Structured Streaming

Big Data Analysis with Scala and Spark

Heather Miller

Why Structured Streaming?

DStreams were nice, but in the last session, aggregation operations like a simple word count quickly stopped looking like regular (batch) Spark.

Why Structured Streaming?

DStreams were nice, but in the last session, aggregation operations like a simple word count quickly stopped looking like regular (batch) Spark.

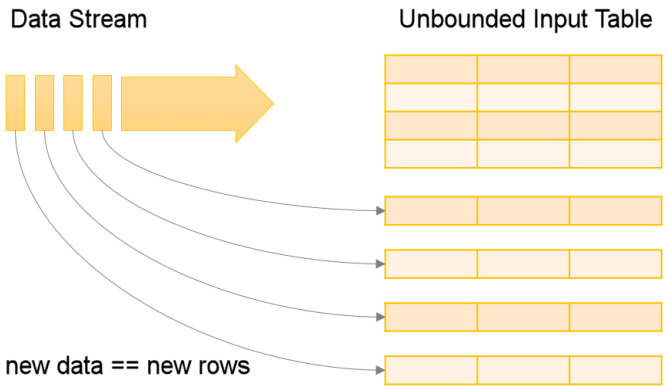
Spark Streaming is to DStreams what DataFrames are to RDDs.

Simply put, the Structured Streaming APIs aim to be:

- Simpler to use.
- More performant.

Structured Streaming

Conceptually, Structured Streaming treats all the data arriving as an unbounded input table.



Structured Streaming's Model of Computation

Two main steps.

1

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.

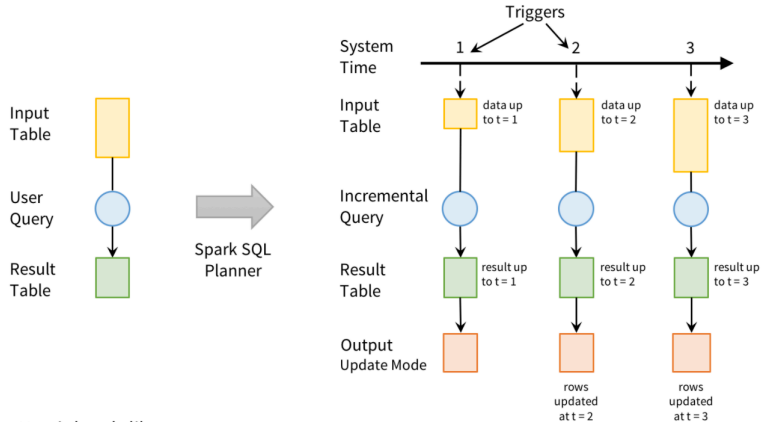
Spark automatically converts this batch-like query to a streaming execution plan.

2

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.

Structured Streaming's Model of Computation



Structured Streaming Processing Model

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

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.

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.

Streaming provides three output modes:

1. **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).
2. **Complete**. The entire updated result table will be written to external storage.
3. **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.

Intuitive Structured Streaming Example

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.

Intuitive Structured Streaming Example

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")
```

Intuitive Structured Streaming Example

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")
```

We would like our resulting program to look as similar between batch and streaming modes as possible.

Intuitive Structured Streaming Example

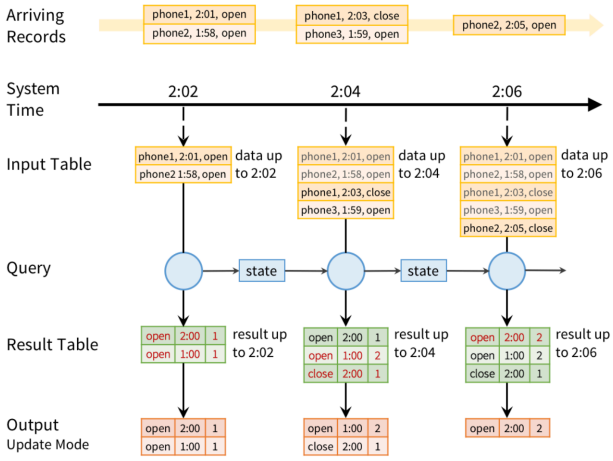
Our batch query is to compute a count of actions grouped by (action, hour).

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. (Append, Complete, or Update.)

Intuitive Structured Streaming Example

The figure below shows this execution using the **Update** output mode:



Data Streams in Structured Streaming

Streams in Structured Streaming are represented as DataFrames or Datasets with the isStreaming property set to true.

They can also be created via the special read methods for different sources. E.g, from S3:

```
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.

Another Structured Streaming Example

Let's say we want to maintain a running word count of text data received from a data server listening on a TCP socket.

```
import org.apache.spark.sql.functions._
import org.apache.spark.sql.SparkSession

val spark = SparkSession
  .builder
  .appName("StructuredNetworkWordCount")
  .getOrCreate()

import spark.implicits._
```

First, we have to import the necessary classes and create a local `SparkSession`.

Another Structured Streaming Example

Next, let's create a streaming DataFrame that represents text data received from a server listening on localhost:9999, and transform the DataFrame to calculate word counts.

```
// Create DataFrame representing the stream of input lines from localhost:9999
val lines = spark.readStream
  .format("socket")
  .option("host", "localhost")
  .option("port", 9999)
  .load()

// Split the lines into words
val words = lines.as[String].flatMap(x => x.split(" "))

// Generate running word count
val wordCounts = words.groupBy("value").count()
```

Another Structured Streaming Example

What's happened so far?

Another Structured Streaming Example

What's happened so far?

Nothing.

We have only set up the query on the streaming data.

Another Structured Streaming Example

What's happened so far?

Nothing.

We have only set up the query on the streaming data.

We want to print the entire set of counts to the console. So we'll also have to make sure we set up the output mode correctly before we kick off computation.

Another Structured Streaming Example

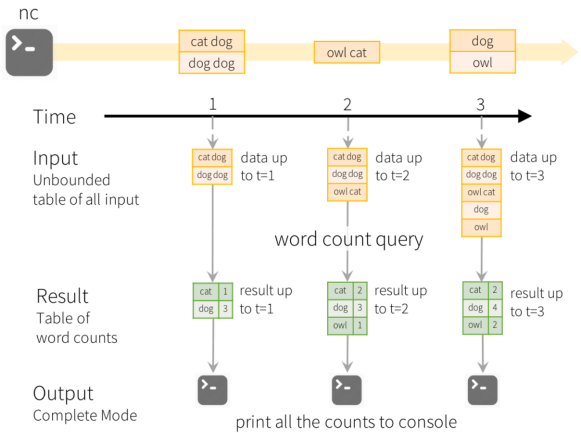
We want to print the entire set of counts to the console. So we'll also have to make sure we set up the output mode correctly before we kick off computation.

```
// Start running the query that prints the running counts to the console
val query = wordCounts.writeStream
  .outputMode("complete")
  .format("console")
  .start()

query.awaitTermination()
```

Another Structured Streaming Example (Visualized)

Streaming word count, visualized.



Important to Note

Structured Streaming does not materialize the entire table.
It reads the latest available data from the streaming data source, processes it incrementally to update the result, and then discards the source data. It only keeps around the minimal intermediate state data as required to update the result (e.g. intermediate counts in the earlier example).

Important to Note

Structured Streaming does not materialize the entire table.
It reads the latest available data from the streaming data source, processes it incrementally to update the result, and then discards the source data. It only keeps around the minimal intermediate state data as required to update the result (e.g. intermediate counts in the earlier example).

Note that you do not have to maintain running aggregations, you don't have to reason about fault-tolerance, or data consistency (at-least-once, or at-most-once, or exactly-once)

Important to Note

Structured Streaming does not materialize the entire table.
It reads the latest available data from the streaming data source, processes it incrementally to update the result, and then discards the source data. It only keeps around the minimal intermediate state data as required to update the result (e.g. intermediate counts in the earlier example).

Note that you do not have to maintain running aggregations, you don't have to reason about fault-tolerance, or data consistency (at-least-once, or at-most-once, or exactly-once)

Spark simply updates the Result Table when there is new data, thus relieving the users from reasoning about it.

Operations on Streaming DataFrames

Basic Operations - Selection, Projection, Aggregation

Most of the common operations on DataFrame/Dataset are supported for streaming.

The handful of operations that are not supported:

- `take` and `limit`
- `distinct`
- sorting operations
- some outer joins
- certain actions (eager evaluation does not make sense on streaming Datasets)
  - `count()` returns a running count, not a single count
  - `foreach()` use a foreach sink instead, e.g., `writeStream.foreach(...)`
  - `show()` use a console sink instead, e.g., `writeStream.format("console")`