, Structured Streaming's strong guarantee of prefix integrity makes it equivalent to batch jobs and easy to integrate into larger applications. Moreover, building on Spark enables integration with batch and interactive queries.

Property	Structured Streaming	Spark Streaming	Apache Storm	Apache Flink	Kafka Streams	Google Dataflow
Streaming API	incrementalize batch queries	integrates with batch	separate from batch	separate from batch	separate from batch	integrates with batch
Prefix Integrity Guarantee	✓	✓	✗	✗	✗	✗
Internal Processing	exactly once	exactly once	at least once	exactly once	at least once	exactly once
Transactional Sources/Sinks	✓	some	some	some	✗	✗
Interactive Queries	✓	✓	✗	✗	✗	✗
Joins with Static Data	✓	✓	✗	✗	✗	✗

Conclusion

Structured Streaming promises to be a much simpler model for building end-to-end real-time applications, built on the features that work best in Spark Streaming. Although Structured Streaming is in alpha for Apache Spark 2.0, we hope this post encourages you to try it out.

Long-term, much like the DataFrame API, we expect Structured Streaming to complement Spark Streaming by providing a more restricted but higher-level interface. If you are running Spark Streaming today, don't worry—it will continue to be supported. But we believe that Structured Streaming can open up real-time computation to many more users.

Structured Streaming is also fully supported on Databricks, including in the free [Databricks Community Edition](#). Try out any of our sample notebooks to see it in action:

- [Scala notebook for monitoring app](#)
- [Python notebook for monitoring app](#)

Read More

In addition, the following resources cover Structured Streaming:

- [Structuring Spark: DataFrames, Datasets and Streaming](#)
- [Structured Streaming Programming Guide](#)
- [Apache Spark 2.0 and Structured Streaming](#)
- [A Deep Dive Into Structured Streaming](#)



Part 2:

Real-time ETL Pipelines

Real-time Streaming ETL with Structured Streaming in Apache Spark 2.1

Part 1 of Scalable Data @ Databricks

January 19, 2017 | by Tathagata Das, Michael Armbrust and Tyson Condie

 Try this notebook in Databricks: [Scala Notebook](#), [Python Notebook](#)

We are well into the Big Data era, with organizations collecting massive amounts of data on a continual basis. Yet, the value of this data deluge hinges on the ability to extract actionable insights in a timely fashion. Hence, there is an increasing need for *continuous applications* that can derive real-time actionable insights from massive data ingestion pipelines.

However, building production-grade continuous applications can be challenging, as developers need to overcome many obstacles, including:

- **Providing end-to-end reliability and correctness guarantees** – Long running data processing systems must be resilient to failures by ensuring that outputs are consistent with results processed in

batch. Additionally, unusual activities (e.g failures in upstream components, traffic spikes, etc.) must be continuously monitored and automatically mitigated to ensure highly available insights are delivered in real-time.

- **Performing complex transformations** – Data arrives in a myriad formats (CSV, JSON, Avro, etc.) that often must be restructured, transformed and augmented before being consumed. Such restructuring requires that all the traditional tools from batch processing systems are available, but without the added latencies that they typically entail.
- **Handling late or out-of-order data** – When dealing with the physical world, data arriving late or out-of-order is a fact of life. As a result, aggregations and other complex computations must be continuously (and accurately) revised as new information arrives.
- **Integrating with other systems** – Information originates from a variety of sources (Kafka, HDFS, S3, etc), which must be integrated to see the complete picture.

Structured Streaming in Apache Spark builds upon the strong foundation of Spark SQL, leveraging its powerful APIs to provide a seamless query interface, while simultaneously optimizing its execution engine to enable low-latency, continually updated answers. This blog post kicks off a series in which we will explore how

we are using the new features of Apache Spark 2.1 to overcome the above challenges and build our own production pipelines.

In this first post, we will focus on an ETL pipeline that converts raw [AWS CloudTrail audit logs](#) into a [JIT data warehouse](#) for faster ad-hoc queries. **We will show how easy it is to take an existing batch ETL job and subsequently productize it as a real-time streaming pipeline using Structured Streaming in Databricks.** Using this pipeline, we have converted **3.8 million** JSON files containing **7.9 billion** records into a Parquet table, which allows us to do ad-hoc queries on updated-to-the-minute Parquet table 10x faster than those on raw JSON files.

The Need for Streaming ETL

Extract, Transform, and Load (ETL) pipelines prepare raw, unstructured data into a form that can be queried easily and efficiently. Specifically, they need to be able to do the following:

- **Filter, transform, and clean up data** – Raw data is naturally messy and needs to be cleaned up to fit into a well-defined structured format. For example, parsing timestamp strings to date/time types for faster comparisons, filtering corrupted data, nesting/unnesting/flattening complex structures to better organize important columns, etc.
- **Convert to a more efficient storage format** – Text, JSON and CSV data are easy to generate and are human readable, but are very expensive to query. Converting it to more efficient formats like

Parquet, Avro, or ORC can reduce file size and improve processing speed.

- **Partition data by important columns** – By partitioning the data based on the value of one or more columns, common queries can be answered more efficiently by reading only the relevant fraction of the total dataset.

Traditionally, ETL is performed as periodic batch jobs. For example, dump the raw data in real time, and then convert it to structured form every few hours to enable efficient queries. We had initially setup our system this way, but this technique incurred a high latency; we had to wait for few hours before getting any insights. For many use cases, this delay is unacceptable. When something suspicious is happening in an account, we need to be able to ask questions immediately. Waiting minutes to hours could result in an unreasonable delay in responding to an incident.

Fortunately, Structured Streaming makes it easy to convert these periodic batch jobs to a real-time data pipeline. Streaming jobs are expressed using the same APIs as batch data. Additionally, the engine provides the same fault-tolerance and data consistency guarantees as periodic batch jobs, while providing much lower end-to-end latency.

In the rest of post, we dive into the details of how we transform [AWS CloudTrail audit logs](#) into an efficient, partitioned, parquet data warehouse. AWS CloudTrail allows us to track all actions performed in

a variety of AWS accounts, by delivering gzipped JSON logs files to a S3 bucket. These files enable a variety of business and mission critical intelligence, such as cost attribution and security monitoring. However, in their original form, they are very costly to query, even with the capabilities of Apache Spark. To enable rapid insight, we run a Continuous Application that transforms the raw JSON logs files into an optimized Parquet table. Let's dive in and look at how to write this pipeline. If you want to see the full code, here are the [Scala](#) and [Python](#) notebooks. [Import them into Databricks](#) and run them yourselves.

Transforming Raw Logs with Structured Streaming

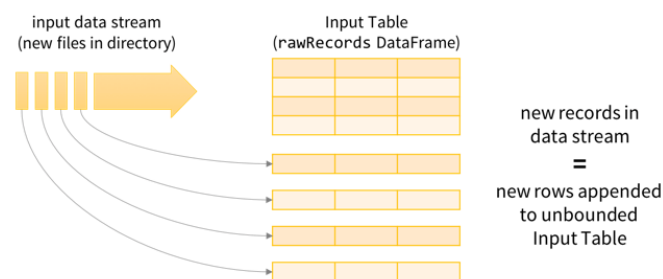
We start by defining the schema of the JSON records based on [CloudTrail documentation](#).

```
val cloudTrailSchema = new StructType()
  .add("Records", ArrayType(new StructType()
    .add("additionalEventData", StringType)
    .add("apiVersion", StringType)
    .add("awsRegion", StringType)
  // ...
```

See the attached notebook for the full schema. With this, we can define a streaming DataFrame that represents the data stream from CloudTrail files that are being written in a S3 bucket.

```
val rawRecords = spark.readStream
  .schema(cloudTrailSchema)
  .json("s3n://mybucket/AWSLogs/*/CloudTrail/*/2017/*/*")
```

A good way to understand what this `rawRecords` DataFrame represents is to first understand the [Structured Streaming programming model](#). The key idea is to treat any data stream as an unbounded table: new records added to the stream are like rows being appended to the table.



Structured Streaming Model
treat data streams as unbounded tables

This allows us to treat both batch and streaming data as tables. Since tables and DataFrames/Datasets are semantically synonymous, the same batch-like DataFrame/Dataset queries can be applied to both batch and streaming data. In this case, we will transform the raw JSON data such that it's easier to query using Spark SQL's built-in support for manipulating complex nested schemas. Here is an abridged version of the transformation.

```

val cloudtrailEvents = rawRecords
  .select(explode($"records") as 'record)
  .select(
    unix_timestamp(
      $"record.eventTime",
      "yyyy-MM-dd'T'hh:mm:ss").cast("timestamp") as
    'timestamp, $"record.*")

```

Here, we `explode` (split) the array of records loaded from each file into separate records. We also parse the string event time string in each record to Spark's timestamp type, and flatten out the nested columns for easier querying. Note that if `cloudtrailEvents` was a batch DataFrame on a fixed set of files, then we would have written the same query, and we would have written the results only once as `parsed.write.parquet("/cloudtrail")`. Instead, we will start a **StreamingQuery** that runs continuously to transform new data as it arrives.

```

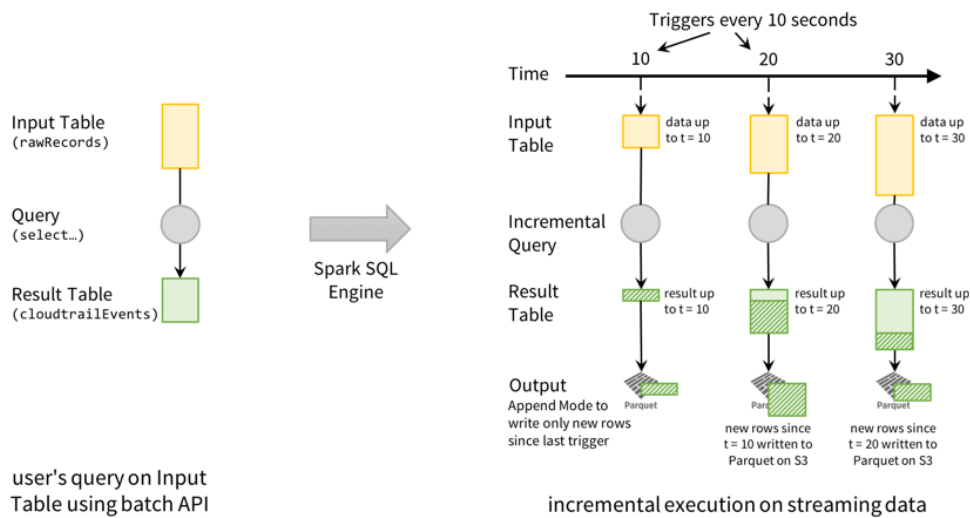
val streamingETLQuery = cloudtrailEvents
  .withColumn("date", $"timestamp".cast("date") // derive the
  date
  .writeStream
  .trigger(ProcessingTime("10 seconds")) // check for files
  every 10s
  .format("parquet") // write as Parquet partitioned by date
  .partitionBy("date")
  .option("path", "/cloudtrail")
  .option("checkpointLocation", "/cloudtrail.checkpoint/")
  .start()

```

Here we are specifying the following configurations for the StreamingQuery before starting it.

- Derive the date from the timestamp column
- Check for new files every 10 seconds (i.e., trigger interval)
- Write the transformed data from parsed DataFrame as a Parquet-formatted table at the path `/cloudtrail`.
- Partition the Parquet table by date so that we can later efficiently query time slices of the data; a key requirement in monitoring applications.
- Save checkpoint information at the path `checkpoints/cloudtrail` for fault-tolerance (explained later in the blog)

In terms of the Structured Streaming Model, this is how the execution of this query is performed.



Structured Streaming Model

users express query on streaming data using a batch API; Spark incrementalizes them to run on streams

Conceptually, the `rawRecords` DataFrame is an append-only Input Table, and the `cloudtrailEvents` DataFrame is the transformed Result Table. In other words, when new rows are appended to the input (`rawRecords`), the result table (`cloudtrailEvents`) will have new transformed rows. In this particular case, every 10 seconds, Spark SQL engine **triggers** a check for new files. When it finds new data (i.e., new rows in the Input Table), it transforms the data to generate new rows in the Result Table, which then get written out as Parquet files.

Furthermore, while this streaming query is running, you can use Spark SQL to simultaneously query the Parquet table. The streaming query writes the Parquet data transactionally such that concurrent

interactive query processing will always see a consistent view of the latest data. This strong guarantee is known as **prefix-integrity** and it makes Structured Streaming pipelines integrate nicely with the larger Continuous Application.

You can read more details about the Structured Streaming model, and its advantages over other streaming engines in our [previous blog](#).

Solving Production Challenges

Earlier, we highlighted a number of challenges that must be solved for running a streaming ETL pipeline in production. Let's see how Structured Streaming running on the Databricks platform solves them.

Recovering from Failures to get Exactly-once Fault-tolerance Guarantees

Long running pipelines must be able to tolerate machine failures. With Structured Streaming, achieving fault-tolerance is as easy as specifying a checkpoint location for the query. In the earlier code snippet, we did so in the following line.

```
.option("checkpointLocation", "/cloudtrail.checkpoint/")
```

This checkpoint directory is per query, and while a query is active, Spark continuously writes metadata of the processed data to the checkpoint directory. Even if the entire cluster fails, the query can be restarted on a new cluster, using the same checkpoint directory, and

consistently recover. More specifically, on the new cluster, Spark uses the metadata to start the new query where the failed one left off, thus ensuring end-to-end exactly-once guarantees and data consistency (see [Fault Recovery section of our previous blog](#)).

Furthermore, this same mechanism allows you to upgrade your query between restarts, as long as the input sources and output schema remain the same. Since Spark 2.1, we encode the checkpoint data in JSON for future-proof compatibility. So you can restart your query even after updating your Spark version. In all cases, you will get the same fault-tolerance and consistency guarantees.

Note that **Databricks makes it very easy to set up automatic recovery**, as we will show in the next section.

Monitoring, Alerting and Upgrading

For a Continuous Application to run smoothly, it must be robust to individual machine or even whole cluster failures. In Databricks, we have developed tight integration with Structured Streaming that allows us continuously monitor your StreamingQueries for failures (and automatically restart them. All you have to do is create a new Job, and configure the Job retry policy. You can also configure the job to send emails to notify you of failures.

Streaming Application Job



Delete

Task: Notebook at [/Users/tdas@databricks.com/Streaming Application Notebook](#) - [Edit](#) / [Remove](#)

◦ Dependent Libraries: [Add](#)

Cluster: Driver: r3.xlarge, Workers: r3.xlarge, 274.5 GB, On-Demand and Spot, fall back to On-Demand, Spark 2.1.0-db1 (Scala 2.11) [Edit](#)

Schedule: None [Edit](#)

Advanced ▾

Alerts: [Edit](#)

◦ On error: [tdas@databricks.com](#)

Maximum Concurrent Runs: 1 [Edit](#)

Timeout: None [Edit](#)

Retries: Limit 30x [Edit](#) / [Remove](#)

Application upgrades can be easily made by updating your code and/or Spark version and then restarting the Job. See our [guide on running Structured Streaming in Production](#) for more details.

Machine failures are not the only situations that we need to handle to ensure robust processing. We will discuss how to monitor for traffic spikes and upstream failures in more detail later in this series.

Combining Live Data with Historical/Batch Data

Many applications require historical/batch data to be combined with live data. For example, besides the incoming audit logs, we may already have a large backlog of logs waiting to be converted. Ideally, we would like to achieve both, interactively query the latest data as soon as possible, and also have access to historical data for future analysis. It is often complex to set up such a pipeline using most existing systems as you would have to set up multiples processes: a batch job to convert the historical data, a streaming pipeline to convert the live data, and maybe a another step to combine the results.

Structured Streaming eliminates this challenge. You can configure the above query to prioritize the processing new data files as they arrive, while using the space cluster capacity to process the old files. First, we set the option `latestFirst` for the file source to true, so that new files are processed first. Then, we set the `maxFilesPerTrigger` to limit how many files to process every time. This tunes the query to update the downstream data warehouse more frequently, so that the latest data is made available for querying as soon as possible. Together, we can define the `rawLogs` DataFrame as follows:

In this way, we can write a single query that easily combines live data with historical data, while ensuring low-latency, efficiency and data consistency.

```
val rawJson = spark.readStream
  .schema(cloudTrailSchema)
  .option("latestFirst", "true")
  .option("maxFilesPerTrigger", "20")
  .json("s3n://mybucket/AWSLogs/*/CloudTrail/*/2017/01/*")
```

Conclusion

Structured Streaming in Apache Spark is the best framework for writing your streaming ETL pipelines, and Databricks makes it easy to run them in production at scale, as we demonstrated above. We shared a high level overview of the steps—extracting, transforming, loading and finally querying—to set up your streaming ETL production pipeline. We also discussed and demonstrated how Structured

Streaming overcomes the challenges in solving and setting up high-volume and low-latency streaming pipelines in production.

In the future blog posts in this series, we'll cover how we address other hurdles, including:

- [Applying complex transformations to nested JSON data](#)
- [Processing Data in Apache Kafka with Structured Streaming in Apache Spark 2.2](#)
- [Monitoring your streaming applications](#)
- [Integrating Structured Streaming with Apache Kafka](#)
- [Computing event time aggregations with Structured Streaming](#)
- [Running Streaming Jobs Once a Day For 10x Cost Savings](#)

If you want to learn more about the Structured Streaming, here are a few useful links.

- Previous blogs posts explaining the motivation and concepts of Structured Streaming:
 - [Continuous Applications: Evolving Streaming in Apache Spark 2.0](#)
 - [Structured Streaming In Apache Spark](#)
- [Structured Streaming Programming Guide for Apache Spark 2.1](#)
- [Spark Summit 2016 Talk – A Deep Dive Into Structured Streaming](#)

What's Next

You can try two notebooks with your own AWS CloudTrail Logs. [Import the notebooks into Databricks.](#)

- [Try the Scala Notebook](#)
- [Try the Python Notebook](#)




Part 3:

Fixing Data Transformation
Challenges

Working with Complex Data Formats with Structured Streaming in Apache Spark 2.1

Part 2 of Scalable Data @ Databricks

February 23, 2017 | by Burak Yavuz, Michael Armbrust, Tathagata Das and Tyson Condie

 Try this notebook in Databricks: [Scala Notebook](#), [Python Notebook](#), [SQL Notebook](#)

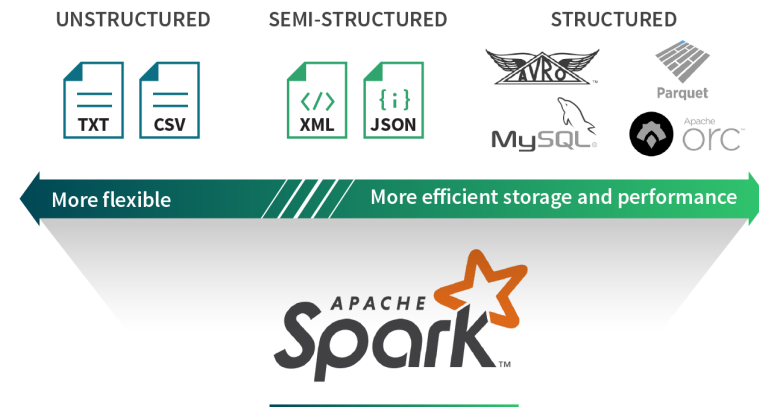
In [part 1](#) of this series on Structured Streaming blog posts, we demonstrated how easy it is to write an end-to-end streaming ETL pipeline using Structured Streaming that converts JSON CloudTrail logs into a Parquet table. The blog highlighted that one of the major challenges in building such pipelines is to read and transform data from various sources and complex formats. In this blog post, we are going to examine this problem in further detail, and show how Apache Spark SQL's built-in functions can be used to solve all your data transformation challenges.

Specifically, we are going to discuss the following:

- What are the different data formats and their tradeoffs
- How to work with them easily using Spark SQL
- How to choose the right final format for your use case

Data sources and formats

Data is available in a myriad of different formats. Spreadsheets can be expressed in XML, CSV, TSV; application metrics can be written out in raw text or JSON. Every use case has a particular data format tailored for it. In the world of Big Data, we commonly come across formats like Parquet, ORC, Avro, JSON, CSV, SQL and NoSQL data sources, and plain text files. We can broadly classify these data formats into three categories: structured, semi-structured, and unstructured data. Let's try to understand the benefits and shortcomings of each category.



Structured data

Structured data sources define a schema on the data. With this extra bit of information about the underlying data, structured data sources provide efficient storage and performance. For example, columnar formats such as Parquet and ORC make it much easier to extract values from a subset of columns. Reading each record row by row first, then extracting the values from the specific columns of interest can read much more data than what is necessary when a query is only interested in a small fraction of the columns. A row-based storage format such as Avro efficiently serializes and stores data providing storage benefits. However, these advantages often come at the cost of flexibility. For example, because of rigidity in structure, evolving a schema can be challenging.

Unstructured data

By contrast, unstructured data sources are generally free-form text or binary objects that contain no markup, or metadata (e.g., commas in CSV files), to define the organization of data. Newspaper articles, medical records, image blobs, application logs are often treated as unstructured data. These sorts of sources generally require context around the data to be parseable. That is, you need to know that the file is an image or is a newspaper article. Most sources of data are unstructured. The cost of having unstructured formats is that it becomes cumbersome to extract value out of these data sources as many transformations and feature extraction techniques are required to interpret these datasets.

Semi-structured data

Semi-structured data sources are structured per record but don't necessarily have a well-defined global schema spanning all records. As a result, each data record is augmented with its schema information. JSON and XML are popular examples. The benefits of semi-structured data formats are that they provide the most flexibility in expressing your data as each record is self-describing. These formats are very common across many applications as many lightweight parsers exist for dealing with these records, and they also have the benefit of being human readable. However, the main drawback for these formats is that they incur extra parsing overheads, and are not particularly built for ad-hoc querying.

Interchanging data formats with Spark SQL

In our [previous blog post](#), we discussed how transforming Cloudtrail Logs from JSON into Parquet shortened the runtime of our ad-hoc queries by 10x. Spark SQL allows users to ingest data from these classes of data sources, both in batch and streaming queries. It natively supports reading and writing data in Parquet, ORC, JSON, CSV, and text format and a plethora of other connectors exist on [Spark Packages](#). You may also connect to SQL databases using the JDBC DataSource.

Apache Spark can be used to interchange data formats as easily as:

```

events = spark.readStream \
  .format("json") \           # or parquet, kafka, orc...
  .option() \                # format specific options
  .schema(my_schema) \      # required
  .load("path/to/data")

output = ...                  # perform your transformations

output.writeStream \        # write out your data
  .format("parquet") \
  .start("path/to/write")

```

Whether batch or streaming data, we know how to read and write to different data sources and formats, but different sources support different kinds of schema and data types. Traditional databases only support primitive data types, whereas formats like JSON allow users to nest objects within columns, have an array of values or represent a set of key-value pairs. Users will generally have to go in-between these data types to efficiently store and represent their data. Fortunately, Spark SQL makes it easy to handle both primitive and complex data types. Let's now dive into a quick overview of how we can go from complex data types to primitive data types and vice-a-versa.

Transforming complex data types

```

{
  "a_struct": {
    "field_1": 1,
    "field_2": "b",
    "field_3": { ... }
  }
}

{
  "a_map": {
    "key_1": 1,
    "key_2": 2,
    "key_3": 3
  }
}

{
  "an_array": [
    "Alice",
    "Bob",
    "Chris"
  ]
}

```

It is common to have complex data types such as structs, maps, and arrays when working with semi-structured formats. For example, you may be logging API requests to your web server. This API request will contain HTTP Headers, which would be a string-string map. The request payload may contain form-data in the form of JSON, which may contain nested fields or arrays. Some sources or formats may or may not support complex data types. Some formats may provide performance benefits when storing the data in a specific data type. For example, when using Parquet, all struct columns will receive the same treatment as top-level columns. Therefore, if you have filters on a nested field, you will get the same benefits as a top-level column. However, maps are treated as two array columns, hence you wouldn't receive efficient filtering semantics.

Let's look at some examples on how Spark SQL allows you to shape your data ad libitum with some data transformation techniques.

Selecting from nested columns

Dots (.) can be used to access nested columns for structs and maps.

```
// input
{
  "a": {
    "b": 1
  }
}

Python: events.select("a.b")
Scala: events.select("a.b")
SQL: select a.b from events

// output
{
  "b": 1
}
```

Flattening structs

A star (*) can be used to select all of the subfields in a struct.

```
// input
{
  "a": {
    "b": 1,
    "c": 2
  }
}

Python: events.select("a.*")
Scala: events.select("a.*")
SQL: select a.* from events

// output
{
  "b": 1,
  "c": 2
}
```

Nesting columns

The struct function or just parentheses in SQL can be used to create a new struct.

```
// input
{
  "a": 1,
  "b": 2,
  "c": 3
}

Python: events.select(struct(col("a").alias("y")).alias("x"))
Scala: events.select(struct('a as 'y) as 'x)
SQL: select named_struct("y", a) as x from events

// output
{
  "x": {
    "y": 1
  }
}
```

Nesting all columns

The star (*) can also be used to include all columns in a nested struct.

```
// input
{
  "a": 1,
  "b": 2
}

Python: events.select(struct("*").alias("x"))
Scala: events.select(struct("*") as 'x)
SQL: select struct(*) as x from events

// output
{
  "x": {
    "a": 1,
    "b": 2
  }
}
```

Selecting a single array or map element

`getItem()` or square brackets (i.e. `[]`) can be used to select a single element out of an array or a map.

```
// input
{
  "a": [1, 2]
}
```

```
Python: events.select(col("a").getItem(0).alias("x"))
```

```
Scala: events.select('a.getItem(0) as 'x)
```

```
SQL: select a[0] as x from events
```

```
// output
{ "x": 1 }
```

```
// input
{
  "a": {
    "b": 1
  }
}
```

```
Python: events.select(col("a").getItem("b").alias("x"))
```

```
Scala: events.select('a.getItem("b") as 'x)
```

```
SQL: select a['b'] as x from events
```

```
// output
{ "x": 1 }
```

Creating a row for each array or map element

`explode()` can be used to create a new row for each element in an array or each key-value pair. This is similar to LATERAL VIEW EXPLODE in HiveQL.

```
// input
{
  "a": [1, 2]
}
```

```
Python: events.select(explode("a").alias("x"))
```

```
Scala: events.select(explode('a) as 'x)
```

```
SQL: select explode(a) as x from events
```

```
// output
[{"x": 1 }, {"x": 2 }]
```

```
// input
{
  "a": {
    "b": 1,
    "c": 2
  }
}
```

```
Python: events.select(explode("a").alias("x", "y"))
```

```
Scala: events.select(explode('a) as Seq("x", "y"))
```

```
SQL: select explode(a) as (x, y) from events
```

```
// output
[{"x": "b", "y": 1 }, {"x": "c", "y": 2 }]
```

Collecting multiple rows into an array

`collect_list()` and `collect_set()` can be used to aggregate items into an array.

```
// input
[{"x": 1 }, {"x": 2 }]

Python: events.select(collect_list("x").alias("x"))
Scala: events.select(collect_list('x) as 'x)
SQL: select collect_list(x) as x from events

// output
{"x": [1, 2] }
```

```
// input
[{"x": 1, "y": "a" }, {"x": 2, "y": "b" }]

Python: events.groupBy("y").agg(collect_list("x").alias("x"))
Scala: events.groupBy("y").agg(collect_list('x) as 'x)
SQL: select y, collect_list(x) as x from events group by y

// output
[{"y": "a", "x": [1]}, {"y": "b", "x": [2]}]
```

Selecting one field from each item in an array

When you use dot notation on an array we return a new array where that field has been selected from each array element.

```
// input
{
  "a": [
    {"b": 1},
    {"b": 2}
  ]
}

Python: events.select("a.b")
Scala: events.select("a.b")
SQL: select a.b from events

// output
{
  "b": [1, 2]
}
```

Power of `to_json()` and `from_json()`

What if you really want to preserve your column's complex structure but you need it to be encoded as a string to store it? Are you doomed? Of course not! Spark SQL provides functions like `to_json()` to encode a struct as a string and `from_json()` to retrieve the struct as a complex type. Using JSON strings as columns are useful when reading from or writing to a streaming source like Kafka. Each Kafka key-value record will be augmented with some metadata, such as the ingestion timestamp into Kafka, the offset in Kafka, etc. If the "value" field that contains your data is in JSON, you could use `from_json()` to extract your data, enrich it, clean it, and then push it downstream to Kafka again or write it out to a file.

Encode a struct as json

`to_json()` can be used to turn structs into JSON strings. This method is particularly useful when you would like to re-encode multiple columns into a single one when writing data out to Kafka. This method is not presently available in SQL.

```
// input
{
  "a": {
    "b": 1
  }
}
```

```
Python: events.select(to_json("a").alias("c"))
Scala: events.select(to_json('a) as 'c)
```

```
// output
{
  "c": "{\"b\":1}"
}
```

Decode json column as a struct

`from_json()` can be used to turn a string column with JSON data into a struct. Then you may flatten the struct as described above to have individual columns. This method is not presently available in SQL.

```
// input
{
  "a": "{\"b\":1}"
}
```

Python:

```
schema = StructType().add("b", IntegerType())
events.select(from_json("a", schema).alias("c"))
```

Scala:

```
val schema = new StructType().add("b", IntegerType)
events.select(from_json('a, schema) as 'c)
```

```
// output
{
  "c": {
    "b": 1
  }
}
```

Sometimes you may want to leave a part of the JSON string still as JSON to avoid too much complexity in your schema.


```
// input
{
  "a": "{\"b\": {\"x\": 1, \"y\": {\"z\": 2}}}"
}

Python:
schema = StructType().add("b", StructType().add("x",
IntegerType())
                                .add("y", StringType()))
events.select(from_json("a", schema).alias("c"))
Scala:
val schema = new StructType().add("b", new
StructType().add("x", IntegerType)
                                .add("y", StringType))
events.select(from_json('a, schema) as 'c)

// output
{
  "c": {
    "b": {
      "x": 1,
      "y": "{\"z\": 2}"
    }
  }
}
```

Parse a set of fields from a column containing JSON

`json_tuple()` can be used to extract fields available in a string column with JSON data.

```
// input
{
  "a": "{\"b\": 1}"
}

Python: events.select(json_tuple("a", "b").alias("c"))
Scala:  events.select(json_tuple('a, "b") as 'c)
SQL:    select json_tuple(a, "b") as c from events

// output
{ "c": 1 }
```

Sometimes a string column may not be self-describing as JSON, but may still have a well-formed structure. For example, it could be a log message generated using a specific Log4j format. Spark SQL can be used to structure those strings for you with ease!

Parse a well-formed string column

`regexp_extract()` can be used to parse strings using regular expressions.

```
// input
[{"a": "x: 1" }, {"a": "y: 2" }]

Python: events.select(regexp_extract("a", "([a-z]):",
1).alias("c"))
Scala:  events.select(regexp_extract('a, "([a-z]):", 1) as 'c)
SQL:    select regexp_extract(a, "([a-z]):", 1) as c from
events

// output
[{"c": "x" }, {"c": "y" }]
```

That's a lot of transformations! Let's now look at some real life use cases to put all of these data formats, and data manipulation capabilities to good use.

Harnessing all of this power

At Databricks, we collect logs from our services and use them to perform real-time monitoring to detect issues, before our customers are affected. Log files are unstructured files, but they are parseable because they have a well-defined Log4j format. We run a log collector service that sends each log entry and additional metadata about the entry (e.g. source) in JSON to Kinesis. These JSON records are then batch-uploaded to S3 as files. Querying these JSON logs to answer any question is tedious: these files contain duplicates, and for answering any query, even if it involves a single column, the whole JSON record may require deserialization.

To address this issue, we run a pipeline that reads these JSON records and performs de-duplication on the metadata. Now we are left with the original log record, which may be in JSON format or as unstructured text. If we're dealing with JSON, we use `from_json()` and several of the transformations described above to format our data. If it is text, we use methods such as `regexp_extract()` to parse our Log4j format into a more structured form. Once we are done with all of our transformations and restructuring, we save the records in Parquet partitioned by date. This gives us 10-100x speed-up when answering questions like "how many ERROR messages did we see between

10:00-10:30 for this specific service"? The speed-ups can be attributed to:

- We no longer pay the price of deserializing JSON records
- We don't have to perform complex string comparisons on the original log message
- We only have to extract two columns in our query: the time, and the log level

Here are a few more common use cases that we have seen among our customers:

"I would like to run a Machine Learning pipeline with my data. My data is already pre-processed, and I will use all my features throughout the pipeline."

Avro is a good choice when you will access the whole row of data.

"I have an IoT use case where my sensors send me events. For each event the metadata that matters is different."

In cases where you would like flexibility in your schema, you may consider using JSON to store your data.

“I would like to train a speech recognition algorithm on newspaper articles or sentiment analysis on product comments.”

In cases where your data may not have a fixed schema, nor a fixed pattern/structure, it may just be easier to store it as plain text files. You may also have a pipeline that performs feature extraction on this unstructured data and stores it as Avro in preparation for your Machine Learning pipeline.

Conclusion

In this blog post, we discussed how Spark SQL allows you to consume data from many sources and formats, and easily perform transformations and interchange between these data formats. We shared how we curate our data at Databricks, and considered other production use cases where you may want to do things differently.

Spark SQL provides you with the necessary tools to access your data wherever it may be, in whatever format it may be in and prepare it for downstream applications either with low latency on streaming data or high throughput on old historical data!

In the future blog posts in this series, we'll cover more on:

- Monitoring your streaming applications
- Integrating Structured Streaming with Apache Kafka
- Computing event time aggregations with Structured Streaming

If you want to learn more about the Structured Streaming, here are a few useful links.

- Previous blogs posts explaining the motivation and concepts of Structured Streaming:
 - [Continuous Applications: Evolving Streaming in Apache Spark 2.0](#)
 - [Structured Streaming In Apache Spark](#)
- [Processing Data in Apache Kafka with Structured Streaming in Apache Spark 2.2](#)
 - [Real-time Streaming ETL with Structured Streaming in Apache Spark 2.1](#)
- [Structured Streaming Programming Guide](#)
- [Talk at Spark Summit 2017 East – Making Structured Streaming Ready for Production and Future Directions](#)

Finally, try our example notebooks that demonstrate transforming complex data types in Python, Scala, or SQL in [Databricks](#):

- [Python Notebook](#)
- [Scala Notebook](#)
- [SQL Notebook](#)



Part 4:

Processing Data in

Apache Kafka

Processing Data in Apache Kafka with Structured Streaming in Apache Spark 2.2

Part 3 of Scalable Data @ Databricks

April 26, 2017 | by Kunal Khamar, Tyson Condie and Michael Armbrust

In this blog, we will show how Spark SQL's APIs can be leveraged to consume and transform complex data streams from [Apache Kafka](#). Using these simple APIs, you can express complex transformations like exactly-once event-time aggregation and output the results to a variety of systems. Together, you can use Apache Spark and Apache Kafka to:

- Transform and augment real-time data read from Apache Kafka using the same APIs as working with batch data.
- Integrate data read from Kafka with information stored in other systems including S3, HDFS, or MySQL.
- Automatically benefit from incremental execution provided by the Catalyst optimizer and subsequent efficient code generation by Tungsten.

We start with a review of Kafka terminology and then present examples of Structured Streaming queries that read data from and write data to Apache Kafka. And finally, we'll explore an end-to-end real-world use case.

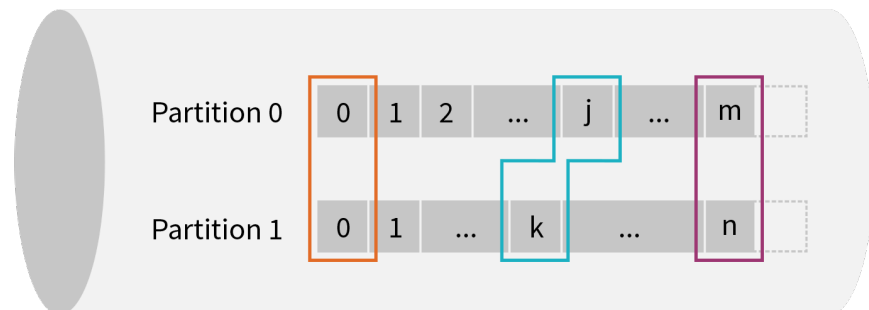
Apache Kafka

Kafka is a distributed pub-sub messaging system that is popular for ingesting real-time data streams and making them available to downstream consumers in a parallel and fault-tolerant manner. This renders Kafka suitable for building real-time streaming data pipelines that reliably move data between heterogeneous processing systems. Before we dive into the details of Structured Streaming's Kafka support, let's recap some basic concepts and terms.

Data in Kafka is organized into *topics* that are split into *partitions* for parallelism. Each partition is an ordered, immutable sequence of *records*, and can be thought of as a structured commit log. Producers append records to the tail of these logs and *consumers* read the logs at their own pace. Multiple consumers can *subscribe* to a topic and receive incoming records as they arrive. As new records arrive to a partition in a Kafka topic, they are assigned a sequential id number called the *offset*. A Kafka cluster retains all published records—whether or not they have been consumed—for a configurable retention period, after which they are marked for deletion.

Specifying What Data to Read from Kafka

Kafka Topic



Earliest	Assign	Latest
"0" → 0 "1" → 0	"0" → j "1" → k	"0" → m "1" → n

A Kafka topic can be viewed as an infinite stream where data is retained for a configurable amount of time. The infinite nature of this stream means that when starting a new query, we have to first decide what data to read and where in time we are going to begin. At a high level, there are three choices:

- *earliest* — start reading at the beginning of the stream. This excludes data that has already been deleted from Kafka because it was older than the retention period (“aged out” data).
- *latest* — start now, processing only new data that arrives after the query has started.
- *per-partition assignment* — specify the precise offset to start from for every partition, allowing fine-grained control over exactly where processing should start. For example, if we want to pick up exactly where some other system or query left off, then this option can be leveraged.

As you will see below, the `startingOffsets` option accepts one of the three options above, and is only used when starting a query from a fresh checkpoint. If you [restart a query from an existing checkpoint](#), then it will always resume exactly where it left off, except when the data at that offset has been aged out. If any unprocessed data was aged out, the query behavior will depend on what is set by the `failOnDataLoss` option, which is described in the [Kafka Integration Guide](#).

Existing users of the `KafkaConsumer` will notice that Structured Streaming provides a more granular version of the configuration option, `auto.offset.reset`. Instead of one option, we split these concerns into two different parameters, one that says what to do when the stream is first starting (`startingOffsets`), and another that handles

what to do if the query is not able to pick up from where it left off, because the desired data has already been aged out (*failOnDataLoss*).

Apache Kafka support in Structured Streaming

Structured Streaming provides a unified batch and streaming API that enables us to view data published to Kafka as a `DataFrame`. When processing unbounded data in a streaming fashion, we use the same API and get the same data consistency guarantees as in batch processing. The system ensures end-to-end exactly-once fault-tolerance guarantees, so that a user does not have to reason about low-level aspects of streaming.

Let's examine and explore examples of reading from and writing to Kafka, followed by an end-to-end application.

Reading Records from Kafka topics

The first step is to specify the location of our Kafka cluster and which topic we are interested in reading from. Spark allows you to read an individual topic, a specific set of topics, a regex pattern of topics, or even a specific set of partitions belonging to a set of topics. We will only look at an example of reading from an individual topic, the other possibilities are covered in the [Kafka Integration Guide](#).

```
# Construct a streaming DataFrame that reads from topic1
df = spark \
    .readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers",
"host1:port1,host2:port2") \
    .option("subscribe", "topic1") \
    .option("startingOffsets", "earliest") \
    .load()
```

The `DataFrame` above is a streaming `DataFrame` subscribed to “topic1”. The configuration is set by providing options to the `DataStreamReader`, and the minimal required parameters are the location of the `kafka.bootstrap.servers` (i.e. `host:port`) and the topic that we want to subscribe to. Here, we have also specified `startingOffsets` to be “earliest”, which will read all data available in the topic at the start of the query. If the `startingOffsets` option is not specified, the default value of “latest” is used and only data that arrives after the query starts will be processed.

`df.printSchema()` reveals the schema of our `DataFrame`.

```

root
 |-- key: binary (nullable = true)
 |-- value: binary (nullable = true)
 |-- topic: string (nullable = true)
 |-- partition: integer (nullable = true)
 |-- offset: long (nullable = true)
 |-- timestamp: timestamp (nullable = true)
 |-- timestampType: integer (nullable = true)

```

The returned DataFrame contains all the familiar fields of a Kafka record and its associated metadata. We can now use all of the familiar DataFrame or Dataset operations to transform the result. Typically, however, we'll start by parsing the binary values present in the key and value columns. How to interpret these blobs is application specific. Fortunately, Spark SQL contains many built-in transformations for common types of serialization as we'll show below.

Data Stored as a UTF8 String

If the bytes of the Kafka records represent UTF8 strings, we can simply use a cast to convert the binary data into the correct type.

```
df.selectExpr("CAST(key AS STRING)", "CAST(value AS STRING)")
```

Data Stored as JSON

JSON is another common format for data that is written to Kafka. In this case, we can use the built-in `from_json` function along with the expected schema to convert a binary value into a Spark SQL struct.

```

# value schema: { "a": 1, "b": "string" }
schema = StructType().add("a", IntegerType()).add("b",
StringType())
df.select( \
    col("key").cast("string"),
    from_json(col("value").cast("string"), schema))

```

User Defined Serializers and Deserializers

In some cases, you may already have code that implements the Kafka Deserializer interface. You can take advantage of this code by wrapping it as a user defined function (UDF) using the Scala code shown below.

```

object MyDeserializerWrapper {
    val deser = new MyDeserializer
}
spark.udf.register("deserialize", (topic: String, bytes:
Array[Byte]) =>
    MyDeserializerWrapper.deser.deserialize(topic, bytes)
)

df.selectExpr("""deserialize("topic1", value) AS message""")

```

Note that the DataFrame code above is analogous to specifying `value.deserializer` when using the standard Kafka consumer.

Using Spark as a Kafka Producer

Writing data from any Spark supported data source into Kafka is as simple as calling `writeStream` on any DataFrame that contains a

column named “value”, and optionally a column named “key”. If a key column is not specified, then a null valued key column will be automatically added. A null valued key column may, in some cases, [lead to uneven data partitioning in Kafka](#), and should be used with care.

The destination topic for the records of the DataFrame can either be specified statically as an option to the `DataStreamWriter` or on a per-record basis as a column named “topic” in the DataFrame.

```
# Write key-value data from a DataFrame to a Kafka topic
# specified in an option
query = df \
  .selectExpr("CAST(userId AS STRING) AS key",
    "to_json(struct(*)) AS value") \
  .writeStream \
  .format("kafka") \
  .option("kafka.bootstrap.servers",
    "host1:port1,host2:port2") \
  .option("topic", "topic1") \
  .option("checkpointLocation", "/path/to/HDFS/dir") \
  .start()
```

The above query takes a DataFrame containing user information and writes it to Kafka. The `userId` is serialized as a string and used as the key. We take all the columns of the DataFrame and serialize them as a JSON string, putting the results in the value of the record.

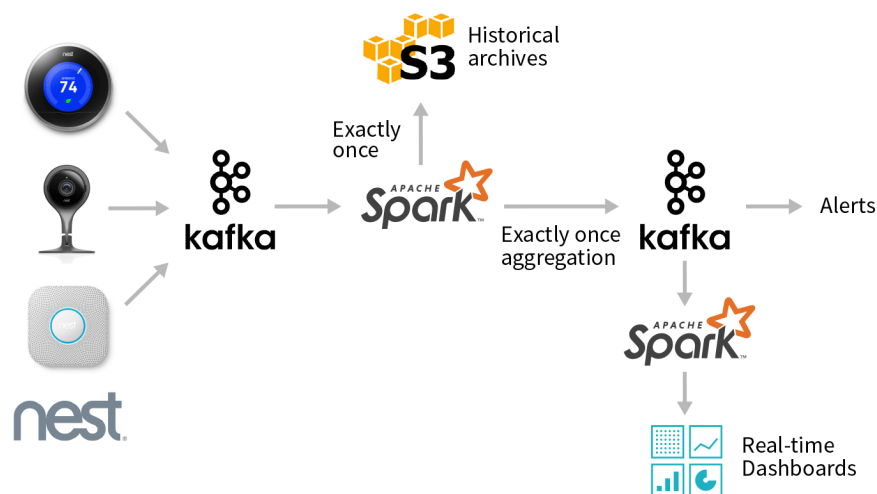
The two required options for writing to Kafka are the `kafka.bootstrap.servers` and the `checkpointLocation`. As in the above example, an additional topic option can be used to set a single topic to write to, and this option will override the “topic” column if it exists in the DataFrame.

End-to-End Example with Nest Devices

In this section, we will explore an end-to-end pipeline involving Kafka along with other data sources and sinks. We will work with a data set involving a collection of [Nest](#) device logs, with a JSON format [described here](#). We’ll specifically examine data from Nest’s cameras, which look like the following JSON:

```
"devices": {
  "cameras": {
    "device_id": "awJo6rH...",
    "last_event": {
      "has_sound": true,
      "has_motion": true,
      "has_person": true,
      "start_time": "2016-12-29T00:00:00.000Z",
      "end_time": "2016-12-29T18:42:00.000Z"
    }
  }
}
```

We'll also be joining with a static dataset (called "device_locations") that contains a mapping from `device_id` to the `zip_code` where the device was registered.



At a high-level, the desired workflow looks like the graph above. Given a stream of updates from Nest cameras, we want to use Spark to perform several different tasks:

- Create an efficient, queryable historical archive of all events using a columnar format like Parquet.
- Perform low-latency event-time aggregation and push the results back to Kafka for other consumers.
- Perform batch reporting on the data stored in a compacted topic in Kafka.

While these may sound like wildly different use-cases, you can perform all of them using DataFrames and Structured Streaming in a single end-to-end Spark application! In the following sections, we'll walk through individual steps, starting from ingest to processing to storing aggregated results.

Read Nest Device Logs From Kafka

Our first step is to read the raw Nest data stream from Kafka and project out the camera data that we are interested in. We first parse the Nest JSON from the Kafka records, by calling the `from_json` function and supplying the expected JSON schema and timestamp format. Then, we apply various transformations to the data and project the columns related to camera data in order to simplify working with the data in the sections to follow.

Expected Schema for JSON data

```
schema = StructType() \
    .add("metadata", StructType() \
        .add("access_token", StringType()) \
        .add("client_version", IntegerType())) \
    .add("devices", StructType() \
        .add("thermostats", MapType(StringType(), StructType().add(...))) \
        .add("smoke_co_alarms", MapType(StringType(), StructType().add(...))) \
        .add("cameras", MapType(StringType(), StructType().add(...))) \
        .add("companyName", StructType().add(...))) \
    .add("structures", MapType(StringType(), StructType().add(...)))

nestTimestampFormat = "yyyy-MM-dd'T'HH:mm:ss.sss'Z'"
```

Parse the Raw JSON

```
jsonOptions = { "timestampFormat": nestTimestampFormat }
parsed = spark \
  .readStream \
  .format("kafka") \
  .option("kafka.bootstrap.servers", "localhost:9092") \
  .option("subscribe", "nest-logs") \
  .load() \
  .select(from_json(col("value").cast("string"), schema,
jsonOptions).alias("parsed_value"))
```

Project Relevant Columns

```
camera = parsed \
  .select(explode("parsed_value.devices.cameras")) \
  .select("value.*")

sightings = camera \
  .select("device_id", "last_event.has_person",
"last_event.start_time") \
  .where(col("has_person") == True)
```

To create the `camera` DataFrame, we first unnest the “cameras” json field to make it top level. Since “cameras” is a `MapType`, each resulting row contains a map of key-value pairs. So, we use the `explode` function to create a new row for each key-value pair, flattening the data. Lastly, we use `star ()` to unnest the “value” column. The following is the result of calling `camera.printSchema()`

```
root
 |-- device_id: string (nullable = true)
 |-- software_version: string (nullable = true)
 |-- structure_id: string (nullable = true)
 |-- where_id: string (nullable = true)
 |-- where_name: string (nullable = true)
 |-- name: string (nullable = true)
 |-- name_long: string (nullable = true)
 |-- is_online: boolean (nullable = true)
 |-- is_streaming: boolean (nullable = true)
 |-- is_audio_input_enable: boolean (nullable = true)
 |-- last_is_online_change: timestamp (nullable = true)
 |-- is_video_history_enabled: boolean (nullable = true)
 |-- web_url: string (nullable = true)
 |-- app_url: string (nullable = true)
 |-- is_public_share_enabled: boolean (nullable = true)
 |-- activity_zones: array (nullable = true)
 |   |-- element: struct (containsNull = true)
 |   |   |-- name: string (nullable = true)
 |   |   |-- id: string (nullable = true)
 |-- public_share_url: string (nullable = true)
 |-- snapshot_url: string (nullable = true)
 |-- last_event: struct (nullable = true)
 |   |-- has_sound: boolean (nullable = true)
 |   |-- has_motion: boolean (nullable = true)
 |   |-- has_person: boolean (nullable = true)
 |   |-- start_time: timestamp (nullable = true)
 |   |-- end_time: timestamp (nullable = true)
 |   |-- urls_expire_time: timestamp (nullable = true)
 |   |-- web_url: string (nullable = true)
 |   |-- app_url: string (nullable = true)
 |   |-- image_url: string (nullable = true)
 |   |-- animated_image_url: string (nullable = true)
 |   |-- activity_zone_ids: array (nullable = true)
 |   |   |-- element: string (containsNull = true)
```

Aggregate and Write Back to Kafka

We will now process the `sightings` DataFrame by augmenting each sighting with its location. Recall that we have some location data that lets us look up the zip code of a device by its device id. We first create a DataFrame representing this location data, and then join it with the `sightings` DataFrame, matching on device id. What we are doing here is joining the *streaming* DataFrame `sightings` with a *static* DataFrame of locations!

Add Location Data

```
locationDF =  
spark.table("device_locations").select("device_id",  
"zip_code")  
sightingLoc = sightings.join(locationDF, "device_id")
```

Aggregate Statistics and Write Out to Kafka

Now, let's generate a streaming aggregate that counts the number of camera person sightings in each zip code for each hour, and write it out to a compacted Kafka topic¹ called "nest-camera-stats".

```
sightingLoc \  
  .groupBy("zip_code", window("start_time", "1 hour")) \  
  .count() \  
  .select( \  
    to_json(struct("zip_code", "window")).alias("key"),  
    col("count").cast("string").alias("value")) \  
  .writeStream \  
  .format("kafka") \  
  .option("kafka.bootstrap.servers", "localhost:9092") \  
  .option("topic", "nest-camera-stats") \  
  .option("checkpointLocation", "/path/to/HDFS/dir") \  
  .outputMode("complete") \  
  .start()
```

The above query will process any sighting as it occurs and write out the updated count of the sighting to Kafka, keyed on the zip code and hour window of the sighting. Over time, many updates to the same key will result in many records with that key, and Kafka topic compaction will delete older updates as new values arrive for the key. This way, compaction tries to ensure that eventually, only the latest value is kept for any given key.

Archive Results in Persistent Storage

In addition to writing out aggregation results to Kafka, we may want to save the raw camera records in persistent storage for later use. The following example writes out the camera DataFrame to S3 in Parquet format. We have chosen Parquet for compression and columnar storage, though many different formats such as ORC, Avro, CSV, etc. are supported to tailor to varied use-cases.

```
camera.writeStream \  
  .format("parquet") \  
  .option("startingOffsets", "earliest") \  
  .option("path", "s3://nest-logs") \  
  .option("checkpointLocation", "/path/to/HDFS/dir") \  
  .start()
```

Note that we can simply reuse the same `camera` DataFrame to start multiple streaming queries. For instance, we can query the DataFrame to get a list of cameras that are offline, and send a notification to the network operations center for further investigation.

Batch Query for Reporting

Our next example is going to run a batch query over the Kafka “nest-camera-stats” compacted topic and generate a report showing zip codes with a significant number of sightings.

Writing batch queries is similar to streaming queries with the exception that we use the `read` method instead of the `readStream` method and `write` instead of `writeStream`.

Batch Read and Format the Data

```
report = spark \  
  .read \  
  .format("kafka") \  
  .option("kafka.bootstrap.servers", "localhost:9092") \  
  .option("subscribe", "nest-camera-stats") \  
  .load() \  
  .select( \  
    json_tuple(col("key").cast("string"), "zip_code", \  
      "window").alias("zip_code", "window"), \  
    col("value").cast("string").cast("integer").alias("count")) \  
  .where("count > 1000") \  
  .select("zip_code", "window") \  
  .distinct()
```

This report DataFrame can be used for reporting or to create a real-time dashboard showing events with extreme sightings.

Conclusion

In this blog post, we showed examples of consuming and transforming real-time data streams from Kafka. We implemented an end-to-end example of a [continuous application](#), demonstrating the conciseness and ease of programming with Structured Streaming APIs, while leveraging the powerful exactly-once semantics these APIs provide.

In the future blog posts in this series, we’ll cover more on:

- Monitoring your streaming applications
- Computing event-time aggregations with Structured Streaming

If you want to learn more about the Structured Streaming, here are a few useful links:

- Previous blogs posts explaining the motivation and concepts of Structured Streaming:
 - [Continuous Applications: Evolving Streaming in Apache Spark 2.0](#)
 - [Structured Streaming In Apache Spark](#)
 - [Real-time Streaming ETL with Structured Streaming in Apache Spark 2.1](#)
 - [Working with Complex with Structured Streaming in Apache Spark 2.1](#)
- [Structured Streaming Programming Guide](#)
- [Talk at Spark Summit 2017 East – Making Structured Streaming Ready for Production and Future Directions](#)

To try Structured Streaming in Apache Spark 2.1, [try Databricks today](#).

Additional Configuration

[Kafka Integration Guide](#)

Contains further examples and Spark specific configuration options for processing data in Kafka.

[Kafka Consumer and Producer Configuration Docs](#)

Kafka's own configurations can be set via `DataStreamReader.option` and `DataStreamWriter.option` with the `kafka.` prefix, e.g:

For possible `kafka` parameters, see the [Kafka consumer config](#) docs for parameters related to reading data, and the [Kafka producer config](#) docs for parameters related to writing data.

See the [Kafka Integration Guide](#) for the list of options managed by Spark, which are consequently not configurable.



¹ A compacted Kafka topic is a topic where retention is enforced by compaction to ensure that the log is guaranteed to have at least the last state for each key. See [Kafka Log Compaction](#) for more information.

Part 5: Event-time Aggregations and Watermarking

Event-time Aggregation and Watermarking in Apache Spark's Structured Streaming

Part 4 of Scalable Data @ Databricks

February 23, 2017 | by Tathagata Das

Continuous applications often require near real-time decisions on real-time aggregated statistics—such as health of and readings from IoT devices or detecting anomalous behavior. In this blog, we will explore how easily streaming aggregations can be expressed in Structured Streaming, and how naturally late, and out-of-order data is handled.

Streaming Aggregations

Structured Streaming allows users to express the same streaming query as a batch query, and the Spark SQL engine incrementalizes the query and executes on streaming data. For example, suppose you have a streaming DataFrame having events with signal strength from IoT devices, and you want to calculate the running average signal strength for each device, then you would write the following Python code:

```
# DataFrame w/ schema [eventTime: timestamp, deviceId: string,
signal: bigint]
eventsDF = ...

avgSignalDF = eventsDF.groupBy("deviceId").avg("signal")
```

This code is no different if eventsDF was a DataFrame on static data. However, in this case, the average will be continuously updated as new events arrive. You choose different *output modes* for writing the updated averages to external systems like file systems and databases. Furthermore, you can also implement custom aggregations using [Spark's user-defined aggregation function \(UDAFs\)](#).

Aggregations on Windows over Event-Time

In many cases, rather than running aggregations over the whole stream, you want aggregations over data bucketed by time windows (say, every 5 minutes or every hour). In our earlier example, it's insightful to see what is the average signal strength in last 5 minutes in case if the devices have started to behave anomalously. Also, this 5 minute window should be based on the timestamp embedded in the data (aka. event-time) and not on the time it is being processed (aka. processing-time).

Earlier Spark Streaming DStream APIs made it hard to express such event-time windows as the API was designed solely for processing-time windows (that is, windows on the time the data arrived in Spark). In Structured Streaming, expressing such windows on event-time is simply performing a special grouping using the `window()` function. For example, counts over 5 minute tumbling (non-overlapping) windows on the eventTime column in the event is as following.

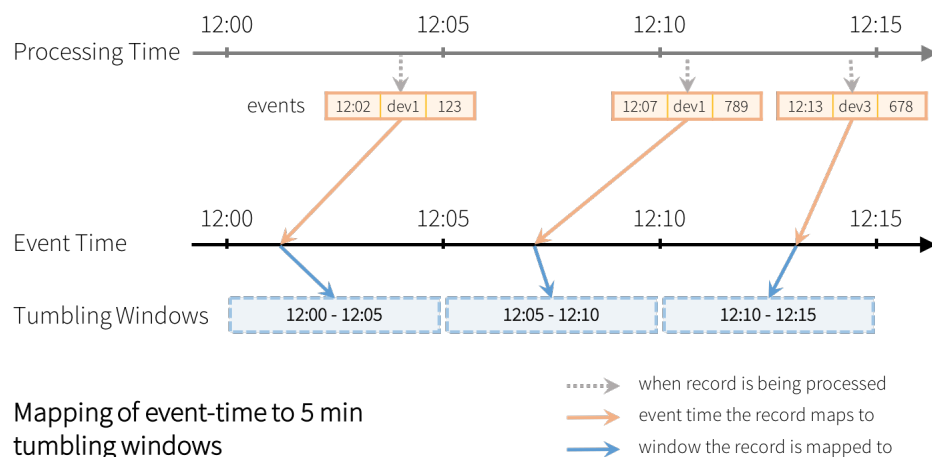

```

from pyspark.sql.functions import *

windowedAvgSignalDF = \
    eventsDF \
        .groupBy(window("eventTime", "5 minute")) \
        .count()

```

In the above query, every record is going to be assigned to a 5 minute tumbling window as illustrated below.



Each window is a group for which running counts are calculated. You can also define overlapping windows by specifying both the window length and the sliding interval. For example:

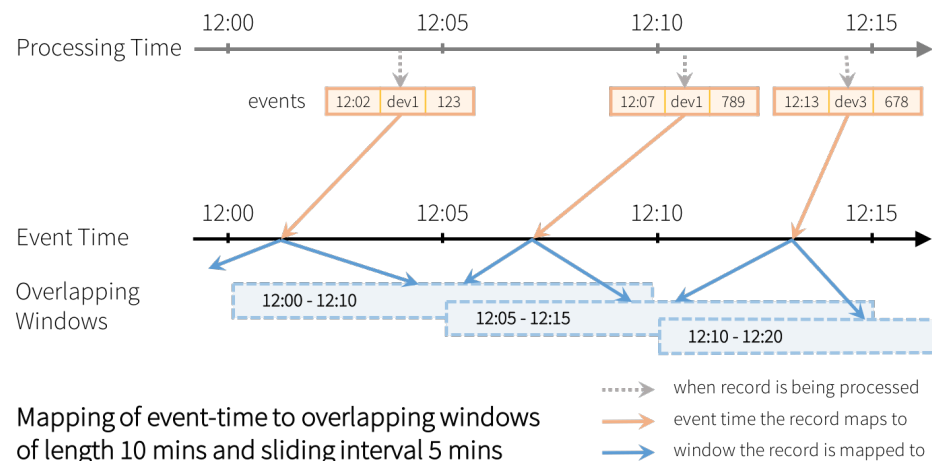
```

from pyspark.sql.functions import *

windowedAvgSignalDF = \
    eventsDF \
        .groupBy(window("eventTime", "10 minutes", "5 minutes")) \
        .count()

```

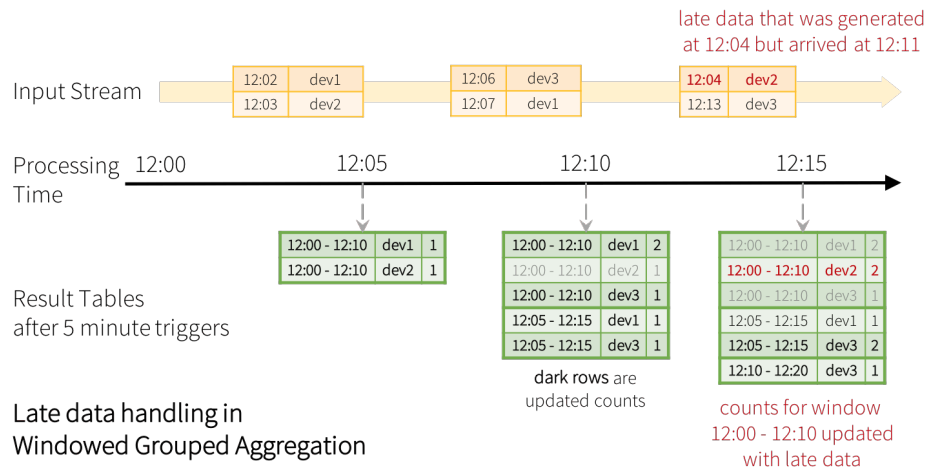
In the above query, every record will be assigned to multiple overlapping windows as illustrated below. Mapping of event-time to overlapping windows of length 10 mins and sliding interval 5 mins



This grouping strategy automatically handles late and out-of-order data — the late event would just update older window groups instead of the latest ones. Here is an end-to-end illustration of a query that is grouped by both the `deviceId` and the overlapping windows. The illustration below shows how the final result of a query changes after

new data is processed with 5 minute triggers when you are grouping by both `deviceId` and sliding windows (for brevity, the “signal” field is omitted).

```
windowedCountsDF = \
  eventsDF \
    .groupBy(
      "deviceId",
      window("eventTime", "10 minutes", "5 minutes")) \
    .count()
```

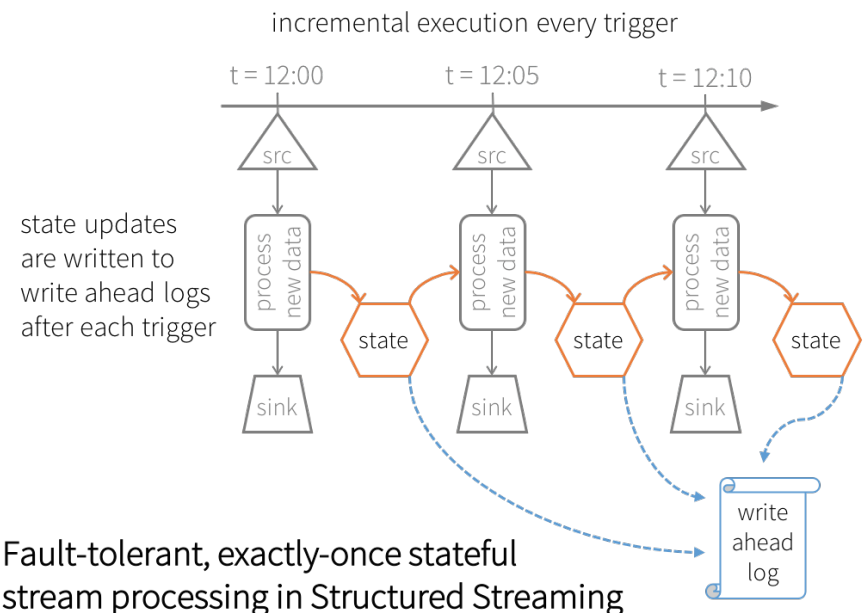


Late data handling in Windowed Grouped Aggregation

Note how the late, out-of-order record [12:04, dev2] updated an old window's count.

Stateful Incremental Execution

While executing any streaming aggregation query, the Spark SQL engine internally maintains the intermediate aggregations as fault-tolerant state. This state is structured as key-value pairs, where the key is the group, and the value is the intermediate aggregation. These pairs are stored in an in-memory, versioned, key-value “state store” in the Spark executors that is checkpointed using write ahead logs in an HDFS-compatible file system (in the configured [checkpoint location](#)). At every trigger, the state is read and updated in the state store, and all updates are saved to the write ahead log. In case of any failure, the correct version of the state is restored from checkpoint information, and the query proceeds from the point it failed. Together with replayable sources, and idempotent sinks, Structured Streaming ensures exactly-once guarantees for stateful stream processing.



Fault-tolerant, exactly-once stateful stream processing in Structured Streaming

This fault-tolerant state management naturally incurs some processing overheads. To keep these overheads bounded within acceptable limits, the size of the state data should not grow indefinitely. However, with sliding windows, the number of windows/groups will grow indefinitely, and so can the size of state (proportional to the number of groups). To bound the state size, we have to be able to drop old aggregates that are not going to be updated any more, for example seven day old averages. We achieve this using *watermarking*.

Watermarking to Limit State while Handling Late Data

As mentioned before, the arrival of late data can result in updates to older windows. This complicates the process of defining which old aggregates are not going to be updated and therefore can be dropped from the state store to limit the state size. In Apache Spark 2.1, we have introduced *watermarking* that enables automatic dropping of old state data.

Watermark is a moving threshold in event-time that trails behind the maximum event-time seen by the query in the processed data. The trailing gap defines how long we will wait for late data to arrive. By knowing the point at which no more data will arrive for a given group, we can limit the total amount of state that we need to maintain for a query. For example, suppose the configured maximum lateness is 10 minutes. That means the events that are up to 10 minutes late will be allowed to aggregate. And if the maximum observed event time is 12:33, then all the future events with event-time older than 12:23 will

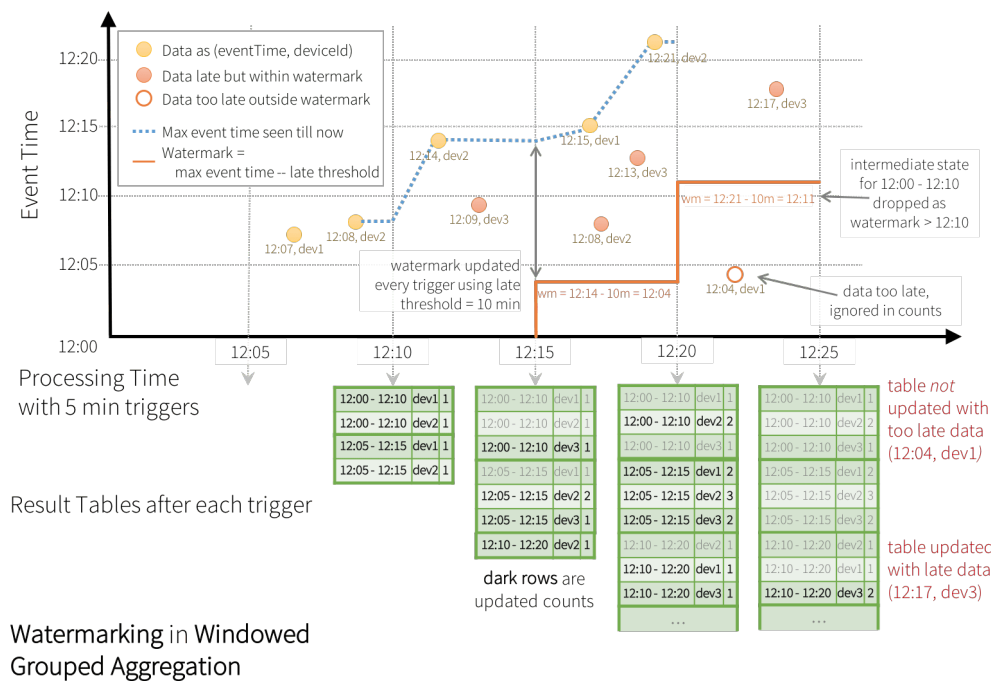
be considered as “too late” and dropped. Additionally, all the state for windows older than 12:23 will be cleared. You can set this parameter based on the requirements of your application — larger values of this parameter allows data to arrive later but at the cost of increased state size, that is, memory usage and vice versa.

Here is the earlier example but with watermarking.

```
windowedCountsDF = \  
  eventsDF \  
    .withWatermark("eventTime", "10 minutes") \  
    .groupBy(  
      "deviceId",  
      window("eventTime", "10 minutes", "5 minutes")) \  
    .count()
```

When this query is executed, Spark SQL will automatically keep track of the maximum observed value of the eventTime column, update the watermark and clear old state. This is illustrated below.





Note the two events that arrive between the processing-times 12:20 and 12:25. The watermark is used to differentiate between the late and the “too-late” events and treat them accordingly.

Conclusion

In short, I covered Structured Streaming’s windowing strategy to handle key streaming aggregations: windows over event-time and late and out-of-order data. Using this windowing strategy allows Structured Streaming engine to implement watermarking, in which

late data can be discarded. As a result of this design, we can manage the size of the state-store.

In the upcoming version of Apache Spark 2.2, we have added more advanced stateful stream processing operations to streaming DataFrames/Datasets. Stay tuned to this blog series for more information. If you want to learn more about Structured Streaming, read our previous posts in the series.

- [Structured Streaming In Apache Spark](#)
- [Real-time Streaming ETL with Structured Streaming in Apache Spark 2.1](#)
- [Working with Complex Data Formats with Structured Streaming in Apache Spark 2.1](#)
- [Processing Data in Apache Kafka with Structured Streaming in Apache Spark 2.2](#)

To try Structured Streaming in Apache Spark 2.0, [try Databricks today](#).



Part 6:
Vital Steps to Ensure
Production Readiness

Taking Apache Spark's Structured Streaming to Production

Part 5 of Scalable Data @ Databricks

February 23, 2017 | by Bill Chambers and Michael Armbrust

At Databricks, we've migrated our production pipelines to Structured Streaming over the past several months and wanted to share our out-of-the-box deployment model to allow our customers to rapidly build production pipelines in Databricks.

A production application requires monitoring, alerting, and an automatic (cloud native) approach to failure recovery. This post will not just walk you through the APIs available for tackling these challenges but will also show you how Databricks makes running Structured Streaming in production simple.

Metrics and Monitoring

[Structured Streaming in Apache Spark](#) provides a simple programmatic API to get information about a stream that is currently executing. There are two key commands that you can run on a currently active stream in order to get relevant information about the

query execution in progress: a command to get the current status of the query and a command to get *recentProgress* of the query.


Status

The first question you might ask is, "what processing is my stream performing right now?" The status maintains information about the current state of the stream, and is accessible through the object that was returned when you started the query. For example, you might have a simple counts stream that provides counts of IOT devices defined by the following query.

```
query = streamingCountsDF \  
  .writeStream \  
  .format("memory") \  
  .queryName("counts") \  
  .outputMode("complete") \  
  .start()
```

Running `query.status` will return the current status of the stream. This gives us details about what is happening at that point in time in the stream.

```
{  
  "message" : "Getting offsets from FileStreamSource[dbfs:/  
databricks-datasets/structured-streaming/events]",  
  "isDataAvailable" : true,  
  "isTriggerActive" : true  
}
```

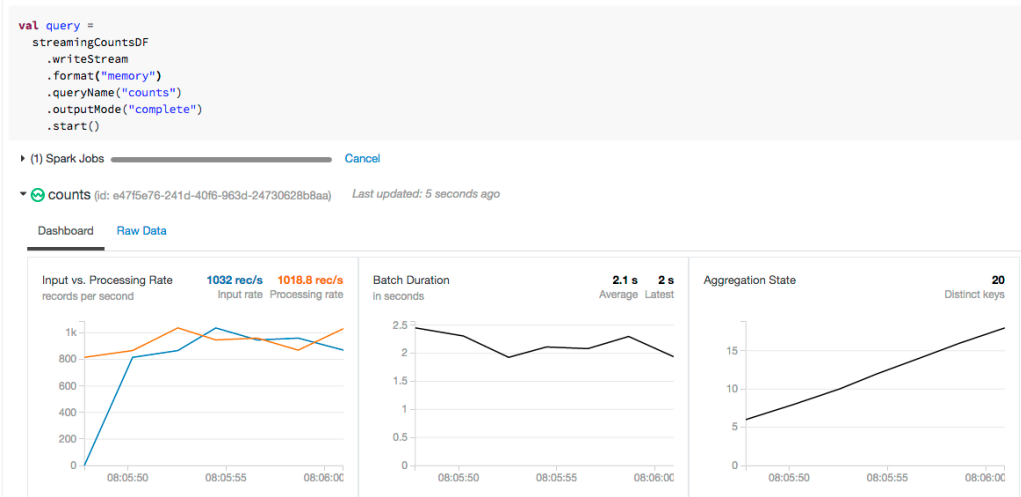
Databricks notebooks give you a simple way to see that status of any streaming query. Simply hover over the green streaming icon  available in a streaming query. You'll get the same information, making it much more convenient to quickly understand the state of your stream.



Recent Progress

While the query status is certainly important, equally important is an ability to view query's historical progress. Progress metadata will allow us to answer questions like "At what rate am I processing tuples?" or "How fast are tuples arriving from the source?"

By running `stream.recentProgress` you'll get access to some more time-based information like the processing rate and batch durations. However, a picture is worth a thousand JSON blobs, so at Databricks, we created visualizations in order to facilitate rapid analysis of the recent progress of the stream.



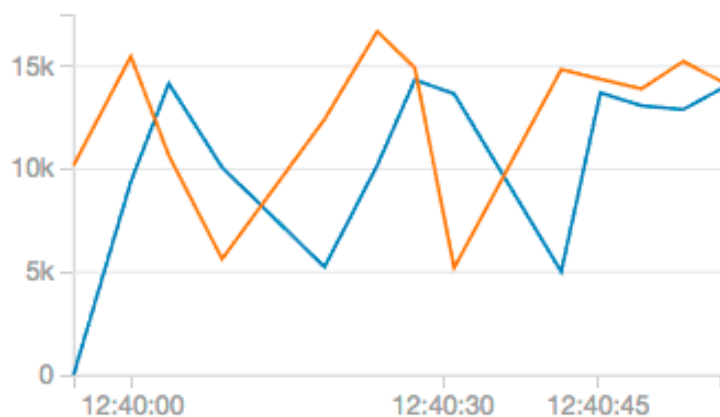
Let's explore why we chose to display these metrics and why they're important for you to understand.

Input Rate and Processing Rate

The input rate specifies how much data is flowing into Structured Streaming from a system like Kafka or Kinesis. The processing rate is how quickly we were able to analyze that data. In the ideal case, these should vary consistently together; however, they will vary according to how much input data exists when processing starts. If the input rate far outpaces the processing rate, our streams will fall behind, and we will have to scale the cluster up to a larger size to handle the greater load.

Input vs. Processing Rate
records per second

12.9k rec/s 14.4k rec/s
Input rate Processing rate

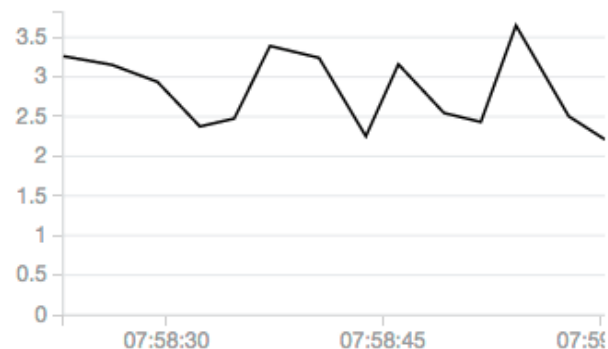


Batch Duration

Nearly all streaming systems utilize batching to operate at any reasonable throughput (some have an option of high latency in exchange for lower throughput). Structured Streaming achieves both. As it operates on the data, you will likely see this oscillate as Structured Streaming processes varying numbers of events over time. On this single core cluster on Community Edition, we can see that our batch duration is oscillating consistently around three seconds. Larger clusters will naturally have much faster processing rates as well as much shorter batch durations.

Batch Duration
in seconds

2.8 s 2.2 s
Average Latest



Production Alerting on Streaming Jobs

Metrics and Monitoring are all well and good, but in order to react quickly to any issues that arise without having to babysit your streaming jobs all day, you're going to need a robust alerting story. Databricks makes alerting easy by allowing you to run your Streaming jobs as production pipelines.

For instance, let's define a Databricks jobs with the following specifications:

Production Streaming ETL



Job ID: 212089

Task: Notebook at /databricks/streaming.logs - [Edit](#) / [Remove](#)

◦ Dependent Libraries: [Add](#)

Cluster: Driver: r3.xlarge, Workers: r3.xlarge, 61 GB, On-Demand and Spot, fall back to On-Demand, Spark 2.1.0-db3 (Scala 2.11) [Edit](#)

Schedule: None [Edit](#)

Advanced ▾

Alerts: [Edit](#)

◦ On error: critical-alerts@pct-databricks.pagerduty.com

Maximum Concurrent Runs: 1 [Edit](#)

Timeout: None [Edit](#)

Retries: Limit 30x [Edit](#) / [Remove](#)

Notice how we set an email address to trigger an alert in PagerDuty. This will trigger a product alert (or to the level that you specify) when the job fails.

Automated Failure Recovery

While alerting is convenient, having to force a human to respond to an outage is inconvenient at best and impossible at worst. In order to truly productionize Structured Streaming, you're going to want to be able to recover automatically to failures as quickly as you can, while ensuring data consistency and no data loss. Databricks makes this seamless: simply set the number of retries before a *unrecoverable failure* and Databricks will try to recover the streaming job automatically for you. On each failure, you can trigger a notification as a production outage.

You get the best of both worlds. The system will attempt to self-heal while keeping employees and developers informed of the status.

Updating Your Application

There are two circumstances that you need to reason about when you are updating your streaming application. For the most part, if you're not changing significant business logic (like the output schema) you can simply restart the streaming job using the same checkpoint directory. The new updated streaming application will pick up where it left off and continue functioning.

However, if you're changing stateful operations (like aggregations or the output schema), the update is a bit more involved. You'll have to start an entirely new stream with a new checkpoint directory. Luckily,

it's easy to start up another stream in Databricks in order to run both in parallel while you transition to the new stream.

Advanced Alerting and Monitoring

There are several other advanced monitoring techniques that Databricks supports as well. For example, you can output notifications using a system like [Datadog](#), [Apache Kafka](#), or [Coda Hale Metrics](#). These advanced techniques can be used to implement external monitoring and alerting systems.

Below is an example of how you can create a `StreamingQueryListener` that will forward all query progress information to Kafka.

```
class KafkaMetrics(servers: String) extends StreamingQueryListener {
  val kafkaProperties = new Properties()
  kafkaProperties.put("bootstrap.servers", servers)
  kafkaProperties.put("key.serializer",
    "kafkashaded.org.apache.kafka.common.serialization.StringSerializer")
  kafkaProperties.put("value.serializer",
    "kafkashaded.org.apache.kafka.common.serialization.StringSerializer")

  val producer = new KafkaProducer[String, String](kafkaProperties)

  def onQueryProgress(event: org.apache.spark.sql.streaming.StreamingQueryListener.QueryProgressEvent): Unit = {
    producer.send(new ProducerRecord("streaming-metrics", event.progress.json))
  }
}
```

```
def onQueryStarted(event: org.apache.spark.sql.streaming.StreamingQueryListener.QueryStartedEvent): Unit = {}
def onQueryTerminated(event: org.apache.spark.sql.streaming.StreamingQueryListener.QueryTerminatedEvent): Unit = {}
}
```

Conclusion

In this post, we showed how simple it is to take Structured Streaming from prototype to production using Databricks. To read more about other aspects of Structured Streaming, read our series of blogs:

- [Structured Streaming In Apache Spark](#)
- [Real-time Streaming ETL with Structured Streaming in Apache Spark 2.1](#)
- [Working with Complex Data Formats with Structured Streaming in Apache Spark 2.1](#)
- [Processing Data in Apache Kafka with Structured Streaming in Apache Spark 2.2](#)
- [Event-time Aggregation and Watermarking in Apache Spark's Structured Streaming](#)

You can learn more about using streaming from the [Databricks Documentation](#) or sign up to [start a free trial today](#).



Part 7:

Better Cost Management
through APIs

Running Streaming Jobs Once a Day For 10x Cost Savings

Part 6 of Scalable Data @ Databricks

May 22, 2017 | by Burak Yavuz and Tyson Condie

Traditionally, when people think about streaming, terms such as “real-time,” “24/7,” or “always on” come to mind. You may have cases where data only arrives at fixed intervals. That is, data appears every hour or once a day. For these use cases, it is still beneficial to perform incremental processing on this data. However, it would be wasteful to keep a cluster up and running 24/7 just to perform a short amount of processing once a day.

Fortunately, by using the new Run Once trigger feature added to Structured Streaming in Apache Spark 2.2, you will get all the benefits of the Catalyst Optimizer incrementalizing your workload, and the cost savings of not having an idle cluster lying around. In this post, we will examine how to employ triggers to accomplish both.

Triggers in Structured Streaming

In Structured Streaming, triggers are used to specify how often a streaming query should produce results. Once a trigger fires, Spark checks to see if there is new data available. If there is new data, then the query is executed incrementally on whatever has arrived since the

last trigger. If there is no new data, then the stream sleeps until the next trigger fires.

The default behavior of Structured Streaming is to run with the lowest latency possible, so triggers fire as soon as the previous trigger finishes. For use cases with lower latency requirements, Structured Streaming supports a ProcessingTime trigger which will fire every user-provided interval, for example every minute.

While this is great, it still requires the cluster to remain running 24/7. In contrast, a RunOnce trigger will fire only once and then will stop the query. As we’ll see below, this lets you effectively utilize an external scheduling mechanism such as Databricks Jobs.

Triggers are specified when you start your streams.

PYTHON

```
# Load your Streaming DataFrame
sdf = spark.readStream.load(path="/in/path", format="json",
schema=my_schema)
# Perform transformations and then write...
sdf.writeStream.trigger(once=True).start(path="/out/path",
format="parquet")
```

SCALA

```
import org.apache.spark.sql.streaming.Trigger

// Load your Streaming DataFrame
val sdf =
  spark.readStream.format("json").schema(my_schema).load("/in/
  path")
// Perform transformations and then write...
sdf.writeStream.trigger(Trigger.Once).format("parquet").start(
"/out/path")=
```

Why Streaming and RunOnce is Better than Batch

You may ask, how is this different than simply running a batch job? Let's go over the benefits of running Structured Streaming over a batch job.

Bookkeeping

When you're running a batch job that performs incremental updates, you generally have to deal with figuring out what data is new, what you should process, and what you should not. Structured Streaming already does all this for you. In writing general streaming applications, you should only care about the business logic, and not the low-level bookkeeping.

Table Level Atomicity

The most important feature of a big data processing engine is how it can tolerate faults and failures. The ETL jobs may (in practice, often will) fail. If your job fails, then you need to ensure that the output of your job should be cleaned up, otherwise you will end up with duplicate or garbage data after the next successful run of your job.

While using Structured Streaming to write out a file-based table, Structured Streaming commits all files created by the job to a log after each successful trigger. When Spark reads back the table, it uses this log to figure out which files are valid. This ensures that garbage introduced by failures are not consumed by downstream applications.

Stateful Operations Across Runs

If your data pipeline has the possibility of generating duplicate records, but you would like exactly once semantics, how do you achieve that with a batch workload? With Structured Streaming, it's as easy as setting a watermark and using `dropDuplicates()`. By configuring the watermark long enough to encompass several runs of your streaming job, you will make sure that you don't get duplicate data across runs.

Cost Savings

Running a 24/7 streaming job is a costly ordeal. You may have use cases where latency of hours is acceptable, or data comes in hourly or daily. To get all the benefits of Structured Streaming described above,

you may think you need to keep a cluster up and running all the time. But now, with the “execute once” trigger, you don’t need to!

At Databricks, we had a [two stage data pipeline](#), consisting of one incremental job that would make the latest data available, and one job at the end of the day that processed the whole day’s worth of data, performed de-duplication, and overwrote the output of the incremental job. The second job would use considerably larger resources than the first job (4x), and would run much longer as well (3x). We were able to get rid of the second job in many of our pipelines that amounted to a 10x total cost savings. We were also able to clean up a lot of code in our codebase with the new execute once trigger. Those are cost savings that makes both financial and engineering managers happy!

Scheduling Runs with Databricks

[Databricks’ Jobs scheduler](#) allows users to schedule production jobs with a few simple clicks. Jobs scheduler is ideal for scheduling Structured Streaming jobs that run with the execute once trigger.

The screenshot shows the Databricks Jobs scheduler interface for a job named "Log ETL: Prod - Delta Update". The interface includes a sidebar with navigation options like Home, Workspace, Recent, Tables, Clusters, Jobs, and Search. The main content area displays job details such as Job ID (133), Task (Notebook at /Users/eti@databricks.com/log-etl/...), Cluster (Driver: c3.4xlarge, Workers: c3.4xlarge, 330 GB, Spot), and Schedule (Every 2 hours (US/Pacific)). It also shows active runs and a table of completed runs in the past 60 days.

Run	Start Time	Launched	Duration	Status
Run 52498	2017-04-04 10:00:00 PDT	By scheduler	7m 41s	Succeeded
Run 52497	2017-04-04 08:00:00 PDT	By scheduler	12m 37s	Succeeded

At Databricks, we use the Jobs scheduler to run all of our production jobs. As engineers, we ensure that the business logic within our ETL job is well tested. We upload our code to Databricks as a library, and we set up notebooks to set the configurations for the ETL job such as the input file directory. The rest is up to Databricks to manage clusters, schedule and execute the jobs, and Structured Streaming to figure out which files are new, and process incoming data. The end result is an end-to-end — from data origin to data warehouse, not only within Spark — exactly once data pipeline. Check out [our documentation](#) on how to best run Structured Streaming with Jobs.

Summary

In this blog post we introduced the new “execute once” trigger for Structured Streaming. While the execute once trigger resembles running a batch job, we discussed all the benefits it has over the batch job approach, specifically:

- Managing all the bookkeeping of what data to process
- Providing table level atomicity for ETL jobs to a file store
- Ensuring stateful operations across runs of the job, which allow for easy de-duplication

In addition to all these benefits over batch processing, you also get the cost savings of not having an idle 24/7 cluster up and running for an irregular streaming job. The best of both worlds for batch and streaming processing are now under your fingertips.

Try Structured Streaming today in Databricks by [signing up for a 14-day free trial](#).

Other parts of this blog series explain other benefits as well:

- [Real-time Streaming ETL with Structured Streaming in Apache Spark 2.1](#)
- [Working with Complex Data Formats with Structured Streaming in Apache Spark 2.1](#)

- [Processing Data in Apache Kafka with Structured Streaming in Apache Spark 2.2](#)
- [Event-time Aggregation and Watermarking in Apache Spark’s Structured Streaming](#)
- [Taking Apache Spark’s Structured Structured Streaming to Production](#)
- [Running Streaming Jobs Once a Day For 10x Cost Savings](#)



Part 8:

Customized and Arbitrary
Stateful Processing

Arbitrary Stateful Processing in Apache Spark's Structured Streaming

Part 7 of Scalable Data @ Databricks

October 17, 2017 | by Bill Chambers and Jules Damji

Introduction

Most data streams, though continuous in flow, have discrete events within streams, each marked by a timestamp when an event transpired. As a consequence, this idea of “event-time” is central to how Structured Streaming APIs are fashioned for event-time processing—and the functionality they offer to process these discrete events.

[Event-time basics and event-time processing](#) are adequately covered in [Structured Streaming documentation](#) and our [anthology of technical assets on Structure Streaming](#). So for brevity, we won't cover them here. Built on the concepts developed (and tested at scale) in event-time processing, such as sliding windows, tumbling windows, and watermarking, this blog will focus on two topics:

1. How to handle duplicates in your event streams
2. How to handle arbitrary or custom stateful processing

Dropping Duplicates

No streaming events are free of duplicate entries. Dropping duplicate entries in record-at-a-time systems is imperative—and often a cumbersome operation for a couple of reasons. First, you'll have to process small or large batches of records at time to discard them. Second, some events, because of network high latencies, may arrive out-of-order or late, which may force you to reiterate or repeat the process. How do you account for that?

Structured Streaming, which ensures exactly once-semantics, can drop duplicate messages as they come in based on arbitrary keys. To deduplicate data, Spark will maintain a number of user-specified keys and ensure that duplicates, when encountered, are discarded.

Just as other stateful processing APIs in Structured Streaming are bounded by declaring [watermarking for late data](#) semantics, so is dropping duplicates. Without watermarking, the maintained state can grow infinitely over the course of your stream.

The API to instruct Structured Streaming to drop duplicates is as simple as all other APIs we have shown so far in our blogs and documentation. Using the API, you can declare arbitrarily columns on which to drop duplicates—for example, `user_id` and `timestamp`. An entry with same timestamp and `user_id` is marked as duplicate and dropped, but the same entry with two different timestamps is not.

Let's see an example how we can use the simple API to drop duplicates.

PYTHON

```
from pyspark.sql.functions import expr

withEventTime\
    .withWatermark("event_time", "5 seconds")\
    .dropDuplicates(["User", "event_time"])\
    .groupBy("User")\
    .count()\
    .writeStream\
    .queryName("pydeduplicated")\
    .format("memory")\
    .outputMode("complete")\
    .start()
```

SCALA

```
import org.apache.spark.sql.functions.expr

withEventTime
    .withWatermark("event_time", "5 seconds")
    .dropDuplicates("User", "event_time")
    .groupBy("User")
    .count()
    .writeStream
    .queryName("deduplicated")
    .format("memory")
    .outputMode("complete")
    .start()
```

Over the course of the query, if you were to issue a SQL query, you will get an accurate results, with all duplicates dropped.

```
SELECT * FROM deduplicated
+-----+-----+
|User|count|
+-----+-----+
| a | 8085 |
| b | 9123 |
| c | 7715 |
| g | 9167 |
| h | 7733 |
| e | 9891 |
| f | 9206 |
| d | 8124 |
| i | 9255 |
+-----+-----+
```

Next, we will expand on how to implement a customized stateful processing using two Structured Streaming APIs.

Working with Arbitrary or Custom Stateful Processing

Not all event-time based processing is equal or as simple as aggregating a specific data column within an event. Others events are more complex; they require processing by rows of events ascribed to a group; and they only make sense when processed in their entirety by

emitting either a single result or multiple rows of results, depending on your use cases.

Consider these use-cases where arbitrary or customized stateful processing become imperative:

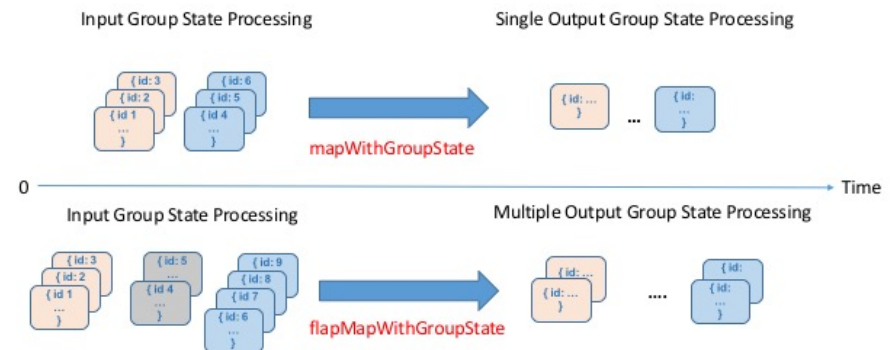
1. We want to emit an alert based on a group or type of events if we observe that they exceed a threshold over time
2. We want to maintain user sessions, over definite or indefinite time and persist those sessions for post analysis.

All of the above scenarios require customized processing. Structured Streaming APIs offer a set of APIs to handle these cases:

`mapGroupsWithState` and `flatMapGroupsWithState`.

`mapGroupsWithState` can operate on groups and output only a single result row for each group, whereas `flatMapGroupsWithState` can emit a single row or multiple rows of results per group.

Arbitrary Stateful Processing in Structured Streaming



Timeouts and State

One thing to note is that because we manage the state of the group based on user-defined concepts, as expressed above for the use-cases, the semantics of watermark (expiring or discarding an event) may not always apply here. Instead, we have to specify an appropriate timeout ourselves. Timeout dictates how long we should wait before timing out some intermediate state.

Timeouts can either be based on processing time (`GroupStateTimeout.ProcessingTimeTimeout`) or event time (`GroupStateTimeout.EventTimeTimeout`). When using timeouts, you can check for timeout first before processing the values by checking the flag `state.hasTimedOut`.

To set processing timeout, use `GroupState.setTimeoutDuration(...)` method. That means the timeout guarantee will occur under the following conditions:

- Timeout will never occur before the clock has advanced X ms specified in the method
- Timeout will eventually occur when there is a trigger in the query, after X ms

To set event time timeout, use `GroupState.setTimeoutTimestamp(...)`. Only for timeouts based on event time must you specify watermark. As such all events in the group older than watermark will be filtered out, and the timeout will occur when the watermark has advanced beyond the set timestamp.

When timeouts occur, your function supplied in the streaming query will be invoked with arguments: the key by which you keep the state; an iterator rows of input, and an old state. The example with `mapGroupsWithState` below defines a number of functional classes and objects used.

Example with `mapGroupsWithState`

Let's take a simple example where we want to find out when (timestamp) a user performed his or her first and last activity in a given dataset in a stream. In this case, we will group on (or map on) on a user key and activity key combination.

But first, `mapGroupsWithState` requires a number of functional classes and objects:

1. Three class definitions: an input definition, a state definition, and optionally an output definition.
2. An update function based on a key, an iterator of events, and a previous state.
3. A timeout parameter as described above.

So let's define our input, output, and state data structure definitions.

```
case class InputRow(user:String, timestamp:java.sql.Timestamp, activity:String)
case class UserState(user:String,
  var activity:String,
  var start:java.sql.Timestamp,
  var end:java.sql.Timestamp)
```

Based on a given input row, we define our update function

```
def updateUserStateWithEvent(state:UserState,
input:InputRow):UserState = {
// no timestamp, just ignore it
if (Option(input.timestamp).isEmpty) {
    return state
}
//does the activity match for the input row
if (state.activity == input.activity) {
    if (input.timestamp.after(state.end)) {
        state.end = input.timestamp
    }
    if (input.timestamp.before(state.start)) {
        state.start = input.timestamp
    }
} else {
//some other activity
    if (input.timestamp.after(state.end)) {
        state.start = input.timestamp
        state.end = input.timestamp
        state.activity = input.activity
    }
}
//return the updated state
state
}
```

And finally, we write our function that defines the way state is updated based on an epoch of rows.

```
import org.apache.spark.sql.streaming.{GroupStateTimeout,
OutputMode, GroupState}

def updateAcrossEvents(user:String,
    inputs: Iterator[InputRow],
    oldState: GroupState[UserState]):UserState = {
    var state:UserState = if (oldState.exists) oldState.get
    else UserState(user,
        "",
        new java.sql.Timestamp(62841600000000L),
        new java.sql.Timestamp(6284160L)
    )
    // we simply specify an old date that we can compare against
    and
    // immediately update based on the values in our data

    for (input <- inputs) {
        state = updateUserStateWithEvent(state, input)
        oldState.update(state)
    }
    state
}
```

With these pieces in place, we can now use them in our query. As discussed above, we have to specify our timeout so that the method can timeout a given group's state and we can control what should be done with the state when no update is received after a timeout. For this illustration, we will maintain state indefinitely.

```
import org.apache.spark.sql.streaming.GroupStateTimeout

withEventTime
  .selectExpr("User as user", "cast(Creation_Time/1000000000
as timestamp) as timestamp", "gt as activity")
  .as[InputRow]
  // group the state by user key
  .groupByKey(_.user)
  .mapGroupsWithState(GroupStateTimeout.NoTimeout)
(updateAcrossEvents)
  .writeStream
  .queryName("events_per_window")
  .format("memory")
  .outputMode("update")
  .start()
```

We can now query our results in the stream:

```
SELECT * FROM events_per_window order by user, start
```

And our sample result that shows user activity for the first and last time stamp:

```
+-----+-----+-----+-----+
|user|activity|          start|          end|
+-----+-----+-----+-----+
| a |  bike | 2015-02-23 13:30:... | 2015-02-23 14:06:... |
| a |  bike | 2015-02-23 13:30:... | 2015-02-23 14:06:... |
...
| b |  bike | 2015-02-24 14:01:... | 2015-02-24 14:38:... |
| b |  bike | 2015-02-24 14:01:... | 2015-02-24 14:38:... |
| c |  bike | 2015-02-23 12:40:... | 2015-02-23 13:15:... |
...
| d |  bike | 2015-02-24 13:07:... | 2015-02-24 13:42:... |
+-----+-----+-----+-----+
```

What's Next

In this blog, we expanded on two additional functionalities and APIs for advanced streaming analytics. The first allows removing duplicates bounded by a watermark. With the second, you can implement customized stateful aggregations, beyond [event-time basics](#) and [event-time processing](#).

Through an example using `mapGroupsWithState` APIs, we demonstrated how you can implement your customized stateful aggregation for events whose processing semantics can be defined not only by timeout but also by user semantics and business logic.

Our next blog in this series, we will explore advanced aspects of `flatMapGroupsWithState` use cases, as was discussed at [Spark Summit Europe](#), in Dublin, in a deep dive session on Structured Streaming: [Session 1](#). [Session 2](#).

Read More

Over the course of Structured Streaming development and release since Apache Spark 2.0, we have compiled a comprehensive compendium of technical assets, including our Structured Series blogs. You can read the relevant assets here:

- [Anthology of Technical Assets on Apache Spark's Structured Streaming](#)

Try Apache Spark's Structured Streaming latest APIs on [Databricks' Unified Analytics Platform](#).



Conclusion

Our mission at Databricks is to dramatically simplify big data and AI. Our [Unified Analytics Platform](#) enables this through the unification of data science, engineering, and the business — accelerating innovation that delivers transformative business outcomes. We hope this eBook will provide you with the insights and tools to help you solve your streaming problems. If you enjoyed the technical content in this eBook, visit the [Databricks Blog](#) for more technical tips, best practices, and case studies from the Spark experts at Databricks.

To try out Databricks for yourself, sign-up for a 14-day free trial today!

To try Databricks yourself, start your [free trial](#) today!