



APACHE SPARK STREAMING APPLICATION

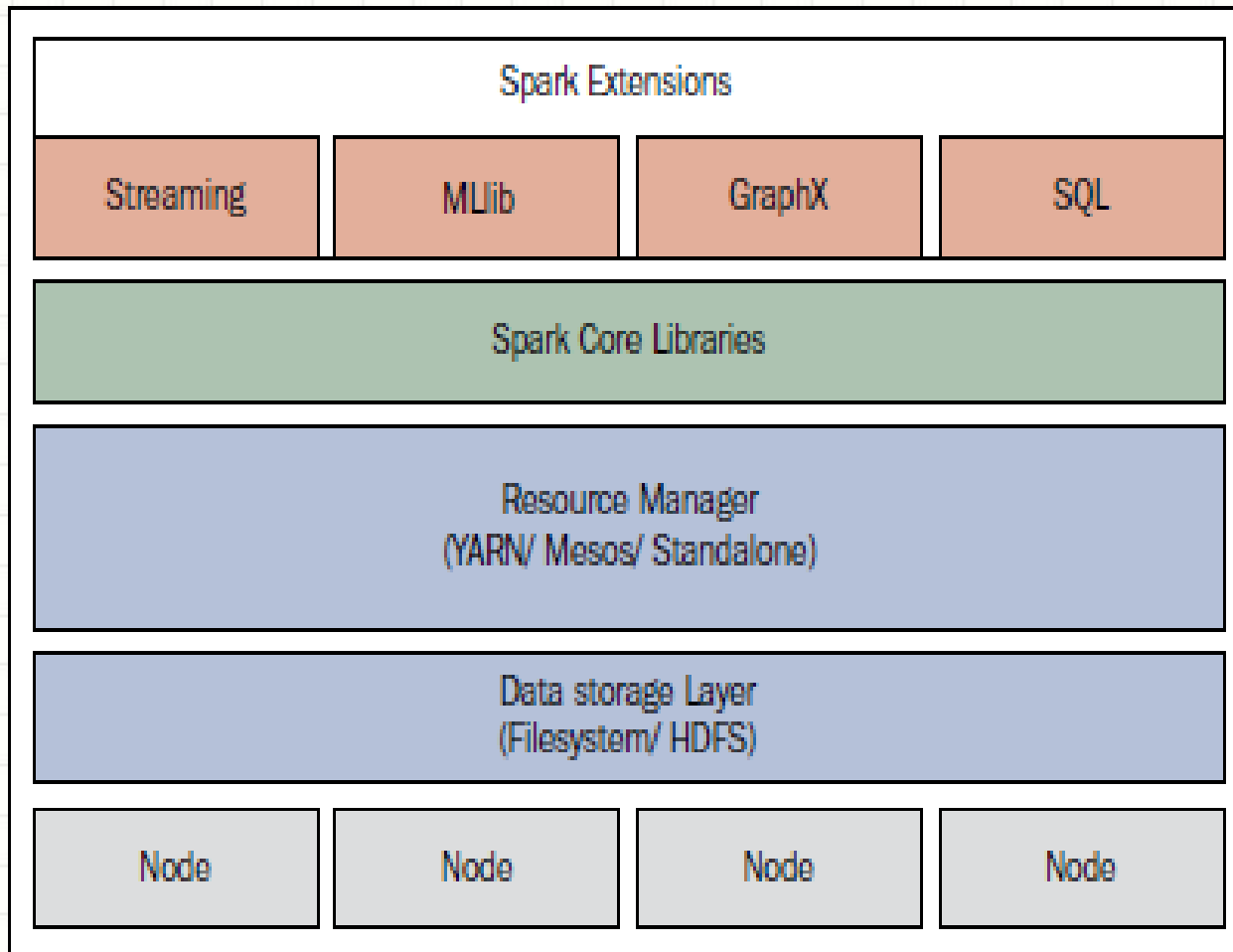
Bin Jiang

11/26/2016

Components of Spark

- SQL for structured data processing
- MLlib for iterative data processing - machine learning
- GraphX for graph processing
- Spark Streaming – Real-time data processing of streaming data

Architecture of Spark



Batch Processing

- A series of jobs which are connected with each other or executed after one other in a sequence or in parallel
- Input data is collected in batches over a period of time
- Output of one batch can be the input of another batch jobs
- Scheduled and run at predefined intervals or at a specific time

Example of Batch Processing

- Log analysis
- Billing applications
- Backups
- Data warehouses
- Model Training

Real-time Data Processing

- Receiving constantly changing data and processes it sufficiently rapidly to be able to control the source of the data
- The response time for processing data in real time is instant and expected to be in milliseconds

Example of Real-time Data Processing

- Bank ATMs
- Real-time monitoring
- Real-time business intelligence
- Operational intelligence
- POS
- Assembly lines
- Recommendation
- Real-time prediction

Streaming over Big Data

➤ Real Time Streaming Data Processing

- ☐ Build a global News Scanner that scrapes news in near real time, and uses sophisticated text analysis, SimHash, Random Indexing and Streaming K-Means to produce a geopolitical monitoring tool that allows users to track major world events as they unfold
- ☐ Real-Time Image Recognition
- ☐ Approximate computing for stream analytics

Streaming over Big Data

➤ Real Time Streaming Data Processing

- ☐ User behavior anomaly detection for information security
- ☐ Deduplication and author-disambiguation of streaming records via supervised models based on content encoders
- ☐ Large-scale ads CTR prediction
- ☐ Real-Time Detection of Anomalies in the Database Infrastructure

Complexity of Real-time Processing

- System responsiveness
- Fault-tolerant
- Scalable
- In memory

Other Streaming Frameworks

- ☐ Flume
- ☐ NiFi
- ☐ Gearpump
- ☐ Apex
- ☐ Kafka Streams
- ☐ Spark Streaming
- ☐ Storm (and Trident)
- ☐ Flink
- ☐ Samza
- ☐ Ignite
- ☐ Beam

What is Spark Streaming

- **Spark Streaming** is the incremental stream processing framework for Spark
- Offers the data abstraction called DStream that hides the complexity of dealing with a continuous data stream and makes it as easy for programmers as using one single RDD at a time
- Shines on performing the **T** stage
- Micro-batching streaming framework

What is Spark Streaming

- DStream is similar to work with as a RDD with the DStream API to match RDD API
- Can reuse your RDD-based code and apply it to Dstream
- Runs streaming jobs every batch duration to pull and process data (often called records) from one or many input streams
- Each batch computes (generates) a RDD for data in input streams for a given batch and submits a Spark job to compute the result. It does this over and over again until the streaming context is stopped

What is Spark Streaming

- Spark Streaming supports checkpointing that writes received records to a highly-available HDFS-compatible storage and allows to recover from temporary downtimes
- Spark Streaming allows for integration with real-time data sources ranging from such basic ones like a HDFS-compatible file system or socket connection to more advanced ones like Apache Kafka or Apache Flume
- Checkpointing is also the foundation of stateful and windowed operations

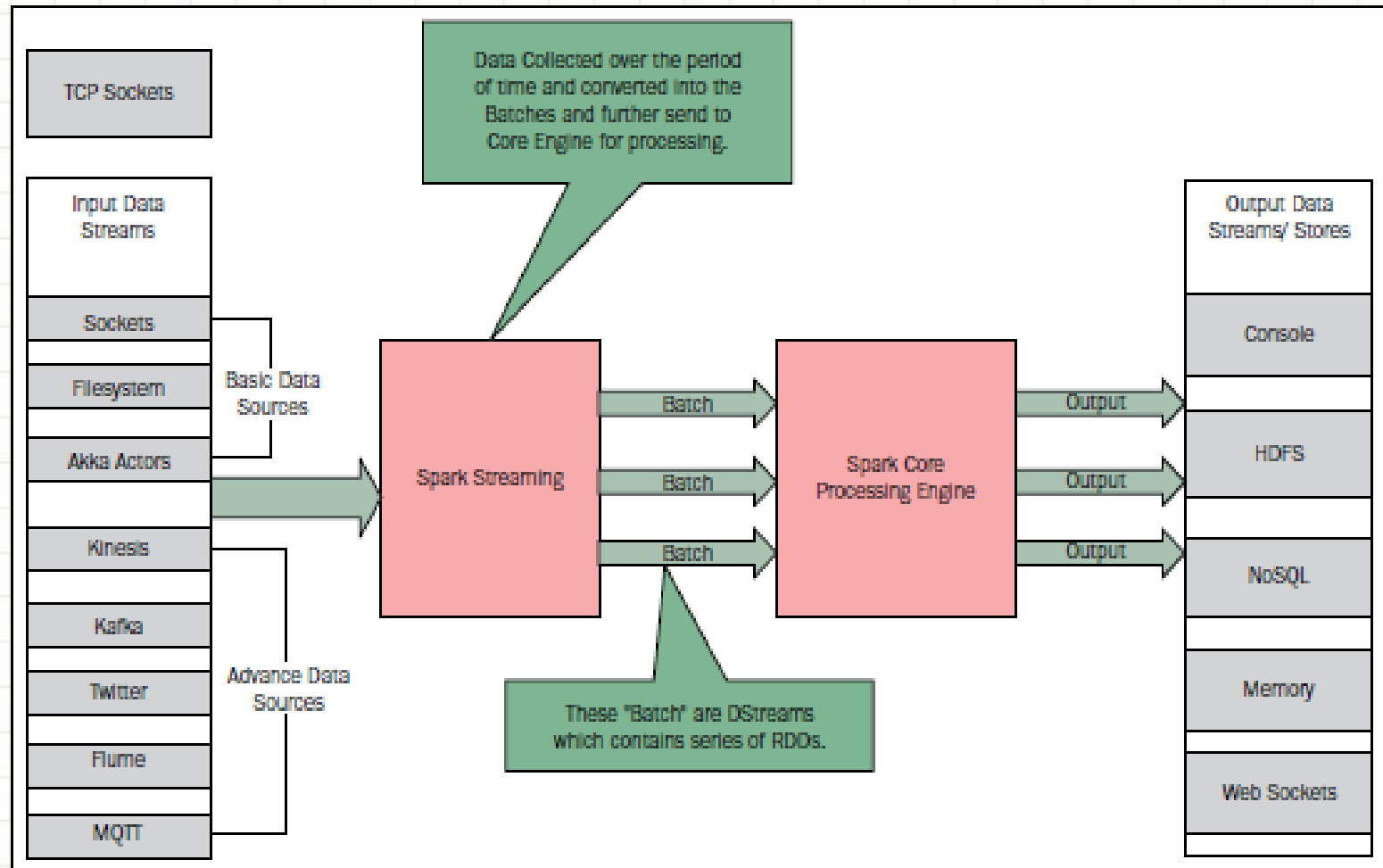
What is Spark Streaming

- Spark Streaming is an extension of the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams. Data can be ingested from many sources like Kafka, Flume, Twitter, ZeroMQ, Kinesis, or TCP sockets, and can be processed using complex algorithms expressed with high-level functions like map, reduce, join and window. Finally, processed data can be pushed out to filesystems, databases, and live dashboards. In fact, you can apply Spark's machine learning and graph processing algorithms on data stream

Micro Batch - Spark Streaming

- Spark Streaming implemented the concept of micro-batching where the streaming data is divided into a series of deterministic micro-batches
- Each batch is processed as an individual record and the further output of each batch is sent to the user-defined output streams
- Can be further persisted into HDFS, NoSQL
- Can be used to create live dashboards.

Architecture - Spark Streaming



Architecture - Spark Streaming

- Input data streams:
 - Basic data sources and Advanced data sources
- Spark Streaming:
 - The API which provides the consumption of streaming data at a predefined interval and converts the streaming data into series batches which are further sent to the Spark Core engine for processing

Architecture - Spark Streaming

- Batch
 - Batches are nothing more than a series of RDDs. Spark Streaming provides DStreams which holds the reference of a series of RDDs which are based on the data directly provided by input streams or processed data streams generated by transforming the input streams
- Spark Core Engine
 - Receives the input in the form of RDD and further processes and finally sends it to the associated output streams for storage

Architecture - Spark Streaming

- Output data streams:
 - The output of each processed batch is directly sent to the output streams for further actions. These output streams can be of varied types, ranging from a raw file system, NoSQL, Queues or web sockets for visualizing the streaming data

Hello World - Spark Streaming

- `val conf = new SparkConf().setMaster("local[2]").setAppName("NetworkWordCount")`
- `val ssc = new StreamingContext(sparkConf, Seconds(1))`
- `val lines = ssc.socketTextStream("127.0.0.1", 9999, StorageLevel.MEMORY_AND_DISK_SER)`
- `val words = lines.flatMap(_.split(" "))`
- `val pairs = words.map(word => (word, 1))`
- `val wordCounts = pairs.reduceByKey(_ + _)`
- `wordCounts.print()`
- `ssc.start()`
- `ssc.awaitTermination()`

Packaging and Deploying

- Export your project as a .jar file, name it spark_demo.jar and save this .jar file
- Execute the following command for deploying your Spark Streaming job
 - `spark-submit --class ca.bigdata.training.example --master <SPARK-MASTER-URL> spark_demo.jar`

Basic Concepts - Linking

```
<dependency>  
  <groupId>org.apache.spark</groupId>  
  <artifactId>spark-streaming_2.11</artifactId>  
  <version>2.2.0</version>  
</dependency>
```

Source	Artifact
Kafka	spark-streaming-kafka-0-8_2.11
Flume	spark-streaming-flume_2.11
Kinesis	spark-streaming-kinesis-asl_2.11 [Amazon Software License]

StreamingContext

- The Entry Point to Spark Streaming

```
import org.apache.spark.streaming._  
val sc = SparkContext.getOrCreate  
val ssc = new StreamingContext(sc, Seconds(5))
```

- With an instance of StreamingContext, you can create ReceiverInputDStreams or set the checkpoint directory
- Once streaming pipelines are developed, you start StreamingContext to set the stream transformations in motion

StreamingContext

- Creating StreamingContext from Scratch
- Recreating StreamingContext from a checkpoint file
- Creating ReceiverInputDStreams
- Checkpoint Interval
- Checkpoint Directory
- Initial Checkpoint
- Starting StreamingContext
- Stopping StreamingContext

Discretized Streams (DStreams)

- **DStream** is the basic abstraction provided by Spark Streaming. It represents a continuous stream of data, either the input data stream received from source, or the processed data stream generated by transforming the input stream
- Internally, a DStream is represented by a continuous series of RDDs

Input DStreams and Receivers

- Built-in streaming sources

- Basic sources:

Source directly available in the StreamingContext API.
Examples: file system and socket connections

- Advanced sources:

Sources like Kafka, Flume, Kinesis etc are available
through extra utility classes

Input DStreams and Receivers

- Note

When running a Spark Streaming program locally, do not use “local” or “local[1]” as the master URL. Either of these means that only one thread will be used for running tasks locally. If you are using an input DStream based on a receiver (e.g. sockets, Kafka, Flume, etc.), then the single thread will be used to run the receiver, leaving no thread for processing the received data. Hence, when running locally, always use “local[n]” as the master URL, where $n > \text{number of receivers to run}$

Basic Sources

- File Streams:

- For reading data from files on any file system compatible with the HDFS API (that is, HDFS, S3, NFS, etc.), a DStream can be created as:
File Streams: For reading data from files on any file system compatible with the HDFS API (that is, HDFS, S3, NFS, etc.), a DStream can be created as:

```
streamingContext.fileStream[KeyClass, ValueClass,  
InputFormatClass](dataDirectory)
```


Basic Sources

- Streams based on Custom Receivers:
 - DStreams can be created with data streams received through custom receivers
- Queue of RDDs as a Stream:
 - For testing a Spark Streaming application with test data, one can also create a DStream based on a queue of RDDs, using `streamingContext.queueStream(queueOfRDDs)`

Advanced Sources (Kafka)

- Kafka: Kafka broker versions 0.8.2.1 or higher
- Flume: Flume 1.6.0
- Kinesis: Kinesis Client Library 1.2.1

Advanced Sources (Kafka 0.8)

- Receiver-based Approach
 - This approach uses a Receiver to receive the data. The Receiver is implemented using the Kafka high-level consumer API. As with all receivers, the data received from Kafka through a Receiver is stored in Spark executors, and then jobs launched by Spark Streaming processes the data
 - To ensure zero-data loss, you have to additionally enable Write Ahead Logs in Spark Streaming

Advanced Sources (Kafka 0.8)

- Receiver-based Approach

- Link

`artifactId = spark-streaming-kafka-0-8_2.11`

- Programming

```
val kafkaStream = KafkaUtils.createStream(streamingContext,  
    [ZK quorum], [consumer group id], [per-topic number of Kafka  
    partitions to consume])
```

Advanced Sources (Kafka 0.8)

- Receiver-based Approach

➤ Note

Topic partitions in Kafka does not correlate to partitions of RDDs generated in Spark Streaming. So increasing the number of topic-specific partitions in the `KafkaUtils.createStream()` only increases the number of threads using which topics that are consumed within a single receiver.

Advanced Sources (Kafka 0.8)

- Direct Approach – No Receivers
 - Instead of using receivers to receive data, this approach periodically queries Kafka for the latest offsets in each topic+partition, and accordingly defines the offset ranges to process in each batch. When the jobs to process the data are launched, Kafka's simple consumer API is used to read the defined ranges of offsets from Kafka
 - No need to create multiple input Kafka streams and union them. With `directStream`, Spark Streaming will create as many RDD partitions as there are Kafka partitions to consume

Advanced Sources (Kafka 0.8)

- Direct Approach – No Receivers
 - Eliminates the problem as there is no receiver, and hence no need for Write Ahead Logs
 - Exactly once

Advanced Sources (Kafka 0.8)

- Direct Approach – No Receivers

- Programming

```
val directKafkaStream = KafkaUtils.createDirectStream[  
  [key class], [value class], [key decoder class], [value decoder  
  class] ](  
  streamingContext, [map of Kafka parameters], [set of topics to  
  consume])
```

- Link

```
artifactId = spark-streaming-kafka-0-8_2.11
```

Advanced Sources (Kafka 0.10)

- The Spark Streaming integration for Kafka 0.10 is similar in design to the 0.8 Direct Stream approach. It provides simple parallelism, 1:1 correspondence between Kafka partitions and Spark partitions, and access to offsets and metadata
- [Link](#)

`artifactId = spark-streaming-kafka-0-10_2.11`

Advanced Sources (Kafka 0.10)

- Programming

```
val kafkaParams = Map[String, Object](  
  "bootstrap.servers" -> "localhost:9092,anotherhost:9092",  
  "key.deserializer" -> classOf[StringDeserializer],  
  "value.deserializer" -> classOf[StringDeserializer],  
  "group.id" -> "use_a_separate_group_id_for_each_stream",  
  "auto.offset.reset" -> "latest",  
  "enable.auto.commit" -> (false: java.lang.Boolean)  
)  
  
val topics = Array("topicA", "topicB")  
val stream = KafkaUtils.createDirectStream[String, String](  
  streamingContext,  
  PreferConsistent,  
  Subscribe[String, String](topics, kafkaParams)  
)
```

Advanced Sources (Kafka 0.10)

- LocationStrategies
 - Spark integration keep cached consumers on executors (rather than recreating them for each batch)
 - Prefer to schedule partitions on the host locations that have the appropriate consumers
 - Use PreferBrokers to schedule partitions on the Kafka leader for that partition

Advanced Sources (Kafka 0.10)

- ConsumerStrategies
 - Provides an abstraction that allows Spark to obtain properly configured consumers even after restart from checkpoint
 - Use ConsumerStrategies.Subscribe to subscribe to a fixed collection of topics
 - Use ConsumerStrategies.SubscribePattern to subscribe to particular topics which based on a regex
 - Use ConsumerStrategies.Assign to specify a fixed collection of partitions

Advanced Sources (Kafka 0.10)

- Creating an RDD

```
val offsetRanges = Array(  
  // topic, partition, inclusive starting offset, exclusive ending offset  
  OffsetRange("test", 0, 0, 100),  
  OffsetRange("test", 1, 0, 100)  
)  
  
val rdd = KafkaUtils.createRDD[String, String](sparkContext, kafkaParams, offsetRanges, PreferConsistent)
```


Advanced Sources (Kafka 0.10)

- Storing Offsets

- Checkpoints

If enable Spark checkpointing, offsets will be stored in the checkpoint

- Kafka

Kafka has an offset commit API that stores offsets in a special Kafka topic. By default, the new consumer will periodically auto-commit offsets

Advanced Sources (Kafka 0.10)

- Storing Offsets

- Zookeeper

- Custom Data Store

For data stores that support transactions, saving offsets in the same transaction as the results can keep the two in sync

Custom Sources

- Implementing a Custom Receiver
 - A custom receiver must extend the abstract class by implementing two methods: `onStart` and `onStop`
- Using the Custom Receiver
 - The custom receiver can be used in a Spark Streaming application by using `streamingContext.receiverStream`

Receiver Reliability

- Reliable Receiver
 - A reliable receiver correctly sends acknowledgment to a reliable source when the data has been received and stored in Spark with replication.
- Unreliable Receiver
 - An unreliable receiver does not send acknowledgment to a source. This can be used for sources that do not support acknowledgment, or even for reliable sources when one does not want or need to go into the complexity of acknowledgment.

Components of Spark Streaming API

- All Spark Streaming classes are packaged in the `org.apache.spark.streaming.*` package
- Spark Streaming defines two critical classes, `StreamingContext.scala` and `DStream.scala`
- `StreamingContext` is similar to `SparkContext` but provides an entry point to Spark Streaming functionality.
- `DStream` provides the basic abstraction of Spark Streaming. It provides the sequence of RDDs created from the live data or transforming the existing `DStreams`.

Components of Spark Streaming API

- `org.apache.spark.streaming.flume.*` provides classes for consuming input data from Fume
- `org.apache.spark.streaming.kafka.*` provides classes for consuming input data from Kafka
- `org.apache.spark.streaming.mqtt.*` provides classes for consuming input data from MQTT
- `org.apache.spark.streaming.kinesis.*` provides classes for consuming input data from Amazon Kinesis
- `org.apache.spark.streaming.twitter.*` provides classes for consuming input data from Twitter feeds

Multiple Discretized Streams

- Multiple DStreams per Spark Streaming context
- Each DStreams contains a series of RDDs
- Each RDD is the snapshot of the data received at a particular point in time from the receiver

Duration of DStreams

- The duration or interval after which data needs to be converted or computed into RDDs is defined as the second parameter of StreamingContext:

```
val streamCtx = new StreamingContext(conf, Seconds(2))
```

Operations of DStreams

- DStreams support two types of operations:

➤ Transformations:

Supports all transformation operations as supported by RDD like `map()`, `flatMap()` and creates a new DStream from the processed data. Transformation are applied separately on the RDD at each time interval

➤ Output operations:

These operations help to save the final output to external systems or the same external cache, which is similar to the output operations supported by RDD

Operations of DStreams

- DStreams support three additional operations:

➤ Windowing:

Windowing is a special type of operation which is provided only by DStreams and groups all the records from a sliding window of past time intervals into one RDD

➤ Incremental aggregation:

This provides the functionality for a common use case where we may need to compute an aggregate like a count or max over a sliding window. DStreams provide several variants of an incremental operations. All methods in DStream prefixed with Window provide incremental aggregations like `countByWindow`, `reduceByWindow`

Operations of DStreams

- DStreams support three additional operations:

➤ Stateful Processing:

Dstream operators are stateless and know nothing about the previous records and hence a state. If you would like to react to new records appropriately given the previous records you would have to resort to using persistent storages outside Spark Streaming

Stream Operators

- (output operator) print
- (output operator) foreachRDD
- (output operator) saveAsObjectFiles
- (output operator) saveAsTextFiles
- map
- slice
- window
- reduceByWindow
- reduce
- glom
- transform

Stream Operators

- `map(func)`
- `flatMap(func)`
- `filter(func)`
- `repartition(numPartitions)`
- `union(otherStream)`
- `count()`
- `reduce(func)`
- `countByValue()`
- `reduceByKey(func, [numTasks])`
- `join(otherStream, [numTasks])`
- `cogroup(otherStream, [numTasks])`

Functional Operations of DStreams

- flatMap(flatMapFunc)
 - DStream[U]—Similar to map(...) but, before returning, it flattens the results and then returns the final result set.
- forEachRDD(forEachFunc)
 - Applies the given function to all RDDs in a given stream. It is a special type of function and it is worth noting that the given function is applied to all RDDs on the driver node itself but there could be actions defined in the RDD and all those actions are performed over the cluster.

Functional Operations of DStreams

- `filter(filterFunc)`
 - `DStream[T]`—Applies the provided function to all the elements of RDD and generates the RDD only for those elements which return `TRUE`
- `map(mapFunc)`
 - `DStream[U]`—Applies a given function `mapFunc` to all elements of RDD and generates a new RDD.

Functional Operations of DStreams

- `mapPartitions(mapPartFunc, preservePartitioning)`
 - Return a new DStream in which each RDD is generated by applying `mapPartitions()` to each RDD in the invoking DStream. Applying `mapPartitions()` to an RDD applies the given function to each partition of the RDD.

Transform Operation

- Return a new DStream by applying a RDD-to-RDD function to every RDD of the source DStream. This can be used to do arbitrary RDD operations on the Dstream

```
val spamInfoRDD = ssc.sparkContext.newAPIHadoopRDD(...) // RDD containing  
spam information
```

```
val cleanedDStream = wordCounts.transform(rdd => {  
    rdd.join(spamInfoRDD).filter(...) // join data stream with spam information to  
    do data cleaning  
    ...  
})
```

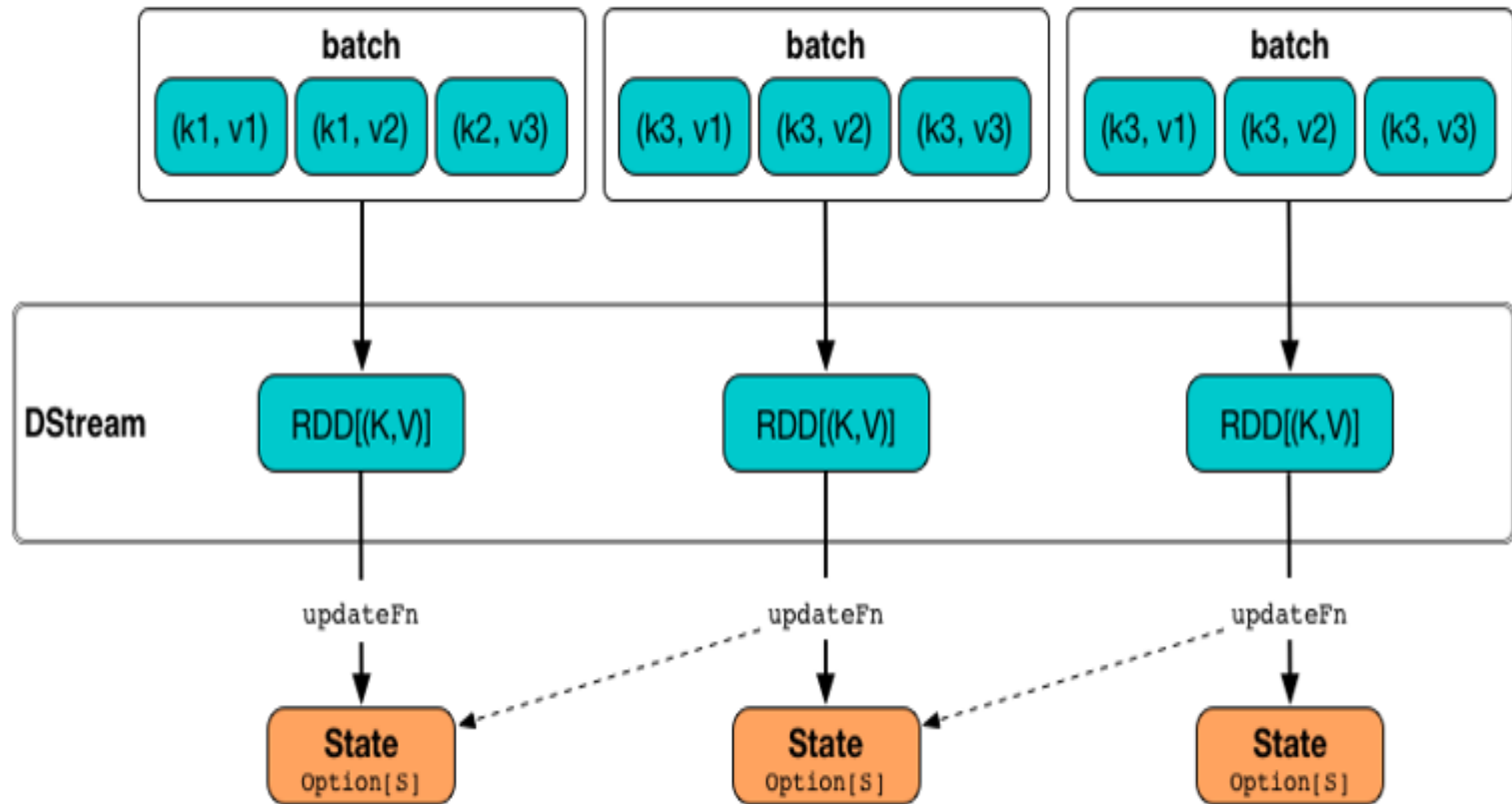
Stateful Operations of DStreams

- Return a new DStream by applying a RDD-to-RDD function to every RDD of the source DStream. This

Stateful Operations of DStreams

- Return a new DStream by applying a RDD-to-RDD function to every RDD of the source DStream. This

Stateful Operations of DStreams



Stateful Operations of DStreams

- Building Stateful Stream Processing Pipelines
- Also called cumulative calculations
- The motivation for the stateful operators is that by design streaming operators are stateless and know nothing about the previous records and hence a state. If you'd like to react to new records appropriately given the previous records you would have to resort to using persistent storages outside Spark Streaming

UpdateStateByKey Operation

- Return a new "state" DStream where the state for each key is updated by applying the given function on the previous state of the key and the new values for the key. This can be used to maintain arbitrary state data for each key
- Enable "checkpoint"

Micro-batching vs Windowing

- Micro-batching provides some flexibility where we can accumulate events and then start our computing jobs but in micro-batching all batches of data are independent and only contain new events as they appear or are received by the streams, they do not add new events to the older events without changing the batch size

Windowing Operation

- Return a new "state" DStream where the state for each key is updated by applying the given function on the previous state of the key and the new values for the key. This can be used to maintain arbitrary state data for each key

Windowing Operation

- Windowing functions provide exactly the same functionality where they define the scope of the data which needs to be analyzed, as in the last 10 minutes, 20 minutes, and so on, and further slide this window by a configured interval like one or two minutes.

Windowing Operation

- Allow you to apply transformations over a **sliding window** of data, i.e. build a *stateful computation* across multiple batches
- The transformations know nothing about the past without windowed operators

Windowing Operation

- Streaming Windowed Operators:
 - `window(windowLength, slideInterval)`
 - `countByWindow(windowLength, slideInterval)`
 - `reduceByWindow(func, windowLength, slideInterval)`
 - `reduceByKeyAndWindow(func, windowLength, slideInterval, [numTasks])`
 - `reduceByKeyAndWindow(func, invFunc, windowLength, slideInterval, [numTasks])`
 - `countByValueAndWindow(windowLength, slideInterval, [numTasks])`

Windowing Operation

- `window(WindowDuration, SlideDuration)`
 - This operation works directly on the stream and provides all elements of DStream within the given duration
- `reduceByKeyAndWindow(reduceFunc, WindowDuration, SlideDuration)`
 - This operation works with the reduce function where it applies the reduce function only on the RDDs in the sliding window
- `groupByKeyAndWindow(WindowDuration, SlideDuration)`
 - Applies the groupBy operation on the available keys within the duration as specified by SlidingDuration

Join Operations of DStreams

- Streams can be very easily joined other streams

```
val stream1: DStream[String, String] = ...
```

```
val stream2: DStream[String, String] = ...
```

```
val joinedStream = stream1.join(stream2)
```

```
val windowedStream1 = stream1.window(Seconds(20))
```

```
val windowedStream2 = stream2.window(Minutes(1))
```

```
val joinedStream = windowedStream1.join(windowedStream2)
```

Output Operations on DStreams

- Output operations allow Dstream's data to be pushed out to external systems like a database or a file system, they trigger the actual execution of all the Dstream transformations (similar to actions for RDDs)

Output Operations on DStreams

- `saveAsObjectFiles`
- `saveAsTextFiles`
- `print()`
- `saveAsHadoopFiles()`
- `foreachRDD()`

MLlib Operations

- Can also easily use machine learning algorithms provided by MLlib
- There are streaming machine learning algorithms (e.g. Streaming Linear Regression, Streaming KMeans, etc.) which can simultaneously learn from the streaming data as well as apply the model on the streaming data.
- For a much larger class of machine learning algorithms, you can learn a learning model offline and then apply the model online on streaming data

DataFrame and SQL Operations

- Can easily use DataFrames and SQL operations on streaming data
- Create a SQLContext using the SparkContext that the StreamingContext is using
- Each RDD is converted to a DataFrame, registered as a temporary table and then queried using SQL

DataFrame and SQL Operations

- Example

```
val words: DStream[String] = ...
words.foreachRDD { rdd =>
  // Get the singleton instance of SQLContext
  val sqlContext = SQLContext.getOrCreate(rdd.sparkContext)
  import sqlContext.implicits._

  // Convert RDD[String] to DataFrame
  val wordsDataFrame = rdd.toDF("word")

  // Register as table
  wordsDataFrame.registerTempTable("words")

  // Do word count on DataFrame using SQL and print it
  val wordCountsDataFrame =
    sqlContext.sql("select word, count(*) as total from words group by word")
  wordCountsDataFrame.show()
}
```

Design Patterns for using foreachRDD

- `dstream.foreachRDD` is a powerful primitive that allows data to be sent out to external systems. However, it is important to understand how to use this primitive correctly and efficiently. Some of the common mistakes to avoid are as follows.

Design Patterns for using foreachRDD

```
dstream.foreachRDD { rdd =>
  val connection = createNewConnection() // executed at the driver
  rdd.foreach { record =>
    connection.send(record) // executed at the worker
  }
}
```

```
dstream.foreachRDD { rdd =>
  rdd.foreach { record =>
    val connection = createNewConnection()
    connection.send(record)
    connection.close()
  }
}
```


Design Patterns for using foreachRDD

```
dstream.foreachRDD { rdd =>
  val connection = createNewConnection() // executed at the driver
  rdd.foreach { record =>
    connection.send(record) // executed at the worker
  }
} // NEVER WORK
```

```
dstream.foreachRDD { rdd =>
  rdd.foreach { record =>
    val connection = createNewConnection()
    connection.send(record)
    connection.close()
  }
} // BAD PATTERN
```

Design Patterns for using foreachRDD

```
dstream.foreachRDD { rdd =>
  rdd.foreachPartition { partitionOfRecords =>
    val connection = createNewConnection()
    partitionOfRecords.foreach(record => connection.send(record))
    connection.close()
  }
} // BETTER PATTERN
```

```
dstream.foreachRDD { rdd =>
  rdd.foreachPartition { partitionOfRecords =>
    // ConnectionPool is a static, lazily initialized pool of connections
    val connection = ConnectionPool.getConnection()
    partitionOfRecords.foreach(record => connection.send(record))
    ConnectionPool.returnConnection(connection) // return to the pool for
    future reuse
  }
} // BEST PATTERN
```

Caching / Persistence

- Similar to RDDs, DStreams also allow developers to persist the stream's data in memory
- This is useful if the data in the DStream will be computed multiple times
- For window-based operations like `reduceByWindow` and `reduceByKeyAndWindow` and state-based operations like `updateStateByKey`, this is implicitly true
- DStreams generated by window-based operations are automatically persisted in memory, without the developer calling `persist()`

Caching / Persistence

- Using the `persist()` method on a DStream will automatically persist every RDD of that DStream in memory

Checkpointing

- A streaming application must operate 24/7
- Checkpoint enough information to a fault- tolerant storage system such that it can recover from failures
- Two types of data that are checkpointed:
 - Metadata checkpointing
 - ✓ Configuration
 - ✓ DStream operations
 - ✓ Incomplete batches
 - Data checkpointing

When to enable Checkpointing

- Usage of stateful transformations - If either `updateStateByKey` or `reduceByKeyAndWindow` (with inverse function) is used in the application, then the checkpoint directory must be provided to allow for periodic RDD checkpointing
- Recovering from failures of the driver running the application - Metadata checkpoints are used to recover with progress information

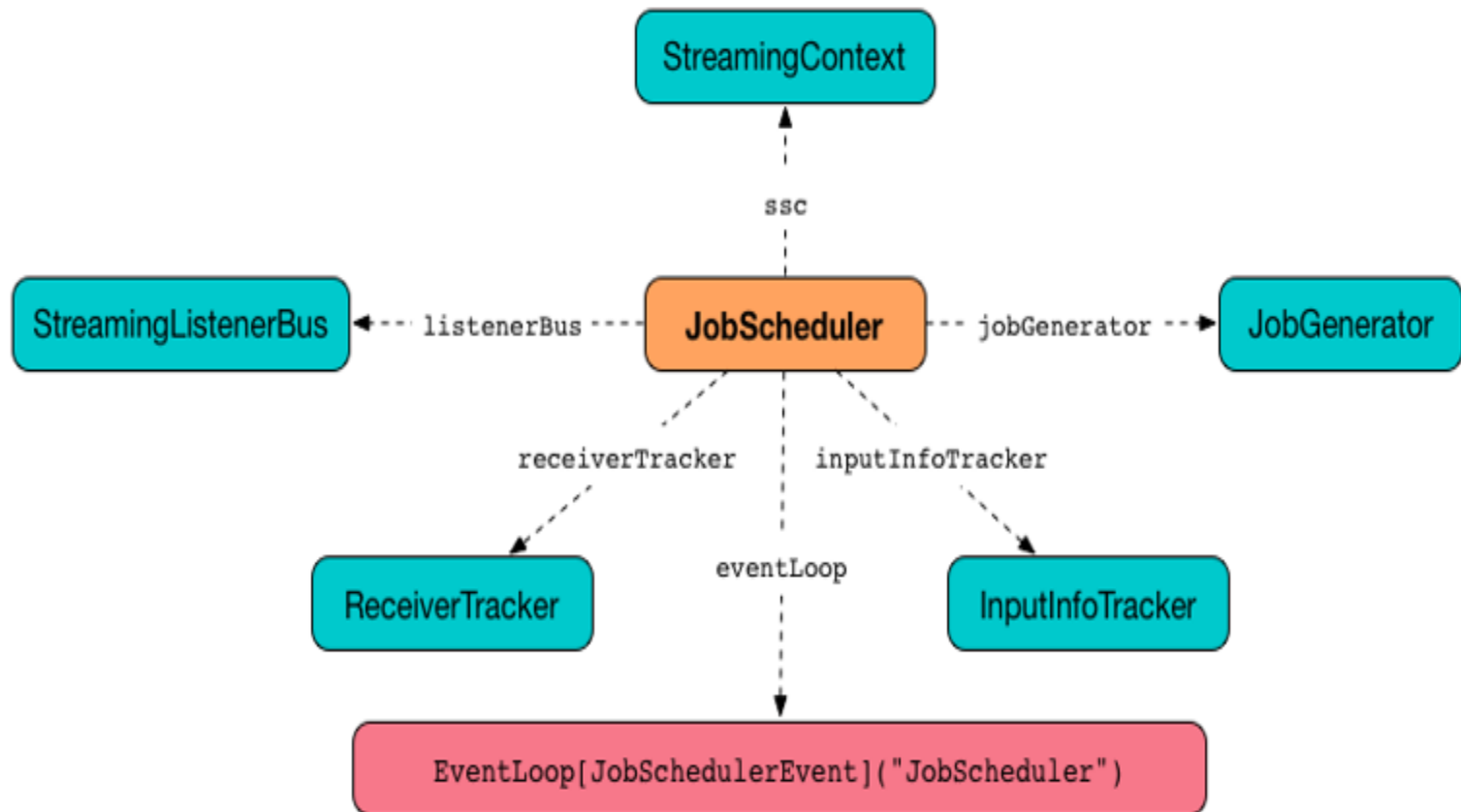
How to configure Checkpointing

- Checkpointing can be enabled by setting a directory in a fault-tolerant, reliable file system via calling `streamingContext.checkpoint(checkpointDirectory)`
- When the program is being started for the first time, it will create a new `StreamingContext`, set up all the streams and then call `start()`
- When the program is being restarted after failure, it will re-create a `StreamingContext` from the checkpoint data in the checkpoint directory

Spark Streaming - Backpressure

- With backpressure you can guarantee that your Spark Streaming application is stable, e.g. receives data only as fast as it can process it
- Backpressure shifts the trouble of buffering input record to the sender so it keeps records until they could be processed by a streaming application. You could alternatively use dynamic allocation feature in Spark Streaming to increase the capacity of streaming infrastructure without slowing down the senders
- Backpressure is disabled by default and can be turned on using `spark.streaming.backpressure.enabled` setting

Spark Streaming - JobScheduler



Performance Tuning

- Reducing the Batch Processing Times
- Level of Parallelism in Data Receiving
- Level of Parallelism in Data Processing
- Data Serialization
- Setting the Right Batch Interval
- Memory Tuning

Overview – 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

Hello World – Structured Streaming

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

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

import spark.implicits._

val lines = spark.readStream
  .format("socket")
  .option("host", "localhost")
  .option("port", 9999)
  .load()

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

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

val query = wordCounts.writeStream
  .outputMode("complete")
  .format("console")
  .start()

query.awaitTermination()
```


Structured Streaming – Output

- Complete Mode
 - The entire updated Result Table will be written to the external storage. It is up to the storage connector to decide how to handle writing of the entire table.

Structured Streaming – Output

- Append Mode
 - Only the new rows appended in the Result Table since the last trigger will be written to the external storage. This is applicable only on the queries where existing rows in the Result Table are not expected to change.

Structured Streaming – Output

- Update Mode
 - Only the rows that were updated in the Result Table since the last trigger will be written to the external storage. Note that this is different from the Complete Mode in that this mode only outputs the rows that have changed since the last trigger. If the query doesn't contain aggregations, it will be equivalent to Append mode.

Creating Streaming DataFrames

- File source

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

Creating Streaming DataFrames

- Kafka source

```
val df = spark
  .readStream
  .format("kafka")
  .option("kafka.bootstrap.servers", "host1:port1,host2:port2")
  .option("subscribe", "topic1")
  .load()
df.selectExpr("CAST(key AS STRING)", "CAST(value AS STRING)")
  .as[(String, String)]

// Subscribe to multiple topics
val df = spark
  .readStream
  .format("kafka")
  .option("kafka.bootstrap.servers", "host1:port1,host2:port2")
  .option("subscribe", "topic1,topic2")
  .load()
df.selectExpr("CAST(key AS STRING)", "CAST(value AS STRING)")
  .as[(String, String)]
```

Creating Streaming DataFrames

- Socket source

```
val spark: SparkSession = ...

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

socketDF.isStreaming    // Returns True for DataFrames that have streaming sources

socketDF.printSchema
```


Structured Streaming – Basic Operations

- Selection, Projection and Aggregation

```
case class DeviceData(device: String, deviceType: String, signal: Double, time: DateTime)

val df: DataFrame = ... // streaming DataFrame with IOT device data with schema { device: string, deviceType: string, signal: double, time: string }
val ds: Dataset[DeviceData] = df.as[DeviceData] // streaming Dataset with IOT device data

// Select the devices which have signal more than 10
df.select("device").where("signal > 10") // using untyped APIs
ds.filter(_.signal > 10).map(_.device) // using typed APIs

// Running count of the number of updates for each device type
df.groupBy("deviceType").count() // using untyped API

// Running average signal for each device type
import org.apache.spark.sql.expressions.scalalang.typed
ds.groupByKey(_.deviceType).agg(typed.avg(_.signal)) // using typed API
```

Structured Streaming – Window Operations

```
import spark.implicits._

val words = ... // streaming DataFrame of schema { timestamp: Timestamp, word: String }

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

Structured Streaming – Late Data and Watermarking

- Watermarking, which lets the engine automatically track the current event time in the data and attempt to clean up old state accordingly

```
import spark.implicits._

val words = ... // streaming DataFrame of schema { timestamp: Timestamp, word: String }

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

Structured Streaming – Output Sinks

- File sink

```
writeStream  
  .format("parquet")           // can be "orc", "json", "csv", etc.  
  .option("path", "path/to/destination/dir")  
  .start()
```

- Foreach sink

```
writeStream  
  .foreach(...)  
  .start()
```

Structured Streaming – Output Sinks

- Console sink

```
writeStream  
  .format("console")  
  .start()
```

- Memory sink

```
writeStream  
  .format("memory")  
  .queryName("tableName")  
  .start()
```

Structured Streaming – Output Sinks

- Kafka sink

```
val ds = df
  .selectExpr("CAST(key AS STRING)", "CAST(value AS STRING)")
  .writeStream
  .format("kafka")
  .option("kafka.bootstrap.servers", "host1:port1,host2:port2")
  .option("topic", "topic1")
  .start()
```


Understanding streaming challenges

- Late arriving/out-of-order data
- Maintaining the state in between batches
- Message delivery reliability
- Streaming is not an island

Understanding streaming challenges

- Late arriving/out-of-order data

Time in streaming:

➤ Event time:

When an event actually happened

➤ Processing time

When the application processed the event

Understanding streaming challenges

- Late arriving/out-of-order data

The time lag between the event time and processing time varies, and this leads to late or out-of-order data.

Reasons:

- Network latencies
- Variance in data load
- Batching of events

Understanding streaming challenges

- Maintaining the state in between batches

Structured Streaming has rewritten state management to maintain this running intermediate state in the memory, backed by write ahead logs (WAL) in the file system for fault-tolerance.

Understanding streaming challenges

- Message delivery reliability
 - At most one delivery
 - At least one delivery
 - Exactly one delivery

Understanding streaming challenges

- Message delivery reliability

Structured Streaming provides an end-to-end and exactly one message delivery guarantee based on the following features:

- Offset tracking in WAL
- State management (using the in-memory state and WAL)
- Fault-tolerant sources and sinks (using WAL)