Deep Dive into Stateful Stream Processing in Structured Streaming

Tathagata "*TD*" Das

@tathadas

Spark Summit Europe 2017

25th October, Dublin

databricks

Structured Streaming

stream processing on Spark SQL engine
fast, scalable, fault-tolerant

rich, unified, high level APIs
deal with *complex data* and *complex workloads*

rich ecosystem of data sources
integrate with many *storage systems*

you

should not have to

reason about streaming

you

should write simple queries

&

Spark

should continuously update the answer

Anatomy of a Streaming Word Count

```
spark.readStream
  .format("kafka")
  .option("subscribe", "input")
  .load()
```

Source

Specify one or more locations to read data from

Built in support for Files/Kafka/Socket, pluggable.

Generates a DataFrame

Anatomy of a Streaming Word Count

```
spark.readStream
  .format("kafka")
  .option("subscribe", "input")
  .load()
  .groupBy('value.cast("string") as 'key)
  .agg(count("*") as 'value)
```

Transformation

DataFrame, Dataset operations and/or SQL queries

Spark SQL figures out how to execute it incrementally

Internal processing always exactly-once

Anatomy of a Streaming Word Count

```
spark.readStream
  .format("kafka")
  .option("subscribe", "input")
  .load()
  .groupBy('value.cast("string") as 'key)
  .agg(count("*") as 'value)
  .writeStream
  .format("kafka")
  .option("topic", "output")
```

Sink

Accepts the output of each batch

When supported sinks are transactional and exactly once (e.g. files)

Use `foreach` to execute arbitrary code

Anatomy of a Streaming Word Count

```
spark.readStream
  .format("kafka")
  .option("subscribe", "input")
  .load()
  .groupBy('value.cast("string") as 'key)
  .agg(count("*") as 'value)
  .writeStream
  .format("kafka")
  .option("topic", "output")
  .trigger("1 minute")
  .outputMode("update")
```

Output mode – What's output

Complete – Output the whole answer every time

Update – Output changed rows

Append – Output new rows only

Trigger – When to output

Specified as a time, eventually supports data size

No trigger means as fast as possible

Anatomy of a Streaming Word Count

```
spark.readStream
  .format("kafka")
  .option("subscribe", "input")
  .load()
  .groupBy('value.cast("string") as 'key)
  .agg(count("*") as 'value)
  .writeStream
  .format("kafka")
  .option("topic", "output")
  .trigger("1 minute")
  .outputMode("update")
  .option("checkpointLocation", "/cp/")
  .start()
```

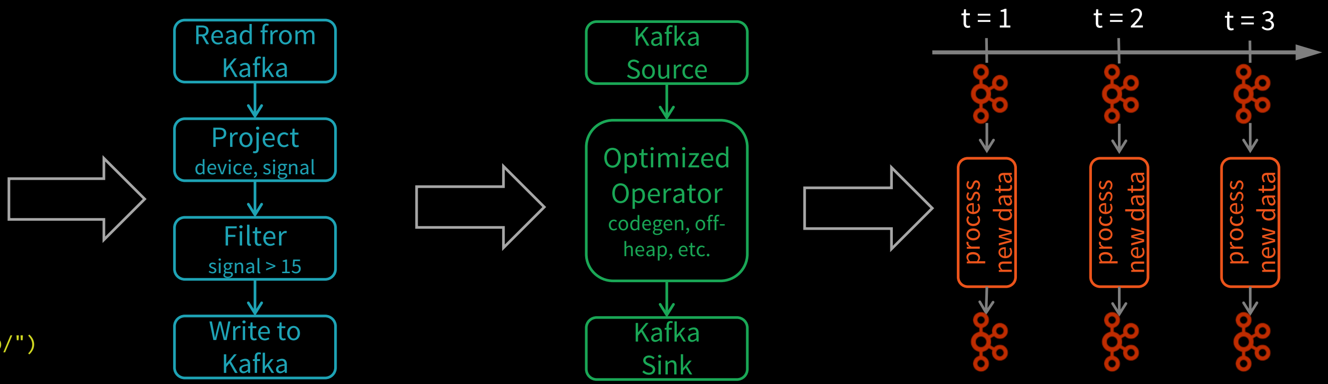
Checkpoint

Tracks the progress of a query in persistent storage

Can be used to restart the query if there is a failure

Spark automatically streamifies!

```
spark.readStream
  .format("kafka")
  .option("subscribe", "input")
  .load()
  .groupBy(
    'value.cast("string") as 'key)
  .agg(count("*") as 'value)
  .writeStream
  .format("kafka")
  .option("topic", "output")
  .trigger("1 minute")
  .outputMode("update")
  .option("checkpointLocation", "/cp/")
  .start()
```



DataFrames, Datasets, SQL

Logical Plan

Optimized Physical Plan

Series of Incremental Execution Plans

Spark SQL converts batch-like query to a series of incremental execution plans operating on new batches of data

databricks

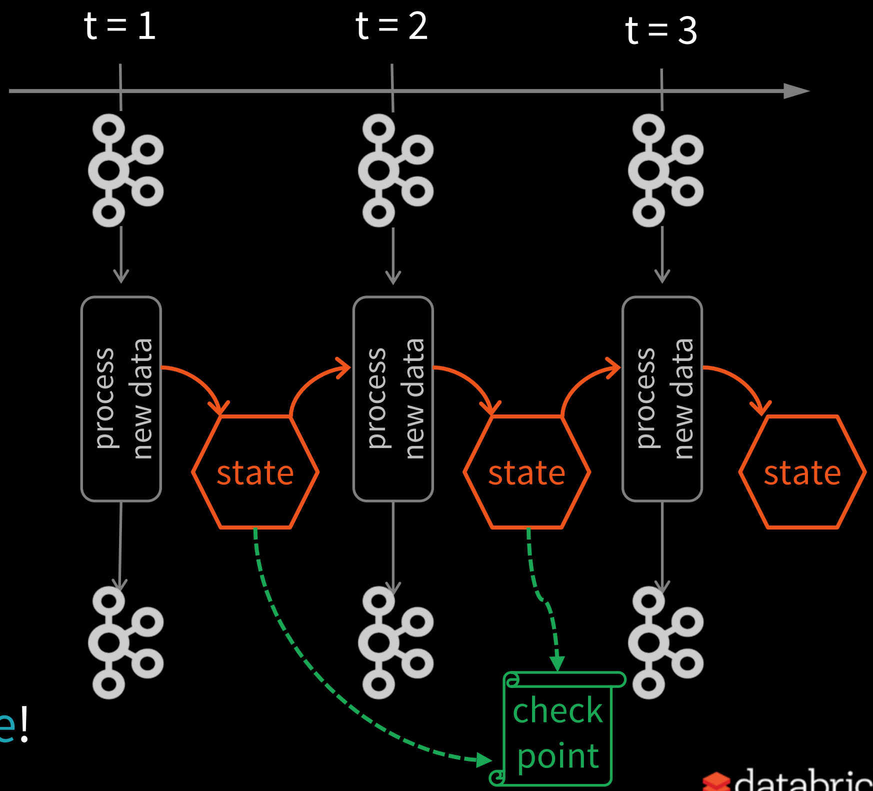
Spark automatically streamifies!

Partial counts carries across triggers as *distributed state*

Each execution reads previous state and writes out updated state

State stored in executor memory (hashmap in Apache, RocksDB in Databricks Runtime), backed by *checkpoints* in HDFS/S3

Fault-tolerant, exactly-once guarantee!



This Talk

Explore built-in stateful operations

How to use watermarks to control state size

How to build arbitrary stateful operations

How to monitor and debug stateful queries

[For a general overview of SS, see my earlier talks]

Streaming *Aggregation*

Aggregation by key and/or time windows

Aggregation by key only

```
events
  .groupBy("key")
  .count()
```

Aggregation by event time windows

```
events
  .groupBy(window("timestamp","10 mins"))
  .avg("value")
```

Aggregation by both

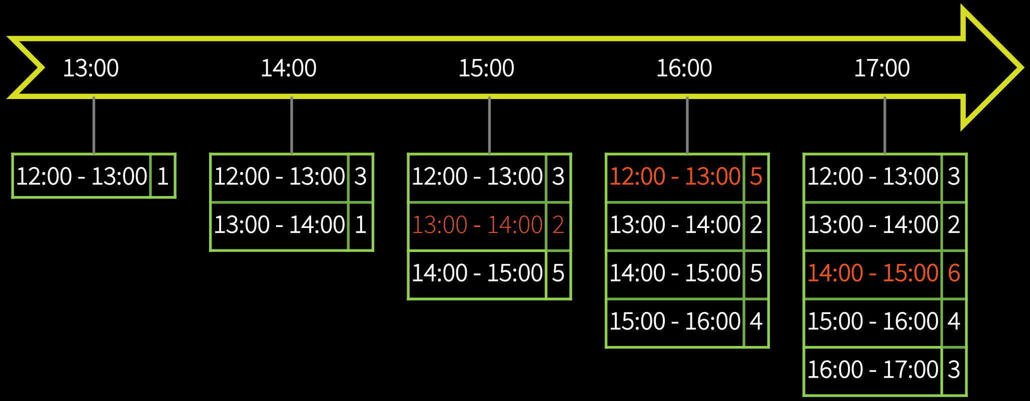
```
events
  .groupBy(
    'key,
    window("timestamp","10 mins"))
  .agg(avg("value"), corr("value"))
```

Supports multiple aggregations,
user-defined functions (UDAFs)!

databricks

Automatically handles Late Data

Keeping state allows late data to update counts of old windows



But size of the state increases indefinitely if old windows are not dropped

red = state updated with late data

Watermarking

Watermark - moving threshold of how late data is expected to be and when to drop old state

Trails behind max event time seen by the engine

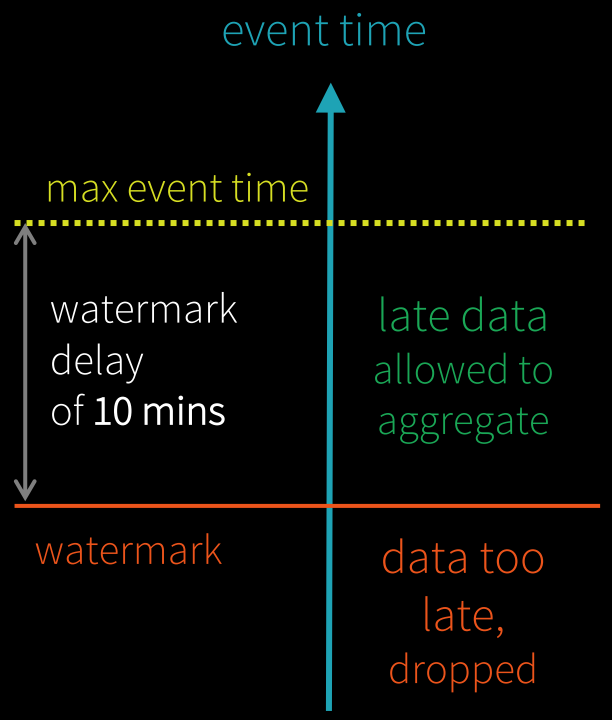
Watermark delay = trailing gap

Watermarking

Data newer than watermark may be late, but allowed to aggregate

Data older than watermark is "too late" and dropped

Windows older than watermark automatically deleted to limit the amount of intermediate state

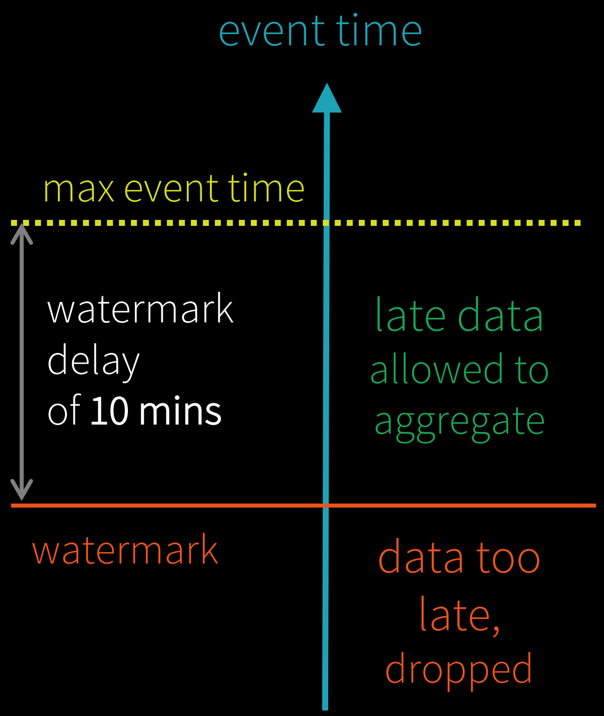


Watermarking

```
parsedData
  .withWatermark("timestamp", "10 minutes")
  .groupBy(window("timestamp","5 minutes"))
  .count()
```

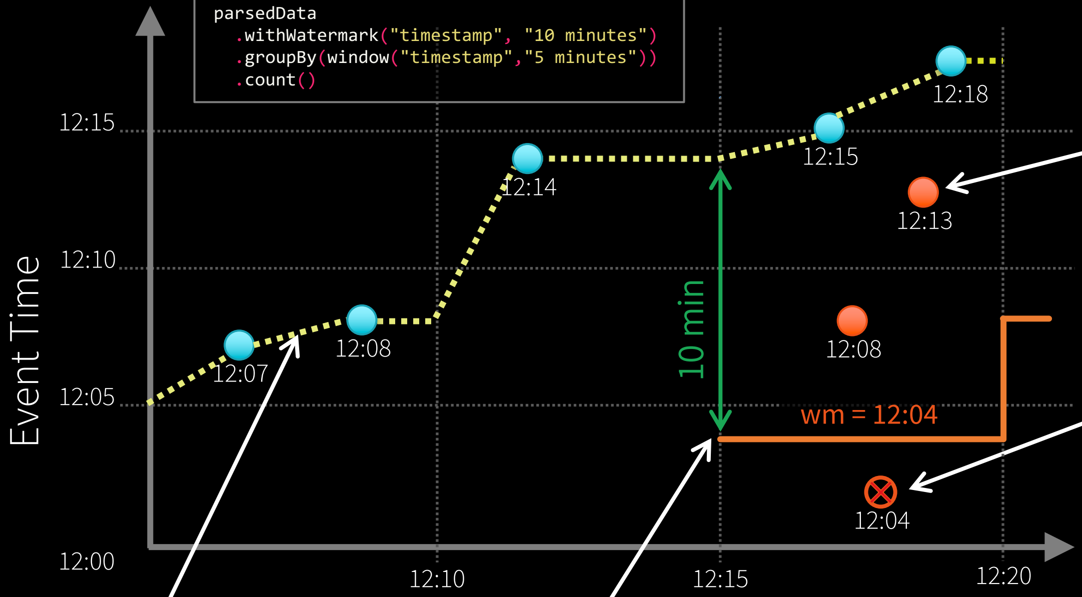
Used only in stateful operations

Ignored in non-stateful streaming queries and batch queries



Watermarking

```
parsedData
  .withWatermark("timestamp", "10 minutes")
  .groupBy(window("timestamp","5 minutes"))
  .count()
```



data is late, but considered in counts

data too late, ignored in counts, state dropped

system tracks max observed event time

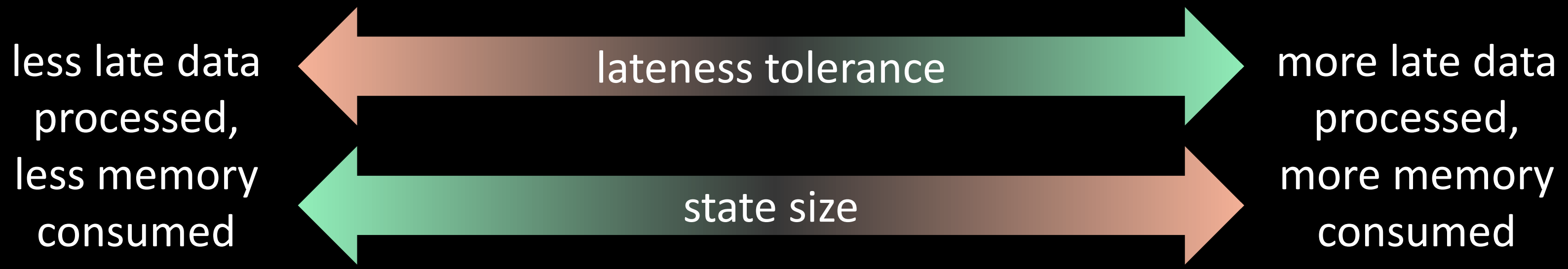
watermark updated to
12:14 - 10m = 12:04
for next trigger,
state < 12:04 deleted

More details in my [blog post]

databricks

Watermarking

Trade off between lateness tolerance and state size



Streaming *Deduplication*

databricks

Streaming Deduplication

Drop duplicate records in a stream

Specify columns which uniquely identify a record

```
userActions
  .dropDuplicates("uniqueRecordId")
```

Spark SQL will store past unique column values as state and drop any record that matches the state

Streaming Deduplication with Watermark

Timestamp as a unique column along with watermark allows old values in state to dropped

- Records older than watermark delay is not going to get any further duplicates

Timestamp must be same for duplicated records

```
userActions
  .withWatermark("timestamp")
  .dropDuplicates(
    "uniqueRecordId",
    "timestamp")
```

Streaming *Joins*

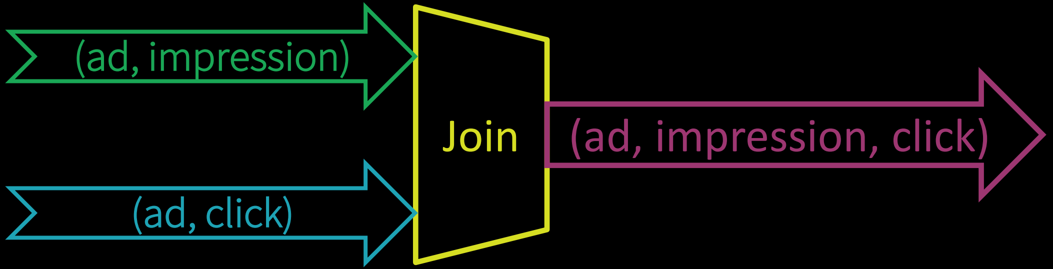
Streaming Joins

Spark 2.0+ support joins between streams and static datasets

Spark 2.3+ will support joins between multiple streams

Example: Ad Monetization

Join stream of ad impressions with another stream of their corresponding user clicks

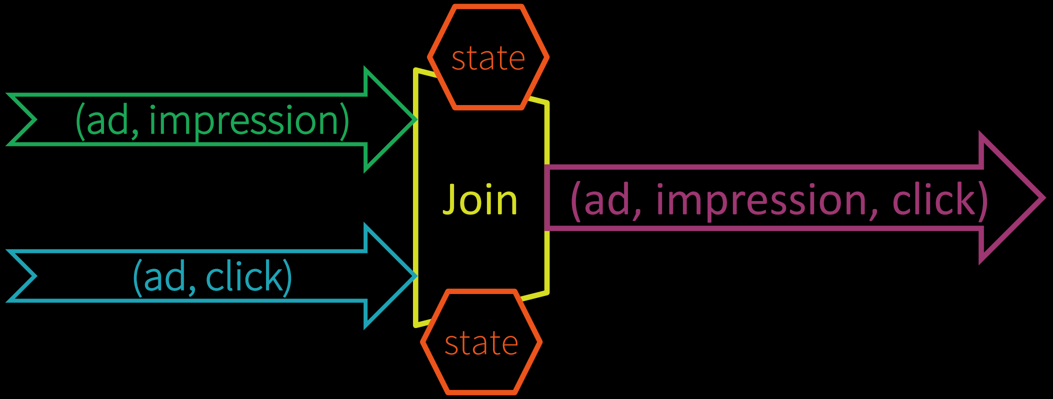


Streaming Joins

Most of the time click events arrive after their impressions

Sometimes, due to delays, impressions can arrive after clicks

Each stream in a join needs to buffer past events as *state* for matching with future events of the other stream



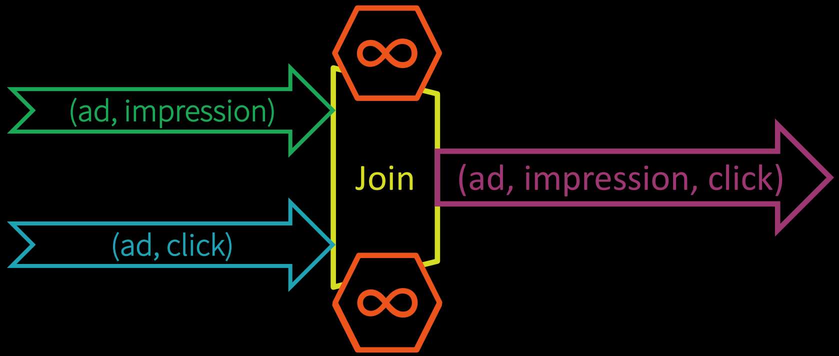
Simple Inner Join

Inner join by ad ID column

Need to buffer all past events as state, a match can come on the other stream any time in the future

To allow buffered events to be dropped, query needs to provide more time constraints

```
impressions.join(
  clicks,
  expr("clickAdId = impressionAdId")
)
```



Inner Join + Time constraints + Watermarks

Time constraints

- Impressions can be 2 hours late

```
val impressionsWithWatermark = impressions
      .withWatermark("impressionTime", "2 hours")
```

- Clicks can be 3 hours late

```
val clicksWithWatermark = clicks
      .withWatermark("clickTime", "3 hours")
```

- A click can occur within 1 hour after the corresponding impression

Range Join

```
impressionsWithWatermark.join(
  clicksWithWatermark,
  expr("""
    clickAdId = impressionAdId AND
    clickTime >= impressionTime AND
    clickTime <= impressionTime + interval 1 hour
    """
  ))
```

databricks

Inner Join + Time constraints + Watermarks

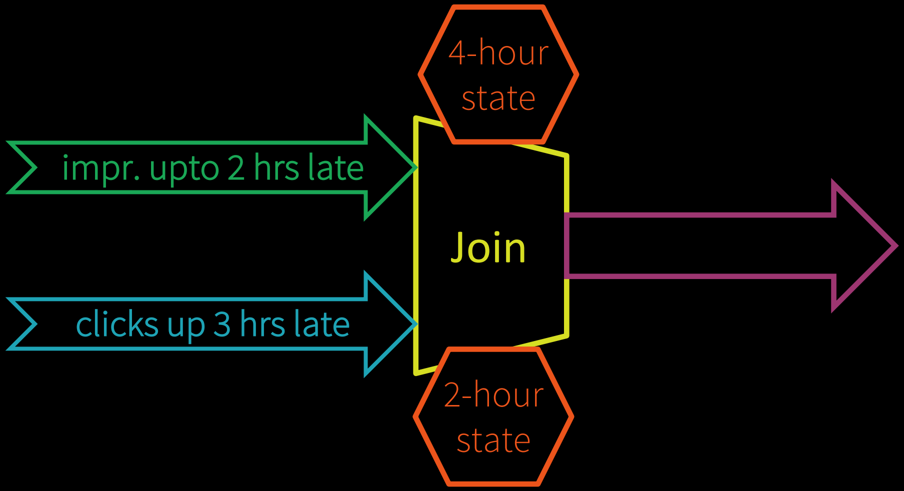
Spark calculates

- impressions need to be buffered for 4 hours

- clicks need to be buffered for 2 hours

Spark drops events older than these thresholds

3-hour-late click may match with impression received 4 hours ago



2-hour-late impression may match with click received 2 hours ago

databricks

Outer Join + Time constraints + Watermarks

Left and right outer joins are allowed only with time constraints and watermarks

Needed for correctness, Spark must output nulls when an event cannot get any future match

```
impressionsWithWatermark.join(
  clicksWithWatermark,
  expr("""
    clickAdId = impressionAdId AND
    clickTime >= impressionTime AND
    clickTime <= impressionTime + interval 1 hour
    """
  ),
  joinType = "leftOuter"
)
```

Can be "inner" (default) / "leftOuter"/ "rightOuter"

Arbitrary Stateful Operations

Arbitrary Stateful Operations

Many use cases require more complicated logic than SQL ops

Example: Tracking user activity on your product

- Input: User actions (login, clicks, logout, …)
- Output: Latest user status (online, active, inactive, …)

Solution: MapGroupsWithState / FlatMapGroupsWithState

- General API for per-key user-defined stateful processing
- More powerful + efficient than DStream's mapWithState/updateStateByKey
- Since Spark 2.2, for Scala and Java only

MapGroupsWithState - How to use?

1. Define the data structures

   - *Input event:* UserAction
   - *State data:* UserStatus
   - *Output event:* UserStatus
     (can be different from state)

```
case class UserAction(
    userId: String, action: String)

case class UserStatus(
    userId: String, active: Boolean)
```

MapGroupsWithState - How to use?

2. Define function to update state of each grouping key using the new data

   - *Input*

     - *Grouping key:* userId

     - *New data:* new user actions

     - *Previous state:* previous status of this user

```
case class UserAction(
    userId: String, action: String)

case class UserStatus(
    userId: String, active: Boolean)


def updateState(
  userId: String,
  actions: Iterator[UserAction],
  state: GroupState[UserStatus]):UserStatus = {


}
```

databricks

MapGroupsWithState - How to use?

2. Define function to update state of each grouping key using the new data

   - *Body*

     - Get previous user status
     - Update user status with actions
     - Update state with latest user status
     - Return the status

```
def updateState(
  userId: String,
  actions: Iterator[UserAction],
  state: GroupState[UserStatus]):UserStatus = {

  val prevStatus = state.getOption.getOrElse {
    new UserStatus()
  }

  actions.foreah { action =>
    prevStatus.updateWith(action)
  }

  state.update(prevStatus)

  return prevStatus

}
```

databricks

MapGroupsWithState - How to use?

3. Use the user-defined function on a grouped Dataset

```
def updateState(
  userId: String,
  actions: Iterator[UserAction],
  state: GroupState[UserStatus]):UserStatus = {

  // process actions, update and return status

}
```

Works with both batch and streaming queries

- In batch query, the function is called only once per group with no prior state

```
userActions
  .groupByKey(_.userId)
  .mapGroupsWithState(updateState)
```

Timeouts

Example: Mark a user as inactive when there is no actions in 1 hour

Timeouts: When a group does not get any event for a while, then the function is called for that group with an empty iterator

Must specify a global timeout type, and set per-group timeout timestamp/duration

Ignored in a batch queries

```
userActions
   .groupByKey(_.userId)
   .mapGroupsWithState
     (timeoutConf)(updateState)
```

- NoTimeout (default)
- EventTime Timeout
- ProcessingTime Timeout

Event-time Timeout - How to use?

1. Enable EventTimeTimeout in mapGroupsWithState
2. Enable watermarking
3. Update the mapping function
   - Every time function is called, set the timeout timestamp using the max seen event timestamp + timeout duration
   - Update state when timeout occurs

```
userActions
  .withWatermark("timestamp")
  .groupByKey(_.userId)
  .mapGroupsWithState
    (EventTimeTimeout)(updateState)

 def updateState(...): UserStatus = {
   if (!state.hasTimedOut) {
     // track maxActionTimestamp while
     // processing actions and updating state
     state.setTimeoutTimestamp(
       maxActionTimestamp, "1 hour")
   } else {  // handle timeout
     userStatus.handleTimeout()
     state.remove()
   }
   // return user status
 }
```

databricks

Event-time Timeout - When?

Watermark is calculated with max event time across all groups

For a specific group, if there is no event till watermark exceeds the timeout timestamp,

Then

Function is called with an empty iterator, and hasTimedOut = true

Else

Function is called with new data, and timeout is disabled

Needs to explicitly set timeout timestamp every time

Processing-time Timeout

Instead of setting timeout timestamp, function sets timeout duration (in terms of wall-clock-time) to wait before timing out

Independent of watermarks

Note, query downtimes will cause lots of timeouts after recovery

```
userActions
   .groupByKey(_.userId)
   .mapGroupsWithState
     (ProcessingTimeTimeout)(updateState)


def updateState(...): UserStatus = {
   if (!state.hasTimedOut) {
     // handle new data
     state.setTimeoutDuration("1 hour")
   } else {
     // handle timeout
   }
   return userStatus
}
```

FlatMapGroupsWithState

More general version where the function can return any number of events, possibly none at all

Example: instead of returning user status, want to return specific actions that are significant based on the history

```
userActions
   .groupByKey(_.userId)
   .flatMapGroupsWithState
     (outputMode, timeoutConf)
     (updateState)
```

```
def updateState(
      userId: String,
      actions: Iterator[UserAction],
      state: GroupState[UserStatus]):
   Iterator[SpecialUserAction] = {

}
```

databricks

Function Output Mode

Function output mode* gives Spark insights into the output from this opaque function

> *Update Mode* - Output events are key-value pairs, each output is updating the value of a key in the result table

> *Append Mode* - Output events are independent rows that being appended to the result table

Allows Spark SQL planner to correctly compose flatMapGroupsWithState with other operations

```
userActions
  .groupByKey(_.userId)
  .flatMapGroupsWithState
    (outputMode, timeoutConf)
    (updateState)
```

Update Mode | Append Mode

*Not to be confused with output mode of the query

Monitoring Stateful Streaming Queries

Monitor State Memory Consumption

Get current state metrics using the last progress of the query

- Total number of rows in state
- Total memory consumed (approx.)

```
val progress = query.lastProgress
print(progress.json)
```

```
{
 ...
 "stateOperators" : [ {
   "numRowsTotal" : 660000,
   "memoryUsedBytes" : 120571087
   ...
 } ],
}
```

Get it asynchronously through StreamingQueryListener API

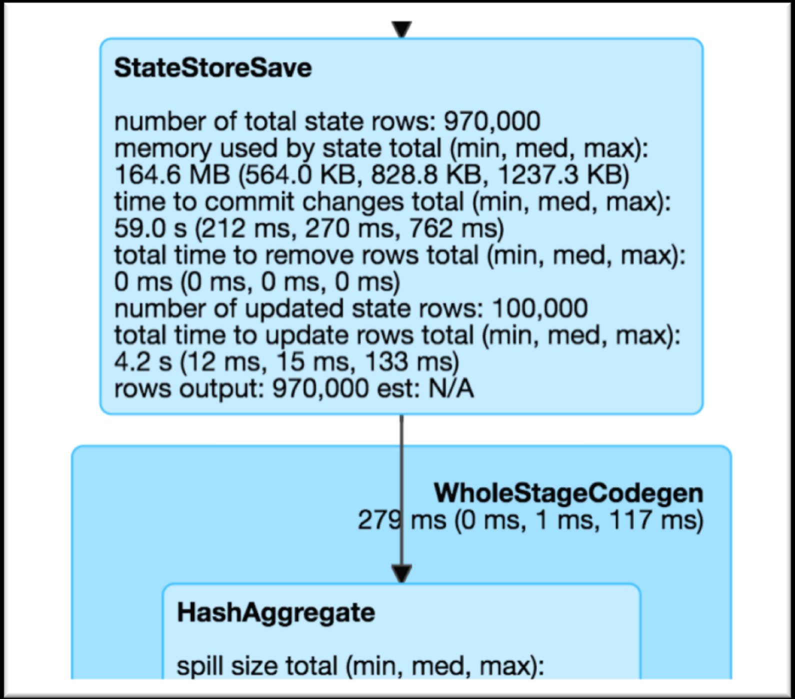
```
new StreamingQueryListener {
  ...
  def onQueryProgress(
   event: QueryProgressEvent)
}
```

Debug Stateful Operations

SQL metrics in the Spark UI (SQL tab, DAG view) expose more operator-specific stats

Answer questions like

- Is the memory usage skewed?
- Is removing rows slow?
- Is writing checkpoints slow?



More Info

Structured Streaming Programming Guide

http://spark.apache.org/docs/latest/structured-streaming-programming-guide.html

Databricks blog posts for more focused discussions

https://databricks.com/blog/2016/07/28/structured-streaming-in-apache-spark.html

https://databricks.com/blog/2017/01/19/real-time-streaming-etl-structured-streaming-apache-spark-2-1.html

https://databricks.com/blog/2017/02/23/working-complex-data-formats-structured-streaming-apache-spark-2-1.html

https://databricks.com/blog/2017/04/26/processing-data-in-apache-kafka-with-structured-streaming-in-apache-spark-2-2.html

https://databricks.com/blog/2017/05/08/event-time-aggregation-watermarking-apache-sparks-structured-streaming.html

https://databricks.com/blog/2017/10/11/benchmarking-structured-streaming-on-databricks-runtime-against-state-of-the-art-streaming-systems.html

and more to come, stay tuned!!

Try Apache Spark in Databricks!

UNIFIED ANALYTICS PLATFORM

- Collaborative cloud environment
- Free version (community edition)

DATABRICKS RUNTIME 3.0

- Apache Spark - optimized for the cloud
- Caching and optimization layer - DBIO
- Enterprise security - DBES

Try for free today

databricks.com