databricks

Introduction to Streaming

LECTURE

Introduction to Structured Streaming

Let’s talk about Structured Streaming. Now that we've talked about bounded vs. unbounded datasets and batch vs. streaming processing, we will get into a specific type of engine to process streaming data called Apache Spark Structured Streaming.

In this lesson we will:

- Describe main features of Apache Structured Streaming.
- Explain how Apache Structured Streaming solves common problems with streaming data.
- Discuss the benefits of Apache Spark Structured Streaming.
- Compare and contrast Structured Streaming with other stream processing engines.

What is Structured Streaming

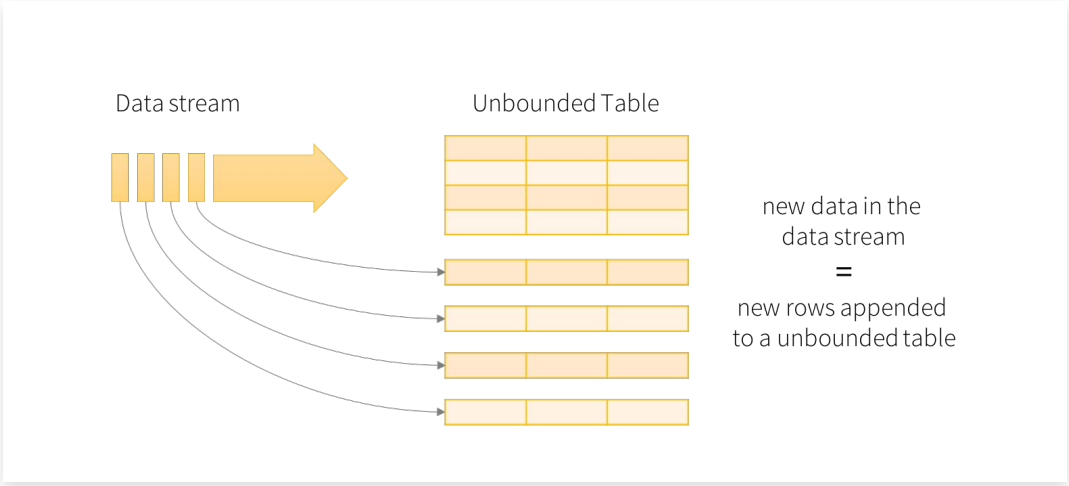
Apache Spark structured streaming basics

- A scalable, fault-tolerant **stream processing framework** built on Spark SQL engine.
- Uses **existing structured APIs** (DataFrames, SQL Engine) and provides similar API as batch processing API.
- Includes **stream specific features**; end-to-end, exactly-once processing, fault-tolerance etc.

What is structured streaming? Apache Spark structured streaming is a scalable fault tolerance stream processing framework that's built on the Spark SQL engine.
It uses existing structured APIs such as DataFrames and the SQL engine to provide an API similar to the batch processing API.

During the demos and labs, we'll see that a lot of code from Apache Spark will look similar in batch and streaming.
The structured streaming API also provides some stream specific features, such as end-to-end, exactly once processing and fault tolerance.

Streams as Unbounded Tables



Batch and Stream programming require using very different constructs. So how do Apache Spark and Databricks unify these into one API?

The underlying principle is clever, yet simple. We all understand how queries run against tables. In batch mode, these tables are bounded and the query runs once against the tables in question.

In structured streaming, we model the data stream as an unbounded table - to which data is being appended on each trigger.
The query as defined by the user is then “applied” or “rerun” against the updated dataset.

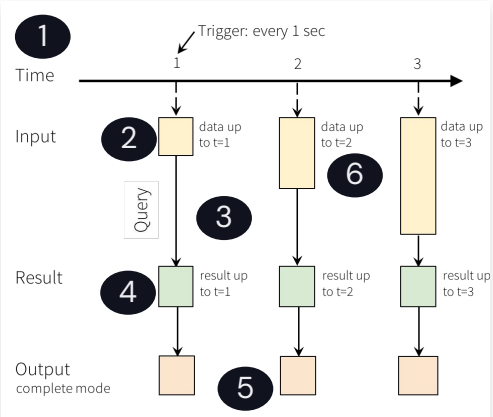
Using dataframes as the construct for bounded and unbounded datasets makes it possible to reuse the same instruction set for all transformations.

See the next slide for a visualization of this.

How Structured Streaming Works

Execution mode

1. Timeline
2. An **input table** is defined by configuring a streaming read against **source**.
3. A **query** is defined against the input table.
4. This logical query on the input table generates the **results table**.
5. The **output** of a streaming pipeline will persist updates to the results table by writing to an external **sink**.
6. New rows are appended to the input table for each **trigger interval**.



Programming Model for Structured Streaming

Let's go over how Structured Streaming Works:

1. First, we must consider a timeline representing when the data is ingested. In this scenario, a trigger interval is configured to run every second.
2. Then we have an Input Table in yellow, which is defined by configuring a streaming read against a Source, which could be a file or another table.
3. Then a query is defined against this input table to append new data or rows
4. This will provide a new “Result Table” in green with the new appended data.
5. Then the “Result table” will persist, writing to an external Sink or a table, and in this scenario, it is configured using an Output mode of “Complete” that will write all rows every time.
6. Then, the cycle continues for each execution provided by the trigger interval. Here, we can see that the Input Table has grown, meaning it is unbounded, and new data will be continuously added.

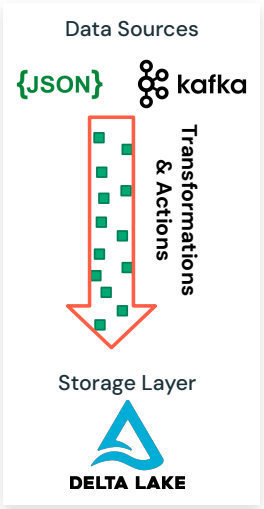
—--

Now you might wonder—if the table keeps growing, won’t the system get heavier and slower over time? How does this process run continuously for hours, days, and weeks? We’ll explore the internals in the next module.

Anatomy of a Streaming Query

Structured streaming core concepts

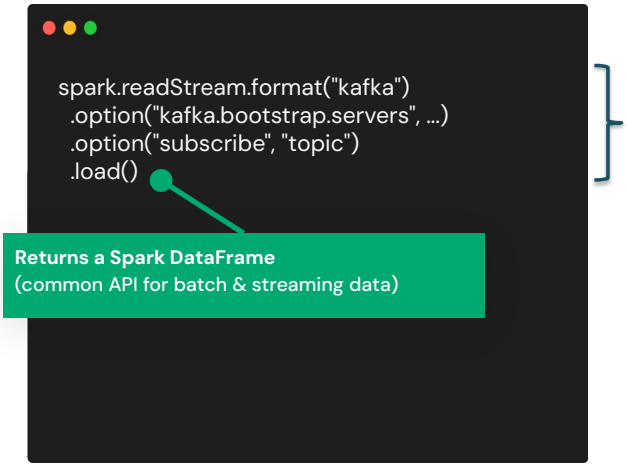
- Example:
  - Read JSON data from Kafka
  - Parse nested JSON
  - Store in structured Delta Lake table
- Core concepts:
  - Input sources
  - Transformations
  - Sinks
  - Triggers



A high-level example of this is reading JSON data from Kafka. Then we parse nested JSON.
Then we store in structured Delta tables as our sink. So the core concepts here, we have the input source, which is our JSON data from Kafka.
We have a sink, which is our Delta table. We have some transformation in action, which is the parsing of the nested JSON data and triggers which move things continuously up in this one every second.

Anatomy of a Streaming Query

Structured streaming core concepts



Source:

- Specify where to read data from
- OS Spark supports Kafka and file sources
- Databricks runtimes include connector libraries supporting Delta, Event Hubs, and Kinesis

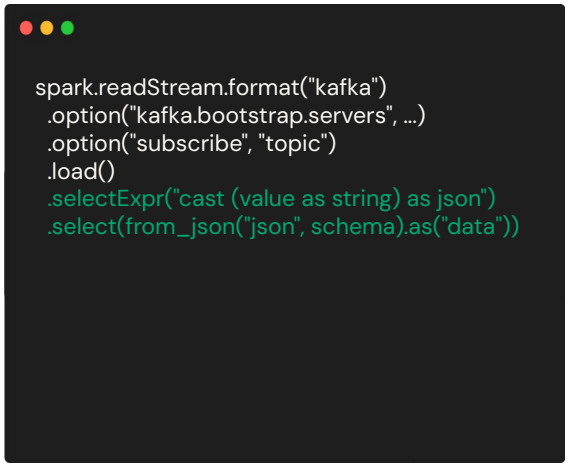
```
spark.readStream.format(<source>)
.option(<>,<>)...
.load()
```


Let's discuss what the code looks like in a streaming query.

Streaming queries start with an instantiated Spark object, followed by a call to read the data as a stream and a format specification.
In this example, we are reading from Kafka, and then we're adding the option to run from kafka.bootstrap servers.
We're also adding the option to subscribe to a topic.
This can include multiple sources of different types. Finally, we call the load function to return a Spark DataFrame.

Anatomy of a Streaming Query

Structured streaming core concepts



Transformations:

- 100s of built-in, optimized SQL functions like from_json
- In this example, cast bytes from Kafka records to a string, parse it as JSON, and generate nested columns

If we're running a transformation on a streaming query, we might add in a couple lines like this.
So here we'll cast bytes from Kafka records to a string, parse it as a JSON, and generate nested columns.
We can also use hundreds of built-in optimized SQL functions from JSON as well as user-defined functions like lambdas, function literals with map, or flat map.

Anatomy of a Streaming Query

Structured streaming core concepts

```
spark.readStream.format("kafka")
  .option("kafka.bootstrap.servers", ...)
  .option("subscribe", "topic")
  .load()
  .selectExpr("cast (value as string) as json")
  .select(from_json("json", schema).as("data"))
  .writeStream
  .format("delta")
  .option("path", "/deltaTable/")
```

Sink: Write transformed output to external storage systems

Databricks runtimes include connector library supporting Delta

OS Spark supports:

- Files and Kafka for production
- Console and memory for development and debugging
- foreachBatch to execute arbitrary code with the output data

After that, we want to sink the data. In the previous diagram, we saw that we were sinking the data to some output table.
Likewise, in this example, we want to write transformed outputs to external storage systems.
We have built-in support for files or Kafka, so you can immediately sink it out back to Kafka, or you could use foreach to execute arbitrary code with the output data.

Some sinks are transactional and exactly once, like files, while others can write out the stream to a Delta format table, the path to which we specify.

Anatomy of a Streaming Query

Structured streaming core concepts

```
spark.readStream.format("kafka")
  .option("kafka.bootstrap.servers", ...)
  .option("subscribe", "topic")
  .load()
  .selectExpr("cast (value as string) as json")
  .select(from_json("json", schema).as("data"))
  .writeStream
  .format("delta")
  .option("path", "/deltaTable/")
  .trigger("1 minute")
  .option("checkpointLocation", "...")
  .start()
```

- **Checkpoint location:** For tracking the progress of the query
- **Output Mode:** Defines how the data is written to the sink; Equivalent to "save" mode on static DataFrames
- **Trigger:** Defines how frequently the input table is checked for new data; Each time a trigger fires, Sparks check for new data and updates the results

Let's talk about triggers. Triggers define how frequently the input table is checked for new data.
So each time a trigger fires, Spark checks for new data and updates the result.
In our diagram from earlier here, we have a trigger that was happening every second.
So every second is pulling our input source and looking for new data.

Then we have a checkpoint location. This tracks the progress of the query.
So here we, in this example I saw a second ago from the output, the timeline, it was triggering every second.
And this trigger, our timing is one minute. So we're triggering out and looking for pulling for new data every minute.

DataFrame as a Unifying API

Batch

```
spark
  .read
  .table("source_name")
  .withColumn("...")
  .select(...)
  .write
[ .outputMode(...) ]
  .saveAsTable("sink_table")
```

Streaming

```
spark
  .readStream
[ .withWatermark(...) ]
  .table("source_name")
  .withColumn("...")
  .select(...)
  .writeStream
[ .trigger(...) ]
[ .queryName(...) ]
  .option("checkpointLocation", ...)
  .toTable("sink_table")
```

As you can see in the presented example, in the vast majority of cases, switching an existing query from batch to streaming can be as simple as replacing the ‘read’ and ‘write’ attributes to ‘readStream’ and ‘writeStream’ and adding a few necessary attributes.

The engine still performs along the classic “Load > Transform > Write” paradigm - except that these actions happen repeatedly - on each trigger (the next slide covers this)

Anatomy of a Streaming Query

Structured streaming core concepts

Trigger Types:

Fixed interval micro batch	.trigger(processingTime = "2 minutes")	Micro-batch processing kicked off at the *user-specified interval*
Triggered One-time micro batch	.trigger(once=True)	DEPRECATED - Process all of the available data as a *single micro-batch* and then automatically stop the query
Triggered One-time micro batches	.trigger(availableNow=True)	Process all of the available data as *multiple micro-batches* and then automatically stop the query
Continuous Processing	.trigger(continuous= "2 seconds")	EXPERIMENTAL - Long-running tasks that *continuously* read, process, and write data as soon events are available, with checkpoints at the specified frequency
Default		Databricks: 500ms fixed interval OS Apache Spark: Process each microbatch as soon as the previous has been processed

There are several different ways to trigger a streaming query.

- The default for OS Apache Spark is to process each micro batch as soon as the previous one has been processed. We also have fixed interval micro batches. So processing time of two minutes, this kicks off at a user-specified interval.
- On Databricks, Structured Streaming defaults to fixed interval micro-batches of 500ms.
    - Databricks recommends you always specify a tailored trigger to minimize costs associated with checking if new data has arrived and processing undersized batches.

- We have triggered one-time micro batch, so we call the trigger once by setting once equal to true. This will process all the available data as a single micro-batch and then automatically stop the query.
- Then triggered one-time micro batches process all the available data as my multi microbiomes, and then automatically stuff the queries
- and then continuous processing. So these are long running tasks to continuously read, process and write data as soon as events are available.
    - Potentially add "with checkpoints at the specified frequency"

Anatomy of a Streaming Query

Structured Streaming Core Concepts

Output Modes:

Complete	• The entire updated Result Table is written to the sink. • The individual sink implementation decides how to handle writing the entire table.
Append	Only the new rows appended to the Result Table since the last trigger are written to the sink.
Update	Only new rows and the rows in the Result Table that were updated since the last trigger will be outputted to the sink.

Note: The output modes supported depends on the type of transformations and sinks used by the streaming query. Refer to the the Structured Streaming Programming Guide for details.

There are three modes to choose from when outputting the results of a structured streaming query: complete, append, and update.

- In C**omplete** mode, the entire updated result table is written out to the sink. Individual sink implementation decides how to handle writing to the entire table.
- **Append** is only the new rows get appended to the result table since the last trigger is already written out to the sink.
- **Update** mode only outputs to the sink the rows in the result table that were updated since the last trigger.

Note:
The supported output modes will depend on the type of transformations and sinks used by the streaming query - refer to the Structured Streaming Programming Guide for details.

Thank you for completing this lesson and continuing your journey to develop your skills with us.