Table of Contents

Introduction 1.1
Spark Structured Streaming — Streaming Datasets 1.2

Developing Streaming Applications

DataStreamReader — Loading Data from Streaming Data Source 2.1
DataStreamWriter — Writing Datasets To Streaming Data Sinks 2.2
ForeachWriter 2.2.1
OutputMode 2.2.2
Trigger — How Frequently to Check Sources For New Data 2.2.3
StreamingQuery 2.3
Streaming Operators / Streaming Dataset API 2.4
dropDuplicates Operator — Streaming Deduplication 2.4.1
explain Operator — Explaining Query Plan 2.4.2
groupBy Operator — Untyped Streaming Aggregation (with Implicit State Logic)
groupByKey Operator — Streaming Aggregation (with Explicit State Logic) 2.4.3
withWatermark Operator — Event Time Watermark 2.4.5 2.4.4
window Function — Stream Time Windows 2.5
KeyValueGroupedDataset — Streaming Aggregation 2.6
mapGroupsWithState Operator — Stateful Streaming Aggregation (with Explicit State Logic) 2.6.1
flatMapGroupsWithState Operator — Arbitrary Stateful Streaming Aggregation (with Explicit State Logic) 2.6.2
GroupState — State Per Group in Stateful Streaming Aggregation 2.7
GroupStateImpl 2.7.1
GroupStateTimeout 2.7.2
StreamingQueryManager — Streaming Query Management 2.8
Configuration Properties 2.9

Kafka Data Source

KafkaSourceProvider — Data Source Provider for Apache Kafka 3.1
KafkaSource 3.2
KafkaRelation 3.3
KafkaSourceRDD 3.4
CachedKafkaConsumer 3.5
KafkaOffsetReader 3.6
ConsumerStrategy Contract for KafkaConsumer Providers 3.7
KafkaSourceOffset 3.8
KafkaSink 3.9

Text File Data Source

FileStreamSource 4.1
FileStreamSink — Streaming Sink for Parquet Format 4.2

Rate Source

RateSourceProvider 5.1
RateStreamSource 5.2

Text Socket Source

TextSocketSourceProvider 6.1
TextSocketSource 6.2

Memory Data Sink

MemorySink 7.1
MemoryStream 7.2

Console Data Sink

ConsoleSinkProvider 8.1
ConsoleSink for Showing DataFrames to Console 8.2

Query Planning and Execution

StreamExecution — Base of Streaming Query Executions 9.1
MicroBatchExecution 9.1.1
ContinuousExecution 9.1.2
StreamingQueryWrapper 9.1.3
ProgressReporter Contract 9.2
QueryExecutionThread 9.3
TriggerExecutor 9.4
IncrementalExecution — QueryExecution of Streaming Datasets 9.5
StatefulOperatorStateInfo 9.6
StreamingQueryListenerBus — Notification Bus for Streaming Events 9.7

Logical Operators

EventTimeWatermark Unary Logical Operator 10.1
FlatMapGroupsWithState Unary Logical Operator 10.2
Deduplicate Unary Logical Operator 10.3
MemoryPlan Logical Query Plan 10.4
StreamingRelation Leaf Logical Operator for Streaming Source 10.5
StreamingExecutionRelation Leaf Logical Operator for Streaming Source At Execution 10.6

Physical Operators

EventTimeWatermarkExec Unary Physical Operator for Accumulating Event Time Watermark 11.1
EventTimeStatsAccum Accumulator 11.1.1
FlatMapGroupsWithStateExec Unary Physical Operator 11.2
StateStoreRestoreExec Unary Physical Operator — Restoring State of Streaming Aggregates 11.3
StateStoreSaveExec Unary Physical Operator — Saving State of Streaming Aggregates 11.4
Demo: StateStoreSaveExec with Complete Output Mode 11.4.1
Demo: StateStoreSaveExec with Update Output Mode 11.4.2

StreamingDeduplicateExec Unary Physical Operator for Streaming Deduplication 11.5
StreamingRelationExec Leaf Physical Operator 11.6
StreamingSymmetricHashJoinExec 11.7
WatermarkSupport Contract for Streaming Watermark in Unary Physical Operators 11.8

Execution Planning Strategies

FlatMapGroupsWithStateStrategy Execution Planning Strategy for FlatMapGroupsWithState Logical Operator 12.1
StatefulAggregationStrategy Execution Planning Strategy for EventTimeWatermark and Aggregate Logical Operators 12.2
StreamingDeduplicationStrategy Execution Planning Strategy for Deduplicate Logical Operator 12.3
StreamingRelationStrategy Execution Planning Strategy for StreamingRelation and StreamingExecutionRelation Logical Operators 12.4

Offsets and Checkpointing

Offset 13.1
MetadataLog — Contract for Metadata Storage 13.2
HDFSMetadataLog — MetadataLog with Hadoop HDFS for Reliable Storage 13.3
    CommitLog 13.3.1
    CompactibleFileStreamLog 13.3.2
    OffsetSeqLog — HDFSMetadataLog with OffsetSeq Metadata 13.3.3
    OffsetSeq 13.3.4
OffsetSeqMetadata 13.4

Managing State in Stateful Streaming Aggregations

StateStore — Streaming Aggregation State Management 14.1
    StateStoreOps — Implicits Methods for Creating StateStoreRDD 14.1.1
    StateStoreProvider 14.1.2
    StateStoreUpdater 14.1.3

StateStoreWriter — Recording Metrics For Writing to StateStore 14.1.4
HDFSBackedStateStore 14.2
HDFSBackedStateStoreProvider 14.2.1
StateStoreRDD — RDD for Updating State (in StateStores Across Spark Cluster) 14.3
StateStoreCoordinator — Tracking Locations of StateStores for StateStoreRDD 14.4
StateStoreCoordinatorRef Interface for Communication with StateStoreCoordinator 14.4.1

Monitoring

StreamingQueryListener — Intercepting Streaming Events 15.1
StreamingQueryProgress 15.1.1
Web UI 15.2
Logging 15.3

Extending Structured Streaming

DataSource — Pluggable Data Source 16.1
StreamSourceProvider — Streaming Data Source Provider 16.2
Streaming Source 16.3
StreamSinkProvider 16.4
Streaming Sink — Adding Batches of Data to Storage 16.5
ForeachSink 16.5.1

Structured Streaming V2

StreamWriteSupport 17.1

Demos

groupBy Streaming Aggregation with Append Output Mode 18.1
Developing Custom Streaming Sink (and Monitoring SQL Queries in web UI) 18.2
current_timestamp Function For Processing Time in Streaming Queries 18.3
Using StreamingQueryManager for Query Termination Management 18.4

Varia

UnsupportedOperationChecker	19.1
StreamProgress Custom Scala Map	19.2

Spark Structured Streaming (Apache Spark 2.3.1)

Welcome to **Spark Structured Streaming** gitbook!

I'm Jacek Laskowski, an independent consultant, developer and trainer focusing exclusively on **Apache Spark**, Apache Kafka and Kafka Streams (with Scala and sbt on Apache Mesos, Hadoop YARN and DC/OS). I offer courses, workshops, mentoring and software development services.

I lead Warsaw Scala Enthusiasts and Warsaw Spark meetups in Warsaw, Poland.

Contact me at jacek@japila.pl or @jaceklaskowski to discuss Apache Spark and Apache Kafka opportunities, e.g. courses, workshops, mentoring or application development services.

If you like the gitbook you should seriously consider participating in my own, very hands-on, in-depth Apache Spark Workshops and Webinars and in particular brand new and shiny Spark Structured Streaming (Apache Spark 2.2) Workshop.

<table>
<tr>
<td>Note</td>
<td>
If you'd like to participate in Spark Structured Streaming (Apache Spark 2.2) Workshop go to the tweet and like it. That's how I figure the interest in the workshop.

</td>
</tr>
</table>

Spark Structured Streaming gitbook serves as the ultimate place of mine to collect all the nuts and bolts of using Spark Structured Streaming in the most effective way. The notes aim to help me designing and developing better products with Apache Spark. It is also a viable proof of my current understanding of Apache Spark. I do eventually want to reach the highest level of mastery in Apache Spark (as do you!)

The collection of notes serves as **the study material** for my trainings, workshops, videos and courses about Apache Spark. Follow me on twitter @jaceklaskowski for up to date news and to learn about the upcoming events about Apache Spark.

Tip	I'm also writing Mastering Apache Spark 2 (Spark 2.2+) and Mastering Apache Kafka (Kafka 0.11.0.0+).

Expect text and code snippets from Spark's mailing lists, the official documentation of Apache Spark, StackOverflow, blog posts, books from O'Reilly (and other publishers), press releases, conferences, YouTube or Vimeo videos, Quora, the source code of Apache Spark, etc. Attribution follows whenever possible.

Structured Streaming — Streaming Datasets

Structured Streaming is a stream processing engine with a high-level declarative streaming API built on top of Spark SQL allowing for continuous incremental execution of a structured query.

The semantics of the Structured Streaming model is as follows (see the article Structured Streaming In Apache Spark):

> At any time, the output of a continuous application is equivalent to executing a batch job on a prefix of the data.

Note	As of Spark 2.2.0, Structured Streaming has been marked stable and ready for production use. With that the other older streaming module Spark Streaming should *de facto* be considered obsolete and not used for developing new Spark applications.

Structured Streaming attempts to unify streaming, interactive, and batch queries over structured datasets for developing end-to-end stream processing applications dubbed **continuous applications** using Spark SQL's Datasets API with additional support for the following features:

- Continuous streaming aggregations
- Streaming watermark (for state expiration and late events)
- Continuous window aggregations (aka **windowing**) using `groupBy` operator with `window` function
- arbitrary stateful stream aggregation

In Structured Streaming, Spark developers describe custom streaming computations in the same way as with Spark SQL. Internally, Structured Streaming applies the user-defined structured query to the continuously and indefinitely arriving data to analyze real-time streaming data.

With Structured Streaming, Spark 2 aims at simplifying **streaming analytics** with little to no need to reason about effective data streaming (trying to hide the unnecessary complexity in your streaming analytics architectures).

Structured Streaming introduces **streaming Datasets** that are *infinite datasets* with primitives like input streaming data sources and output streaming data sinks.

Tip	A `Dataset` is **streaming** (aka *continuous*) when its logical plan is streaming.
	`val batchQuery = spark.`
	`  read. // <-- batch non-streaming query`
	`  csv("sales")`
	`scala> batchQuery.isStreaming`
	`res0: Boolean = false`
	
	`val streamingQuery = spark.`
	`  readStream. // <-- streaming query`
	`  format("rate").`
	`  load`
	`scala> streamingQuery.isStreaming`
	`res1: Boolean = true`
	
	More information about Spark SQL, Datasets and logical plans is available in Mastering Apache Spark 2.

Structured Streaming models a stream of data as an infinite (and hence continuous) table that could be changed every streaming batch.

You can specify output mode of a streaming dataset which is what gets written to a streaming sink (i.e. the infinite table) when there is new data available.

Streaming Datasets use **streaming query plans** (as opposed to regular batch Datasets that are based on batch query plans).

Note	From this perspective, batch queries can be considered streaming Datasets executed once only (and is why some batch queries, e.g. KafkaSource, can easily work in batch mode).
	`val batchQuery = spark.read.format("rate").load`
	`scala> batchQuery.isStreaming`
	`res0: Boolean = false`
	
	`val streamingQuery = spark.readStream.format("rate").load`
	`scala> streamingQuery.isStreaming`
	`res1: Boolean = true`

```
// The following example executes a streaming query over CSV files
// CSV format requires a schema before you can start the query

// You could build your schema manually
import org.apache.spark.sql.types._
val schema = StructType(
  StructField("id", LongType, false) ::
  StructField("name", StringType, true) ::
  StructField("city", StringType, true) :: Nil)

// ...or using the Schema DSL
val schema = new StructType().
```

```
  add($"long".long.copy(nullable = false)).
  add($"name".string).
  add($"city".string)

// ...but is error-prone and time-consuming, isn't it?

// Use the business object that describes the dataset
case class Person(id: Long, name: String, city: String)

import org.apache.spark.sql.Encoders
val schema = Encoders.product[Person].schema

val people = spark.
  readStream.
  schema(schema).
  csv("in/*.csv").
  as[Person]

// people has this additional capability of being streaming
scala> people.isStreaming
res0: Boolean = true

// ...but it is still a Dataset.
// (Almost) any Dataset operation is available
val population = people.
  groupBy('city).
  agg(count('city) as "population")

// Start the streaming query
// Write the result using console format, i.e. print to the console
// Only Complete output mode supported by groupBy
import scala.concurrent.duration._
import org.apache.spark.sql.streaming.{OutputMode, Trigger}
val populationStream = population.
  writeStream.
  format("console").
  trigger(Trigger.ProcessingTime(30.seconds)).
  outputMode(OutputMode.Complete).
  queryName("textStream").
  start

scala> populationStream.isActive
res1: Boolean = true

scala> populationStream.explain(extended = true)
== Parsed Logical Plan ==
Aggregate [city#112], [city#112, count(city#112) AS population#19L]
+- Relation[id#110L,name#111,city#112] csv

== Analyzed Logical Plan ==
city: string, population: bigint
Aggregate [city#112], [city#112, count(city#112) AS population#19L]
+- Relation[id#110L,name#111,city#112] csv
```

```
== Optimized Logical Plan ==
Aggregate [city#112], [city#112, count(city#112) AS population#19L]
+- Project [city#112]
   +- Relation[id#110L,name#111,city#112] csv

== Physical Plan ==
*HashAggregate(keys=[city#112], functions=[count(city#112)], output=[city#112, populat
ion#19L])
+- Exchange hashpartitioning(city#112, 200)
   +- *HashAggregate(keys=[city#112], functions=[partial_count(city#112)], output=[cit
y#112, count#118L])
      +- *FileScan csv [city#112] Batched: false, Format: CSV, InputPaths: file:/Users
/jacek/dev/oss/spark/in/1.csv, file:/Users/jacek/dev/oss/spark/in/2.csv, file:/Users/j
..., PartitionFilters: [], PushedFilters: [], ReadSchema: struct<city:string>

// Let's query for all active streams
scala> spark.streams.active.foreach(println)
Streaming Query - Population [state = ACTIVE]

// You may eventually want to stop the streaming query
populationStream.stop

scala> populationStream.isActive
res2: Boolean = false
```

Structured streaming is defined by the following data abstractions in `org.apache.spark.sql.streaming` package:

1. StreamingQuery
2. Streaming Source
3. Streaming Sink
4. StreamingQueryManager

Structured Streaming follows micro-batch model and periodically fetches data from the data source (and uses the `DataFrame` data abstraction to represent the fetched data for a certain batch).

With Datasets as Spark SQL's view of structured data, structured streaming checks input sources for new data every trigger (time) and executes the (continuous) queries.

Tip	Structured Streaming was introduced in SPARK-8360 Structured Streaming (aka Streaming DataFrames).

Tip	Read the official programming guide of Spark about Structured Streaming.

Note	The feature has also been called **Streaming Spark SQL Query**, **Streaming DataFrames**, **Continuous DataFrame** or **Continuous Query**. There have been lots of names before the Spark project settled on Structured Streaming.

Example — Streaming Query for Running Counts (over Words from Socket with Output to Console)

Note	The example is "borrowed" from the official documentation of Spark. Changes and errors are only mine.

Tip	You need to run `nc -lk 9999` first before running the example.

```
val lines = spark.readStream
  .format("socket")
  .option("host", "localhost")
  .option("port", 9999)
  .load
  .as[String]

val words = lines.flatMap(_.split("\\W+"))

scala> words.printSchema
root
 |-- value: string (nullable = true)

val counter = words.groupBy("value").count

// nc -lk 9999 is supposed to be up at this point

import org.apache.spark.sql.streaming.OutputMode.Complete
val query = counter.writeStream
  .outputMode(Complete)
  .format("console")
  .start

query.stop
```

Example — Streaming Query over CSV Files with Output to Console Every 5 Seconds

Below you can find a complete example of a streaming query in a form of `DataFrame` of data from `csv-logs` files in `csv` format of a given schema into a ConsoleSink every 5 seconds.

Tip	Copy and paste it to Spark Shell in `:paste` mode to run it.

```
// Explicit schema with nullables false
```

```
import org.apache.spark.sql.types._
val schemaExp = StructType(
  StructField("name", StringType, false) ::
  StructField("city", StringType, true) ::
  StructField("country", StringType, true) ::
  StructField("age", IntegerType, true) ::
  StructField("alive", BooleanType, false) :: Nil
)

// Implicit inferred schema
val schemaImp = spark.read
  .format("csv")
  .option("header", true)
  .option("inferSchema", true)
  .load("csv-logs")
  .schema

val in = spark.readStream
  .schema(schemaImp)
  .format("csv")
  .option("header", true)
  .option("maxFilesPerTrigger", 1)
  .load("csv-logs")

scala> in.printSchema
root
 |-- name: string (nullable = true)
 |-- city: string (nullable = true)
 |-- country: string (nullable = true)
 |-- age: integer (nullable = true)
 |-- alive: boolean (nullable = true)

println("Is the query streaming" + in.isStreaming)

println("Are there any streaming queries?" + spark.streams.active.isEmpty)

import scala.concurrent.duration._
import org.apache.spark.sql.streaming.{OutputMode, Trigger}
val out = in.
  writeStream.
  format("console").
  option("truncate", false).
  trigger(Trigger.ProcessingTime("5 seconds")).
  queryName("consoleStream").
  outputMode(Output.Append).
  start

16/07/13 12:32:11 TRACE FileStreamSource: Listed 3 file(s) in 4.274022 ms
16/07/13 12:32:11 TRACE FileStreamSource: Files are:
        file:///Users/jacek/dev/oss/spark/csv-logs/people-1.csv
        file:///Users/jacek/dev/oss/spark/csv-logs/people-2.csv
        file:///Users/jacek/dev/oss/spark/csv-logs/people-3.csv
16/07/13 12:32:11 DEBUG FileStreamSource: New file: file:///Users/jacek/dev/oss/spark/
```

```
csv-logs/people-1.csv
16/07/13 12:32:11 TRACE FileStreamSource: Number of new files = 3
16/07/13 12:32:11 TRACE FileStreamSource: Number of files selected for batch = 1
16/07/13 12:32:11 TRACE FileStreamSource: Number of seen files = 1
16/07/13 12:32:11 INFO FileStreamSource: Max batch id increased to 0 with 1 new files
16/07/13 12:32:11 INFO FileStreamSource: Processing 1 files from 0:0
16/07/13 12:32:11 TRACE FileStreamSource: Files are:
        file:///Users/jacek/dev/oss/spark/csv-logs/people-1.csv
-------------------------------------------
Batch: 0
-------------------------------------------
+-----+--------+-------+---+-----+
| name|    city|country|age|alive|
+-----+--------+-------+---+-----+
|Jacek|Warszawa| Polska| 42| true|
+-----+--------+-------+---+-----+

spark.streams
  .active
  .foreach(println)
// Streaming Query - consoleStream [state = ACTIVE]

scala> spark.streams.active(0).explain
== Physical Plan ==
*Scan csv [name#130,city#131,country#132,age#133,alive#134] Format: CSV, InputPaths: f
ile:/Users/jacek/dev/oss/spark/csv-logs/people-3.csv, PushedFilters: [], ReadSchema: s
truct<name:string,city:string,country:string,age:int,alive:boolean>
```

Further reading or watching

- (article) Structured Streaming In Apache Spark
- (video) The Future of Real Time in Spark from Spark Summit East 2016 in which Reynold Xin presents the concept of **Streaming DataFrames** to the public
- (video) Structuring Spark: DataFrames, Datasets, and Streaming
- (article) What Spark's Structured Streaming really means
- (video) A Deep Dive Into Structured Streaming by Tathagata "TD" Das from Spark Summit 2016
- (video) Arbitrary Stateful Aggregations in Structured Streaming in Apache Spark by Burak Yavuz

DataStreamReader — Loading Data from Streaming Data Source

`DataStreamReader` is the interface to describe how data is loaded to a streaming `Dataset` from a streaming source.

Table 1. DataStreamReader's Methods

Method	Description
csv	Sets `csv` as the source format `csv(path: String): DataFrame`
format	Sets the format of datasets `format(source: String): DataStreamReader`
json	Sets `json` as the source format `json(path: String): DataFrame`

load	Loads data from a streaming source to a streaming `Dataset`
	`load(): DataFrame`
	`load(path: String): DataFrame`

option	Sets a loading option
	`option(key: String, value: Boolean): DataStreamReader`
	`option(key: String, value: Double): DataStreamReader`
	`option(key: String, value: Long): DataStreamReader`
	`option(key: String, value: String): DataStreamReader`

options	Sets one or more loading options
	`options(options: scala.collection.Map[String, String]): DataStreamReader`
	`options(options: java.util.Map[String, String]): DataStreamReader`

orc	Sets `orc` as the source format

	`orc(path: String): DataFrame`
parquet	Sets `parquet` as the source format `parquet(path: String): DataFrame`

schema	Sets the schema of datasets
	(New in 2.3.0) `schema` that accepts a DDL-formatted table schema, e.g. `a INT, b STRING`.
	`schema(schemaString: String): DataStreamReader`
	`schema(schema: StructType): DataStreamReader`

text	Sets `text` as the source format `text(path: String): DataFrame`
textFile	Returns `Dataset[String]` (not `DataFrame`) `textFile(path: String): Dataset[String]`

Figure 1. DataStreamReader and The Others

`DataStreamReader` is used for a Spark developer to describe how Spark Structured Streaming loads datasets from a streaming source (that in the end creates a logical plan for a streaming query).

Note	`DataStreamReader` is the Spark developer-friendly API to create a StreamingRelation logical operator (that represents a streaming source in a logical plan).

You can access `DataStreamReader` using `SparkSession.readStream` method.

```
import org.apache.spark.sql.SparkSession
val spark: SparkSession = ...

val streamReader = spark.readStream
```

`DataStreamReader` supports many source formats natively and offers the interface to define custom formats:

- json
- csv
- parquet
- text

Note	`DataStreamReader` assumes parquet file format by default that you can change using `spark.sql.sources.default` property.

Note	`hive` source format is not supported.

After you have described the **streaming pipeline** to read datasets from an external streaming data source, you eventually trigger the loading using format-agnostic load or format-specific (e.g. json, csv) operators.

Table 2. DataStreamReader's Internal Properties (in alphabetical order)

Name	Initial Value	Description
`source`	`spark.sql.sources.default` property	Source format of datasets in a streaming data source
`userSpecifiedSchema`	(empty)	Optional user-defined schema
`extraOptions`	(empty)	Collection of key-value configuration options

Specifying Format — `format` Method

```
format(source: String): DataStreamReader
```

`format` specifies the `source` format of datasets in a streaming data source.

Internally, `schema` sets source internal property.

Specifying Schema — `schema` Method

```
schema(schema: StructType): DataStreamReader
schema(schemaString: String): DataStreamReader  (1)
```

1. Uses the input DDL-formatted string

`schema` specifies the `schema` of the streaming data source.

Internally, `schema` sets userSpecifiedSchema internal property.

Specifying Loading Options — `option` Method

```
option(key: String, value: String): DataStreamReader
option(key: String, value: Boolean): DataStreamReader
option(key: String, value: Long): DataStreamReader
option(key: String, value: Double): DataStreamReader
```

`option` family of methods specifies additional options to a streaming data source.

There is support for values of `String`, `Boolean`, `Long`, and `Double` types for user convenience, and internally are converted to `String` type.

Internally, `option` sets extraOptions internal property.

Note	You can also set options in bulk using options method. You have to do the type conversion yourself, though.

Specifying Loading Options — `options` Method

```
options(options: scala.collection.Map[String, String]): DataStreamReader
```

`options` method allows specifying one or many options of the streaming input data source.

Note	You can also set options one by one using option method.

Loading Data From Streaming Source (to Streaming Dataset) — `load` Method

```
load(): DataFrame
load(path: String): DataFrame (1)
```

1. Specifies `path` option before passing the call to parameterless `load()`

`load` loads data from a streaming data source to a streaming dataset.

Internally, `load` first creates a DataSource (using user-specified schema, the name of the source and options) followed by creating a `DataFrame` with a StreamingRelation logical operator (for the `DataSource`).

`load` makes sure that the name of the source is not `hive` . Otherwise, `load` reports a `AnalysisException` .

```
Hive data source can only be used with tables, you can not read files of Hive data sou
rce directly.
```

Built-in Formats

```
json(path: String): DataFrame
csv(path: String): DataFrame
parquet(path: String): DataFrame
text(path: String): DataFrame
textFile(path: String): Dataset[String] (1)
```

1. Returns `Dataset[String]` not `DataFrame`

`DataStreamReader` can load streaming datasets from data sources of the following formats:

- `json`
- `csv`
- `parquet`
- `text`

The methods simply pass calls to format followed by load(path).

DataStreamWriter — Writing Datasets To Streaming Data Sinks

`DataStreamWriter` is the interface to describe how the result of executing a streaming query is written to a streaming sink.

Table 1. DataStreamWriter's Methods

Method	Description
foreach	Sets ForeachWriter in the full control of streaming writes. `foreach(writer: ForeachWriter[T]): DataStreamWriter[T]`
`format`	Specifies the format of the output (which is an output data source and indi specifies the streaming sink to write the rows to) `format(source: String): DataStreamWriter[T]` Internally, `format` is referred to as a *source* (as in the output data *source*) Recognized "special" output data sources (in the code): • `hive` • `memory` • foreach • `console`

option	`option(key: String, value: Boolean): DataStreamWriter[T]`
	`option(key: String, value: Double): DataStreamWriter[T]`
	`option(key: String, value: Long): DataStreamWriter[T]`
	`option(key: String, value: String): DataStreamWriter[T]`

`options`	`options(options: scala.collection.Map[String, String]): DataStreamWrit` `options(options: java.util.Map[String, String]): DataStreamWriter[T]`

outputMode	Specifies the output mode
	`outputMode(outputMode: OutputMode): DataStreamWriter[T]`
	`outputMode(outputMode: String): DataStreamWriter[T]`

<table>
<tr><td><code>partitionBy</code></td><td><pre>partitionBy(colNames: String*): DataStreamWriter[T]</pre></td></tr>
<tr><td>queryName</td><td>Assigns the name of a query<pre>queryName(queryName: String): DataStreamWriter[T]</pre></td></tr>
<tr><td>start</td><td><pre>start(): StreamingQuery
start(path: String): StreamingQuery</pre></td></tr>
<tr><td>trigger</td><td>Sets the Trigger for how often a streaming query should be executed and result saved.<pre>trigger(trigger: Trigger): DataStreamWriter[T]</pre></td></tr>
</table>

Note

A streaming query is a Dataset with a streaming logical plan.

```
scala> spark.version
res0: String = 2.3.0-SNAPSHOT

import org.apache.spark.sql.streaming.Trigger
import scala.concurrent.duration._
import org.apache.spark.sql.DataFrame
val rates: DataFrame = spark.
  readStream.
  format("rate").
  load

scala> rates.isStreaming
res1: Boolean = true

scala> rates.queryExecution.logical.isStreaming
res2: Boolean = true
```

`DataStreamWriter` is available using `writeStream` method of a streaming `Dataset`.

```
import org.apache.spark.sql.streaming.DataStreamWriter
import org.apache.spark.sql.Row

val streamingQuery: Dataset[Long] = ...

scala> streamingQuery.isStreaming
res0: Boolean = true

val writer: DataStreamWriter[Row] = streamingQuery.writeStream
```

Like the batch `DataFrameWriter` , `DataStreamWriter` has a direct support for many file formats and an extension point to plug in new formats.

```
// see above for writer definition

// Save dataset in JSON format
writer.format("json")
```

In the end, you start the actual continuous writing of the result of executing a `Dataset` to a sink using start operator.

```
writer.save
```

Beside the above operators, there are the following to work with a `Dataset` as a whole.

Note	`hive` is not supported for streaming writing (and leads to a `AnalysisException`).

Note	`DataFrameWriter` is responsible for writing in a batch fashion.

Table 2. DataStreamWriter's Internal Properties (in alphabetical order)

Name	Initial Value	Description
`extraOptions`		
`foreachWriter`		
`partitioningColumns`		
`source`		
`outputMode`	`OutputMode.Append`	OutputMode of the streaming sink Set using outputMode method.
`trigger`		

Specifying Write Option — `option` Method

```
option(key: String, value: String): DataStreamWriter[T]
option(key: String, value: Boolean): DataStreamWriter[T]
option(key: String, value: Long): DataStreamWriter[T]
option(key: String, value: Double): DataStreamWriter[T]
```

Internally, `option` adds the `key` and `value` to extraOptions internal option registry.

Specifying Output Mode — `outputMode` Method

```
outputMode(outputMode: String): DataStreamWriter[T]
outputMode(outputMode: OutputMode): DataStreamWriter[T]
```

`outputMode` specifies the output mode of a streaming `Dataset`.

Note	When unspecified explicitly, Append output mode is the default.

`outputMode` can be a name or typed `OutputMode`.

Note	Output mode describes what data is written to a streaming sink when there is new data available in streaming data sources.

Setting Query Name — `queryName` method

```
queryName(queryName: String): DataStreamWriter[T]
```

`queryName` sets the name of a streaming query.

Internally, it is just an additional option with the key `queryName`.

Setting How Often to Execute Streaming Query — `trigger` method

```
trigger(trigger: Trigger): DataStreamWriter[T]
```

`trigger` method sets the time interval of the **trigger** (that executes a batch runner) for a streaming query.

Note	`Trigger` specifies how often results should be produced by a StreamingQuery. See Trigger.

The default trigger is ProcessingTime(0L) that runs a streaming query as often as possible.

Tip	Consult Trigger to learn about `Trigger` and `ProcessingTime` types.

Starting Continuous Writing to Sink — `start` Method

```
start(): StreamingQuery
start(path: String): StreamingQuery  (1)
```

1. Sets `path` option to `path` and passes the call on to `start()`

`start` starts a streaming query.

`start` gives a StreamingQuery to control the execution of the continuous query.

Note	Whether or not you have to specify `path` option depends on the streaming sink in use.

Internally, `start` branches off per `source`.

- `memory`
- `foreach`
- other formats

…FIXME

Table 3. start's Options

Option	Description
`queryName`	Name of active streaming query
`checkpointLocation`	Directory for checkpointing (and to store query metadata like offsets before and after being processed, the query id, etc.)

`start` reports a `AnalysisException` when `source` is `hive`.

```
val q =  spark.
  readStream.
  text("server-logs/*").
  writeStream.
  format("hive") <-- hive format used as a streaming sink
scala> q.start
org.apache.spark.sql.AnalysisException: Hive data source can only be used with tables,
 you can not write files of Hive data source directly.;
  at org.apache.spark.sql.streaming.DataStreamWriter.start(DataStreamWriter.scala:234)
  ... 48 elided
```

Note	Define options using option or options methods.

Making ForeachWriter in Charge of Streaming Writes — `foreach` method

```
foreach(writer: ForeachWriter[T]): DataStreamWriter[T]
```

`foreach` sets the input ForeachWriter to be in control of streaming writes.

Internally, `foreach` sets the streaming output format as `foreach` and `foreachWriter` as the input `writer`.

Note	`foreach` uses `SparkSession` to access `SparkContext` to clean the `ForeachWriter`.

Note	`foreach` reports an `IllegalArgumentException` when `writer` is `null`. `foreach writer cannot be null`

ForeachWriter

`ForeachWriter` is the contract for a **foreach writer** that is a streaming format that controls streaming writes.

Note	`ForeachWriter` is set using foreach operator.

```
val foreachWriter = new ForeachWriter[String] { ... }
streamingQuery.
  writeStream.
  foreach(foreachWriter).
  start
```

ForeachWriter Contract

```
package org.apache.spark.sql

abstract class ForeachWriter[T] {
  def open(partitionId: Long, version: Long): Boolean
  def process(value: T): Unit
  def close(errorOrNull: Throwable): Unit
}
```

Table 1. ForeachWriter Contract

Method	Description
`open`	Used when…
`process`	Used when…
`close`	Used when…

OutputMode

Output mode (`OutputMode`) describes what data is written to a streaming sink when there is new data available in streaming data sources (in a trigger / streaming batch).

Output mode of a streaming query is specified using outputMode method of `DataStreamWriter` .

```
val inputStream = spark.
  readStream.
  format("rate").
  load
import org.apache.spark.sql.streaming.{OutputMode, Trigger}
import scala.concurrent.duration._
val consoleOutput = inputStream.
  writeStream.
  format("console").
  option("truncate", false).
  trigger(Trigger.ProcessingTime(10.seconds)).
  queryName("rate-console").
  option("checkpointLocation", "checkpoint").
  outputMode(OutputMode.Update).  // <-- update output mode
  start
```

Table 1. Available Output Modes

<table>
<tr><th>OutputMode</th><th>Name</th><th>Behaviour</th></tr>
<tr><td><code>Append</code></td><td><code>append</code></td><td>Default output mode that writes "new" rows only.

Note: For streaming aggregations, "new" row is when the intermediate state becomes final, i.e. when new events for the grouping key can only be considered late which is when watermark moves past the event time of the key.

Note: <code>Append</code> output mode requires that a streaming query defines event time watermark (using withWatermark operator) on the event time column that is used in aggregation (directly or using window function).

Required for datasets with <code>FileFormat</code> format (to create FileStreamSink)

Used for flatMapGroupsWithState operator

Note: <code>Append</code> is mandatory when multiple <code>flatMapGroupsWithState</code> operators are used in a structured query.</td></tr>
<tr><td><code>Complete</code></td><td><code>complete</code></td><td>Writes all rows (every time there are updates) and therefore corresponds to a traditional batch query.

Note: Supported only for streaming queries with groupBy or groupByKey aggregations (as asserted by UnsupportedOperationChecker).</td></tr>
<tr><td><code>Update</code></td><td><code>update</code></td><td>Write the rows that were updated (every time there are updates). If the query does not contain aggregations, it is equivalent to Append mode.

Used for mapGroupsWithState and flatMapGroupsWithState operators</td></tr>
</table>

Trigger — How Frequently to Check Sources For New Data

`Trigger` defines how frequently a streaming query should be executed and therefore emit a new data (which `StreamExecution` uses to resolve a TriggerExecutor).

Note	A trigger can also be called a **batch interval** (as in the older Spark Streaming).

Table 1. Triggers

Trigger	Creating Instance
`ContinuousTrigger`	
ProcessingTime	`Trigger.ProcessingTime(long intervalMs)`
	`Trigger.ProcessingTime(Duration interval)`
	`Trigger.ProcessingTime(String interval)`
`OneTimeTrigger`	

Note	You specify the trigger for a streaming query using `DataStreamWriter` 's trigger method.

```
import org.apache.spark.sql.streaming.Trigger
val query = spark.
  readStream.
  format("rate").
  load.
  writeStream.
  format("console").
  option("truncate", false).
  trigger(Trigger.Once). // <-- execute once and stop
  queryName("rate-once").
  start

scala> query.isActive
res0: Boolean = false

scala> println(query.lastProgress)
{
  "id" : "2ae4b0a4-434f-4ca7-a523-4e859c07175b",
  "runId" : "24039ce5-906c-4f90-b6e7-bbb3ec38a1f5",
  "name" : "rate-once",
  "timestamp" : "2017-07-04T18:39:35.998Z",
  "numInputRows" : 0,
  "processedRowsPerSecond" : 0.0,
  "durationMs" : {
    "addBatch" : 1365,
    "getBatch" : 29,
    "getOffset" : 0,
    "queryPlanning" : 285,
    "triggerExecution" : 1742,
    "walCommit" : 40
  },
  "stateOperators" : [ ],
  "sources" : [ {
    "description" : "RateSource[rowsPerSecond=1, rampUpTimeSeconds=0, numPartitions=8]"
,
    "startOffset" : null,
    "endOffset" : 0,
    "numInputRows" : 0,
    "processedRowsPerSecond" : 0.0
  } ],
  "sink" : {
    "description" : "org.apache.spark.sql.execution.streaming.ConsoleSink@7dbf277"
  }
}
```

Note	Although `Trigger` allows for custom implementations, `StreamExecution` refuses such attempts and reports an `IllegalStateException`.

```
case object MyTrigger extends Trigger
scala> val query = spark.
  readStream.
  format("rate").
  load.
  writeStream.
  format("console").
  option("truncate", false).
  trigger(MyTrigger). // <-- use custom trigger
  queryName("rate-once").
  start
java.lang.IllegalStateException: Unknown type of trigger: MyTrigger
  at org.apache.spark.sql.execution.streaming.StreamExecution.<init>(StreamExecution.s
cala:178)
  at org.apache.spark.sql.streaming.StreamingQueryManager.createQuery(StreamingQueryMa
nager.scala:240)
  at org.apache.spark.sql.streaming.StreamingQueryManager.startQuery(StreamingQueryMan
ager.scala:278)
  at org.apache.spark.sql.streaming.DataStreamWriter.start(DataStreamWriter.scala:284)
  ... 57 elided
```

Note	`Trigger` was introduced in the commit for [SPARK-14176][SQL] Add DataFrameWriter.trigger to set the stream batch period.

ProcessingTime

`ProcessingTime` is a `Trigger` that assumes that milliseconds is the minimum time unit.

You can create an instance of `ProcessingTime` using the following constructors:

- `ProcessingTime(Long)` that accepts non-negative values that represent milliseconds.

  ```
  ProcessingTime(10)
  ```

- `ProcessingTime(interval: String)` or `ProcessingTime.create(interval: String)` that accept `CalendarInterval` instances with or without leading `interval` string.

  ```
  ProcessingTime("10 milliseconds")
  ProcessingTime("interval 10 milliseconds")
  ```

- `ProcessingTime(Duration)` that accepts `scala.concurrent.duration.Duration` instances.

  ```
  ProcessingTime(10.seconds)
  ```

- `ProcessingTime.create(interval: Long, unit: TimeUnit)` for `Long` and `java.util.concurrent.TimeUnit` instances.

  ```
  ProcessingTime.create(10, TimeUnit.SECONDS)
  ```

StreamingQuery

`StreamingQuery` is the contract for a streaming query that is executed continuously and concurrently (i.e. on a separate thread).

Note	`StreamingQuery` is called **continuous query** or **stream query**.

Note	`StreamingQuery` is a Scala trait with the only implementation being StreamExecution (and less importanly `StreamingQueryWrapper` for serializing non-serializable `StreamExecution`).

`StreamingQuery` can be in two states:

- active (started)
- inactive (stopped)

If inactive, `StreamingQuery` may have transitioned into the state due to an `StreamingQueryException` (that is available under `exception`).

`StreamingQuery` tracks current state of all the sources, i.e. `SourceStatus` , as `sourceStatuses` .

There could only be a single Sink for a `StreamingQuery` with many Sources.

`StreamingQuery` can be stopped by `stop` or an exception.

StreamingQuery Contract

```
package org.apache.spark.sql.streaming

trait StreamingQuery {
  def name: String
  def id: UUID
  def runId: UUID
  def sparkSession: SparkSession
  def isActive: Boolean
  def exception: Option[StreamingQueryException]
  def status: StreamingQueryStatus
  def recentProgress: Array[StreamingQueryProgress]
  def lastProgress: StreamingQueryProgress
  def awaitTermination(): Unit
  def awaitTermination(timeoutMs: Long): Boolean
  def processAllAvailable(): Unit
  def stop(): Unit
  def explain(): Unit
  def explain(extended: Boolean): Unit
}
```

Table 1. StreamingQuery Contract

Method	Description
`name`	Optional query name that is unique across all active queries
`id`	Unique identifier of a streaming query
`runId`	Unique identifier of the current execution of a streaming query
`sparkSession`	`SparkSession`
`isActive`	Boolean
`exception`	`StreamingQueryException` if the query has finished due to an exception
`status`	`StreamingQueryStatus` (as `StreamExecution` has accumulated being a `ProgressReporter` while running a streaming query)
`recentProgress`	Collection of recent StreamingQueryProgress updates.
`lastProgress`	The last StreamingQueryProgress update.
`awaitTermination`	
`processAllAvailable`	Wait until there are no data available in sources or the query has been terminated.
`stop`	
`explain`	

Streaming Operators / Streaming Dataset API

Dataset API has a set of operators that are of particular use in Apache Spark's Structured Streaming that together constitute so-called **Streaming Dataset API**.

Table 1. Streaming Operators

<table>
<tr><th>Operator</th><th>Description</th></tr>
<tr><td>dropDuplicates</td><td>Drops duplicate records (given a subset of columns)<pre>dropDuplicates(): Dataset[T]
dropDuplicates(colNames: Seq[String]): Dataset[T]
dropDuplicates(col1: String, cols: String*): Dataset[T]</pre></td></tr>
<tr><td>explain</td><td>Explains execution plans<pre>explain(): Unit
explain(extended: Boolean): Unit</pre></td></tr>
<tr><td>groupBy</td><td>Aggregates rows by a untyped grouping function<pre>groupBy(cols: Column*): RelationalGroupedDataset
groupBy(col1: String, cols: String*): RelationalGroupedDataset</pre></td></tr>
<tr><td>groupByKey</td><td>Aggregates rows by a typed grouping function<pre>groupByKey(func: T => K): KeyValueGroupedDataset[K, T]</pre></td></tr>
<tr><td>withWatermark</td><td>Defines a streaming watermark on a column<pre>withWatermark(eventTime: String, delayThreshold: String): Dataset[T]</pre></td></tr>
</table>

```
scala> spark.version
res0: String = 2.3.0-SNAPSHOT

// input stream
val rates = spark.
  readStream.
  format("rate").
  option("rowsPerSecond", 1).
  load

// stream processing
// replace [operator] with the operator of your choice
rates.[operator]

// output stream
import org.apache.spark.sql.streaming.{OutputMode, Trigger}
import scala.concurrent.duration._
val sq = rates.
  writeStream.
  format("console").
  option("truncate", false).
  trigger(Trigger.ProcessingTime(10.seconds)).
  outputMode(OutputMode.Complete).
  queryName("rate-console").
  start

// eventually...
sq.stop
```

dropDuplicates Operator — Streaming Deduplication

```
dropDuplicates(): Dataset[T]
dropDuplicates(colNames: Seq[String]): Dataset[T]
dropDuplicates(col1: String, cols: String*): Dataset[T]
```

`dropDuplicates` operator…FIXME

Note	For a streaming Dataset, `dropDuplicates` will keep all data across triggers as intermediate state to drop duplicates rows. You can use withWatermark operator to limit how late the duplicate data can be and system will accordingly limit the state. In addition, too late data older than watermark will be dropped to avoid any possibility of duplicates.

```
scala> spark.version
res0: String = 2.3.0-SNAPSHOT

// Start a streaming query
// Using old-fashioned MemoryStream (with the deprecated SQLContext)
import org.apache.spark.sql.execution.streaming.MemoryStream
import org.apache.spark.sql.SQLContext
implicit val sqlContext: SQLContext = spark.sqlContext
val source = MemoryStream[(Int, Int)]
val ids = source.toDS.toDF("time", "id").
  withColumn("time", $"time" cast "timestamp"). // <-- convert time column from Int to
 Timestamp
  dropDuplicates("id").
  withColumn("time", $"time" cast "long")  // <-- convert time column back from Timest
amp to Int

// Conversions are only for display purposes
// Internally we need timestamps for watermark to work
// Displaying timestamps could be too much for such a simple task

scala> println(ids.queryExecution.analyzed.numberedTreeString)
00 Project [cast(time#10 as bigint) AS time#15L, id#6]
01 +- Deduplicate [id#6], true
02    +- Project [cast(time#5 as timestamp) AS time#10, id#6]
03       +- Project [_1#2 AS time#5, _2#3 AS id#6]
04          +- StreamingExecutionRelation MemoryStream[_1#2,_2#3], [_1#2, _2#3]

import org.apache.spark.sql.streaming.{OutputMode, Trigger}
import scala.concurrent.duration._
val q = ids.
  writeStream.
  format("memory").
```

```
  queryName("dups").
  outputMode(OutputMode.Append).
  trigger(Trigger.ProcessingTime(30.seconds)).
  option("checkpointLocation", "checkpoint-dir"). // <-- use checkpointing to save sta
te between restarts
  start

// Publish duplicate records
source.addData(1 -> 1)
source.addData(2 -> 1)
source.addData(3 -> 1)

q.processAllAvailable()

// Check out how dropDuplicates removes duplicates
// --> per single streaming batch (easy)
scala> spark.table("dups").show
+----+---+
|time| id|
+----+---+
|   1|  1|
+----+---+

source.addData(4 -> 1)
source.addData(5 -> 2)

// --> across streaming batches (harder)
scala> spark.table("dups").show
+----+---+
|time| id|
+----+---+
|   1|  1|
|   5|  2|
+----+---+

// Check out the internal state
scala> println(q.lastProgress.stateOperators(0).prettyJson)
{
  "numRowsTotal" : 2,
  "numRowsUpdated" : 1,
  "memoryUsedBytes" : 17751
}

// You could use web UI's SQL tab instead
// Use Details for Query

source.addData(6 -> 2)

scala> spark.table("dups").show
+----+---+
|time| id|
+----+---+
|   1|  1|
```

```
|   5|  2|
+----+---+

// Check out the internal state
scala> println(q.lastProgress.stateOperators(0).prettyJson)
{
  "numRowsTotal" : 2,
  "numRowsUpdated" : 0,
  "memoryUsedBytes" : 17751
}

// Restart the streaming query
q.stop

val q = ids.
  writeStream.
  format("memory").
  queryName("dups").
  outputMode(OutputMode.Complete).  // <-- memory sink supports checkpointing for Comp
lete output mode only
  trigger(Trigger.ProcessingTime(30.seconds)).
  option("checkpointLocation", "checkpoint-dir"). // <-- use checkpointing to save sta
te between restarts
  start

// Doh! MemorySink is fine, but Complete is only available with a streaming aggregation

// Answer it if you know why --> https://stackoverflow.com/q/45756997/1305344

// It's a high time to work on https://issues.apache.org/jira/browse/SPARK-21667
// to understand the low-level details (and the reason, it seems)

// Disabling operation checks and starting over
// ./bin/spark-shell -c spark.sql.streaming.unsupportedOperationCheck=false
// it works now --> no exception!

scala> spark.table("dups").show
+----+---+
|time| id|
+----+---+
+----+---+

source.addData(0 -> 1)
// wait till the batch is triggered
scala> spark.table("dups").show
+----+---+
|time| id|
+----+---+
|   0|  1|
+----+---+

source.addData(1 -> 1)
source.addData(2 -> 1)
```

```
// wait till the batch is triggered
scala> spark.table("dups").show
+----+---+
|time| id|
+----+---+
+----+---+

// What?! No rows?! It doesn't look as if it worked fine :(

// Use groupBy to pass the requirement of having streaming aggregation for Complete ou
tput mode
val counts = ids.groupBy("id").agg(first($"time") as "first_time")
scala> counts.explain
== Physical Plan ==
*HashAggregate(keys=[id#246], functions=[first(time#255L, false)])
+- StateStoreSave [id#246], StatefulOperatorStateInfo(<unknown>,3585583b-42d7-4547-8d62
-255581c48275,0,0), Append, 0
   +- *HashAggregate(keys=[id#246], functions=[merge_first(time#255L, false)])
      +- StateStoreRestore [id#246], StatefulOperatorStateInfo(<unknown>,3585583b-42d7
-4547-8d62-255581c48275,0,0)
         +- *HashAggregate(keys=[id#246], functions=[merge_first(time#255L, false)])
            +- *HashAggregate(keys=[id#246], functions=[partial_first(time#255L, false
)])
               +- *Project [cast(time#250 as bigint) AS time#255L, id#246]
                  +- StreamingDeduplicate [id#246], StatefulOperatorStateInfo(<unknown
>,3585583b-42d7-4547-8d62-255581c48275,1,0), 0
                     +- Exchange hashpartitioning(id#246, 200)
                        +- *Project [cast(_1#242 as timestamp) AS time#250, _2#243 AS
id#246]
                           +- StreamingRelation MemoryStream[_1#242,_2#243], [_1#242,
_2#243]
val q = counts.
  writeStream.
  format("memory").
  queryName("dups").
  outputMode(OutputMode.Complete).  // <-- memory sink supports checkpointing for Comp
lete output mode only
  trigger(Trigger.ProcessingTime(30.seconds)).
  option("checkpointLocation", "checkpoint-dir"). // <-- use checkpointing to save sta
te between restarts
  start

source.addData(0 -> 1)
source.addData(1 -> 1)
// wait till the batch is triggered
scala> spark.table("dups").show
+---+----------+
| id|first_time|
+---+----------+
|  1|         0|
+---+----------+

// Publish duplicates
```

```
// Check out how dropDuplicates removes duplicates

// Stop the streaming query
// Specify event time watermark to remove old duplicates
```

explain Operator — Explaining Query Plan

```
explain(): Unit (1)
explain(extended: Boolean): Unit
```

1. Calls `explain` with `extended` flag disabled

`explain` prints the logical and (with `extended` flag enabled) physical plans to the console.

```
val records = spark.
  readStream.
  format("rate").
  load
scala> records.explain
== Physical Plan ==
StreamingRelation rate, [timestamp#0, value#1L]

scala> records.explain(extended = true)
== Parsed Logical Plan ==
StreamingRelation DataSource(org.apache.spark.sql.SparkSession@4071aa13,rate,List(),No
ne,List(),None,Map(),None), rate, [timestamp#0, value#1L]

== Analyzed Logical Plan ==
timestamp: timestamp, value: bigint
StreamingRelation DataSource(org.apache.spark.sql.SparkSession@4071aa13,rate,List(),No
ne,List(),None,Map(),None), rate, [timestamp#0, value#1L]

== Optimized Logical Plan ==
StreamingRelation DataSource(org.apache.spark.sql.SparkSession@4071aa13,rate,List(),No
ne,List(),None,Map(),None), rate, [timestamp#0, value#1L]

== Physical Plan ==
StreamingRelation rate, [timestamp#0, value#1L]
```

Internally, `explain` creates a `ExplainCommand` runnable command with the logical plan and `extended` flag.

`explain` then executes the plan with `ExplainCommand` runnable command and collects the results that are printed out to the standard output.

Note	`explain` uses `SparkSession` to access the current `SessionState` to execute the plan.
	
	`import org.apache.spark.sql.execution.command.ExplainCommand`
	`val explain = ExplainCommand(...)`
	`spark.sessionState.executePlan(explain)`

For streaming Datasets, `ExplainCommand` command simply creates a IncrementalExecution for the `SparkSession` and the logical plan.

Note	For the purpose of `explain`, `IncrementalExecution` is created with the output mode `Append`, checkpoint location `<unknown>`, run id a random number, current batch id `0` and offset metadata empty. They do not really matter when explaining the load-part of a streaming query.

groupBy Operator — Untyped Streaming Aggregation (with Implicit State Logic)

```
groupBy(cols: Column*): RelationalGroupedDataset
groupBy(col1: String, cols: String*): RelationalGroupedDataset
```

`groupBy` operator…FIXME

```
scala> spark.version
res0: String = 2.3.0-SNAPSHOT

// Since I'm with SNAPSHOT
// Remember to remove ~/.ivy2/cache/org.apache.spark
// Make sure that ~/.ivy2/jars/org.apache.spark_spark-sql-kafka-0-10_2.11-2.3.0-SNAPSH
OT.jar is the latest
// Start spark-shell as follows
/**
./bin/spark-shell --packages org.apache.spark:spark-sql-kafka-0-10_2.11:2.3.0-SNAPSHOT
*/

val fromTopic1 = spark.
  readStream.
  format("kafka").
  option("subscribe", "topic1").
  option("kafka.bootstrap.servers", "localhost:9092").
  load

// extract event time et al
// time,key,value
/*
2017-08-23T00:00:00.002Z,1,now
2017-08-23T00:05:00.002Z,1,5 mins later
2017-08-23T00:09:00.002Z,1,9 mins later
2017-08-23T00:11:00.002Z,1,11 mins later
2017-08-23T01:00:00.002Z,1,1 hour later
// late event = watermark should be (1 hour - 10 minutes) already
2017-08-23T00:49:59.002Z,1,==> SHOULD NOT BE INCLUDED in aggregation as too late <==

CAUTION: FIXME SHOULD NOT BE INCLUDED is included contrary to my understanding?!
*/
val timedValues = fromTopic1.
  select('value cast "string").
  withColumn("tokens", split('value, ",")).
  withColumn("time", to_timestamp('tokens(0))).
  withColumn("key", 'tokens(1) cast "int").
  withColumn("value", 'tokens(2)).
  select("time", "key", "value")
```

```
// aggregation with watermark
val counts = timedValues.
  withWatermark("time", "10 minutes").
  groupBy("key").
  agg(collect_list('value) as "values", collect_list('time) as "times")

// Note that StatefulOperatorStateInfo is mostly generic
// since no batch-specific values are currently available
// only after the first streaming batch
scala> counts.explain
== Physical Plan ==
ObjectHashAggregate(keys=[key#27], functions=[collect_list(value#33, 0, 0), collect_li
st(time#22-T600000ms, 0, 0)])
+- Exchange hashpartitioning(key#27, 200)
   +- StateStoreSave [key#27], StatefulOperatorStateInfo(<unknown>,25149816-1f14-4901-
af13-896286a26d42,0,0), Append, 0
      +- ObjectHashAggregate(keys=[key#27], functions=[merge_collect_list(value#33, 0,
0), merge_collect_list(time#22-T600000ms, 0, 0)])
         +- Exchange hashpartitioning(key#27, 200)
            +- StateStoreRestore [key#27], StatefulOperatorStateInfo(<unknown>,25149816
-1f14-4901-af13-896286a26d42,0,0)
               +- ObjectHashAggregate(keys=[key#27], functions=[merge_collect_list(val
ue#33, 0, 0), merge_collect_list(time#22-T600000ms, 0, 0)])
                  +- Exchange hashpartitioning(key#27, 200)
                     +- ObjectHashAggregate(keys=[key#27], functions=[partial_collect_
list(value#33, 0, 0), partial_collect_list(time#22-T600000ms, 0, 0)])
                        +- EventTimeWatermark time#22: timestamp, interval 10 minutes
                           +- *Project [cast(split(cast(value#1 as string), ,)[0] as t
imestamp) AS time#22, cast(split(cast(value#1 as string), ,)[1] as int) AS key#27, spl
it(cast(value#1 as string), ,)[2] AS value#33]
                              +- StreamingRelation kafka, [key#0, value#1, topic#2, pa
rtition#3, offset#4L, timestamp#5, timestampType#6]

import org.apache.spark.sql.streaming._
import scala.concurrent.duration._
val sq = counts.writeStream.
  format("console").
  option("truncate", false).
  trigger(Trigger.ProcessingTime(30.seconds)).
  outputMode(OutputMode.Update).  // <-- only Update or Complete acceptable because of
 groupBy aggregation
  start

// After StreamingQuery was started,
// the physical plan is complete (with batch-specific values)
scala> sq.explain
== Physical Plan ==
ObjectHashAggregate(keys=[key#27], functions=[collect_list(value#33, 0, 0), collect_li
st(time#22-T600000ms, 0, 0)])
+- Exchange hashpartitioning(key#27, 200)
   +- StateStoreSave [key#27], StatefulOperatorStateInfo(file:/private/var/folders/0w/
kb0d3rqn4zb9fcc91pxhgn8w0000gn/T/temporary-635d6519-b6ca-4686-9b6b-5db0e83cfd51/state,
```

```
855cec1c-25dc-4a86-ae54-c6cdd4ed02ec,0,0), Update, 0
      +- ObjectHashAggregate(keys=[key#27], functions=[merge_collect_list(value#33, 0,
0), merge_collect_list(time#22-T600000ms, 0, 0)])
         +- Exchange hashpartitioning(key#27, 200)
            +- StateStoreRestore [key#27], StatefulOperatorStateInfo(file:/private/var
/folders/0w/kb0d3rqn4zb9fcc91pxhgn8w0000gn/T/temporary-635d6519-b6ca-4686-9b6b-5db0e83
cfd51/state,855cec1c-25dc-4a86-ae54-c6cdd4ed02ec,0,0)
               +- ObjectHashAggregate(keys=[key#27], functions=[merge_collect_list(val
ue#33, 0, 0), merge_collect_list(time#22-T600000ms, 0, 0)])
                  +- Exchange hashpartitioning(key#27, 200)
                     +- ObjectHashAggregate(keys=[key#27], functions=[partial_collect_
list(value#33, 0, 0), partial_collect_list(time#22-T600000ms, 0, 0)])
                        +- EventTimeWatermark time#22: timestamp, interval 10 minutes
                           +- *Project [cast(split(cast(value#76 as string), ,)[0] as
timestamp) AS time#22, cast(split(cast(value#76 as string), ,)[1] as int) AS key#27, s
plit(cast(value#76 as string), ,)[2] AS value#33]
                              +- Scan ExistingRDD[key#75,value#76,topic#77,partition#78
,offset#79L,timestamp#80,timestampType#81]
```

groupByKey Operator — Streaming Aggregation (with Explicit State Logic)

```
groupByKey[K: Encoder](func: T => K): KeyValueGroupedDataset[K, T]
```

`groupByKey` operator is used to combine rows (of type `T`) into KeyValueGroupedDataset with the keys (of type `K`) being generated by a `func` key-generating function and the values collections of one or more rows associated with a key.

`groupByKey` uses a `func` function that takes a row (of type `T`) and gives the group key (of type `K`) the row is associated with.

```
func: T => K
```

Note	The type of the input argument of `func` is the type of rows in the Dataset (i.e. `Dataset[T]`).

`groupByKey` might group together customer orders from the same postal code (wherein the "key" would be the postal code of each individual order, and the "value" would be the order itself).

The following example code shows how to apply `groupByKey` operator to a structured stream of timestamped values of different devices.

```
scala> spark.version
res0: String = 2.3.0-SNAPSHOT

// input stream
import java.sql.Timestamp
val signals = spark.
  readStream.
  format("rate").
  option("rowsPerSecond", 1).
  load.
  withColumn("value", $"value" % 10)  // <-- randomize the values (just for fun)
  withColumn("deviceId", lit(util.Random.nextInt(10))). // <-- 10 devices randomly ass
igned to values
  as[(Timestamp, Long, Int)] // <-- convert to a "better" type (from "unpleasant" Row)

// stream processing using groupByKey operator
// groupByKey(func: ((Timestamp, Long, Int)) => K): KeyValueGroupedDataset[K, (Timesta
mp, Long, Int)]
// K becomes Int which is a device id
val deviceId: ((Timestamp, Long, Int)) => Int = { case (_, _, deviceId) => deviceId }
scala> val signalsByDevice = signals.groupByKey(deviceId)
signalsByDevice: org.apache.spark.sql.KeyValueGroupedDataset[Int,(java.sql.Timestamp,
Long, Int)] = org.apache.spark.sql.KeyValueGroupedDataset@19d40bc6
```

Internally, creates a KeyValueGroupedDataset with the following:

- Encoders for `K` keys and `T` rows
- `QueryExecution` for `AppendColumns` unary logical operator with the `func` function and the analyzed logical plan of the Dataset (`groupBy` is executed on)
- Grouping attributes

Credits

- The example with customer orders and postal codes is borrowed from Apache Beam's Using GroupByKey Programming Guide.

withWatermark Operator — Event Time Watermark

```
withWatermark(eventTime: String, delayThreshold: String): Dataset[T]
```

`withWatermark` specifies the `eventTime` column for **event time watermark** and `delayThreshold` for **event lateness**.

`eventTime` specifies the column to use for watermark and can be either part of `Dataset` from the source or custom-generated using `current_time` or `current_timestamp` functions.

Note	**Watermark** tracks a point in time before which it is assumed no more late events are supposed to arrive (and if they have, the late events are considered really late and simply dropped).

Note	Spark Structured Streaming uses watermark for the following: • To know when a given time window aggregation (using groupBy operator with window function) can be finalized and thus emitted when using output modes that do not allow updates, like Append output mode. • To minimize the amount of state that we need to keep for ongoing aggregations, e.g. mapGroupsWithState (for implicit state management), flatMapGroupsWithState (for user-defined state management) and dropDuplicates operators.

The **current watermark** is computed by looking at the maximum `eventTime` seen across all of the partitions in a query minus a user-specified `delayThreshold`. Due to the cost of coordinating this value across partitions, the actual watermark used is only guaranteed to be at least `delayThreshold` behind the actual event time.

Note	In some cases Spark may still process records that arrive more than `delayThreshold` late.

window Function — Stream Time Windows

`window` is a standard function that generates **tumbling**, **sliding** or **delayed** stream time window ranges (on a timestamp column).

```
window(
  timeColumn: Column,
  windowDuration: String): Column  (1)
window(
  timeColumn: Column,
  windowDuration: String,
  slideDuration: String): Column   (2)
window(
  timeColumn: Column,
  windowDuration: String,
  slideDuration: String,
  startTime: String): Column       (3)
```

1. Creates a tumbling time window with `slideDuration` as `windowDuration` and `0 second` for `startTime`
2. Creates a sliding time window with `0 second` for `startTime`
3. Creates a delayed time window

Note	From Tumbling Window (Azure Stream Analytics): > **Tumbling windows** are a series of fixed-sized, non-overlapping and contiguous time intervals.

Note	From Introducing Stream Windows in Apache Flink: > **Tumbling windows** group elements of a stream into finite sets where each set corresponds to an interval. > **Tumbling windows** discretize a stream into non-overlapping windows.

```
scala> val timeColumn = window($"time", "5 seconds")
timeColumn: org.apache.spark.sql.Column = timewindow(time, 5000000, 5000000, 0) AS `wi
ndow`
```

`timeColumn` should be of `TimestampType`, i.e. with java.sql.Timestamp values.

Tip	Use java.sql.Timestamp.from or java.sql.Timestamp.valueOf factory methods to create `Timestamp` instances.

```
// https://docs.oracle.com/javase/8/docs/api/java/time/LocalDateTime.html
import java.time.LocalDateTime
// https://docs.oracle.com/javase/8/docs/api/java/sql/Timestamp.html
import java.sql.Timestamp
val levels = Seq(
  // (year, month, dayOfMonth, hour, minute, second)
  ((2012, 12, 12, 12, 12, 12), 5),
  ((2012, 12, 12, 12, 12, 14), 9),
  ((2012, 12, 12, 13, 13, 14), 4),
  ((2016, 8,  13, 0, 0, 0), 10),
  ((2017, 5,  27, 0, 0, 0), 15)).
  map { case ((yy, mm, dd, h, m, s), a) => (LocalDateTime.of(yy, mm, dd, h, m, s), a)
}.
  map { case (ts, a) => (Timestamp.valueOf(ts), a) }.
  toDF("time", "level")
scala> levels.show
+-------------------+-----+
|               time|level|
+-------------------+-----+
|2012-12-12 12:12:12|    5|
|2012-12-12 12:12:14|    9|
|2012-12-12 13:13:14|    4|
|2016-08-13 00:00:00|   10|
|2017-05-27 00:00:00|   15|
+-------------------+-----+

val q = levels.select(window($"time", "5 seconds"), $"level")
scala> q.show(truncate = false)
+---------------------------------------------+-----+
|window                                       |level|
+---------------------------------------------+-----+
|[2012-12-12 12:12:10.0,2012-12-12 12:12:15.0]|5    |
|[2012-12-12 12:12:10.0,2012-12-12 12:12:15.0]|9    |
|[2012-12-12 13:13:10.0,2012-12-12 13:13:15.0]|4    |
|[2016-08-13 00:00:00.0,2016-08-13 00:00:05.0]|10   |
|[2017-05-27 00:00:00.0,2017-05-27 00:00:05.0]|15   |
+---------------------------------------------+-----+

scala> q.printSchema
root
 |-- window: struct (nullable = true)
 |    |-- start: timestamp (nullable = true)
 |    |-- end: timestamp (nullable = true)
 |-- level: integer (nullable = false)

// calculating the sum of levels every 5 seconds
val sums = levels.
  groupBy(window($"time", "5 seconds")).
  agg(sum("level") as "level_sum").
  select("window.start", "window.end", "level_sum")
scala> sums.show
+-------------------+-------------------+---------+
```

```
|               start|                 end|level_sum|
+-------------------+-------------------+---------+
|2012-12-12 13:13:10|2012-12-12 13:13:15|        4|
|2012-12-12 12:12:10|2012-12-12 12:12:15|       14|
|2016-08-13 00:00:00|2016-08-13 00:00:05|       10|
|2017-05-27 00:00:00|2017-05-27 00:00:05|       15|
+-------------------+-------------------+---------+
```

`windowDuration` and `slideDuration` are strings specifying the width of the window for duration and sliding identifiers, respectively.

Tip	Use `CalendarInterval` for valid window identifiers.

There are a couple of rules governing the durations:

1. The window duration must be greater than 0
2. The slide duration must be greater than 0.
3. The start time must be greater than or equal to 0.
4. The slide duration must be less than or equal to the window duration.
5. The start time must be less than the slide duration.

Note	Only one `window` expression is supported in a query.

Note	`null` values are filtered out in `window` expression.

Internally, `window` creates a Column with `TimeWindow` Catalyst expression under `window` alias.

```
scala> val timeColumn = window($"time", "5 seconds")
timeColumn: org.apache.spark.sql.Column = timewindow(time, 5000000, 5000000, 0) AS `wi
ndow`

val windowExpr = timeColumn.expr
scala> println(windowExpr.numberedTreeString)
00 timewindow('time, 5000000, 5000000, 0) AS window#23
01 +- timewindow('time, 5000000, 5000000, 0)
02    +- 'time
```

Internally, `TimeWindow` Catalyst expression is simply a struct type with two fields, i.e. `start` and `end`, both of `TimestampType` type.

```
scala> println(windowExpr.dataType)
StructType(StructField(start,TimestampType,true), StructField(end,TimestampType,true))

scala> println(windowExpr.dataType.prettyJson)
{
  "type" : "struct",
  "fields" : [ {
    "name" : "start",
    "type" : "timestamp",
    "nullable" : true,
    "metadata" : { }
  }, {
    "name" : "end",
    "type" : "timestamp",
    "nullable" : true,
    "metadata" : { }
  } ]
}
```

Note	`TimeWindow` time window Catalyst expression is planned (i.e. *converted*) in `TimeWindowing` logical optimization rule (i.e. `Rule[LogicalPlan]`) of the Spark SQL logical query plan analyzer. Find more about the Spark SQL logical query plan analyzer in Mastering Apache Spark 2 gitbook.

Example — Traffic Sensor

Note	The example is borrowed from Introducing Stream Windows in Apache Flink.

The example shows how to use `window` function to model a traffic sensor that counts every 15 seconds the number of vehicles passing a certain location.

KeyValueGroupedDataset — Streaming Aggregation

`KeyValueGroupedDataset` represents a **grouped dataset** as a result of groupByKey operator (that aggregates records by a grouping function).

```
// Dataset[T]
groupByKey(func: T => K): KeyValueGroupedDataset[K, T]
```

`KeyValueGroupedDataset` works for batch and streaming aggregations, but shines the most when used for **streaming aggregation** (with streaming Datasets).

```
import java.sql.Timestamp
scala> val numGroups = spark.
  readStream.
  format("rate").
  load.
  as[(Timestamp, Long)].
  groupByKey { case (time, value) => value % 2 }
numGroups: org.apache.spark.sql.KeyValueGroupedDataset[Long,(java.sql.Timestamp, Long)
] = org.apache.spark.sql.KeyValueGroupedDataset@616c1605

import org.apache.spark.sql.streaming.Trigger
import scala.concurrent.duration._
numGroups.
  mapGroups { case(group, values) => values.size }.
  writeStream.
  format("console").
  trigger(Trigger.ProcessingTime(10.seconds)).
  start

-------------------------------------------
Batch: 0
-------------------------------------------
+-----+
|value|
+-----+
+-----+

-------------------------------------------
Batch: 1
-------------------------------------------
+-----+
|value|
+-----+
|    3|
|    2|
+-----+

-------------------------------------------
Batch: 2
-------------------------------------------
+-----+
|value|
+-----+
|    5|
|    5|
+-----+

// Eventually...
spark.streams.active.foreach(_.stop)
```

The most prestigious use case of `KeyValueGroupedDataset` however is **stateful streaming aggregation** that allows for accumulating **streaming state** (by means of GroupState) using mapGroupsWithState and the more advanced flatMapGroupsWithState operators.

Table 1. KeyValueGroupedDataset's Operators

Operator	Description
agg	
cogroup	
count	
flatMapGroups	
flatMapGroupsWithState	Creates a `Dataset` with FlatMapGroupsWithState logical operator Note: The difference between `flatMapGroupsWithState` and mapGroupsWithState is the state function that generates zero or more elements (that are in turn the rows in the result `Dataset`).
keyAs	
keys	
mapGroups	
mapGroupsWithState	Creates a `Dataset` with FlatMapGroupsWithState logical operator Note: The difference between `mapGroupsWithState` and flatMapGroupsWithState is the state function that generates exactly one element (that is in turn the row in the result `Dataset`).
mapValues	
queryExecution	
reduceGroups	

Creating KeyValueGroupedDataset Instance

`KeyValueGroupedDataset` takes the following when created:

- `Encoder` for keys
- `Encoder` for values
- `QueryExecution`
- Data attributes
- Grouping attributes

mapGroupsWithState Operator — Stateful Streaming Aggregation (with Explicit State Logic)

```
mapGroupsWithState[S: Encoder, U: Encoder](
  func: (K, Iterator[V], GroupState[S]) => U): Dataset[U] (1)
mapGroupsWithState[S: Encoder, U: Encoder](
  timeoutConf: GroupStateTimeout)(
  func: (K, Iterator[V], GroupState[S]) => U): Dataset[U]
```

1. Uses `GroupStateTimeout.NoTimeout` for `timeoutConf`

`mapGroupsWithState` operator…FIXME

Note	`mapGroupsWithState` is a special case of flatMapGroupsWithState operator with the following: • `func` being transformed to return a single-element `Iterator` • Update output mode `mapGroupsWithState` also creates a `FlatMapGroupsWithState` with isMapGroupsWithState internal flag enabled.

```
// numGroups defined at the beginning
scala> :type numGroups
org.apache.spark.sql.KeyValueGroupedDataset[Long,(java.sql.Timestamp, Long)]

import org.apache.spark.sql.streaming.GroupState
def mappingFunc(key: Long, values: Iterator[(java.sql.Timestamp, Long)], state: GroupS
tate[Long]): Long = {
  println(s">>> key: $key => state: $state")
  val newState = state.getOption.map(_ + values.size).getOrElse(0L)
  state.update(newState)
  key
}

import org.apache.spark.sql.streaming.GroupStateTimeout
val longs = numGroups.mapGroupsWithState(
    timeoutConf = GroupStateTimeout.ProcessingTimeTimeout)(
    func = mappingFunc)

import org.apache.spark.sql.streaming.{OutputMode, Trigger}
import scala.concurrent.duration._
val q = longs.
  writeStream.
```

```
  format("console").
  trigger(Trigger.ProcessingTime(10.seconds)).
  outputMode(OutputMode.Update). // <-- required for mapGroupsWithState
  start

// Note GroupState

-------------------------------------------
Batch: 1
-------------------------------------------
>>> key: 0 => state: GroupState(<undefined>)
>>> key: 1 => state: GroupState(<undefined>)
+-----+
|value|
+-----+
|    0|
|    1|
+-----+

-------------------------------------------
Batch: 2
-------------------------------------------
>>> key: 0 => state: GroupState(0)
>>> key: 1 => state: GroupState(0)
+-----+
|value|
+-----+
|    0|
|    1|
+-----+

-------------------------------------------
Batch: 3
-------------------------------------------
>>> key: 0 => state: GroupState(4)
>>> key: 1 => state: GroupState(4)
+-----+
|value|
+-----+
|    0|
|    1|
+-----+

// in the end
spark.streams.active.foreach(_.stop)
```

flatMapGroupsWithState Operator — Arbitrary Stateful Streaming Aggregation (with Explicit State Logic)

```
flatMapGroupsWithState[S: Encoder, U: Encoder](
  outputMode: OutputMode,
  timeoutConf: GroupStateTimeout)(
  func: (K, Iterator[V], GroupState[S]) => Iterator[U]): Dataset[U]
```

Note	`flatMapGroupsWithState` requires Append or Update output modes.

Note	Every time the state function `func` is executed for a key, the state (as `GroupState[S]`) is for this key only.

Caution	FIXME Why can't `flatMapGroupsWithState` work with Complete output mode?

<table><tr><td>Note</td><td><code>K</code> is the type of the keys in <code>KeyValueGroupedDataset</code>
<code>V</code> is the type of the values (per key) in <code>KeyValueGroupedDataset</code>
<code>S</code> is the user-defined type of the state as maintained for each group
<code>U</code> is the type of rows in the result <code>Dataset</code></td></tr></table>

```
scala> spark.version
res0: String = 2.3.0-SNAPSHOT

import java.sql.Timestamp
type DeviceId = Int
case class Signal(timestamp: java.sql.Timestamp, value: Long, deviceId: DeviceId)

// input stream
import org.apache.spark.sql.functions._
val signals = spark.
  readStream.
  format("rate").
  option("rowsPerSecond", 1).
  load.
  withColumn("value", $"value" % 10).  // <-- randomize the values (just for fun)
  withColumn("deviceId", rint(rand() * 10) cast "int"). // <-- 10 devices randomly ass
igned to values
  as[Signal] // <-- convert to our type (from "unpleasant" Row)
scala> signals.explain
== Physical Plan ==
```

```
*Project [timestamp#0, (value#1L % 10) AS value#5L, cast(ROUND((rand(44402963953411529
93) * 10.0)) as int) AS deviceId#9]
+- StreamingRelation rate, [timestamp#0, value#1L]

// stream processing using flatMapGroupsWithState operator
val device: Signal => DeviceId = { case Signal(_, _, deviceId) => deviceId }
val signalsByDevice = signals.groupByKey(device)

import org.apache.spark.sql.streaming.GroupState
type Key = Int
type Count = Long
type State = Map[Key, Count]
case class EventsCounted(deviceId: DeviceId, count: Long)
def countValuesPerKey(deviceId: Int, signalsPerDevice: Iterator[Signal], state: GroupS
tate[State]): Iterator[EventsCounted] = {
  val values = signalsPerDevice.toList
  println(s"Device: $deviceId")
  println(s"Signals (${values.size}):")
  values.zipWithIndex.foreach { case (v, idx) => println(s"$idx. $v") }
  println(s"State: $state")

  // update the state with the count of elements for the key
  val initialState: State = Map(deviceId -> 0)
  val oldState = state.getOption.getOrElse(initialState)
  // the name to highlight that the state is for the key only
  val newValue = oldState(deviceId) + values.size
  val newState = Map(deviceId -> newValue)
  state.update(newState)

  // you must not return as it's already consumed
  // that leads to a very subtle error where no elements are in an iterator
  // iterators are one-pass data structures
  Iterator(EventsCounted(deviceId, newValue))
}
import org.apache.spark.sql.streaming.{GroupStateTimeout, OutputMode}
val signalCounter = signalsByDevice.flatMapGroupsWithState(
  outputMode = OutputMode.Append,
  timeoutConf = GroupStateTimeout.NoTimeout)(func = countValuesPerKey)

import org.apache.spark.sql.streaming.{OutputMode, Trigger}
import scala.concurrent.duration._
val sq = signalCounter.
  writeStream.
  format("console").
  option("truncate", false).
  trigger(Trigger.ProcessingTime(10.seconds)).
  outputMode(OutputMode.Append).
  start
...
-------------------------------------------
Batch: 0
-------------------------------------------
+--------+-----+
```

```
|deviceId|count|
+--------+-----+
+--------+-----+
...
17/08/21 08:57:29 INFO StreamExecution: Streaming query made progress: {
  "id" : "a43822a6-500b-4f02-9133-53e9d39eedbf",
  "runId" : "79cb037e-0f28-4faf-a03e-2572b4301afe",
  "name" : null,
  "timestamp" : "2017-08-21T06:57:26.719Z",
  "batchId" : 0,
  "numInputRows" : 0,
  "processedRowsPerSecond" : 0.0,
  "durationMs" : {
    "addBatch" : 2404,
    "getBatch" : 22,
    "getOffset" : 0,
    "queryPlanning" : 141,
    "triggerExecution" : 2626,
    "walCommit" : 41
  },
  "stateOperators" : [ {
    "numRowsTotal" : 0,
    "numRowsUpdated" : 0,
    "memoryUsedBytes" : 12599
  } ],
  "sources" : [ {
    "description" : "RateSource[rowsPerSecond=1, rampUpTimeSeconds=0, numPartitions=8]"
,
    "startOffset" : null,
    "endOffset" : 0,
    "numInputRows" : 0,
    "processedRowsPerSecond" : 0.0
  } ],
  "sink" : {
    "description" : "ConsoleSink[numRows=20, truncate=false]"
  }
}
17/08/21 08:57:29 DEBUG StreamExecution: batch 0 committed
...
-------------------------------------------
Batch: 1
-------------------------------------------
Device: 3
Signals (1):
0. Signal(2017-08-21 08:57:27.682,1,3)
State: GroupState(<undefined>)
Device: 8
Signals (1):
0. Signal(2017-08-21 08:57:26.682,0,8)
State: GroupState(<undefined>)
Device: 7
Signals (1):
0. Signal(2017-08-21 08:57:28.682,2,7)
```

```
State: GroupState(<undefined>)
+--------+-----+
|deviceId|count|
+--------+-----+
|3       |1    |
|8       |1    |
|7       |1    |
+--------+-----+
...
17/08/21 08:57:31 INFO StreamExecution: Streaming query made progress: {
  "id" : "a43822a6-500b-4f02-9133-53e9d39eedbf",
  "runId" : "79cb037e-0f28-4faf-a03e-2572b4301afe",
  "name" : null,
  "timestamp" : "2017-08-21T06:57:30.004Z",
  "batchId" : 1,
  "numInputRows" : 3,
  "inputRowsPerSecond" : 0.91324200913242,
  "processedRowsPerSecond" : 2.2388059701492535,
  "durationMs" : {
    "addBatch" : 1245,
    "getBatch" : 22,
    "getOffset" : 0,
    "queryPlanning" : 23,
    "triggerExecution" : 1340,
    "walCommit" : 44
  },
  "stateOperators" : [ {
    "numRowsTotal" : 3,
    "numRowsUpdated" : 3,
    "memoryUsedBytes" : 18095
  } ],
  "sources" : [ {
    "description" : "RateSource[rowsPerSecond=1, rampUpTimeSeconds=0, numPartitions=8]"
,
    "startOffset" : 0,
    "endOffset" : 3,
    "numInputRows" : 3,
    "inputRowsPerSecond" : 0.91324200913242,
    "processedRowsPerSecond" : 2.2388059701492535
  } ],
  "sink" : {
    "description" : "ConsoleSink[numRows=20, truncate=false]"
  }
}
17/08/21 08:57:31 DEBUG StreamExecution: batch 1 committed
...
-------------------------------------------
Batch: 2
-------------------------------------------
Device: 1
Signals (1):
0. Signal(2017-08-21 08:57:36.682,0,1)
State: GroupState(<undefined>)
```

```
Device: 3
Signals (2):
0. Signal(2017-08-21 08:57:32.682,6,3)
1. Signal(2017-08-21 08:57:35.682,9,3)
State: GroupState(Map(3 -> 1))
Device: 5
Signals (1):
0. Signal(2017-08-21 08:57:34.682,8,5)
State: GroupState(<undefined>)
Device: 4
Signals (1):
0. Signal(2017-08-21 08:57:29.682,3,4)
State: GroupState(<undefined>)
Device: 8
Signals (2):
0. Signal(2017-08-21 08:57:31.682,5,8)
1. Signal(2017-08-21 08:57:33.682,7,8)
State: GroupState(Map(8 -> 1))
Device: 7
Signals (2):
0. Signal(2017-08-21 08:57:30.682,4,7)
1. Signal(2017-08-21 08:57:37.682,1,7)
State: GroupState(Map(7 -> 1))
Device: 0
Signals (1):
0. Signal(2017-08-21 08:57:38.682,2,0)
State: GroupState(<undefined>)
+--------+-----+
|deviceId|count|
+--------+-----+
|1       |1    |
|3       |3    |
|5       |1    |
|4       |1    |
|8       |3    |
|7       |3    |
|0       |1    |
+--------+-----+
...
17/08/21 08:57:41 INFO StreamExecution: Streaming query made progress: {
  "id" : "a43822a6-500b-4f02-9133-53e9d39eedbf",
  "runId" : "79cb037e-0f28-4faf-a03e-2572b4301afe",
  "name" : null,
  "timestamp" : "2017-08-21T06:57:40.005Z",
  "batchId" : 2,
  "numInputRows" : 10,
  "inputRowsPerSecond" : 0.9999000099990002,
  "processedRowsPerSecond" : 9.242144177449168,
  "durationMs" : {
    "addBatch" : 1032,
    "getBatch" : 8,
    "getOffset" : 0,
    "queryPlanning" : 19,
```

```
    "triggerExecution" : 1082,
    "walCommit" : 21
  },
  "stateOperators" : [ {
    "numRowsTotal" : 7,
    "numRowsUpdated" : 7,
    "memoryUsedBytes" : 19023
  } ],
  "sources" : [ {
    "description" : "RateSource[rowsPerSecond=1, rampUpTimeSeconds=0, numPartitions=8]"
,
    "startOffset" : 3,
    "endOffset" : 13,
    "numInputRows" : 10,
    "inputRowsPerSecond" : 0.9999000099990002,
    "processedRowsPerSecond" : 9.242144177449168
  } ],
  "sink" : {
    "description" : "ConsoleSink[numRows=20, truncate=false]"
  }
}
17/08/21 08:57:41 DEBUG StreamExecution: batch 2 committed

// In the end...
sq.stop

// Use stateOperators to access the stats
scala> println(sq.lastProgress.stateOperators(0).prettyJson)
{
  "numRowsTotal" : 7,
  "numRowsUpdated" : 7,
  "memoryUsedBytes" : 19023
}
```

Internally, `flatMapGroupsWithState` operator creates a `Dataset` with FlatMapGroupsWithState unary logical operator.

```
scala> :type signalCounter
org.apache.spark.sql.Dataset[EventsCounted]
scala> println(signalCounter.queryExecution.logical.numberedTreeString)
00 'SerializeFromObject [assertnotnull(assertnotnull(input[0, $line27.$read$$iw$$iw$Ev
entsCounted, true])).deviceId AS deviceId#25, assertnotnull(assertnotnull(input[0, $li
ne27.$read$$iw$$iw$EventsCounted, true])).count AS count#26L]
01 +- 'FlatMapGroupsWithState <function3>, unresolveddeserializer(upcast(getcolumnbyor
dinal(0, IntegerType), IntegerType, - root class: "scala.Int"), value#20), unresolvedd
eserializer(newInstance(class $line17.$read$$iw$$iw$Signal), timestamp#0, value#5L, de
viceId#9), [value#20], [timestamp#0, value#5L, deviceId#9], obj#24: $line27.$read$$iw$
$iw$EventsCounted, class[value[0]: map<int,bigint>], Append, false, NoTimeout
02    +- AppendColumns <function1>, class $line17.$read$$iw$$iw$Signal, [StructField(t
imestamp,TimestampType,true), StructField(value,LongType,false), StructField(deviceId,
IntegerType,false)], newInstance(class $line17.$read$$iw$$iw$Signal), [input[0, int, f
alse] AS value#20]
03       +- Project [timestamp#0, value#5L, cast(ROUND((rand(4440296395341152993) * ca
st(10 as double))) as int) AS deviceId#9]
04          +- Project [timestamp#0, (value#1L % cast(10 as bigint)) AS value#5L]
05             +- StreamingRelation DataSource(org.apache.spark.sql.SparkSession@385c6
d6b,rate,List(),None,List(),None,Map(rowsPerSecond -> 1),None), rate, [timestamp#0, va
lue#1L]

scala> signalCounter.explain
== Physical Plan ==
*SerializeFromObject [assertnotnull(input[0, $line27.$read$$iw$$iw$EventsCounted, true
]).deviceId AS deviceId#25, assertnotnull(input[0, $line27.$read$$iw$$iw$EventsCounted
, true]).count AS count#26L]
+- FlatMapGroupsWithState <function3>, value#20: int, newInstance(class $line17.$read$
$iw$$iw$Signal), [value#20], [timestamp#0, value#5L, deviceId#9], obj#24: $line27.$rea
d$$iw$$iw$EventsCounted, StatefulOperatorStateInfo(<unknown>,50c7ece5-0716-4e43-9b56-0
9842db8baf1,0,0), class[value[0]: map<int,bigint>], Append, NoTimeout, 0, 0
   +- *Sort [value#20 ASC NULLS FIRST], false, 0
      +- Exchange hashpartitioning(value#20, 200)
         +- AppendColumns <function1>, newInstance(class $line17.$read$$iw$$iw$Signal)
, [input[0, int, false] AS value#20]
            +- *Project [timestamp#0, (value#1L % 10) AS value#5L, cast(ROUND((rand(44
40296395341152993) * 10.0)) as int) AS deviceId#9]
               +- StreamingRelation rate, [timestamp#0, value#1L]
```

`flatMapGroupsWithState` reports a `IllegalArgumentException` when the input `outputMode` is neither `Append` nor `Update`.

```
scala> val result = signalsByDevice.flatMapGroupsWithState(
     |   outputMode = OutputMode.Complete,
     |   timeoutConf = GroupStateTimeout.NoTimeout)(func = stateFn)
java.lang.IllegalArgumentException: The output mode of function should be append or up
date
  at org.apache.spark.sql.KeyValueGroupedDataset.flatMapGroupsWithState(KeyValueGroupe
dDataset.scala:381)
  ... 54 elided
```

Caution	FIXME Examples for append and update output modes (to demo the difference)

Caution	FIXME Examples for `GroupStateTimeout.EventTimeTimeout` with `withWatermark` operator

GroupState — State Per Group in Stateful Streaming Aggregation

`GroupState` is the contract for working with a state (of type `S`) per group for arbitrary stateful aggregation (using mapGroupsWithState or flatMapGroupsWithState operators).

Note	GroupStateImpl is the one and only implementation of `GroupState` available.

GroupState Contract

```
package org.apache.spark.sql.streaming

trait GroupState[S] extends LogicalGroupState[S] {
  def exists: Boolean
  def get: S
  def getOption: Option[S]
  def update(newState: S): Unit
  def remove(): Unit
  def hasTimedOut: Boolean
  def setTimeoutDuration(durationMs: Long): Unit
  def setTimeoutDuration(duration: String): Unit
  def setTimeoutTimestamp(timestampMs: Long): Unit
  def setTimeoutTimestamp(timestampMs: Long, additionalDuration: String): Unit
  def setTimeoutTimestamp(timestamp: java.sql.Date): Unit
  def setTimeoutTimestamp(timestamp: java.sql.Date, additionalDuration: String): Unit
}
```

Table 1. GroupState Contract

Method	Description
`exists`	
`get`	Gives the state
`getOption`	Gives the state as `Some(…)` if available or `None`
`update`	Replaces the state with a new state (per group)
`remove`	
`hasTimedOut`	

GroupStateImpl

`GroupStateImpl` …FIXME

GroupStateTimeout

`GroupStateTimeout` represents the possible timeouts that you can use for the state-aware Dataset operations:

- mapGroupsWithState
- flatMapGroupsWithState

`GroupStateTimeout` is part of `org.apache.spark.sql.streaming` package.

```
import org.apache.spark.sql.streaming.GroupStateTimeout
```

Table 1. Types of GroupStateTimeouts (in alphabetical order)

GroupStateTimeout	Description
`ProcessingTimeTimeout`	Timeout based on the processing time. Note: `FlatMapGroupsWithStateExec` requires that `batchTimestampMs` is specified when `ProcessingTimeTimeout` is used. `batchTimestampMs` is defined when `IncrementalExecution` is created (and so is state). `IncrementalExecution` is given `OffsetSeqMetadata` when `StreamExecution` runs a streaming batch. Caution: FIXME Describe `OffsetSeqMetadata` and `StreamExecution.offsetSeqMetadata`
`EventTimeTimeout`	Timeout based on the event time Used when…FIXME
`NoTimeout`	No timeout Used when…FIXME

StreamingQueryManager — Streaming Query Management

`StreamingQueryManager` is the management interface for streaming queries in a single SparkSession.

Table 1. StreamingQueryManager API

Method	Description
active	Gets all active structured queries `active: Array[StreamingQuery]`
addListener	Registers a StreamingQueryListener `addListener(listener: StreamingQueryListener): Unit`

awaitAnyTermination	Waits for any streaming query to be terminated
	`awaitAnyTermination(): Unit`
	`awaitAnyTermination(timeoutMs: Long): Boolean`

get	Gets a StreamingQuery by id
	`get(id: String): StreamingQuery`
	`get(id: UUID): StreamingQuery`

removeListener	De-registering a StreamingQueryListener `removeListener(listener: StreamingQueryListener): Unit`
resetTerminated	`resetTerminated(): Unit`

`StreamingQueryManager` is available using `SparkSession` and `streams` property.

```
val spark: SparkSession = ...
val queries = spark.streams
```

`StreamingQueryManager` is created when `SessionState` is created.

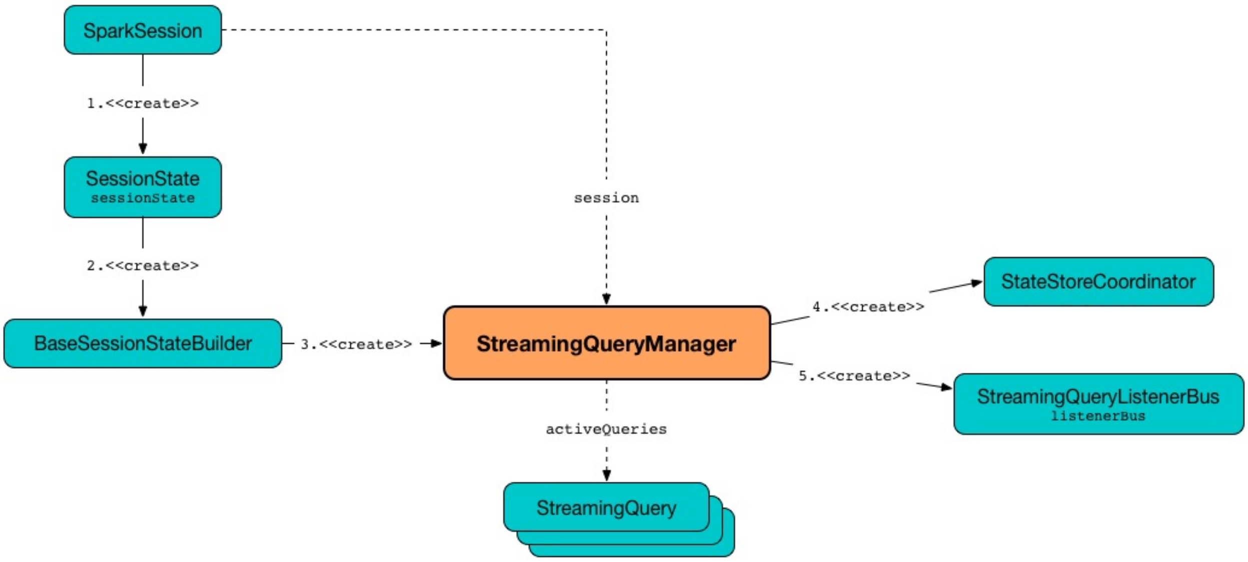

Figure 1. StreamingQueryManager

Tip	Refer to the Mastering Apache Spark 2 gitbook to learn about `SessionState`.

`StreamingQueryManager` is used (internally) to create a StreamingQuery (with its StreamExecution).

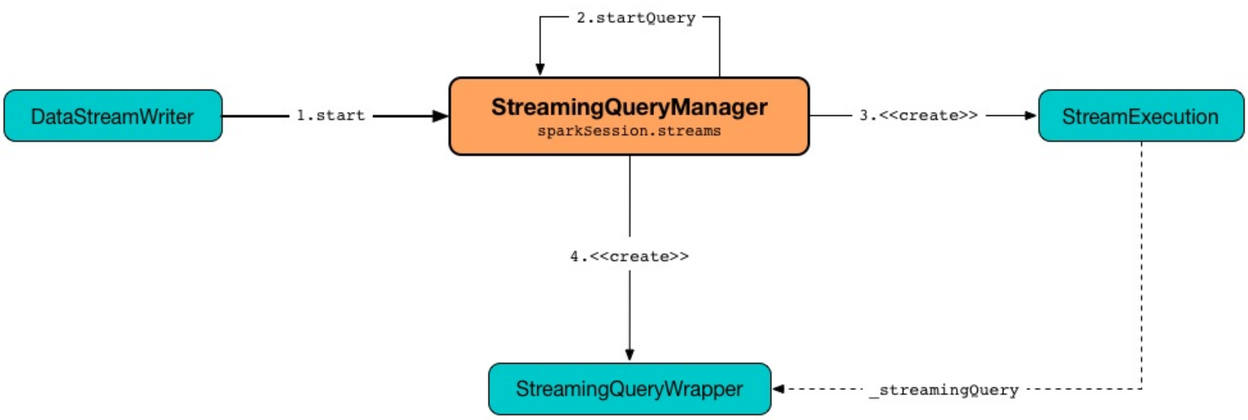

Figure 2. StreamingQueryManager Creates StreamingQuery (and StreamExecution)

`StreamingQueryManager` is notified about state changes of a structured query and passes them along (to query listeners).

`StreamingQueryManager` takes a single `SparkSession` when created.

Table 2. StreamingQueryManager's Internal Registries and Counters (in alphabetical order)

Name	Description
`activeQueries`	Registry of StreamingQueries per `UUID` Used when `StreamingQueryManager` is requested for active streaming queries, get a streaming query by id, starts a streaming query and is notified that a streaming query has terminated.

`activeQueriesLock`	
`awaitTerminationLock`	
`lastTerminatedQuery`	StreamingQuery that has recently been terminated, i.e. stopped or due to an exception. `null` when no streaming query has terminated yet or resetTerminated. • Used in awaitAnyTermination to know when a streaming query has terminated • Set when `StreamingQueryManager` is notified that a streaming query has terminated
`listenerBus`	StreamingQueryListenerBus (for the current SparkSession) Used to: • register or deregister a `StreamingQueryListener` • Post a streaming event (and notify `StreamingQueryListener` listeners about streaming events)
`stateStoreCoordinator`	StateStoreCoordinatorRef with the **StateStoreCoordinator** RPC Endpoint • Created when `StreamingQueryManager` is created Used when: • `StreamingQueryManager` is notified that a streaming query has terminated • Stateful operators are executed, i.e. FlatMapGroupsWithStateExec, StateStoreRestoreExec, StateStoreSaveExec, StreamingDeduplicateExec and StreamingSymmetricHashJoinExec • Creating StateStoreRDD (with storeUpdateFunction aborting StateStore when a task fails)

Getting All Active Streaming Queries — `active` Method

```
active: Array[StreamingQuery]
```

`active` gets all active streaming queries.

Getting Active Continuous Query By Name — `get` Method

```
get(name: String): StreamingQuery
```

`get` method returns a StreamingQuery by `name`.

It may throw an `IllegalArgumentException` when no StreamingQuery exists for the `name`.

```
java.lang.IllegalArgumentException: There is no active query with name hello
  at org.apache.spark.sql.StreamingQueryManager$$anonfun$get$1.apply(StreamingQueryMan
ager.scala:59)
  at org.apache.spark.sql.StreamingQueryManager$$anonfun$get$1.apply(StreamingQueryMan
ager.scala:59)
  at scala.collection.MapLike$class.getOrElse(MapLike.scala:128)
  at scala.collection.AbstractMap.getOrElse(Map.scala:59)
  at org.apache.spark.sql.StreamingQueryManager.get(StreamingQueryManager.scala:58)
  ... 49 elided
```

Registering StreamingQueryListener — `addListener` Method

```
addListener(listener: StreamingQueryListener): Unit
```

`addListener` requests the StreamingQueryListenerBus to add the input `listener`.

De-Registering StreamingQueryListener — `removeListener` Method

```
removeListener(listener: StreamingQueryListener): Unit
```

`removeListener` requests StreamingQueryListenerBus to remove the input `listener`.

Waiting for Any Streaming Query Termination — `awaitAnyTermination` Method

```
awaitAnyTermination(): Unit
awaitAnyTermination(timeoutMs: Long): Boolean
```

`awaitAnyTermination` acquires a lock on awaitTerminationLock and waits until any streaming query has finished (i.e. lastTerminatedQuery is available) or `timeoutMs` has expired.

`awaitAnyTermination` re-throws the `StreamingQueryException` from lastTerminatedQuery if it reported one.

`resetTerminated` Method

```
resetTerminated(): Unit
```

`resetTerminated` forgets about the past-terminated query (so that awaitAnyTermination can be used again to wait for a new streaming query termination).

Internally, `resetTerminated` acquires a lock on awaitTerminationLock and simply resets lastTerminatedQuery (i.e. sets it to `null`).

Creating Serializable StreamingQuery (StreamingQueryWrapper with StreamExecution) — `createQuery` Internal Method

```
createQuery(
  userSpecifiedName: Option[String],
  userSpecifiedCheckpointLocation: Option[String],
  df: DataFrame,
  sink: Sink,
  outputMode: OutputMode,
  useTempCheckpointLocation: Boolean,
  recoverFromCheckpointLocation: Boolean,
  trigger: Trigger,
  triggerClock: Clock): StreamingQueryWrapper
```

`createQuery` creates a StreamingQueryWrapper (for a StreamExecution per the input user-defined properties).

Internally, `createQuery` first finds the name of the checkpoint directory of a query (aka **checkpoint location**) in the following order:

1. Exactly the input `userSpecifiedCheckpointLocation` if defined
2. spark.sql.streaming.checkpointLocation Spark property if defined for the parent directory with a subdirectory per the optional `userSpecifiedName` (or a randomly-generated UUID)

3. (only when `useTempCheckpointLocation` is enabled) A temporary directory (as specified by `java.io.tmpdir` JVM property) with a subdirectory with `temporary` prefix.

Note	`userSpecifiedCheckpointLocation` can be any path that is acceptable by Hadoop's Path.

If the directory name for the checkpoint location could not be found, `createQuery` reports a `AnalysisException`.

```
checkpointLocation must be specified either through option("checkpointLocation", ...)
or SparkSession.conf.set("spark.sql.streaming.checkpointLocation", ...)
```

`createQuery` reports a `AnalysisException` when the input `recoverFromCheckpointLocation` flag is turned off but there is **offsets** directory in the checkpoint location.

`createQuery` makes sure that the logical plan of the structured query is analyzed (i.e. no logical errors have been found).

Unless spark.sql.streaming.unsupportedOperationCheck Spark property is turned on, `createQuery` checks the logical plan of the streaming query for unsupported operations.

(only when `spark.sql.adaptive.enabled` Spark property is turned on) `createQuery` prints out a WARN message to the logs:

```
WARN spark.sql.adaptive.enabled is not supported in streaming DataFrames/Datasets and
will be disabled.
```

In the end, `createQuery` creates a StreamingQueryWrapper with a new MicroBatchExecution.

<table><tr><td>Note</td><td><code>recoverFromCheckpointLocation</code> flag corresponds to <code>recoverFromCheckpointLocation</code> flag that <code>StreamingQueryManager</code> uses to start a streaming query and which is enabled by default (and is in fact the only place where <code>createQuery</code> is used).
<code>memory</code> sink has the flag enabled for Complete output mode only
<code>foreach</code> sink has the flag always enabled
<code>console</code> sink has the flag always disabled
all other sinks have the flag always enabled</td></tr></table>

Note	`userSpecifiedName` corresponds to `queryName` option (that can be defined using `DataStreamWriter` 's queryName method) while `userSpecifiedCheckpointLocation` is `checkpointLocation` option.

Note	`createQuery` is used exclusively when `StreamingQueryManager` is requested to start executing a streaming query.

Starting Streaming Query Execution — `startQuery` Internal Method

```
startQuery(
  userSpecifiedName: Option[String],
  userSpecifiedCheckpointLocation: Option[String],
  df: DataFrame,
  extraOptions: Map[String, String],
  sink: BaseStreamingSink,
  outputMode: OutputMode,
  useTempCheckpointLocation: Boolean = false,
  recoverFromCheckpointLocation: Boolean = true,
  trigger: Trigger = ProcessingTime(0),
  triggerClock: Clock = new SystemClock()): StreamingQuery
```

`startQuery` starts a streaming query.

Note	`trigger` defaults to `0` milliseconds (as ProcessingTime(0)).

Internally, `startQuery` first creates a StreamingQueryWrapper, registers it in activeQueries internal registry (by the id), requests it for the underlying StreamExecution and starts it.

In the end, `startQuery` returns the StreamingQueryWrapper (as part of the fluent API so you can chain operators) or throws the exception that was reported when attempting to start the query.

`startQuery` throws an `IllegalArgumentException` when there is another query registered under `name`. `startQuery` looks it up in the activeQueries internal registry.

```
Cannot start query with name [name] as a query with that name is already active
```

`startQuery` throws an `IllegalStateException` when a query is started again from checkpoint. `startQuery` looks it up in activeQueries internal registry.

```
Cannot start query with id [id] as another query with same id is
already active. Perhaps you are attempting to restart a query
from checkpoint that is already active.
```

Note	`startQuery` is used exclusively when `DataStreamWriter` is requested to start.

Posting StreamingQueryListener Event to StreamingQueryListenerBus — `postListenerEvent` Internal Method

```
postListenerEvent(event: StreamingQueryListener.Event): Unit
```

`postListenerEvent` simply posts the input `event` to StreamingQueryListenerBus.

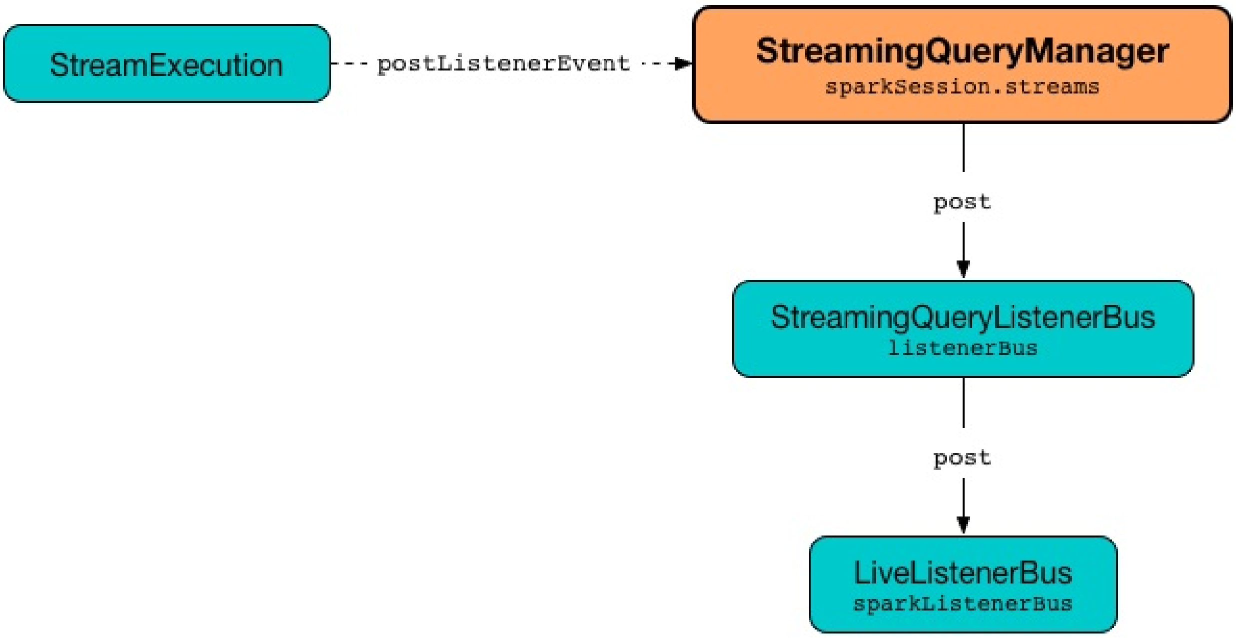

Figure 3. StreamingQueryManager Propagates StreamingQueryListener Events

Note	`postListenerEvent` is used exclusively when `StreamExecution` posts a streaming event.

Marking Streaming Query as Terminated (and Deactivating Query in StateStoreCoordinator) — `notifyQueryTermination` Internal Method

```
notifyQueryTermination(terminatedQuery: StreamingQuery): Unit
```

`notifyQueryTermination` removes the `terminatedQuery` from activeQueries internal registry (by the query id).

`notifyQueryTermination` records the `terminatedQuery` in lastTerminatedQuery internal registry (when no earlier streaming query was recorded or the `terminatedQuery` terminated due to an exception).

`notifyQueryTermination` notifies others that are blocked on awaitTerminationLock.

In the end, `notifyQueryTermination` requests StateStoreCoordinator to deactivate all active runs of the streaming query.

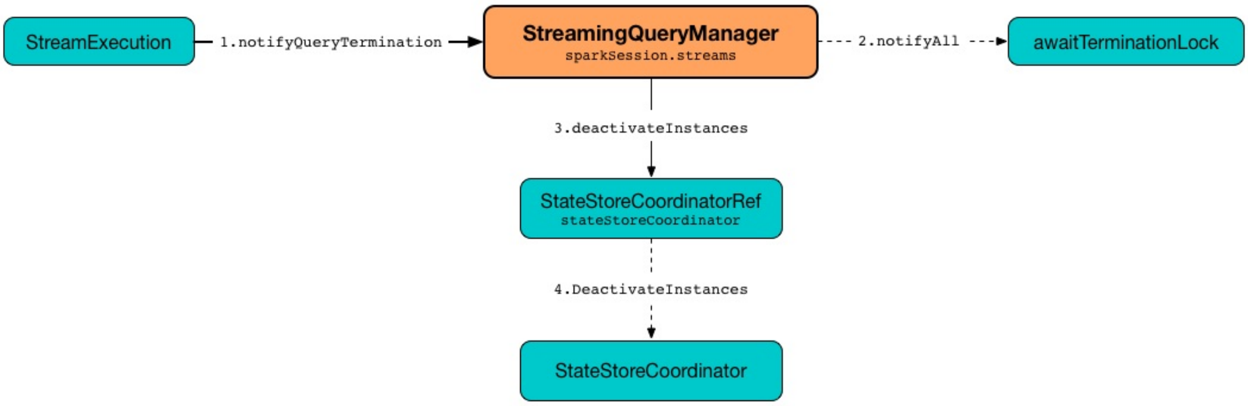

Figure 4. StreamingQueryManager's Marking Streaming Query as Terminated

Note	`notifyQueryTermination` is used exclusively when `StreamExecution` has finished (running streaming batches) (possibly due to an exception).

Configuration Properties

The following list are the properties that you can use to fine-tune Spark Structured Streaming applications.

You can set them in a SparkSession upon instantiation using config method.

```
import org.apache.spark.sql.SparkSession
val spark: SparkSession = SparkSession.builder
  .master("local[*]")
  .appName("My Spark Application")
  .config("spark.sql.streaming.metricsEnabled", true)
  .getOrCreate
```

Table 1. Structured Streaming's Propertie

Name	
`spark.sql.streaming.checkpointLocation`	(empty)
`spark.sql.streaming.metricsEnabled`	`false`
`spark.sql.streaming.minBatchesToRetain`	`100`
`spark.sql.streaming.numRecentProgressUpdates`	`100`
`spark.sql.streaming.pollingDelay`	`10` (millis)

`spark.sql.streaming.stateStore.maintenanceInterval`	`60s`
`spark.sql.streaming.stateStore.providerClass`	org.apache.spark.sql.execution.st
`spark.sql.streaming.unsupportedOperationCheck`	`true`

KafkaSourceProvider — Data Source Provider for Apache Kafka

`KafkaSourceProvider` is a streaming data source provider for KafkaSource (that is both the batch and streaming data source for Apache Kafka).

`KafkaSourceProvider` (as a `DataSourceRegister`) is registered as **kafka** format.

```
spark.readStream.format("kafka")
```

`KafkaSourceProvider` requires the following options (that you can set using `option` method of `DataStreamReader` or `DataStreamWriter`):

1. Exactly one option for `subscribe`, `subscribepattern` or `assign`
2. `kafka.bootstrap.servers` (that becomes `bootstrap.servers` property of the Kafka client)

Tip	Refer to KafkaSource's Options for the supported options.

Note	`endingoffsets` option is not allowed in streaming queries.

Note	`KafkaSourceProvider` is part of spark-sql-kafka-0-10 Library Dependency and so has to be "installed" in spark-shell using `--package` command-line option.

```
// See the section about KafkaSource for a complete example
val records = spark.
  readStream.
  format("kafka").  // <-- use KafkaSourceProvider
  option("subscribe", "raw-events").
  option("kafka.bootstrap.servers", "localhost:9092").
  option("startingoffsets", "earliest").
  option("maxOffsetsPerTrigger", 1).
  load
```

Creating KafkaSource — `createSource` Method

```
createSource(
  sqlContext: SQLContext,
  metadataPath: String,
  schema: Option[StructType],
  providerName: String,
  parameters: Map[String, String]): Source
```

Internally, `createSource` first validates stream options.

Caution	FIXME

Note	`createSource` is a part of StreamSourceProvider Contract to create a streaming source for **kafka** format.

spark-sql-kafka-0-10 Library Dependency

The new structured streaming API for Kafka is part of the `spark-sql-kafka-0-10` artifact. Add the following dependency to sbt project to use the streaming integration:

```
libraryDependencies += "org.apache.spark" %% "spark-sql-kafka-0-10" % "2.2.0"
```

> **Tip**
>
> `spark-sql-kafka-0-10` module is not included in the CLASSPATH of spark-shell so you have to start it with `--packages` command-line option.
>
> ```
> ./bin/spark-shell --packages org.apache.spark:spark-sql-kafka-0-10_2.11:2.2.0
> ```

Note	Replace `2.2.0` or `2.3.0-SNAPSHOT` with one of the available versions found at The Central Repository's Search that matches your version of Spark.

Validating General Options For Batch And Streaming Queries — `validateGeneralOptions` Internal Method

```
validateGeneralOptions(parameters: Map[String, String]): Unit
```

Note	Parameters are case-insensitive, i.e. `OptioN` and `option` are equal.

`validateGeneralOptions` makes sure that exactly one topic subscription strategy is used in `parameters` and can be:

- `subscribe`

- `subscribepattern`
- `assign`

`validateGeneralOptions` reports an `IllegalArgumentException` when there is no subscription strategy in use or there are more than one strategies used.

`validateGeneralOptions` makes sure that the value of subscription strategies meet the requirements:

- `assign` strategy starts with `{` (the opening curly brace)
- `subscribe` strategy has at least one topic (in a comma-separated list of topics)
- `subscribepattern` strategy has the pattern defined

`validateGeneralOptions` makes sure that `group.id` has not been specified and reports an `IllegalArgumentException` otherwise.

```
Kafka option 'group.id' is not supported as user-specified consumer groups are not use
d to track offsets.
```

`validateGeneralOptions` makes sure that `auto.offset.reset` has not been specified and reports an `IllegalArgumentException` otherwise.

```
Kafka option 'auto.offset.reset' is not supported.
Instead set the source option 'startingoffsets' to 'earliest' or
'latest' to specify where to start. Structured Streaming manages
which offsets are consumed internally, rather than relying on
the kafkaConsumer to do it. This will ensure that no data is
missed when new topics/partitions are dynamically subscribed.
Note that 'startingoffsets' only applies when a new Streaming
query is started, and
that resuming will always pick up from where the query left off.
See the docs for more details.
```

`validateGeneralOptions` makes sure that the following options have not been specified and reports an `IllegalArgumentException` otherwise:

- `kafka.key.deserializer`
- `kafka.value.deserializer`
- `kafka.enable.auto.commit`

- `kafka.interceptor.classes`

In the end, `validateGeneralOptions` makes sure that `kafka.bootstrap.servers` option was specified and reports an `IllegalArgumentException` otherwise.

```
Option 'kafka.bootstrap.servers' must be specified for configuring Kafka consumer
```

Note	`validateGeneralOptions` is used when `KafkaSourceProvider` validates options for streaming and batch queries.

Creating ConsumerStrategy — `strategy` Internal Method

```
strategy(caseInsensitiveParams: Map[String, String])
```

Internally, `strategy` finds the keys in the input `caseInsensitiveParams` that are one of the following and creates a corresponding ConsumerStrategy.

Table 1. KafkaSourceProvider.strategy's Key to ConsumerStrategy Conversion

Key	ConsumerStrategy
`assign`	AssignStrategy with Kafka's TopicPartitions. --- `strategy` uses `JsonUtils.partitions` method to parse a JSON with topic names and partitions, e.g. `{"topicA":[0,1],"topicB":[0,1]}` The topic names and partitions are mapped directly to Kafka's `TopicPartition` objects.
`subscribe`	SubscribeStrategy with topic names --- `strategy` extracts topic names from a comma-separated string, e.g. `topic1,topic2,topic3`
`subscribepattern`	SubscribePatternStrategy with topic subscription regex pattern (that uses Java's java.util.regex.Pattern for the pattern), e.g. `` ` topic\d ` ``

Note	`strategy` is used when: • `KafkaSourceProvider` creates a KafkaOffsetReader for KafkaSource. • `KafkaSourceProvider` creates a KafkaRelation (using `createRelation` method).

Specifying Name and Schema of Streaming Source for Kafka Format — `sourceSchema` Method

```
sourceSchema(
  sqlContext: SQLContext,
  schema: Option[StructType],
  providerName: String,
  parameters: Map[String, String]): (String, StructType)
```

Note	`sourceSchema` is a part of StreamSourceProvider Contract to define the name and the schema of a streaming source.

`sourceSchema` gives the short name (i.e. `kafka`) and the fixed schema.

Internally, `sourceSchema` validates Kafka options and makes sure that the optional input `schema` is indeed undefined.

When the input `schema` is defined, `sourceSchema` reports a `IllegalArgumentException` .

```
Kafka source has a fixed schema and cannot be set with a custom one
```

Note	`sourceSchema` is used exclusively when `DataSource` is requested the name and schema of a streaming source.

Validating Kafka Options for Streaming Queries — `validateStreamOptions` Internal Method

```
validateStreamOptions(caseInsensitiveParams: Map[String, String]): Unit
```

Firstly, `validateStreamOptions` makes sure that `endingoffsets` option is not used. Otherwise, `validateStreamOptions` reports a `IllegalArgumentException` .

```
ending offset not valid in streaming queries
```

`validateStreamOptions` then validates the general options.

Note	`validateStreamOptions` is used when `KafkaSourceProvider` is requested the schema for Kafka source and to create a KafkaSource.

KafkaSource

`KafkaSource` is a streaming source that generates DataFrames of records from one or more topics in Apache Kafka.

Note	Kafka topics are checked for new records every trigger and so there is some noticeable delay between when the records have arrived to Kafka topics and when a Spark application processes them.

<table>
<tr>
<td>Note</td>
<td>
Structured Streaming support for Kafka is in a separate spark-sql-kafka-0-10 module (aka library dependency).
<code>spark-sql-kafka-0-10</code> module is not included by default so you have to start <code>spark-submit</code> (and "derivatives" like <code>spark-shell</code>) with <code>--packages</code> command-line option to "install" it.
<pre>./bin/spark-shell --packages org.apache.spark:spark-sql-kafka-0-10_2.11:2.2.0</pre>
Replace the version of <code>spark-sql-kafka-0-10</code> module (e.g. <code>2.2.0</code> above) with one of the available versions found at The Central Repository's Search that matches your version of Spark.
</td>
</tr>
</table>

`KafkaSource` is created for **kafka** format (that is registered by KafkaSourceProvider).

```
val kafkaSource = spark.
  readStream.
  format("kafka"). // <-- use KafkaSource
  option("subscribe", "input").
  option("kafka.bootstrap.servers", "localhost:9092").
  load
```

Figure 1. KafkaSource Is Created for kafka Format by KafkaSourceProvider

Table 1. KafkaSource's Options

Name	Default Value	Description
`kafkaConsumer.pollTimeoutMs`		
		Number of records to fetch per trigger.

maxOffsetsPerTrigger	(empty)	Note: Use `maxOffsetsPerTrigger` option t of records to fetch per trigger. Unless defined, `KafkaSource` requests Kafka(latest offsets.

`startingoffsets`		Possible values:
		• `latest`
		• `earliest`
		• JSON with topics, partitions and their offs
		`{"topicA":{"part":offset,"p1":-1},"top:`
		
		Tip: Use Scala's tripple quotes for the JS(partitions and offsets.
		`option(`
		`  "startingoffsets",`
		`  """{"topic1":{"0":5,"4":-1},"top`
		`)`

assign		Topic subscription strategy that accepts a JSC and partitions, e.g. `{"topicA":[0,1],"topicB":[0,1]}` Note: Exactly one topic subscription strat (that `KafkaSourceProvider` validates `KafkaSource`).
`subscribe`		Topic subscription strategy that accepts topic separated string, e.g. `topic1,topic2,topic3` Note: Exactly one topic subscription strat (that `KafkaSourceProvider` validates `KafkaSource`).

`subscribepattern`		Topic subscription strategy that uses Java's ja for the topic subscription regex pattern of topi e.g.
		`topic\d`
		Tip: Use Scala's tripple quotes for the reg for topic subscription regex pattern.
		`option("subscribepattern", """topi`
		Note: Exactly one topic subscription strat (that `KafkaSourceProvider` validate `KafkaSource`).

```
/**
  ./bin/kafka-console-producer.sh \
    --topic topic1 \
    --broker-list localhost:9092 \
    --property parse.key=true \
    --property key.separator=,
*/
// Extract
val records = spark.
  readStream.
  format("kafka").
  option("subscribepattern", """topic\d"""). // <-- topics with a digit at the end
  option("kafka.bootstrap.servers", "localhost:9092").
  option("startingoffsets", "latest").
  option("maxOffsetsPerTrigger", 1).
  load
// Transform
val result = records.
  select(
    $"key" cast "string",   // deserialize keys
    $"value" cast "string", // deserialize values
    $"topic",
    $"partition",
    $"offset")
// Load
import org.apache.spark.sql.streaming.{OutputMode, Trigger}
import scala.concurrent.duration._
val sq = result.
  writeStream.
  format("console").
  option("truncate", false).
  trigger(Trigger.ProcessingTime(10.seconds)).
  outputMode(OutputMode.Append).
  queryName("from-kafka-to-console").
  start

// In the end, stop the streaming query
sq.stop
```

`KafkaSource` uses a predefined fixed schema (and cannot be changed).

```
scala> records.printSchema
root
 |-- key: binary (nullable = true)
 |-- value: binary (nullable = true)
 |-- topic: string (nullable = true)
 |-- partition: integer (nullable = true)
 |-- offset: long (nullable = true)
 |-- timestamp: timestamp (nullable = true)
 |-- timestampType: integer (nullable = true)
```

Table 2. KafkaSource's Dataset Schema (in the positional order)

Name	Type
`key`	`BinaryType`
`value`	`BinaryType`
`topic`	`StringType`
`partition`	`IntegerType`
`offset`	`LongType`
`timestamp`	`TimestampType`
`timestampType`	`IntegerType`

Tip: Use `cast` method (of `Column`) to cast `BinaryType` to a string (for `key` and `value` columns).

```
$"value" cast "string"
```

`KafkaSource` also supports batch Datasets.

```
val topic1 = spark
  .read // <-- read one batch only
  .format("kafka")
  .option("subscribe", "topic1")
  .option("kafka.bootstrap.servers", "localhost:9092")
  .load
scala> topic1.printSchema
root
 |-- key: binary (nullable = true)
 |-- value: binary (nullable = true)
 |-- topic: string (nullable = true)
 |-- partition: integer (nullable = true)
 |-- offset: long (nullable = true)
 |-- timestamp: timestamp (nullable = true)
 |-- timestampType: integer (nullable = true)
```

Table 3. KafkaSource's Internal Registries and Counters

Name	Description
`currentPartitionOffsets`	Current partition offsets (as `Map[TopicPartition, Long]`) Initially `NONE` and set when `KafkaSource` is requested to get the maximum available offsets or generate a DataFrame with records from Kafka for a batch.

<table>
<tr>
<td>initialPartitionOffsets</td>
<td>
Initial partition offsets (as <code>Map[TopicPartition, Long]</code>)
Set when <code>KafkaSource</code> is first requested to get the available offsets (from metadata log or Kafka directly).
Used when <code>KafkaSource</code> is requested to generate a DataFrame with records from Kafka for a streaming batch (when the start offsets are not defined, i.e. before <code>StreamExecution</code> commits the first streaming batch and so nothing is in committedOffsets registry for a <code>KafkaSource</code> data source yet).
While being initialized, <code>initialPartitionOffsets</code> creates a custom HDFSMetadataLog (with KafkaSourceOffset) and gets the <code>0</code> th batch's metadata (as <code>KafkaSourceOffset</code>) if available.
<table>
<tr><td>Note</td><td><code>initialPartitionOffsets</code> uses a HDFSMetadataLog with custom <code>serialize</code> and <code>deserialize</code> methods to write to and read serialized metadata from the log.</td></tr>
</table>
Otherwise, if the <code>0</code> th batch's metadata is not available, <code>initialPartitionOffsets</code> uses KafkaOffsetReader to fetch offsets per KafkaOffsetRangeLimit input parameter.

For <code>startingOffsets</code> as <code>EarliestOffsetRangeLimit</code> (i.e. <code>earliest</code> in startingoffsets option), <code>initialPartitionOffsets</code> requests for the earliest offsets
For <code>startingOffsets</code> as <code>LatestOffsetRangeLimit</code> (i.e. <code>latest</code> in startingoffsets option), <code>initialPartitionOffsets</code> requests for the latest offsets
For <code>startingOffsets</code> as <code>SpecificOffsetRangeLimit</code> (i.e. a JSON in startingoffsets option), <code>initialPartitionOffsets</code> requests for specific offsets

<code>initialPartitionOffsets</code> adds the offsets to the the metadata log as <code>0</code> th batch.
<table>
<tr><td>Note</td><td>The <code>0</code> th batch is persisted in the streaming metadata log unless stored already.</td></tr>
</table>
You should see the following INFO message in the logs:
<pre>INFO KafkaSource: Initial offsets: [offsets]</pre>
</td>
</tr>
</table>

Tip	Enable `INFO` or `DEBUG` logging levels for `org.apache.spark.sql.kafka010.KafkaSource` to see what happens inside. Add the following line to `conf/log4j.properties`: `log4j.logger.org.apache.spark.sql.kafka010.KafkaSource=DEBUG` Refer to Logging.

`rateLimit` Internal Method

```
rateLimit(
  limit: Long,
  from: Map[TopicPartition, Long],
  until: Map[TopicPartition, Long]): Map[TopicPartition, Long]
```

`rateLimit` requests KafkaOffsetReader to fetchEarliestOffsets.

Caution	FIXME

Note	`rateLimit` is used exclusively when `KafkaSource` gets available offsets (when maxOffsetsPerTrigger option is specified).

`getSortedExecutorList` Method

Caution	FIXME

`reportDataLoss` Internal Method

Caution	FIXME

Note	`reportDataLoss` is used when `KafkaSource` does the following: • fetches and verifies specific offsets • generates a DataFrame with records from Kafka for a batch

Generating DataFrame with Records From Kafka for Streaming Batch — `getBatch` Method

```
getBatch(start: Option[Offset], end: Offset): DataFrame
```

Note	`getBatch` is a part of Source Contract.

`getBatch` initializes initial partition offsets (unless initialized already).

You should see the following INFO message in the logs:

```
INFO KafkaSource: GetBatch called with start = [start], end = [end]
```

`getBatch` requests `KafkaSourceOffset` for end partition offsets for the input `end` offset (known as `untilPartitionOffsets`).

`getBatch` requests `KafkaSourceOffset` for start partition offsets for the input `start` offset (if defined) or uses initial partition offsets (known as `fromPartitionOffsets`).

`getBatch` finds the new partitions (as the difference between the topic partitions in `untilPartitionOffsets` and `fromPartitionOffsets`) and requests KafkaOffsetReader to fetch their earliest offsets.

`getBatch` reports a data loss if the new partitions don't match to what KafkaOffsetReader fetched.

```
Cannot find earliest offsets of [partitions]. Some data may have been missed
```

You should see the following INFO message in the logs:

```
INFO KafkaSource: Partitions added: [partitionOffsets]
```

`getBatch` reports a data loss if the new partitions don't have their offsets `0`.

```
Added partition [partition] starts from [offset] instead of 0. Some data may have been
 missed
```

`getBatch` reports a data loss if the `fromPartitionOffsets` partitions differ from `untilPartitionOffsets` partitions.

```
[partitions] are gone. Some data may have been missed
```

You should see the following DEBUG message in the logs:

```
DEBUG KafkaSource: TopicPartitions: [comma-separated topicPartitions]
```

`getBatch` gets the executors (sorted by `executorId` and `host` of the registered block managers).

Important	That is when `getBatch` goes very low-level to allow for cached `KafkaConsumers` in the executors to be re-used to read the same partition in every batch (aka *location preference*).

You should see the following DEBUG message in the logs:

```
DEBUG KafkaSource: Sorted executors: [comma-separated sortedExecutors]
```

`getBatch` creates a `KafkaSourceRDDOffsetRange` per `TopicPartition`.

`getBatch` filters out `KafkaSourceRDDOffsetRanges` for which until offsets are smaller than from offsets. `getBatch` reports a data loss if they are found.

```
Partition [topicPartition]'s offset was changed from [fromOffset] to [untilOffset], so
me data may have been missed
```

`getBatch` creates a KafkaSourceRDD (with executorKafkaParams, pollTimeoutMs and `reuseKafkaConsumer` flag enabled) and maps it to an RDD of `InternalRow`.

Important	`getBatch` creates a `KafkaSourceRDD` with `reuseKafkaConsumer` flag enabled.

You should see the following INFO message in the logs:

```
INFO KafkaSource: GetBatch generating RDD of offset range: [comma-separated offsetRang
es sorted by topicPartition]
```

`getBatch` sets currentPartitionOffsets if it was empty (which is when…FIXME)

In the end, `getBatch` creates a `DataFrame` from the RDD of `InternalRow` and schema.

Fetching Offsets (From Metadata Log or Kafka Directly) — `getOffset` Method

```
getOffset: Option[Offset]
```

Note	`getOffset` is a part of the Source Contract.

Internally, `getOffset` fetches the initial partition offsets (from the metadata log or Kafka directly).

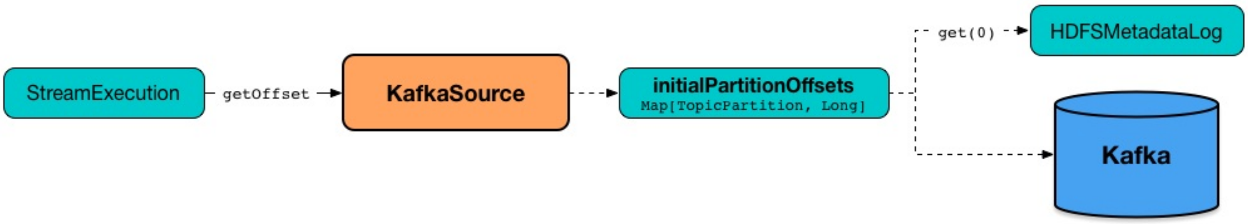

Figure 2. KafkaSource Initializing initialPartitionOffsets While Fetching Initial Offsets

Note	initialPartitionOffsets is a lazy value and is initialized the very first time `getOffset` is called (which is when `StreamExecution` constructs a streaming batch).

```
scala> spark.version
res0: String = 2.3.0-SNAPSHOT

// Case 1: Checkpoint directory undefined
// initialPartitionOffsets read from Kafka directly
val records = spark.
  readStream.
  format("kafka").
  option("subscribe", "topic1").
  option("kafka.bootstrap.servers", "localhost:9092").
  load
// Start the streaming query
// dump records to the console every 10 seconds
import org.apache.spark.sql.streaming.{OutputMode, Trigger}
import scala.concurrent.duration._
val q = records.
  writeStream.
  format("console").
  option("truncate", false).
  trigger(Trigger.ProcessingTime(10.seconds)).
  outputMode(OutputMode.Update).
  start
// Note the temporary checkpoint directory
17/08/07 11:09:29 INFO StreamExecution: Starting [id = 75dd261d-6b62-40fc-a368-9d95d3c
b6f5f, runId = f18a5eb5-ccab-4d9d-8a81-befed41a72bd] with file:///private/var/folders/
0w/kb0d3rqn4zb9fcc91pxhgn8w0000gn/T/temporary-d0055630-24e4-4d9a-8f36-7a12a0f11bc0 to
store the query checkpoint.
...
INFO KafkaSource: Initial offsets: {"topic1":{"0":1}}

// Stop the streaming query
q.stop

// Case 2: Checkpoint directory defined
// initialPartitionOffsets read from Kafka directly
// since the checkpoint directory is not available yet
// it will be the next time the query is started
```

```
val records = spark.
  readStream.
  format("kafka").
  option("subscribe", "topic1").
  option("kafka.bootstrap.servers", "localhost:9092").
  load.
  select($"value" cast "string", $"topic", $"partition", $"offset")
import org.apache.spark.sql.streaming.{OutputMode, Trigger}
import scala.concurrent.duration._
val q = records.
  writeStream.
  format("console").
  option("truncate", false).
  option("checkpointLocation", "/tmp/checkpoint"). // <-- checkpoint directory
  trigger(Trigger.ProcessingTime(10.seconds)).
  outputMode(OutputMode.Update).
  start
// Note the checkpoint directory in use
17/08/07 11:21:25 INFO StreamExecution: Starting [id = b8f59854-61c1-4c2f-931d-62bbaf9
0ee3b, runId = 70d06a3b-f2b1-4fa8-a518-15df4cf59130] with file:///tmp/checkpoint to st
ore the query checkpoint.
...
INFO KafkaSource: Initial offsets: {"topic1":{"0":1}}
...
INFO StreamExecution: Stored offsets for batch 0. Metadata OffsetSeqMetadata(0,1502098
526848,Map(spark.sql.shuffle.partitions -> 200, spark.sql.streaming.stateStore.provide
rClass -> org.apache.spark.sql.execution.streaming.state.HDFSBackedStateStoreProvider)
)

// Review the checkpoint location
// $ ls -ltr /tmp/checkpoint/offsets
// total 8
// -rw-r--r--  1 jacek  wheel  248  7 sie 11:21 0
// $ tail -2 /tmp/checkpoint/offsets/0 | jq

// Produce messages to Kafka so the latest offset changes
// And more importanly the offset gets stored to checkpoint location
-------------------------------------------
Batch: 1
-------------------------------------------
+---------------------------+------+---------+------+
|value                      |topic |partition|offset|
+---------------------------+------+---------+------+
|testing checkpoint location|topic1|0        |2     |
+---------------------------+------+---------+------+

// and one more
// Note the offset
-------------------------------------------
Batch: 2
-------------------------------------------
+------------+------+---------+------+
|value       |topic |partition|offset|
```

```
+------------+------+---------+------+
|another test|topic1|0        |3     |
+------------+------+---------+------+

// See what was checkpointed
// $ ls -ltr /tmp/checkpoint/offsets
// total 24
// -rw-r--r--  1 jacek  wheel  248  7 sie 11:35 0
// -rw-r--r--  1 jacek  wheel  248  7 sie 11:37 1
// -rw-r--r--  1 jacek  wheel  248  7 sie 11:38 2
// $ tail -2 /tmp/checkpoint/offsets/2 | jq

// Stop the streaming query
q.stop

// And start over to see what offset the query starts from
// Checkpoint location should have the offsets
val q = records.
  writeStream.
  format("console").
  option("truncate", false).
  option("checkpointLocation", "/tmp/checkpoint"). // <-- checkpoint directory
  trigger(Trigger.ProcessingTime(10.seconds)).
  outputMode(OutputMode.Update).
  start
// Whoops...console format does not support recovery (!)
// Reported as https://issues.apache.org/jira/browse/SPARK-21667
org.apache.spark.sql.AnalysisException: This query does not support recovering from ch
eckpoint location. Delete /tmp/checkpoint/offsets to start over.;
  at org.apache.spark.sql.streaming.StreamingQueryManager.createQuery(StreamingQueryMa
nager.scala:222)
  at org.apache.spark.sql.streaming.StreamingQueryManager.startQuery(StreamingQueryMan
ager.scala:278)
  at org.apache.spark.sql.streaming.DataStreamWriter.start(DataStreamWriter.scala:284)
  ... 61 elided

// Change the sink (= output format) to JSON
val q = records.
  writeStream.
  format("json").
  option("path", "/tmp/json-sink").
  option("checkpointLocation", "/tmp/checkpoint"). // <-- checkpoint directory
  trigger(Trigger.ProcessingTime(10.seconds)).
  start
// Note the checkpoint directory in use
17/08/07 12:09:02 INFO StreamExecution: Starting [id = 02e00924-5f0d-4501-bcb8-80be8a8
be385, runId = 5eba2576-dad6-4f95-9031-e72514475edc] with file:///tmp/checkpoint to st
ore the query checkpoint.
...
17/08/07 12:09:02 INFO KafkaSource: GetBatch called with start = Some({"topic1":{"0":3
}}), end = {"topic1":{"0":4}}
17/08/07 12:09:02 INFO KafkaSource: Partitions added: Map()
17/08/07 12:09:02 DEBUG KafkaSource: TopicPartitions: topic1-0
```

```
17/08/07 12:09:02 DEBUG KafkaSource: Sorted executors:
17/08/07 12:09:02 INFO KafkaSource: GetBatch generating RDD of offset range: KafkaSour
ceRDDOffsetRange(topic1-0,3,4,None)
17/08/07 12:09:03 DEBUG KafkaOffsetReader: Partitions assigned to consumer: [topic1-0]
. Seeking to the end.
17/08/07 12:09:03 DEBUG KafkaOffsetReader: Got latest offsets for partition : Map(topi
c1-0 -> 4)
17/08/07 12:09:03 DEBUG KafkaSource: GetOffset: ArrayBuffer((topic1-0,4))
17/08/07 12:09:03 DEBUG StreamExecution: getOffset took 122 ms
17/08/07 12:09:03 DEBUG StreamExecution: Resuming at batch 3 with committed offsets {K
afkaSource[Subscribe[topic1]]: {"topic1":{"0":4}}} and available offsets {KafkaSource[
Subscribe[topic1]]: {"topic1":{"0":4}}}
17/08/07 12:09:03 DEBUG StreamExecution: Stream running from {KafkaSource[Subscribe[to
pic1]]: {"topic1":{"0":4}}} to {KafkaSource[Subscribe[topic1]]: {"topic1":{"0":4}}}
```

`getOffset` requests KafkaOffsetReader to fetchLatestOffsets (known later as `latest`).

Note	(Possible performance degradation?) It is possible that `getOffset` will request the latest offsets from Kafka twice, i.e. while initializing initialPartitionOffsets (when no metadata log is available and KafkaSource's KafkaOffsetRangeLimit is `LatestOffsetRangeLimit`) and always as part of `getOffset` itself.

`getOffset` then calculates currentPartitionOffsets based on the maxOffsetsPerTrigger option.

Table 4. getOffset's Offset Calculation per maxOffsetsPerTrigger

maxOffsetsPerTrigger	Offsets
Unspecified (i.e. `None`)	`latest`
Defined (but currentPartitionOffsets is empty)	rateLimit with `limit` limit, initialPartitionOffsets as `from`, `until` as `latest`
Defined (and currentPartitionOffsets contains partitions and offsets)	rateLimit with `limit` limit, currentPartitionOffsets as `from`, `until` as `latest`

You should see the following DEBUG message in the logs:

```
DEBUG KafkaSource: GetOffset: [offsets]
```

In the end, `getOffset` creates a KafkaSourceOffset with `offsets` (as `Map[TopicPartition, Long]`).

Creating KafkaSource Instance

`KafkaSource` takes the following when created:

- SQLContext
- KafkaOffsetReader
- Parameters of executors (reading from Kafka)
- Collection of key-value options
- `metadataPath` — streaming metadata log directory where `KafkaSource` persists KafkaSourceOffset offsets in JSON format.
- `KafkaOffsetRangeLimit` (as defined using startingoffsets option)
- Flag used to create `KafkaSourceRDDs` every trigger and when checking to report a IllegalStateException on data loss.

`KafkaSource` initializes the internal registries and counters.

Fetching and Verifying Specific Offsets — `fetchAndVerify` Internal Method

```
fetchAndVerify(specificOffsets: Map[TopicPartition, Long]): KafkaSourceOffset
```

`fetchAndVerify` requests KafkaOffsetReader to fetchSpecificOffsets for the given `specificOffsets`.

`fetchAndVerify` makes sure that the starting offsets in `specificOffsets` are the same as in Kafka and reports a data loss otherwise.

```
startingOffsets for [tp] was [off] but consumer reset to [result(tp)]
```

In the end, `fetchAndVerify` creates a KafkaSourceOffset (with the result of KafkaOffsetReader).

Note	`fetchAndVerify` is used exclusively when `KafkaSource` initializes initial partition offsets.

KafkaRelation

`KafkaRelation` is…FIXME

`KafkaRelation` is created when…FIXME

Creating KafkaRelation Instance

`KafkaRelation` takes the following when created:

- `SQLContext`
- ConsumerStrategy
- `Source` options
- User-defined Kafka parameters
- `failOnDataLoss` flag
- `KafkaOffsetRangeLimit`
- `KafkaOffsetRangeLimit`

`KafkaRelation` initializes the internal registries and counters.

`getPartitionOffsets` Internal Method

```
getPartitionOffsets(
  kafkaReader: KafkaOffsetReader,
  kafkaOffsets: KafkaOffsetRangeLimit): Map[TopicPartition, Long]
```

Caution	FIXME

Note	`getPartitionOffsets` is used exclusively when `KafkaRelation` builds RDD of rows (from the tuples).

Building RDD with Records (from Topics) — `buildScan` Method

```
buildScan(): RDD[Row]
```

Note	`buildScan` is a part of `TableScan` contract.

Caution	FIXME

KafkaSourceRDD

`KafkaSourceRDD` is an `RDD` of Kafka's ConsumerRecords (with keys and values being collections of bytes, i.e. `Array[Byte]`).

`KafkaSourceRDD` is created when:

- `KafkaRelation` buildScan
- `KafkaSource` getBatch

`getPreferredLocations` Method

Caution	FIXME

`compute` Method

Caution	FIXME

`getPartitions` Method

Caution	FIXME

`persist` Method

Caution	FIXME

Creating KafkaSourceRDD Instance

`KafkaSourceRDD` takes the following when created:

- `SparkContext`
- Collection of key-value settings for executors reading records from Kafka topics
- Collection of `KafkaSourceRDDOffsetRange` offsets
- Timeout (in milliseconds) to poll data from Kafka

  Used when `KafkaSourceRDD` is requested for records (for given offsets) and in turn requests `CachedKafkaConsumer` to poll for Kafka's `ConsumerRecords` .

- Flag to…FIXME
- Flag to…FIXME

`KafkaSourceRDD` initializes the internal registries and counters.

CachedKafkaConsumer

Caution	FIXME

`poll` Internal Method

Caution	FIXME

`fetchData` Internal Method

Caution	FIXME

KafkaOffsetReader

`KafkaOffsetReader` is created when:

- `KafkaRelation` builds an RDD with rows that are records from Kafka
- `KafkaSourceProvider` creates a KafkaSource (for **kafka** format)

Table 1. KafkaOffsetReader's Options

Name	Default Value	Description
`fetchOffset.numRetries`	`3`	
`fetchOffset.retryIntervalMs`	`1000`	How long to wait before retries.

`KafkaOffsetReader` defines the predefined fixed schema of Kafka source.

Table 2. KafkaOffsetReader's Internal Registries and Counters

Name	Description
`consumer`	Kafka's Consumer (with keys and values of `Array[Byte]` type) Initialized when `KafkaOffsetReader` is created. Used when `KafkaOffsetReader`: • fetchTopicPartitions • fetches offsets for selected TopicPartitions • fetchEarliestOffsets • fetchLatestOffsets • resetConsumer • is closed
`execContext`	
`groupId`	
`kafkaReaderThread`	
`maxOffsetFetchAttempts`	
`nextId`	
`offsetFetchAttemptIntervalMs`	

Tip	Enable `INFO` or `DEBUG` logging levels for `org.apache.spark.sql.kafka010.KafkaOffsetReader` to see what happens inside. Add the following line to `conf/log4j.properties` : `log4j.logger.org.apache.spark.sql.kafka010.KafkaOffsetReader=DEBUG` Refer to Logging.

`nextGroupId` Internal Method

Caution	FIXME

`resetConsumer` Internal Method

Caution	FIXME

`fetchTopicPartitions` Method

```
fetchTopicPartitions(): Set[TopicPartition]
```

Caution	FIXME

Note	`fetchTopicPartitions` is used when `KafkaRelation` getPartitionOffsets.

Fetching Earliest Offsets — `fetchEarliestOffsets` Method

```
fetchEarliestOffsets(newPartitions: Seq[TopicPartition]): Map[TopicPartition, Long]
```

Caution	FIXME

Note	`fetchEarliestOffsets` is used when `KafkaSource` rateLimit and generates a DataFrame for a batch (when new partitions have been assigned).

Fetching Latest Offsets — `fetchLatestOffsets` Method

```
fetchLatestOffsets(): Map[TopicPartition, Long]
```

Caution	FIXME

Note	`fetchLatestOffsets` is used when `KafkaSource` gets offsets or `initialPartitionOffsets` is initialized.

`withRetriesWithoutInterrupt` Internal Method

```
withRetriesWithoutInterrupt(body: => Map[TopicPartition, Long]): Map[TopicPartition, L
ong]
```

Creating KafkaOffsetReader Instance

`KafkaOffsetReader` takes the following when created:

- ConsumerStrategy
- Kafka parameters (as name-value pairs that are used exclusively to create a Kafka consumer
- Options (as name-value pairs)
- Prefix for group id

`KafkaOffsetReader` initializes the internal registries and counters.

Fetching Offsets for Selected TopicPartitions — `fetchSpecificOffsets` Method

```
fetchSpecificOffsets(partitionOffsets: Map[TopicPartition, Long]): Map[TopicPartition,
Long]
```

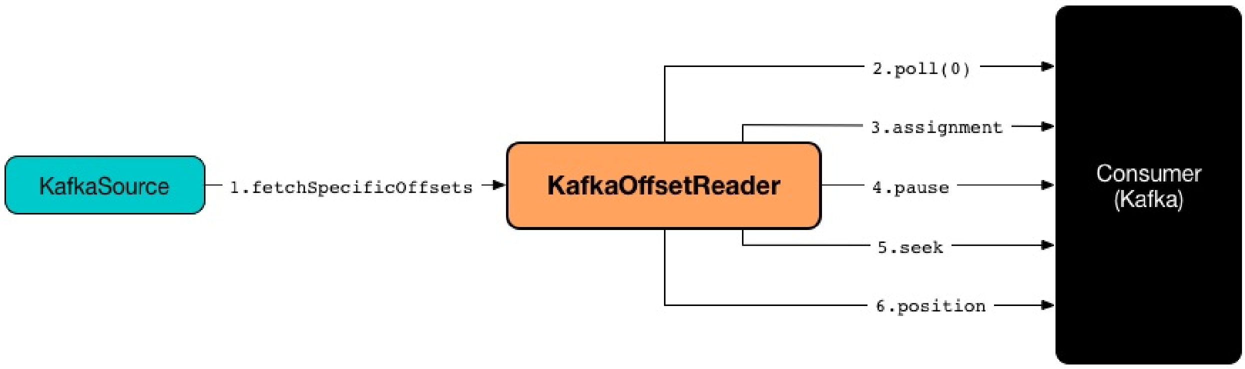

Figure 1. KafkaOffsetReader's fetchSpecificOffsets

`fetchSpecificOffsets` requests the Kafka Consumer to `poll(0)`.

`fetchSpecificOffsets` requests the Kafka Consumer for assigned partitions (using `Consumer.assignment()`).

`fetchSpecificOffsets` requests the Kafka Consumer to `pause(partitions)`.

You should see the following DEBUG message in the logs:

```
DEBUG KafkaOffsetReader: Partitions assigned to consumer: [partitions]. Seeking to [pa
rtitionOffsets]
```

For every partition offset in the input `partitionOffsets`, `fetchSpecificOffsets` requests the Kafka Consumer to:

- `seekToEnd` for the latest (aka `-1`)
- `seekToBeginning` for the earliest (aka `-2`)
- `seek` for other offsets

In the end, `fetchSpecificOffsets` creates a collection of Kafka's `TopicPartition` and `position` (using the Kafka Consumer).

Note	`fetchSpecificOffsets` is used when `KafkaSource` fetches and verifies initial partition offsets.

Creating Kafka Consumer — `createConsumer` Internal Method

```
createConsumer(): Consumer[Array[Byte], Array[Byte]]
```

`createConsumer` requests ConsumerStrategy to create a Kafka Consumer with driverKafkaParams and new generated group.id Kafka property.

Note	`createConsumer` is used when `KafkaOffsetReader` is created (and initializes consumer) and resetConsumer

ConsumerStrategy Contract for KafkaConsumer Providers

`ConsumerStrategy` is the contract for components that can create a KafkaConsumer using the given Kafka parameters.

```
createConsumer(kafkaParams: java.util.Map[String, Object]): Consumer[Array[Byte], Array
[Byte]]
```

Table 1. Available ConsumerStrategies

ConsumerStrategy	createConsumer
`AssignStrategy`	Uses KafkaConsumer.assign(Collection<TopicPartition> partitions)
`SubscribeStrategy`	Uses KafkaConsumer.subscribe(Collection<String> topics)
`SubscribePatternStrategy`	Uses KafkaConsumer.subscribe(Pattern pattern, ConsumerRebalanceListener listener) with `NoOpConsumerRebalanceListener`. Tip: Refer to java.util.regex.Pattern for the format of supported topic subscription regex patterns.

KafkaSourceOffset

Caution	FIXME

`KafkaSourceOffset` is created for `partitionToOffsets` collection of `TopicPartitions` and their offsets.

Creating KafkaSourceOffset Instance

Caution	FIXME

Getting Partition Offsets — `getPartitionOffsets` Method

```
getPartitionOffsets(offset: Offset): Map[TopicPartition, Long]
```

`getPartitionOffsets` takes KafkaSourceOffset.partitionToOffsets from `offset`.

If `offset` is `KafkaSourceOffset`, `getPartitionOffsets` takes the partitions and offsets straight from it.

If however `offset` is `SerializedOffset`, `getPartitionOffsets` deserializes the offsets from JSON.

`getPartitionOffsets` reports an `IllegalArgumentException` when `offset` is neither `KafkaSourceOffset` or `SerializedOffset`.

```
Invalid conversion from offset of [class] to KafkaSourceOffset
```

Note	`getPartitionOffsets` is used exclusively when `KafkaSource` generates a DataFrame with records from Kafka for a batch.

KafkaSink

`KafkaSink` is a streaming sink that KafkaSourceProvider registers as the `kafka` format.

```
// start spark-shell or a Spark application with spark-sql-kafka-0-10 module
// spark-shell --packages org.apache.spark:spark-sql-kafka-0-10_2.11:2.3.0-SNAPSHOT
import org.apache.spark.sql.SparkSession
val spark: SparkSession = ...
spark.
  readStream.
  format("text").
  load("server-logs/*.out").
  as[String].
  writeStream.
  queryName("server-logs processor").
  format("kafka").  // <-- uses KafkaSink
  option("topic", "topic1").
  option("checkpointLocation", "/tmp/kafka-sink-checkpoint"). // <-- mandatory
  start

// in another terminal
$ echo hello > server-logs/hello.out

// in the terminal with Spark
FIXME
```

Creating KafkaSink Instance

`KafkaSink` takes the following when created:

- `SQLContext`
- Kafka parameters (used on executor) as a map of `(String, Object)` pairs
- Optional topic name

`addBatch` Method

```
addBatch(batchId: Long, data: DataFrame): Unit
```

Internally, `addBatch` requests `KafkaWriter` to write the input `data` to the topic (if defined) or a topic in executorKafkaParams.

Note	`addBatch` is a part of Sink Contract to "add" a batch of data to the sink.

FileStreamSource

`FileStreamSource` is a Source that reads text files from `path` directory as they appear. It uses `LongOffset` offsets.

Note	It is used by DataSource.createSource for `FileFormat` .

You can provide the schema of the data and `dataFrameBuilder` - the function to build a `DataFrame` in getBatch at instantiation time.

```
// NOTE The source directory must exist
// mkdir text-logs

val df = spark.readStream
  .format("text")
  .option("maxFilesPerTrigger", 1)
  .load("text-logs")

scala> df.printSchema
root
|-- value: string (nullable = true)
```

Batches are indexed.

It lives in `org.apache.spark.sql.execution.streaming` package.

```
import org.apache.spark.sql.types._
val schema = StructType(
  StructField("id", LongType, nullable = false) ::
  StructField("name", StringType, nullable = false) ::
  StructField("score", DoubleType, nullable = false) :: Nil)

// You should have input-json directory available
val in = spark.readStream
  .format("json")
  .schema(schema)
  .load("input-json")

val input = in.transform { ds =>
  println("transform executed")  // <-- it's going to be executed once only
  ds
}

scala> input.isStreaming
res9: Boolean = true
```

It tracks already-processed files in `seenFiles` hash map.

Tip	Enable `DEBUG` or `TRACE` logging level for `org.apache.spark.sql.execution.streaming.FileStreamSource` to see what happens inside. Add the following line to `conf/log4j.properties` : `log4j.logger.org.apache.spark.sql.execution.streaming.FileStreamSource=TRACE` Refer to Logging.

Creating FileStreamSource Instance

Caution	FIXME

Options

`maxFilesPerTrigger`

`maxFilesPerTrigger` option specifies the maximum number of files per trigger (batch). It limits the file stream source to read the `maxFilesPerTrigger` number of files specified at a time and hence enables rate limiting.

It allows for a static set of files be used like a stream for testing as the file set is processed `maxFilesPerTrigger` number of files at a time.

`schema`

If the schema is specified at instantiation time (using optional `dataSchema` constructor parameter) it is returned.

Otherwise, `fetchAllFiles` internal method is called to list all the files in a directory.

When there is at least one file the schema is calculated using `dataFrameBuilder` constructor parameter function. Else, an `IllegalArgumentException("No schema specified")` is thrown unless it is for **text** provider (as `providerName` constructor parameter) where the default schema with a single `value` column of type `StringType` is assumed.

Note	**text** as the value of `providerName` constructor parameter denotes **text file stream provider**.

`getOffset` Method

The maximum offset (`getOffset`) is calculated by fetching all the files in `path` excluding files that start with `_` (underscore).

When computing the maximum offset using `getOffset` , you should see the following DEBUG message in the logs:

```
DEBUG Listed ${files.size} in ${(endTime.toDouble - startTime) / 1000000}ms
```

When computing the maximum offset using `getOffset` , it also filters out the files that were already seen (tracked in `seenFiles` internal registry).

You should see the following DEBUG message in the logs (depending on the status of a file):

```
new file: $file
// or
old file: $file
```

Generating DataFrame for Streaming Batch — `getBatch` Method

`FileStreamSource.getBatch` asks metadataLog for the batch.

You should see the following INFO and DEBUG messages in the logs:

```
INFO Processing ${files.length} files from ${startId + 1}:$endId
DEBUG Streaming ${files.mkString(", ")}
```

The method to create a result batch is given at instantiation time (as `dataFrameBuilder` constructor parameter).

`metadataLog`

`metadataLog` is a metadata storage using `metadataPath` path (which is a constructor parameter).

Note	It extends `HDFSMetadataLog[Seq[String]]` .

Caution	FIXME Review `HDFSMetadataLog`

FileStreamSink — Streaming Sink for Parquet Format

`FileStreamSink` is the streaming sink that writes out the results of a streaming query to `parquet` files.

```
import scala.concurrent.duration._
import org.apache.spark.sql.streaming.{OutputMode, Trigger}
val out = in.
  writeStream.
  format("parquet").
  option("path", "parquet-output-dir").
  option("checkpointLocation", "checkpoint-dir").
  trigger(Trigger.ProcessingTime(10.seconds)).
  outputMode(OutputMode.Append).
  start
```

`FileStreamSink` is created exclusively when `DataSource` is requested to create a streaming sink.

`FileStreamSink` supports Append output mode only.

`FileStreamSink` uses spark.sql.streaming.fileSink.log.deletion (as `isDeletingExpiredLog`)

The textual representation of `FileStreamSink` is **FileSink[path]**

Table 1. FileStreamSink's Internal Properties (e.g. Registries, Counters and Flags)

Name	Description
`basePath`	FIXME Used when…FIXME
`logPath`	FIXME Used when…FIXME
`fileLog`	FIXME Used when…FIXME
`hadoopConf`	FIXME Used when…FIXME

`addBatch` Method

```
addBatch(batchId: Long, data: DataFrame): Unit
```

Note	`addBatch` is a part of Sink Contract to "add" a batch of data to the sink.

`addBatch` …FIXME

Creating FileStreamSink Instance

`FileStreamSink` takes the following when created:

- `SparkSession`
- Path with the metadata directory
- `FileFormat`
- Names of the partition columns
- Configuration options

`FileStreamSink` initializes the internal registries and counters.

RateSourceProvider

`RateSourceProvider` is a StreamSourceProvider for RateStreamSource (that acts as the source for **rate** format).

Note	`RateSourceProvider` is also a `DataSourceRegister`.

The short name of the data source is **rate**.

RateStreamSource

`RateStreamSource` is a streaming source that generates consecutive numbers with timestamp that can be useful for testing and PoCs.

`RateStreamSource` is created for **rate** format (that is registered by RateSourceProvider).

```
val rates = spark.
  readStream.
  format("rate"). // <-- use RateStreamSource
  option("rowsPerSecond", 1).
  load
```

Table 1. RateStreamSource's Options

Name	Default Value	Description
`numPartitions`	(default parallelism)	Number of partitions to use
`rampUpTime`	`0` (seconds)	
`rowsPerSecond`	`1`	Number of rows to generate per second (has to be greater than `0`)

`RateStreamSource` uses a predefined schema that cannot be changed.

```
val schema = rates.schema
scala> println(schema.treeString)
root
 |-- timestamp: timestamp (nullable = true)
 |-- value: long (nullable = true)
```

Table 2. RateStreamSource's Dataset Schema (in the positional order)

Name	Type
`timestamp`	`TimestampType`
`value`	`LongType`

Table 3. RateStreamSource's Internal Registries and Counters

Name	Description
`clock`	
`lastTimeMs`	
`maxSeconds`	
`startTimeMs`	

Tip	Enable `INFO` or `DEBUG` logging levels for `org.apache.spark.sql.execution.streaming.RateStreamSource` to see what happens inside. Add the following line to `conf/log4j.properties`: `log4j.logger.org.apache.spark.sql.execution.streaming.RateStreamSource=DEBUG` Refer to Logging.

Getting Maximum Available Offsets — `getOffset` Method

```
getOffset: Option[Offset]
```

Note	`getOffset` is a part of the Source Contract.

Caution	FIXME

Generating DataFrame for Streaming Batch — `getBatch` Method

```
getBatch(start: Option[Offset], end: Offset): DataFrame
```

Note	`getBatch` is a part of Source Contract.

Internally, `getBatch` calculates the seconds to start from and end at (from the input `start` and `end` offsets) or assumes `0`.

`getBatch` then calculates the values to generate for the start and end seconds.

You should see the following DEBUG message in the logs:

```
DEBUG RateStreamSource: startSeconds: [startSeconds], endSeconds: [endSeconds], rangeS
tart: [rangeStart], rangeEnd: [rangeEnd]
```

If the start and end ranges are equal, `getBatch` creates an empty `DataFrame` (with the schema) and returns.

Otherwise, when the ranges are different, `getBatch` creates a `DataFrame` using `SparkContext.range` operator (for the start and end ranges and numPartitions partitions).

Creating RateStreamSource Instance

`RateStreamSource` takes the following when created:

- `SQLContext`
- Path to the metadata
- Rows per second
- RampUp time in seconds
- Number of partitions
- Flag to whether to use `ManualClock` (`true`) or `SystemClock` (`false`)

`RateStreamSource` initializes the internal registries and counters.

TextSocketSourceProvider

`TextSocketSourceProvider` is a StreamSourceProvider for TextSocketSource that read records from `host` and `port` .

`TextSocketSourceProvider` is a DataSourceRegister, too.

The short name of the data source is `socket` .

It requires two mandatory options (that you can set using `option` method):

1. `host` which is the host name.
2. `port` which is the port number. It must be an integer.

`TextSocketSourceProvider` also supports includeTimestamp option that is a boolean flag that you can use to include timestamps in the schema.

includeTimestamp Option

Caution	FIXME

createSource

`createSource` grabs the two mandatory options — `host` and `port` — and returns an TextSocketSource.

sourceSchema

`sourceSchema` returns `textSocket` as the name of the source and the schema that can be one of the two available schemas:

1. `SCHEMA_REGULAR` (default) which is a schema with a single `value` field of String type.
2. `SCHEMA_TIMESTAMP` when `includeTimestamp` flag option is set. It is not, i.e. `false` , by default. The schema are `value` field of `StringType` type and `timestamp` field of TimestampType type of format `yyyy-MM-dd HH:mm:ss` .

Tip	Read about schema.

Internally, it starts by printing out the following WARN message to the logs:

```
WARN TextSocketSourceProvider: The socket source should not be used for production app
lications! It does not support recovery and stores state indefinitely.
```

It then checks whether `host` and `port` parameters are defined and if not it throws a `AnalysisException`:

```
Set a host to read from with option("host", ...).
```

TextSocketSource

`TextSocketSource` is a streaming source that reads lines from a socket at the `host` and `port` (defined by parameters).

It uses lines internal in-memory buffer to keep all of the lines that were read from a socket forever.

Caution	This source is **not** for production use due to design contraints, e.g. infinite in-memory collection of lines read and no fault recovery. It is designed only for tutorials and debugging.

```
import org.apache.spark.sql.SparkSession
val spark: SparkSession = SparkSession.builder.getOrCreate()

// Connect to localhost:9999
// You can use "nc -lk 9999" for demos
val textSocket = spark.
  readStream.
  format("socket").
  option("host", "localhost").
  option("port", 9999).
  load

import org.apache.spark.sql.Dataset
val lines: Dataset[String] = textSocket.as[String].map(_.toUpperCase)

val query = lines.writeStream.format("console").start

// Start typing the lines in nc session
// They will appear UPPERCASE in the terminal

-------------------------------------------
Batch: 0
-------------------------------------------
+---------+
|    value|
+---------+
|UPPERCASE|
+---------+

scala> query.explain
== Physical Plan ==
*SerializeFromObject [staticinvoke(class org.apache.spark.unsafe.types.UTF8String, Str
ingType, fromString, input[0, java.lang.String, true], true) AS value#21]
+- *MapElements <function1>, obj#20: java.lang.String
   +- *DeserializeToObject value#43.toString, obj#19: java.lang.String
      +- LocalTableScan [value#43]

scala> query.stop
```

lines Internal Buffer

```
lines: ArrayBuffer[(String, Timestamp)]
```

`lines` is the internal buffer of all the lines `TextSocketSource` read from the socket.

Maximum Available Offset (getOffset method)

Note	`getOffset` is a part of the Streaming Source Contract.

`TextSocketSource`'s offset can either be none or `LongOffset` of the number of lines in the internal lines buffer.

Schema (schema method)

`TextSocketSource` supports two schemas:

1. A single `value` field of String type.
2. `value` field of `StringType` type and `timestamp` field of TimestampType type of format `yyyy-MM-dd HH:mm:ss`.

Tip	Refer to sourceSchema for `TextSocketSourceProvider`.

Creating TextSocketSource Instance

```
TextSocketSource(
  host: String,
  port: Int,
  includeTimestamp: Boolean,
  sqlContext: SQLContext)
```

When `TextSocketSource` is created (see TextSocketSourceProvider), it gets 4 parameters passed in:

1. `host`
2. `port`
3. includeTimestamp flag
4. SQLContext

Caution	It appears that the source did not get "renewed" to use SparkSession instead.

It opens a socket at given `host` and `port` parameters and reads a buffering character-input stream using the default charset and the default-sized input buffer (of `8192` bytes) line by line.

Caution	FIXME Review Java's `Charset.defaultCharset()`

It starts a `readThread` daemon thread (called `TextSocketSource(host, port)`) to read lines from the socket. The lines are added to the internal lines buffer.

Stopping TextSocketSource (stop method)

When stopped, `TextSocketSource` closes the socket connection.

MemorySink

`MemorySink` is a streaming Sink that stores records in memory. It is particularly useful for testing.

`MemorySink` is used for `memory` format and requires a query name (by `queryName` method or `queryName` option).

```
val spark: SparkSession = ???
val logs = spark.readStream.textFile("logs/*.out")

scala> val outStream = logs.writeStream
  .format("memory")
  .queryName("logs")
  .start()
outStream: org.apache.spark.sql.streaming.StreamingQuery = org.apache.spark.sql.execut
ion.streaming.StreamingQueryWrapper@690337df

scala> sql("select * from logs").show(truncate = false)
```

Note	`MemorySink` was introduced in the pull request for [SPARK-14288][SQL] Memory Sink for streaming.

Use `toDebugString` to see the batches.

Its aim is to allow users to test streaming applications in the Spark shell or other local tests.

You can set `checkpointLocation` using `option` method or it will be set to spark.sql.streaming.checkpointLocation property.

If `spark.sql.streaming.checkpointLocation` is set, the code uses `$location/$queryName` directory.

Finally, when no `spark.sql.streaming.checkpointLocation` is set, a temporary directory `memory.stream` under `java.io.tmpdir` is used with `offsets` subdirectory inside.

Note	The directory is cleaned up at shutdown using `ShutdownHookManager.registerShutdownDeleteDir`.

It creates `MemorySink` instance based on the schema of the DataFrame it operates on.

It creates a new DataFrame using `MemoryPlan` with `MemorySink` instance created earlier and registers it as a temporary table (using DataFrame.registerTempTable method).

Note	At this point you can query the table as if it were a regular non-streaming table using sql method.

A new StreamingQuery is started (using StreamingQueryManager.startQuery) and returned.

Table 1. MemorySink's Internal Registries and Counters

Name	Description
`batches`	FIXME Used when…FIXME

Tip	Enable `DEBUG` logging level for `org.apache.spark.sql.execution.streaming.MemorySink` logger to see what happens in `MemorySink`. Add the following line to `conf/log4j.properties`: `log4j.logger.org.apache.spark.sql.execution.streaming.MemorySink=DEBUG` Refer to Logging.

`addBatch` Method

```
addBatch(batchId: Long, data: DataFrame): Unit
```

`addBatch` checks if `batchId` has already been committed (i.e. added to batches internal registry).

If `batchId` was already committed, you should see the following DEBUG message in the logs:

```
DEBUG Skipping already committed batch: [batchId]
```

Otherwise, if the `batchId` is not already committed, you should see the following DEBUG message in the logs:

```
DEBUG Committing batch [batchId] to [this]
```

For `Append` and `Update` output modes, `addBatch` collects records from `data` and registers `batchId` (i.e. adds to batches internal registry).

Note	`addBatch` uses `collect` operator to collect records. It is when the records are "downloaded" to the driver's memory.

For `Complete` output mode, `addBatch` collects records (as for the other output modes), but before registering `batchId` clears batches internal registry.

When the output mode is invalid, `addBatch` reports a `IllegalArgumentException` with the following error message.

```
Output mode [outputMode] is not supported by MemorySink
```

Note	`addBatch` is a part of Sink Contract to "add" a batch of data to the sink.

MemoryStream

`MemoryStream` is a streaming Source that produces values to memory.

`MemoryStream` uses the internal batches collection of datasets.

Caution	This source is **not** for production use due to design contraints, e.g. infinite in-memory collection of lines read and no fault recovery. `MemoryStream` is designed primarily for unit tests, tutorials and debugging.

```
val spark: SparkSession = ???

implicit val ctx = spark.sqlContext

import org.apache.spark.sql.execution.streaming.MemoryStream
// It uses two implicits: Encoder[Int] and SQLContext
val intsIn = MemoryStream[Int]

val ints = intsIn.toDF
  .withColumn("t", current_timestamp())
  .withWatermark("t", "5 minutes")
  .groupBy(window($"t", "5 minutes") as "window")
  .agg(count("*") as "total")

import org.apache.spark.sql.streaming.{OutputMode, Trigger}
import scala.concurrent.duration._
val totalsOver5mins = ints.
  writeStream.
  format("memory").
  queryName("totalsOver5mins").
  outputMode(OutputMode.Append).
  trigger(Trigger.ProcessingTime(10.seconds)).
  start

scala> val zeroOffset = intsIn.addData(0, 1, 2)
zeroOffset: org.apache.spark.sql.execution.streaming.Offset = #0

totalsOver5mins.processAllAvailable()
spark.table("totalsOver5mins").show

scala> intsOut.show
+-----+
|value|
+-----+
|    0|
|    1|
|    2|
+-----+

memoryQuery.stop()
```

```
17/02/28 20:06:01 DEBUG StreamExecution: Starting Trigger Calculation
17/02/28 20:06:01 DEBUG StreamExecution: getOffset took 0 ms
17/02/28 20:06:01 DEBUG StreamExecution: triggerExecution took 0 ms
17/02/28 20:06:01 DEBUG StreamExecution: Execution stats: ExecutionStats(Map(),List(),
Map(watermark -> 1970-01-01T00:00:00.000Z))
17/02/28 20:06:01 INFO StreamExecution: Streaming query made progress: {
  "id" : "ec5addda-0e46-4c3c-b2c2-604a854ee19a",
  "runId" : "d850cabc-94d0-4931-8a2d-e054086e39c3",
  "name" : "totalsOver5mins",
  "timestamp" : "2017-02-28T19:06:01.175Z",
  "numInputRows" : 0,
  "inputRowsPerSecond" : 0.0,
  "durationMs" : {
    "getOffset" : 0,
    "triggerExecution" : 0
  },
  "eventTime" : {
    "watermark" : "1970-01-01T00:00:00.000Z"
  },
  "stateOperators" : [ ],
  "sources" : [ {
    "description" : "MemoryStream[value#1]",
    "startOffset" : null,
    "endOffset" : null,
    "numInputRows" : 0,
    "inputRowsPerSecond" : 0.0
  } ],
  "sink" : {
    "description" : "MemorySink"
  }
}
```

Tip

Enable `DEBUG` logging level for `org.apache.spark.sql.execution.streaming.MemoryStream` logger to see what happens inside.

Add the following line to `conf/log4j.properties` :

```
log4j.logger.org.apache.spark.sql.execution.streaming.MemoryStream=DEBUG
```

Refer to Logging.

Creating MemoryStream Instance

```
apply[A : Encoder](implicit sqlContext: SQLContext): MemoryStream[A]
```

`MemoryStream` object defines `apply` method that you can use to create instances of `MemoryStream` streaming sources.

Adding Data to Source (addData methods)

```
addData(data: A*): Offset
addData(data: TraversableOnce[A]): Offset
```

`addData` methods add the input `data` to batches internal collection.

When executed, `addData` adds a `DataFrame` (created using toDS implicit method) and increments the internal `currentOffset` offset.

You should see the following DEBUG message in the logs:

```
DEBUG MemoryStream: Adding ds: [ds]
```

Generating Next Streaming Batch — `getBatch` Method

Note	`getBatch` is a part of Streaming Source contract.

When executed, `getBatch` uses the internal batches collection to return requested offsets.

You should see the following DEBUG message in the logs:

```
DEBUG MemoryStream: MemoryBatch [[startOrdinal], [endOrdinal]]: [newBlocks]
```

StreamingExecutionRelation Logical Plan

`MemoryStream` uses StreamingExecutionRelation logical plan to build Datasets or DataFrames when requested.

```
scala> val ints = MemoryStream[Int]
ints: org.apache.spark.sql.execution.streaming.MemoryStream[Int] = MemoryStream[value#
13]

scala> ints.toDS.queryExecution.logical.isStreaming
res14: Boolean = true

scala> ints.toDS.queryExecution.logical
res15: org.apache.spark.sql.catalyst.plans.logical.LogicalPlan = MemoryStream[value#13]
```

Schema (schema method)

`MemoryStream` works with the data of the schema as described by the Encoder (of the `Dataset`).

ConsoleSinkProvider

`ConsoleSinkProvider` is a StreamSinkProvider for ConsoleSink.

`ConsoleSinkProvider` is a DataSourceRegister that registers the `ConsoleSink` streaming sink as `console` format.

ConsoleSink for Showing DataFrames to Console

`ConsoleSink` is a streaming sink that shows the DataFrame (for a batch) to the console.

`ConsoleSink` is registered as **console** format (by ConsoleSinkProvider).

Table 1. ConsoleSink's Options

Name	Default Value	Description
`numRows`	`20`	Number of rows to display
`truncate`	`true`	Truncate the data to display to 20 characters

```
scala> spark.version
res0: String = 2.3.0-SNAPSHOT

import org.apache.spark.sql.streaming.{OutputMode, Trigger}
import scala.concurrent.duration._
val query = spark.
  readStream.
  format("rate").
  load.
  writeStream.
  format("console").  // <-- use ConsoleSink
  option("truncate", false).
  option("numRows", 10).
  trigger(Trigger.ProcessingTime(10.seconds)).
  queryName("rate-console").
  start

-------------------------------------------
Batch: 0
-------------------------------------------
+---------+-----+
|timestamp|value|
+---------+-----+
+---------+-----+
```

Adding Batch (by Showing DataFrame to Console) — `addBatch` Method

```
addBatch(batchId: Long, data: DataFrame): Unit
```

Note	`addBatch` is a part of Sink Contract to "add" a batch of data to the sink.

Internally, `addBatch` records the input `batchId` in lastBatchId internal property.

`addBatch` collects the input `data` `DataFrame` and creates a brand new DataFrame that it then shows (per numRowsToShow and isTruncated properties).

```
-------------------------------------------
Batch: [batchId]
-------------------------------------------
+---------+-----+
|timestamp|value|
+---------+-----+
+---------+-----+
```

Note	You may see `Rerun batch:` instead if the input `batchId` is below lastBatchId (likely due to a batch failure).

StreamExecution — Base of Streaming Query Executions

`StreamExecution` is the base of streaming query executions that can execute structured query continuously and concurrently (as a stream execution thread).

Note	**Continuous query**, **streaming query**, **continuous Dataset**, **streaming Dataset** are synonyms, and `StreamExecution` uses analyzed logical plan internally to refer to it.

```
package org.apache.spark.sql.execution.streaming

abstract class StreamExecution(...) extends ... {
  // only required properties (vals and methods) that have no implementation
  // the others follow
  def logicalPlan: LogicalPlan
  def runActivatedStream(sparkSessionForStream: SparkSession): Unit
}
```

Table 1. (Subset of) StreamExecution Contract

Property	Description
`logicalPlan`	LogicalPlan Note: `logicalPlan` is part of ProgressReporter Contract to…FIXME. Used when `StreamExecution` is requested to runStream and toDebugString
`runActivatedStream`	Running the activated streaming query Used exclusively when `StreamExecution` is requested to runStream for the very first time (when transitioning from `INITIALIZING` to `ACTIVE` state)

Table 2. StreamExecutions

StreamExecution	Description
ContinuousExecution	
MicroBatchExecution	

`StreamExecution` is the **execution environment** of a single continuous query (aka *streaming Dataset*) that is executed every trigger and in the end adds the results to a sink.

Note	`StreamExecution` corresponds to a single streaming query with one or more streaming sources and exactly one streaming sink.

```
scala> spark.version
res0: String = 2.3.0-SNAPSHOT

import org.apache.spark.sql.streaming.Trigger
import scala.concurrent.duration._
val q = spark.
  readStream.
  format("rate").
  load.
  writeStream.
  format("console").
  trigger(Trigger.ProcessingTime(10.minutes)).
  start
scala> :type q
org.apache.spark.sql.streaming.StreamingQuery

// Pull out StreamExecution off StreamingQueryWrapper
import org.apache.spark.sql.execution.streaming.{StreamExecution, StreamingQueryWrapper
}
val se = q.asInstanceOf[StreamingQueryWrapper].streamingQuery
scala> :type se
org.apache.spark.sql.execution.streaming.StreamExecution
```

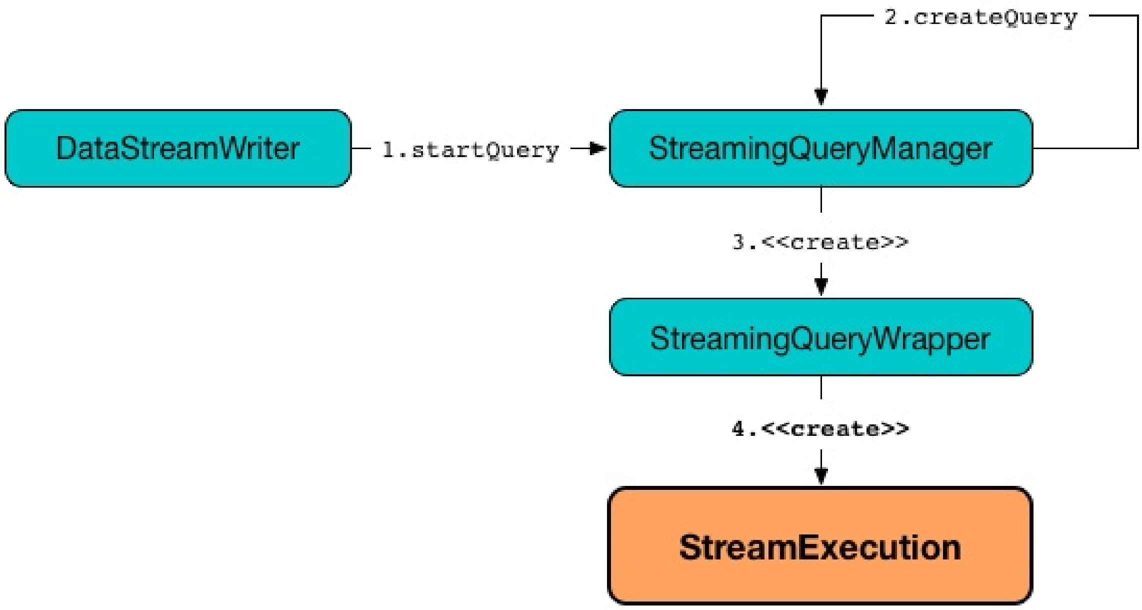

Figure 1. Creating Instance of StreamExecution

Note	DataStreamWriter describes how the results of executing batches of a streaming query are written to a streaming sink.

`StreamExecution` starts a thread of execution that runs the streaming query continuously and concurrently (and polls for new records in the streaming data sources to create a batch every trigger).

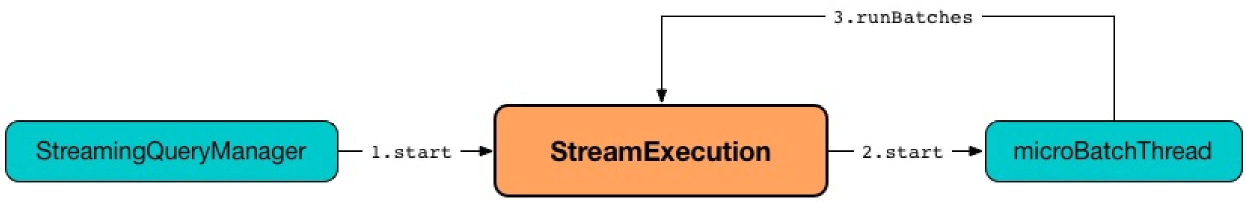

Figure 2. StreamExecution's Starting Streaming Query (on Execution Thread)

`StreamExecution` can be in three states:

- `INITIALIZED` when the instance was created.
- `ACTIVE` when batches are pulled from the sources.
- `TERMINATED` when executing streaming batches has been terminated due to an error, all batches were successfully processed or `StreamExecution` has been stopped.

`StreamExecution` is a ProgressReporter and reports status of the streaming query (i.e. when it starts, progresses and terminates) by posting `StreamingQueryListener` events.

`StreamExecution` tracks streaming data sources in uniqueSources internal registry.

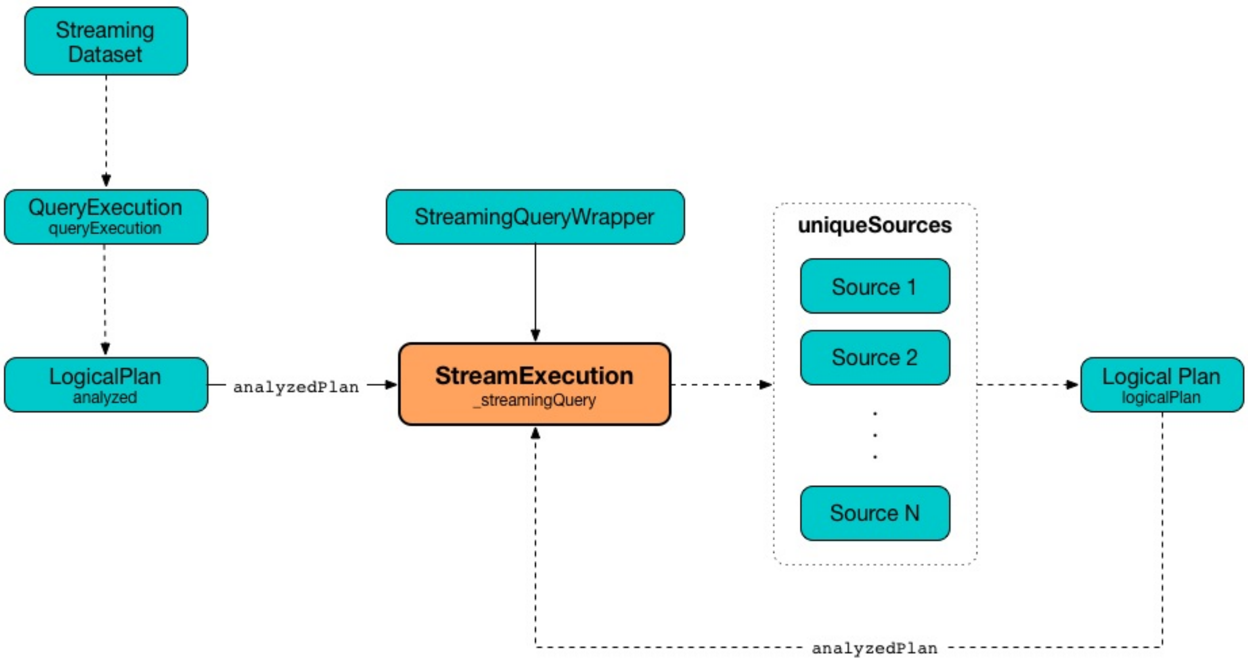

Figure 3. StreamExecution's uniqueSources Registry of Streaming Data Sources

`StreamExecution` collects `durationMs` for the execution units of streaming batches.

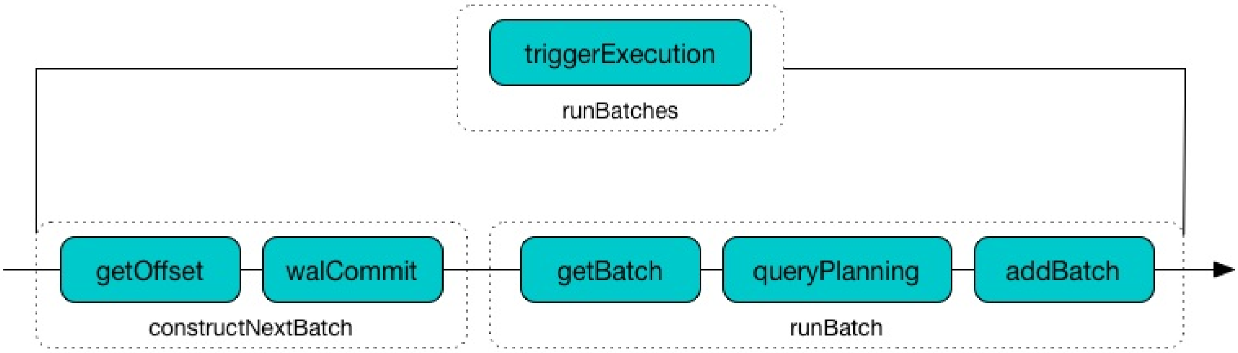

Figure 4. StreamExecution's durationMs

```
scala> :type q
org.apache.spark.sql.streaming.StreamingQuery

scala> println(q.lastProgress)
{
  "id" : "03fc78fc-fe19-408c-a1ae-812d0e28fcee",
  "runId" : "8c247071-afba-40e5-aad2-0e6f45f22488",
  "name" : null,
  "timestamp" : "2017-08-14T20:30:00.004Z",
  "batchId" : 1,
  "numInputRows" : 432,
  "inputRowsPerSecond" : 0.9993568953312452,
  "processedRowsPerSecond" : 1380.1916932907347,
  "durationMs" : {
    "addBatch" : 237,
    "getBatch" : 26,
    "getOffset" : 0,
    "queryPlanning" : 1,
    "triggerExecution" : 313,
    "walCommit" : 45
  },
  "stateOperators" : [ ],
  "sources" : [ {
    "description" : "RateSource[rowsPerSecond=1, rampUpTimeSeconds=0, numPartitions=8]"
,
    "startOffset" : 0,
    "endOffset" : 432,
    "numInputRows" : 432,
    "inputRowsPerSecond" : 0.9993568953312452,
    "processedRowsPerSecond" : 1380.1916932907347
  } ],
  "sink" : {
    "description" : "ConsoleSink[numRows=20, truncate=true]"
  }
}
```

`StreamExecution` uses OffsetSeqLog and BatchCommitLog metadata logs for **write-ahead log** (to record offsets to be processed) and that have already been processed and committed to a streaming sink, respectively.

Tip	Monitor `offsets` and `commits` metadata logs to know the progress of a streaming query.

`StreamExecution` delays polling for new data for 10 milliseconds (when no data was available to process in a batch). Use spark.sql.streaming.pollingDelay Spark property to control the delay.

Table 3. StreamExecution's Internal Registrie

<table>
<tr><th>Name</th><th></th></tr>
<tr><td><code>availableOffsets</code></td><td>
StreamProgress of the streaming sources with their availal
<table><tr><td>Note</td><td><code>availableOffsets</code> is a part of Progress</td></tr></table>
<table><tr><td>Note</td><td>StreamProgress is an enhanced <code>immutable.Map</code></td></tr></table>
<hr>
Set when (in order):

<code>StreamExecution</code> resumes and populates the start offs (and committed to the batch commit log so they are us
<code>StreamExecution</code> constructs the next streaming batch

<table><tr><td>Note</td><td>You can see availableOffsets in the DEBUG me<pre>DEBUG Resuming at batch [currentBatchId] with</pre></td></tr></table>
Used when:

<code>StreamExecution</code> starts running streaming batches for
<code>StreamExecution</code> checks whether a new data is availa
<code>StreamExecution</code> constructs the next streaming batch
<code>StreamExecution</code> runs a streaming batch (and fetches offsets registry)
<code>StreamExecution</code> finishes running streaming batches a sink (and being added to committed offsets registry)
<code>StreamExecution</code> prints out debug information when a

<table><tr><td>Note</td><td><code>availableOffsets</code> works in tandem with</td></tr></table>
</td></tr>
</table>

`awaitProgressLock`	Java's fair reentrant mutual exclusion java.util.concurrent.l contention).
`awaitProgressLockCondition`	
`callSite`	
`commitLog`	CommitLog with `commits` metadata checkpoint directory fo the batch id). Note: **Metadata log** or **metadata checkpoint** are s Used exclusively by the extensions for the following: • `MicroBatchExecution` is requested to runActivatedStre populateStartOffsets or constructNextBatch) • `ContinuousExecution` is requested to getStartOffsets,
`committedOffsets`	StreamProgress of the streaming sources and the committ Note: `committedOffsets` is a part of Progress
`currentBatchId`	Current batch number • `-1` when `StreamExecution` is created • `0` when `StreamExecution` populates start offsets (and • Incremented when `StreamExecution` runs streaming b committing the batch).
`id`	Unique identifier of the streaming query Set as the `id` of streamMetadata when `StreamExecution` Note: `id` can get fetched from checkpoint metadata i failure or a planned stop).
`initializationLatch`	
`lastExecution`	Last IncrementalExecution
	Lazily-generated logical plan (i.e. `LogicalPlan`) of the stre Note: `logicalPlan` is a part of ProgressRep Initialized right after `StreamExecution` starts running strean Used mainly when `StreamExecution` replaces StreamingEx arrived since the last batch.

`logicalPlan`	While initializing, `logicalPlan` transforms the analyzed log StreamingExecutionRelation. `logicalPlan` creates a `Stre` `/sources/[nextSourceId]` under the checkpoint directory. Note: `nextSourceId` is the unique identifier of every Note: `logicalPlan` uses `DataSource.createSource` fac or `FileFormat` as the implementations of the str While initializing, `logicalPlan` also initializes sources and
`newData`	Registry of the streaming sources (in logical query plan) th `DataFrame` . Note: `newData` is a part of ProgressReport Set exclusively when `StreamExecution` requests unproces Used exclusively when `StreamExecution` replaces Streami running a single streaming batch).
`noNewData`	Flag whether there are any new offsets available for proce Turned on (i.e. enabled) when constructing the next strean
`offsetLog`	OffsetSeqLog with `offsets` metadata checkpoint directory Note: **Metadata log** or **metadata checkpoint** are s Used when `StreamExecution` populates the start offsets ar write-ahead log and retrieve the previous batch's offsets rig Note: `StreamExecution` discards offsets from the offse spark.sql.streaming.minBatchesToRetain Spark
`offsetSeqMetadata`	OffsetSeqMetadata Note: `offsetSeqMetadata` is a part of Progress • Initialized with `0` for `batchWatermarkMs` and `batchTim` • Updated with `0` for `batchWatermarkMs` and `batchTime` `StreamExecution` runs streaming batches. • Used in…FIXME • Copied with `batchTimestampMs` updated with the curre batch.
	Time delay before polling new data again when no data wa

`pollingDelayMs`	Set to spark.sql.streaming.pollingDelay Spark property. Used when `StreamExecution` has started running streamin

`prettyIdString`	Pretty-identified string for identification in logs (with name i
	`queryName [id = xyz, runId = abc]`
	`[id = xyz, runId = abc]`

`resolvedCheckpointRoot`	Qualified path of the checkpoint directory (as defined using Note: checkpointRoot is defined using `checkpointLoca` `queryName` option. `checkpointLocation` and `queryName` options are Used when creating the path to the checkpoint directory ar Used for logicalPlan (while transforming analyzedPlan and `StreamingExecutionRelation` physical operators with the st checkpointing metadata). Note: You can see `resolvedCheckpointRoot` in the IN `INFO StreamExecution: Starting [id] with [r` Internally, `resolvedCheckpointRoot` creates a Hadoop `org.` Note: `resolvedCheckpointRoot` uses `SparkSession`
`runId`	Current run id
`sources`	All streaming Sources in logical query plan (that are the so
`startLatch`	Java's java.util.concurrent.CountDownLatch with count `1` Used when `StreamExecution` is requested to start to pause
`state`	Java's java.util.concurrent.atomic.AtomicReference for the • `INITIALIZING` (default) • `ACTIVE` (after the first execution of runBatches) • `TERMINATED`
`streamDeathCause`	`StreamingQueryException`

`streamMetadata`	`StreamMetadata` from the `metadata` file from checkpoint di
`uniqueSources`	Unique streaming data sources in a streaming Dataset (aft query plan). Note: StreamingExecutionRelation is a leaf logical ope corresponds to a single StreamingRelation in ar Used when `StreamExecution` : • Constructs the next streaming batch (and gets new of • Stops all streaming data sources

> **Tip**
>
> Enable `INFO` or `DEBUG` logging levels for `org.apache.spark.sql.execution.streaming.StreamExecution` to see what happens inside.
>
> Add the following line to `conf/log4j.properties` :
>
> ```
> log4j.logger.org.apache.spark.sql.execution.streaming.StreamExecution=DEBUG
> ```
>
> Refer to Logging.

`stop` Method

Caution	FIXME

`stopSources` Internal Method

```
stopSources(): Unit
```

Caution	FIXME

Running Streaming Batches — `runBatches` Internal Method

```
runBatches(): Unit
```

`runBatches` runs streaming batches of data (that are datasets from every streaming source).

```
import org.apache.spark.sql.streaming.Trigger
import scala.concurrent.duration._

val out = spark.
  readStream.
  text("server-logs").
  writeStream.
  format("console").
  queryName("debug").
  trigger(Trigger.ProcessingTime(10.seconds))
scala> val debugStream = out.start
INFO StreamExecution: Starting debug [id = 8b57b0bd-fc4a-42eb-81a3-777d7ba5e370, runId
 = 920b227e-6d02-4a03-a271-c62120258cea]. Use file:///private/var/folders/0w/kb0d3rqn4
zb9fcc91pxhgn8w0000gn/T/temporary-274f9ae1-1238-4088-b4a1-5128fc520c1f to store the qu
ery checkpoint.
debugStream: org.apache.spark.sql.streaming.StreamingQuery = org.apache.spark.sql.exec
ution.streaming.StreamingQueryWrapper@58a5b69c

// Enable the log level to see the INFO and DEBUG messages
// log4j.logger.org.apache.spark.sql.execution.streaming.StreamExecution=DEBUG

17/06/18 21:21:07 INFO StreamExecution: Starting new streaming query.
17/06/18 21:21:07 DEBUG StreamExecution: getOffset took 5 ms
17/06/18 21:21:07 DEBUG StreamExecution: Stream running from {} to {}
17/06/18 21:21:07 DEBUG StreamExecution: triggerExecution took 9 ms
17/06/18 21:21:07 DEBUG StreamExecution: Execution stats: ExecutionStats(Map(),List(),
Map())
17/06/18 21:21:07 INFO StreamExecution: Streaming query made progress: {
  "id" : "8b57b0bd-fc4a-42eb-81a3-777d7ba5e370",
  "runId" : "920b227e-6d02-4a03-a271-c62120258cea",
  "name" : "debug",
  "timestamp" : "2017-06-18T19:21:07.693Z",
  "numInputRows" : 0,
  "processedRowsPerSecond" : 0.0,
  "durationMs" : {
    "getOffset" : 5,
    "triggerExecution" : 9
  },
  "stateOperators" : [ ],
  "sources" : [ {
    "description" : "FileStreamSource[file:/Users/jacek/dev/oss/spark/server-logs]",
    "startOffset" : null,
    "endOffset" : null,
    "numInputRows" : 0,
    "processedRowsPerSecond" : 0.0
  } ],
  "sink" : {
    "description" : "org.apache.spark.sql.execution.streaming.ConsoleSink@2460208a"
  }
}
17/06/18 21:21:10 DEBUG StreamExecution: Starting Trigger Calculation
17/06/18 21:21:10 DEBUG StreamExecution: getOffset took 3 ms
```

```
17/06/18 21:21:10 DEBUG StreamExecution: triggerExecution took 3 ms
17/06/18 21:21:10 DEBUG StreamExecution: Execution stats: ExecutionStats(Map(),List(),
Map())
```

Internally, `runBatches` assigns the group id (to all the Spark jobs started by this thread) as runId (with the group description to display in web UI as getBatchDescriptionString and `interruptOnCancel` flag enabled).

Note	`runBatches` uses SparkSession to access `SparkContext` and assign the group id. You can find the details on `SparkContext.setJobGroup` method in the Mastering Apache Spark 2 gitbook.

`runBatches` sets a local property `sql.streaming.queryId` as id.

`runBatches` registers a metric source when spark.sql.streaming.metricsEnabled property is enabled (which is disabled by default).

Caution	FIXME Metrics

`runBatches` notifies `StreamingQueryListeners` that a streaming query has been started (by posting a QueryStartedEvent with id, runId and name).

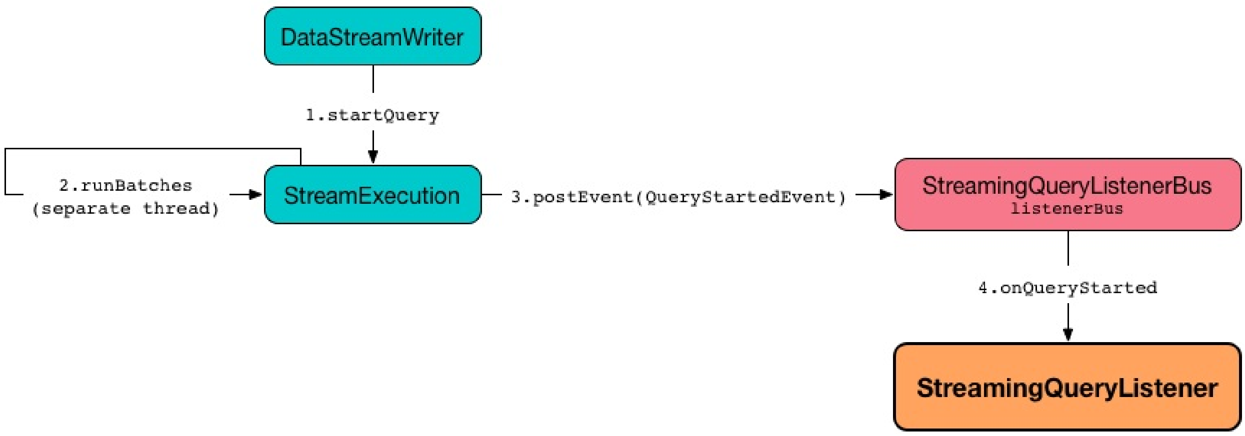

Figure 5. StreamingQueryListener Notified about Query's Start (onQueryStarted)

`runBatches` unblocks the main starting thread (by decrementing the count of startLatch that goes to `0` and lets the starting thread continue).

Caution	FIXME A picture with two parallel lanes for the starting thread and daemon one for the query.

`runBatches` updates the status message to **Initializing sources** followed by initialization of the logical plan (of the streaming Dataset).

`runBatches` disables adaptive query execution (using `spark.sql.adaptive.enabled` property which is disabled by default) as it could change the number of shuffle partitions.

`runBatches` initializes offsetSeqMetadata internal variable.

`runBatches` sets state to `ACTIVE` (only when the current state is `INITIALIZING` that prevents from repeating the initialization)

Note	`runBatches` does the work only when first started (i.e. when state is `INITIALIZING`).

`runBatches` decrements the count of initializationLatch.

Caution	FIXME `initializationLatch` so what?

`runBatches` requests TriggerExecutor to start executing batches (aka *triggers*) by executing a batch runner.

Once TriggerExecutor has finished executing batches, `runBatches` updates the status message to **Stopped**.

Note	TriggerExecutor finishes executing batches when batch runner returns whether the streaming query is stopped or not (which is when the internal state is not `TERMINATED`).

Caution	FIXME Describe `catch` block for exception handling

Caution	FIXME Describe `finally` block for query termination

Note	`runBatches` is used exclusively when `StreamExecution` starts the stream execution thread for a streaming query (i.e. the thread that runs the micro-batches of this stream).

TriggerExecutor's Batch Runner

Batch Runner (aka `batchRunner`) is an executable block executed by TriggerExecutor in runBatches.

`batchRunner` starts trigger calculation.

As long as the query is not stopped (i.e. state is not `TERMINATED`), `batchRunner` executes the streaming batch for the trigger.

In **triggerExecution** time-tracking section, `runBatches` branches off per currentBatchId.

Table 4. Current Batch Execution per currentBatchId

currentBatchId < 0	currentBatchI >= 0
1. populateStartOffsets 2. Setting Job Description as getBatchDescriptionString `DEBUG Stream running from [committedOffsets] to [availableOffsets]`	1. Constructing the next streaming batcl

If there is data available in the sources, `batchRunner` marks currentStatus with `isDataAvailable` enabled.

> **Note**
>
> You can check out the status of a streaming query using status method.
>
> ```
> scala> spark.streams.active(0).status
> res1: org.apache.spark.sql.streaming.StreamingQueryStatus =
> {
>   "message" : "Waiting for next trigger",
>   "isDataAvailable" : false,
>   "isTriggerActive" : false
> }
> ```

`batchRunner` then updates the status message to **Processing new data** and runs the current streaming batch.

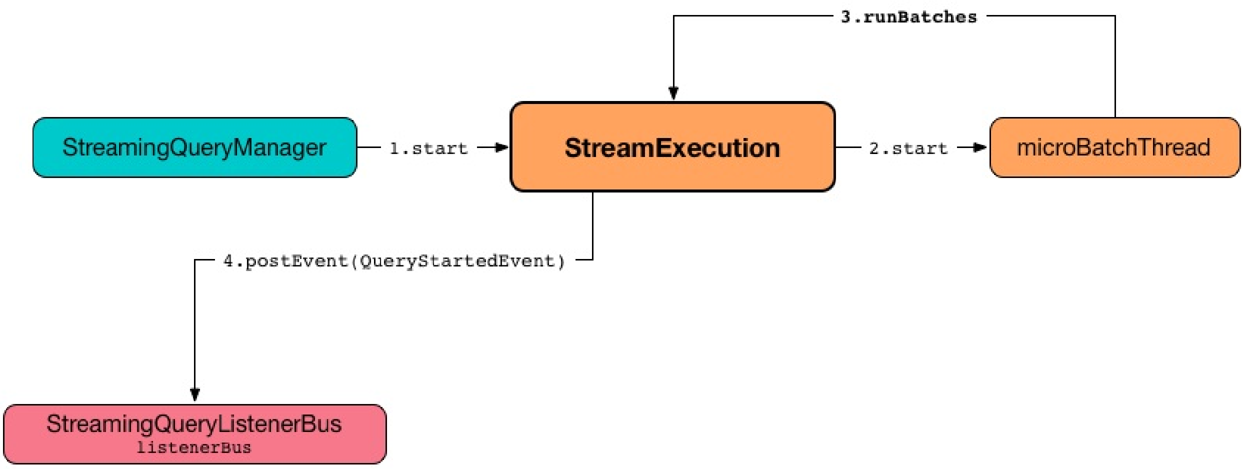

Figure 6. StreamExecution's Running Batches (on Execution Thread)

After **triggerExecution** section has finished, `batchRunner` finishes the streaming batch for the trigger (and collects query execution statistics).

When there was data available in the sources, `batchRunner` updates committed offsets (by adding the current batch id to BatchCommitLog and adding availableOffsets to committedOffsets).

You should see the following DEBUG message in the logs:

```
DEBUG batch $currentBatchId committed
```

`batchRunner` increments the current batch id and sets the job description for all the following Spark jobs to include the new batch id.

When no data was available in the sources to process, `batchRunner` does the following:

1. Marks currentStatus with `isDataAvailable` disabled
2. Updates the status message to **Waiting for data to arrive**
3. Sleeps the current thread for pollingDelayMs milliseconds.

`batchRunner` updates the status message to **Waiting for next trigger** and returns whether the query is currently active or not (so TriggerExecutor can decide whether to finish executing the batches or not)

`getBatchDescriptionString` Internal Method

```
getBatchDescriptionString: String
```

Caution	FIXME

`toDebugString` Internal Method

```
toDebugString(includeLogicalPlan: Boolean): String
```

`toDebugString` …FIXME

Note	`toDebugString` is used exclusively when `StreamExecution` is requested to run the underlying streaming query plan (and a streaming query terminated with an exception).

Starting Streaming Query (on Stream Execution Thread) — `start` Method

```
start(): Unit
```

When called, `start` prints out the following INFO message to the logs:

```
Starting [id]. Use [resolvedCheckpointRoot] to store the query checkpoint.
```

`start` then starts the queryExecutionThread as a daemon thread.

Note	`start` uses Java's java.lang.Thread.start to run the streaming query on a separate execution thread.

Note	When started, a streaming query runs in its own execution thread on JVM.

In the end, `start` pauses the main thread (using the startLatch until `StreamExecution` was requested to run the streaming query).

Note	`start` is used exclusively when `StreamingQueryManager` is requested to start a streaming query.

Creating StreamExecution Instance

`StreamExecution` takes the following when created:

- `SparkSession`
- Query name
- Path to the checkpoint directory (aka *metadata directory*)
- Analyzed logical query plan (i.e. `LogicalPlan`)
- Streaming sink
- Trigger
- `Clock`
- Output mode (that is only used when creating `IncrementalExecution` for a streaming batch in query planning)
- `deleteCheckpointOnStop` flag to control whether to delete the checkpoint directory on stop

`StreamExecution` initializes the internal registries and counters.

Note	`StreamExecution` is a Scala abstract class and cannot be created directly. It is created indirectly when the concrete StreamExecutions are.

Creating Path to Checkpoint Directory — `checkpointFile` Internal Method

```
checkpointFile(name: String): String
```

`checkpointFile` gives the path of a directory with `name` in checkpoint directory.

Note	`checkpointFile` uses Hadoop's `org.apache.hadoop.fs.Path`.

Note	`checkpointFile` is used for streamMetadata, OffsetSeqLog, BatchCommitLog, and lastExecution (for runBatch).

Posting StreamingQueryListener Event — `postEvent` Method

```
postEvent(event: StreamingQueryListener.Event): Unit
```

Note	`postEvent` is a part of ProgressReporter Contract.

`postEvent` simply requests the `StreamingQueryManager` to post the input event (to the StreamingQueryListenerBus in the current `SparkSession`).

Note	`postEvent` uses `SparkSession` to access the current `StreamingQueryManager`.

Note	`postEvent` is used when: • `ProgressReporter` reports update progress (while finishing a trigger) • `StreamExecution` runs streaming batches (and announces starting a streaming query by posting a QueryStartedEvent and query termination by posting a QueryTerminatedEvent)

Waiting Until No Data Available in Sources or Query Has Been Terminated — `processAllAvailable` Method

```
processAllAvailable(): Unit
```

Note	`processAllAvailable` is a part of StreamingQuery Contract.

`processAllAvailable` reports streamDeathCause exception if defined (and returns).

Note	streamDeathCause is defined exclusively when `StreamExecution` runs streaming batches (and terminated with an exception).

`processAllAvailable` returns when isActive flag is turned off (which is when `StreamExecution` is in `TERMINATED` state).

`processAllAvailable` acquires a lock on awaitProgressLock and turns noNewData flag off.

`processAllAvailable` keeps waiting 10 seconds for awaitProgressLockCondition until noNewData flag is turned on or `StreamExecution` is no longer active.

Note	noNewData flag is turned on exclusively when `StreamExecution` constructs the next streaming batch (and finds that no data is available).

In the end, `processAllAvailable` releases awaitProgressLock lock.

Stream Execution Thread — `queryExecutionThread` Property

```
queryExecutionThread: QueryExecutionThread
```

`queryExecutionThread` is a Java thread of execution (java.util.Thread) that runs a structured query.

`queryExecutionThread` uses the name `stream execution thread for [id]` (that uses prettyIdString for the id).

`queryExecutionThread` is a Spark Core `UninterruptibleThread` that provides `runUninterruptibly` method that allows running a block of code without being interrupted by `Thread.interrupt()`.

`queryExecutionThread` is started as a daemon thread when `StreamExecution` is requested to start.

When started, `queryExecutionThread` sets the thread-local properties as the call site and runs the streaming query.

When `StreamExecution` finishes running the streaming query, it uses `queryExecutionThread` to execute the `runUninterruptibly` code block uninterruptibly.

`queryExecutionThread` is also used when `StreamExecution` is requested to stop

Tip

Use Java's jconsole or jstack to monitor the streaming threads.

```
$ jstack <driver-pid> \| grep -e "stream execution thread"
"stream execution thread for kafka-topic1 [id =...
```

MicroBatchExecution

`MicroBatchExecution` is a StreamExecution that…FIXME

`MicroBatchExecution` is created exclusively when `DataStreamWriter` is requested to start for a sink that is not a StreamWriteSupport or a trigger that is not a ContinuousTrigger (when `StreamingQueryManager` is requested to create a serializable StreamingQuery).

Table 1. MicroBatchExecution's Internal Properties (e.g. Registries, Counters and Flags)

Name	Description
`sources`	Collection of `BaseStreamingSource` instances Used when…FIXME
`triggerExecutor`	TriggerExecutor for the Trigger: • `ProcessingTimeExecutor` for `ProcessingTime` • `OneTimeExecutor` for `OneTimeTrigger` (aka Once trigger) Used when `StreamExecution` starts running streaming batches. Note: `StreamExecution` throws a `IllegalStateException` when the Trigger is not one of the two built-in implementations: `OneTimeExecutor` or `ProcessingTimeExecutor`.

Populating Start Offsets — `populateStartOffsets` Internal Method

```
populateStartOffsets(sparkSessionToRunBatches: SparkSession): Unit
```

`populateStartOffsets` requests OffsetSeqLog for the latest committed batch id with its metadata if available.

Note	The batch id could not be available in metadata log if a streaming query started with a new metadata log or no batch was committed before.

With the latest committed batch id with the metadata (from OffsetSeqLog) `populateStartOffsets` sets current batch id to the latest committed batch id, and availableOffsets to its offsets (considering them unprocessed yet).

Note	`populateStartOffsets` may re-execute the latest committed batch.

If the latest batch id is greater than `0`, `populateStartOffsets` requests OffsetSeqLog for the second latest batch with its metadata (or reports a `IllegalStateException` if not found). `populateStartOffsets` sets committed offsets to the second latest committed offsets.

`populateStartOffsets` updates the offset metadata.

Caution	FIXME Why is the update needed?

`populateStartOffsets` requests BatchCommitLog for the latest processed batch id with its metadata if available.

(only when the latest batch in OffsetSeqLog is also the latest batch in BatchCommitLog) With the latest processed batch id with the metadata (from BatchCommitLog), `populateStartOffsets` sets current batch id as the next after the latest processed batch. `populateStartOffsets` sets committed offsets to availableOffsets.

Caution	FIXME Describe what happens with `availableOffsets`.

`populateStartOffsets` constructs the next streaming batch.

Caution	FIXME Describe the WARN message when `latestCommittedBatchId < latestBatchId - 1`.

```
WARN Batch completion log latest batch id is
[latestCommittedBatchId], which is not trailing batchid
[latestBatchId] by one
```

You should see the following DEBUG message in the logs:

```
DEBUG Resuming at batch [currentBatchId] with committed offsets [committedOffsets] and
 available offsets [availableOffsets]
```

Caution	FIXME Include an example of Resuming at batch

When the latest committed batch id with the metadata could not be found in BatchCommitLog, `populateStartOffsets` prints out the following INFO message to the logs:

```
INFO no commit log present
```

Caution	FIXME Include an example of the case when no commit log present.

When the latest committed batch id with the metadata could not be found in OffsetSeqLog, it is assumed that the streaming query is started for the first time. You should see the following INFO message in the logs:

```
INFO StreamExecution: Starting new streaming query.
```

`populateStartOffsets` sets current batch id to `0` and constructs the next streaming batch.

Note	`populateStartOffsets` is used exclusively when `MicroBatchExecution` is requested to runActivatedStream.

Running Activated Streaming Query — `runActivatedStream` Method

```
runActivatedStream(sparkSessionForStream: SparkSession): Unit
```

Note	`runActivatedStream` is part of StreamExecution Contract to run the activated streaming query.

`runActivatedStream` …FIXME

Creating MicroBatchExecution Instance

`MicroBatchExecution` takes the following when created:

- `SparkSession`
- Query name
- Path to the checkpoint directory (aka *metadata directory*)
- Analyzed logical query plan (i.e. `LogicalPlan`)
- Streaming sink
- Trigger
- `Clock`

- Output mode (that is only used when creating `IncrementalExecution` for a streaming batch in query planning)
- Map[String, String]
- `deleteCheckpointOnStop` flag to control whether to delete the checkpoint directory on stop

`MicroBatchExecution` initializes the internal registries and counters.

`logicalPlan` Lazy Value

```
logicalPlan: LogicalPlan
```

Note	`logicalPlan` is part of StreamExecution Contract to…FIXME.

`logicalPlan` …FIXME

Constructing Next Streaming Batch — `constructNextBatch` Internal Method

```
constructNextBatch(): Unit
```

`constructNextBatch` is made up of the following three parts:

1. Firstly, checking if there is new data available by requesting new offsets from every streaming source
2. There is some data to process (and so the next batch is constructed)
3. No data is available

Note	`constructNextBatch` is used when `StreamExecution`: • Runs streaming batches • Populates the start offsets

Checking Whether New Data Is Available (by Requesting New Offsets from Sources)

`constructNextBatch` starts by checking whether or not a new data is available in any of the streaming sources (in the logical query plan).

`constructNextBatch` acquires awaitProgressLock and gets the latest offset from every streaming data source.

Note	`constructNextBatch` checks out the latest offset in every streaming data source sequentially, i.e. one data source at a time.

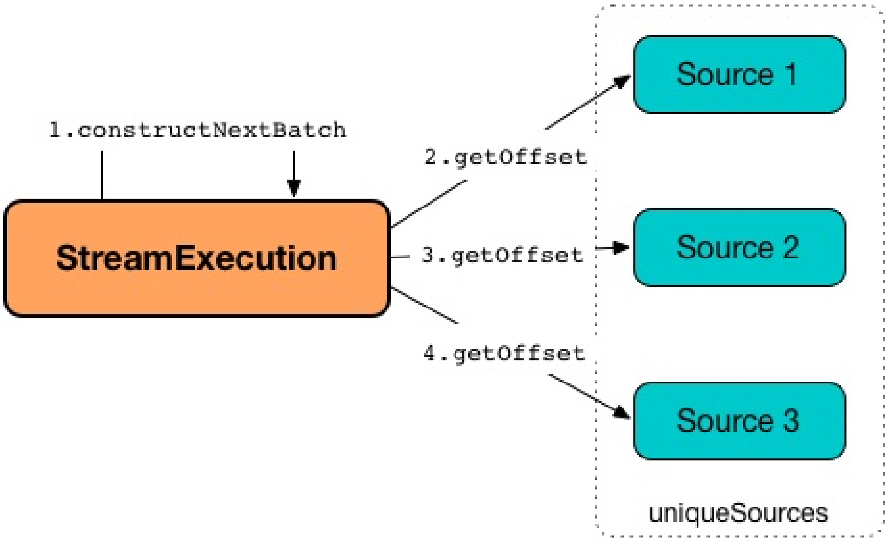

Figure 1. StreamExecution's Getting Offsets From Streaming Sources

Note	`constructNextBatch` uses the `Source` contract to get the latest offset (using `Source.getOffset` method).

`constructNextBatch` updates the status message to **Getting offsets from [source]** for every streaming data source.

In **getOffset** time-tracking section, `constructNextBatch` gets the offsets.

`constructNextBatch` prints out the following DEBUG message to the logs:

```
DEBUG StreamExecution: getOffset took [time] ms
```

`constructNextBatch` adds the streaming sources that have the available offsets to availableOffsets.

If there is no data available (i.e. no offsets unprocessed in any of the streaming data sources), `constructNextBatch` turns noNewData flag on.

In the end (of this checking-data block), `constructNextBatch` releases awaitProgressLock

New Data Available

When new data is available, `constructNextBatch` updates the event time watermark (tracked using offsetSeqMetadata) if it finds one in the last IncrementalExecution.

If lastExecution is available (which may not when `constructNextBatch` is executed the very first time), `constructNextBatch` takes the executed physical plan (i.e. `SparkPlan`) and collects all `EventTimeWatermarkExec` physical operators with the count of eventTimeStats greater than `0` .

Note	The executed physical plan is available as `executedPlan` property of IncrementalExecution (which is a custom `QueryExecution`).

You should see the following DEBUG message in the logs:

```
DEBUG StreamExecution: Observed event time stats: [eventTimeStats]
```

`constructNextBatch` calculates the difference between the maximum value of `eventTimeStats` and delayMs for every `EventTimeWatermarkExec` physical operator.

Note	The maximum value of `eventTimeStats` is the youngest time, i.e. the time the closest to the current time.

`constructNextBatch` then takes the first difference (if available at all) and uses it as a possible new event time watermark.

If the event time watermark candidate is greater than the current watermark (i.e. later time-wise), `constructNextBatch` prints out the following INFO message to the logs:

```
INFO StreamExecution: Updating eventTime watermark to: [newWatermarkMs] ms
```

`constructNextBatch` creates a new OffsetSeqMetadata with the new event time watermark and the current time.

Otherwise, if the eventTime watermark candidate is not greater than the current watermark, `constructNextBatch` simply prints out the following DEBUG message to the logs:

```
DEBUG StreamExecution: Event time didn't move: [newWatermarkMs] <= [batchWatermarkMs]
```

`constructNextBatch` creates a new OffsetSeqMetadata with just the current time.

Note	Although `constructNextBatch` collects all the `EventTimeWatermarkExec` physical operators in the executed physical plan of lastExecution, only the first matters if available.

Note	A physical plan can have as many `EventTimeWatermarkExec` physical operators as withWatermark operator was used to create a streaming query.

Note	Streaming watermark can be changed between a streaming query's restarts (and be different between what is checkpointed and the current version of the query). FIXME True? Example?

`constructNextBatch` then adds the offsets to metadata log.

`constructNextBatch` updates the status message to **Writing offsets to log**.

In **walCommit** time-tracking section, `constructNextBatch` adds the offsets in the batch to OffsetSeqLog.

Note	While writing the offsets to the metadata log, `constructNextBatch` uses the following internal registries: • currentBatchId for the current batch id • StreamProgress for the available offsets • sources for the streaming sources • OffsetSeqMetadata

`constructNextBatch` reports a `AssertionError` when writing to the metadata log has failed.

```
Concurrent update to the log. Multiple streaming jobs detected for [currentBatchId]
```

Tip	Use StreamingQuery.lastProgress to access `walCommit` duration.
	`scala> :type sq`
	`org.apache.spark.sql.streaming.StreamingQuery`
	`sq.lastProgress.durationMs.get("walCommit")`

Tip	Enable INFO logging level for `org.apache.spark.sql.execution.streaming.StreamExecution` logger to be notified about `walCommit` duration.
	 <pre>17/08/11 09:04:17 INFO StreamExecution: Streaming query made progress: {
	  "id" : "ec8f8228-90f6-4e1f-8ad2-80222affed63",
	  "runId" : "f605c134-cfb0-4378-88c1-159d8a7c232e",
	  "name" : "rates-to-console",
	  "timestamp" : "2017-08-11T07:04:17.373Z",
	  "batchId" : 0,
	  "numInputRows" : 0,
	  "processedRowsPerSecond" : 0.0,
	  "durationMs" : {
	    "addBatch" : 38,
	    "getBatch" : 1,
	    "getOffset" : 0,
	    "queryPlanning" : 1,
	    "triggerExecution" : 62,
	    "walCommit" : 19 // <-- walCommit
	  },</pre>

`constructNextBatch` commits the offsets for the batch (only when current batch id is not `0`, i.e. when the query has just been started and `constructNextBatch` is called the first time).

`constructNextBatch` takes the previously-committed batch (from OffsetSeqLog), extracts the stored offsets per source.

Note	`constructNextBatch` uses OffsetSeq.toStreamProgress and sources registry to extract the offsets per source.

`constructNextBatch` requests every streaming source to commit the offsets

Note	`constructNextBatch` uses the `Source` contract to commit the offsets (using `Source.commit` method).

`constructNextBatch` reports a `IllegalStateException` when current batch id is `0`.

```
batch [currentBatchId] doesn't exist
```

In the end, `constructNextBatch` purges OffsetSeqLog and BatchCommitLog when current batch id is above spark.sql.streaming.minBatchesToRetain Spark property.

No New Data Available

If there is no new data available, `constructNextBatch` acquires a lock on awaitProgressLock, wakes up all waiting threads that are waiting for the stream to progress (using awaitProgressLockCondition), followed by releasing the lock on awaitProgressLock.

Checking Whether Data Is Available in Streaming Sources — `dataAvailable` Internal Method

```
dataAvailable: Boolean
```

`dataAvailable` finds the streaming sources in availableOffsets for which the offsets committed (as recorded in committedOffsets) are different or do not exist at all.

If there are any differences in the number of sources or their committed offsets, `dataAvailable` is enabled (i.e. `true`).

Note	`dataAvailable` is used when `StreamExecution` runs streaming batches and constructs the next streaming batch.

Running Single Streaming Batch — `runBatch` Internal Method

```
runBatch(sparkSessionToRunBatch: SparkSession): Unit
```

`runBatch` performs the following steps (aka *phases*):

1. getBatch Phase — Requesting New (and Hence Unprocessed) Data From Streaming Sources
2. withNewSources Phase — Replacing StreamingExecutionRelations (in Logical Plan) With Relations With New Data or Empty LocalRelation
3. triggerLogicalPlan Phase — Transforming Catalyst Expressions
4. queryPlanning Phase — Creating IncrementalExecution for Current Streaming Batch
5. nextBatch Phase — Creating Dataset (with IncrementalExecution for New Data)
6. addBatch Phase — Adding Current Streaming Batch to Sink
7. awaitBatchLock Phase — Waking Up Threads Waiting For Stream to Progress

Note	`runBatch` is used exclusively when `StreamExecution` runs streaming batches.

getBatch Phase — Requesting New (and Hence Unprocessed) Data From Streaming Sources

Internally, `runBatch` first requests the streaming sources for unprocessed data (and stores them as `DataFrames` in newData internal registry).

In **getBatch** time-tracking section, `runBatch` goes over the available offsets per source and processes the offsets that have not been committed yet.

`runBatch` then requests every source for the data (as `DataFrame` with the new records).

Note	`runBatch` requests the streaming sources for new DataFrames sequentially, source by source.

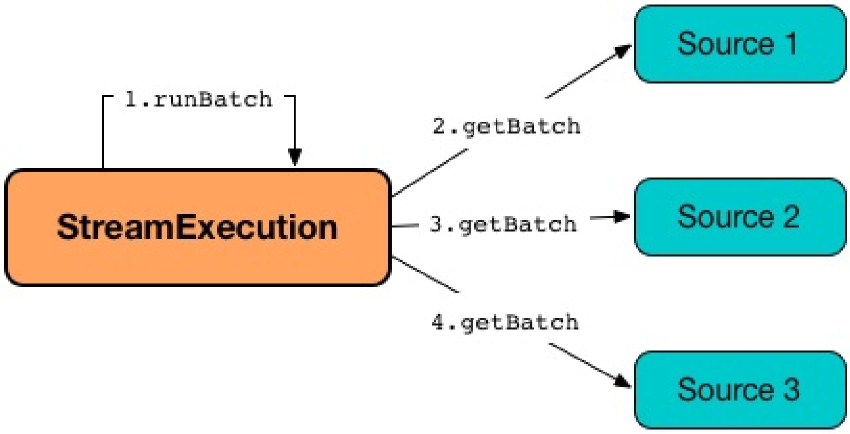

Figure 2. StreamExecution's Running Single Streaming Batch (getBatch Phase)

You should see the following DEBUG message in the logs:

```
DEBUG StreamExecution: Retrieving data from [source]: [current] -> [available]
```

You should then see the following DEBUG message in the logs:

```
DEBUG StreamExecution: getBatch took [timeTaken] ms
```

withNewSources Phase — Replacing StreamingExecutionRelations (in Logical Plan) With Relations With New Data or Empty LocalRelation

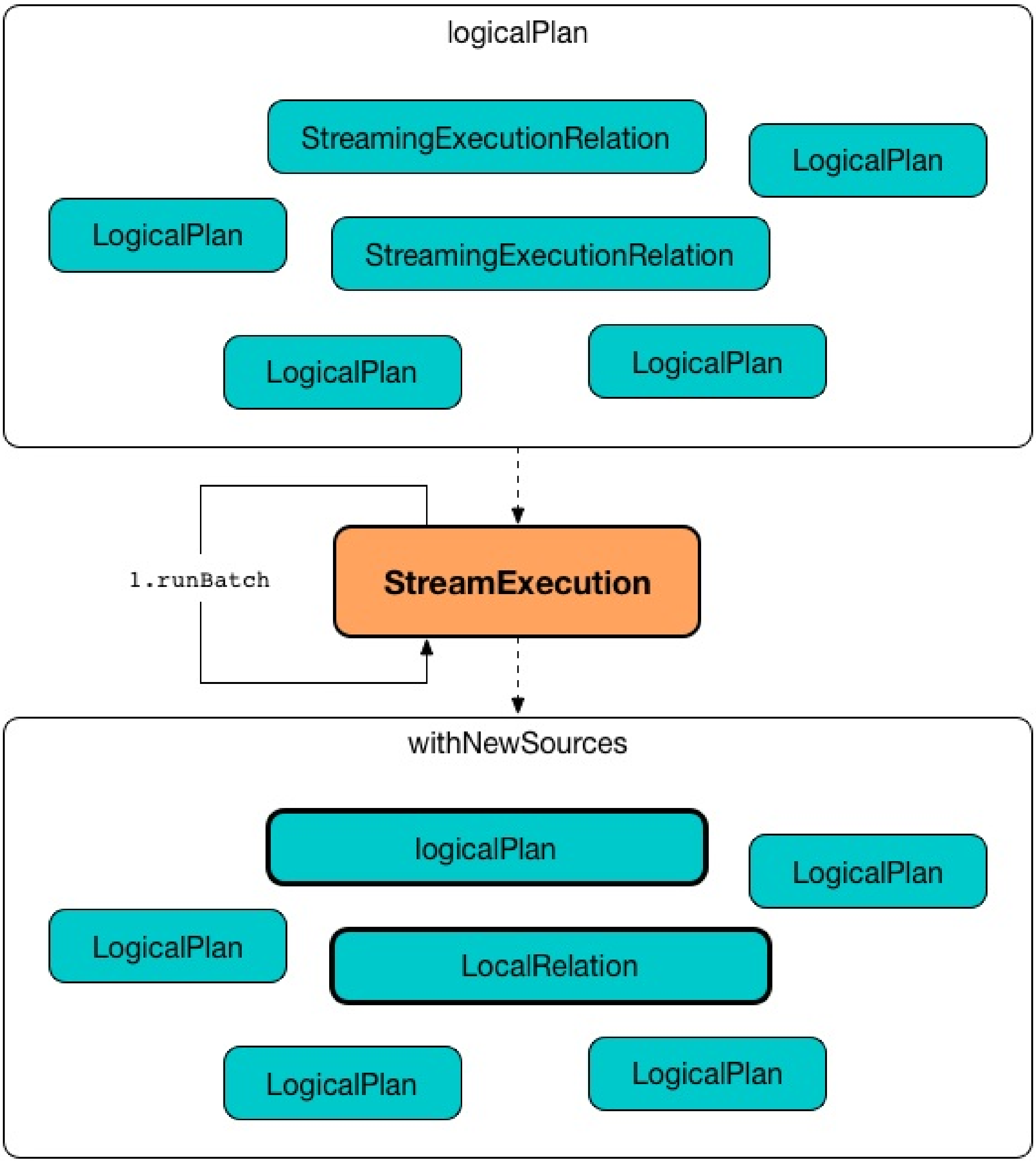

Figure 3. StreamExecution's Running Single Streaming Batch (withNewSources Phase)

In **withNewSources** phase, `runBatch` transforms logical query plan and replaces every StreamingExecutionRelation logical operator with the logical plan of the `DataFrame` with the input data in a batch for the corresponding streaming source.

Note	StreamingExecutionRelation logical operator is used to represent a streaming source in the logical query plan of a streaming `Dataset`.

`runBatch` finds the corresponding `DataFrame` (with the input data) per streaming source in newData internal registry. If found, `runBatch` takes the logical plan of the `DataFrame`. If not, `runBatch` creates a `LocalRelation` logical relation (for the output schema).

Note	newData internal registry contains entries for streaming sources that have new data available in the current batch.

While replacing `StreamingExecutionRelation` operators, `runBatch` records the output schema of the streaming source (from `StreamingExecutionRelation`) and the `DataFrame` with the new data (in `replacements` temporary internal buffer).

`runBatch` makes sure that the output schema of the streaming source with a new data in the batch has not changed. If the output schema has changed, `runBatch` reports…FIXME

triggerLogicalPlan Phase — Transforming Catalyst Expressions

`runBatch` transforms Catalyst expressions in `withNewSources` new logical plan (using `replacements` temporary internal buffer).

- Catalyst `Attribute` is replaced with one if recorded in `replacements` internal buffer (that corresponds to the attribute in the `DataFrame` with the new input data in the batch)
- `CurrentTimestamp` and `CurrentDate` Catalyst expressions are replaced with `CurrentBatchTimestamp` expression (with `batchTimestampMs` from OffsetSeqMetadata).

Note	`CurrentTimestamp` Catalyst expression corresponds to `current_timestamp` function. Find more about `current_timestamp` function in Mastering Apache Spark 2 gitbook.

Note	`CurrentDate` Catalyst expression corresponds to `current_date` function. Find more about `current_date` function in Mastering Apache Spark 2 gitbook.

queryPlanning Phase — Creating IncrementalExecution for Current Streaming Batch

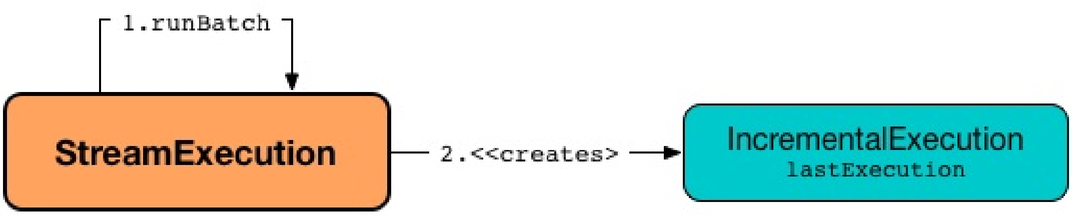

Figure 4. StreamExecution's Query Planning (queryPlanning Phase)

In **queryPlanning** time-tracking section, `runBatch` creates a new `IncrementalExecution` with the following:

- Transformed logical query plan with logical relations for every streaming source and corresponding attributes

- the streaming query's output mode
- `state` checkpoint directory for managing state
- current run id
- current batch id
- OffsetSeqMetadata

The new `IncrementalExecution` is recorded in lastExecution property.

Before leaving **queryPlanning** section, `runBatch` forces preparation of the physical plan for execution (i.e. requesting IncrementalExecution for executedPlan).

Note	executedPlan is a physical plan (i.e. `SparkPlan`) ready for execution with state optimization rules applied.

nextBatch Phase — Creating Dataset (with IncrementalExecution for New Data)

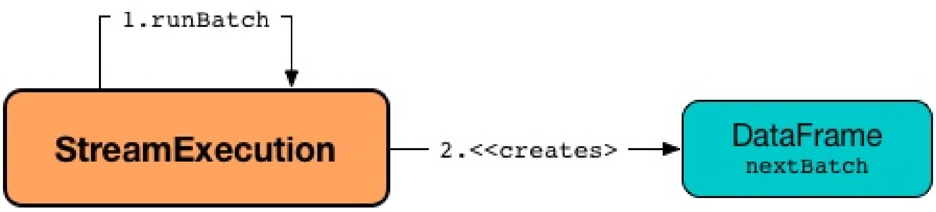

Figure 5. StreamExecution Creates DataFrame with New Data

`runBatch` creates a `DataFrame` with the new IncrementalExecution (as `QueryExecution`) and its analyzed output schema.

Note	The new `DataFrame` represents the result of a streaming query.

addBatch Phase — Adding Current Streaming Batch to Sink

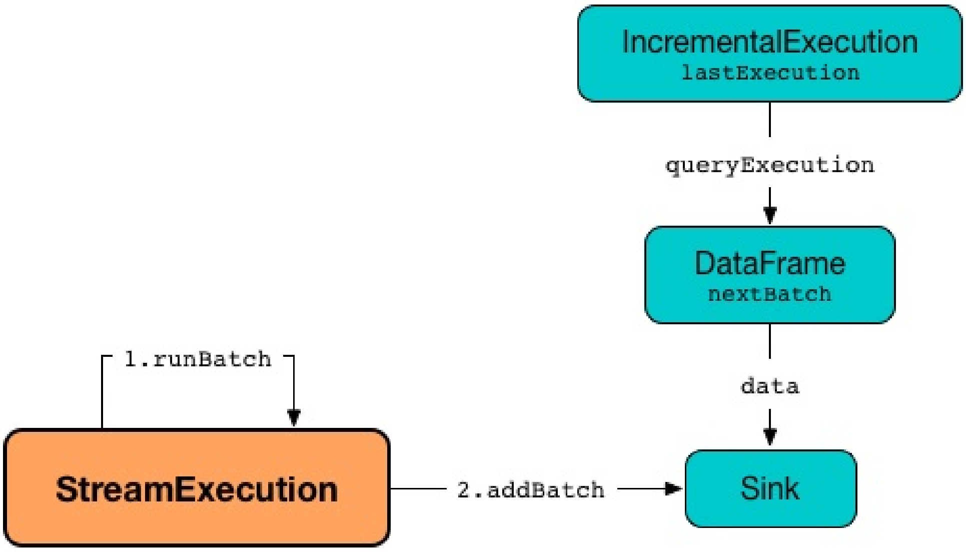

Figure 6. StreamExecution Creates DataFrame with New Data

In **addBatch** time-tracking section, `runBatch` requests the one and only streaming Sink to add the results of a streaming query (as the `DataFrame` created in nextBatch Phase).

Note	`runBatch` uses Sink.addBatch method to request the `Sink` to add the results.

Note	`runBatch` uses `SQLExecution.withNewExecutionId` to execute and track all the Spark actions (under one execution id) that `Sink` can use when requested to add the results.

Note	The new `DataFrame` will only be executed in `Sink.addBatch`.

Note	`SQLExecution.withNewExecutionId` posts a `SparkListenerSQLExecutionStart` event before executing `Sink.addBatch` and a `SparkListenerSQLExecutionEnd` event right afterwards.

Tip	Register `SparkListener` to get notified about the SQL execution events. You can find more information on `SparkListener` in Mastering Apache Spark 2 gitbook.

awaitBatchLock Phase — Waking Up Threads Waiting For Stream to Progress

In **awaitBatchLock** code block (it is not a time-tracking section), `runBatch` acquires a lock on awaitProgressLock, wakes up all waiting threads on awaitProgressLockCondition and immediatelly releases awaitProgressLock lock.

Note	awaitProgressLockCondition is used mainly when `StreamExecution` processAllAvailable (and also when `awaitOffset`, but that seems mainly for testing).

ContinuousExecution

`ContinuousExecution` is…FIXME

`ContinuousExecution` is created exclusively when `DataStreamWriter` is requested to start for a StreamWriteSupport sink and a ContinuousTrigger (when `StreamingQueryManager` is requested to create a serializable StreamingQuery).

Table 1. ContinuousExecution's Internal Properties (e.g. Registries, Counters and Flags)

Name	Description
`continuousSources`	FIXME Used when…FIXME
`currentEpochCoordinatorId`	FIXME Used when…FIXME
`logicalPlan`	
`triggerExecutor`	TriggerExecutor for the Trigger: • `ProcessingTimeExecutor` for `ContinuousTrigger` Used when…FIXME Note: `StreamExecution` throws an `IllegalStateException` when the Trigger is not a ContinuousTrigger.

`getStartOffsets` Internal Method

```
getStartOffsets(sparkSessionToRunBatches: SparkSession): OffsetSeq
```

`getStartOffsets` …FIXME

Note	`getStartOffsets` is used when…FIXME

`commit` Method

```
commit(epoch: Long): Unit
```

`commit` …FIXME

Note	`commit` is used when…FIXME

`awaitEpoch` Internal Method

```
awaitEpoch(epoch: Long): Unit
```

`awaitEpoch` …FIXME

Note	`awaitEpoch` is used when…FIXME

`addOffset` Method

```
addOffset(
  epoch: Long,
  reader: ContinuousReader,
  partitionOffsets: Seq[PartitionOffset]): Unit
```

`addOffset` …FIXME

Note	`addOffset` is used when…FIXME

`sources` Method

```
sources: Seq[BaseStreamingSource]
```

Note	`sources` is part of ProgressReporter Contract to…FIXME.

`sources` …FIXME

`logicalPlan` Value

```
logicalPlan: LogicalPlan
```

Note	`logicalPlan` is part of StreamExecution Contract to…FIXME.

`logicalPlan` …FIXME

Running Activated Streaming Query — `runActivatedStream` Method

```
runActivatedStream(sparkSessionForStream: SparkSession): Unit
```

Note	`runActivatedStream` is part of StreamExecution Contract to run the activated streaming query.

`runActivatedStream` …FIXME

`runContinuous` Internal Method

```
runContinuous(sparkSessionForQuery: SparkSession): Unit
```

`runContinuous` …FIXME

Note	`runContinuous` is used when…FIXME

Creating ContinuousExecution Instance

`ContinuousExecution` takes the following when created:

- `SparkSession`
- Query name
- Path to the checkpoint directory (aka *metadata directory*)
- Analyzed logical query plan (i.e. `LogicalPlan`)
- StreamWriteSupport
- Trigger
- `Clock`
- Output mode
- Map[String, String]
- `deleteCheckpointOnStop` flag to control whether to delete the checkpoint directory on stop

`ContinuousExecution` initializes the internal registries and counters.

StreamingQueryWrapper

`StreamingQueryWrapper` is a wrapper around StreamExecution for…FIXME

`StreamingQueryWrapper` is created when…FIXME

Creating StreamingQueryWrapper Instance

`StreamingQueryWrapper` takes the following when created:

- StreamExecution

`StreamingQueryWrapper` initializes the internal registries and counters.

ProgressReporter Contract

`ProgressReporter` is the contract that StreamExecution uses to report query progress.

```
import org.apache.spark.sql.streaming.Trigger
import scala.concurrent.duration._
val sampleQuery = spark.
  readStream.
  format("rate").
  load.
  writeStream.
  format("console").
  option("truncate", false).
  trigger(Trigger.ProcessingTime(10.seconds)).
  start

// Using public API
import org.apache.spark.sql.streaming.SourceProgress
scala> sampleQuery.
     |   lastProgress.
     |   sources.
     |   map { case sp: SourceProgress =>
     |     s"source = ${sp.description} => endOffset = ${sp.endOffset}" }.
     |   foreach(println)
source = RateSource[rowsPerSecond=1, rampUpTimeSeconds=0, numPartitions=8] => endOffse
t = 663

scala> println(sampleQuery.lastProgress.sources(0))
res40: org.apache.spark.sql.streaming.SourceProgress =
{
  "description" : "RateSource[rowsPerSecond=1, rampUpTimeSeconds=0, numPartitions=8]",
  "startOffset" : 333,
  "endOffset" : 343,
  "numInputRows" : 10,
  "inputRowsPerSecond" : 0.9998000399920015,
  "processedRowsPerSecond" : 200.0
}

// With a hack
import org.apache.spark.sql.execution.streaming.StreamingQueryWrapper
val offsets = sampleQuery.
  asInstanceOf[StreamingQueryWrapper].
  streamingQuery.
  availableOffsets.
  map { case (source, offset) =>
    s"source = $source => offset = $offset" }
scala> offsets.foreach(println)
source = RateSource[rowsPerSecond=1, rampUpTimeSeconds=0, numPartitions=8] => offset =
293
```

Table 1. ProgressReporter’s Internal Registries and Counters (in alphabetical order)

Name	Description

<table>
<tr>
<td><code>currentDurationsMs</code></td>
<td>
Scala's scala.collection.mutable.HashMap of action names (aka triggerDetailKey) and their cumulative times (in milliseconds).
The action names can be as follows:

<code>addBatch</code>
<code>getBatch</code> (when <code>StreamExecution</code> runs a streaming batch)
<code>getOffset</code>
<code>queryPlanning</code>
<code>triggerExecution</code>
<code>walCommit</code> when writing offsets to log

Starts empty when <code>ProgressReporter</code> sets the state for a new batch with new entries added or updated when reporting execution time (of an action).
<table>
<tr>
<td>Tip</td>
<td>
You can see the current value of <code>currentDurationsMs</code> in progress reports under <code>durationMs</code> .
<pre><code>scala> query.lastProgress.durationMs
res3: java.util.Map[String,Long] = {triggerExecution=60, queryPlanning=1, getBatch=5, getOffset=0, addBatch=30, walCommit=23}</code></pre>
</td>
</tr>
</table>
</td>
</tr>
<tr>
<td><code>currentStatus</code></td>
<td>
<code>StreamingQueryStatus</code> to track the status of a streaming query.
Available using status method.
</td>
</tr>
<tr>
<td><code>currentTriggerEndTimestamp</code></td>
<td>Timestamp of when the current batch/trigger has ended</td>
</tr>
<tr>
<td><code>currentTriggerStartTimestamp</code></td>
<td>Timestamp of when the current batch/trigger has started</td>
</tr>
<tr>
<td><code>noDataProgressEventInterval</code></td>
<td>FIXME</td>
</tr>
<tr>
<td><code>lastNoDataProgressEventTime</code></td>
<td>FIXME</td>
</tr>
</table>

`lastTriggerStartTimestamp`	Timestamp of when the last batch/trigger started
`progressBuffer`	Scala's scala.collection.mutable.Queue of StreamingQueryProgress Elements are added and removed when `ProgressReporter` updates progress. Used when `ProgressReporter` does `lastProgress` or `recentProgress`.

Creating Execution Statistics — `extractExecutionStats` Internal Method

```
extractExecutionStats(hasNewData: Boolean): ExecutionStats
```

Caution	FIXME

SourceProgress

Caution	FIXME

SinkProgress

Caution	FIXME

ProgressReporter Contract

```
package org.apache.spark.sql.execution.streaming

trait ProgressReporter {
  // only required methods that have no implementation
  def availableOffsets: StreamProgress
  def committedOffsets: StreamProgress
  def currentBatchId: Long
  def id: UUID
  def logicalPlan: LogicalPlan
  def name: String
  def newData: Map[Source, DataFrame]
  def offsetSeqMetadata: OffsetSeqMetadata
  def postEvent(event: StreamingQueryListener.Event): Unit
  def runId: UUID
  def sink: Sink
  def sources: Seq[Source]
  def triggerClock: Clock
}
```

Table 2. (Subset of) ProgressReporter Contract (in alphabetical order)

Method	Description
`availableOffsets`	StreamProgress Used when: • `ProgressReporter` is requested to finishTrigger (for the JSON-ified offsets of every streaming source to report progress) • `StreamExecution` runs streaming batches, runs a single streaming batch, constructs the next streaming batch, populateStartOffsets, and dataAvailable.
`committedOffsets`	StreamProgress Used when: • `StreamExecution` runs batches, …FIXME
`currentBatchId`	Id of the current batch
`id`	UUID of…FIXME
`logicalPlan`	Logical query plan (i.e. `LogicalPlan`) of a streaming Dataset that…FIXME Used when `ProgressReporter` is requested to extract statistics from the most recent query execution (to add `watermark` metric when a streaming watermark is used)

`name`	Name of…FIXME
`newData`	Streaming sources with the new data as a DataFrame. Used when: • `ProgressReporter` extracts statistics from the most recent query execution (to calculate the so-called `inputRows`)
`offsetSeqMetadata`	
`postEvent`	FIXME
`runId`	UUID of…FIXME
`sink`	Streaming sink
`sources`	Streaming sources
`triggerClock`	`Clock` to track the time

`status` Method

```
status: StreamingQueryStatus
```

`status` gives the current StreamingQueryStatus.

Note	`status` is used when `StreamingQueryWrapper` is requested for the current status of a streaming query (that is part of StreamingQuery Contract).

Reporting Streaming Query Progress — `updateProgress` Internal Method

```
updateProgress(newProgress: StreamingQueryProgress): Unit
```

`updateProgress` records the input `newProgress` and posts a QueryProgressEvent event.

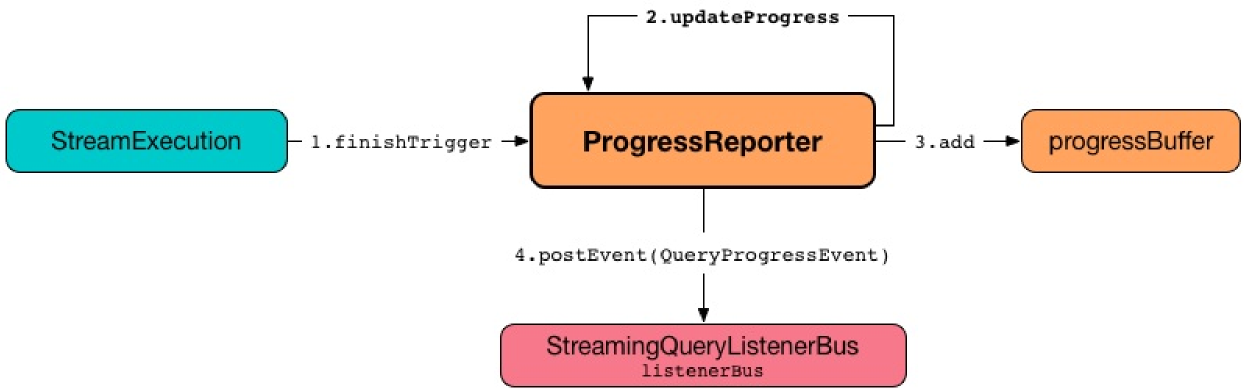

Figure 1. ProgressReporter's Reporting Query Progress

`updateProgress` adds the input `newProgress` to progressBuffer.

`updateProgress` removes elements from progressBuffer if their number is or exceeds the value of spark.sql.streaming.numRecentProgressUpdates property.

`updateProgress` posts a QueryProgressEvent (with the input `newProgress`).

`updateProgress` prints out the following INFO message to the logs:

```
INFO StreamExecution: Streaming query made progress: [newProgress]
```

Note	`updateProgress` synchronizes concurrent access to progressBuffer.

Note	`updateProgress` is used exclusively when `ProgressReporter` finishes a trigger.

Setting State For New Trigger — `startTrigger` Method

```
startTrigger(): Unit
```

When called, `startTrigger` prints out the following DEBUG message to the logs:

```
DEBUG StreamExecution: Starting Trigger Calculation
```

`startTrigger` sets lastTriggerStartTimestamp as currentTriggerStartTimestamp.

`startTrigger` sets currentTriggerStartTimestamp using triggerClock.

`startTrigger` enables `isTriggerActive` flag of StreamingQueryStatus.

`startTrigger` clears currentDurationsMs.

Note	`startTrigger` is used exclusively when `StreamExecution` starts running batches (as part of TriggerExecutor executing a batch runner).

Finishing Trigger (by Updating Progress and Marking Current Status As Trigger Inactive) — `finishTrigger` Method

```
finishTrigger(hasNewData: Boolean): Unit
```

Internally, `finishTrigger` sets currentTriggerEndTimestamp to the current time (using triggerClock).

`finishTrigger` extractExecutionStats.

`finishTrigger` calculates the **processing time** (in seconds) as the difference between the end and start timestamps.

`finishTrigger` calculates the **input time** (in seconds) as the difference between the start time of the current and last triggers.

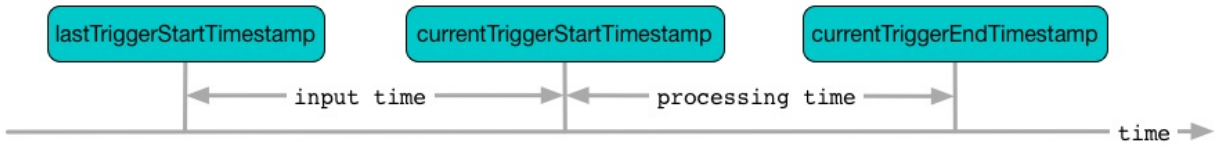

Figure 2. ProgressReporter's finishTrigger and Timestamps

`finishTrigger` prints out the following DEBUG message to the logs:

```
DEBUG StreamExecution: Execution stats: [executionStats]
```

`finishTrigger` creates a SourceProgress (aka source statistics) for every source used.

`finishTrigger` creates a SinkProgress (aka sink statistics) for the sink.

`finishTrigger` creates a StreamingQueryProgress.

If there was any data (using the input `hasNewData` flag), `finishTrigger` resets lastNoDataProgressEventTime (i.e. becomes the minimum possible time) and updates query progress.

Otherwise, when no data was available (using the input `hasNewData` flag), `finishTrigger` updates query progress only when lastNoDataProgressEventTime passed.

In the end, `finishTrigger` disables `isTriggerActive` flag of StreamingQueryStatus (i.e. sets it to `false`).

Note	`finishTrigger` is used exclusively when `StreamExecution` runs streaming batches (after TriggerExecutor has finished executing a streaming batch for a trigger).

Tracking and Recording Execution Time — `reportTimeTaken` Method

```
reportTimeTaken[T](triggerDetailKey: String)(body: => T): T
```

`reportTimeTaken` measures the time to execute `body` and records it in currentDurationsMs.

In the end, `reportTimeTaken` prints out the following DEBUG message to the logs and returns the result of executing `body`.

```
DEBUG StreamExecution: [triggerDetailKey] took [time] ms
```

Note	`reportTimeTaken` is used when `StreamExecution` wants to record the time taken for (as `triggerDetailKey` in the DEBUG message above): • `addBatch` • `getBatch` • `getOffset` • `queryPlanning` • `triggerExecution` • `walCommit` when writing offsets to log

`updateStatusMessage` Method

```
updateStatusMessage(message: String): Unit
```

`updateStatusMessage` updates `message` in StreamingQueryStatus internal registry.

QueryExecutionThread

`QueryExecutionThread` is…FIXME

TriggerExecutor

`TriggerExecutor` is the interface for **trigger executors** that `StreamExecution` uses to execute a batch runner.

Note	**Batch runner** is an executable code that is executed at regular intervals. It is also called a **trigger handler**.

```
package org.apache.spark.sql.execution.streaming

trait TriggerExecutor {
  def execute(batchRunner: () => Boolean): Unit
}
```

Note	`StreamExecution` reports a `IllegalStateException` when TriggerExecutor is different from the two built-in implementations: `OneTimeExecutor` or `ProcessingTimeExecutor`.

Table 1. TriggerExecutor's Available Implementations

TriggerExecutor	Description
`OneTimeExecutor`	Executes `batchRunner` exactly once.
`ProcessingTimeExecutor`	Executes `batchRunner` at regular intervals (as defined using P DataStreamWriter.trigger method). `ProcessingTimeExecutor(processingTime: ProcessingTime, cloc` Note: Processing terminates when `batchRunner` return

`notifyBatchFallingBehind` Method

Caution	FIXME

IncrementalExecution — QueryExecution of Streaming Datasets

`IncrementalExecution` is a `QueryExecution` of a streaming Dataset that StreamExecution creates when incrementally executing the logical query plan (every trigger).

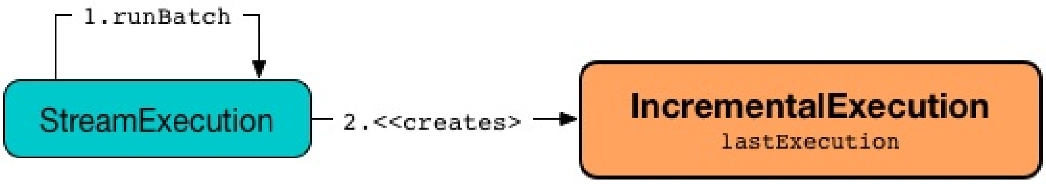

Figure 1. StreamExecution creates IncrementalExecution (every trigger / streaming batch)

Tip	Details on `QueryExecution` contract can be found in the Mastering Apache Spark 2 gitbook.

`IncrementalExecution` registers state physical preparation rule with the parent `QueryExecution` 's `preparations` that prepares the streaming physical plan (using batch-specific execution properties).

`IncrementalExecution` is created when:

- `StreamExecution` plans a streaming query
- `ExplainCommand` is executed (for explain operator)

Table 1. IncrementalExecution's Internal Registries and Counters (in alphabetical order)

Name	Description
`planner`	`SparkPlanner` with the following extra planning strategies (in the order of execution): 1. StatefulAggregationStrategy 2. FlatMapGroupsWithStateStrategy 3. StreamingRelationStrategy 4. StreamingDeduplicationStrategy Note: `planner` is used to plan (aka *convert*) an optimized logical plan into a physical plan (that is later available as `sparkPlan`). `sparkPlan` physical plan is then prepared for execution using preparations physical optimization rules. The result is later available as `executedPlan` physical plan.

`state`	State preparation rule (i.e. `Rule[SparkPlan]`) that transforms a streaming physical plan (i.e. `SparkPlan` with StateStoreSaveExec, StreamingDeduplicateExec and FlatMapGroupsWithStateExec physical operators) and fills missing properties that are batch-specific, e.g.
	• `StateStoreSaveExec` and `StateStoreRestoreExec` operator pair, `state` assigns:
	  ◦ `StateStoreSaveExec` operator gets nextStatefulOperationStateInfo, OutputMode and batchWatermarkMs
	  ◦ `StateStoreRestoreExec` operator gets nextStatefulOperationStateInfo that was used for `StateStoreSaveExec` operator
	• `StreamingDeduplicateExec` operator gets nextStatefulOperationStateInfo and batchWatermarkMs
	• `FlatMapGroupsWithStateExec` gets nextStatefulOperationStateInfo, batchWatermarkMs and batchWatermarkMs
	Used when `IncrementalExecution` prepares a physical plan (i.e. `SparkPlan`) for execution (which is when `StreamExecution` runs a streaming batch and plans a streaming query).

`statefulOperatorId`	Java's `AtomicInteger` • `0` when `IncrementalExecution` is created • Incremented…FIXME

`nextStatefulOperationStateInfo` Internal Method

```
nextStatefulOperationStateInfo(): StatefulOperatorStateInfo
```

`nextStatefulOperationStateInfo` creates a `StatefulOperatorStateInfo` with checkpointLocation, runId, the next statefulOperatorId and currentBatchId.

Note	All the properties of `StatefulOperatorStateInfo` are specified when `IncrementalExecution` is created.

Note	`nextStatefulOperationStateInfo` is used exclusively when `IncrementalExecution` is requested to transform a streaming physical plan using state preparation rule.

Creating IncrementalExecution Instance

`IncrementalExecution` takes the following when created:

- `SparkSession`
- Logical query plan (i.e. `LogicalPlan` with the logical plans of the data sources that have new data and new column attributes)
- OutputMode (as specified using outputMode method of `DataStreamWriter`)
- `state` checkpoint directory (as specified using checkpointLocation option or spark.sql.streaming.checkpointLocation Spark property with `queryName` option)
- Run id
- Current batch id
- OffsetSeqMetadata

`IncrementalExecution` initializes the internal registries and counters.

StatefulOperatorStateInfo

`StatefulOperatorStateInfo` is…FIXME

StreamingQueryListenerBus — Notification Bus for Streaming Events

`StreamingQueryListenerBus` is an event bus (i.e. `ListenerBus`) to dispatch streaming events to StreamingQueryListener streaming event listeners.

`StreamingQueryListenerBus` is created when `StreamingQueryManager` is created (as the internal listenerBus).

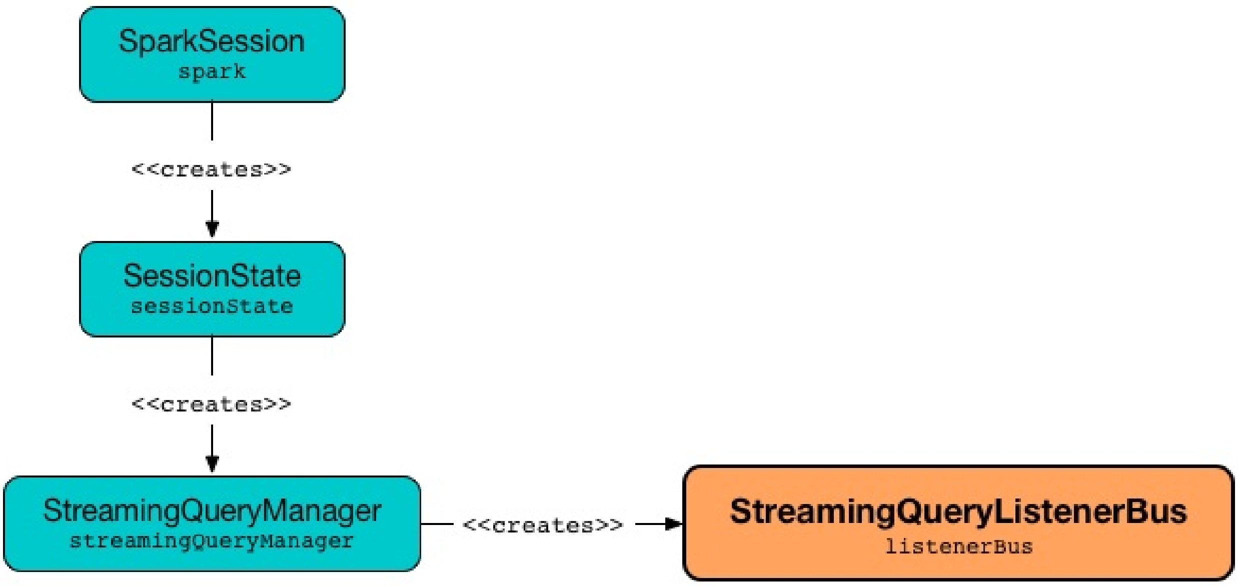

Figure 1. StreamingQueryListenerBus is Created Once In SparkSession

`StreamingQueryListenerBus` is also a `SparkListener` and registers itself with LiveListenerBus to intercept a QueryStartedEvent.

Table 1. StreamingQueryListenerBus's Internal Registries and Counters

Name	Description
`activeQueryRunIds`	Collection of active streaming queries by their runIds. • `runId` is added when `StreamingQueryListenerBus` posts a QueryStartedEvent event to LiveListenerBus • `runId` is removed when `StreamingQueryListenerBus` postToAll a QueryTerminatedEvent event Used mainly when `StreamingQueryListenerBus` dispatches an event to listeners (for queries started in the same `SparkSession`).

Posting StreamingQueryListener Events to LiveListenerBus — `post` Method

```
post(event: StreamingQueryListener.Event): Unit
```

`post` simply posts the input `event` straight to LiveListenerBus except QueryStartedEvent events.

For `QueryStartedEvent` events, `post` adds the query's `runId` to activeQueryRunIds registry first before posting the event to LiveListenerBus followed by postToAll.

Note	`post` is used exclusively when `StreamingQueryManager` posts StreamingQueryListener event.

`onOtherEvent` Method

Caution	FIXME

`doPostEvent` Method

Caution	FIXME

`postToAll` Method

Caution	FIXME

Creating StreamingQueryListenerBus Instance

`StreamingQueryListenerBus` takes the following when created:

- `LiveListenerBus`

`StreamingQueryListenerBus` registers itself with LiveListenerBus.

`StreamingQueryListenerBus` initializes the internal registries and counters.

EventTimeWatermark Unary Logical Operator

`EventTimeWatermark` is a unary logical operator (i.e. `UnaryNode`) that is created as the result of withWatermark operator.

```
val q = spark.
  readStream.
  format("rate").
  load.
  withWatermark(eventTime = "timestamp", delayThreshold = "30 seconds") // <-- creates
 EventTimeWatermark
scala> q.explain(extended = true)
== Parsed Logical Plan ==
'EventTimeWatermark 'timestamp, interval 30 seconds
+- StreamingRelation DataSource(org.apache.spark.sql.SparkSession@3d97b0a,rate,List(),
None,List(),None,Map(),None), rate, [timestamp#10, value#11L]

== Analyzed Logical Plan ==
timestamp: timestamp, value: bigint
EventTimeWatermark timestamp#10: timestamp, interval 30 seconds
+- StreamingRelation DataSource(org.apache.spark.sql.SparkSession@3d97b0a,rate,List(),
None,List(),None,Map(),None), rate, [timestamp#10, value#11L]

== Optimized Logical Plan ==
EventTimeWatermark timestamp#10: timestamp, interval 30 seconds
+- StreamingRelation DataSource(org.apache.spark.sql.SparkSession@3d97b0a,rate,List(),
None,List(),None,Map(),None), rate, [timestamp#10, value#11L]

== Physical Plan ==
EventTimeWatermark timestamp#10: timestamp, interval 30 seconds
+- StreamingRelation rate, [timestamp#10, value#11L]
```

`EventTimeWatermark` uses `spark.watermarkDelayMs` key (in `Metadata` in output) to hold the event-time watermark delay.

Note	The **event-time watermark delay** is used to calculate the difference between the event time of an event (that is modelled as a row in the Dataset for a streaming batch) and the time in the past.

Note	`EliminateEventTimeWatermark` logical optimization rule (i.e. `Rule[LogicalPlan]`) ren `EventTimeWatermark` logical operator from a logical plan if child logical operator is r streaming, i.e. when withWatermark operator is used on a batch query.

```
val logs = spark.
  read. // <-- batch non-streaming query that makes `EliminateEventTimeWatermar
plicable
  format("text").
  load("logs")

// logs is a batch Dataset
scala> logs.isStreaming
res0: Boolean = false

val q = logs.
  withWatermark(eventTime = "timestamp", delayThreshold = "30 seconds") // <--
entTimeWatermark
scala> println(q.queryExecution.logical.numberedTreeString) // <-- no EventTime
as it was removed immediately
00 Relation[value#0] text
```

Note	`EventTimeWatermark` is converted (aka *planned*) to EventTimeWatermarkExec physical operator in StatefulAggregationStrategy execution planning strategy.

`output` Property

```
output: Seq[Attribute]
```

Note	`output` is a part of the `QueryPlan` Contract to describe the attributes of (the schema of) the output.

`output` finds eventTime column in the child's output schema and updates the `Metadata` of the column with spark.watermarkDelayMs key and the milliseconds for the delay.

`output` removes spark.watermarkDelayMs key from the other columns.

```
// See q created above
// FIXME How to access/show the eventTime column with the metadata updated to include
spark.watermarkDelayMs?
import org.apache.spark.sql.catalyst.plans.logical.EventTimeWatermark
val etw = q.queryExecution.logical.asInstanceOf[EventTimeWatermark]
scala> etw.output.toStructType.printTreeString
root
 |-- timestamp: timestamp (nullable = true)
 |-- value: long (nullable = true)
```

Creating EventTimeWatermark Instance

`EventTimeWatermark` takes the following when created:

- Event time column
- Delay `CalendarInterval`
- Child logical plan

FlatMapGroupsWithState Unary Logical Operator

`FlatMapGroupsWithState` is a unary logical operator (i.e. `LogicalPlan`) that is created to represent the following operators in KeyValueGroupedDataset:

- mapGroupsWithState
- flatMapGroupsWithState

Note	`FlatMapGroupsWithState` is translated to: • FlatMapGroupsWithStateExec physical operator in FlatMapGroupsWithStateStrategy execution planning strategy for streaming Datasets (aka streaming plans) • `MapGroupsExec` physical operator in `BasicOperators` execution planning strategy for non-streaming/batch Datasets (aka batch plans)

Creating SerializeFromObject with FlatMapGroupsWithState — `apply` Factory Method

```
apply[K: Encoder, V: Encoder, S: Encoder, U: Encoder](
  func: (Any, Iterator[Any], LogicalGroupState[Any]) => Iterator[Any],
  groupingAttributes: Seq[Attribute],
  dataAttributes: Seq[Attribute],
  outputMode: OutputMode,
  isMapGroupsWithState: Boolean,
  timeout: GroupStateTimeout,
  child: LogicalPlan): LogicalPlan
```

`apply` creates a `SerializeFromObject` logical operator with a `FlatMapGroupsWithState` as its child logical operator.

Internally, `apply` creates `SerializeFromObject` object consumer (aka unary logical operator) with `FlatMapGroupsWithState` logical plan.

Internally, `apply` finds `ExpressionEncoder` for the type `S` and creates a `FlatMapGroupsWithState` with `UnresolvedDeserializer` for the types `K` and `V` .

In the end, `apply` creates a `SerializeFromObject` object consumer with the `FlatMapGroupsWithState` .

Note	`apply` is used when flatMapGroupsWithState is executed.

Creating FlatMapGroupsWithState Instance

`FlatMapGroupsWithState` takes the following when created:

- State function of type `(Any, Iterator[Any], LogicalGroupState[Any]) ⇒ Iterator[Any]`
- Key deserializer Catalyst expression
- Value deserializer Catalyst expression
- Grouping attributes
- Data attributes
- Output object attribute
- State `ExpressionEncoder`
- Output mode
- `isMapGroupsWithState` Flag (disabled by default)
- GroupStateTimeout
- Child logical operator

Deduplicate Unary Logical Operator

`Deduplicate` is a unary logical operator (i.e. `LogicalPlan`) that is created to represent dropDuplicates operator (that drops duplicate records for a given subset of columns).

`Deduplicate` has streaming flag enabled for streaming Datasets.

```
val uniqueRates = spark.
  readStream.
  format("rate").
  load.
  dropDuplicates("value")  // <-- creates Deduplicate logical operator
// Note the streaming flag
scala> println(uniqueRates.queryExecution.logical.numberedTreeString)
00 Deduplicate [value#33L], true  // <-- streaming flag enabled
01 +- StreamingRelation DataSource(org.apache.spark.sql.SparkSession@4785f176,rate,List
(),None,List(),None,Map(),None), rate, [timestamp#32, value#33L]
```

Caution	FIXME Example with duplicates across batches to show that `Deduplicate` keeps state and withWatermark operator should also be used to limit how much is stored (to not cause OOM)

Note

`UnsupportedOperationChecker` ensures that dropDuplicates operator is not used aft

The following code is not supported in Structured Streaming and results in an `Ana`

```
val counts = spark.
  readStream.
  format("rate").
  load.
  groupBy(window($"timestamp", "5 seconds") as "group").
  agg(count("value") as "value_count").
  dropDuplicates  // <-- after groupBy

import scala.concurrent.duration._
import org.apache.spark.sql.streaming.{OutputMode, Trigger}
val sq = counts.
  writeStream.
  format("console").
  trigger(Trigger.ProcessingTime(10.seconds)).
  outputMode(OutputMode.Complete).
  start
org.apache.spark.sql.AnalysisException: dropDuplicates is not supported after a
```

Note	`Deduplicate` logical operator is translated (aka *planned*) to: • StreamingDeduplicateExec physical operator in StreamingDeduplicationStrategy execution planning strategy for streaming Datasets (aka *streaming plans*) • `Aggregate` physical operator in `ReplaceDeduplicateWithAggregate` execution planning strategy for non-streaming/batch Datasets (aka *batch plans*)

The output schema of `Deduplicate` is exactly the child's output schema.

Creating Deduplicate Instance

`Deduplicate` takes the following when created:

- Attributes for keys
- Child logical operator (i.e. `LogicalPlan`)
- Flag whether the logical operator is for streaming (enabled) or batch (disabled) mode

`MemoryPlan` Logical Operator

`MemoryPlan` is a leaf logical operator (i.e. `LogicalPlan`) that is used to query the data that has been written into a MemorySink. `MemoryPlan` is created when starting continuous writing (to a `MemorySink`).

Tip	See the example in MemoryStream.

```
scala> intsOut.explain(true)
== Parsed Logical Plan ==
SubqueryAlias memstream
+- MemoryPlan org.apache.spark.sql.execution.streaming.MemorySink@481bf251, [value#21]

== Analyzed Logical Plan ==
value: int
SubqueryAlias memstream
+- MemoryPlan org.apache.spark.sql.execution.streaming.MemorySink@481bf251, [value#21]

== Optimized Logical Plan ==
MemoryPlan org.apache.spark.sql.execution.streaming.MemorySink@481bf251, [value#21]

== Physical Plan ==
LocalTableScan [value#21]
```

When executed, `MemoryPlan` is translated to `LocalTableScanExec` physical operator (similar to `LocalRelation` logical operator) in `BasicOperators` execution planning strategy.

StreamingRelation Leaf Logical Operator for Streaming Source

`StreamingRelation` is a leaf logical operator (i.e. `LogicalPlan`) that represents a streaming source in a logical plan.

`StreamingRelation` is created when `DataStreamReader` is requested to load data from a streaming source and creates a streaming `Dataset` .

Figure 1. StreamingRelation Represents Streaming Source

```
val rate = spark.
  readStream.     // <-- creates a DataStreamReader
  format("rate").
  load("hello")   // <-- creates a StreamingRelation
scala> println(rate.queryExecution.logical.numberedTreeString)
00 StreamingRelation DataSource(org.apache.spark.sql.SparkSession@4e5dcc50,rate,List(),
None,List(),None,Map(path -> hello),None), rate, [timestamp#0, value#1L]
```

`isStreaming` is always enabled (i.e. `true`).

```
import org.apache.spark.sql.execution.streaming.StreamingRelation
val relation = rate.queryExecution.logical.asInstanceOf[StreamingRelation]
scala> relation.isStreaming
res1: Boolean = true
```

`toString` gives the source name.

```
scala> println(relation)
rate
```

Note	`StreamingRelation` is resolved (aka *planned*) to StreamingExecutionRelation (right after `StreamExecution` starts running batches).

Creating StreamingRelation for DataSource — `apply` Factory Method

```
apply(dataSource: DataSource): StreamingRelation
```

`apply` creates a StreamingRelation for the input streaming DataSource and the short name and the schema of the streaming source (behind the `DataSource`).

Note	`apply` creates a `StreamingRelation` logical operator (for the input DataSource) that represents a streaming source.

Note	`apply` is used exclusively when `DataStreamReader` is requested to load data from a streaming source to a streaming Dataset.

Creating StreamingRelation Instance

`StreamingRelation` takes the following when created:

- DataSource
- Short name of the streaming source
- Output attributes of the schema of the streaming source

StreamingExecutionRelation Leaf Logical Operator for Streaming Source At Execution

`StreamingExecutionRelation` is a leaf logical operator (i.e. `LogicalPlan`) that represents a streaming source in the logical query plan of a streaming `Dataset`.

The main use of `StreamingExecutionRelation` logical operator is to be a "placeholder" in a logical query plan that will be replaced with the real relation (with new data that has arrived since the last batch) or an empty `LocalRelation` when `StreamExecution` runs a single streaming batch.

`StreamingExecutionRelation` is created for a StreamingRelation in analyzed logical query plan (that is the execution representation of a streaming Dataset).

Note	Right after `StreamExecution` has started running streaming batches it initializes the streaming sources by transforming the analyzed logical plan of the streaming Dataset so that every StreamingRelation logical operator is replaced by the corresponding `StreamingExecutionRelation`.

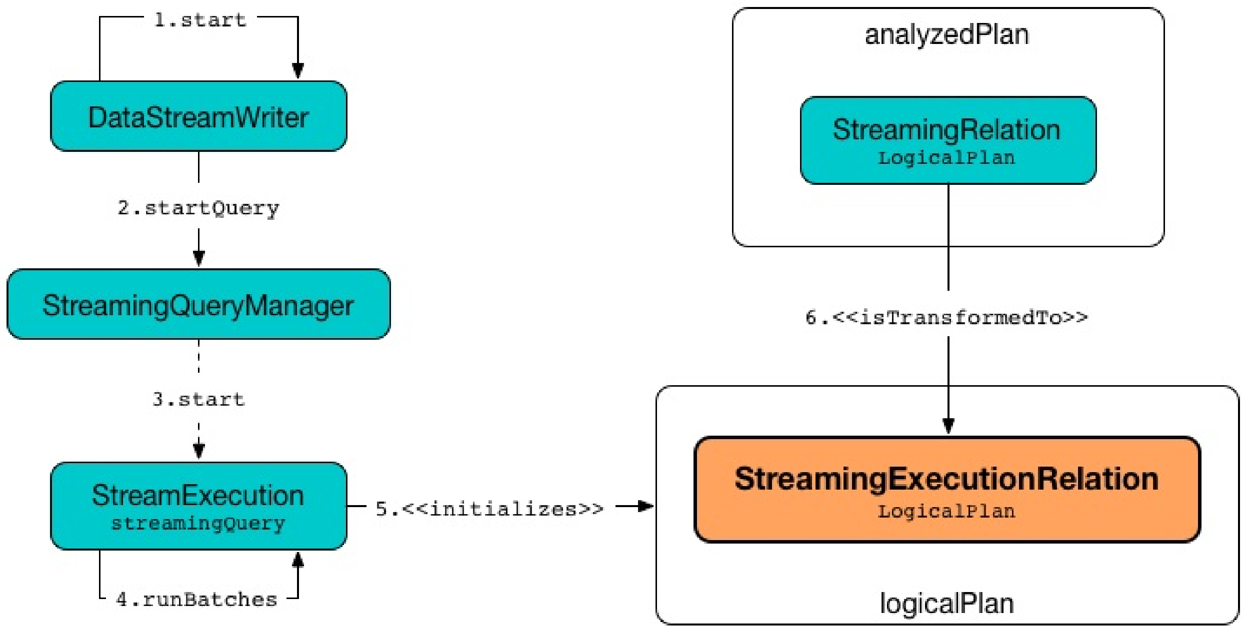

Figure 1. StreamingExecutionRelation Represents Streaming Source At Execution

Note	`StreamingExecutionRelation` is also resolved (aka *planned*) to a StreamingRelationExec physical operator in StreamingRelationStrategy execution planning strategy only when explaining a streaming `Dataset`.

Creating StreamingExecutionRelation Instance

`StreamingExecutionRelation` takes the following when created:

- Streaming source
- Output attributes

Creating StreamingExecutionRelation (for MemoryStream Source) — `apply` Factory Method

```
apply(source: Source): StreamingExecutionRelation
```

`apply` creates a `StreamingExecutionRelation` for the input `source` and with the attributes of the schema of the `source`.

Note	`apply` is used exclusively when `MemoryStream` is created (and the logical plan initialized).

EventTimeWatermarkExec Unary Physical Operator for Accumulating Event Time Watermark

`EventTimeWatermarkExec` is a unary physical operator (aka `UnaryExecNode`) that accumulates the event time values (that appear in eventTime watermark column).

Note	`EventTimeWatermarkExec` uses eventTimeStats accumulator to send the statistics (i.e. the maximum, minimum, average and count) for the event time column in a streaming batch that is later used in: • `ProgressReporter` for creating execution statistics for the most recent query execution. You should then see `max`, `min`, `avg`, and `watermark` eventTime watermark statistics. • `StreamExecution` to observe and possibly update eventTime watermark while constructing the next streaming batch.

`EventTimeWatermarkExec` is created when StatefulAggregationStrategy execution planning strategy plans a `EventTimeWatermark` logical operator for execution.

Note	EventTimeWatermark logical operator is created as the result of withWatermark operator.

```
val rates = spark.
  readStream.
  format("rate").
  load.
  withWatermark(eventTime = "timestamp", delayThreshold = "10 seconds") // <-- use Eve
ntTimeWatermark logical operator
scala> rates.explain
== Physical Plan ==
EventTimeWatermark timestamp#0: timestamp, interval 10 seconds
+- StreamingRelation rate, [timestamp#0, value#1L]

import org.apache.spark.sql.streaming.{OutputMode, Trigger}
import scala.concurrent.duration._
val sq = rates.
  writeStream.
  format("console").
  option("truncate", false).
  trigger(Trigger.ProcessingTime(10.seconds)).
  outputMode(OutputMode.Append).
  queryName("rates-to-console").
  start
17/08/11 09:04:17 INFO StreamExecution: Starting rates-to-console [id = ec8f8228-90f6-
```

```
4e1f-8ad2-80222affed63, runId = f605c134-cfb0-4378-88c1-159d8a7c232e] with file:///pri
vate/var/folders/0w/kb0d3rqn4zb9fcc91pxhgn8w0000gn/T/temporary-3869a982-9824-4715-8cce
-cce7c8251299 to store the query checkpoint.
...
-------------------------------------------
Batch: 0
-------------------------------------------
+---------+-----+
|timestamp|value|
+---------+-----+
+---------+-----+
...
17/08/11 09:04:17 DEBUG StreamExecution: Execution stats: ExecutionStats(Map(RateSource
[rowsPerSecond=1, rampUpTimeSeconds=0, numPartitions=8] -> 0),ArrayBuffer(),Map(waterm
ark -> 1970-01-01T00:00:00.000Z))
...
17/08/11 09:04:17 INFO StreamExecution: Streaming query made progress: {
  "id" : "ec8f8228-90f6-4e1f-8ad2-80222affed63",
  "runId" : "f605c134-cfb0-4378-88c1-159d8a7c232e",
  "name" : "rates-to-console",
  "timestamp" : "2017-08-11T07:04:17.373Z",
  "batchId" : 0,
  "numInputRows" : 0,
  "processedRowsPerSecond" : 0.0,
  "durationMs" : {
    "addBatch" : 38,
    "getBatch" : 1,
    "getOffset" : 0,
    "queryPlanning" : 1,
    "triggerExecution" : 62,
    "walCommit" : 19
  },
  "eventTime" : {
    "watermark" : "1970-01-01T00:00:00.000Z"  // <-- no watermark found yet
  },
...
17/08/11 09:04:17 DEBUG StreamExecution: batch 0 committed
...
-------------------------------------------
Batch: 1
-------------------------------------------
+-----------------------+-----+
|timestamp              |value|
+-----------------------+-----+
|2017-08-11 09:04:17.282|0    |
|2017-08-11 09:04:18.282|1    |
+-----------------------+-----+
...
17/08/11 09:04:20 DEBUG StreamExecution: Execution stats: ExecutionStats(Map(RateSource
[rowsPerSecond=1, rampUpTimeSeconds=0, numPartitions=8] -> 2),ArrayBuffer(),Map(max ->
2017-08-11T07:04:18.282Z, min -> 2017-08-11T07:04:17.282Z, avg -> 2017-08-11T07:04:17.
782Z, watermark -> 1970-01-01T00:00:00.000Z))
...
```

```
//
// Notice eventTimeStats in eventTime section below
// They are only available when watermark is used and
// EventTimeWatermarkExec.eventTimeStats.value.count > 0, i.e.
// there were input rows (with event time)
// Note that watermark has NOT been changed yet (perhaps it should have)
//
17/08/11 09:04:20 INFO StreamExecution: Streaming query made progress: {
  "id" : "ec8f8228-90f6-4e1f-8ad2-80222affed63",
  "runId" : "f605c134-cfb0-4378-88c1-159d8a7c232e",
  "name" : "rates-to-console",
  "timestamp" : "2017-08-11T07:04:20.004Z",
  "batchId" : 1,
  "numInputRows" : 2,
  "inputRowsPerSecond" : 0.7601672367920943,
  "processedRowsPerSecond" : 25.31645569620253,
  "durationMs" : {
    "addBatch" : 48,
    "getBatch" : 6,
    "getOffset" : 0,
    "queryPlanning" : 1,
    "triggerExecution" : 79,
    "walCommit" : 23
  },
  "eventTime" : {
    "avg" : "2017-08-11T07:04:17.782Z",
    "max" : "2017-08-11T07:04:18.282Z",
    "min" : "2017-08-11T07:04:17.282Z",
    "watermark" : "1970-01-01T00:00:00.000Z"
  },
...
17/08/11 09:04:20 DEBUG StreamExecution: batch 1 committed
...
//
// At long last!
// I think it should have been a batch earlier
// I did ask about it on the dev mailing list today (on 17/08/11)
//
17/08/11 09:04:30 DEBUG StreamExecution: Observed event time stats: EventTimeStats(150
2435058282,1502435057282,1.502435057782E12,2)
17/08/11 09:04:30 INFO StreamExecution: Updating eventTime watermark to: 1502435048282
 ms
...
-------------------------------------------
Batch: 2
-------------------------------------------
+-----------------------+-----+
|timestamp              |value|
+-----------------------+-----+
|2017-08-11 09:04:19.282|2    |
|2017-08-11 09:04:20.282|3    |
|2017-08-11 09:04:21.282|4    |
|2017-08-11 09:04:22.282|5    |
```

```
|2017-08-11 09:04:23.282|6    |
|2017-08-11 09:04:24.282|7    |
|2017-08-11 09:04:25.282|8    |
|2017-08-11 09:04:26.282|9    |
|2017-08-11 09:04:27.282|10   |
|2017-08-11 09:04:28.282|11   |
+-----------------------+-----+
...
17/08/11 09:04:30 DEBUG StreamExecution: Execution stats: ExecutionStats(Map(RateSource
[rowsPerSecond=1, rampUpTimeSeconds=0, numPartitions=8] -> 10),ArrayBuffer(),Map(max -
> 2017-08-11T07:04:28.282Z, min -> 2017-08-11T07:04:19.282Z, avg -> 2017-08-11T07:04:2
3.782Z, watermark -> 2017-08-11T07:04:08.282Z))
...
17/08/11 09:04:30 INFO StreamExecution: Streaming query made progress: {
  "id" : "ec8f8228-90f6-4e1f-8ad2-80222affed63",
  "runId" : "f605c134-cfb0-4378-88c1-159d8a7c232e",
  "name" : "rates-to-console",
  "timestamp" : "2017-08-11T07:04:30.003Z",
  "batchId" : 2,
  "numInputRows" : 10,
  "inputRowsPerSecond" : 1.000100010001,
  "processedRowsPerSecond" : 56.17977528089888,
  "durationMs" : {
    "addBatch" : 147,
    "getBatch" : 6,
    "getOffset" : 0,
    "queryPlanning" : 1,
    "triggerExecution" : 178,
    "walCommit" : 22
  },
  "eventTime" : {
    "avg" : "2017-08-11T07:04:23.782Z",
    "max" : "2017-08-11T07:04:28.282Z",
    "min" : "2017-08-11T07:04:19.282Z",
    "watermark" : "2017-08-11T07:04:08.282Z"
  },
...
17/08/11 09:04:30 DEBUG StreamExecution: batch 2 committed
...

// In the end, stop the streaming query
sq.stop
```

Table 1. EventTimeWatermarkExec's Internal Registries and Counters

Name	Description
`delayMs`	FIXME Used when…FIXME
`eventTimeStats`	EventTimeStatsAccum accumulator to accumulate eventTime values from every row in a streaming batch (when `EventTimeWatermarkExec` is executed). Note: `EventTimeStatsAccum` is a Spark accumulator of `EventTimeStats` from `Longs` (i.e. `AccumulatorV2[Long, EventTimeStats]`). Note: Every Spark accumulator has to be registered before use, and `eventTimeStats` is registered when `EventTimeWatermarkExec` is created.

Executing EventTimeWatermarkExec (And Collecting Event Times) — `doExecute` Method

```
doExecute(): RDD[InternalRow]
```

Note	`doExecute` is a part of `SparkPlan` contract to produce the result of a physical operator as an RDD of internal binary rows (i.e. `InternalRow`).

Internally, `doExecute` executes child physical operator and maps over the partitions (using `RDD.mapPartitions`) that does the following:

1. Creates an unsafe projection for eventTime in the output schema of child physical operator.
2. For every row (as `InternalRow`)
    - Adds eventTime to eventTimeStats acumulator

Creating EventTimeWatermarkExec Instance

`EventTimeWatermarkExec` takes the following when created:

- Name of the eventTime watermark column
- Delay `CalendarInterval`

- Child physical plan

While being created, `EventTimeWatermarkExec` registers eventTimeStats accumulator (with the current `SparkContext`).

`EventTimeWatermarkExec` initializes the internal registries and counters.

EventTimeStatsAccum Accumulator

`EventTimeStatsAccum` is a Spark accumulator of `EventTimeStats` from `Longs` (i.e. `AccumulatorV2[Long, EventTimeStats]`) that collects statistics on event time in a streaming batch.

`EventTimeStatsAccum` takes a `EventTimeStats` when created.

`EventTimeStatsAccum` is used exclusively when `EventTimeWatermarkExec` is executed (and accumulates `eventTime` values from every row in a streaming batch).

FlatMapGroupsWithStateExec Unary Physical Operator

`FlatMapGroupsWithStateExec` is a unary physical operator (aka `UnaryExecNode`) that is created when FlatMapGroupsWithStateStrategy execution planning strategy plans FlatMapGroupsWithState logical operator for execution.

Note	FlatMapGroupsWithState logical operator is created as the result of flatMapGroupsWithState operator.

```
import java.sql.Timestamp
import org.apache.spark.sql.streaming.GroupState
val stateFunc = (key: Long, values: Iterator[(Timestamp, Long)], state: GroupState[Long
]) => {
  Iterator((key, values.size))
}
import java.sql.Timestamp
import org.apache.spark.sql.streaming.{GroupState, GroupStateTimeout, OutputMode}
val rateGroups = spark.
  readStream.
  format("rate").
  load.
  withWatermark(eventTime = "timestamp", delayThreshold = "10 seconds").  // required
for EventTimeTimeout
  as[(Timestamp, Long)].  // leave DataFrame for Dataset
  groupByKey { case (time, value) => value % 2 }. // creates two groups
  flatMapGroupsWithState(OutputMode.Update, GroupStateTimeout.EventTimeTimeout)(stateF
unc)  // EventTimeTimeout requires watermark (defined above)

// Check out the physical plan with FlatMapGroupsWithStateExec
scala> rateGroups.explain
== Physical Plan ==
*SerializeFromObject [assertnotnull(input[0, scala.Tuple2, true])._1 AS _1#35L, assert
notnull(input[0, scala.Tuple2, true])._2 AS _2#36]
+- FlatMapGroupsWithState <function3>, value#30: bigint, newInstance(class scala.Tuple2
), [value#30L], [timestamp#20-T10000ms, value#21L], obj#34: scala.Tuple2, StatefulOper
atorStateInfo(<unknown>,63491721-8724-4631-b6bc-3bb1edeb4baf,0,0), class[value[0]: big
int], Update, EventTimeTimeout, 0, 0
   +- *Sort [value#30L ASC NULLS FIRST], false, 0
      +- Exchange hashpartitioning(value#30L, 200)
         +- AppendColumns <function1>, newInstance(class scala.Tuple2), [input[0, bigi
nt, false] AS value#30L]
            +- EventTimeWatermark timestamp#20: timestamp, interval 10 seconds
               +- StreamingRelation rate, [timestamp#20, value#21L]

// Execute the streaming query
import org.apache.spark.sql.streaming.{OutputMode, Trigger}
import scala.concurrent.duration._
val sq = rateGroups.
  writeStream.
  format("console").
  trigger(Trigger.ProcessingTime(10.seconds)).
  outputMode(OutputMode.Update).  // Append is not supported
  start

// Eventually...
sq.stop
```

Table 1. FlatMapGroupsWithStateExec's SQLMetrics

Name	Description
`numTotalStateRows`	Number of keys in the StateStore Incremented when `FlatMapGroupsWithStateExec` is executed (and the iterator has finished generating the rows).
`stateMemory`	Memory used by the StateStore

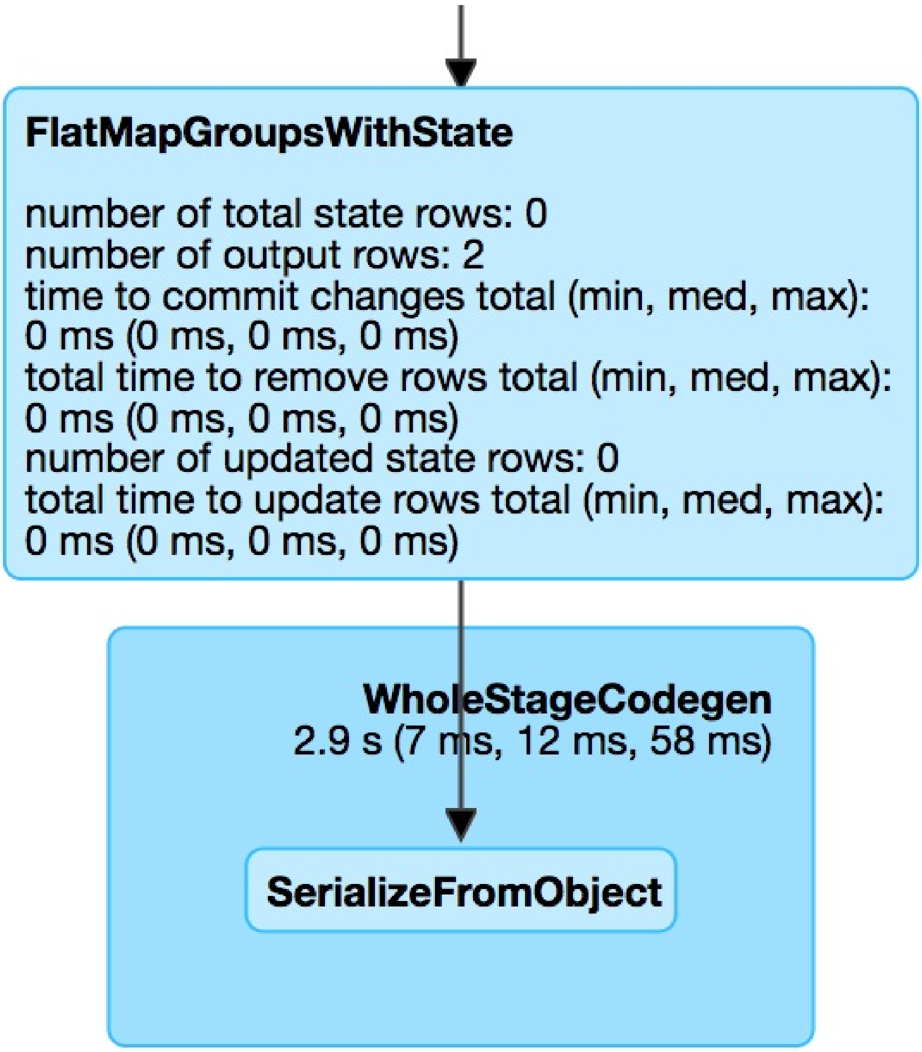

Figure 1. FlatMapGroupsWithStateExec in web UI (Details for Query)

`FlatMapGroupsWithStateExec` is a `ObjectProducerExec` that…FIXME

`FlatMapGroupsWithStateExec` is a `StateStoreWriter` that…FIXME

`FlatMapGroupsWithStateExec` supports watermark which is…FIXME

Note	`FlatMapGroupsWithStateStrategy` converts FlatMapGroupsWithState unary logical operator to `FlatMapGroupsWithStateExec` physical operator with undefined StatefulOperatorStateInfo, batchTimestampMs, and eventTimeWatermark. StatefulOperatorStateInfo, batchTimestampMs, and eventTimeWatermark are defined when `IncrementalExecution` query execution pipeline is requested to apply the physical plan preparation rules.

When executed, `FlatMapGroupsWithStateExec` requires that the optional values are properly defined given timeoutConf:

- batchTimestampMs for `ProcessingTimeTimeout`
- eventTimeWatermark and watermarkExpression for `EventTimeTimeout`

Caution	FIXME Where are the optional values defined?

Table 2. FlatMapGroupsWithStateExec's Internal Registries and Counters (in alphabetical order)

Name	Description
`isTimeoutEnabled`	
`stateAttributes`	
`stateDeserializer`	
`stateSerializer`	
`timestampTimeoutAttribute`	

Tip

Enable `INFO` logging level for `org.apache.spark.sql.execution.streaming.FlatMapGroupsWithStateExec` to see what happens inside.

Add the following line to `conf/log4j.properties`:

```
log4j.logger.org.apache.spark.sql.execution.streaming.FlatMapGroupsWithStateExec
```

Refer to Logging.

`keyExpressions` Method

Caution	FIXME

Executing FlatMapGroupsWithStateExec — `doExecute` Method

```
doExecute(): RDD[InternalRow]
```

Note	`doExecute` is a part of `SparkPlan` contract to produce the result of a physical operator as an RDD of internal binary rows (i.e. `InternalRow`).

Internally, `doExecute` initializes metrics.

`doExecute` then executes child physical operator and creates a StateStoreRDD with `storeUpdateFunction` that:

1. Creates a StateStoreUpdater
2. Filters out rows from `Iterator[InternalRow]` that match `watermarkPredicateForData` (when defined and timeoutConf is `EventTimeTimeout`)
3. Generates an output `Iterator[InternalRow]` with elements from `StateStoreUpdater` 's updateStateForKeysWithData and updateStateForTimedOutKeys
4. In the end, `storeUpdateFunction` creates a `CompletionIterator` that executes a completion function (aka `completionFunction`) after it has successfully iterated through all the elements (i.e. when a client has consumed all the rows). The completion method requests `StateStore` to commit followed by updating `numTotalStateRows` metric with the number of keys in the state store.

Creating FlatMapGroupsWithStateExec Instance

`FlatMapGroupsWithStateExec` takes the following when created:

- State function of type `(Any, Iterator[Any], LogicalGroupState[Any]) ⇒ Iterator[Any]`
- Key deserializer Catalyst expression
- Value deserializer Catalyst expression
- Grouping attributes (as used for grouping in KeyValueGroupedDataset for `mapGroupsWithState` or `flatMapGroupsWithState` operators)
- Data attributes
- Output object attribute
- Optional StatefulOperatorStateInfo

- State `ExpressionEncoder`
- OutputMode
- GroupStateTimeout
- Optional `batchTimestampMs`
- Optional event time watermark
- Child physical operator

`FlatMapGroupsWithStateExec` initializes the internal registries and counters.

StateStoreRestoreExec Unary Physical Operator — Restoring State of Streaming Aggregates

`StateStoreRestoreExec` is a unary physical operator (i.e. `UnaryExecNode`) that restores a state from a state store (for the keys in the input rows).

`StateStoreRestoreExec` is created exclusively when `StatefulAggregationStrategy` plans streaming aggregate operators (aka *streaming aggregates*).

Figure 1. StateStoreRestoreExec and StatefulAggregationStrategy

Note	`Aggregate` logical operator is the result of: • `RelationalGroupedDataset` aggregations, i.e. `agg` and `pivot` operators • `KeyValueGroupedDataset` aggregations, i.e. `mapGroups`, `flatMapGroups`, `mapGroupsWithState`, `flatMapGroupsWithState`, `reduceGroups`, and `agg`, `cogroup` operators • SQL's `GROUP BY` clause (possibly with `WITH CUBE` or `WITH ROLLUP`)

The optional property StatefulOperatorStateInfo is initially undefined (i.e. when `StateStoreRestoreExec` is created). `StateStoreRestoreExec` is updated to hold the streaming batch-specific execution property when `IncrementalExecution` prepares a streaming physical plan for execution (and state preparation rule is executed when `StreamExecution` plans a streaming query for a streaming batch).

Figure 2. StateStoreRestoreExec and IncrementalExecution

```
val counts = spark.
  readStream.
  format("rate").
  load.
  withWatermark(eventTime = "timestamp", delayThreshold = "20 seconds").
  groupBy(window($"timestamp", "5 seconds") as "group").
  agg(count("value") as "value_count").
  orderBy($"value_count".asc)
```

```
// Logical plan with Aggregate logical operator
scala> println(counts.queryExecution.logical.numberedTreeString)
00 'Sort ['value_count ASC NULLS FIRST], true
01 +- Aggregate [window#66-T20000ms], [window#66-T20000ms AS group#59, count(value#53L
) AS value_count#65L]
02    +- Filter isnotnull(timestamp#52-T20000ms)
03       +- Project [named_struct(start, precisetimestampconversion(((((CASE WHEN (cas
t(CEIL((cast((precisetimestampconversion(timestamp#52-T20000ms, TimestampType, LongType
) - 0) as double) / cast(5000000 as double))) as double) = (cast((precisetimestampconv
ersion(timestamp#52-T20000ms, TimestampType, LongType) - 0) as double) / cast(5000000
as double))) THEN (CEIL((cast((precisetimestampconversion(timestamp#52-T20000ms, Times
tampType, LongType) - 0) as double) / cast(5000000 as double))) + cast(1 as bigint)) E
LSE CEIL((cast((precisetimestampconversion(timestamp#52-T20000ms, TimestampType, LongT
ype) - 0) as double) / cast(5000000 as double))) END + cast(0 as bigint)) - cast(1 as
bigint)) * 5000000) + 0), LongType, TimestampType), end, precisetimestampconversion(((
(((CASE WHEN (cast(CEIL((cast((precisetimestampconversion(timestamp#52-T20000ms, Times
tampType, LongType) - 0) as double) / cast(5000000 as double))) as double) = (cast((pr
ecisetimestampconversion(timestamp#52-T20000ms, TimestampType, LongType) - 0) as doubl
e) / cast(5000000 as double))) THEN (CEIL((cast((precisetimestampconversion(timestamp#
52-T20000ms, TimestampType, LongType) - 0) as double) / cast(5000000 as double))) + ca
st(1 as bigint)) ELSE CEIL((cast((precisetimestampconversion(timestamp#52-T20000ms, Ti
mestampType, LongType) - 0) as double) / cast(5000000 as double))) END + cast(0 as big
int)) - cast(1 as bigint)) * 5000000) + 0) + 5000000), LongType, TimestampType)) AS wi
ndow#66, timestamp#52-T20000ms, value#53L]
04          +- EventTimeWatermark timestamp#52: timestamp, interval 20 seconds
05             +- StreamingRelation DataSource(org.apache.spark.sql.SparkSession@4785f
176,rate,List(),None,List(),None,Map(),None), rate, [timestamp#52, value#53L]

// Physical plan with StateStoreRestoreExec (as StateStoreRestore in the output)
scala> counts.explain
== Physical Plan ==
*Sort [value_count#65L ASC NULLS FIRST], true, 0
+- Exchange rangepartitioning(value_count#65L ASC NULLS FIRST, 200)
   +- *HashAggregate(keys=[window#66-T20000ms], functions=[count(value#53L)])
      +- StateStoreSave [window#66-T20000ms], StatefulOperatorStateInfo(<unknown>,c4a6
8192-b90b-40cc-b2c5-d996584eb0da,0,0), Append, 0
         +- *HashAggregate(keys=[window#66-T20000ms], functions=[merge_count(value#53L
)])
            +- StateStoreRestore [window#66-T20000ms], StatefulOperatorStateInfo(<unkn
own>,c4a68192-b90b-40cc-b2c5-d996584eb0da,0,0)
               +- *HashAggregate(keys=[window#66-T20000ms], functions=[merge_count(val
ue#53L)])
                  +- Exchange hashpartitioning(window#66-T20000ms, 200)
                     +- *HashAggregate(keys=[window#66-T20000ms], functions=[partial_c
ount(value#53L)])
                        +- *Project [named_struct(start, precisetimestampconversion(((
((CASE WHEN (cast(CEIL((cast((precisetimestampconversion(timestamp#52-T20000ms, Timest
ampType, LongType) - 0) as double) / 5000000.0)) as double) = (cast((precisetimestampc
onversion(timestamp#52-T20000ms, TimestampType, LongType) - 0) as double) / 5000000.0)
) THEN (CEIL((cast((precisetimestampconversion(timestamp#52-T20000ms, TimestampType, L
ongType) - 0) as double) / 5000000.0)) + 1) ELSE CEIL((cast((precisetimestampconversio
n(timestamp#52-T20000ms, TimestampType, LongType) - 0) as double) / 5000000.0)) END + 0
) - 1) * 5000000) + 0), LongType, TimestampType), end, precisetimestampconversion((((
```

```
CASE WHEN (cast(CEIL((cast((precisetimestampconversion(timestamp#52-T20000ms, Timestam
pType, LongType) - 0) as double) / 5000000.0)) as double) = (cast((precisetimestampcon
version(timestamp#52-T20000ms, TimestampType, LongType) - 0) as double) / 5000000.0))
THEN (CEIL((cast((precisetimestampconversion(timestamp#52-T20000ms, TimestampType, Lon
gType) - 0) as double) / 5000000.0)) + 1) ELSE CEIL((cast((precisetimestampconversion(
timestamp#52-T20000ms, TimestampType, LongType) - 0) as double) / 5000000.0)) END + 0)
 - 1) * 5000000) + 5000000), LongType, TimestampType)) AS window#66, value#53L]
                           +- *Filter isnotnull(timestamp#52-T20000ms)
                              +- EventTimeWatermark timestamp#52: timestamp, interval
20 seconds
                                 +- StreamingRelation rate, [timestamp#52, value#53L]
```

Table 1. StateStoreRestoreExec's SQLMetrics

Key	Name (in UI)	Description
`numOutputRows`	number of output rows	The number of input rows from the child physical operator (for which `StateStoreRestoreExec` tried to find the state)

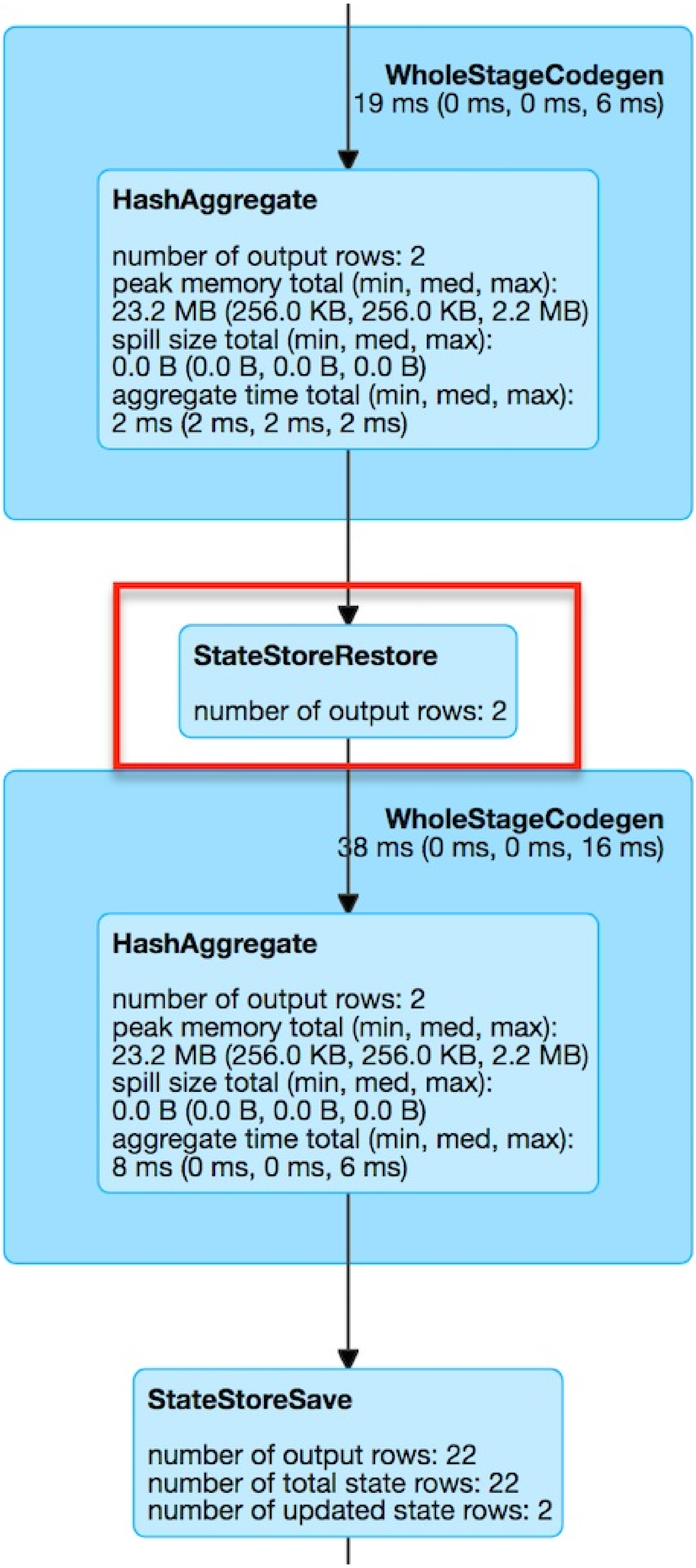
WholeStageCodegen
19 ms (0 ms, 0 ms, 6 ms)
HashAggregate
number of output rows: 2
peak memory total (min, med, max):
23.2 MB (256.0 KB, 256.0 KB, 2.2 MB)
spill size total (min, med, max):
0.0 B (0.0 B, 0.0 B, 0.0 B)
aggregate time total (min, med, max):
2 ms (2 ms, 2 ms, 2 ms)
StateStoreRestore
number of output rows: 2
WholeStageCodegen
38 ms (0 ms, 0 ms, 16 ms)
HashAggregate
number of output rows: 2
peak memory total (min, med, max):
23.2 MB (256.0 KB, 256.0 KB, 2.2 MB)
spill size total (min, med, max):
0.0 B (0.0 B, 0.0 B, 0.0 B)
aggregate time total (min, med, max):
8 ms (0 ms, 0 ms, 6 ms)
StateStoreSave
number of output rows: 22
number of total state rows: 22
number of updated state rows: 2

Figure 3. StateStoreRestoreExec in web UI (Details for Query)

When executed, `StateStoreRestoreExec` executes the child physical operator and creates a StateStoreRDD to map over partitions with `storeUpdateFunction` that restores the saved state for the keys in input rows if available.

The output schema of `StateStoreRestoreExec` is exactly the child's output schema.

The output partitioning of `StateStoreRestoreExec` is exactly the child's output partitioning.

Executing StateStoreRestoreExec — `doExecute` Method

```
doExecute(): RDD[InternalRow]
```

Note	`doExecute` is a part of `SparkPlan` contract to produce the result of a physical operator as an RDD of internal binary rows (i.e. `InternalRow`).

Internally, `doExecute` executes child physical operator and creates a StateStoreRDD with `storeUpdateFunction` that does the following per child operator's RDD partition:

1. Generates an unsafe projection to access the key field (using keyExpressions and the output schema of child operator).
2. For every input row (as `InternalRow`)
    - Extracts the key from the row (using the unsafe projection above)
    - Gets the saved state in `StateStore` for the key if available (it might not be if the key appeared in the input the first time)
    - Increments numOutputRows metric (that in the end is the number of rows from the child operator)
    - Generates collection made up of the current row and possibly the state for the key if available

Note	The number of rows from `StateStoreRestoreExec` is the number of rows from the child operator with additional rows for the saved state.

Note	There is no way in `StateStoreRestoreExec` to find out how many rows had associated state available in a state store. You would have to use the corresponding `StateStoreSaveExec` operator's metrics (most likely `number of total state rows` but that could depend on the output mode).

Creating StateStoreRestoreExec Instance

`StateStoreRestoreExec` takes the following when created:

- Catalyst expressions for keys (as used for aggregation in groupBy operator)
- Optional StatefulOperatorStateInfo
- Child physical plan (i.e. `SparkPlan`)

StateStoreSaveExec Unary Physical Operator — Saving State of Streaming Aggregates

`StateStoreSaveExec` is a unary physical operator (i.e. `UnaryExecNode`) that saves a streaming state to a state store with support for streaming watermark.

`StateStoreSaveExec` is created exclusively when `StatefulAggregationStrategy` plans streaming aggregate operators (aka *streaming aggregates*).

Figure 1. StateStoreSaveExec and StatefulAggregationStrategy

Note	`Aggregate` logical operator is the result of: • `RelationalGroupedDataset` aggregations, i.e. `agg` and `pivot` operators • `KeyValueGroupedDataset` aggregations, i.e. `mapGroups` , `flatMapGroups` , `mapGroupsWithState` , `flatMapGroupsWithState` , `reduceGroups` , and `agg` , `cogroup` operators • SQL's `GROUP BY` clause (possibly with `WITH CUBE` or `WITH ROLLUP`)

The optional properties, i.e. StatefulOperatorStateInfo, output mode, and event time watermark, are undefined when `StateStoreSaveExec` is created. `StateStoreSaveExec` is updated to hold their streaming batch-specific execution properties when `IncrementalExecution` prepares a streaming physical plan for execution (and state preparation rule is executed when `StreamExecution` plans a streaming query for a streaming batch).

Figure 2. StateStoreSaveExec and IncrementalExecution

Note	Unlike StateStoreRestoreExec operator, `StateStoreSaveExec` takes output mode and event time watermark when created.

When executed, `StateStoreSaveExec` creates a StateStoreRDD to map over partitions with `storeUpdateFunction` that manages the `StateStore` .

Figure 3. StateStoreSaveExec creates StateStoreRDD

```
scala> spark.version
res0: String = 2.3.0-SNAPSHOT

// START: Only for easier debugging
// The state is then only for one partition
// which should make monitoring it easier
import org.apache.spark.sql.internal.SQLConf.SHUFFLE_PARTITIONS
spark.sessionState.conf.setConf(SHUFFLE_PARTITIONS, 1)
scala> spark.sessionState.conf.numShufflePartitions
res2: Int = 1
// END: Only for easier debugging

val counts = spark.
  readStream.
  format("rate").
  load.
  groupBy(window($"timestamp", "5 seconds") as "group").
  agg(count("value") as "value_count") // <-- creates a Aggregate logical operator
scala> counts.explain(true)
== Parsed Logical Plan ==
'Aggregate [timewindow('timestamp, 5000000, 5000000, 0) AS window#5 AS group#6], [time
window('timestamp, 5000000, 5000000, 0) AS window#5 AS group#6, count('value) AS value
_count#12]
+- StreamingRelation DataSource(org.apache.spark.sql.SparkSession@489cbbcb,rate,List(),
None,List(),None,Map(),None), rate, [timestamp#0, value#1L]
...
== Physical Plan ==
*HashAggregate(keys=[window#18], functions=[count(value#1L)], output=[group#6, value_c
ount#12L])
+- StateStoreSave [window#18], StatefulOperatorStateInfo(<unknown>,9a6d381e-1066-4e2c-
abd2-27884a6c2d16,0,0), Append, 0
   +- *HashAggregate(keys=[window#18], functions=[merge_count(value#1L)], output=[wind
ow#18, count#20L])
      +- StateStoreRestore [window#18], StatefulOperatorStateInfo(<unknown>,9a6d381e-1
066-4e2c-abd2-27884a6c2d16,0,0)
         +- *HashAggregate(keys=[window#18], functions=[merge_count(value#1L)], output
=[window#18, count#20L])
            +- Exchange hashpartitioning(window#18, 1)
               +- *HashAggregate(keys=[window#18], functions=[partial_count(value#1L)]
, output=[window#18, count#20L])
                  +- *Project [named_struct(start, precisetimestampconversion(((((CASE
WHEN (cast(CEIL((cast((precisetimestampconversion(timestamp#0, TimestampType, LongType
) - 0) as double) / 5000000.0)) as double) = (cast((precisetimestampconversion(timesta
mp#0, TimestampType, LongType) - 0) as double) / 5000000.0)) THEN (CEIL((cast((precise
timestampconversion(timestamp#0, TimestampType, LongType) - 0) as double) / 5000000.0)
) + 1) ELSE CEIL((cast((precisetimestampconversion(timestamp#0, TimestampType, LongType
) - 0) as double) / 5000000.0)) END + 0) - 1) * 5000000) + 0), LongType, TimestampType
), end, precisetimestampconversion(((((CASE WHEN (cast(CEIL((cast((precisetimestampcon
version(timestamp#0, TimestampType, LongType) - 0) as double) / 5000000.0)) as double)
 = (cast((precisetimestampconversion(timestamp#0, TimestampType, LongType) - 0) as dou
ble) / 5000000.0)) THEN (CEIL((cast((precisetimestampconversion(timestamp#0, Timestamp
```

```
Type, LongType) - 0) as double) / 5000000.0)) + 1) ELSE CEIL((cast((precisetimestampco
nversion(timestamp#0, TimestampType, LongType) - 0) as double) / 5000000.0)) END + 0)
- 1) * 5000000) + 5000000), LongType, TimestampType)) AS window#18, value#1L]
                     +- *Filter isnotnull(timestamp#0)
                        +- StreamingRelation rate, [timestamp#0, value#1L]

// Start the query and hence execute StateStoreSaveExec
import scala.concurrent.duration._
import org.apache.spark.sql.streaming.{OutputMode, Trigger}
val sq = counts.
  writeStream.
  format("console").
  option("truncate", false).
  trigger(Trigger.ProcessingTime(1.hour)). // <-- should be enough time for exploration

  outputMode(OutputMode.Complete).
  start

// wait till the first batch which should happen right after start

import org.apache.spark.sql.execution.streaming._
val streamingBatch = sq.asInstanceOf[StreamingQueryWrapper].streamingQuery.lastExecuti
on

scala> println(streamingBatch.logical.numberedTreeString)
00 Aggregate [window#13], [window#13 AS group#6, count(value#25L) AS value_count#12L]
01 +- Filter isnotnull(timestamp#24)
02    +- Project [named_struct(start, precisetimestampconversion(((((CASE WHEN (cast(C
EIL((cast((precisetimestampconversion(timestamp#24, TimestampType, LongType) - 0) as d
ouble) / cast(5000000 as double))) as double) = (cast((precisetimestampconversion(time
stamp#24, TimestampType, LongType) - 0) as double) / cast(5000000 as double))) THEN (C
EIL((cast((precisetimestampconversion(timestamp#24, TimestampType, LongType) - 0) as d
ouble) / cast(5000000 as double))) + cast(1 as bigint)) ELSE CEIL((cast((precisetimest
ampconversion(timestamp#24, TimestampType, LongType) - 0) as double) / cast(5000000 as
 double))) END + cast(0 as bigint)) - cast(1 as bigint)) * 5000000) + 0), LongType, Ti
mestampType), end, precisetimestampconversion((((((CASE WHEN (cast(CEIL((cast((precise
timestampconversion(timestamp#24, TimestampType, LongType) - 0) as double) / cast(5000
000 as double))) as double) = (cast((precisetimestampconversion(timestamp#24, Timestam
pType, LongType) - 0) as double) / cast(5000000 as double))) THEN (CEIL((cast((precise
timestampconversion(timestamp#24, TimestampType, LongType) - 0) as double) / cast(5000
000 as double))) + cast(1 as bigint)) ELSE CEIL((cast((precisetimestampconversion(time
stamp#24, TimestampType, LongType) - 0) as double) / cast(5000000 as double))) END + c
ast(0 as bigint)) - cast(1 as bigint)) * 5000000) + 0) + 5000000), LongType, Timestamp
Type)) AS window#13, timestamp#24, value#25L]
03       +- LogicalRDD [timestamp#24, value#25L], true

// Note the number of partitions
// 200 is the default, but we have changed it above
scala> println(streamingBatch.toRdd.toDebugString)
(1) MapPartitionsRDD[20] at toRdd at <console>:38 []
 |  StateStoreRDD[19] at toRdd at <console>:38 []
 |  MapPartitionsRDD[18] at toRdd at <console>:38 []
 |  StateStoreRDD[17] at toRdd at <console>:38 []
```

```
   |  MapPartitionsRDD[16] at toRdd at <console>:38 []
   |  ShuffledRowRDD[4] at start at <console>:36 []
   +-(0) MapPartitionsRDD[3] at start at <console>:36 []
      |  MapPartitionsRDD[2] at start at <console>:36 []
      |  MapPartitionsRDD[1] at start at <console>:36 []
      |  EmptyRDD[0] at start at <console>:36 []

scala> spark.sessionState.conf.numShufflePartitions
res6: Int = 1
```

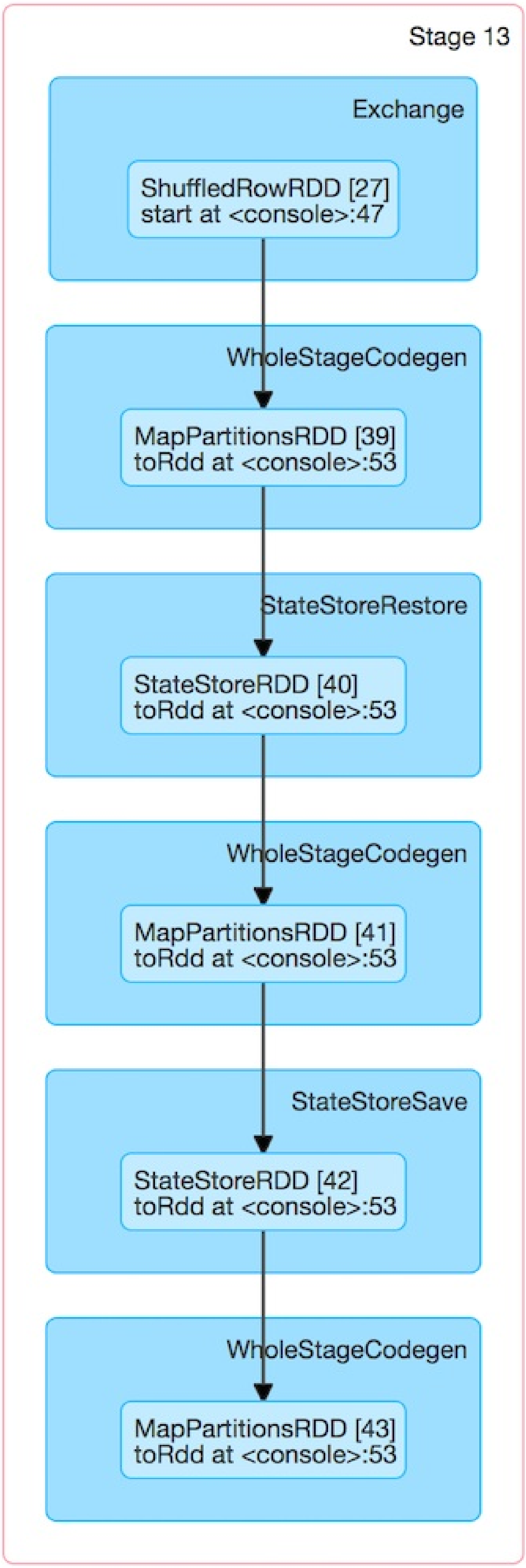
Stage 13
Exchange
ShuffledRowRDD [27]
start at <console>:47
WholeStageCodegen
MapPartitionsRDD [39]
toRdd at <console>:53
StateStoreRestore
StateStoreRDD [40]
toRdd at <console>:53
WholeStageCodegen
MapPartitionsRDD [41]
toRdd at <console>:53
StateStoreSave
StateStoreRDD [42]
toRdd at <console>:53
WholeStageCodegen
MapPartitionsRDD [43]
toRdd at <console>:53

Figure 4. StateStoreSaveExec and StateStoreRDD (after streamingBatch.toRdd.count)

Note	The number of partitions of StateStoreRDD (and hence the number of Spark tasks) is what was defined for the child physical plan. There will be that many `StateStores` as there are partitions in `StateStoreRDD` .

Note	`StateStoreSaveExec` behaves differently per output mode.

Table 1. StateStoreSaveExec's SQLMetrics

Key	Name (in UI)	Description
`allUpdatesTimeMs`	total time to update rows	
`allRemovalsTimeMs`	total time to remove rows	
`commitTimeMs`	time to commit changes	
`numOutputRows`	number of output rows	
`numTotalStateRows`	number of total state rows	Number of the state keys in the state store Corresponds to `numRowsTotal` in `stateOperators` in StreamingQueryProgress (and is available as `sq.lastProgress.stateOperators(0).numRowsTotal` for
`numUpdatedStateRows`	number of updated	Number of the state keys that were stored as updates trigger and for the keys in the result rows of the upstr • In `Complete` output mode, `numUpdatedStateRows` rows (which should be exactly the number of out upstream operator) Caution: FIXME • In `Append` output mode, `numUpdatedStateRows` is rows with keys that have not expired yet (per req

	state rows	• In `Update` output mode, `numUpdatedStateRows` is output rows, i.e. the number of keys that have nc watermark has been defined at all (which is optic Caution: FIXME Note: You can see the current value as `numRowsU` `stateOperators` in StreamingQueryProgre as `StreamingQuery.lastProgress.stateOperator` for `n` th operator).
`stateMemory`	memory used by state	Memory used by the StateStore

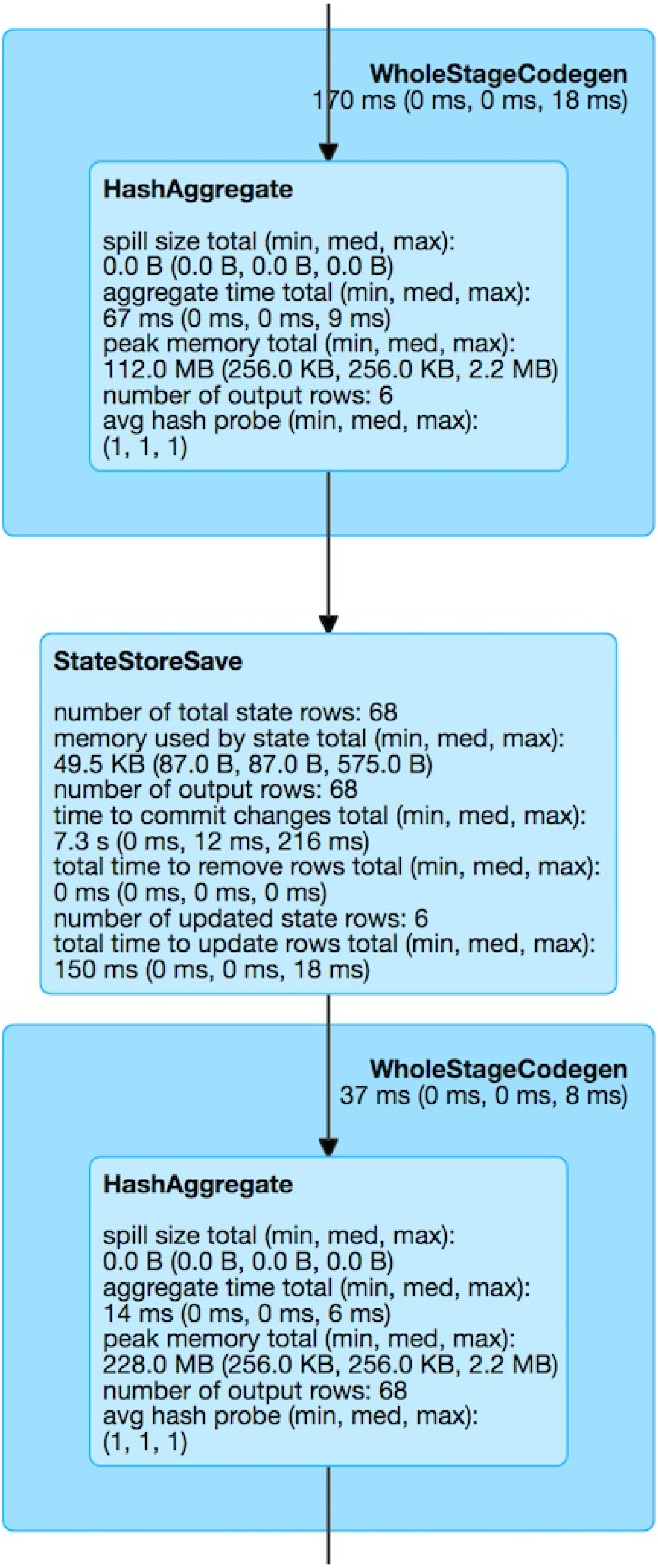
WholeStageCodegen
170 ms (0 ms, 0 ms, 18 ms)
HashAggregate
spill size total (min, med, max):
0.0 B (0.0 B, 0.0 B, 0.0 B)
aggregate time total (min, med, max):
67 ms (0 ms, 0 ms, 9 ms)
peak memory total (min, med, max):
112.0 MB (256.0 KB, 256.0 KB, 2.2 MB)
number of output rows: 6
avg hash probe (min, med, max):
(1, 1, 1)
StateStoreSave
number of total state rows: 68
memory used by state total (min, med, max):
49.5 KB (87.0 B, 87.0 B, 575.0 B)
number of output rows: 68
time to commit changes total (min, med, max):
7.3 s (0 ms, 12 ms, 216 ms)
total time to remove rows total (min, med, max):
0 ms (0 ms, 0 ms, 0 ms)
number of updated state rows: 6
total time to update rows total (min, med, max):
150 ms (0 ms, 0 ms, 18 ms)
WholeStageCodegen
37 ms (0 ms, 0 ms, 8 ms)
HashAggregate
spill size total (min, med, max):
0.0 B (0.0 B, 0.0 B, 0.0 B)
aggregate time total (min, med, max):
14 ms (0 ms, 0 ms, 6 ms)
peak memory total (min, med, max):
228.0 MB (256.0 KB, 256.0 KB, 2.2 MB)
number of output rows: 68
avg hash probe (min, med, max):
(1, 1, 1)

Figure 5. StateStoreSaveExec in web UI (Details for Query)

When executed, `StateStoreSaveExec` executes the child physical operator and creates a StateStoreRDD (with `storeUpdateFunction` specific to the output mode).

The output schema of `StateStoreSaveExec` is exactly the child's output schema.

The output partitioning of `StateStoreSaveExec` is exactly the child's output partitioning.

<table>
<tr><td>Tip</td><td>
Enable <code>INFO</code> logging level for <code>org.apache.spark.sql.execution.streaming.StateStoreSaveExec</code> to see what happens inside.
Add the following line to <code>conf/log4j.properties</code>:
<pre>log4j.logger.org.apache.spark.sql.execution.streaming.StateStoreSaveExec=INFO</pre>
Refer to Logging.
</td></tr>
</table>

Executing StateStoreSaveExec — `doExecute` Method

```
doExecute(): RDD[InternalRow]
```

Note	`doExecute` is a part of `SparkPlan` contract to produce the result of a physical operator as an RDD of internal binary rows (i.e. `InternalRow`).

Internally, `doExecute` initializes metrics.

Note	`doExecute` requires that the optional outputMode is at this point defined (that should have happened when `IncrementalExecution` had prepared a streaming aggregation for execution).

`doExecute` executes child physical operator and creates a StateStoreRDD with `storeUpdateFunction` that:

1. Generates an unsafe projection to access the key field (using keyExpressions and the output schema of child).
2. Branches off per output mode.

Table 2. doExecute's Behaviour per Output Mode

Output Mode	doExecute's Behaviour
	Note: `Append` is the default output mode when unspecified.

Note	`Append` output mode requires that a streaming query defines event time watermark (using withWatermark operator) on the event time column that is used in aggregation (directly or using window function).

1. Finds late (aggregate) rows from child physical operator (that have expired per watermark)
2. Stores the late rows in the state store (and increments numUpdatedStateRows metric)
3. Gets all the added (late) rows from the state store
4. Creates an iterator that removes the late rows from the state store when requested the next row and in the end commits the state updates

Note	numUpdatedStateRows metric is the number of rows that…FIXME

Tip	Refer to Demo: StateStoreSaveExec with Append Output Mode for an example of `StateStoreSaveExec` in `Append` output mode.

`Append`

Caution	FIXME When is "Filtering state store on:" printed out?

Caution	FIXME Track numUpdatedStateRows metric

---

1. Uses watermarkPredicateForData predicate to exclude matching rows and (like in Complete output mode) stores all the remaining rows in `StateStore`.
2. (like in Complete output mode) While storing the rows, increments numUpdatedStateRows metric (for every row) and records the total time in allUpdatesTimeMs metric.
3. Takes all the rows from `StateStore` and returns a `NextIterator` that:
   - In `getNext`, finds the first row that matches watermarkPredicateForKeys predicate, removes it from `StateStore`, and returns it back.

     If no row was found, `getNext` also marks the iterator as finished.

<table>
<tr>
<td></td>
<td>
In <code>close</code>, records the time to iterate over all the rows in allRemovalsTimeMs metric, commits the updates to <code>StateStore</code> followed by recording the time in commitTimeMs metric and recording StateStore metrics.
</td>
</tr>
<tr>
<td>Complete</td>
<td>

Takes all <code>UnsafeRow</code> rows (from the parent iterator)
Stores the rows by key in the state store eagerly (i.e. all rows that are available in the parent iterator before proceeding)
Commits the state updates
In the end, <code>doExecute</code> reads the key-row pairs from the state store and passes the rows along (i.e. to the following physical operator)

The number of keys stored in the state store is recorded in numUpdatedStateRows metric.
<table><tr><td>Note</td><td>In <code>Complete</code> output mode numOutputRows metric is exactly numTotalStateRows metric.</td></tr></table>
<table><tr><td>Tip</td><td>Refer to Demo: StateStoreSaveExec with Complete Output Mode for an example of <code>StateStoreSaveExec</code> in <code>Complete</code> output mode.</td></tr></table>
<hr>

Stores all rows (as <code>UnsafeRow</code>) in <code>StateStore</code>.
While storing the rows, increments numUpdatedStateRows metric (for every row) and records the total time in allUpdatesTimeMs metric.
Records <code>0</code> in allRemovalsTimeMs metric.
Commits the state updates to <code>StateStore</code> and records the time in commitTimeMs metric.
Records StateStore metrics.
In the end, takes all the rows stored in <code>StateStore</code> and increments numOutputRows metric.

</td>
</tr>
<tr>
<td></td>
<td>Returns an iterator that filters out late aggregate rows (per watermark if defined) and stores the "young" rows in the state store (one by one, i.e. every <code>next</code>). With no more rows available, that removes the late rows from the state store (all at once) and commits the state updates.</td>
</tr>
</table>

<table>
<tr><td></td><td>
<table><tr><td>Tip</td><td>Refer to Demo: StateStoreSaveExec with Update Output Mode for an example of <code>StateStoreSaveExec</code> in <code>Update</code> output mode.</td></tr></table>
</td></tr>
<tr><td><code>Update</code></td><td>
Returns <code>Iterator</code> of rows that uses watermarkPredicateForData predicate to filter out late rows.
In <code>hasNext</code> , when rows are no longer available:

Records the total time to iterate over all the rows in allUpdatesTimeMs metric.
removeKeysOlderThanWatermark and records the time in allRemovalsTimeMs metric.
Commits the updates to <code>StateStore</code> and records the time in commitTimeMs metric.
Records StateStore metrics.

In <code>next</code> , stores a row in <code>StateStore</code> and increments numOutputRows and numUpdatedStateRows metrics.
</td></tr>
</table>

`doExecute` reports a `UnsupportedOperationException` when executed with an invalid output mode.

```
Invalid output mode: [outputMode]
```

Creating StateStoreSaveExec Instance

`StateStoreSaveExec` takes the following when created:

- Catalyst expressions for keys (as used for aggregation in groupBy operator)
- Optional StatefulOperatorStateInfo
- Output mode
- Event time watermark (as `long` number)
- Child physical plan (i.e. `SparkPlan`)

Demo: StateStoreSaveExec with Complete Output Mode

The following example code shows the behaviour of StateStoreSaveExec in Complete output mode.

```
// START: Only for easier debugging
// The state is then only for one partition
// which should make monitoring it easier
import org.apache.spark.sql.internal.SQLConf.SHUFFLE_PARTITIONS
spark.sessionState.conf.setConf(SHUFFLE_PARTITIONS, 1)
scala> spark.sessionState.conf.numShufflePartitions
res1: Int = 1
// END: Only for easier debugging

// Read datasets from a Kafka topic
// ./bin/spark-shell --packages org.apache.spark:spark-sql-kafka-0-10_2.11:2.3.0-SNAPS
HOT
// Streaming aggregation using groupBy operator is required to have StateStoreSaveExec
 operator
val valuesPerGroup = spark.
  readStream.
  format("kafka").
  option("subscribe", "topic1").
  option("kafka.bootstrap.servers", "localhost:9092").
  load.
  withColumn("tokens", split('value, ",")).
  withColumn("group", 'tokens(0)).
  withColumn("value", 'tokens(1) cast "int").
  select("group", "value").
  groupBy($"group").
  agg(collect_list("value") as "values").
  orderBy($"group".asc)

// valuesPerGroup is a streaming Dataset with just one source
// so it knows nothing about output mode or watermark yet
// That's why StatefulOperatorStateInfo is generic
// and no batch-specific values are printed out
// That will be available after the first streaming batch
// Use sq.explain to know the runtime-specific values
scala> valuesPerGroup.explain
== Physical Plan ==
*Sort [group#25 ASC NULLS FIRST], true, 0
+- Exchange rangepartitioning(group#25 ASC NULLS FIRST, 1)
   +- ObjectHashAggregate(keys=[group#25], functions=[collect_list(value#36, 0, 0)])
      +- Exchange hashpartitioning(group#25, 1)
         +- StateStoreSave [group#25], StatefulOperatorStateInfo(<unknown>,899f0fd1-b2
02-45cd-9ebd-09101ca90fa8,0,0), Append, 0
```

```
            +- ObjectHashAggregate(keys=[group#25], functions=[merge_collect_list(valu
e#36, 0, 0)])
               +- Exchange hashpartitioning(group#25, 1)
                  +- StateStoreRestore [group#25], StatefulOperatorStateInfo(<unknown>,
899f0fd1-b202-45cd-9ebd-09101ca90fa8,0,0)
                     +- ObjectHashAggregate(keys=[group#25], functions=[merge_collect_
list(value#36, 0, 0)])
                        +- Exchange hashpartitioning(group#25, 1)
                           +- ObjectHashAggregate(keys=[group#25], functions=[partial_
collect_list(value#36, 0, 0)])
                              +- *Project [split(cast(value#1 as string), ,)[0] AS gro
up#25, cast(split(cast(value#1 as string), ,)[1] as int) AS value#36]
                                 +- StreamingRelation kafka, [key#0, value#1, topic#2,
 partition#3, offset#4L, timestamp#5, timestampType#6]

// Start the query and hence StateStoreSaveExec
// Use Complete output mode
import scala.concurrent.duration._
import org.apache.spark.sql.streaming.{OutputMode, Trigger}
val sq = valuesPerGroup.
  writeStream.
  format("console").
  option("truncate", false).
  trigger(Trigger.ProcessingTime(10.seconds)).
  outputMode(OutputMode.Complete).
  start

-------------------------------------------
Batch: 0
-------------------------------------------
+-----+------+
|group|values|
+-----+------+
+-----+------+

// there's only 1 stateful operator and hence 0 for the index in stateOperators
scala> println(sq.lastProgress.stateOperators(0).prettyJson)
{
  "numRowsTotal" : 0,
  "numRowsUpdated" : 0,
  "memoryUsedBytes" : 60
}

// publish 1 new key-value pair in a single streaming batch
// 0,1

-------------------------------------------
Batch: 1
-------------------------------------------
+-----+------+
|group|values|
+-----+------+
|0    |[1]   |
```

```
+-----+------+

// it's Complete output mode so numRowsTotal is the number of keys in the state store
// no keys were available earlier (it's just started!) and so numRowsUpdated is 0
scala> println(sq.lastProgress.stateOperators(0).prettyJson)
{
  "numRowsTotal" : 1,
  "numRowsUpdated" : 0,
  "memoryUsedBytes" : 324
}

// publish new key and old key in a single streaming batch
// new keys
// 1,1
// updates to already-stored keys
// 0,2

-------------------------------------------
Batch: 2
-------------------------------------------
+-----+------+
|group|values|
+-----+------+
|0    |[2, 1]|
|1    |[1]   |
+-----+------+

// it's Complete output mode so numRowsTotal is the number of keys in the state store
// no keys were available earlier and so numRowsUpdated is...0?!
// Think it's a BUG as it should've been 1 (for the row 0,2)
// 8/30 Sent out a question to the Spark user mailing list
scala> println(sq.lastProgress.stateOperators(0).prettyJson)
{
  "numRowsTotal" : 2,
  "numRowsUpdated" : 0,
  "memoryUsedBytes" : 572
}

// In the end...
sq.stop
```

Demo: StateStoreSaveExec with Update Output Mode

Caution	FIXME Example of Update with StateStoreSaveExec (and optional watermark)

StreamingDeduplicateExec Unary Physical Operator for Streaming Deduplication

`StreamingDeduplicateExec` is a unary physical operator (i.e. `UnaryExecNode`) that writes state to StateStore with support for streaming watermark.

`StreamingDeduplicateExec` is created exclusively when `StreamingDeduplicationStrategy` plans Deduplicate unary logical operators.

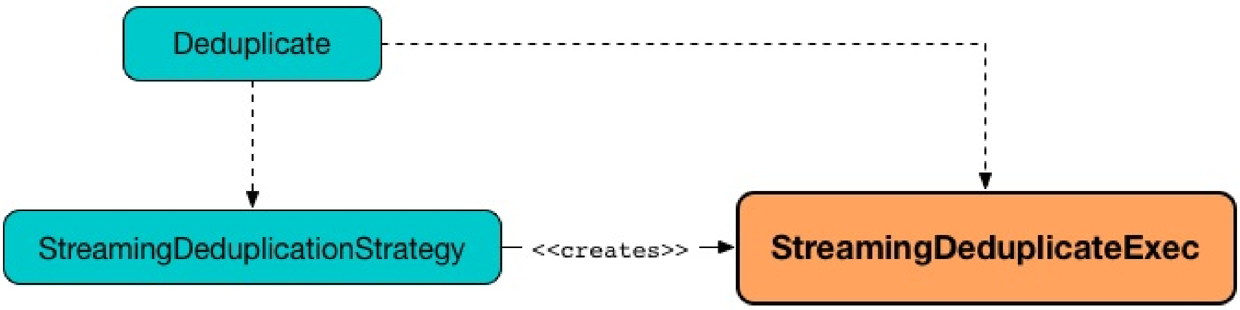

Figure 1. StreamingDeduplicateExec and StreamingDeduplicationStrategy

```
val uniqueValues = spark.
  readStream.
  format("rate").
  load.
  dropDuplicates("value")  // <-- creates Deduplicate logical operator

scala> println(uniqueValues.queryExecution.logical.numberedTreeString)
00 Deduplicate [value#214L], true
01 +- StreamingRelation DataSource(org.apache.spark.sql.SparkSession@4785f176,rate,List
(),None,List(),None,Map(),None), rate, [timestamp#213, value#214L]

scala> uniqueValues.explain
== Physical Plan ==
StreamingDeduplicate [value#214L], StatefulOperatorStateInfo(<unknown>,5a65879c-67bc-4
e77-b417-6100db6a52a2,0,0), 0
+- Exchange hashpartitioning(value#214L, 200)
   +- StreamingRelation rate, [timestamp#213, value#214L]

// Start the query and hence StreamingDeduplicateExec
import scala.concurrent.duration._
import org.apache.spark.sql.streaming.{OutputMode, Trigger}
val sq = uniqueValues.
  writeStream.
  format("console").
  option("truncate", false).
  trigger(Trigger.ProcessingTime(10.seconds)).
  outputMode(OutputMode.Update).
  start
```

```
// sorting not supported for non-aggregate queries
// and so values are unsorted

-------------------------------------------
Batch: 0
-------------------------------------------
+---------+-----+
|timestamp|value|
+---------+-----+
+---------+-----+

-------------------------------------------
Batch: 1
-------------------------------------------
+-----------------------+-----+
|timestamp              |value|
+-----------------------+-----+
|2017-07-25 22:12:03.018|0    |
|2017-07-25 22:12:08.018|5    |
|2017-07-25 22:12:04.018|1    |
|2017-07-25 22:12:06.018|3    |
|2017-07-25 22:12:05.018|2    |
|2017-07-25 22:12:07.018|4    |
+-----------------------+-----+

-------------------------------------------
Batch: 2
-------------------------------------------
+-----------------------+-----+
|timestamp              |value|
+-----------------------+-----+
|2017-07-25 22:12:10.018|7    |
|2017-07-25 22:12:09.018|6    |
|2017-07-25 22:12:12.018|9    |
|2017-07-25 22:12:13.018|10   |
|2017-07-25 22:12:15.018|12   |
|2017-07-25 22:12:11.018|8    |
|2017-07-25 22:12:14.018|11   |
|2017-07-25 22:12:16.018|13   |
|2017-07-25 22:12:17.018|14   |
|2017-07-25 22:12:18.018|15   |
+-----------------------+-----+

// Eventually...
sq.stop
```

Table 1. StreamingDeduplicateExec's SQLMetrics

Name	Description
`allUpdatesTimeMs`	
`allRemovalsTimeMs`	
`commitTimeMs`	
`numTotalStateRows`	Number of keys in the StateStore
`numOutputRows`	
`numTotalStateRows`	Number of keys in the StateStore
`numUpdatedStateRows`	
`stateMemory`	Memory used by the StateStore

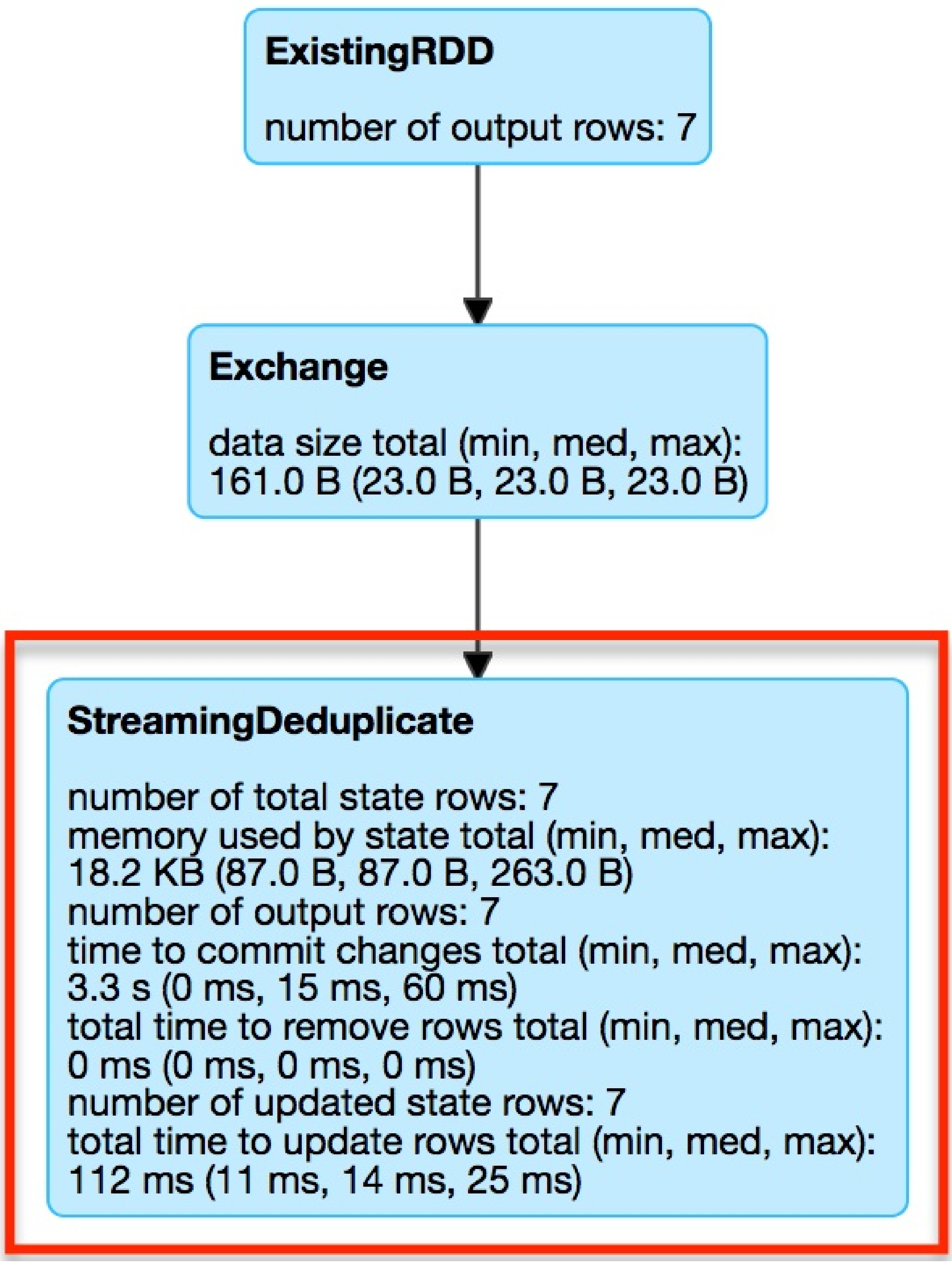

Figure 2. StreamingDeduplicateExec in web UI (Details for Query)

The output schema of `StreamingDeduplicateExec` is exactly the child's output schema.

The output partitioning of `StreamingDeduplicateExec` is exactly the child's output partitioning.

```
/**
// Start spark-shell with debugging and Kafka support
  SPARK_SUBMIT_OPTS="-agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=500
5" \
  ./bin/spark-shell \
```

```
  --packages org.apache.spark:spark-sql-kafka-0-10_2.11:2.3.0-SNAPSHOT
*/
// Reading
val topic1 = spark.
  readStream.
  format("kafka").
  option("subscribe", "topic1").
  option("kafka.bootstrap.servers", "localhost:9092").
  option("startingoffsets", "earliest").
  load

// Processing with deduplication
// Don't use watermark
// The following won't work due to https://issues.apache.org/jira/browse/SPARK-21546
/**
val records = topic1.
  withColumn("eventtime", 'timestamp).  // <-- just to put the right name given the pu
rpose
  withWatermark(eventTime = "eventtime", delayThreshold = "30 seconds"). // <-- use th
e renamed eventtime column
  dropDuplicates("value").  // dropDuplicates will use watermark
                            // only when eventTime column exists
  // include the watermark column => internal design leak?
  select('key cast "string", 'value cast "string", 'eventtime).
  as[(String, String, java.sql.Timestamp)]
*/

val records = topic1.
  dropDuplicates("value").
  select('key cast "string", 'value cast "string").
  as[(String, String)]

scala> records.explain
== Physical Plan ==
*Project [cast(key#0 as string) AS key#249, cast(value#1 as string) AS value#250]
+- StreamingDeduplicate [value#1], StatefulOperatorStateInfo(<unknown>,68198b93-6184-49
ae-8098-006c32cc6192,0,0), 0
   +- Exchange hashpartitioning(value#1, 200)
      +- *Project [key#0, value#1]
         +- StreamingRelation kafka, [key#0, value#1, topic#2, partition#3, offset#4L,
 timestamp#5, timestampType#6]

// Writing
import org.apache.spark.sql.streaming.{OutputMode, Trigger}
import scala.concurrent.duration._
val sq = records.
  writeStream.
  format("console").
  option("truncate", false).
  trigger(Trigger.ProcessingTime(10.seconds)).
  queryName("from-kafka-topic1-to-console").
  outputMode(OutputMode.Update).
  start
```

```
// Eventually...
sq.stop
```

Tip	Enable `INFO` logging level for `org.apache.spark.sql.execution.streaming.StreamingDeduplicateExec` to see what happens inside. Add the following line to `conf/log4j.properties`: `log4j.logger.org.apache.spark.sql.execution.streaming.StreamingDeduplicateExec=I` Refer to Logging.

Executing StreamingDeduplicateExec — `doExecute` Method

```
doExecute(): RDD[InternalRow]
```

Note	`doExecute` is a part of `SparkPlan` contract to produce the result of a physical operator as an RDD of internal binary rows (i.e. `InternalRow`).

Internally, `doExecute` initializes metrics.

`doExecute` executes child physical operator and creates a StateStoreRDD with `storeUpdateFunction` that:

1. Generates an unsafe projection to access the key field (using keyExpressions and the output schema of child).
2. Filters out rows from `Iterator[InternalRow]` that match `watermarkPredicateForData` (when defined and timeoutConf is `EventTimeTimeout`)
3. For every row (as `InternalRow`)
   - Extracts the key from the row (using the unsafe projection above)
   - Gets the saved state in `StateStore` for the key
   - (when there was a state for the key in the row) Filters out (aka *drops*) the row
   - (when there was *no* state for the key in the row) Stores a new (and empty) state for the key and increments numUpdatedStateRows and numOutputRows metrics.

4. In the end, `storeUpdateFunction` creates a `CompletionIterator` that executes a completion function (aka `completionFunction`) after it has successfully iterated through all the elements (i.e. when a client has consumed all the rows).

   The completion function does the following:

   - Updates allUpdatesTimeMs metric (that is the total time to execute `storeUpdateFunction`)
   - Updates allRemovalsTimeMs metric with the time taken to remove keys older than the watermark from the StateStore
   - Updates commitTimeMs metric with the time taken to commit the changes to the StateStore
   - Sets StateStore-specific metrics

Creating StreamingDeduplicateExec Instance

`StreamingDeduplicateExec` takes the following when created:

- Duplicate keys (as used in dropDuplicates operator)
- Child physical plan (i.e. `SparkPlan`)
- Optional StatefulOperatorStateInfo
- Optional event time watermark

StreamingRelationExec Leaf Physical Operator

`StreamingRelationExec` is a leaf physical operator (i.e. `LeafExecNode`) that…FIXME

`StreamingRelationExec` is created when `StreamingRelationStrategy` plans `StreamingRelation` and `StreamingExecutionRelation` logical operators.

```
scala> spark.version
res0: String = 2.3.0-SNAPSHOT

val rates = spark.
  readStream.
  format("rate").
  load

// StreamingRelation logical operator
scala> println(rates.queryExecution.logical.numberedTreeString)
00 StreamingRelation DataSource(org.apache.spark.sql.SparkSession@31ba0af0,rate,List(),
None,List(),None,Map(),None), rate, [timestamp#0, value#1L]

// StreamingRelationExec physical operator (shown without "Exec" suffix)
scala> rates.explain
== Physical Plan ==
StreamingRelation rate, [timestamp#0, value#1L]
```

`StreamingRelationExec` is not supposed to be executed and is used…FIXME

Creating StreamingRelationExec Instance

`StreamingRelationExec` takes the following when created:

- The name of a streaming data source
- Output attributes

StreamingSymmetricHashJoinExec

`StreamingSymmetricHashJoinExec` is…FIXME

Executing StreamingSymmetricHashJoinExec — `doExecute` Method

```
doExecute(): RDD[InternalRow]
```

`doExecute` …FIXME

Note	`doExecute` is used when…FIXME

WatermarkSupport Contract for Streaming Watermark in Unary Physical Operators

`WatermarkSupport` is the contract for unary physical operators (i.e. `UnaryExecNode`) with streaming watermark support.

Note	**Watermark** (aka **"allowed lateness"**) is a moving threshold of **event time** and specifies what data to consider for aggregations, i.e. the threshold of late data so the engine can automatically drop incoming late data given event time and clean up old state accordingly. Read the official documentation of Spark in Handling Late Data and Watermarking.

Table 1. WatermarkSupport's (Lazily-Initialized) Properties

<table>
<tr><th>Property</th><th>Description</th></tr>
<tr><td><code>watermarkExpression</code></td><td>Optional Catalyst expression that matches rows older than t watermark.

Note: Use withWatermark operator to specify streamin

When initialized, <code>watermarkExpression</code> finds spark.waterma watermark attribute in the child output's metadata.

If found, <code>watermarkExpression</code> creates <code>evictionExpression</code> attribute that is less than or equal eventTimeWatermark.

The watermark attribute may be of type <code>StructType</code>. If it is, <code>watermarkExpression</code> uses the first field as the watermark.

<code>watermarkExpression</code> prints out the following INFO messag spark.watermarkDelayMs watermark attribute is found.

<code>INFO [physicalOperator]Exec: Filtering state store on: [</code>

Note: <code>physicalOperator</code> can be FlatMapGroupsWithSt StateStoreSaveExec or StreamingDeduplicateEx

Tip: Enable INFO logging level for one of the stateful pl see the INFO message in the logs.</td></tr>
<tr><td><code>watermarkPredicateForData</code></td><td>Optional <code>Predicate</code> that uses watermarkExpression and th match rows older than the watermark.</td></tr>
<tr><td><code>watermarkPredicateForKeys</code></td><td>Optional <code>Predicate</code> that uses keyExpressions to match row event time watermark.</td></tr>
</table>

WatermarkSupport Contract

```
package org.apache.spark.sql.execution.streaming

trait WatermarkSupport extends UnaryExecNode {
  // only required methods that have no implementation
  def eventTimeWatermark: Option[Long]
  def keyExpressions: Seq[Attribute]
}
```

Table 2. WatermarkSupport Contract

Method	Description
`eventTimeWatermark`	Used mainly in watermarkExpression to create a `LessThanOrEqual` Catalyst binary expression that matches rows older than the watermark.
`keyExpressions`	Grouping keys (in FlatMapGroupsWithStateExec), duplicate keys (in StreamingDeduplicateExec) or key attributes (in StateStoreSaveExec) with at most one that may have spark.watermarkDelayMs watermark attribute in metadata Used in watermarkPredicateForKeys to create a `Predicate` to match rows older than the event time watermark. Used also when StateStoreSaveExec and StreamingDeduplicateExec physical operators are executed.

Removing Keys Older Than Watermark From StateStore — `removeKeysOlderThanWatermark` Internal Method

```
removeKeysOlderThanWatermark(store: StateStore): Unit
```

`removeKeysOlderThanWatermark` requests the input `store` for all rows.

`removeKeysOlderThanWatermark` then uses watermarkPredicateForKeys to remove matching rows from the store.

Note	`removeKeysOlderThanWatermark` is used when: • `StateStoreSaveExec` is executed (for Update output mode only) • `StreamingDeduplicateExec` is executed

FlatMapGroupsWithStateStrategy Execution Planning Strategy for FlatMapGroupsWithState Logical Operator

`FlatMapGroupsWithStateStrategy` is an execution planning strategy (i.e. `Strategy`) that IncrementalExecution uses to plan `FlatMapGroupsWithState` logical operators.

`FlatMapGroupsWithStateStrategy` resolves FlatMapGroupsWithState unary logical operator to FlatMapGroupsWithStateExec physical operator (with undefined `StatefulOperatorStateInfo` , `batchTimestampMs` , and `eventTimeWatermark`).

```
import org.apache.spark.sql.streaming.GroupState
val stateFunc = (key: Long, values: Iterator[(Timestamp, Long)], state: GroupState[Long
]) => {
  Iterator((key, values.size))
}
import java.sql.Timestamp
import org.apache.spark.sql.streaming.{GroupStateTimeout, OutputMode}
val numGroups = spark.
  readStream.
  format("rate").
  load.
  as[(Timestamp, Long)].
  groupByKey { case (time, value) => value % 2 }.
  flatMapGroupsWithState(OutputMode.Update, GroupStateTimeout.NoTimeout)(stateFunc)

scala> numGroups.explain(true)
== Parsed Logical Plan ==
'SerializeFromObject [assertnotnull(assertnotnull(input[0, scala.Tuple2, true]))._1 AS
 _1#267L, assertnotnull(assertnotnull(input[0, scala.Tuple2, true]))._2 AS _2#268]
+- 'FlatMapGroupsWithState <function3>, unresolveddeserializer(upcast(getcolumnbyordin
al(0, LongType), LongType, - root class: "scala.Long"), value#262L), unresolveddeseria
lizer(newInstance(class scala.Tuple2), timestamp#253, value#254L), [value#262L], [time
stamp#253, value#254L], obj#266: scala.Tuple2, class[value[0]: bigint], Update, false,
NoTimeout
   +- AppendColumns <function1>, class scala.Tuple2, [StructField(_1,TimestampType,tru
e), StructField(_2,LongType,false)], newInstance(class scala.Tuple2), [input[0, bigint
, false] AS value#262L]
      +- StreamingRelation DataSource(org.apache.spark.sql.SparkSession@38bcac50,rate,
List(),None,List(),None,Map(),None), rate, [timestamp#253, value#254L]

...

== Physical Plan ==
*SerializeFromObject [assertnotnull(input[0, scala.Tuple2, true])._1 AS _1#267L, asser
tnotnull(input[0, scala.Tuple2, true])._2 AS _2#268]
+- FlatMapGroupsWithState <function3>, value#262: bigint, newInstance(class scala.Tupl
e2), [value#262L], [timestamp#253, value#254L], obj#266: scala.Tuple2, StatefulOperato
rStateInfo(<unknown>,84b5dccb-3fa6-4343-a99c-6fa5490c9b33,0,0), class[value[0]: bigint
], Update, NoTimeout, 0, 0
   +- *Sort [value#262L ASC NULLS FIRST], false, 0
      +- Exchange hashpartitioning(value#262L, 200)
         +- AppendColumns <function1>, newInstance(class scala.Tuple2), [input[0, bigi
nt, false] AS value#262L]
            +- StreamingRelation rate, [timestamp#253, value#254L]
```

StatefulAggregationStrategy Execution Planning Strategy for EventTimeWatermark and Aggregate Logical Operators

`StatefulAggregationStrategy` is an execution planning strategy (i.e. `Strategy`) that IncrementalExecution uses to plan `EventTimeWatermark` and `Aggregate` logical operators in streaming Datasets.

Note	EventTimeWatermark logical operator is the result of withWatermark operator.

Note	`Aggregate` logical operator represents groupBy and groupByKey aggregations (and SQL's `GROUP BY` clause).

`StatefulAggregationStrategy` is available using `SessionState`.

```
spark.sessionState.planner.StatefulAggregationStrategy
```

Table 1. StatefulAggregationStrategy's Logical to Physical Operator Conversions

Logical Operator	Physical Operator
EventTimeWatermark	EventTimeWatermarkExec
`Aggregate`	In the order of preference: 1. `HashAggregateExec` 2. `ObjectHashAggregateExec` 3. `SortAggregateExec` Tip: Read Aggregation Execution Planning Strategy for Aggregate Physical Operators in **Mastering Apache Spark 2** gitbook.

```
val counts = spark.
  readStream.
  format("rate").
  load.
  groupBy(window($"timestamp", "5 seconds") as "group").
  agg(count("value") as "count").
  orderBy("group")
scala> counts.explain
== Physical Plan ==
*Sort [group#6 ASC NULLS FIRST], true, 0
+- Exchange rangepartitioning(group#6 ASC NULLS FIRST, 200)
   +- *HashAggregate(keys=[window#13], functions=[count(value#1L)])
      +- StateStoreSave [window#13], StatefulOperatorStateInfo(<unknown>,736d67c2-6daa
-4c4c-9c4b-c12b15af20f4,0,0), Append, 0
         +- *HashAggregate(keys=[window#13], functions=[merge_count(value#1L)])
            +- StateStoreRestore [window#13], StatefulOperatorStateInfo(<unknown>,736d
67c2-6daa-4c4c-9c4b-c12b15af20f4,0,0)
               +- *HashAggregate(keys=[window#13], functions=[merge_count(value#1L)])
                  +- Exchange hashpartitioning(window#13, 200)
                     +- *HashAggregate(keys=[window#13], functions=[partial_count(valu
e#1L)])
                        +- *Project [named_struct(start, precisetimestampconversion(((
((CASE WHEN (cast(CEIL((cast((precisetimestampconversion(timestamp#0, TimestampType, L
ongType) - 0) as double) / 5000000.0)) as double) = (cast((precisetimestampconversion(
timestamp#0, TimestampType, LongType) - 0) as double) / 5000000.0)) THEN (CEIL((cast((
precisetimestampconversion(timestamp#0, TimestampType, LongType) - 0) as double) / 500
0000.0)) + 1) ELSE CEIL((cast((precisetimestampconversion(timestamp#0, TimestampType,
LongType) - 0) as double) / 5000000.0)) END + 0) - 1) * 5000000) + 0), LongType, Times
tampType), end, precisetimestampconversion(((((CASE WHEN (cast(CEIL((cast((precisetime
stampconversion(timestamp#0, TimestampType, LongType) - 0) as double) / 5000000.0)) as
 double) = (cast((precisetimestampconversion(timestamp#0, TimestampType, LongType) - 0
) as double) / 5000000.0)) THEN (CEIL((cast((precisetimestampconversion(timestamp#0, T
imestampType, LongType) - 0) as double) / 5000000.0)) + 1) ELSE CEIL((cast((precisetim
estampconversion(timestamp#0, TimestampType, LongType) - 0) as double) / 5000000.0)) E
ND + 0) - 1) * 5000000) + 5000000), LongType, TimestampType)) AS window#13, value#1L]
                           +- *Filter isnotnull(timestamp#0)
                              +- StreamingRelation rate, [timestamp#0, value#1L]

import org.apache.spark.sql.streaming.{OutputMode, Trigger}
import scala.concurrent.duration._
val consoleOutput = counts.
  writeStream.
  format("console").
  option("truncate", false).
  trigger(Trigger.ProcessingTime(10.seconds)).
  queryName("counts").
  outputMode(OutputMode.Complete).  // <-- required for groupBy
  start

// Eventually...
consoleOutput.stop
```

Selecting Aggregate Physical Operator Given Aggregate Expressions — `AggUtils.planStreamingAggregation` Internal Method

```
planStreamingAggregation(
  groupingExpressions: Seq[NamedExpression],
  functionsWithoutDistinct: Seq[AggregateExpression],
  resultExpressions: Seq[NamedExpression],
  child: SparkPlan): Seq[SparkPlan]
```

`planStreamingAggregation` takes the grouping attributes (from `groupingExpressions`).

Note	`groupingExpressions` corresponds to the grouping function in groupBy operator.

`planStreamingAggregation` creates an aggregate physical operator (called `partialAggregate`) with:

- `requiredChildDistributionExpressions` undefined (i.e. `None`)
- `initialInputBufferOffset` as `0`
- `functionsWithoutDistinct` in `Partial` mode
- `child` operator as the input `child`

<table><tr><td>Note</td><td><code>planStreamingAggregation</code> creates one of the following aggregate physical operators (in the order of preference):
1. <code>HashAggregateExec</code>
2. <code>ObjectHashAggregateExec</code>
3. <code>SortAggregateExec</code>
<code>planStreamingAggregation</code> uses <code>AggUtils.createAggregate</code> method to select an aggregate physical operator that you can read about in Selecting Aggregate Physical Operator Given Aggregate Expressions — <code>AggUtils.createAggregate</code> Internal Method in Mastering Apache Spark 2 gitbook.</td></tr></table>

`planStreamingAggregation` creates an aggregate physical operator (called `partialMerged1`) with:

- `requiredChildDistributionExpressions` based on the input `groupingExpressions`
- `initialInputBufferOffset` as the length of `groupingExpressions`
- `functionsWithoutDistinct` in `PartialMerge` mode
- `child` operator as partialAggregate aggregate physical operator created above

`planStreamingAggregation` creates StateStoreRestoreExec with the grouping attributes, undefined `StatefulOperatorStateInfo`, and partialMerged1 aggregate physical operator created above.

`planStreamingAggregation` creates an aggregate physical operator (called `partialMerged2`) with:

- `child` operator as StateStoreRestoreExec physical operator created above

Note	The only difference between partialMerged1 and partialMerged2 steps is the child physical operator.

`planStreamingAggregation` creates StateStoreSaveExec with:

- the grouping attributes based on the input `groupingExpressions`
- No `stateInfo`, `outputMode` and `eventTimeWatermark`
- `child` operator as partialMerged2 aggregate physical operator created above

In the end, `planStreamingAggregation` creates the final aggregate physical operator (called `finalAndCompleteAggregate`) with:

- `requiredChildDistributionExpressions` based on the input `groupingExpressions`
- `initialInputBufferOffset` as the length of `groupingExpressions`
- `functionsWithoutDistinct` in `Final` mode
- `child` operator as StateStoreSaveExec physical operator created above

Note	`planStreamingAggregation` is used exclusively when `StatefulAggregationStrategy` plans a streaming aggregation.

StreamingDeduplicationStrategy Execution Planning Strategy for Deduplicate Logical Operator

`StreamingDeduplicationStrategy` is an execution planning strategy (i.e. `Strategy`) that IncrementalExecution uses to plan `Deduplicate` logical operators in streaming Datasets.

Note	Deduplicate logical operator is the result of dropDuplicates operator.

`StreamingDeduplicationStrategy` is available using `SessionState` .

```
spark.sessionState.planner.StreamingDeduplicationStrategy
```

`StreamingDeduplicationStrategy` resolves streaming Deduplicate unary logical operators to StreamingDeduplicateExec physical operators.

```
FIXME
```

StreamingRelationStrategy Execution Planning Strategy for StreamingRelation and StreamingExecutionRelation Logical Operators

`StreamingRelationStrategy` is an streaming execution planning strategy (i.e. `Strategy`) that converts `StreamingRelation` and `StreamingExecutionRelation` logical operators (in the logical query plan of a streaming Dataset) to `StreamingRelationExec` physical operator.

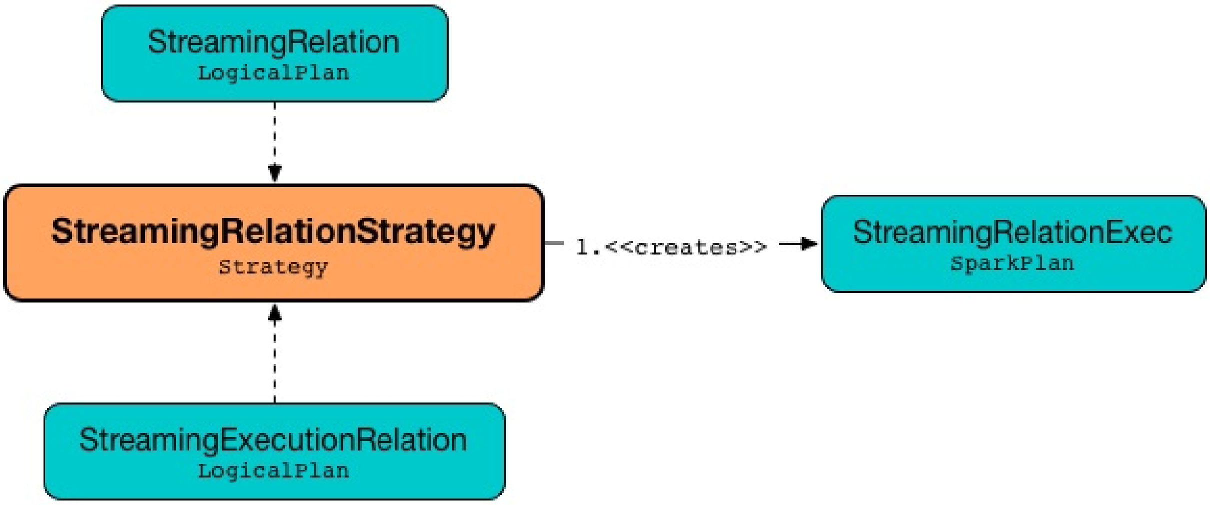

Figure 1. StreamingRelationStrategy, StreamingRelation, StreamingExecutionRelation and StreamingRelationExec Operators

Note	StreamingRelation logical operator represents a streaming source in a logical plan and is created when `DataStreamReader` loads data from a streaming source (that gives a streaming Dataset). StreamingExecutionRelation logical operator also represents a streaming source in a logical plan, but is used internally when `StreamExecution` (of a streaming Dataset) initializes the logical query plan.

`StreamingRelationStrategy` is used exclusively when IncrementalExecution plans the logical plan of a streaming Dataset for explain operator.

`StreamingRelationStrategy` converts StreamingRelation and StreamingExecutionRelation logical operators in a logical query plan to a StreamingRelationExec physical operator (with their `sourceName` and `output` schema) to give a corresponding physical query plan.

`StreamingRelationStrategy` is available using `SessionState` (of a `SparkSession`).

```
spark.sessionState.planner.StreamingRelationStrategy
```

```
val rates = spark.
  readStream.
  format("rate").
  load // <-- gives a streaming Dataset with a logical plan with StreamingRelation log
ical operator

// StreamingRelation logical operator for the rate streaming source
scala> println(rates.queryExecution.logical.numberedTreeString)
00 StreamingRelation DataSource(org.apache.spark.sql.SparkSession@31ba0af0,rate,List(),
None,List(),None,Map(),None), rate, [timestamp#0, value#1L]

// StreamingRelationExec physical operator (shown without "Exec" suffix)
scala> rates.explain
== Physical Plan ==
StreamingRelation rate, [timestamp#0, value#1L]

// Let's do the planning manually
import spark.sessionState.planner.StreamingRelationStrategy
val physicalPlan = StreamingRelationStrategy.apply(rates.queryExecution.logical).head
scala> println(physicalPlan.numberedTreeString)
00 StreamingRelation rate, [timestamp#0, value#1L]
```

Offset

`Offset` is…FIXME

MetadataLog — Contract for Metadata Storage

`MetadataLog` is the contract to store metadata.

MetadataLog Contract

```
package org.apache.spark.sql.execution.streaming

trait MetadataLog[T] {
  def add(batchId: Long, metadata: T): Boolean
  def get(batchId: Long): Option[T]
  def get(startId: Option[Long], endId: Option[Long]): Array[(Long, T)]
  def getLatest(): Option[(Long, T)]
  def purge(thresholdBatchId: Long): Unit
}
```

Table 1. MetadataLog Contract

Method	Description
`add`	
`get`	
`getLatest`	Retrieves the latest-committed batch with the metadata if available from the metadata storage. Note: It is assumed (i.e. FileStreamSink) that the latest batch id is of the batch which has already been committed and a streaming query can start from.
`purge`	

HDFSMetadataLog — MetadataLog with Hadoop HDFS for Reliable Storage

`HDFSMetadataLog` is a MetadataLog that uses Hadoop HDFS for a reliable storage.

Note	`HDFSMetadataLog` uses path (specified when created) that is created automatically unless exists already.

`HDFSMetadataLog` is created when:

- `KafkaSource` is first requested for initial partition offsets (from the metadata storage)
- `RateStreamSource` is created (and looks up startTimeMs in the metadata storage)

`HDFSMetadataLog` is further customized to…FIXME

Table 1. HDFSMetadataLog's Available Implementations

HDFSMetadataLog	Description
BatchCommitLog	
CompactibleFileStreamLog	
OffsetSeqLog	

Table 2. HDFSMetadataLog's Internal Registries and Counters

Name	Description
`fileManager`	`FileManager` that…FIXME
`batchFilesFilter`	Filter of batch files
`metadataPath`	The path to metadata directory

Writing Metadata in Serialized Format — `serialize` Method

Caution	FIXME

`deserialize` Method

Caution	FIXME

`createFileManager` Internal Method

```
createFileManager(): FileManager
```

Caution	FIXME

Note	`createFileManager` is used exclusively when `HDFSMetadataLog` is created (and the internal FileManager is created alongside).

Retrieving Metadata By Batch Id — `get` Method

Caution	FIXME

`add` Method

Caution	FIXME

Retrieving Latest Committed Batch Id with Metadata If Available — `getLatest` Method

```
getLatest(): Option[(Long, T)]
```

Note	`getLatest` is a part of MetadataLog Contract to retrieve the recently-committed batch id and the corresponding metadata if available in the metadata storage.

`getLatest` requests the internal FileManager for the files in metadata directory that match batch file filter.

`getLatest` takes the batch ids (the batch files correspond to) and sorts the ids in reverse order.

`getLatest` gives the first batch id with the metadata which could be found in the metadata storage.

Note	It is possible that the batch id could be in the metadata storage, but not available for retrieval.

Creating HDFSMetadataLog Instance

`HDFSMetadataLog` takes the following when created:

- `SparkSession`
- Path of the metadata log directory

`HDFSMetadataLog` initializes the internal registries and counters.

`HDFSMetadataLog` creates the path unless exists already.

CommitLog — HDFSMetadataLog for Batch Completion Log

`CommitLog` is a HDFSMetadataLog with metadata as regular text (i.e. `String`).

Note	HDFSMetadataLog is a `MetadataLog` that uses Hadoop HDFS for a reliable storage.

`CommitLog` is created along with StreamExecution.

`add` Method

```
add(batchId: Long): Unit
```

`add` …FIXME

Note	`add` is used when…FIXME

`add` Method

```
add(batchId: Long, metadata: String): Boolean
```

Note	`add` is part of MetadataLog Contract to…FIXME.

`add` …FIXME

`serialize` Method

```
serialize(metadata: String, out: OutputStream): Unit
```

Note	`serialize` is part of HDFSMetadataLog Contract to write a metadata in serialized format.

`serialize` writes out the version prefixed with `v` on a single line (e.g. `v1`) followed by the empty JSON (i.e. `{}`).

Note	The version in Spark 2.2 is **1** with the charset being **UTF-8**.

Note	`serialize` always writes an empty JSON as the name of the files gives the meaning.

```
$ ls -tr [checkpoint-directory]/commits
0 1 2 3 4 5 6 7 8 9

$ cat [checkpoint-directory]/commits/8
v1
{}
```

`deserialize` Method

```
deserialize(in: InputStream): String
```

Note	`deserialize` is part of HDFSMetadataLog Contract to…FIXME.

`deserialize` …FIXME

Creating CommitLog Instance

`CommitLog` takes the following when created:

- `SparkSession`
- Path of the metadata log directory

CompactibleFileStreamLog

OffsetSeqLog — HDFSMetadataLog with OffsetSeq Metadata

`OffsetSeqLog` is a HDFSMetadataLog with metadata as OffsetSeq.

Note	HDFSMetadataLog is a `MetadataLog` that uses Hadoop HDFS for a reliable storage.

`OffsetSeqLog` is created exclusively for write-ahead log of offsets in `StreamExecution`.

`OffsetSeqLog` uses OffsetSeq for metadata which holds an ordered collection of zero or more offsets and optional metadata (as OffsetSeqMetadata for keeping track of event time watermark as set up by a Spark developer and what was found in the records).

`serialize` Method

```
serialize(offsetSeq: OffsetSeq, out: OutputStream): Unit
```

Note	`serialize` is a part of HDFSMetadataLog Contract to write a metadata in serialized format.

`serialize` firstly writes out the version prefixed with `v` on a single line (e.g. `v1`) followed by the optional metadata in JSON format.

Note	The version in Spark 2.2 is **1** with the charset being **UTF-8**.

`serialize` then writes out the offsets in JSON format, one per line.

Note	No offsets to write in `offsetSeq` for a streaming source is marked as **-** (a dash) in the log.

```
$ ls -tr [checkpoint-directory]/offsets
0 1 2 3 4 5 6

$ cat [checkpoint-directory]/offsets/6
v1
{"batchWatermarkMs":0,"batchTimestampMs":1502872590006,"conf":{"spark.sql.shuffle.part
itions":"200","spark.sql.streaming.stateStore.providerClass":"org.apache.spark.sql.exe
cution.streaming.state.HDFSBackedStateStoreProvider"}}
51
```

`deserialize` Method

```
deserialize(in: InputStream): OffsetSeq
```

Caution	FIXME

Creating OffsetSeqLog Instance

`OffsetSeqLog` takes the following when created:

- `SparkSession`
- Path of the metadata log directory

OffsetSeq

`OffsetSeq` is…FIXME

`OffsetSeq` is created when…MEFIXME

`toStreamProgress` Method

```
toStreamProgress(sources: Seq[BaseStreamingSource]): StreamProgress
```

`toStreamProgress` then creates a new StreamProgress with the only `sources` that have the offsets defined.

`toStreamProgress` throws a `AssertionError` if the number of the input `sources` does not match the offsets.

Note	`toStreamProgress` is used when: • `MicroBatchExecution` is requested to populateStartOffsets and constructNextBatch • `ContinuousExecution` is requested to getStartOffsets

Creating OffsetSeq Instance

`OffsetSeq` takes the following when created:

- Collection of Offsets
- OffsetSeqMetadata (default: `None`)

OffsetSeqMetadata

`OffsetSeqMetadata` holds the metadata for…FIXME:

- `batchWatermarkMs` - the current event time watermark in milliseconds (used to bound the lateness of data that will processed)
- `batchTimestampMs` - the current batch processing timestamp in milliseconds
- `conf` - batch configuration with `spark.sql.shuffle.partitions` and spark.sql.streaming.stateStore.providerClass Spark properties

Note	`OffsetSeqMetadata` is used mainly when `IncrementalExecution` is created.

StateStore — Streaming Aggregation State Management

`StateStore` is the contract of a versioned and fault-tolerant key-value store for persisting state of running aggregates across streaming batches (for aggregate operations on streaming Datasets).

Tip	Read the motivation and design in State Store for Streaming Aggregations.

`StateStore` describes a key-value store that lives on every executor (across the nodes in a Spark cluster) for persistent keyed aggregates.

`StateStore` is identified with the aggregating operator id and the partition id.

```
package org.apache.spark.sql.execution.streaming.state

trait StateStore {
  def abort(): Unit
  def commit(): Long
  def get(key: UnsafeRow): UnsafeRow
  def getRange(start: Option[UnsafeRow], end: Option[UnsafeRow]): Iterator[UnsafeRowPa
ir]
  def hasCommitted: Boolean
  def id: StateStoreId
  def iterator(): Iterator[UnsafeRowPair]
  def metrics: StateStoreMetrics
  def put(key: UnsafeRow, value: UnsafeRow): Unit
  def remove(key: UnsafeRow): Unit
  def version: Long
}
```

Table 1. StateStore Contract

<table>
<tr><th>Method</th><th>Description</th></tr>
<tr><td><code>abort</code></td><td></td></tr>
<tr><td><code>commit</code></td><td></td></tr>
<tr><td><code>get</code></td><td>Used exclusively when <code>StateStoreRDD</code> is executed.</td></tr>
<tr><td><code>getRange</code></td><td></td></tr>
<tr><td><code>hasCommitted</code></td><td></td></tr>
<tr><td><code>id</code></td><td></td></tr>
<tr><td><code>iterator</code></td><td></td></tr>
<tr><td><code>metrics</code></td><td></td></tr>
<tr><td><code>put</code></td><td>Stores a value for a non-null key (both of <code>UnsafeRow</code> type)
Used when:
<code>StateStoreSaveExec</code> is executed (and…FIXME)
<code>StreamingDeduplicateExec</code> is executed (and…FIXME)
<code>StateStoreUpdater</code> attempts to write the current state when rows are processed (which is when their iterator is fully consumed).
<table><tr><td>Caution</td><td>FIXME Review StateStoreUpdater.callFunctionAndUpdateState</td></tr></table></td></tr>
<tr><td><code>remove</code></td><td></td></tr>
<tr><td><code>version</code></td><td></td></tr>
</table>

Table 2. StateStore's Internal Registries and Counters

<table>
<tr><th>Name</th><th>Description</th></tr>
<tr><td><code>loadedProviders</code></td><td>Registry of StateStoreProviders per <code>StateStoreProviderId</code>
Used in…FIXME</td></tr>
<tr><td><code>_coordRef</code></td><td>StateStoreCoordinatorRef (a <code>RpcEndpointRef</code> to StateStoreCoordinator).
Used in…FIXME</td></tr>
</table>

Note	`StateStore` was introduced in [SPARK-13809][SQL] State store for streaming aggregations.

Creating StateStoreCoordinatorRef (for Executors) — `coordinatorRef` Internal Method

Caution	FIXME

Removing StateStoreProvider From Provider Registry — `unload` Internal Method

Caution	FIXME

`verifyIfStoreInstanceActive` Internal Method

Caution	FIXME

Announcing New StateStoreProvider — `reportActiveStoreInstance` Internal Method

```
reportActiveStoreInstance(storeProviderId: StateStoreProviderId): Unit
```

`reportActiveStoreInstance` takes the current host and executorId (from `BlockManager`) and requests `StateStoreCoordinatorRef` to reportActiveInstance.

Note	`reportActiveStoreInstance` uses `SparkEnv` to access the current `BlockManager`.

You should see the following INFO message in the logs:

```
Reported that the loaded instance [storeProviderId] is active
```

Note	`reportActiveStoreInstance` is used exclusively when `StateStore` is requested to find the StateStore by StateStoreProviderId.

`numKeys` Method

Caution	FIXME

Finding StateStore by StateStoreProviderId — `get` Method

```
get(
  storeProviderId: StateStoreProviderId,
  keySchema: StructType,
  valueSchema: StructType,
  indexOrdinal: Option[Int],
  version: Long,
  storeConf: StateStoreConf,
  hadoopConf: Configuration): StateStore
```

`get` finds `StateStore` for `StateStoreProviderId`.

Internally, `get` looks up the `StateStoreProvider` (for `storeProviderId`) in loadedProviders registry. If unavailable, `get` creates and initializes one.

`get` will also start the periodic maintenance task (unless already started) and announce the new StateStoreProvider.

In the end, `get` gets the `StateStore` (for the `version`).

Note	`get` is used exclusively when `StateStoreRDD` is computed.

Starting Periodic Maintenance Task (Unless Already Started) — `startMaintenanceIfNeeded` Internal Method

```
startMaintenanceIfNeeded(): Unit
```

`startMaintenanceIfNeeded` schedules MaintenanceTask to start after and every spark.sql.streaming.stateStore.maintenanceInterval (defaults to `60s`).

Note	`startMaintenanceIfNeeded` does nothing when the maintenance task has already been started and is still running.

Note	`startMaintenanceIfNeeded` is used exclusively when `StateStore` is requested to find the StateStore by StateStoreProviderId.

`MaintenanceTask` Daemon Thread

`MaintenanceTask` is a daemon thread that triggers maintenance work of every registered StateStoreProvider.

When an error occurs, `MaintenanceTask` clears loadedProviders registry.

`MaintenanceTask` is scheduled on **state-store-maintenance-task** thread pool.

Note	Use spark.sql.streaming.stateStore.maintenanceInterval Spark property (default: `60s`) to control the initial delay and how often the thread should be executed.

Triggering Maintenance of Registered StateStoreProviders — `doMaintenance` Internal Method

```
doMaintenance(): Unit
```

Internally, `doMaintenance` prints the following DEBUG message to the logs:

```
DEBUG Doing maintenance
```

`doMaintenance` then requests every StateStoreProvider (registered in loadedProviders) to do its own internal maintenance (only when a `StateStoreProvider` is still active).

When a `StateStoreProvider` is inactive, `doMaintenance` removes it from the provider registry and prints the following INFO message to the logs:

```
INFO Unloaded [provider]
```

Note	`doMaintenance` is used exclusively in MaintenanceTask daemon thread.

StateStoreOps — Extension Methods for Creating StateStoreRDD

`StateStoreOps` is a **Scala implicit class** to create `StateStoreRDD` when the following physical operators are executed:

- FlatMapGroupsWithStateExec
- StateStoreRestoreExec
- StateStoreSaveExec
- StreamingDeduplicateExec

Note	Implicit Classes are a language feature in Scala for **implicit conversions** with **extension methods** for existing types.

Creating StateStoreRDD (with storeUpdateFunction Aborting StateStore When Task Fails) — `mapPartitionsWithStateStore` Method

```
mapPartitionsWithStateStore[U](
  sqlContext: SQLContext,
  stateInfo: StatefulOperatorStateInfo,
  keySchema: StructType,
  valueSchema: StructType,
  indexOrdinal: Option[Int])(
  storeUpdateFunction: (StateStore, Iterator[T]) => Iterator[U]): StateStoreRDD[T, U] (
1)
mapPartitionsWithStateStore[U](
  stateInfo: StatefulOperatorStateInfo,
  keySchema: StructType,
  valueSchema: StructType,
  indexOrdinal: Option[Int],
  sessionState: SessionState,
  storeCoordinator: Option[StateStoreCoordinatorRef])(
  storeUpdateFunction: (StateStore, Iterator[T]) => Iterator[U]): StateStoreRDD[T, U]
```

1. Uses `sqlContext.streams.stateStoreCoordinator` to access `StateStoreCoordinator`

Internally, `mapPartitionsWithStateStore` requests `SparkContext` to clean `storeUpdateFunction` function.

Note	`mapPartitionsWithStateStore` uses the enclosing RDD to access the current `SparkContext` .

Note	**Function Cleaning** is to clean a closure from unreferenced variables before it is serialized and sent to tasks. `SparkContext` reports a `SparkException` when the closure is not serializable.

`mapPartitionsWithStateStore` then creates a (wrapper) function to abort the `StateStore` if state updates had not been committed before a task finished (which is to make sure that the `StateStore` has been committed or aborted in the end to follow the contract of `StateStore`).

Note	`mapPartitionsWithStateStore` uses `TaskCompletionListener` to be notified when a task has finished.

In the end, `mapPartitionsWithStateStore` creates a StateStoreRDD (with the wrapper function, `SessionState` and StateStoreCoordinatorRef).

Note	`mapPartitionsWithStateStore` is used when the following physical operators are executed: • FlatMapGroupsWithStateExec • StateStoreRestoreExec • StateStoreSaveExec • StreamingDeduplicateExec

StateStoreProvider

`StateStoreProvider` is…FIXME

`doMaintenance` Method

Caution	FIXME

`createAndInit` Method

Caution	FIXME

`getStore` Method

Caution	FIXME

Note	`getStore` is used exclusively when `StateStore` is requested for a state store (given a `StateStoreProviderId`).

StateStoreUpdater

`StateStoreUpdater` is…FIXME

`updateStateForKeysWithData` Method

Caution	FIXME

`updateStateForTimedOutKeys` Method

Caution	FIXME

StateStoreWriter — Recording Metrics For Writing to StateStore

`StateStoreWriter` is a contract for physical operators (i.e. `SparkPlan`) to record metrics when writing to a StateStore.

Table 1. StateStoreWriter's SQLMetrics

Name	Description
`numOutputRows`	Number of output rows
`numTotalStateRows`	number of total state rows
`numUpdatedStateRows`	number of updated state rows
`allUpdatesTimeMs`	total time to update rows
`allRemovalsTimeMs`	total time to remove rows
`commitTimeMs`	time to commit changes
`stateMemory`	memory used by state (store)

Setting StateStore-Specific Metrics for Physical Operator — `setStoreMetrics` Method

```
setStoreMetrics(store: StateStore): Unit
```

`setStoreMetrics` requests `store` for metrics to use them to record the following metrics of a physical operator:

- `numTotalStateRows` as `StateStore.numKeys`
- `stateMemory` as `StateStore.memoryUsedBytes`

`setStoreMetrics` records the implementation-specific metrics.

Note	`setStoreMetrics` is used when: • `FlatMapGroupsWithStateExec` is executed • `StateStoreSaveExec` is executed • `StreamingDeduplicateExec` is executed

HDFSBackedStateStore

`HDFSBackedStateStore` is a StateStore that uses a HDFS-compatible file system for versioned state persistence.

`HDFSBackedStateStore` is created exclusively when `HDFSBackedStateStoreProvider` is requested for a state store (which is when…FIXME).

`HDFSBackedStateStore` can be in the following states:

- `UPDATING`
- `COMMITTED`
- `ABORTED`

Table 1. HDFSBackedStateStore's Internal Registries and Counters

Name	Description
`newVersion`	
`tempDeltaFile`	
`state`	
`tempDeltaFileStream`	
`finalDeltaFile`	

`writeUpdateToDeltaFile` Internal Method

```
writeUpdateToDeltaFile(
  output: DataOutputStream,
  key: UnsafeRow,
  value: UnsafeRow): Unit
```

Caution	FIXME

`put` Method

```
put(key: UnsafeRow, value: UnsafeRow): Unit
```

Note	`put` is a part of StateStore Contract to…FIXME

`put` stores the copies of the key and value in mapToUpdate internal registry followed by writing them to a delta file (using tempDeltaFileStream).

<table><tr><td>Note</td><td><code>put</code> can only be used when <code>HDFSBackedStateStore</code> is in <code>UPDATING</code> state and reports a <code>IllegalStateException</code> otherwise.<pre><code>Cannot put after already committed or aborted</code></pre></td></tr></table>

`commit` Method

Caution	FIXME

Creating HDFSBackedStateStore Instance

`HDFSBackedStateStore` takes the following when created:

- Version
- Key-value registry of `UnsafeRows` (as Java's java.util.concurrent.ConcurrentHashMap)

`HDFSBackedStateStore` initializes the internal registries and counters.

HDFSBackedStateStoreProvider

`HDFSBackedStateStoreProvider` is…FIXME

Creating HDFSBackedStateStore — `getStore` Method

```
getStore(version: Long): StateStore
```

Note	`getStore` is a part of StateStoreProvider Contract to..FIXME.

Caution	FIXME

StateStoreRDD — RDD for Updating State (in StateStores Across Spark Cluster)

`StateStoreRDD` is an `RDD` for executing storeUpdateFunction with StateStore (and data from partitions of a new batch RDD).

`StateStoreRDD` is created when executing physical operators that work with state:

- FlatMapGroupsWithStateExec (for mapGroupsWithState and flatMapGroupsWithState operators)
- StateStoreRestoreExec and StateStoreSaveExec (for groupBy, `rollup`, and `cube` operators)
- StreamingDeduplicateExec (for dropDuplicates operator)

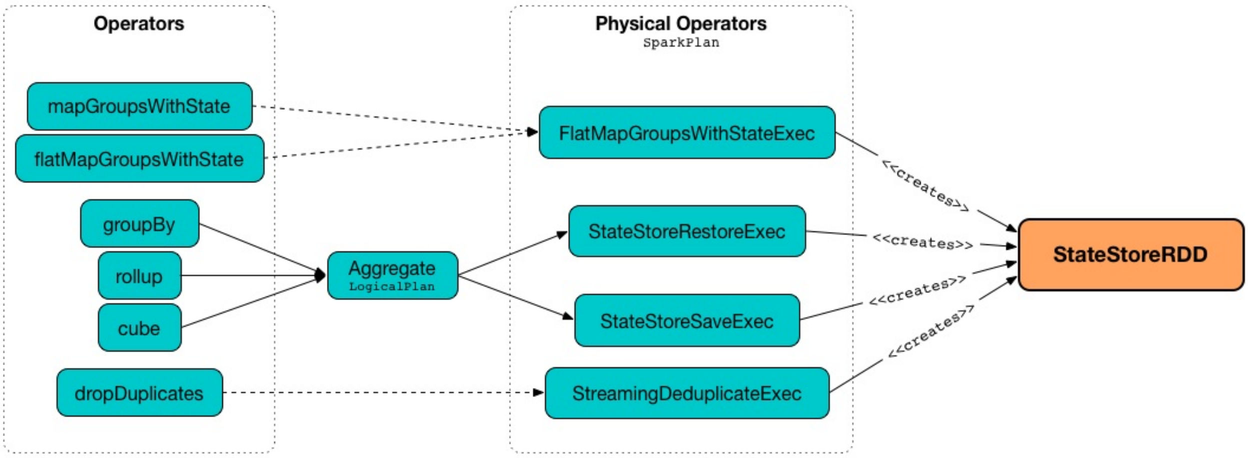

Figure 1. StateStoreRDD, Physical and Logical Plans, and operators

`StateStoreRDD` uses `StateStoreCoordinator` for preferred locations for job scheduling.

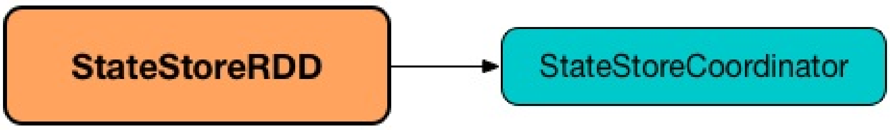

Figure 2. StateStoreRDD and StateStoreCoordinator

`getPartitions` is exactly the partitions of the data RDD.

Table 1. StateStoreRDD's Internal Registries and Counters

Name	Description
`hadoopConfBroadcast`	
`storeConf`	Configuration parameters (as `StateStoreConf`) using the current `SQLConf` (from `SessionState`)

Computing Partition (in TaskContext) — `compute` Method

```
compute(partition: Partition, ctxt: TaskContext): Iterator[U]
```

Note	`compute` is a part of the RDD Contract to compute a given partition in a `TaskContext`.

`compute` computes dataRDD passing the result on to storeUpdateFunction (with a configured StateStore).

Internally, (and similarly to getPreferredLocations) `compute` creates a `StateStoreProviderId` with `StateStoreId` (using checkpointLocation, operatorId and the index of the input `partition`) and queryRunId.

`compute` then requests `StateStore` for the store for the StateStoreProviderId.

In the end, `compute` computes dataRDD (using the input `partition` and `ctxt`) followed by executing storeUpdateFunction (with the store and the result).

Getting Placement Preferences of Partition — `getPreferredLocations` Method

```
getPreferredLocations(partition: Partition): Seq[String]
```

Note	`getPreferredLocations` is a part of the RDD Contract to specify placement preferences (aka *preferred task locations*), i.e. where tasks should be executed to be as close to the data as possible.

`getPreferredLocations` creates a `StateStoreProviderId` with `StateStoreId` (using checkpointLocation, operatorId and the index of the input `partition`) and queryRunId.

Note	checkpointLocation and operatorId are shared across different partitions and so the only difference in `StateStoreProviderIds` is the partition index.

In the end, `getPreferredLocations` requests StateStoreCoordinatorRef for the location of the state store for `StateStoreProviderId`.

Note	StateStoreCoordinator coordinates instances of `StateStores` across Spark executors in the cluster, and tracks their locations for job scheduling.

Creating StateStoreRDD Instance

`StateStoreRDD` takes the following when created:

- `RDD` with the new streaming batch data (to update the aggregates in a state store)
- Store update function (i.e. `(StateStore, Iterator[T]) ⇒ Iterator[U]` with `T` being the type of the new batch data)
- The path to the checkpoint location
- `queryRunId`
- Operator id
- Store version
- Schema of the keys
- Schema of the values
- Optional index
- `SessionState`
- Optional StateStoreCoordinatorRef

`StateStoreRDD` initializes the internal registries and counters.

StateStoreCoordinator — Tracking Locations of StateStores for StateStoreRDD

`StateStoreCoordinator` keeps track of StateStores loaded in Spark executors (across the nodes in a Spark cluster).

The main purpose of `StateStoreCoordinator` is for `StateStoreRDD` to get the location preferences for partitions (based on the location of the stores).

`StateStoreCoordinator` uses `instances` internal registry of StateStoreProviders by their ids and `ExecutorCacheTaskLocations`.

`StateStoreCoordinator` is a `ThreadSafeRpcEndpoint` RPC endpoint that manipulates instances registry through RPC messages.

Table 1. StateStoreCoordinator RPC Endpoint's Messages and Message H

Message	Message Handle
`DeactivateInstances`	Removes `StateStoreProviderIds` (from instances) with `queryR` You should see the following DEBUG message in the logs: `DEBUG Deactivating instances related to checkpoint location`
`GetLocation`	Gives the location of `StateStoreProviderId` (from instances) w You should see the following DEBUG message in the logs: `DEBUG Got location of the state store [id]: [executorId]`
`ReportActiveInstance`	Registers `StateStoreProviderId` that is active on an executor (You should see the following DEBUG message in the logs: `DEBUG Reported state store [id] is active at [executorId]`
`StopCoordinator`	Stops `StateStoreCoordinator` RPC Endpoint You should see the following DEBUG message in the logs: `INFO StateStoreCoordinator stopped`
`VerifyIfInstanceActive`	Verifies if `StateStoreProviderId` is registered (in instances) on You should see the following DEBUG message in the logs: `DEBUG Verified that state store [id] is active: [response]`

Tip

Enable `INFO` or `DEBUG` logging level for `org.apache.spark.sql.execution.streaming.state.StateStoreCoordinator` to see what inside.

Add the following line to `conf/log4j.properties` :

```
log4j.logger.org.apache.spark.sql.execution.streaming.state.StateStoreCoordinato
```

Refer to Logging.

StateStoreCoordinatorRef Interface for Communication with StateStoreCoordinator

`StateStoreCoordinatorRef` allows for communication with StateStoreCoordinator (through `rpcEndpointRef` reference).

Table 1. StateStoreCoordinatorRef's Methods and Underlying RPC Messages (in alphabetical order)

Method	RPC Message	Description
`deactivateInstances`	`DeactivateInstances`	Synchronous event to announce that `StreamingQueryManager` has been informed that a query terminated (which is when `StreamExecution` has finished (running streaming batches)). Note: Refer to DeactivateInstances (of `StateStoreCoordinator`) to know how the event is handled.
`getLocation`	`GetLocation`	
`reportActiveInstance`	`ReportActiveInstance`	
`stop`	`StopCoordinator`	
`verifyIfInstanceActive`	`VerifyIfInstanceActive`	

Creating StateStoreCoordinatorRef with StateStoreCoordinator RPC Endpoint — `forDriver` Method

```
forDriver(env: SparkEnv): StateStoreCoordinatorRef
```

`forDriver` …FIXME

Note	`forDriver` is used exclusively when `StreamingQueryManager` is created.

`forExecutor` Method

```
forExecutor(env: SparkEnv): StateStoreCoordinatorRef
```

`forExecutor` ...FIXME

Note	`forExecutor` is used exclusively when `StateStore` creates a StateStoreCoordinatorRef (for executors).

StreamingQueryListener — Intercepting Streaming Events

`StreamingQueryListener` is the contract for listeners that want to be notified about the life cycle events of streaming queries, i.e. start, progress and termination of a query.

```
package org.apache.spark.sql.streaming

abstract class StreamingQueryListener {
  def onQueryStarted(event: QueryStartedEvent): Unit
  def onQueryProgress(event: QueryProgressEvent): Unit
  def onQueryTerminated(event: QueryTerminatedEvent): Unit
}
```

Table 1. StreamingQueryListener's Life Cycle Events and Callbacks

Event	Callback	When Posted
`QueryStartedEvent` • `id` • `runId` • `name`	`onQueryStarted`	Right after `StreamExecution` has started running streaming batches.
`QueryProgressEvent` • StreamingQueryProgress	`onQueryProgress`	`ProgressReporter` reports query progress (which is when `StreamExecution` runs batches and a trigger has finished).
`QueryTerminatedEvent` • `id` • `runId` • Optional `exception` if terminated due to an error	`onQueryTerminated`	Right before `StreamExecution` finishes running streaming batches (due to a stop or an exception).

You can register a `StreamingQueryListener` using StreamingQueryManager.addListener method.

```
val queryListener: StreamingQueryListener = ...
spark.streams.addListener(queryListener)
```

You can remove a `StreamingQueryListener` using StreamingQueryManager.removeListener method.

```
val queryListener: StreamingQueryListener = ...
spark.streams.removeListener(queryListener)
```

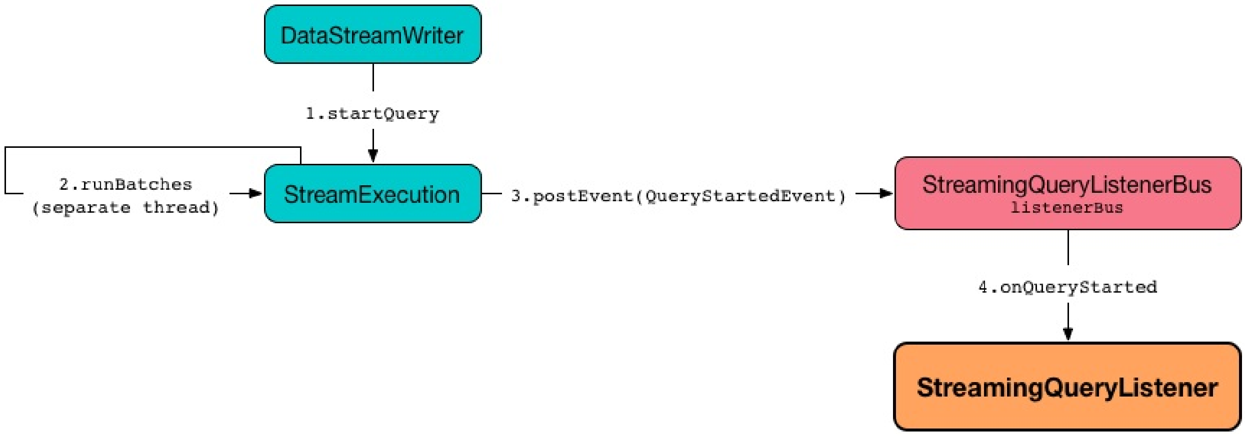

Figure 1. StreamingQueryListener Notified about Query's Start (onQueryStarted)

Note	`onQueryStarted` is used internally to unblock the starting thread of `StreamExecution` .

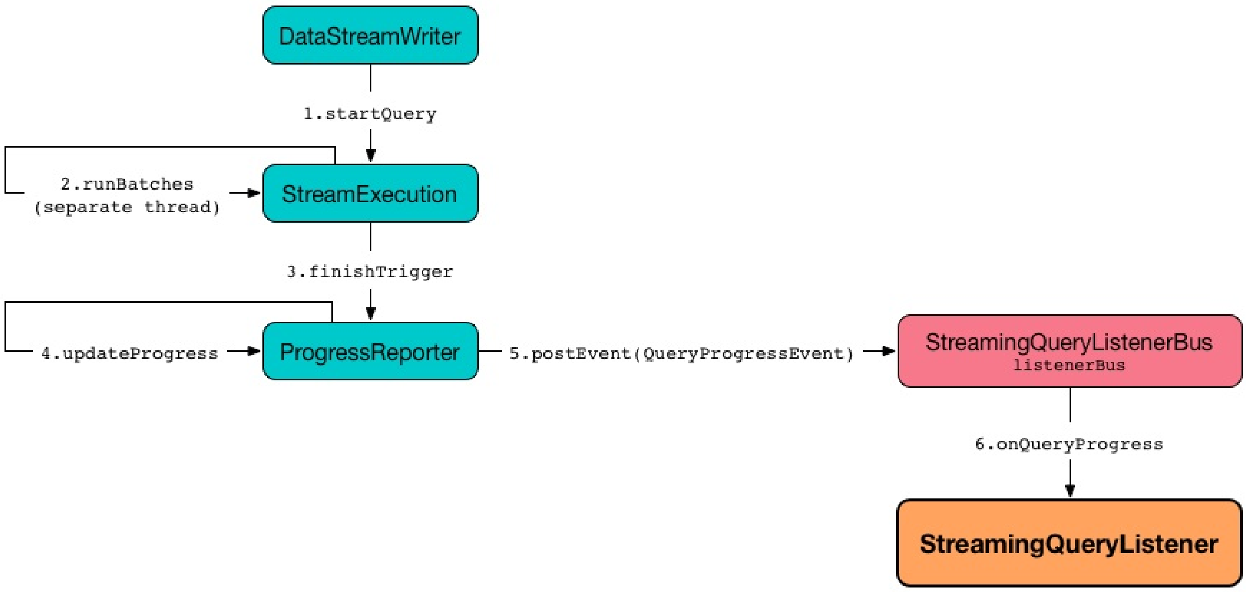

Figure 2. StreamingQueryListener Notified about Query's Progress (onQueryProgress)

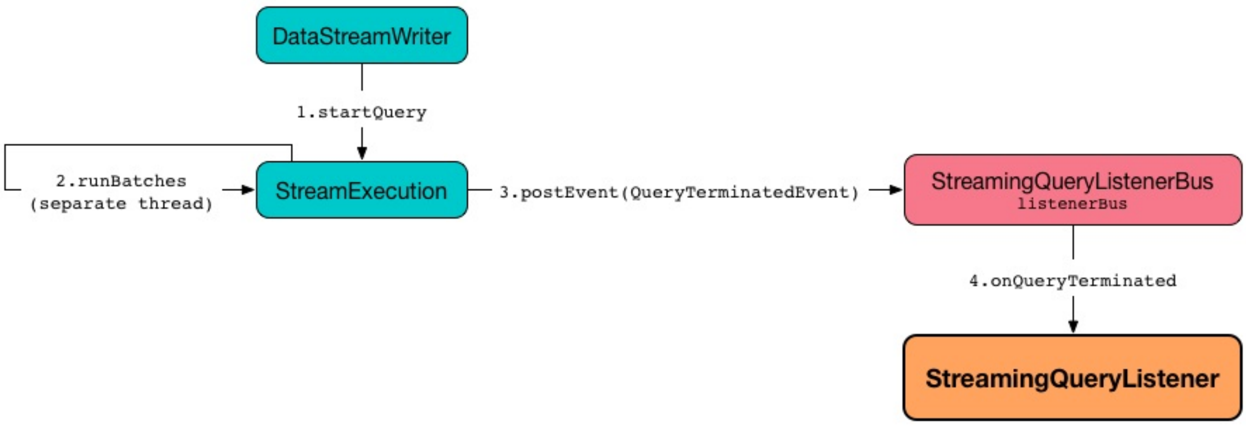

Figure 3. StreamingQueryListener Notified about Query's Termination (onQueryTerminated)

Note	You can also register a streaming event listener using the general `SparkListener` interface. Details on `SparkListener` interface can be found in the Mastering Apache Spark 2 gitbook.

StreamingQueryProgress

`StreamingQueryProgress` holds information about the progress of a streaming query.

`StreamingQueryProgress` is created exclusively when `StreamExecution` finishes a trigger.

Note	Use lastProgress property of a `StreamingQuery` to access the most recent `StreamingQueryProgress` update.
	`val sq: StreamingQuery = ...`
	`sq.lastProgress`

Note	Use recentProgress property of a `StreamingQuery` to access the most recent `StreamingQueryProgress` updates.
	`val sq: StreamingQuery = ...`
	`sq.recentProgress`

Note	Use StreamingQueryListener to get notified about `StreamingQueryProgress` updates while a streaming query is executed.

Table 1. StreamingQueryProgress's Properties

Name	Description
`id`	Unique identifier of a streaming query
`runId`	Unique identifier of the current execution of a streaming query
`name`	Optional query name
`timestamp`	Time when the trigger has started (in ISO8601 format).
`batchId`	Unique id of the current batch
`durationMs`	Durations of the internal phases (in milliseconds)
`eventTime`	Statistics of event time seen in this batch
`stateOperators`	Information about stateful operators in the query that store state.
`sources`	Statistics about the data read from every streaming source in a streaming query
`sink`	Information about progress made for a sink

Web UI

Web UI…FIXME

Caution	FIXME What’s visible on the plan diagram in the SQL tab of the UI

Logging

Caution	FIXME

DataSource — Pluggable Data Source

`DataSource` is…FIXME

`DataSource` is created when…FIXME

Tip	Read DataSource — Pluggable Data Sources (for Spark SQL's batch structured queries).

Table 1. DataSource's Internal Registries and Counters

Name	Description
`sourceInfo`	`SourceInfo` with the name, the schema, and optional partitioning columns of a source. Used when: • `DataSource` creates a FileStreamSource (that requires the schema and the optional partitioning columns) • `StreamingRelation` is created (for a `DataSource`)

Describing Name and Schema of Streaming Source — `sourceSchema` Internal Method

```
sourceSchema(): SourceInfo
```

`sourceSchema` …FIXME

Note	`sourceSchema` is used exclusively when `DataSource` is requested SourceInfo.

Creating DataSource Instance

`DataSource` takes the following when created:

- `SparkSession`
- Name of the class
- Paths (default: `Nil` , i.e. an empty collection)
- Optional user-defined schema (default: `None`)
- Names of the partition columns (default: empty)

- Optional `BucketSpec` (default: `None`)
- Configuration options (default: empty)
- Optional `CatalogTable` (default: `None`)

`DataSource` initializes the internal registries and counters.

`createSource` Method

```
createSource(metadataPath: String): Source
```

`createSource` …FIXME

Note	`createSource` is used when…FIXME

Creating Streaming Sink — `createSink` Method

```
createSink(outputMode: OutputMode): Sink
```

`createSink` …FIXME

Note	`createSink` is used exclusively when `DataStreamWriter` is requested to start.

StreamSourceProvider — Streaming Data Source Provider

`StreamSourceProvider` is the contract for objects that can create a streaming data source for a format (e.g. text file) or system (e.g. Apache Kafka) by their short names.

`StreamSourceProvider` is used when `DataSource` is requested for the name and schema of a streaming source or just creates one.

Table 1. Streaming Source Providers (in alphabetical order)

Name	Description
KafkaSourceProvider	Creates KafkaSourceProvider for `kafka` format.
TextSocketSourceProvider	Creates TextSocketSources for `socket` format.

StreamSourceProvider Contract

```
trait StreamSourceProvider {
  def sourceSchema(
      sqlContext: SQLContext,
      schema: Option[StructType],
      providerName: String,
      parameters: Map[String, String]): (String, StructType)
  def createSource(
      sqlContext: SQLContext,
      metadataPath: String,
      schema: Option[StructType],
      providerName: String,
      parameters: Map[String, String]): Source
}
```

Note	`StreamSourceProvider` is an experimental contract.

Table 2. StreamSourceProvider Contract (in alphabetical order)

<table>
<tr><th>Method</th><th>Description</th></tr>
<tr><td><code>createSource</code></td><td>Creates a streaming source for a format or system (to continually read data).

Note: <code>metadataPath</code> is the value of the optional user-specified <code>checkpointLocation</code> option or resolved by StreamingQueryManager.

Used exclusively when Spark SQL's <code>DataSource</code> is requested for a <code>Source</code> for a <code>StreamSourceProvider</code> (which is when <code>StreamingRelation</code> is requested for a logical plan).

Tip: Read DataSource — Pluggable Data Sources.</td></tr>
<tr><td><code>sourceSchema</code></td><td>Defines the name and the schema of a streaming source</td></tr>
</table>

Streaming Data Source

Streaming Data Source is a "continuous" stream of data and is described using the Source Contract.

`Source` can generate a streaming DataFrame (aka **batch**) given start and end offsets in a batch.

For fault tolerance, `Source` must be able to replay data given a start offset.

`Source` should be able to replay an arbitrary sequence of past data in a stream using a range of offsets. Streaming sources like Apache Kafka and Amazon Kinesis (with their per-record offsets) fit into this model nicely. This is the assumption so structured streaming can achieve end-to-end exactly-once guarantees.

Table 1. Sources

Format	Source
Any `FileFormat` • `csv` • `hive` • `json` • `libsvm` • `orc` • `parquet` • `text`	FileStreamSource
`kafka`	KafkaSource
`memory`	MemoryStream
`rate`	RateStreamSource
`socket`	TextSocketSource

Source Contract

```
package org.apache.spark.sql.execution.streaming

trait Source {
  def commit(end: Offset) : Unit = {}
  def getBatch(start: Option[Offset], end: Offset): DataFrame
  def getOffset: Option[Offset]
  def schema: StructType
  def stop(): Unit
}
```

Table 2. Source Contract

Method	Description
`getBatch`	Generates a `DataFrame` (with new rows) for a given batch (described using the optional start and end offsets). Used when `StreamExecution` runs a batch and populateStartOffsets.
`getOffset`	Finding the latest offset Note: Offset is…FIXME Used exclusively when `StreamExecution` runs streaming batches (and constructing the next streaming batch for every streaming data source in a streaming Dataset)
`schema`	Schema of the data from this source Used when: • `KafkaSource` generates a DataFrame with records from Kafka for a streaming batch • `FileStreamSource` generates a DataFrame for a streaming batch • `RateStreamSource` generates a DataFrame for a streaming batch • `StreamingExecutionRelation` is created (for MemoryStream)

StreamSinkProvider

`StreamSinkProvider` is the contract for creating streaming sinks for a specific format or system.

`StreamSinkProvider` defines the one and only createSink method that creates a streaming sink.

```
package org.apache.spark.sql.sources

trait StreamSinkProvider {
  def createSink(
      sqlContext: SQLContext,
      parameters: Map[String, String],
      partitionColumns: Seq[String],
      outputMode: OutputMode): Sink
}
```

Streaming Sink — Adding Batches of Data to Storage

`Sink` is the contract for **streaming writes**, i.e. adding batches to an output every trigger.

Note	`Sink` is part of the so-called **Structured Streaming V1** that is currently being rewritten to StreamWriteSupport in V2.

`Sink` is a single-method interface with `addBatch` method.

```
package org.apache.spark.sql.execution.streaming

trait Sink {
  def addBatch(batchId: Long, data: DataFrame): Unit
}
```

`addBatch` is used to "add" a batch of data to the sink (for `batchId` batch).

`addBatch` is used when `StreamExecution` runs a batch.

Table 1. Sinks

Format / Operator	Sink
`console`	ConsoleSink
Any `FileFormat` • `csv` • `hive` • `json` • `libsvm` • `orc` • `parquet` • `text`	FileStreamSink
foreach operator	ForeachSink
`kafka`	KafkaSink
`memory`	MemorySink

Tip	You can create your own streaming format implementing StreamSinkProvider.

When creating a custom `Sink` it is recommended to accept the options (e.g. `Map[String, String]`) that the `DataStreamWriter` was configured with. You can then use the options to fine-tune the write path.

```
class HighPerfSink(options: Map[String, String]) extends Sink {
  override def addBatch(batchId: Long, data: DataFrame): Unit = {
    val bucketName = options.get("bucket").orNull
    ...
  }
}
```

ForeachSink

`ForeachSink` is a typed streaming sink that passes rows (of the type `T`) to ForeachWriter (one record at a time per partition).

Note	`ForeachSink` is assigned a `ForeachWriter` when `DataStreamWriter` is started.

`ForeachSink` is used exclusively in foreach operator.

```
val records = spark.
  readStream
  format("text").
  load("server-logs/*.out").
  as[String]

import org.apache.spark.sql.ForeachWriter
val writer = new ForeachWriter[String] {
  override def open(partitionId: Long, version: Long) = true
  override def process(value: String) = println(value)
  override def close(errorOrNull: Throwable) = {}
}

records.writeStream
  .queryName("server-logs processor")
  .foreach(writer)
  .start
```

Internally, `addBatch` (the only method from the Sink Contract) takes records from the input DataFrame (as `data`), transforms them to expected type `T` (of this `ForeachSink`) and (now as a Dataset) processes each partition.

```
addBatch(batchId: Long, data: DataFrame): Unit
```

`addBatch` then opens the constructor's ForeachWriter (for the current partition and the input batch) and passes the records to process (one at a time per partition).

Caution	FIXME Why does Spark track whether the writer failed or not? Why couldn't it `finally` and do `close`?

Caution	FIXME Can we have a constant for `"foreach"` for `source` in `DataStreamWriter`?

StreamWriteSupport

`StreamWriteSupport` is…FIXME

Demo: groupBy Streaming Aggregation with Append Output Mode

The following example code shows a groupBy streaming aggregation with Append output mode.

`Append` output mode requires that a streaming aggregation defines a watermark (using withWatermark operator) on at least one of the grouping expressions (directly or using window function).

Note	withWatermark operator has to be used before the aggregation operator (for the watermark to be used).

In `Append` output mode the current watermark level is used to:

1. Output saved state rows that became expired (as **Expired state** in the below events table)
2. Drop late events, i.e. don't save them to a state store or include in aggregation (as **Late events** in the below events table)

Note	Sorting is only supported on streaming aggregated Datasets with `Complete` output mode.

Table 1. Streaming Batches, Events, Watermark and State Rows

<table>
<tr><th>Batch / Events</th><th>Current Watermark Level [ms]</th><th>Expired State, Late Events and Saved State Rows</th></tr>
<tr><td><table><tr><th>event_time</th><th>id</th><th>batch</th></tr><tr><td>1</td><td>1</td><td>1</td></tr><tr><td>15</td><td>2</td><td>1</td></tr></table></td><td>0</td><td>Saved State Rows<table><tr><th>event_time</th><th>id</th><th>batch</th></tr><tr><td>1</td><td>1</td><td>1</td></tr><tr><td>15</td><td>2</td><td>1</td></tr></table></td></tr>
<tr><td></td><td></td><td>Expired State<table><tr><th>event_time</th><th>id</th><th>batch</th></tr><tr><td>1</td><td>1</td><td>1</td></tr></table></td></tr>
</table>

<table>
<tr>
<td>
<table>
<tr><th>event_time</th><th>id</th><th>batch</th></tr>
<tr><td>1</td><td>1</td><td>2</td></tr>
<tr><td>15</td><td>2</td><td>2</td></tr>
<tr><td>35</td><td>3</td><td>2</td></tr>
</table>
</td>
<td>
5000

(Maximum event time <code>15</code> minus the <code>delayThreshold</code> as defined using withWatermark operator, i.e. <code>10</code>)
</td>
<td>
Late Events
<table>
<tr><th>event_time</th><th>id</th><th>batch</th></tr>
<tr><td>1</td><td>1</td><td>2</td></tr>
</table>
Saved State Rows
<table>
<tr><th>event_time</th><th>id</th><th>batch</th></tr>
<tr><td>15</td><td>2</td><td>1</td></tr>
<tr><td>15</td><td>2</td><td>2</td></tr>
<tr><td>35</td><td>3</td><td>2</td></tr>
</table>
</td>
</tr>
<tr>
<td>
<table>
<tr><th>event_time</th><th>id</th><th>batch</th></tr>
<tr><td>15</td><td>1</td><td>3</td></tr>
<tr><td>15</td><td>2</td><td>3</td></tr>
<tr><td>20</td><td>3</td><td>3</td></tr>
<tr><td>26</td><td>4</td><td>3</td></tr>
</table>
</td>
<td>
25000

(Maximum event time from the previous batch is <code>35</code> and <code>10</code> seconds of <code>delayThreshold</code>)
</td>
<td>
Expired State
<table>
<tr><th>event_time</th><th>id</th><th>batch</th></tr>
<tr><td>15</td><td>2</td><td>1</td></tr>
<tr><td>15</td><td>2</td><td>2</td></tr>
</table>
Late Events
<table>
<tr><th>event_time</th><th>id</th><th>batch</th></tr>
<tr><td>15</td><td>1</td><td>3</td></tr>
<tr><td>15</td><td>2</td><td>3</td></tr>
<tr><td>20</td><td>3</td><td>3</td></tr>
</table>
Saved State Rows
<table>
<tr><th>event_time</th><th>id</th><th>batch</th></tr>
<tr><td>35</td><td>3</td><td>2</td></tr>
<tr><td>26</td><td>4</td><td>3</td></tr>
</table>
</td>
</tr>
</table>

<table>
<tr>
<td>
<table>
<tr><th>event_time</th><th>id</th><th>batch</th></tr>
<tr><td>36</td><td>1</td><td>4</td></tr>
</table>
</td>
<td>

25000

(Maximum event time from the previous batch is `26`)
</td>
<td>

Saved State Rows

<table>
<tr><th>event_time</th><th>id</th><th>batch</th></tr>
<tr><td>35</td><td>3</td><td>2</td></tr>
<tr><td>26</td><td>4</td><td>3</td></tr>
<tr><td>36</td><td>1</td><td>4</td></tr>
</table>
</td>
</tr>
<tr>
<td>
<table>
<tr><th>event_time</th><th>id</th><th>batch</th></tr>
<tr><td>50</td><td>1</td><td>5</td></tr>
</table>
</td>
<td>

26000

(Maximum event time from the previous batch is `36`)
</td>
<td>

Expired State

<table>
<tr><th>event_time</th><th>id</th><th>batch</th></tr>
<tr><td>26</td><td>4</td><td>3</td></tr>
</table>

Saved State Rows

<table>
<tr><th>event_time</th><th>id</th><th>batch</th></tr>
<tr><td>35</td><td>3</td><td>2</td></tr>
<tr><td>36</td><td>1</td><td>4</td></tr>
<tr><td>50</td><td>1</td><td>5</td></tr>
</table>
</td>
</tr>
</table>

Note	Event time watermark may advance based on the maximum event time from the previous events (from the previous batch exactly as the level advances every trigger so the earlier levels are already counted in).

Note	Event time watermark can only change when the maximum event time is bigger than the current watermark minus the `delayThreshold` (as defined using withWatermark operator).

Tip	Use the following to publish events to Kafka.
	
	`// 1st streaming batch`
	`$ cat /tmp/1`
	`1,1,1`
	`15,2,1`
	
	`$ kafkacat -P -b localhost:9092 -t topic1 -l /tmp/1`
	
	`// Alternatively (and slower due to JVM bootup)`
	`$ cat /tmp/1 | ./bin/kafka-console-producer.sh --topic topic1 --broker-list loca`

```
/**
 * Reading datasets with records from a Kafka topic
 */
/**
TIP (only when working with SNAPSHOT version)
Remove the SNAPSHOT package from the local cache
rm -rf \
  ~/.ivy2/cache/org.apache.spark \
  ~/.ivy2/jars/org.apache.spark_spark-sql-kafka-0-10_2.11-2.3.0-SNAPSHOT.jar
*/

/**
TIP: Start spark-shell with spark-sql-kafka-0-10 package
./bin/spark-shell --packages org.apache.spark:spark-sql-kafka-0-10_2.11:2.3.0-SNAPSHOT
*/

/**
TIP: Copy the following code to append.txt and use :load command in spark-shell to loa
d it
:load append.txt
*/

// START: Only for easier debugging
// The state is then only for one partition
// which should make monitoring it easier
import org.apache.spark.sql.internal.SQLConf.SHUFFLE_PARTITIONS
spark.sessionState.conf.setConf(SHUFFLE_PARTITIONS, 1)
scala> spark.sessionState.conf.numShufflePartitions
res1: Int = 1
// END: Only for easier debugging

// Use streaming aggregation with groupBy operator to have StateStoreSaveExec operator
// Since the demo uses Append output mode
// it has to define a streaming event time watermark using withWatermark operator
// UnsupportedOperationChecker makes sure that the requirement holds
val idsPerBatch = spark.
  readStream.
  format("kafka").
  option("subscribe", "topic1").
  option("kafka.bootstrap.servers", "localhost:9092").
```

```
  load.
  withColumn("tokens", split('value, ",")).
  withColumn("seconds", 'tokens(0) cast "long").
  withColumn("event_time", to_timestamp(from_unixtime('seconds))). // <-- Event time h
as to be a timestamp
  withColumn("id", 'tokens(1)).
  withColumn("batch", 'tokens(2) cast "int").
  withWatermark(eventTime = "event_time", delayThreshold = "10 seconds"). // <-- defin
e watermark (before groupBy!)
  groupBy($"event_time"). // <-- use event_time for grouping
  agg(collect_list("batch") as "batches", collect_list("id") as "ids").
  withColumn("event_time", to_timestamp($"event_time")) // <-- convert to human-readab
le date

// idsPerBatch is a streaming Dataset with just one Kafka source
// so it knows nothing about output mode or the current streaming watermark yet
// - Output mode is defined on writing side
// - streaming watermark is read from rows at runtime
// That's why StatefulOperatorStateInfo is generic (and uses the default Append for ou
tput mode)
// and no batch-specific values are printed out
// They will be available right after the first streaming batch
// Use explain on a streaming query to know the trigger-specific values
scala> idsPerBatch.explain
== Physical Plan ==
*Project [event_time#36-T10000ms AS event_time#97, batches#90, ids#92]
+- ObjectHashAggregate(keys=[event_time#36-T10000ms], functions=[collect_list(batch#61
, 0, 0), collect_list(id#48, 0, 0)])
   +- Exchange hashpartitioning(event_time#36-T10000ms, 1)
      +- StateStoreSave [event_time#36-T10000ms], StatefulOperatorStateInfo(<unknown>,7
c5641eb-8ff9-447b-b9ba-b347c057d08f,0,0), Append, 0
         +- ObjectHashAggregate(keys=[event_time#36-T10000ms], functions=[merge_collec
t_list(batch#61, 0, 0), merge_collect_list(id#48, 0, 0)])
            +- Exchange hashpartitioning(event_time#36-T10000ms, 1)
               +- StateStoreRestore [event_time#36-T10000ms], StatefulOperatorStateInfo
(<unknown>,7c5641eb-8ff9-447b-b9ba-b347c057d08f,0,0)
                  +- ObjectHashAggregate(keys=[event_time#36-T10000ms], functions=[mer
ge_collect_list(batch#61, 0, 0), merge_collect_list(id#48, 0, 0)])
                     +- Exchange hashpartitioning(event_time#36-T10000ms, 1)
                        +- ObjectHashAggregate(keys=[event_time#36-T10000ms], function
s=[partial_collect_list(batch#61, 0, 0), partial_collect_list(id#48, 0, 0)])
                           +- EventTimeWatermark event_time#36: timestamp, interval 10
 seconds
                              +- *Project [cast(from_unixtime(cast(split(cast(value#1
as string), ,)[0] as bigint), yyyy-MM-dd HH:mm:ss, Some(Europe/Berlin)) as timestamp)
AS event_time#36, split(cast(value#1 as string), ,)[1] AS id#48, cast(split(cast(value#
1 as string), ,)[2] as int) AS batch#61]
                                 +- StreamingRelation kafka, [key#0, value#1, topic#2,
 partition#3, offset#4L, timestamp#5, timestampType#6]

// Start the query and hence StateStoreSaveExec
// Note Append output mode
import scala.concurrent.duration._
```

```
import org.apache.spark.sql.streaming.{OutputMode, Trigger}
val sq = idsPerBatch.
  writeStream.
  format("console").
  option("truncate", false).
  trigger(Trigger.ProcessingTime(5.seconds)).
  outputMode(OutputMode.Append). // <-- Append output mode
  start

-------------------------------------------
Batch: 0
-------------------------------------------
+----------+-------+---+
|event_time|batches|ids|
+----------+-------+---+
+----------+-------+---+

// there's only 1 stateful operator and hence 0 for the index in stateOperators
scala> println(sq.lastProgress.stateOperators(0).prettyJson)
{
  "numRowsTotal" : 0,
  "numRowsUpdated" : 0,
  "memoryUsedBytes" : 77
}

// Current watermark
// We've just started so it's the default start time
scala> println(sq.lastProgress.eventTime.get("watermark"))
1970-01-01T00:00:00.000Z

-------------------------------------------
Batch: 1
-------------------------------------------
+----------+-------+---+
|event_time|batches|ids|
+----------+-------+---+
+----------+-------+---+

// it's Append output mode so numRowsTotal is...FIXME
// no keys were available earlier (it's just started!) and so numRowsUpdated is 0
scala> println(sq.lastProgress.stateOperators(0).prettyJson)
{
  "numRowsTotal" : 2,
  "numRowsUpdated" : 2,
  "memoryUsedBytes" : 669
}

// Current watermark
// One streaming batch has passed so it's still the default start time
// that will get changed the next streaming batch
// watermark is always one batch behind
scala> println(sq.lastProgress.eventTime.get("watermark"))
1970-01-01T00:00:00.000Z
```

```
// Could be 0 if the time to update the lastProgress is short
// FIXME Explain it in detail
scala> println(sq.lastProgress.numInputRows)
2

-------------------------------------------
Batch: 2
-------------------------------------------
+-------------------+-------+---+
|event_time         |batches|ids|
+-------------------+-------+---+
|1970-01-01 01:00:01|[1]    |[1]|
+-------------------+-------+---+

scala> println(sq.lastProgress.stateOperators(0).prettyJson)
{
  "numRowsTotal" : 2,
  "numRowsUpdated" : 2,
  "memoryUsedBytes" : 701
}

// Current watermark
// Updated and so the output with the final aggregation (aka expired state)
scala> println(sq.lastProgress.eventTime.get("watermark"))
1970-01-01T00:00:05.000Z

scala> println(sq.lastProgress.numInputRows)
3

-------------------------------------------
Batch: 3
-------------------------------------------
+-------------------+-------+------+
|event_time         |batches|ids   |
+-------------------+-------+------+
|1970-01-01 01:00:15|[2, 1] |[2, 2]|
+-------------------+-------+------+

scala> println(sq.lastProgress.stateOperators(0).prettyJson)
{
  "numRowsTotal" : 2,
  "numRowsUpdated" : 1,
  "memoryUsedBytes" : 685
}

// Current watermark
// Updated and so the output with the final aggregation (aka expired state)
scala> println(sq.lastProgress.eventTime.get("watermark"))
1970-01-01T00:00:25.000Z

scala> println(sq.lastProgress.numInputRows)
4
```

```
-------------------------------------------
Batch: 4
-------------------------------------------
+----------+-------+---+
|event_time|batches|ids|
+----------+-------+---+
+----------+-------+---+

scala> println(sq.lastProgress.stateOperators(0).prettyJson)
{
  "numRowsTotal" : 3,
  "numRowsUpdated" : 1,
  "memoryUsedBytes" : 965
}

scala> println(sq.lastProgress.eventTime.get("watermark"))
1970-01-01T00:00:25.000Z

scala> println(sq.lastProgress.numInputRows)
1

// publish new records
// See the events table above

-------------------------------------------
Batch: 5
-------------------------------------------
+-------------------+-------+---+
|event_time         |batches|ids|
+-------------------+-------+---+
|1970-01-01 01:00:26|[3]    |[4]|
+-------------------+-------+---+

scala> println(sq.lastProgress.stateOperators(0).prettyJson)
{
  "numRowsTotal" : 3,
  "numRowsUpdated" : 1,
  "memoryUsedBytes" : 997
}

// Current watermark
// Updated and so the output with the final aggregation (aka expired state)
scala> println(sq.lastProgress.eventTime.get("watermark"))
1970-01-01T00:00:26.000Z

scala> println(sq.lastProgress.numInputRows)
1

// In the end...
sq.stop
```

Demo: Developing Custom Streaming Sink (and Monitoring SQL Queries in web UI)

The demo shows the steps to develop a custom streaming sink and use it to monitor whether and what SQL queries are executed at runtime (using web UI's SQL tab).

Note	The main motivation was to answer the question Why does a single structured query run multiple SQL queries per batch? that happened to have turned out fairly surprising. You're *very* welcome to upvote the question and answers at your earliest convenience. Thanks!

The steps are as follows:

1. Creating Custom Sink — DemoSink
2. Creating StreamSinkProvider — DemoSinkProvider
3. Optional Sink Registration using META-INF/services
4. build.sbt Definition
5. Packaging DemoSink
6. Using DemoSink in Streaming Query
7. Monitoring SQL Queries using web UI's SQL Tab

Findings (aka *surprises*):

1. Custom sinks require that you define a checkpoint location using checkpointLocation option (or spark.sql.streaming.checkpointLocation Spark property). Remove the checkpoint directory (or use a different one every start of a streaming query) to have consistent results.

Creating Custom Sink — DemoSink

A streaming sink follows the Sink contract and a sample implementation could look as follows.

```
package pl.japila.spark.sql.streaming

case class DemoSink(
  sqlContext: SQLContext,
  parameters: Map[String, String],
  partitionColumns: Seq[String],
  outputMode: OutputMode) extends Sink {

  override def addBatch(batchId: Long, data: DataFrame): Unit = {
    println(s"addBatch($batchId)")
    data.explain()
    // Why so many lines just to show the input DataFrame?
    data.sparkSession.createDataFrame(
      data.sparkSession.sparkContext.parallelize(data.collect()), data.schema)
      .show(10)
  }
}
```

Save the file under `src/main/scala` in your project.

Creating StreamSinkProvider — DemoSinkProvider

```
package pl.japila.spark.sql.streaming

class DemoSinkProvider extends StreamSinkProvider
  with DataSourceRegister {

  override def createSink(
    sqlContext: SQLContext,
    parameters: Map[String, String],
    partitionColumns: Seq[String],
    outputMode: OutputMode): Sink = {
    DemoSink(sqlContext, parameters, partitionColumns, outputMode)
  }

  override def shortName(): String = "demo"
}
```

Save the file under `src/main/scala` in your project.

Optional Sink Registration using META-INF/services

The step is optional, but greatly improve the experience when using the custom sink so you can use it by its name (rather than a fully-qualified class name or using a special class name for the sink provider).

Create `org.apache.spark.sql.sources.DataSourceRegister` in `META-INF/services` directory with the following content.

```
pl.japila.spark.sql.streaming.DemoSinkProvider
```

Save the file under `src/main/resources` in your project.

build.sbt Definition

If you use my beloved build tool sbt to manage the project, use the following `build.sbt`.

```
organization := "pl.japila.spark"
name := "spark-structured-streaming-demo-sink"
version := "0.1"

scalaVersion := "2.11.11"

libraryDependencies += "org.apache.spark" %% "spark-sql" % "2.2.0"
```

Packaging DemoSink

The step depends on what build tool you use to manage the project. Use whatever command you use to create a jar file with the above classes compiled and bundled together.

```
$ sbt package
[info] Loading settings from plugins.sbt ...
[info] Loading project definition from /Users/jacek/dev/sandbox/spark-structured-strea
ming-demo-sink/project
[info] Loading settings from build.sbt ...
[info] Set current project to spark-structured-streaming-demo-sink (in build file:/Use
rs/jacek/dev/sandbox/spark-structured-streaming-demo-sink/)
[info] Compiling 1 Scala source to /Users/jacek/dev/sandbox/spark-structured-streaming
-demo-sink/target/scala-2.11/classes ...
[info] Done compiling.
[info] Packaging /Users/jacek/dev/sandbox/spark-structured-streaming-demo-sink/target/
scala-2.11/spark-structured-streaming-demo-sink_2.11-0.1.jar ...
[info] Done packaging.
[success] Total time: 5 s, completed Sep 12, 2017 9:34:19 AM
```

The jar with the sink is `/Users/jacek/dev/sandbox/spark-structured-streaming-demo-sink/target/scala-2.11/spark-structured-streaming-demo-sink_2.11-0.1.jar`.

Using DemoSink in Streaming Query

The following code reads data from the `rate` source and simply outputs the result to our custom `DemoSink`.

```
// Make sure the DemoSink jar is available
$ ls /Users/jacek/dev/sandbox/spark-structured-streaming-demo-sink/target/scala-2.11/s
park-structured-streaming-demo-sink_2.11-0.1.jar
/Users/jacek/dev/sandbox/spark-structured-streaming-demo-sink/target/scala-2.11/spark-
structured-streaming-demo-sink_2.11-0.1.jar

// "Install" the DemoSink using --jars command-line option
$ ./bin/spark-shell --jars /Users/jacek/dev/sandbox/spark-structured-streaming-custom-
sink/target/scala-2.11/spark-structured-streaming-custom-sink_2.11-0.1.jar

scala> spark.version
res0: String = 2.3.0-SNAPSHOT

import org.apache.spark.sql.streaming._
import scala.concurrent.duration._
val sq = spark.
  readStream.
  format("rate").
  load.
  writeStream.
  format("demo").
  option("checkpointLocation", "/tmp/demo-checkpoint").
  trigger(Trigger.ProcessingTime(10.seconds)).
  start

// In the end...
scala> sq.stop
17/09/12 09:59:28 INFO StreamExecution: Query [id = 03cd78e3-94e2-439c-9c12-cfed0c9968
12, runId = 6938af91-9806-4404-965a-5ae7525d5d3f] was stopped
```

Monitoring SQL Queries using web UI's SQL Tab

Open http://localhost:4040/SQL/.

You should find that every trigger (aka *batch*) results in 3 SQL queries. Why?

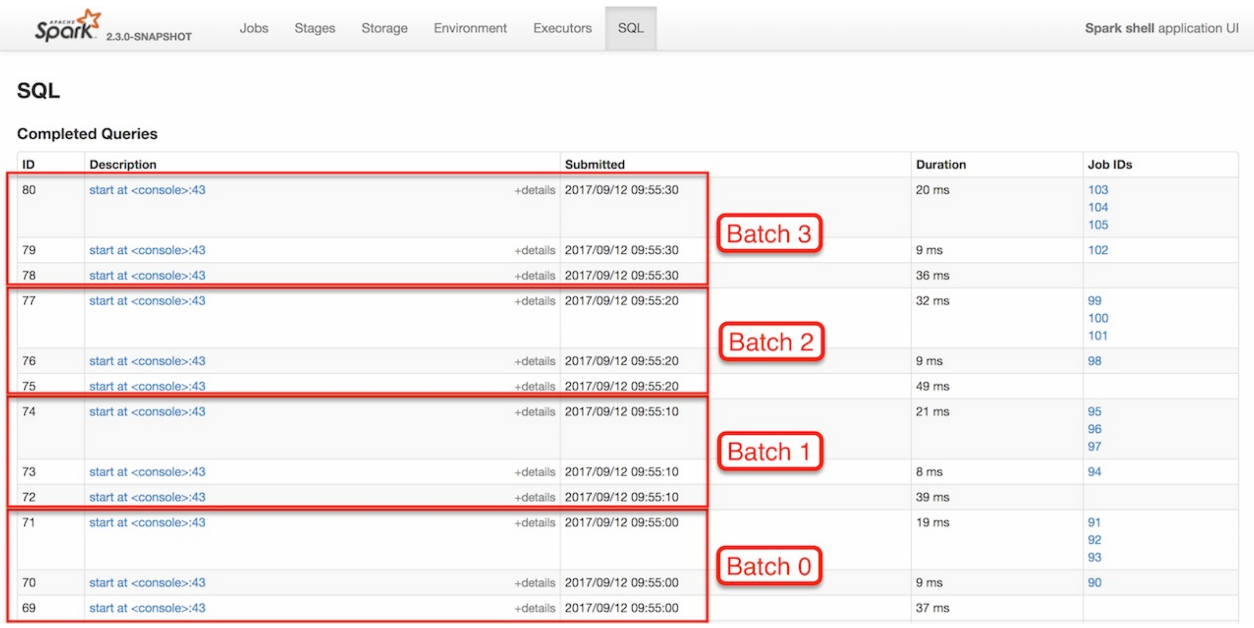

Figure 1. web UI's SQL Tab and Completed Queries (3 Queries per Batch)

The answer lies in what sources and sink a streaming query uses (and differs per streaming query).

In our case, DemoSink `collects` the rows from the input `DataFrame` and `shows` it afterwards. That gives 2 SQL queries (as you can see after executing the following batch queries).

```
// batch non-streaming query
val data = (0 to 3).toDF("id")

// That gives one SQL query
data.collect

// That gives one SQL query, too
data.show
```

The remaining query (which is the first among the queries) is executed when you `load` the data.

That can be observed easily when you change DemoSink to not "touch" the input `data` (in `addBatch`) in any way.

```
override def addBatch(batchId: Long, data: DataFrame): Unit = {
  println(s"addBatch($batchId)")
}
```

Re-run the streaming query (using the new `DemoSink`) and use web UI's SQL tab to see the queries. You should have just one query per batch (and no Spark jobs given nothing is really done in the sink's `addBatch`).

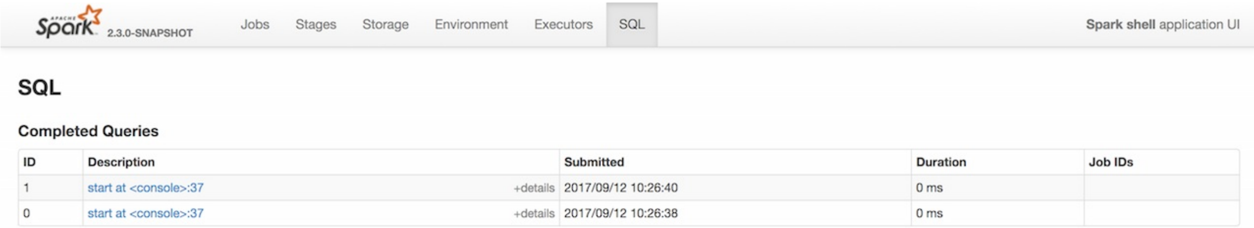

Figure 2. web UI's SQL Tab and Completed Queries (1 Query per Batch)

Demo: current_timestamp Function For Processing Time in Streaming Queries

The demo shows what happens when you use `current_timestamp` function in your structured queries.

Note	The main motivation was to answer the question How to achieve ingestion time? in Spark Structured Streaming. You're *very* welcome to upvote the question and answers at your earliest convenience. Thanks!

Quoting the Apache Flink documentation:

> **Event time** is the time that each individual event occurred on its producing device. This time is typically embedded within the records before they enter Flink and that event timestamp can be extracted from the record.

That is exactly how event time is considered in withWatermark operator which you use to describe what column to use for event time. The column could be part of the input dataset or…generated.

And that is the moment where my confusion starts.

In order to generate the event time column for `withWatermark` operator you could use `current_timestamp` or `current_date` standard functions.

```
// rate format gives event time
// but let's generate a brand new column with ours
// for demo purposes
val values = spark.
  readStream.
  format("rate").
  load.
  withColumn("current_timestamp", current_timestamp)
scala> values.printSchema
root
 |-- timestamp: timestamp (nullable = true)
 |-- value: long (nullable = true)
 |-- current_timestamp: timestamp (nullable = false)
```

Both are special for Spark Structured Streaming as `StreamExecution` replaces their underlying Catalyst expressions, `CurrentTimestamp` and `CurrentDate` respectively, with `CurrentBatchTimestamp` expression and the time of the current batch.

```
import org.apache.spark.sql.streaming.Trigger
import scala.concurrent.duration._
val sq = values.
  writeStream.
  format("console").
  option("truncate", false).
  trigger(Trigger.ProcessingTime(10.seconds)).
  start

// note the value of current_timestamp
// that corresponds to the batch time

-------------------------------------------
Batch: 1
-------------------------------------------
+-----------------------+-----+-------------------+
|timestamp              |value|current_timestamp  |
+-----------------------+-----+-------------------+
|2017-09-18 10:53:31.523|0    |2017-09-18 10:53:40|
|2017-09-18 10:53:32.523|1    |2017-09-18 10:53:40|
|2017-09-18 10:53:33.523|2    |2017-09-18 10:53:40|
|2017-09-18 10:53:34.523|3    |2017-09-18 10:53:40|
|2017-09-18 10:53:35.523|4    |2017-09-18 10:53:40|
|2017-09-18 10:53:36.523|5    |2017-09-18 10:53:40|
|2017-09-18 10:53:37.523|6    |2017-09-18 10:53:40|
|2017-09-18 10:53:38.523|7    |2017-09-18 10:53:40|
+-----------------------+-----+-------------------+

// Use web UI's SQL tab for the batch (Submitted column)
// or sq.recentProgress
scala> println(sq.recentProgress(1).timestamp)
2017-09-18T08:53:40.000Z

// Note current_batch_timestamp

scala> sq.explain(extended = true)
== Parsed Logical Plan ==
'Project [timestamp#2137, value#2138L, current_batch_timestamp(1505725650005, Timestam
pType, None) AS current_timestamp#50]
+- LogicalRDD [timestamp#2137, value#2138L], true

== Analyzed Logical Plan ==
timestamp: timestamp, value: bigint, current_timestamp: timestamp
Project [timestamp#2137, value#2138L, current_batch_timestamp(1505725650005, Timestamp
Type, Some(Europe/Berlin)) AS current_timestamp#50]
+- LogicalRDD [timestamp#2137, value#2138L], true

== Optimized Logical Plan ==
Project [timestamp#2137, value#2138L, 1505725650005000 AS current_timestamp#50]
+- LogicalRDD [timestamp#2137, value#2138L], true

== Physical Plan ==
```

```
*Project [timestamp#2137, value#2138L, 1505725650005000 AS current_timestamp#50]
+- Scan ExistingRDD[timestamp#2137,value#2138L]
```

That *seems* to be closer to processing time than ingestion time given the definition from the Apache Flink documentation:

> **Processing time** refers to the system time of the machine that is executing the respective operation.

> **Ingestion time** is the time that events enter Flink.

What do you think?

Demo: Using StreamingQueryManager for Query Termination Management

The demo shows how to use StreamingQueryManager (and specifically awaitAnyTermination and resetTerminated) for query termination management.

demo-StreamingQueryManager.scala

```
// Save the code as demo-StreamingQueryManager.scala
// Start it using spark-shell
// $ ./bin/spark-shell -i demo-StreamingQueryManager.scala

// Register a StreamingQueryListener to receive notifications about state changes of s
treaming queries
import org.apache.spark.sql.streaming.StreamingQueryListener
val myQueryListener = new StreamingQueryListener {
  import org.apache.spark.sql.streaming.StreamingQueryListener._
  def onQueryTerminated(event: QueryTerminatedEvent): Unit = {
    println(s"Query ${event.id} terminated")
  }

  def onQueryStarted(event: QueryStartedEvent): Unit = {}
  def onQueryProgress(event: QueryProgressEvent): Unit = {}
}
spark.streams.addListener(myQueryListener)

import org.apache.spark.sql.streaming._
import scala.concurrent.duration._

// Start streaming queries

// Start the first query
val q4s = spark.readStream.
  format("rate").
  load.
  writeStream.
  format("console").
  trigger(Trigger.ProcessingTime(4.seconds)).
  option("truncate", false).
  start

// Start another query that is slightly slower
val q10s = spark.readStream.
  format("rate").
  load.
  writeStream.
  format("console").
  trigger(Trigger.ProcessingTime(10.seconds)).
  option("truncate", false).
```

```
  start

// Both queries run concurrently
// You should see different outputs in the console
// q4s prints out 4 rows every batch and twice as often as q10s
// q10s prints out 10 rows every batch

/*
-------------------------------------------
Batch: 7
-------------------------------------------
+-----------------------+-----+
|timestamp              |value|
+-----------------------+-----+
|2017-10-27 13:44:07.462|21   |
|2017-10-27 13:44:08.462|22   |
|2017-10-27 13:44:09.462|23   |
|2017-10-27 13:44:10.462|24   |
+-----------------------+-----+

-------------------------------------------
Batch: 8
-------------------------------------------
+-----------------------+-----+
|timestamp              |value|
+-----------------------+-----+
|2017-10-27 13:44:11.462|25   |
|2017-10-27 13:44:12.462|26   |
|2017-10-27 13:44:13.462|27   |
|2017-10-27 13:44:14.462|28   |
+-----------------------+-----+

-------------------------------------------
Batch: 2
-------------------------------------------
+-----------------------+-----+
|timestamp              |value|
+-----------------------+-----+
|2017-10-27 13:44:09.847|6    |
|2017-10-27 13:44:10.847|7    |
|2017-10-27 13:44:11.847|8    |
|2017-10-27 13:44:12.847|9    |
|2017-10-27 13:44:13.847|10   |
|2017-10-27 13:44:14.847|11   |
|2017-10-27 13:44:15.847|12   |
|2017-10-27 13:44:16.847|13   |
|2017-10-27 13:44:17.847|14   |
|2017-10-27 13:44:18.847|15   |
+-----------------------+-----+
*/

// Stop q4s on a separate thread
// as we're about to block the current thread awaiting query termination
```

```
import java.util.concurrent.Executors
import java.util.concurrent.TimeUnit.SECONDS
def queryTerminator(query: StreamingQuery) = new Runnable {
  def run = {
    println(s"Stopping streaming query: ${query.id}")
    query.stop
  }
}
import java.util.concurrent.TimeUnit.SECONDS
// Stop the first query after 10 seconds
Executors.newSingleThreadScheduledExecutor.
  scheduleWithFixedDelay(queryTerminator(q4s), 10, 60 * 5, SECONDS)
// Stop the other query after 20 seconds
Executors.newSingleThreadScheduledExecutor.
  scheduleWithFixedDelay(queryTerminator(q10s), 20, 60 * 5, SECONDS)

// Use StreamingQueryManager to wait for any query termination (either q1 or q2)
// the current thread will block indefinitely until either streaming query has finished

spark.streams.awaitAnyTermination

// You are here only after either streaming query has finished
// Executing spark.streams.awaitAnyTermination again would return immediately

// You should have received the QueryTerminatedEvent for the query termination

// reset the last terminated streaming query
spark.streams.resetTerminated

// You know at least one query has terminated

// Wait for the other query to terminate
spark.streams.awaitAnyTermination

assert(spark.streams.active.isEmpty)

println("The demo went all fine. Exiting...")

// leave spark-shell
System.exit(0)
```

UnsupportedOperationChecker

`UnsupportedOperationChecker` checks whether the logical plan of a streaming query uses supported operations only.

Note	`UnsupportedOperationChecker` is used exclusively when the internal spark.sql.streaming.unsupportedOperationCheck Spark property is enabled (which is by default).

Note	`UnsupportedOperationChecker` comes actually with two methods, i.e. `checkForBatch` and checkForStreaming, whose names reveal the different flavours of Spark SQL (as of 2.0), i.e. batch and streaming, respectively. The Spark Structured Streaming gitbook is solely focused on checkForStreaming method.

`checkForStreaming` Method

```
checkForStreaming(plan: LogicalPlan, outputMode: OutputMode): Unit
```

`checkForStreaming` asserts that the following requirements hold:

1. Only one streaming aggregation is allowed
2. Streaming aggregation with Append output mode requires watermark (on the grouping expressions)
3. Multiple flatMapGroupsWithState operators are only allowed with Append output mode

`checkForStreaming` …FIXME

`checkForStreaming` finds all streaming aggregates (i.e. `Aggregate` logical operators with streaming sources).

Note	`Aggregate` logical operator represents groupBy and groupByKey aggregations (and SQL's `GROUP BY` clause).

`checkForStreaming` asserts that there is exactly one streaming aggregation in a streaming query.

Otherwise, `checkForStreaming` reports a `AnalysisException` :

```
Multiple streaming aggregations are not supported with streaming
DataFrames/Datasets
```

`checkForStreaming` asserts that watermark was defined for a streaming aggregation with Append output mode (on at least one of the grouping expressions).

Otherwise, `checkForStreaming` reports a `AnalysisException` :

```
Append output mode not supported when there are streaming
aggregations on streaming DataFrames/DataSets without watermark
```

Caution	FIXME

`checkForStreaming` counts all FlatMapGroupsWithState logical operators (on streaming Datasets with `isMapGroupsWithState` flag disabled).

Note	FlatMapGroupsWithState logical operator represents mapGroupsWithState and flatMapGroupsWithState operators.

Note	FlatMapGroupsWithState.isMapGroupsWithState flag is disabled when… FIXME

`checkForStreaming` asserts that multiple FlatMapGroupsWithState logical operators are only used when:

- `outputMode` is Append output mode
- outputMode of the `FlatMapGroupsWithState` logical operators is also Append output mode

Caution	FIXME Reference to an example in `flatMapGroupsWithState`

Otherwise, `checkForStreaming` reports a `AnalysisException` :

```
Multiple flatMapGroupsWithStates are not supported when they are
not all in append mode or the output mode is not append on a
streaming DataFrames/Datasets
```

Caution	FIXME

Note	`checkForStreaming` is used exclusively when `StreamingQueryManager` is requested to create a StreamingQueryWrapper (for starting a streaming query), but only when the internal spark.sql.streaming.unsupportedOperationCheck Spark property is enabled (which is by default).

StreamProgress Custom Scala Map

`StreamProgress` is an extension of Scala's scala.collection.immutable.Map with streaming sources as keys and their `Offsets` as values.

`StreamProgress` is created when…MEFIXME

`StreamProgress` takes a `Map` of Offsets per `BaseStreamingSource` when created.