trainologic

Spark 2.3 Streaming

On Java

trainologic

Spark Streaming

1. Spark Streaming is an extension of the core Spark API
2. It enables scalable, high-throughput, fault-tolerant stream processing of live data streams.
3. Data can be ingested from many sources like Kafka, Flume, Kinesis, or TCP sockets
4. Data can be processed using complex algorithms expressed with high-level functions like map, reduce, join and window.
5. Finally, processed data can be pushed out to file systems, databases, and live dashboards.

→ In fact, you can apply Spark's machine learning and graph processing algorithms on data streams.

trainologic

Micro batching with Spark Streaming

- Takes a partitioned stream of data
- Slices it up by time – usually seconds
- DStream (discretized stream) - composed of RDD slices that contains a collection of items

Taken from official spark documentation

trainologic

Spark streaming application

- Define the time window when creating the StreamingContext.
- Define the input sources for the streaming pipeline.
- Combine transformations.
- Define the output operations.
- Invoke start().
- Wait for the computation to stop (awaitTermination()).

trainologic

Important

- Once *start()* was invoked, no additional operations can be added.
- A stopped context can’t be restarted.

trainologic

Quick Example

```
SparkConf conf = new SparkConf().setAppName("Streaming Test").setMaster("local[*]");
JavaStreamingContext jssc = new JavaStreamingContext(conf, Durations.seconds(15));

JavaDStream<String> dstream = jssc.textFileStream("<in path>");
JavaDStream<Integer> toIntsdStream = dstream.map(Integer::parseInt);
toIntsdStream.print();
jssc.start();
jssc.awaitTermination();
```

trainologic

DSTream operations

Similar to RDD operations with small changes

- **map(func)** - returns a new **DSTream** applying func on every element of the original stream.
- **flatMap(func)** - Similar to map, but each input item can be mapped to 0 or more output items.
- **filter(func)** - returns a new **DSTream** formed by selecting those elements of the source stream on which func returns true.
- **reduce(func)** – returns a new Dstream of **single-element RDDs** by applying the reduce func on every source RDD

→ This is pretty much the same of the RDD operations

trainologic

Using your existing batch logic

- **transform(func)** - operation that creates a new DStream by a applying func to DStream RDDs.

```
JavaReceiverInputDStream<String> lines2 = lines
.transform((Function2<JavaRDD<String>, Time, JavaRDD<Object>>) (v1, v2) ->
v1.union(ignored))
```

trainologic

Streaming - logs analyzes

trainologic

Streaming - logs analyzes

We would like to analyze log files.

To be specific, we would like to count how many log levels (“INFO”, “WARN”, “ERROR”, etc…) there is:

```
17/09/28 12:15:49 INFO TaskSetManager: Starting task 14.0 in stage 176.0 (TID 9009, 10.103.240.64, partition 14, PROCESS_LOCAL, 7085 bytes)
```

1. Create DStream on top of the relevant directory
2. Provide the transformations to count the log level
3. Also, count the numbers over time

→ In order spark streaming to collect file it should: be with new name.

trainologic

Input DStreams and Receivers

1. Input DStreams are DStreams representing the stream of input data received from streaming sources.
2. Every input DStream is associated with a Receiver.
3. Receiver is an object which receives the data from a source and stores it in Spark's memory for processing.
4. Spark Streaming provides two categories of built-in streaming sources.
   a. **Basic sources**: Sources directly available in the StreamingContext API. E.g.: file systems, and socket connections.
   b. **Advanced sources**: Sources like Kafka, Flume, Kinesis, etc. are available through extra utility classes. These require linking against extra dependencies as discussed in the linking section.

trainologic

Input DStreams and Receivers

1. It’s possible to create multiple streams of data in parallel in your streaming application
2. As streaming is a long-running application you’ll need to allocate enough cores
3. You can use streams that uses built-in receivers or create custom receiver
4. Examples of other recievers: Kafka, Flume, Kinesis and more...

trainologic

Streaming - few good questions

1. What about aggregations over time?
2. What about failures?

trainologic

Updating the state

- By default, there is nothing shared between window intervals.
- How do I accumulate results with the current batch?
- ***updateStateByKey(updateFunc) –*** a transformation that creates a new DStream with key-value where the value is updated according to the state and the new values.

trainologic

Updating the state

Updating function:

```
Function2<List<Integer>, Optional<Integer>, Optional<Integer>> updateFunction =
        (values, state) -> {Integer newSum = state.or(0);
            for (int i: values){newSum += i;}
            return Optional.of(newSum);
        };
```

Using it:

```
countWords.updateStateByKey(updateFunction);
```

trainologic

Checkpoints

- **Checkpoints** – periodically saves to reliable storage (HDFS/S3/…) necessary data to recover from failures
- **Metadata checkpoints**
  - Configuration of the stream context
  - DStream definition and operations
  - Incomplete batches
- **Data checkpoints**
  - saving stateful RDD data

trainologic

Checkpoints

Setting up checkpoint:

```
jssc.checkpoint("src/main/resources/streaming/checkpoints/");
```

Recovery:

```
JavaStreamingContext.getOrCreate("checkpointDirectory", "contextFactory");
```

trainologic

Working with the foreach RDD - take I

A common practice is to use the ***foreachRDD(func)*** to push data to an external system

```
Connection connection = new Connection("external connection");
lines.foreachRDD(rdd ->
      rdd.foreach(line ->
            connect.write(line))
);
```

→ The connection opened in the driver.

It'll most probably won't survive serialization and moving across the network

trainologic

Working with the foreach RDD - take II

How about this one:

```
lines.foreachRDD(rdd ->
      rdd.foreach(line ->
            Connection connection = new Connection("external connection");
            connect.write(line))
);
```

→ Will create connection for every line!

trainologic

Working with the foreach RDD - take III

How about this one:

```
lines.foreachRDD(rdd ->
            rdd.foreachPartition(part -> {
                Connection connection = new Connection("external connection");
                while (part.hasNext()){
                    connection.write(part.next());
                }
                connection.close();
            }
        );
```

→ Remember the mapPartition?

This can further optimize using connection pool

trainologic

Structured Streaming

- Structured Streaming is a scalable and fault-tolerant stream processing engine built on the Spark SQL engine.
- You can express your streaming computation the same way you would express a batch computation on static data.
- The Spark SQL engine will take care of running it incrementally and continuously and updating the final result as streaming data continues to arrive.
- You can use the Dataset/DataFrame API in Scala, Java, Python or R to express streaming aggregations, event-time windows, stream-to-batch joins, etc.
- The computation is executed on the same optimized Spark SQL engine.
- The system ensures end-to-end exactly-once fault-tolerance guarantees through checkpointing and Write Ahead Logs.

→ Taken from the official Spark documentation:
https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html

trainologic

Structured Streaming

- Expressing streaming computation the same way as expressed in a batch - uses the same API
- Supports many input and output sources such as:
  - Files
  - Sockets
  - Kafka
- Supports all the known Dataset operations
- Fault tolerance support

trainologic

Structured Streaming - Spark 2.3

By default, **Structured Streaming** queries are processed using a micro-batch processing engine:

1. Processes data streams as a series of small batch jobs
2. End-to-end latencies of about 100 milliseconds
3. Exactly-once fault-tolerance guarantees

Spark 2.3 introduced new mode called **Continuous Processing**

1. which can achieve end-to-end latencies as low as 1 millisecond
2. At-least-once guarantees

trainologic

Structured Streaming

Create stream:

```
SparkSession spark = SparkUtils.createSparkSession();

Dataset<Row> netLines = spark
      .readStream()
      .format("socket")
      .option("host", "localhost")
      .option("port", "9999")
      .load();
```

➔ Beside the *spark.readStream()* looks exactly like the batch api (*spark.read()*)

➔ The return type is DataFrame

trainologic

Structured Streaming

Some transformations:

```
Dataset<String> words = netLines.as(Encoders.STRING())
            .flatMap((FlatMapFunction<String, String>)
                    line -> Arrays.asList(line.split(" ")).iterator(),
                Encoders.STRING());

Dataset<Row> wcs = words.groupBy("value").count();
```

➔ Again, exact same code of the batch processing

trainologic

Structured Streaming

- Start stream
- And… Await termination...

```
StreamingQuery query = wcs.writeStream()
      .format("console")
      .outputMode("complete")
      .start();
query.awaitTermination();
```

trainologic

Structured Streaming

Using nc, you stream data into your program:

```
~$ nc -lk 9999
```

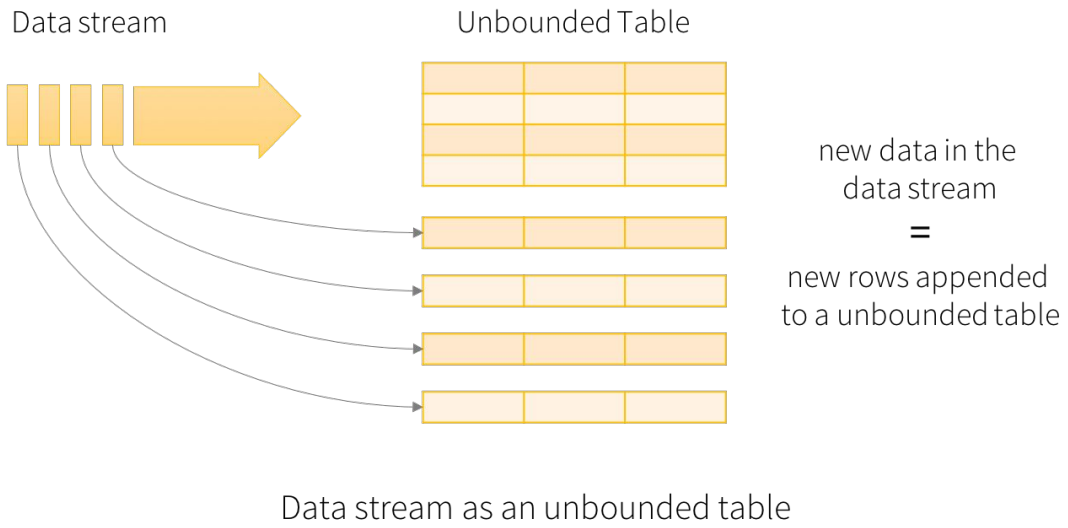
And that’s all!

Naaa, not really all... - let’s dive in a bit

trainologic

Structured Streaming - Programming Model

1. Treat live data stream as table that is continuously appended
2. Than, use regular, batched queries to query the table
3. Spark will run it as incremental query on the *unbounded table*
4. The results of query is written out
   to extrenal storage



Data stream as an unbounded table

→ **Taken from the official Spark documentation: https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html**

trainologic

Structured Streaming - Input Sources

- **File source** - Reads files written in a directory as a stream of data. Supported file formats are text, csv, json, orc, parquet.
  → Note that the files must be atomically placed in the given directory, which in most file systems, can be achieved by file move operations.
- **Kafka source** - Reads data from Kafka.
- **Socket source** (for testing) - Reads UTF8 text data from a socket connection. The listening server socket is at the driver.
  → Note that this should be used only for testing as this does not provide end-to-end fault-tolerance guarantees.
- **Rate source** (for testing) - Generates data at the specified number of rows per second
  This source is intended for testing and benchmarking.

trainologic

Structured Streaming - Output Mode

Append - the default mode

- Write out the NEW rows added since last trigger
- Supported for query results that WON'T change
- e.g. results of *map(), select(), join()* and more…

Complete

- Full result set will be outputted
- Support for aggregation queries

Updated

- New feature (starting 2.1.1)
- Only updated rows outputted

trainologic

Structured Streaming - Sinks

1. File sink - write output into director

```
StreamingQuery query = words
    .writeStream()
    .format("json")
    .option("checkpointLocation", "src/main/resources/output/streaming/check_point")
    .option("path", "src/main/resources/output/streaming/coplete_count.json")
    .outputMode("append")
    .start();
```

2. Kafka sink
3. For each sink - out of the scope
4. Console sink - mainly for debugging
5. Memory sink - for debugging, store the full results as table in the driver memory - use with caution!

trainologic

Structured Streaming - Sinks

Sink	Supported Modes	Options	Fault-tolerant	Notes
File Sink	Append	path: path to the output directory, must be specified. There is file-format-specific options	Yes (exactly-once)	Supports writes to partitioned tables. Partitioning by time may be useful.
Kafka Sink	Append, Update, Complete	There is many….	Yes (at-least-once)	
Console Sink	Append, Update, Complete	numRows: Number of rows to print every trigger (default: 20) truncate: Whether to truncate the output if too long (default: true)	No	
Memory Sink	Append, Complete	None	No. But in Complete Mode, restarted query will recreate the full table.	Table name is the query name.

trainologic

Structured Streaming - Triggering

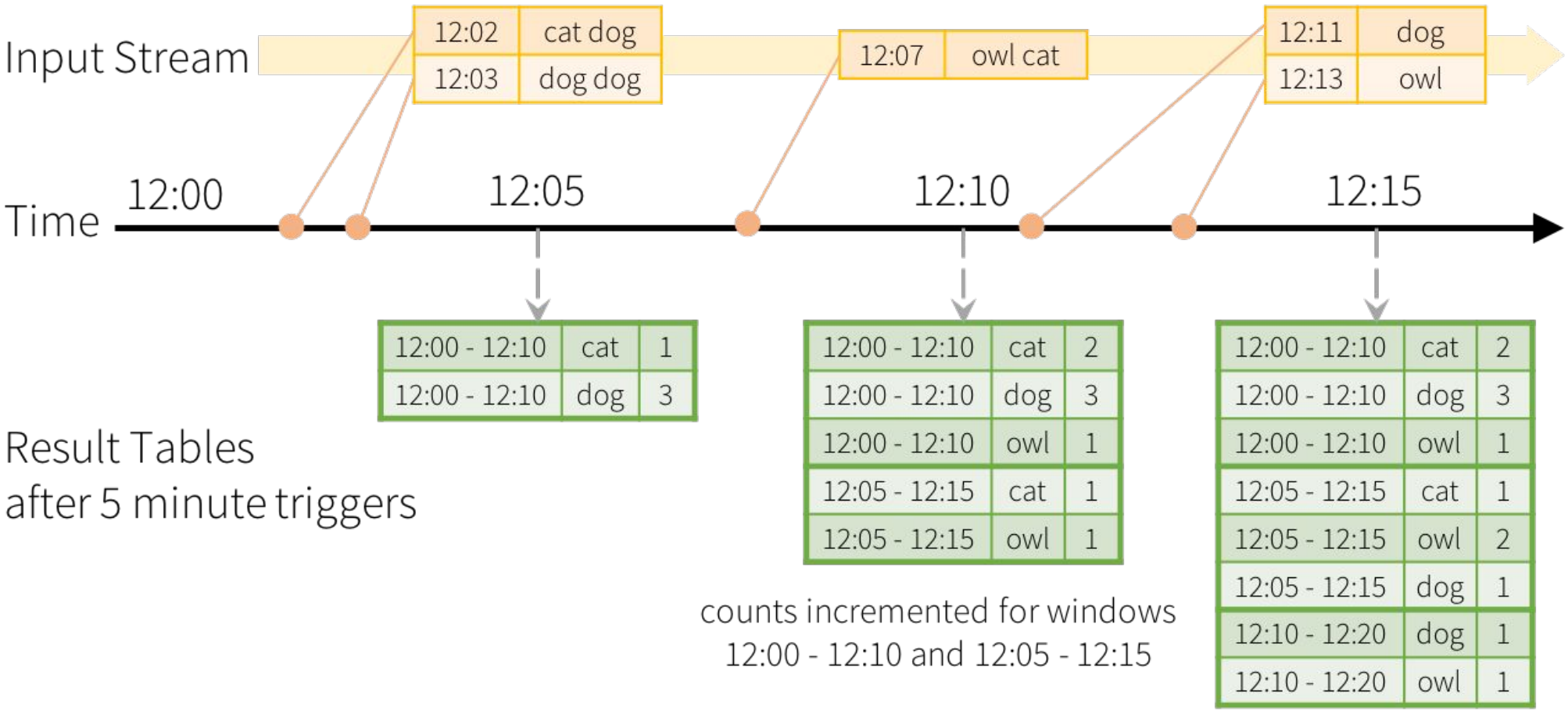
1. Unspecific (default) - the query will be executed in micro-batch mode where each batch processed as soon as last batch finished
2. Fixed interval:
    a. If previous micro batch finished - the engine will wait for the next trigger
    b. If previous micro batch takes longer than the interval - the engine will wait for the end of the previous micro batch and trigger the new micro batch right after the end of it
    c. If no data - no trigger will happen
3. On time micro batch - good for cases where we want to start a cluster, process the last data and terminate it. Mainly for reducing costs.
4. Continuous processing - experimental and out of scope for this course.

trainologic

Window Operations

1. Spark supports windows aggregation operations
2. This, BTW, supported both in batch and structured streaming
3. Window is frame on a field:
    a. Remember the revenue?
        i. Max price per category?
    b. Avg price for the
    c. Example 2: 10 minutes on time field

trainologic



Result Tables
after 5 minute triggers

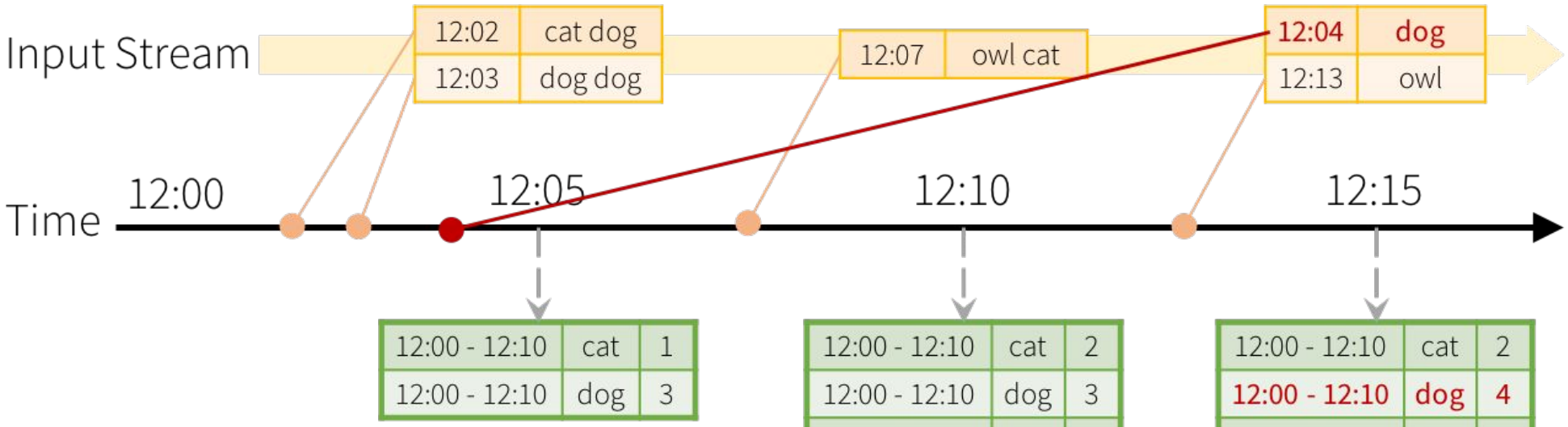
counts incremented for windows
12:00 - 12:10 and 12:05 - 12:15

counts incremented for windows
12:05 - 12:15 and 12:10 - 12:20

Windowed Grouped Aggregation
with 10 min windows, sliding every 5 mins

trainologic

late data that was generated
at 12:04 but arrived at 12:11



12:00 - 12:10	cat	1
12:00 - 12:10	dog	3

12:00 - 12:10	cat	2
12:00 - 12:10	dog	3
12:00 - 12:10	owl	1
12:05 - 12:15	cat	1
12:05 - 12:15	owl	1

12:00 - 12:10	cat	2
12:00 - 12:10	**dog**	**4**
12:00 - 12:10	owl	1
12:05 - 12:15	cat	1
12:05 - 12:15	owl	2
12:10 - 12:20	owl	1

Result Tables
after 5 minute triggers

counts incremented only for
window 12:00 - 12:10

Late data handling in Windowed Grouped Aggregation

trainologic

Window Operations

Setting the window:

```
Dataset<Row> aggRes = rateLines.groupBy(
 functions.window(col("timestamp"), "20 seconds", "5 seconds"), col("value")
).count();
```

Setting high watermark:

```
Dataset<Row> aggRes = rateLines
    .withWatermark("timestamp", "1 minutes")
```

trainologic

Structured Streaming - Stocks

trainologic

Structured Streaming - Stocks

1. Use nc to stream the data
2. Line would be
   msn,1200
   goog,1000
3. Read the stream of stocks prices
4. Find the max price per 10 seconds
5. Continuous max price on window size of 20 seconds, every 10 seconds

→ You'll need timestamp field to win on it...

trainologic

Questions?