Databricks Structured Streaming

# Introduction and Goals

**Welcome to Fundamentals of Structured Streaming.**

A common struggle that organizations face is how to accurately ingest and perform calculations on real-time data.  This data is also referred to as streaming data, and the challenges behind working with it lie in its real-time nature - because it is constantly arriving, mechanisms must be put into place to process and write to a data store.

In this course, you’ll learn about Structured Streaming, an Apache Spark API that helps data practitioners overcome the challenges of working with streaming data. We’ll cover fundamental concepts about batch and streaming data to help set the stage for our discussion on Structured Streaming. Then, we’ll discuss where Structured Streaming fits into an organization’s big data ecosystem. Finally, we’ll review real-world Structured Streaming business use cases.

By the end of this course, you will be able to:

1. Explain the benefits of Structured Streaming when working with streaming data.
2. Distinguish where Structured Streaming fits into an organization’s big data ecosystem.
3. Articulate examples of real-world business use cases for Structured Streaming.

# Fundamentals of Streaming Data

**Lesson Introduction**

In 2011, Marc Andreessen said, “software is eating the world.” This was later iterated on as “data is eating the world” by Dries Buytaert. Organizations that create data and efficiently know how to draw insight and create action from that data are thriving.

As the volume and velocity of data increase, businesses start to need ways to reduce the amount of time it takes to process data. In some use cases, businesses simply cannot wait for an analytics system that processes data at a later point in time.

In this lesson, we will start out by defining streaming data, a catch-all term for several concepts.

By the end of this lesson, you will be able to:

* Define streaming data and explain the difference between streaming and batch data.

When most people refer to streaming data, they likely refer to one concept within streaming data: an engine optimized to process unbounded data in a continuously streaming fashion (contrasted with an engine optimized to process data in a batch fashion). That sentence has three important ideas in it: an engine, unbounded data, and continuous data.

Let’s unpack these concepts to better understand streaming data:

* **An** **Engine:** Also called a “mechanism,” “technology,” or “application” in this course; all of these terms refer to computer software that is designed to meet the challenges of streaming data. Here, we’ll distinguish between streaming engines and batch engines.
* **Unbounded Datasets:** Bounded and unbounded datasets both refer to our input data source. We'll focus on unbounded datasets: an infinitely expanding dataset with no boundaries.
* **Data in Motion:** Data in motion is data which is not at rest. This refers to the timeline of data.

**Batch and Streaming Engines**

Batch processing refers to the processing of all past data after it has settled. Stream processing though refers to the processing of data as it is arriving.

A way to compare and contrast batch and streaming: think of counting votes in a nationwide election. We could count all votes at the end of the day after they have all arrived (batch), or alternatively, we could count votes as they’re coming in (streaming). Closely tied to this processing is the nature of our input data source: bounded or unbounded.

**Bounded and Unbounded Datasets**

Bounded datasets are of a known and finite size. A bound dataset has boundaries: a start-point and an endpoint.

Imagine you want to load a dataset into some program for analysis with the end goal of building a machine learning model to predict sales. More specifically, you want to load a dataset that consists of the total amount of money that your company made every day from 2020. This is a bounded dataset - there is a known start and endpoint, and it is finite. Usually, this sort of analysis would be done with a batch processing engine.

On the other hand, imagine you want to load a dataset into some program for analysis with the end goal of building a machine learning model to predict the next time a sensor on a conveyor belt in your factory is going to fail. We need real-time data because looking at all of the readings from last year only partially tells the story of whether this sensor is going to fail right now, tomorrow, or even the next day. This dataset is unbounded - we don’t know when the data needs to start and stop. It might be useful to think of the dataset as a table that is constantly being appended to.

An unbounded dataset is one in which events are continuously added to the dataset.

Most data in the real world arrives in this fashion. Since the data that your organization collects is likely to have no known endpoint, you can imagine most datasets you create as unbounded. The way that most batch processing engines ingest unbounded data is to wait for some time period (e.g. the end of your business day) and then load all the data up to that point and process it.

Going back to our sensor monitoring though - having data with this sort of delay is unacceptable. We don’t need to be told that a sensor failed yesterday and that our machine learning model accurately predicted that event. We want to predict this fact in the middle of the day, right before the failure occurs.

To handle this, a data processing engine could reduce the amount of time in between processing occurrences. If you could have a processing engine that processed data every second, for example, you’d have a much more granular model which could prevent unnecessary downtime from the failure, saving the company from negative consequences of late data.

**Data in Motion or Data at Rest**

The final concept to unpack is that streaming data usually has use cases where a company needs to see current, up-to-the-minute real-time data. This contrasts with data at rest. To compare these two, we can just imagine how frequently we as business leaders need to see updates to our data. Data at rest has settled. Data in motion is currently moving some way - maybe more records are coming in hourly or maybe our data has 3 events every second that comes into our system. Processing data in motion requires a different set of technical capabilities than it takes to process data at rest.

**Summary**

We just explored streaming data. Specifically, we broke streaming data into three constituent components: an engine that is capable of processing streaming data, an unbounded dataset, and data in motion. In the next section, we’ll explore how these three components are engineering in Structured Streaming.

# Introduction to Structured Streaming

**Lesson Introduction**

Before we begin to understand how Structured Streaming goes about processing streaming data, we’ll take a look at its predecessor, Spark Streaming. By looking at the context in which Spark Streaming was created and understand the problems which existed at the time of its inception, we will be able to better understand how Structured Streaming’s paradigms were established. We’ll also get a great look at the benefits of using Structured Streaming.

By the end of this lesson, you'll be able to explain the benefits of Structured Streaming for working with streaming data.

**A Brief History of Streaming in Spark**

Processing streaming data was not a new problem when Structured Streaming’s genesis happened. Historically, streaming data has had commercially available products since at least the late 1990s. The growth of the open-source ecosystem of stream processing technologies started around the early 2010s. These open-source technologies that came out were incredibly mature and battle-tested. Apache Spark’s Structured Streaming was an iteration of Spark Streaming, so to better understand what Structured Streaming is - let’s understand Spark Streaming.

Structured Streaming’s predecessor, Spark Streaming, was originally based on an abstraction of data called a Discretized Stream (D-Stream). The D-Stream was created and written about in a paper by some of the same brilliant minds behind Apache Spark. As the word “discretized” in the name suggests, we can picture a processing engine operating on separate discrete tasks on our data. D-Streams were created with four goals in mind:

* **Scalable:** The system had to be scalable. Since this system is on a distributed set of servers (note: these individual servers are also called “nodes”), the authors wanted their streaming system to be able to scale up to hundreds of nodes.
* **Minimal Costs:** The cost should stay minimal: costs of data storage and data processing costs.
* **Second-scale latency**: Latency refers to the time in between two actions occurring.  In this context, we’re specifically talking about the time in between an event occurring - like an input source passing our stream some data - and the time in which that data is processed.
* **Second-scale recovery from faults and stragglers:** Faults will occur in a distributed system. Servers go down for a large number of reasons, and the more tolerant we can make our systems as well as the time in which it takes to recover from a fault in the system matters greatly in stream processing. Stragglers are nodes that are running slow (another issue that can happen).

Spark Streaming overcame issues with the other traditional stream processing engines which existed at the time. Specifically, it did this by structuring computations as a set of short, stateless, deterministic tasks instead of continuous, stateful operators. Let’s take a look at each of those three ideas because all three of those ideas are in Structured Streaming as well.

* Short tasks are tasks that can complete quickly. We'll cover mini-batch processing in a bit.
* Stateless refers to the idea that the processing engine does not store the state of the tasks. Spark Streaming does store state. However, the stateless notion means if any of our nodes fails, we can restore the state of the data, which makes our system deterministic.
* Deterministic refers to the idea that when a node goes down, D-Streams offer parallel recovery of the node’s state, meaning that each node in the cluster works to recompute part of the lost node’s data. This also means that the recovery doesn’t require replication like other approaches to recovery.

In short, Spark Streaming, based on D-Streams, was a highly successful and powerful paradigm. Structured Streaming is the next genesis of Spark Streaming

**Structured Streaming**

In 2016 an update to Spark Streaming came about. To quickly explain this update: the Resilient Distributed Dataset was one of the original data abstractions of Spark. This abstraction meant that developers could conceptualize and operate on an entire distributed dataset as though it was a single table.  Structured Streaming was built on top of the fantastic leaps made by Spark Streaming, and switched out the basic data abstraction to the Dataset and DataFrame.

Since Structured Streaming has become the more popular of the two streaming options within Spark, let’s explore two major concepts of Structured Streaming: mini-batch processing and appending writes to an unbounded dataset.

What do these mean for an organization?

First of all, the mini-batch processing (based on the discretized task processing mentioned above) meant that Structured Streaming is operationally and conceptually very similar to batch processing. Developers who were previously writing batch-style processing code with Spark only need to change some small amounts of code to handle unbounded and streaming data. Second, it means that the business doesn’t need to pay for double storage costs. There is only a source that is ingested into the data system and is processed either by batch Spark or Structured Streaming, depending on the use case.

To illustrate the second concept, imagine a file in a popular format like Parquet. Regardless of whether we have streaming data or not, we can use this same file for our batch and streaming operations! You can picture the system writing to a table in our file by appending new rows. These append writes to an unbounded dataset, is like our image above. We would like to append to our dataset as new records come in, rather than wait for some identified time period in which we process all historical data. This provides Structured Streaming the means for a system that is fault-tolerant, a concept we’ll go over in greater detail in a moment.

Structured Streaming was built on top of the Datasets and DataFrame API. The DataFrame API was designed so that developers outside of Spark’s initial core user group of Data Engineers - Data Scientists and Data Analysts - could bring their favorite languages like Python, R, Scala, Java, or SQL to operate on a higher-level API. All of this resulted in code that was more abstracted and easier for developers to write but still was incredibly powerful and fast at low levels.

The same goals and benefits of Spark Streaming that were mentioned carried over to Structured Streaming. In most cases, you only need to add a small amount of code to regular Spark code, to switch to running a streaming computation.

The key takeaway here is that Spark Streaming and Structured Streaming are fairly similar in concept and spirit to one another with the notable difference that Structured Streaming operates on a different data abstraction.

* **Prefix-Integrity Guarantee:** Structured Streaming has a prefix-integrity guarantee. This guarantee means that while a streaming query is running, you can use Spark SQL to simultaneously query a table. The streaming query writes the 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 a data system that has continuously flowing data. We call these queries on the table interactive queries.
* **Always Consistent:** Structured Streaming is always consistent, which differs from eventually consistent. In distributed systems, data has to be written out to each node these writes can either be always consistent or eventually consistent. A record that settles at some point in the future is eventually consistent. In organizations with tight turnaround times of data insight, this is important. The data will settle quicker in an always consistent than it would in an eventually consistent system, which then makes it interactive.
* **Fault Tolerance:** Fault tolerance, which means that when one of the nodes in your distributed network goes down, the whole network is tolerant. Our tasks are deterministic - regardless of the time you start processing the data, there is one outcome.  What this looks like in practicality: during the processing of data on a distributed network, there are many things that can and do go wrong, including the failure of nodes in the network. When a task fails, the whole job fails.  When all tasks succeed, the entire job succeeds and it produces only one outcome.
* Handling of out-of-order data, which is data that arrives later than expected, or data that shows up in the system after other data has already been processed. What if our application collects data from users, but our application goes off the network when the user goes on a hike? Their data will arrive at a later point in time and our data system will need to process that accordingly.
* **Joins with Static Data:** Streaming data can join with static data. A stream that needs to be added to a table that already is at rest is  Joining streaming data to a static data table is a huge benefit of working with Structured Streaming.
* **Easy to Use API:** Structured Streaming also has an easy-to-use API built on top of the Datasets and DataFrames API. This means that Data Scientists, Data Analysts, and Data Engineers can all build Structured Streaming pipelines using the data structures that are familiar to them. In most cases, the differences between Spark code and Structured Streaming code are very slight. This leads to lower technical debt because code can be written and then nearly copy/pasted to account for streaming data.

# The Big Data Ecosystem

**Lesson Introduction**

The big data ecosystem consists of many tools. Sometimes, the language used to describe those tools is fairly high level, so it can appear that one tool does a similar thing to another. In this lesson, we'll show some common places around where Structured Structured exists.

By the end of this lesson, you'll be able to:

* Distinguish where Structured Streaming fits into the big data ecosystem.

**Sinks and Sources**

Structured Streaming can take in data (more formally - one can sink data into Structured Streaming) from a wide variety of other tools. Structured Streaming can also be a source of data for another tool downstream. There exists an ecosystem of big data tools which all fit together like puzzle pieces, and in this section, we are going to explore thoseShape

Description automatically generated

Sources of data (input) coming into Structured Streaming could be files or message bus systems. Sources could also be common data types like JSON or text (also known as strings). Message bus systems act as an intermediary datastore, bussing data from one location to another, but also storing that data until it arrives at its destination. Common tools in this space (that Structured Streaming supports intake from) include Apache Kafka, Azure Eventhub, and Amazon Kinesis.  Using these types of tools is often preferred by developers and architects because they also act as a queuing system which is designed to handle data writes.Shape

Description automatically generated

Once data has been processed in Structured Streaming, developers have a wide array of options of where to output or sink data. Depending on the reason why Structured Streaming was chosen - passing data into a machine learning model for example - the sink of data will change. If our reason for Structured Streaming was creating predictions with a machine learning model, we may store those predictions in a file which then is put into a data lakehouse and tracked with a tool like Delta Tables.

Since Structured Streaming was created with distributed computing being the primary environment in design, it was designed to be used alongside other big data technologies. In the next section, we’ll identify those technologies so that as you architect your system, you can make great decisions as to what would be used alongside Structured Streaming.

**Compare and Contrast Structured Streaming with Other Streaming Tools**

So you’ve decided that your organization should be using Structured Streaming. Where does Structured Streaming fit into the ecosystem of big data streaming processing tools?

First, we note that we’re specifically talking about distributed systems in which data no longer is processed on a single node. This limits the number of potential tools for our end goals.

Second, there are several projects in Apache Software’s ecosystem that do stream processing in one way or another. However, Structured Streaming is the only tool that is built on top of Spark SQL, and the only one that uses the Spark SQL engine as an underlying processing engine. Additionally, as we saw, not all streaming tools perform the same functions as Structured Streaming. In fact, we saw that combining several of these tools makes for very powerful architectures for a variety of use cases.

It can be easy to conflate a streaming tool with Structured Streaming, but in general, the guide below can help you distinguish between various tools and their capabilities:

Table, timeline

Description automatically generated

In this section, we’ve discussed moving data into and out of Structured Streaming. We explored common tools that are used alongside Structured Streaming, and then what other tools offer. In the next section, we’ll tie everything together and look at some use cases of Structured Streaming and it was used alongside other popular tools to solve the needs of that use case.

**Lesson Introduction**

In this lesson, we'll go over three use cases where Structured Streaming superpowers the data system around it. By the end of this lesson, you'll be able to:

Articulate examples of real-world business use cases for Structured Streaming.

**First Use Case: A Medical Device**

In the world of health and medical data, processing data after all of it has settled could mean putting people at extreme risk. In this use case, we’re going to use thresholding of a readout from a medical device to determine if an intervention is necessary.  This alerting system based on thresholds is a very common and very solid use case for structured streaming. We’ll diagram our system as follows:

In this alerting system, we ingest data from a medical device into Kafka. This data contains a critical readout about patients. As discussed in our Sources and Sinks, Kafka is then our source of data flowing into Structured Streaming. With Structured Streaming, we can parse our data into two types - records of data which we need an instant alert on (the red above), and the records which are okay (in gray above). Finally, we use Delta Table as a sink where we later perform Extract Transform Load (ETL) operations to refine our data for consumption on a dashboard.

**Second Use Case: IoT Sensor Monitoring**

A retail business uses several devices to monitor its storefront’s current state. These devices continuously create data: readings from a sensor for predictive maintenance on machines, and the amount of carbon monoxide in the building. This data is stored for several reasons, but also it powers a real-time dashboard. The business chooses to store the data in the cloud in a data lake.

It is possible and also a very common pattern to move data from Kafka into Structured Streaming and then back to Kafka where the data is later processed again.

**Third Use Case:  A Social Media Listener**

Consider the velocity and timeline of comments on social media for a financial company. These comments come in with high velocity and typically can provide a real-time pulse of a customer base’s feelings about a brand. As such, a business may not wish to wait for an analytics report to come out after the fact. A business may need to react in real-time.

We might design an architecture like the following:

In this use case, we would push social media comments from their originating platform to a file. We use a file format called Parquet, as it's a convent data store. Next, we write the Parquet files out to a cloud data lake for long-term storage and simultaneously start processing with Structured Streaming. In Structured Streaming, we run a machine learning model on the social media comments to programmatically determine the sentiment of the comment. Finally, we write the data out to our lakehouse, using Delta Lake. While in Structured Streaming, we use a tool called MLflow to track and store experiments of past model runs. This would allow data scientists to develop a model on top of Delta Lake, and in turn, provide a model which in real-time, can detect certain words or phrases. If those words are found, our model could indicate the sentiment of the customer!