Spark Structured Streaming

Emanuele Della Valle

prof @ Politecnico di Milano
Co-founder @ Quantia Consulting
emanuele.dellavalle@polimi.it
http://emanueledellavalle.org

Overview

Spark Structured Streaming provides fast, scalable, fault-tolerant, end-to-end exactly-once stream processing without the user having to reason about streaming.

Positioning Spark Structured Streaming

- default: **micro-batch processing** engine
  - end-to-end latencies as low as 100 milliseconds
  - exactly-once fault-tolerance guarantees
- since Spark 2.3, also **Continuous Processing**
  - end-to-end latencies as low as 1 millisecond
  - at-least-once guarantees
- choose the mode based on requirements

Programming Model

- key idea: treat a live data stream as a table that is being continuously appended
- Result:
  - user can express streaming computation as standard batch-like query as on a static table
  - Spark runs it as an *incremental* query on the *unbounded* input table

Basic Concepts 1/2

•

Basic Concepts 2/2

-

Output Modes

- *Complete Mode*: The entire updated result is written
- *Append Mode*:
  - Only the new rows appended in the result are written
  - Applicable only when existing rows in the result are not expected to change
- *Update Mode*:
  - Only outputs the rows that have changed since the last trigger
  - If the query doesn't contain aggregations, it will be equivalent to Append mode.

Note that each mode is applicable on certain types of queries.

Example

-

Notes

- Spark Structured Streaming does not materialize the entire table
  - reads the latest available data from the stream
  - processes it incrementally to update the result
  - discards the source data
- It only keeps around the minimal intermediate *state* data
  - e.g. intermediate counts

API using Datasets and DataFrames

Since Spark 2.0, DataFrames and Datasets can represent static, bounded data, as well as streaming, unbounded data.

Creating streaming DataFrames and streaming Datasets

Streaming DataFrames can be created through the `DataStreamReader` interface
(Scala/Java/Python docs)
returned by `SparkSession.readStream()`.

- Input Sources:
  - **File source** - Supported file formats: text, CSV, JSON, ORC, Parquet.
  - **Kafka source** - Reads data from Kafka (see the Kafka Integration Guide)
  - **Socket source (for testing)** - Reads UTF8 text data from a socket connection. It does not provide end-to-end fault-tolerance guarantees.

E.g., python

```
{% highlight python %}
spark = SparkSession. ...

# Read text from socket
socketDF = spark \
    .readStream \
    .format("socket") \
    .option("host", "localhost") \
    .option("port", 9999) \
    .load()

socketDF.isStreaming()    # Returns True for DataFrames that have streaming sources

socketDF.printSchema()

# Read all the csv files written atomically in a directory
userSchema = StructType().add("name", "string").add("age", "integer")
csvDF = spark \
    .readStream \
    .option("sep", ";") \
    .schema(userSchema) \
    .csv("/path/to/directory")  # Equivalent to format("csv").load("/path/to/directory")
```

Basic Operations - Selection, Projection, Aggregation

— NOTE: few operations are not supported

```
df = ...  # streaming DataFrame with IOT device data with schema { device: string, deviceType: string, signal: double, time: DateType }

# Select the devices which have signal more than 10
df.select("device").where("signal > 10")

# Running count of the number of updates for each device type
df.groupBy("deviceType").count()
```

Alternatively, register a streaming DataFrame/Dataset as a temporary view and apply SQL on it

```
df.createOrReplaceTempView("updates")
spark.sql("select count(*) from updates")  # returns another streaming DF
```

Window Operations on Event Time

•

Window Operations on Event Time (cont.)

Spark Structured Streaming treats this type of windows as grouping clauses.

```
words = ...  # streaming DataFrame of schema { timestamp: Timestamp, word: String }

# Group the data by window and word and compute the count of each group
windowedCounts = words.groupBy(
    window(words.timestamp, "10 minutes", "5 minutes"),
    words.word
).count()
```

Handling Late Data and Watermarking

-

Handling Late Data and Watermarking (cont.)

```
words = ...  # streaming DataFrame of schema { timestamp: Timestamp, word: String }

# Group the data by window and word and compute the count of each group
windowedCounts = words \
    .withWatermark("timestamp", "10 minutes") \
    .groupBy(
        window(words.timestamp, "10 minutes", "5 minutes"),
        words.word) \
    .count()
```

Watermarking in Update Mode

-

Watermarking in Append Mode

-

Semantic Guarantees of Aggregation with Watermarking

- A watermark delay of "X hours" guarantees that the engine will never drop any data that is less than x hours delayed.
- **However**, data delayed by more than 2 hours is not guaranteed to be dropped
  - it may or may not get aggregated
  - the more delayed, the less likely

Join Operations

The result of the streaming join is **generated incrementally**

Stream-static Joins

```
staticDf = spark.read. ...
streamingDf = spark.readStream. ...
streamingDf.join(staticDf, "type")  # inner equi-join with a static DF
streamingDf.join(staticDf, "type", "left_outer")  # left outer join with a static DF
```

Notes

- INNER and LEFT-OUTER stream-static joins are not stateful
- OUTER and RIGHT-OUTER stream-static joins are not supported

for more information.

Stream-stream Joins

- **Problem**: Any row received from one input stream can match with any future, yet-to-be-received row from the other input stream.
- **Solution**:
  - **Windows**, but not the kind you can express as a group by clause
    - buffer past input as streaming state
    - handle late, out-of-order data using watermarks

Stream-stream Joins (cont.)

```
from pyspark.sql.functions import expr

impressions = spark.readStream. ...
clicks = spark.readStream. ...

# Apply watermarks on event-time columns
impressionsWithWatermark = impressions.withWatermark("impressionTime", "2 hours")
clicksWithWatermark = clicks.withWatermark("clickTime", "3 hours")

# Join with event-time constraints
impressionsWithWatermark.join(
  clicksWithWatermark,
  expr("""
    clickAdId = impressionAdId AND
    clickTime >= impressionTime AND
    clickTime <= impressionTime + interval 1 hour
    """)
)
```

Stream-stream Joins

Join Type	
Inner	Supported, optionally specify watermark on both sides + time constraints for state cleanup
Left Outer	Conditionally supported, must specify watermark on right + time constraints for correct results, optionally specify watermark on left for all state cleanup
Right Outer	Conditionally supported, must specify watermark on left + time constraints for correct results, optionally specify watermark on right for all state cleanup
Full Outer	Not supported

Policy for handling multiple watermarks

A streaming query can have multiple input streams that are unioned or joined together.
Each of the input streams can have a different threshold of late data that needs to
be tolerated for stateful operations. You specify these thresholds using `withWatermarks("eventTime", delay)` on each of the input streams. For example, consider
a query with stream-stream joins between `inputStream1` and `inputStream2`.

</div>

While executing the query, Structured Streaming individually tracks the maximum

Starting Streaming Queries

- Once you have defined the final result DataFrame/Dataset, all that is left is for you to start the streaming computation.
- To do that use the `DataStreamWriter` (Scala/Java/Python docs) returned through `Dataset.writeStream()`

Starting Streaming Queries (cont.)

- specify:
  - *output sink:* Data format, location, etc.
  - *Output mode:* what gets written
  - *Query name:* Optionally, a unique name of the query
  - *Trigger interval:* Optionally, the trigger interval
  - *Checkpoint location*: for the end-to-end fault-tolerance
- Output Sinks:
  - **File sink** - a directory
  - **Kafka sink** - one or more topics in Kafka
  - **Foreach sink** - Runs arbitrary computation
  - **Console sink (for debugging)** + Memory

Managing Streaming Queries

```
query = df.writeStream.format("console").start()   # get the query object

query.id()          # get the unique identifier of the running query that persists across restarts from checkpoint data

query.runId()       # get the unique id of this run of the query, which will be generated at every start/restart

query.name()        # get the name of the auto-generated or user-specified name

query.explain()   # print detailed explanations of the query

query.stop()      # stop the query

query.awaitTermination()   # block until query is terminated, with stop() or with error

query.exception()       # the exception if the query has been terminated with error

query.recentProgress()  # an array of the most recent progress updates for this query

query.lastProgress()    # the most recent progress update of this streaming query
```

Further Reading

- See and run the Scala/Java/Python/R examples.
  - Instructions on how to run Spark examples
- Read about integrating with Kafka in the Structured Streaming Kafka Integration Guide
- Read more details about using DataFrames/Datasets in the Spark SQL Programming Guide
- Third-party Blog Posts
  - Real-time Streaming ETL with Structured Streaming in Apache Spark 2.1 (Databricks Blog)
  - Real-Time End-to-End Integration with Apache Kafka in Apache Spark's

Talks

- Spark Summit Europe 2017
  - Easy, Scalable, Fault-tolerant Stream Processing with Structured Streaming in Apache Spark - Part 1 slides/video, Part 2 slides/video
  - Deep Dive into Stateful Stream Processing in Structured Streaming - slides/video
- Spark Summit 2016
  - A Deep Dive into Structured Streaming - slides/video

Spark Structured Streaming

Emanuele Della Valle

prof @ Politecnico di Milano
Co-founder @ Quantia Consulting
emanuele.dellavalle@polimi.it
http://emanueledellavalle.org