

Lecture 06

Spark Streaming & Spark Structured Streaming

Zoran B. Djordjević

Spark Streaming APIs

- This lecture follows "Spark Streaming Programming Guide"
<http://spark.apache.org/docs/latest/streaming-programming-guide.htm>
with Spark 2.2 API.
- Spark 2.x has two Streaming APIs:
 - Spark Streaming API and
 - Spark Structured Streaming API.
- *Spark Streaming API* places emphasis on `DStream`(Discretized Streams) objects.
- *Spark Structured Streaming API* places an emphasis on `DataFrame` objects.
- An excellent reference for `DStream` (Discretized Streams) based API, is chapter 6 of "Spark in Action" by P. Zečević & M. Bonači, Manning Publishing, 2016
- Books on Spark Structured Streaming are in print.
- Note: Spark Streaming API will work on Spark 1.6 and Spark 2.X

Spark Streaming Scope

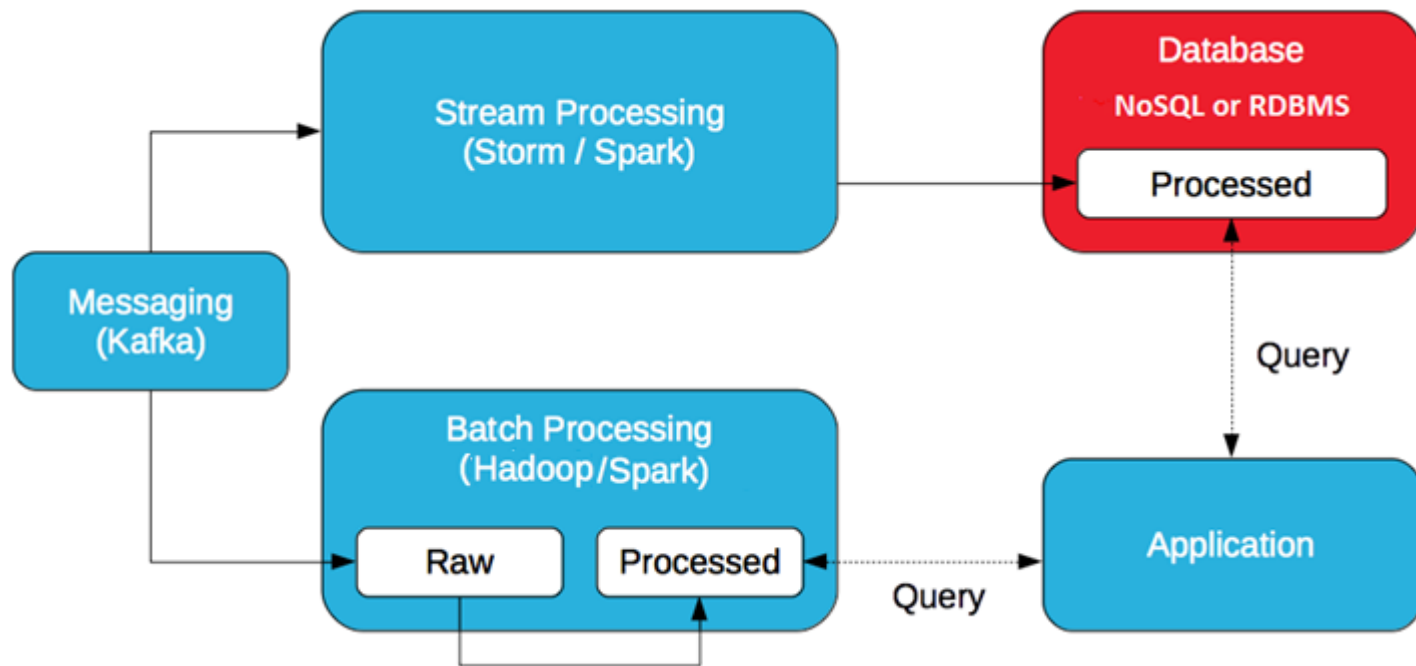
- Most of what we said about Hadoop and Spark so far had a batch processing model in mind. You will collect large volumes of data and then use Hadoop or Spark to analyze those data.
- Many professional fields need real-time data analysis: traffic monitoring, online advertising, stock market trading, social networks, and so on.
- Many scenarios in IT and other industries cannot wait for decisions to be made after many hours, days or months. Decisions often have to be made on the fly, right now, or with a very small latency.
- Spark Streaming attempts to address above needs. Structured Streaming is Production Ready since Spark 2.2.
- Structured streaming allows you to take the same operations that you perform in batch mode and perform them in a streaming fashion. This reduce development time and allows for incremental development and processing.
- Structured Streaming allows us to rapidly get values out of streaming systems with simple switches. You can write your batch job as a prototype and then convert it to streaming job.

Spark Advantages in Real-time World

- In addition to enabling scalable analysis of high-throughput data, Spark is also a unifying platform. You can use the same APIs for streaming and batch programs.
- Spark's Structured Streaming places special emphasis on uniformity of the development effort. Your code for batch analysis is practically the same as the code for streaming analysis.
- There are other systems that can perform near real-time analysis of large volumes of data.
- Flink (<http://flink.apache.org>) supports faster processing than Spark Streaming at least for now. Flink's set of supporting libraries is perhaps not as rich as Spark's. Also, Flink does not offer consistency, interactive queries and a few other features.
- Storm (<http://storm.apache.org>) claims to be a faster streaming engine than Spark. Storm's programming model is more difficult than Spark's. Storm's API set is limited compared to Spark's.
- If you need a super fast stream processing you have to write your code in C++. Most of the above systems work in environments which (more or less) follow what is called the Lambda Architecture.

Lambda Architecture

- **Lambda architecture** is a data-processing architecture designed to handle massive data by taking advantage of both batch and stream processing methods.
- This approach to architecture attempts to balance latency, throughput, and fault-tolerance by using batch processing to provide comprehensive and accurate views of batch data, while simultaneously using real-time stream processing to provide views of online data.
- The two view outputs may be joined before presentation in an application



Spark Streaming Sources & Destinations

- Data can be ingested from many sources like Kafka, Flume, Twitter, ZeroMQ, Kinesis, or TCP sockets, and can be written to HDFS, RDBMS, NoSQL databases, and regular file systems.
- Data can be processed using complex algorithms expressed with high-level functions like map, reduce, join and window.
- Processed data can be pushed out to filesystems, databases, and live dashboards. One can apply Spark's machine learning and graph processing algorithms on the data streams.



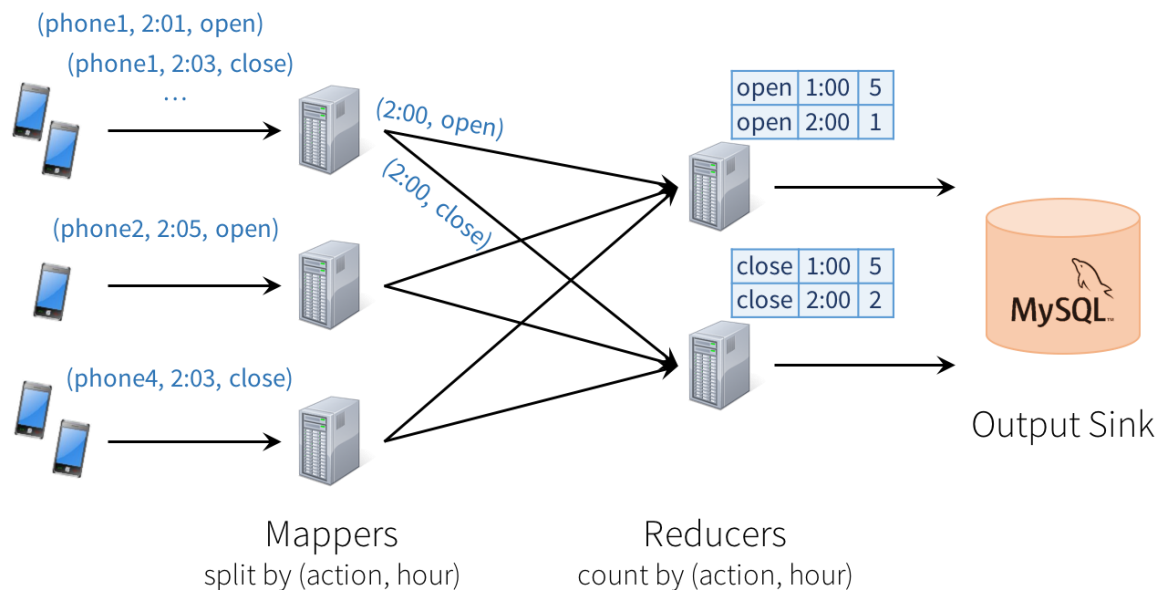
Spark Structured Streaming API

What Streaming Analysis Does

- To start, consider a simple application: we receive (`phone_id`, `time`, `action`) events from a mobile app, and want to count how many actions of each type happened each hour, then store the result in MySQL.
- If we were running this application as a batch job and had a table with all the input events, we could express it as the following SQL query:

```
SELECT action, WINDOW(time, "1 hour"), COUNT(*)  
FROM events  
GROUP BY action, WINDOW(time, "1 hour")
```

In a distributed streaming engine, we might set up nodes to process the data in a “map-reduce” pattern.



Issues in Distributed Systems

- Each node in the first layer reads a partition of the input data (say, the stream from one set of phones), then hashes the events by (action, hour) to send them to a reducer node, which tracks that group's count and periodically updates MySQL. This design can introduce a few challenges:
 - **Consistency:** This distributed design can cause records to be processed in one part of the system before they're processed in another, leading to nonsensical results. For example, suppose our app sends an "open" event when users open it, and a "close" event when closed. If the reducer node responsible for "open" is slower than the one for "close", we might see a *higher total count of "closes" than "opens" in MySQL*, which would not make sense. The image above actually shows one such example.
 - **Fault tolerance:** What happens if one of the mappers or reducers fails? A reducer should not count an action in MySQL twice, but should somehow know how to request old data from the mappers when it comes up. Streaming engines go through a great deal of trouble to provide strong semantics here, at least *within* the engine. In many engines, however, keeping the result consistent in external storage is left to the user.
 - **Out-of-order data:** In the real world, data from different sources can come out of order: for example, a phone might upload its data hours late if it's out of coverage. Just writing the reducer operators to assume data arrives in order of time fields will not work—they need to be prepared to receive out-of-order data, and to update the results in MySQL accordingly.
- In most current streaming systems, some or all of these concerns are left to the user.

Spark Structured Streaming Qualities

- Spark Structured Streaming making a strong guarantee about the system: *at any time, the output of the application is equivalent to executing a batch job on the data received so far*. This is referred to as the *prefix integrity*.
- For example, in our monitoring application, the result table in MySQL will always be equivalent to taking each phone's whatever data made it to the system so far and running the SQL query above.
- There will never be “open” events counted faster than “close” events, duplicate updates on failure, etc. Structured Streaming automatically handles consistency and reliability both within the engine and in interactions with external systems (e.g. updating MySQL transactionally).
- Output tables are **always consistent** with all the records in the received data. For example, as long as each phone uploads its data as a sequential stream (e.g., to the same partition in Apache Kafka), we will always process and count its events in order.
- The **out-of-order data** is handled by the of update of respective row in MySQL. Structured Streaming also supports APIs for filtering out overly old data if the user wants.

Batch vs. Incremental

- The greatest benefit of Structured Streaming is that the API is very easy to use: it is simply Spark's [DataFrame and Dataset](#) API. Users describe the query they want to run, the input and output locations, and optionally a few more details. The system then runs their query incrementally, maintaining enough state to recover from failure, keep the results consistent in external storage, etc.
- For example, here is how to write our streaming monitoring application:

```
// Read data continuously from an S3 location
val inputDF = spark.readStream.json("s3://logs")
// Do operations using the standard DataFrame API and write to MySQL
inputDF.groupBy($"action", window($"time", "1 hour")).count()
    .writeStream.format("jdbc")
    .start("jdbc:mysql://...")
```

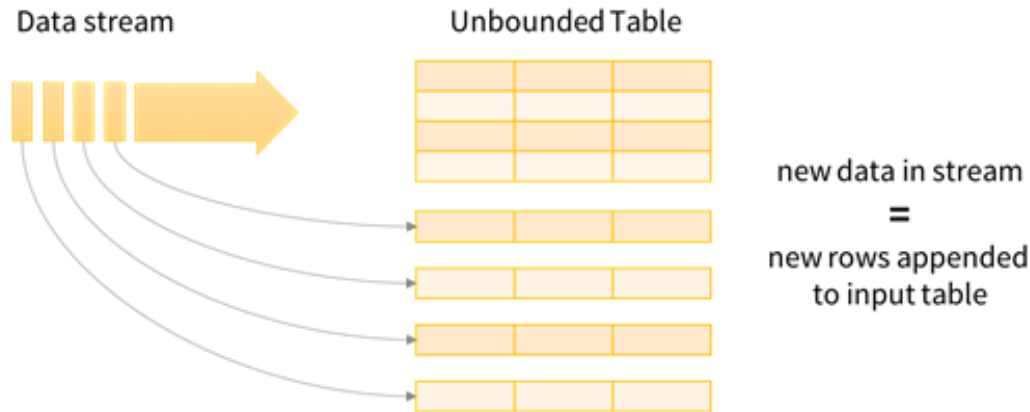
- This is nearly identical to the batch code. The “read” and “write” changed:

```
// Read data once from an S3 location
val inputDF = spark.read.json("s3://logs")

// Do operations using the standard DataFrame API and write to MySQL
inputDF.groupBy($"action", window($"time", "1 hour")).count()
    .write.format("jdbc")
    .save("jdbc:mysql://...")
```

Stream as Unbounded Input Table Model

- Structured Streaming treats all the data arriving as an unbounded **input table**. Each new item in the stream is like a row appended to the input table. Spark does not actually retain all the input, but the results is equivalent to having all of it and running a batch job.



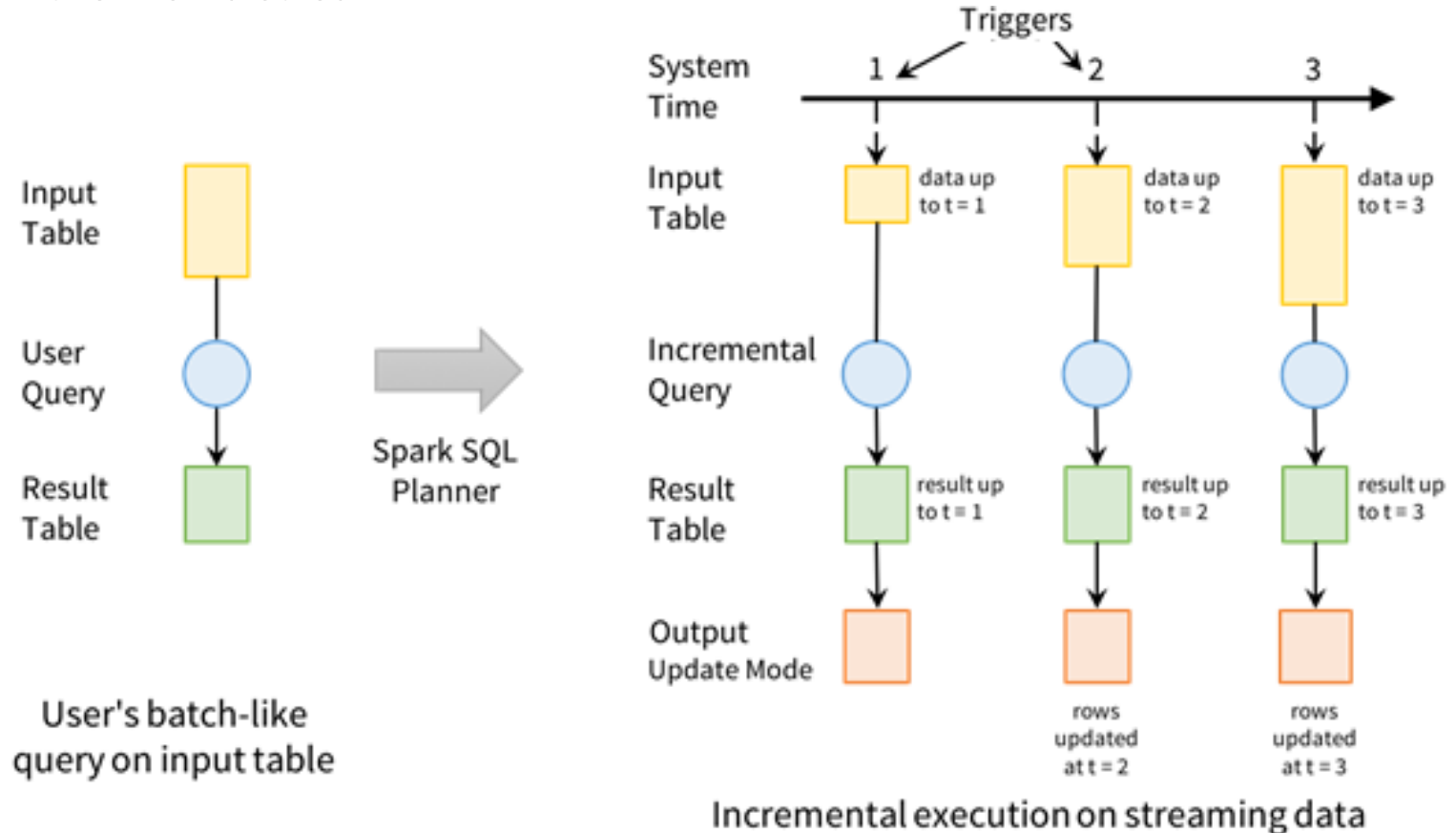
- The developer defines a **query** on this input table, as if it were a static table, to compute a final **result table** that will be written to an output sink.
- Spark automatically converts this batch-like query to a streaming execution plan. This is called *incrementalization*: Spark figures out what state needs to be maintained to update the result each time a record arrives.
- Finally, developers specify **triggers** to control when to update the results. Each time a trigger fires, Spark checks for new data (new row in the input table), and incrementally updates the result.

Output Modes

- Each time the result table is updated, the developer wants to write the changes to an external system, such as S3, HDFS, or a database. We usually want to write output incrementally. For this purpose, Structured Streaming provides three output modes:
 - **Append:** Only the new rows appended to the result table since the last trigger will be written to the external storage. This is applicable only on queries where existing rows in the result table cannot change (e.g. a map on an input stream).
 - **Complete:** The entire updated result table will be written to external storage.
 - **Update:** Only the rows that were updated in the result table since the last trigger will be changed in the external storage. This mode works for output sinks that can be updated in place, such as a MySQL table.

Structured Streaming Processing Model

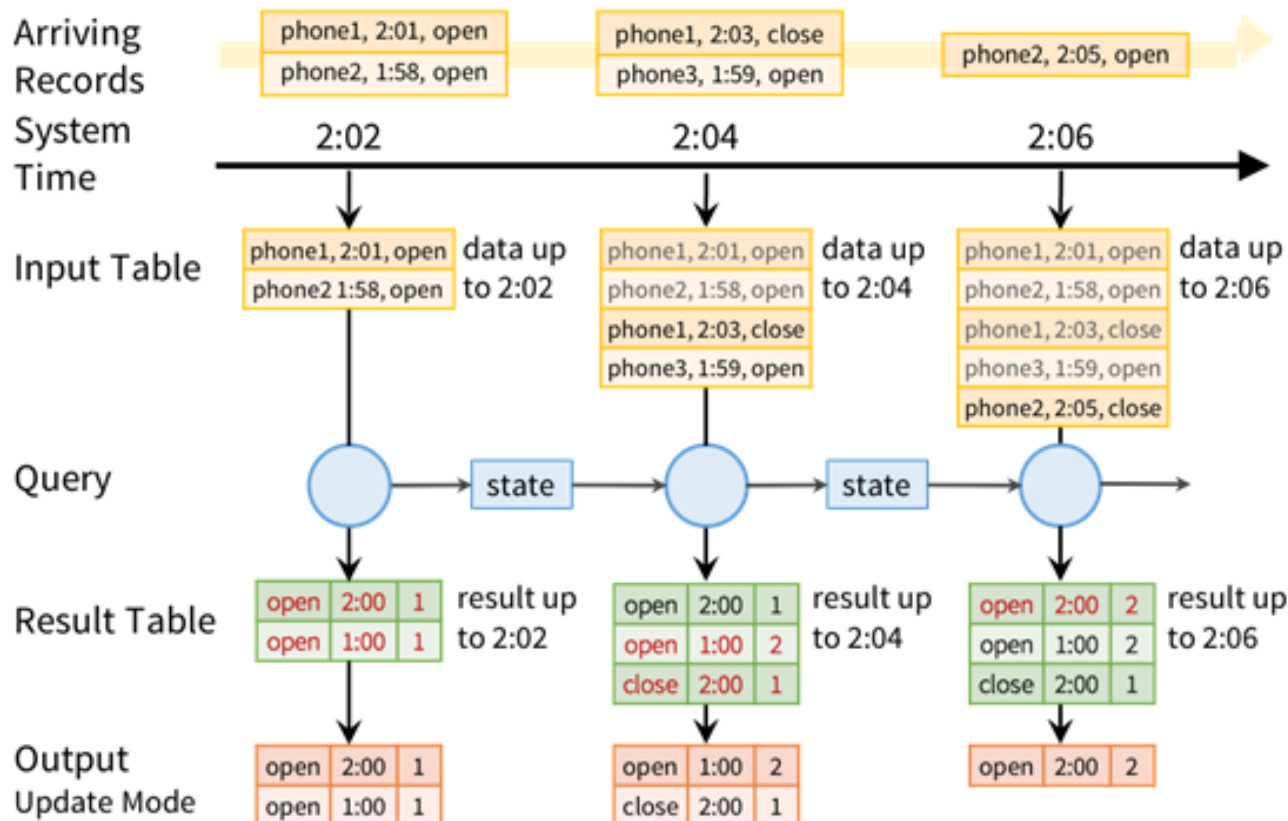
- User expresses queries using batch API. Spark *incrementalizes* those queries to run them on a stream.



- On the following page the batch query computes a count of actions grouped by (action, hour). To run this query incrementally, Spark will maintain some state with the counts for each pair so far, and update when new records arrive.

An Illustration

- The figure below shows this execution using the Update output mode.
- At every trigger point, we take the previous grouped counts and update them with new data that arrived since the last trigger to get a new result table. We then emit only the changes required by our output mode to the sink



Spark automatically handles late data. The “open” event for phone3, which happened at 1:58 on the phone, only gets to the system at 2:02. Even though it’s past 2:00, we update the record for 1:00 in DB. For example, because phone1’s “close” event arrives after its “open” event, we update the “open” count before we update the “close” count.

Fault Requirement & Storage Requirement

Structured Streaming keeps its results valid even if machines fail. To do this, it places two requirements on the input sources and output sinks:

- Input sources must be *replayable*, so that recent data can be re-read if the job crashes. For example, message buses like Amazon Kinesis and Apache Kafka are replayable, as is the file system input source. Only a few minutes' worth of data needs to be retained; Structured Streaming will maintain its own internal state after that.
- Output sinks must support *transactional updates*, so that the system can make a set of records appear atomically. The current version of Structured Streaming implements this for file sinks, and Spark add it for common databases and key-value stores.
- Most Spark applications already use sinks and sources with these properties, because users want their jobs to be reliable.
- Structured Streaming will manage its internal state in a reliable storage system, such as S3 or HDFS, to store data such as the running counts in our example.

API Basic

- Streams in Structured Streaming are represented as DataFrames or Datasets with the `isStreaming` property set to true. You can create them using special read methods from various sources. Suppose we wanted to read data in our monitoring application from JSON files uploaded to HDFS. The code below shows how to do this in Scala:

```
val inputDF = spark.readStream.json("hdfs:///logs.json")
```

- Our resulting DataFrame, `inputDF`, is our input table, which will be continuously extended with new rows as new files are added to the directory. The table has two columns—time and action.
- Now you can use the usual DataFrame/Dataset operations to transform the data. In our example, we want to count action types each hour. To do that we have to group the data by action and 1 hours windows of time.

```
val countsDF = inputDF
    .groupBy($"action", window($"time", "1 hour")).count()
```

- The new DataFrame `countsDF` is our result table, with columns `action`, `window`, and `count`, and will be continuously updated when the query is started. This transformation would give hourly counts even if `inputDF` was a static table.
- Finally, we tell the engine to write this table to a sink and start the streaming computation.

```
val query = countsDF.writeStream.format("jdbc").start("jdbc://...")
```

- The returned query is a `StreamingQuery`, a handle to the active streaming execution and can be used to manage and monitor the execution

Mapping, Filtering and Running Aggregations

- Structured Streaming programs can use DataFrame and Dataset's existing methods to transform data, including map, filter, select, and others.
- In addition, running aggregations or infinite aggregations, such as a count from the beginning of time, are available through the existing APIs.
- Streaming applications often need to compute data on various types of *windows*, including *sliding windows*, which overlap with each other (e.g. a 1-hour window that advances every 5 minutes), and tumbling windows, which do not (e.g. just every hour).
- In Structured Streaming, *windowing is simply represented as a group-by*. Each input event can be mapped to one or more windows, and simply results in updating one or more result table rows.

Windowed Aggregations on Event Time

- Windows can be specified using the window function in DataFrames. For example, we could change our monitoring job to count actions by sliding windows as follows:

```
inputDF.groupBy($"action", window($"time", "1 hour", "5 minutes"))  
        .count()
```

- Whereas our previous application outputted results of the form (hour, action, count), this new one will output results of the form (window, action, count), such as l("1:10-2:10", "open", 17).
- If a late record arrives, we will update all the corresponding windows in MySQL. Windowing is not a special operator for streaming computations; we can run the same code in a batch job to group data in the same way.
- Spark 2.1, added *watermarks*, a feature for dropping overly old data when sufficient time has passed. Without this type of feature, the system might have to track state for all old windows, which would not scale as the application runs.
- In addition, Spark will add support for *session-based windows*, i.e. grouping the events from one source into variable-length sessions according to business logic.

Joining Streams and Static Data

- Because Structured Streaming simply uses the DataFrame API, it is straightforward to join a stream against a static DataFrame, such as an Apache Hive table:

```
// Bring in data about customers from a static "customers" table,  
// then join it with a streaming DataFrame  
val customersDF = spark.table("customers")  
inputDF.join(customersDF, "customer_id")  
    .groupBy($"customer_name", hour($"time"))  
    .count()
```

- The static DataFrame could itself be computed using a Spark query, allowing us to mix batch and streaming computations.

Interactive Queries

- Structured Streaming can expose results directly to interactive queries through Spark's JDBC server. In Spark 2.0, there is a rudimentary "memory" output sink for this purpose that is not designed for large data volumes. However, in future releases, this will let you write query results to an in-memory Spark SQL table, and run queries directly against it.

```
// Save our previous counts query to an in-memory table
countsDF.writeStream.format("memory")
    .queryName("counts")
    .outputMode("complete")
    .start()

// Then any thread can query the table using SQL
sql("select sum(count) from counts where action='login'")
```

Spark vs. Other Streaming Engine

- Structured Streaming's strong guarantee of prefix integrity makes it equivalent to batch jobs and easy to integrate into larger applications.
- Structured Streaming has several advantages over similar engines.

Property	Structured Streaming	Spark Streaming	Apache Storm	Apache Flink	Kafka Streams	Google Dataflow
Streaming API	incrementalize batch queries	integrates with batch	separate from batch	separate from batch	separate from batch	integrates with batch
Prefix Integrity Guarantee	✓	✓	✗	✗	✗	✗
Internal Processing	exactly once	exactly once	at least once	exactly once	at least once	exactly once
Transactional Sources/Sinks	✓	some	some	some	✗	✗
Interactive Queries	✓	✓	✗	✗	✗	✗
Joins with Static Data	✓	✓	✗	✗	✗	✗

Spark Streaming API

Discretized Streams

- Spark has the concept of RDDs. Spark Streaming provides an abstraction called *DStreams*, or *Discretized Streams*.
- A `DStream` is a sequence of data arriving over time, i.e. a sequence of RDDs arriving at each time step (hence the name "discretized").
- `DStreams` can be created from various input sources, such as Flume, Kafka, AWS Kinesis, Twitter or HDFS.
- `DStreams` offer two types of operations: *transformations*, which yield a new `DStream`, and *output operations*, which write data to an external systems, HDFS, Dashboards or Databases.
- `DStreams` provide many of the same operations available on RDDs, plus new operations related to time, such as sliding windows.
- Unlike batch programs, Spark Streaming applications need additional setup in order to operate 24/7.
- *Checkpointing* is the main mechanism Spark Streaming provides for reliable operation and restart operations. `Checkpointing` stores data in a reliable file system such as HDFS.

Internals of Data Stream

- Spark Streaming receives live input data streams and divides the data into "micro" batches, which are then processed by the Spark engine to generate the final stream of results in batches.



- Discretized stream or `DStream` consumes a continuous stream of data but treats it in a discretized manner.
- One can write Spark Streaming programs in Scala, Java or Python (R?).
- There are a few API calls that are either different or yet not available in Python.

All Spark examples are on the GitHub

- Examples are useful. You are learning. Python examples are here <https://github.com/apache/spark/tree/master/examples/src/main/python>
- To see examples for all 4 languages go to: <https://github.com/apache/spark/tree/master/examples/src/main>

The screenshot shows the Apache Spark GitHub repository page. At the top, it says "apache / spark" and "mirrored from git://git.apache.org/spark.git". There are buttons for "Watch" (1,617), "Star" (11,779), and "Fork" (11,069). Below these are tabs for "Code", "Pull requests" (451), "Projects" (0), "Pulse", and "Graphs". The "Code" tab is selected, showing the "Branch: master" dropdown and the path "spark / examples / src / main /". There are buttons for "Create new file", "Upload files", "Find file", and "History". A commit message is visible: "hhbyyh committed with Felix Cheung [SPARK-19337][ML][DOC] Documentation and examples for LinearSVC ... Latest commit 280afe0 2 days ago". Below this is a table of files and directories.

..		
java/org/apache/spark/examples	[SPARK-19337][ML][DOC] Documentation and examples for LinearSVC	2 days ago
python	[SPARK-19337][ML][DOC] Documentation and examples for LinearSVC	2 days ago
r	[SPARK-19639][SPARKR][EXAMPLE] Add spark.svmLinear example and update...	6 days ago
resources	[SPARK-16046][DOCS] Aggregations in the Spark SQL programming guide	a month ago
scala/org/apache/spark/examples	[SPARK-19337][ML][DOC] Documentation and examples for LinearSVC	2 days ago

- To download all examples at once you have to go to: <https://github.com/apache/spark> and clone or download the repository.
- The examples directory will have pom.xml, Maven directives file.

NetCat, nc Utility

- One frequent use of `nc` utility is to verify whether a particular port is active. For example, in order to see whether Zookeeper is listening on its default port 2181, you would do:

```
$ nc -vz localhost 2181
```

```
Connection to localhost 2181 port [tcp/eforward] succeeded!
```

- The response tells you that it does.
- You can also open a two way dialog between terminals.
- In terminal 1, type:

```
$ nc -lk 9998
```

```
Hello, from terminal 1
```

```
Hi, from terminal 1
```

```
how are you, from terminal 2
```

- In terminal 2, type

```
$ nc localhost 9998
```

```
Hello, from terminal 1
```

```
Hi, from terminal 1
```

- Whatever you type in terminal 2, like 'how are you, from terminal 2' will appear in terminal 1.
- To learn more about `nc` command, type `$ man nc >> help_nc.txt` and then read.

network-count.py, Creating StreamingContext

- We start our script `network-count.py` by importing `StreamingContext`, which is the main entry point for all streaming functionality.
- We create a local `StreamingContext` with two execution threads, and batch interval of 1 second.

```
from pyspark import SparkContext
from pyspark.streaming import StreamingContext
# StreamingContext with 2 threads and batch interval of 1 second
sc = SparkContext("local[2]", "NetworkWordCount")
ssc = StreamingContext(sc, 1)
```

- With `sc` we can create a `DStream` that represents streaming data from a TCP source, specified as hostname (e.g. `localhost`) and port (e.g. `9999`).

```
lines = ssc.socketTextStream(sys.argv[1], sys.argv[2])
```

- `DStream lines` represents the stream of data that will be received from the data server. Each record in this `DStream` is a line of text. Next, we want to split the `lines` by spaces into words.

```
words = lines.flatMap(lambda line: line.split(" "))
```

- Each line will be split into multiple words and the stream of words is represented as the `words DStream`. Next, we want to count these words.

Counting Words with `StreamingContext`

Count each word in each batch

```
pairs = words.map(lambda word: (word, 1))
```

```
wordCounts = pairs.reduceByKey(lambda x, y: x + y)
```

- `DStream words` is mapped (one-to-one transformation) to a `DStream` of `(word, 1)` pairs, which is then reduced to get the frequency of words in each batch of data. Finally, `wordCounts.pprint()` will print a few of the counts generated every second.

Print the first ten elements of each RDD

generated in this DStream to the console

```
wordCounts.pprint()
```

- Note that when these lines are executed, Spark Streaming only sets up the computation it will perform when it is started. No real processing has started yet.
- To start the processing after all the transformations have been setup, we finally call

```
ssc.start()    # Start the computation
```

```
ssc.awaitTermination()    # Wait for the computation to terminate
```

network-count.py

- Tidied up code of network-count.py program reads:

```
from __future__ import print_function
import sys
from pyspark import SparkContext
from pyspark.streaming import StreamingContext

if __name__ == "__main__":
    if len(sys.argv) != 3:
        print("Usage: network_wordcount.py <hostname> <port>", file=sys.stderr)
        exit(-1)
    sc = SparkContext(appName="PythonStreamingNetworkWordCount")
    ssc = StreamingContext(sc, 1)

    lines = ssc.socketTextStream(sys.argv[1], int(sys.argv[2]))
    counts = lines.flatMap(lambda line: line.split(" "))\
        .map(lambda word: (word, 1))\
        .reduceByKey(lambda a, b: a+b)
    counts.pprint()
    ssc.start()
    ssc.awaitTermination()
```

Run `network-count.py`

- To test our `network-count.py` code we need to generate a stream of words that our `SparkStreaming` program could intercept and count.
- For that purpose we will use Linux utility `Netcat`. We start `Netcat` by issuing the following command:

```
$ nc -lk 9999
```

- On the lines that follow we will type short sentences and keep hitting returns. `Netcat` will send those sentences to TCP port 9999 for anyone who wants to listen to them.

```
some text
```

```
some more text
```

```
even more words
```

```
-----  
Time: 2017-02-23 14:35:28  
-----
```

```
(u'test', 2)  
(u'word', 1)  
(u'again', 1)  
(u'words', 1)  
(u'agina', 1)  
-----
```

```
Time: 2017-02-23 14:35:29  
-----
```

- In another Linux window, type:

```
$ spark-submit --master local[4] network-count.py localhost 9999
```

Why local[4]

- You might want to start spark master with more threads:

```
/opt/spark/sbin sudo ./start-master.sh -host localhost[4]
```

- When running streaming jobs, you should specify more executors while submitting the application. For example:

```
/opt/spark/sbin/spark-submit --master local[4] your_file.py
```

From Learning Spark book, chapter 10:

- Do not run Spark Streaming programs locally with master configured as local or local[1]. This allocates only one CPU for tasks and if a receiver is running on it, there is no resource left to process the received data.
- Use at least local[2] to have more cores. Otherwise, there are not enough threads to both execute a task and print out the results

Start master and slave

- To make the script work on some systems I had to start Spark Master and the Slave.
- If on Cloudera VM, and you want to use HDFS for your Spark jobs, go to `/etc/init.d` directory and make sure all important services are working
- Then go to `/usr/lib/spark/sbin` directory and run

```
$ sudo ./start-all.sh --host localhost[4]    # 2 dashes - -
```
- If you do `$ ps -ef` you will see Spark Master and Spark Worker (Slave) running.
- Sometimes after above command you might notice that some services like `spark-master` are already running what might appear as an error message. If you want a clean start, first run `stop-all.sh` and then `start-all.sh`.
- You can do the same in other languages Spark supports: Java and Scala.
- R is for some reason missing from Spark Streaming documentation both for release 1.6.x and the most recent 2.1.0. Perhaps R will be added later.
- To verify whether your Spark is up and well on Cloudera VM go to your localhost, ports: 18080 (Master) and 18081 (Worker). On your own VM ports are 8080 (Master) and 8081 (Worker).

Spark Master on Port 18080

Spark Master at spark://... x +

quickstart.cloudera:18080

Cloudera Hue Hadoop HBase Impala Spark Solr Oozie Cloudera Manager Getting Started



Spark Master at spark://192.168.76.158:7077

URL: spark://192.168.76.158:7077

REST URL: spark://192.168.76.158:6066 (*cluster mode*)

Alive Workers: 0

Cores in use: 0 Total, 0 Used

Memory in use: 0.0 B Total, 0.0 B Used

Applications: 0 Running, 0 Completed

Drivers: 0 Running, 0 Completed

Status: ALIVE

Workers

Worker Id	Address	State	Cores	Memory
-----------	---------	-------	-------	--------

Running Applications

Application ID	Name	Cores	Memory per Node	Submitted Time	User	State	Du
----------------	------	-------	-----------------	----------------	------	-------	----


Spark Worker at port 18081

Spark Worker at 192.168... x

quickstart.cloudera:18081

Search

Cloudera Hue Hadoop HBase Impala Spark Solr Oozie Cloudera Manager Getting Star

 1.5.0-cdh5.6.0

Spark Worker at 192.168.76.158:7078

ID: worker-20160324193927-192.168.76.158-7078

Master URL:

Cores: 8 (0 Used)

Memory: 6.7 GB (0.0 B Used)

[Back to Master](#)

Running Executors (0)

ExecutorID	Cores	State	Memory	Job Details
------------	-------	-------	--------	-------------

Structured Streaming API Version

- Spark Streaming version of the same NetworkWordCount program reads:

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import explode
from pyspark.sql.functions import split

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

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

# Split the lines into words
words = lines.select( explode( split(lines.value, " ")).alias("word"))

# Generate running word count
wordCounts = words.groupBy("word").count()

# Start running the query that prints the running counts to the console
query = wordCounts.writeStream.outputMode("complete").format("console") \
    .start()

query.awaitTermination()
```

JavaNetworkWordCount.java

- We start Spark Streaming program by creating a `JavaStreamingContext` object, which is the main entry point for all streaming functionality. In the following we are creating a local `StreamingContext` with two execution threads, and a batch interval of 1 second.

```
import org.apache.spark.*;
import org.apache.spark.api.java.function.*;
import org.apache.spark.streaming.*;
import org.apache.spark.streaming.api.java.*;
import scala.Tuple2;
public class JavaNetworkWordCount
// Local StreamingContext with two working thread and batch interval of 1 second
SparkConf conf = new SparkConf().setMaster("local[2]").setAppName("NetworkWordCount");
JavaStreamingContext jssc = new JavaStreamingContext(conf, Durations.seconds(1));
```

- Using this context, we can create a `DStream` that represents streaming data from a TCP source, specified as hostname (e.g. localhost) and port (e.g. 9999).

```
JavaReceiverInputDStream<String> lines = jssc.socketTextStream("localhost", 9999);
```

- `JavaReceiverInputDStream` represents the stream of data received from the data server.
- Each record in this stream is a line of text. Then, we split the lines by space into words.

```
JavaDStream<String> words = lines.flatMap(
    new FlatMapFunction<String, String>() {
        @Override public Iterable<String> call(String x) {
            return Arrays.asList(x.split(" "));
        }
    });
```

- `flatMap` is a `DStream` operation that creates a new `DStream` with multiple records from each record in the source `DStream`. Each line will be split into multiple words. This transformation is performed by `FlatMapFunction`, a Java convenience class defining `DStream` transformation.

JavaNetworkWordCount, counting words

- Subsequently, we count those words

```
JavaPairDStream<String, Integer> pairs = words.map(  
    new PairFunction<String, String, Integer>() {  
        @Override public Tuple2<String, Integer> call(String s) throws  
Exception {  
            return new Tuple2<String, Integer>(s, 1);  
        }  
    });  
JavaPairDStream<String, Integer> wordCounts = pairs.reduceByKey(  
    new Function2<Integer, Integer, Integer>() {  
        @Override public Integer call(Integer i1, Integer i2) throws Exception  
{  
            return i1 + i2;  
        }  
    });  
wordCounts.print();    // Print a few of the counts to the console
```

- The `words DStream` is mapped (one-to-one transformation) to a `DStream` of `(word, 1)` pairs, using a `PairFunction` object. Then, it is reduced to get the frequency of words in each batch of data, using a `Function2` object.
- Finally, `wordCounts.print()` will print a few of the counts generated every second.

Start processing and Await Termination

- When the above lines are executed, Spark Streaming only sets up the computation it will perform when it is started, and no real processing has started yet.
- To start the processing after all the transformations have been setup, we finally call

```
jssc.start();           // Start the computation
jssc.awaitTermination(); // Wait for the computation to terminate
```

- You run this Java example by compiling it in a Maven project, and export a runnable jar. Start `$ nc -lk 9999` as before.
- You can also use :

```
/usr/lib/spark/bin/run-example streaming.JavaNetworkWordCount
localhost 9999
```

- You stop both the TCP server and your counter by issuing `Ctrl C` in respective Linux window.

Run Examples

- Now that we went through all these troubles we are confident that our Java code compiles and (hopefully) works.
- To run our `JavaNetworkWordCount` java class we can use Eclipse to create Java configuration, export it as a jar file and then use `spark-submit` command to run that jar.
- For now, we can also use `/usr/lib/spark/bin/run-example` command. It is a just simpler. We should type, on the single line:

```
$ /usr/lib/spark/bin/run-example  
org.apache.spark.streaming.examples.JavaNetworkWordCount local[2]  
localhost 9999
```

- `local[2]` means that we will run 2 threads on the local machine and fetch whatever is pushed by TCP protocol to port 9999.
- To push data (words, text) to port 9999 we use Linux utility NetCat: `nc`

```
$ man nc  
nc [-46DdhklmrStUuvzC] [-i interval] [-p source_port]  
    [-s source_ip_address] [-T ToS]  
    [-w timeout] [-X proxy_protocol] [-x proxy_address[:port]]  
[hostname] [port[s]]
```

DESCRIPTION

The `nc` (or `netcat`) utility is used for just about anything under the sun involving TCP or UDP. It can open TCP connections, send UDP packets, listen on arbitrary TCP and UDP ports, do port scanning, and deal with both IPv4 and IPv6. Unlike `telnet(1)`, `nc` scripts nicely, and separates error messages onto standard error instead of sending them to standard output, as `telnet(1)` does with some.

Run spark-submit

- You may export your Eclipse configuration as file `networkcount.jar` and transfer it from Windows to Cloudera VM.
- In the directory where that file resides you type:

```
$ spark-submit --class  
org.apache.spark.examples.streaming.JavaNetworkWordCount --master  
local[2] networkcount.jar localhost 9999
```

- In another command prompt window I start NetCat and type a bunch of words

```
$ nc -lk 9999
```

```
hello
```

```
word
```

```
how are you
```

```
I am fine
```

```
hello again
```

```
We are so happy to see you
```

- You may try doing this as well:

```
$ cat all-bible | nc -lk 9999
```

- When you kill `spark-submit` job, it will print a bunch of those words, in groups captured in 1 sec intervals.

Test Results

Time: 1357008430000 ms

(hello,1)

(word,1)

Time: 1357008431000 ms

(how,1)

(are,1)

(you,1)

Time: 1357008432000 ms

Time: 1357008433000 ms

(I, 1)

(am, 1)

Stream Initialization

- To initialize a Spark Streaming program in Java, a `JavaStreamingContext` object has to be created, which is the main entry point of all Spark Streaming functionality. A `JavaStreamingContext` object can be created by using

```
new JavaStreamingContext(master, appName, batchInterval,  
[sparkHome], [jars])
```

- The `master` parameter is a standard Spark cluster URL and can be "local" for local testing. The `appName` is a name of your program, which will be shown on your cluster's web UI. The `batchInterval` is the size of the batches.
- Finally, the last two parameters are needed to deploy your code to a cluster if running in distributed mode.
- The underlying `SparkContext` can be accessed as `streamingContext.sparkContext`.
- The batch interval must be set based on the latency requirements of your application and available cluster resources.

Example order.txt

- Let us consider a more practical example.
- Clients (`ClientID`) of a brokerage firm place orders (`OrderID`) to buy or sell (`SellOrBuy`) stocks (`StockSymbol`) in a quantity (`NoStocksTraded`) at certain price (`Price`).

Timestamp	OrderID	ClientID	StockSymbol	NoStocksTraded	Price	BuyOrSell
3/22/2016 20:55	1	80	EPE	710	51	B
3/22/2016 20:25	2	70	NFLX	158	8	B
3/22/2016 20:25	3	53	VALE	284	5	B
3/22/2016 20:25	4	14	SRPT	183	34	B
3/22/2016 20:25	5	62	BP	241	36	S
3/22/2016 20:25	6	52	MNKD	296	28	S
3/22/2016 20:25	7	65	CHK	791	60	B
3/22/2016 20:25	8	51	Z	620	21	B
3/22/2016 20:25	9	98	CHK	533	38	S
3/22/2016 20:25	10	97	AU	456	37	B
3/22/2016 20:25	11	5	CTRE	552	15	B
3/22/2016 20:25	12	92	WLL	701	67	B
3/22/2016 20:25	13	92	SRPT	593	23	B

- File orders.txt contains some 500,000 of such trade records.
- This is the data that we will send into our streaming application.

Ingestion of Data

- We will first work with data placed into HDFS by some outside agents in the form of small files.
- Spark has a listener that can read specified HDFS directories and would automatically read any new file dropped into the monitored directories. That particular listener would not read updates to files.
- Later we will work with another practical tools, Kafka, a distributed messaging tool which could transmit messages (data) directly to Spark Streaming jobs.
- Our Demo data contains records of buying and selling activities at a fictional brokerage firm.
- Please note: Real people working at real brokerage firms suffer from an excessive divorce rate, high incidents of alcoholism and drug abuse.

Simulating Data Traffic

- Spark `StreamingContext` object provides the `textFileStream` method, which monitors a directory (any Hadoop-compliant directory, such as HDFS, S3, GlusterFS, and any local directory) and reads each newly created file in the directory.
- The method takes only one argument: the name of the directory to be monitored. Newly created means it won't process the files that already exist in the folder when the streaming context starts, nor will it react to data that is added to a file.
- `textFileStreams` processes only the files copied to the named folder after processing starts. Files have to be moved in the designated directory from the same file system.
- To simulate gradual arrival of 500,000 events over a period of time we will use a Linux shell script named `splitAndSend.sh`, which splits the data file (`orders.txt`) into 50 files, each containing 10,000 lines.
- The shell script then periodically moves the splits to an HDFS directory (supplied as an argument), waiting for three seconds after copying each split.

splitAndSend.sh 1st Version

- Let us first examine what the shell script `splitAndSend.sh` does.

```
#!/bin/bash
if [ -z "$1" ]; then
    echo "Missing output folder name"
    exit 1
fi
# orders.txt is the name of the input file
split -l 10000 orders.txt chunk # chunk is prefix of new file names
for f in `ls chunk*`; do
    if [ "$2" == "local" ];then
    then
        mv $f $1
    else
        hadoop fs -put $f $1/ # the last slash, "/", is significant
        rm -f $f
    fi
    sleep 3 # sleep 3 or any number of seconds.
done
```

- To learn more about shell command `split`, type `$ man split` on you Linux or Cygwin prompt. Make the script executable:

```
$ chmod +x splitAndSend.sh # Invoke it as:
$ ./splitAndSend.sh input local # if sending chunks to a local directory input or
$ ./splitAndSend.sh input # if sending to HDFS directory input
```

- In the first case, chunks are sent to local directory `input`, in the second case to HDFS directory `input`. Note, `./` in front of the script name tells the shell "script is here in `./`".
- New chunks will have file names: `chunkab`, `chunkac`, `chunkad`, etc

Race Condition

Communication by Willard Williamson (student in cscie63, 2016):

- As presented on previous slide, script `splitAndSend.sh` leads to a race condition.
- Note that the documentation for `textFileStream()` indicates that the file must be written to the monitored directory by "moving" it from another location within the same file system. Check the full documentation here: <https://spark.apache.org/docs/1.5.2/api/python/pyspark.streaming.html>.
- In keeping with the API documentation, the problem can be fixed by modifying the `splitAndSend.sh` script as follows. Instead of copying directly to the `input` directory, one first copies to a Hadoop "staging" directory and then move from the `staging` directory into the `input` directory.
- The "`hadoop fs -mv staging/$f input`" creates an atomic file operation in the Hadoop file system and solves the race problem.
- This also could also decrease the script delay to 1 second which decreases test time. The updated script follows:

No Race Condition, splitAndSend.sh, 2nd Version

- The main concern is that by copying directly from local file system to the Hadoop "input" directory (\$1 argument of the script) there are cases where Spark tries to read the file right after it's name was changed. It turns out that while the file is being copied, it has an extension that is something like "chunkah._COPYING_" .
- When the file is finished copying, the extension is changed. So, if Spark sees the new temp file and then tries to read it just after its name was changed, it causes a File not Found Error and the script exits. The move operation is apparently atomic in Hadoop and solved the race condition problem.

```
#!/bin/bash

if [ -z "$1" ]; then
    echo "Missing output folder name"
    exit 1
fi

split -l 10000 orders.txt chunk

for f in `ls chunk*`; do
    if [ "$2" == "local" ]; then
        mv $f $1
    else
        hadoop fs -put $f /user/cloudera/staging
        sleep 3
        hadoop fs -mv /user/cloudera/staging/$f /user/cloudera/$1/
        rm -f $f
    fi
done
```

Process orders, count buys

- We will apply `flatMap()` and not `map()` on every line of `fileStream` because `flatMap()` is more robust and will continue working past any badly formatted line.

```
orders = filestream.flatMap(parseOrder)
```

- Next, we might want to calculate how many buy orders were there per observation interval. We map orders to 2-element tuples. `DStreams` with 2-element tuples are automatically transformed to `PairDStreamFunctions` objects which have functions like `reduceByKey`

```
from operator import add
```

```
numPerType = orders.map(lambda o: (o['buy'], 1L)).reduceByKey(add)
```

- If we re running this on many machines, we want all data back to one.
- We accomplish that with `repartition(1)`. Then we use `DStream`'s `saveAsText()` to save result as txt file.

```
numPerType.repartition(1).saveAsTextFiles(  
    "hdfs:///user/cloudera/output/output", "txt")  
ssc.start()  
ssc.awaitTermination()
```

Run the job, count-buys.py

- Create HDFS directory `/user/cloudera/input`

```
$ hadoop fs -mkdir input
```

- In one of your Linux windows start `count-buys.py` application:

```
$ spark-submit --master local[4] count-buys.py
```

```
SLF4J: Class path contains multiple SLF4J bindings.
```

```
SLF4J: Found binding in [jar:file:/usr/lib/zookeeper/lib/slf4j-log4j12-1.7.5.jar!/org/slf4j/impl/StaticLoggerBinder.class] ...  
.
```

- Application is not hanging. Every 3 seconds it will fetch new file(s) (chunks) from HDFS directory `/user/cloudera/splits` if there are any.
- In another Linux window, from the directory where you have your `orders.txt` file, run the following command:

```
$ ./splitAndShip.sh splits      (Note: "./" are important)
```

- You can examine what is going by querying HDFS directories `splits` and `output`.

```
$ hadoop fs -ls splits
```

```
$ hadoop fs -ls output
```

HDFS splits and output Directories

- `hadoop fs -ls splits` gives you a list of chunks, each with 10,000 lines.
- **Hadoop `fs -ls output` will give you a list of directories with names like:**

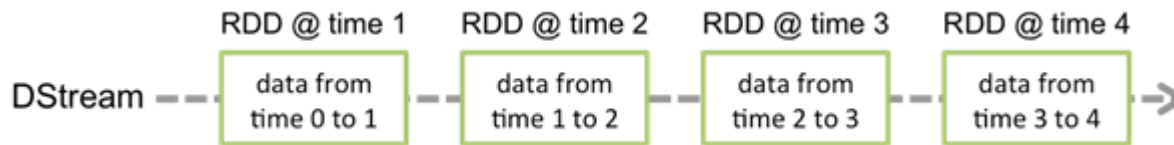
```
drwxr-xr-x - cloudera cloudera output/output-1487963415000.txt
drwxr-xr-x - cloudera cloudera output/output-1487963420000.txt
drwxr-xr-x - cloudera cloudera output/output-1487963425000.txt
```
- **Directory name suffix in the form 1487963415000 represent the number of milliseconds since the beginning of time (1971?), i.e. the moment the directory was created. If you look carefully you will see that files are space 5,000 milliseconds(5 seconds) apart.**
- **Every directory contains two files: `-SUCCESS` and `part-00000`**
- **Some `part-00000` files are empty. Those that are not empty contain the result of counting of stock buys in the particular (5) second interval. The result is in the form:**

```
(False, 4962L)
(True, 5038L)
```
- **In the examined 5 seconds there were 5038 buys and 4962 sells.**
- **You might find it convenient to transfer the content of HDFS output directory to the shared folder and look at it in your host OS.**

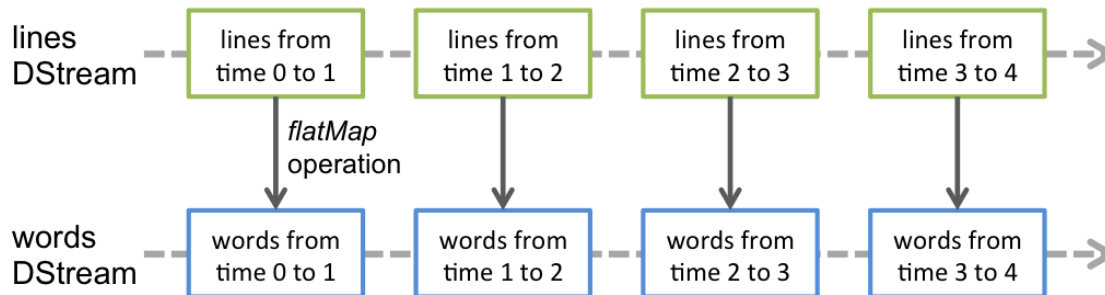
```
$ hadoop fs -get output/* /mnt/hgfs/shared
```

DStream

- Discretized Stream or `DStream` represents a continuous stream of data, the input data stream received from source, or the processed data stream generated by transforming the input stream.
- Internally, it is represented by a continuous sequence of RDDs.
- Each RDD in a `DStream` contains data from a certain interval:



- Any operation applied on a `DStream` translates to operations on the underlying RDDs. For example, in the earlier example of converting a stream of lines to words, the `flatMap()` operation is applied on each RDD in `DStream lines` to generate `DStream words`.



- These underlying RDD transformations are computed by the Spark engine. The `DStream` operations hide most of these details and provides the developer with higher-level API for convenience. These operations are discussed in detail in later sections.

Input Sources

- We have seen `streamingContext.socketTextStream(...)` in the example with `DStream` from text data received over a TCP socket connection.
- Core Spark Streaming API provides methods for creating `DStreams` from files (and Akka actors as input sources). For files, the `DStream` can be created as

```
javaStreamingContext.fileStream(dataDirectory);
```

- Spark Streaming will monitor the directory `dataDirectory` for any Hadoop-compatible filesystem and process any files created in that directory. The files must have the same data format.
- The files must be created in the `dataDirectory` by atomically moving or renaming them into the data directory. Once moved the files must not be changed.
- Additional functionality for creating `DStreams` from sources such as Kafka, Flume, and Twitter can be imported by adding proper dependencies.
- For example, for Kafka, after adding the artifact `spark-streaming-kafka_2.10` to the project dependencies, we can create a `DStream` from Kafka as

```
import org.apache.spark.streaming.kafka.*  
KafkaUtils.createStream(javaStreamingContext, kafkaParams, ...);
```

Operations

- There are two kinds of `DStream` operations - *transformations* and *output operations*.
- Similar to RDD transformations, `DStream` transformations operate on one or more `DStreams` to create new `DStreams` with transformed data.
- After applying a sequence of transformations to the input streams, output operations are called, which write data out to an external data sink, such as a filesystem or a database.
- `DStreams` support many of the transformations available on normal Spark RDD's

Transformations

Transformation	Meaning
<code>map(func)</code>	Return a new DStream by passing each element of the source DStream through a function <i>func</i> .
<code>flatMap(func)</code>	Similar to <code>map</code> , but each input item can be mapped to 0 or more output items.
<code>filter(func)</code>	Return a new DStream by selecting only the records of the source DStream on which <i>func</i> returns true.
<code>repartition(numPartitions)</code>	Changes the level of parallelism in this DStream by creating more or fewer partitions.
<code>union(otherStream)</code>	Return a new DStream that contains the union of the elements in the source DStream and <i>otherDStream</i> .
<code>count()</code>	Return a new DStream of single-element RDDs by counting the number of elements in each RDD of the source DStream.
<code>reduce(func)</code>	Return a new DStream of single-element RDDs by aggregating the elements in each RDD of the source DStream using a function <i>func</i> (which takes two arguments and returns one). The function should be associative so that it can be computed in parallel.
<code>countByKey()</code>	When called on a DStream of elements of type K, return a new DStream of (K, Long) pairs where the value of each key is its frequency in each RDD of the source DStream.
<code>reduceByKey(func, [numTasks])</code>	When called on a DStream of (K, V) pairs, return a new DStream of (K, V) pairs where the values for each key are aggregated using the given reduce function.
<code>join(otherStream, [numTasks])</code>	When called on two DStreams of (K, V) and (K, W) pairs, return a new DStream of (K, (V, W)) pairs with all pairs of elements for each key.
<code>cogroup(otherStream, [numTasks])</code>	When called on DStream of (K, V) and (K, W) pairs, return a new DStream of (K, Seq[V], Seq[W]) tuples.

Transform Operation

- The `transform` operation (along with its variations like `transformWith`) allows arbitrary RDD-to-RDD functions to be applied on a DStream. It can be used to apply any RDD operation that is not exposed in the DStream API. For example, the functionality of joining every batch in a data stream with another dataset is not directly exposed in the DStream API.
- For example, if you want to do real-time data cleaning by joining the input data stream with precomputed spam information and then filtering based on it.

```
// RDD containing spam information
JavaPairRDD<String, Double> spamInfoRDD = javaSparkContext.hadoopFile(...);

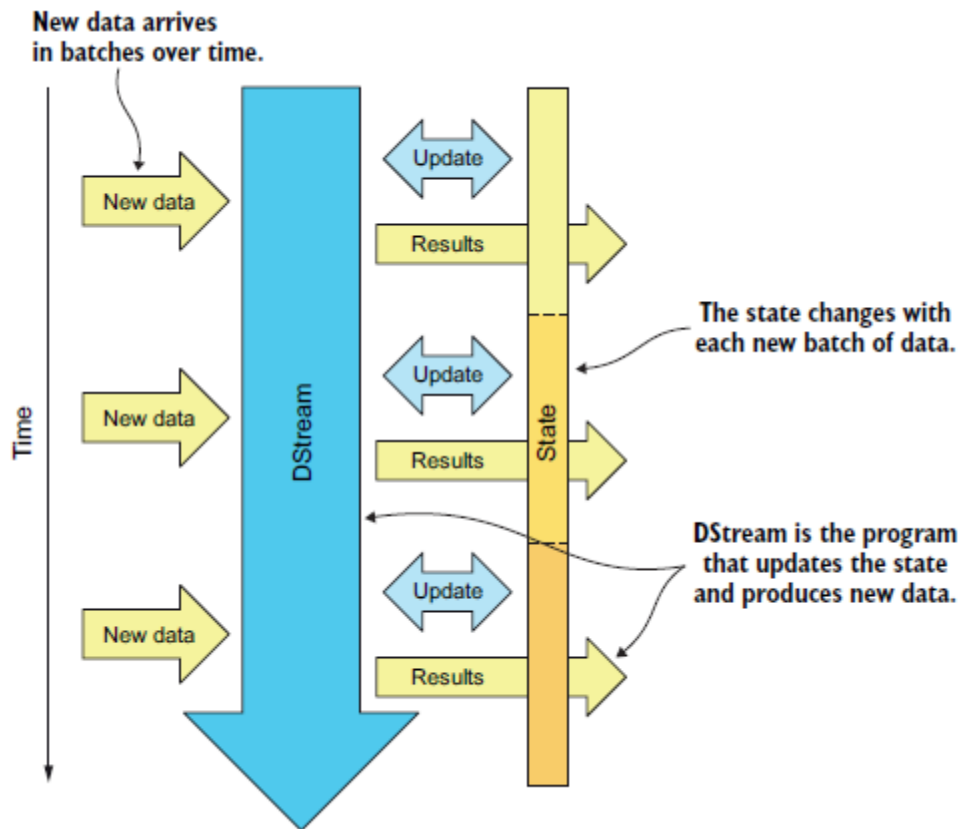
JavaPairDStream<String, Integer> cleanedDStream = inputDStream.transform(
    new Function<JavaPairRDD<String, Integer>, JavaPairRDD<String, Integer>>() {
        @Override public JavaPairRDD<String, Integer> call(JavaPairRDD<String,
Integer> rdd) throws Exception {
            rdd.join(spamInfoRDD).filter(...) // join data stream with spam information
to do data cleaning
            ...
        }
    });
```

- We can also use machine learning and graph computation algorithms in the transform method.

Updating State

- The previous calculations only needed the data from the current mini-batch. Sometime we also need to take into account the data from previous mini-batches.
- To calculate rolling top five clients, you have to keep track of the total dollar amount bought or sold by each client. In other words, you have to keep track of a state that persists over long time and over different mini-batches.
- This principle is illustrated in figure on next slide. New data periodically arrives over time in mini-batches. Each DStream is a program that processes the data and produces results.
- By using Spark Streaming methods to update state, DStreams can combine the persisted data from the state with the new data from the current mini-batch. This results in more powerful streaming programs.

Tracking State over Time



- A `DStream` combines new data arriving in mini-batches with the data from state persisted over time, produces results, and updates the state.

updateStateByKey(), Java

- The `updateStateByKey` operation allows you to maintain arbitrary state while continuously updating it with new information. To use this, you will have to do two steps.
 1. Define the state - The state can be of arbitrary data type.
 2. Define the state update function - Specify with a function how to update the state using the previous state and the new values from input stream.
- For example we want to maintain a running count of each word seen in a text data stream. Here, the running count is the state and it is of type integer.

- We define the update function as

```
Function2<List<Integer>, Optional<Integer>, Optional<Integer>> updateFunction =  
    new Function2<List<Integer>, Optional<Integer>, Optional<Integer>>() {  
        @Override public Optional<Integer> call(List<Integer> values,  
Optional<Integer> state) {  
            Integer newSum = ... // add the new values with the previous running count  
to get the new count  
            return Optional.of(newSum)  
        }  
    }
```

- This is applied on a `DStream` containing words and numbers, (word, 1) pairs in the example `JavaNetworkWordCount`).

```
JavaPairDStream<String, Integer> runningCounts =  
pairs.updateStateByKey(updateFunction);
```

- The update function will be called for each word, with new Values having a sequence of 1's (from the (word, 1) pairs) and the `runningCount` having the previous count.

updateStateByKey() , Python

- In every batch, Spark will apply the state update function for all existing keys, regardless of whether they have new data in a batch or not. If the update function returns `None`, the key-value pair will be ignored.
- For example, we want to maintain a running count of each word seen in a text data stream. Here, the running count is the state of type integer.
- We define the update function as:

```
def updateFunction(newValues, runningCount):
```

```
    if runningCount is None:
```

```
        runningCount = 0
```

```
    return sum(newValues, runningCount)
```

```
# add the new values with the previous
```

```
# running count to get the new count
```

- This is applied on a `DStream` containing pairs (words, numbers) the pairs `DStream` containing `(word, 1)` pairs in an earlier example.

```
runningCounts = pairs.updateStateByKey(updateFunction)
```

- On the following slide we will use `updateStateByKey` to track the number of words appearing in succession of mini-batches.

stateful-wordcount.py

- Script `stateful-wordcount.py` counts words in UTF8 encoded, '`\n`' delimited text received from the network every second.
- **Usage:** `stateful_wordcount.py <hostname> <port>`
- `<hostname>` and `<port>` describe the TCP server that Spark Streaming would connect to receive data.
- To run this on your local machine, you need to first run a `Netcat` server

`$ nc -lk 9999` and then run the counting script

`$ spark-submit -master local[2] stateful_wordcount.py localhost 9999`

From Spark Streaming 1.6.0 Python API Documentation:

`updateStateByKey(updateFunc, numPartitions=None)`

- 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 of the key.
- **Parameters:** `updateFunc` – State update function. If this function returns `None`, then corresponding state key-value pair will be eliminated.

`pprint(num=10)`

- Print the first `num` elements of each RDD generated in this `DStream`. 10 is default
- **Parameters:** `num` – the number of elements from the first will be printed.

stateful-wordcount.py

```
from __future__ import print_function
import sys
from pyspark import SparkContext
from pyspark.streaming import StreamingContext

if __name__ == "__main__":
    if len(sys.argv) != 3:
        print("Usage: stateful_network_wordcount.py <hostname> <port>", file=sys.stderr)
        exit(-1)
    sc = SparkContext(appName="PythonStreamingStatefulNetworkWordCount")
    ssc = StreamingContext(sc, 1)
    ssc.checkpoint("/user/cloudera/checkpoint")
    # HFS directory, is mandatory for stateful streams

    # RDD with initial state (key, value) pairs
    # initialStateRDD = sc.parallelize([(u'hello', 1), (u'world', 1)])

    def updateFunc(new_values, last_sum):
        return sum(new_values) + (last_sum or 0)

    lines = ssc.socketTextStream(sys.argv[1], int(sys.argv[2]))
    running_counts = lines.flatMap(lambda line: line.split(" "))\
        .map(lambda word: (word, 1))\
        .updateStateByKey(updateFunc)
    # .updateStateByKey(updateFunc, initialStateRDD)

    running_counts.pprint()

    ssc.start()
    ssc.awaitTermination()
```

Run `stateful-wordcount.py`

- In one Linux window we run `$ nc -lk 9999` and then start typing repeating words bellow.

hi
hi
hello
hi
hello
new
new
hi
. . .

- In the other Linux window we type:
`$ spark-submit --master local[4] stateful-count.py localhost 9999`
- Our program will write a report every second and will report how many times it has seen every word so far. As we repeat words, the counts will increase.

```
-----  
Time: 2017-02-25 18:59:10  
-----
```

```
(u'hi', 1)  
-----
```

```
Time: 2017-02-25 18:59:11  
-----
```

```
(u'hi', 2)  
-----
```

```
Time: 2017-02-25 18:59:12  
-----
```

```
(u'hi', 2)  
(u'hello', 1)  
-----
```

```
Time: 2017-02-25 18:59:13  
-----
```

```
(u'hi', 3)  
(u'hello', 2)  
(u'new', 1)
```


Find top 5 clients

- The brokerage wants to know the top five clients by their orders in each mini batch. We estimate the value of a client's order by the product:

`noStocksTraded*price`

- The number of stocks traded is contained in value of `hashMap` with the key 'amount' and the price of the stock in the `hashMap` with the key 'price'. Remember when we parsed every line `orders.txt`
- So, we map `orders DStream` into a pair `DStream amountPerClient` with the pair content: `(clientId, valueOfOrder)`

```
amountPerClient = orders.map(  
    lambda o: (o['clientId'],o['amount']*o['price']))
```

- In order to find the total value of a client to the brokerage firm in a mini-batch we will use `updateStateByKey` function. We will store the sum of clients orders in the state `DStream` named `amountState`. That state `DStream` gets updated by new `valueOfOrder` whenever we encounter the same client. All of that is accomplished by this line of code:

```
amountState = amountPerClient.updateStateByKey(lambda vals,  
totalOpt: sum(vals)+totalOpt if totalOpt != None else sum(vals))
```

Find Top 5 Clients

- To identify 5 clients with the highest value of their orders we need to sort `DStream amountState` by the value of client total orders and then select the first five with the largest sum of orders.

- Sorting by the value of the total order is accomplished with this code:

```
amountState.transform(lambda rdd: rdd.sortBy(lambda x: x[1], False))
```

- However, we need to select the first five in the this sorted `DStream`.
- To do that we use function `zipWithIndex()` to add (zip) an index (sequence of values `1, 2, 3, 4, ...`) to the sorted `DStream`. The index is added to the right. Then we will `filter()` the `DStream` and take only those rows which have an index less than 5 (`0, 1, 2, 3, 4`, i.e. 5 of them). This is accomplished with these set of operations:

```
.map(lambda x: x[0]).zipWithIndex().filter(lambda x: x[1] < 5))
```

- The complete transformation of the `DStream amountState` to the `DStream top5clients` is done through a combined action:

```
top5clients = amountState.transform(lambda rdd: rdd.sortBy(lambda x: x[1], False).map(lambda x: x[0]).zipWithIndex().filter(lambda x: x[1] < 5))
```

Painting the Result

- We want to produce a result that tell us what is the number of buys and the number of sells as well as the list of top five clients presented by their `clientIds`

- We create those two lists (`DStreams`) in these lines of code:

```
buySellList = numPerType.map(lambda t: ("BUYS", [str(t[1])]) if t[0]  
else ("SELLS", [str(t[1])]) )
```

```
top5clList = top5clients.repartition(1).map(lambda x:  
str(x[0])).glom().map(lambda arr: ("TOP5CLIENTS", arr))
```

- Function `glom()` coalesces all partitions of an RDD into one.
- We have two `DStreams`: `buySellList` and `top5clList`. To combine them into one output we use method `union()`

```
finalStream = buySellList.union(top5clList)
```

- Finally we repartition `finalStreams` into 1 partition and save results:
`finalStream.repartition(1).saveAsTextFiles("hdfs:///user/cloudera/output/output", "txt")`

- All scripts running stateful processes must perform checkpointing. For that purpose we provide an HDFS directory to the `StreamingContext`.

```
sc.setCheckpointDir("hdfs:///user/cloudera/checkpoint/")
```

- Complete code of the script that finds 5 top clients is given on the following slide:

top-clients.py

```
from pyspark import SparkContext
from pyspark.streaming import StreamingContext
sc = SparkContext(appName="FindTopClients")
ssc = StreamingContext(sc, 3)
filestream = ssc.textFileStream("hdfs:///user/cloudera/input")
from datetime import datetime
def parseOrder(line):
    s = line.split(",")
    try:
        if s[6] != "B" and s[6] != "S":
            raise Exception('Wrong format')
        return [{"time": datetime.strptime(s[0], "%Y-%m-%d %H:%M:%S"), "orderId": long(s[1]),
"clientId": long(s[2]), "symbol": s[3],
        "amount": int(s[4]), "price": float(s[5]), "buy": s[6] == "B"}]
    except Exception as err:
        print("Wrong line format (%s): " % line)
        return []

orders = filestream.flatMap(parseOrder)
from operator import add
numPerType = orders.map(lambda o: (o['buy'], 1L)).reduceByKey(add)
amountPerClient = orders.map(lambda o: (o['clientId'], o['amount']*o['price']))
amountState = amountPerClient.updateStateByKey(lambda vals, totalOpt: sum(vals)+totalOpt if totalOpt
!= None else sum(vals))
top5clients = amountState.transform(lambda rdd: rdd.sortBy(lambda x: x[1], False).map(lambda x:
x[0]).zipWithIndex().filter(lambda x: x[1] < 5))
buySellList = numPerType.map(lambda t: ("BUYS", [str(t[1])]) if t[0] else ("SELLS", [str(t[1])]) )
top5clList = top5clients.repartition(1).map(lambda x: str(x[0])).glom().map(lambda arr:
("TOP5CLIENTS", arr))

finalStream = buySellList.union(top5clList)
finalStream.repartition(1).saveAsTextFiles("hdfs:///user/cloudera/output/output", "txt")
sc.setCheckpointDir("hdfs:///user/cloudera/checkpoint/")
ssc.start()
ssc.awaitTermination()
```

Running `top-clients.py`

- Just like we did with the script `count-buys.py`, we clean HDFS directories input and output, and run bash script `splitAndSend.sh` in the directory where orders file `orders.txt` reside.

- In another Linux window we run

```
$ spark-submit -master local[4] top-clients.py
```

The results will appear in HDFS directories with names like:

```
output/output-1488083181000.txt
```

- In those directories we will find `_SUCCESS` file and `part-00000` file
- When we open one of `part-00000` files, its content will look like this:

```
(False, 4969L)
```

```
(True, 5031L)
```

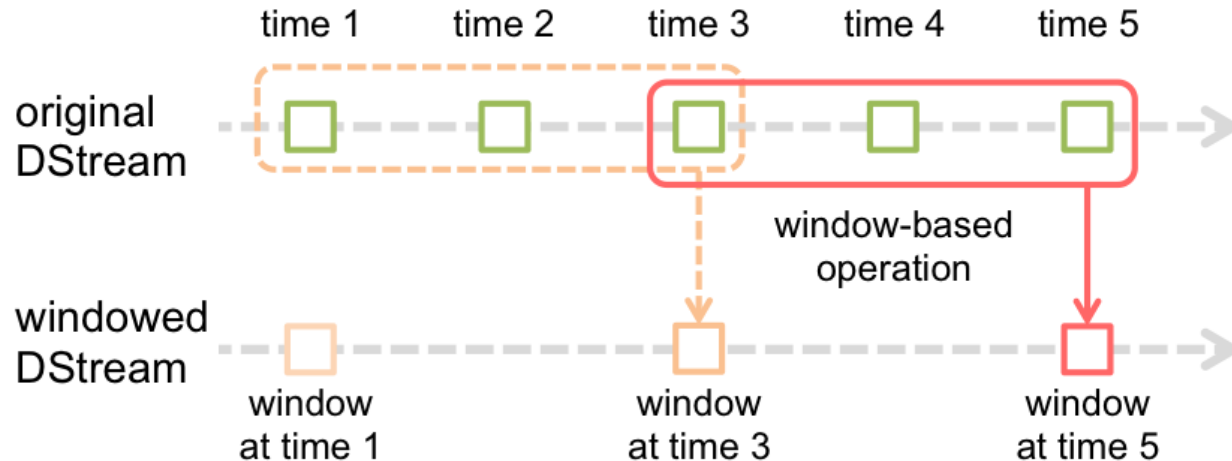
```
('TOP5CLIENTS', ['15', '64', '23', '55', '69'])
```

Finding information for longer periods, Windowing

- Quite often very noisy data, wildly fluctuating from one to another second, are not easy to follow.
- We “tame” such data by using so called rolling averages. We specify a window several times longer than our measurement interval and within that window perform averaging or some other calculations.
- We slide that window by the measurement interval (or a smaller multiple) and present data on a graph that apparently has the same number of steps on the independent axis. However, this graph will be much tamer since the integration or averaging over the longer window will reduce the noise.
- Spark Streaming API has that exact functionality and it is called Windowing or Window operations.

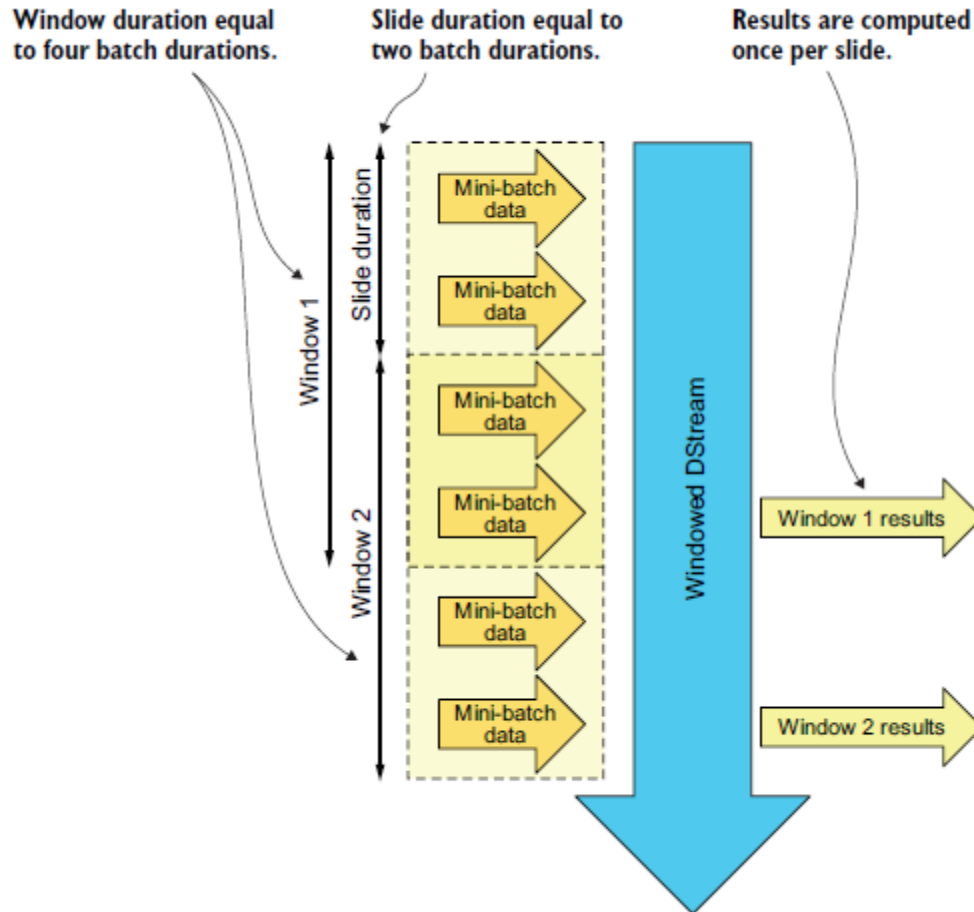
Window Operations

- Spark Streaming also provides window computations, which allow you to apply transformations over a sliding window of data.



- Every time the window slides over a source DStream, the source RDDs that fall within the window are combined and operated upon to produce the RDDs of the windowed DStream. In this figure, the operation is applied over last 3 time units of data, and slides by 2 time units.
- Window-based operation needs to specify two parameters.
 - window length - The duration of the window (3 in the figure)
 - slide interval - The interval at which the window-based operation is performed (2 in the figure).
- These two parameters must be multiples of the batch interval of the source DStream (1 in the figure).

Windowed DStream Processing



- Windowed `DStream` processing data with slide of two mini-batch durations, with window duration of four mini-batch durations.
- Results are computed once per slide.

An Example for window Operation, Java

- Let's say we want to extend the earlier example by generating word counts over last 30 seconds of data, every 10 seconds. To do this, we have to apply the `reduceByKey` operation on the pairs DStream of (word, 1) pairs over the last 30 seconds of data. This is done using the operation `reduceByKeyAndWindow`.

```
// Reducees adding two integers, defined separately for clarity
Function2<Integer, Integer, Integer> reduceFunc = new
Function2<Integer, Integer, Integer>() {
    @Override public Integer call(Integer i1, Integer i2) throws
Exception {
        return i1 + i2;
    }
};
```

- // Reduce last 30 seconds of data, every 10 seconds**

```
JavaPairDStream<String, Integer> windowedWordCounts =
pair.reduceByKeyAndWindow(reduceFunc, new Duration(30000), new
Duration(10000));
```

- All of window operations take the said two parameters – `windowLength` and `slideInterval`.**

window Illustration, Python

- *For example, you want to generate word counts over the last 30 seconds of data, every 10 seconds. To do this, you have to apply the `reduceByKey` operation on the pairs `DStream of (word, 1)` pairs over the last 30 seconds of data. This is done using the operation `reduceByKeyAndWindow`.*

```
# Reduce last 30 seconds of data, every 10 seconds
windowedWordCounts = pairs.reduceByKeyAndWindow(
    lambda x, y: x + y, lambda x, y: x - y, 30, 10)
```

- There are several window transformations. Some operate on ordinary `DStreams` and others only on pair `DStreams` (`byKey` functions). We list them on the next slide.

Common window functions

Transformation	Meaning
<code>window(<i>windowLength</i>, <i>slideInterval</i>)</code>	Return a new DStream which is computed based on windowed batches of the source DStream.
<code>countByWindow(<i>windowLength</i>, <i>slideInterval</i>)</code>	Return a sliding window count of elements in the stream.
<code>reduceByWindow(<i>func</i>, <i>windowLength</i>, <i>slideInterval</i>)</code>	Return a new single-element stream, created by aggregating elements in the stream over a sliding interval using <i>func</i> . The function should be associative so that it can be computed correctly in parallel.
<code>reduceByKeyAndWindow(<i>func</i>, <i>windowLength</i>, <i>slideInterval</i>, [<i>numTasks</i>])</code>	When called on a DStream of (K, V) pairs, returns a new DStream of (K, V) pairs where the values for each key are aggregated using the given reduce function <i>func</i> over batches in a sliding window. Note: By default, this uses Spark's default number of parallel tasks (2 for local machine, 8 for a cluster) to do the grouping. You can pass an optional numTasks argument to set a different number of tasks.
<code>reduceByKeyAndWindow(<i>func</i>, <i>invFunc</i>, <i>windowLength</i>, <i>slideInterval</i>, [<i>numTasks</i>])</code>	A more efficient version of the above reduceByKeyAndWindow() where the reduce value of each window is calculated incrementally using the reduce values of the previous window. This is done by reducing the new data that enter the sliding window, and "inverse reducing" the old data that leave the window. An example would be that of "adding" and "subtracting" counts of keys as the window slides. However, it is applicable to only "invertible reduce functions", that is, those reduce functions which have a corresponding "inverse reduce" function (taken as parameter <i>invFunc</i>). Like in reduceByKeyAndWindow , the number of reduce tasks is configurable through an optional argument.
<code>countByValueAndWindow(<i>windowLength</i>, <i>slideInterval</i>, [<i>numTasks</i>])</code>	When called on a DStream of (K, V) pairs, returns a new DStream of (K, Long) pairs where the value of each key is its frequency within a sliding window. Like in reduceByKeyAndWindow , the number of reduce tasks is configurable through an optional argument.

Practical Objectives, windows-operations.py

- One could use Spark Streaming API to calculate values accumulated over longer periods of time, i.e. many mini-batches.
- We will find the top five most-traded securities during the last hour. This is accomplished using window operations.
- To create windowed DStream we will use `reduceByKeyAndWindow()` method.
- One needs to specify the reduce function and the window duration and optionally the slide duration if different from mini-batch duration.
- We will proceed with the same calculations as in previous examples and then, create a windowed DStream `stocksWindow` that will accumulate pairs of `(symbol, amount)` over a window of duration `60*60` seconds.

```
stocksWindow = orders.map(lambda x: (x['symbol'], x['amount']))  
.window(60*60)
```

- Subsequently we will sum amounts for every stock symbol using `add` operator and `reduceByKey()` function:

```
stocksPerWindow = stocksWindow.reduceByKey(add)
```

- The last operation created new DStream `stocksPerWindow` that contains stocks and aggregate trades (amount-s). In order to find which stocks had the highest traded volume we have to sort previous DStream. Like in the previous example in order to extract 5 stocks on the top of the list we have to use `zipWithIndex()` to transform sorted `stocksPerWindow` DStream

```
topStocks = stocksPerWindow.transform(lambda rdd: rdd.sortBy(lambda x: x[1],  
False).map(lambda x: x[0]).zipWithIndex().filter(lambda x: x[1] <  
5)).repartition(1).\map(lambda x: str(x[0])).glom().  
map(lambda arr: ("TOP5STOCKS", arr))
```

- The complete listing of the script `windows-operations.py` is given on the next slide

windows-operations.py

```
from pyspark.streaming import StreamingContext
ssc = StreamingContext(sc, 3)
filestream = ssc.textFileStream("/user/cloudera/input")
from datetime import datetime
def parseOrder(line):
    s = line.split(",")
    try:
        if s[6] != "B" and s[6] != "S":
            raise Exception('Wrong format')
        return [{"time": datetime.strptime(s[0], "%Y-%m-%d %H:%M:%S"),
"orderId": long(s[1]),
"clientId": long(s[2]), "symbol": s[3],
"amount": int(s[4]), "price": float(s[5]), "buy": s[6] == "B"}]
    except Exception as err:
        print("Wrong line format (%s): " % line)
        return []
orders = filestream.flatMap(parseOrder)
from operator import add
numPerType = orders.map(lambda o: (o['buy'], 1L)).reduceByKey(add)
```

windows-operations.py

```
amountPerClient = orders.map(lambda o: (o['clientId'],
o['amount']*o['price']))
amountState = amountPerClient.updateStateByKey(lambda vals, totalOpt:
sum(vals)+totalOpt
    if totalOpt != None else sum(vals))
top5clients = amountState.transform(lambda rdd: rdd.sortBy(lambda x:
x[1], False).map(lambda x: x[0]).zipWithIndex().filter(lambda x: x[1] <
5))
buySellList = numPerType.map(lambda t: ("BUYS", [str(t[1])]) if t[0]
else ("SELLS", [str(t[1])]) )
top5clList = top5clients.repartition(1).map(lambda x:
str(x[0])).glom().map(lambda arr: ("TOP5CLIENTS", arr))
stocksWindow = orders.map(lambda x: (x['symbol'],
x['amount'])).window(60*60)
stocksPerWindow = stocksWindow.reduceByKey(add)
topStocks = stocksPerWindow.transform(lambda rdd: rdd.sortBy(lambda x:
x[1], False).map(lambda x: x[0]).\
zipWithIndex().filter(lambda x: x[1] < 5)).repartition(1).\
map(lambda x: str(x[0])).glom().\
map(lambda arr: ("TOP5STOCKS", arr))
finalStream = buySellList.union(top5clList).union(topStocks)
finalStream.repartition(1).saveAsTextFiles("/user/cloudera/output/output
", "txt")

sc.setCheckpointDir("/user/cloudera/checkpoint/")
ssc.start()
ssc.awaitTermination()
```

Output Operations

- We perform transformations and operations on RDDs and DStreams in order to extract useful values for our practical or business analysis. Exporting calculated values is done using output operators.
- When an output operator is called, it triggers the computation of a stream.
- In Spark 1.6 and Spark 2.1 the following output operators are defined:

Output Operation	Meaning
<code>print()</code>	Prints first ten elements of every batch of data in a DStream on the driver.
<code>foreachRDD(func)</code>	The fundamental output operator. Applies a function, <i>func</i> , to each RDD generated from the stream. This function should have side effects, such as printing output, saving the RDD to external files, or writing it over the network to an external system.
<code>saveAsObjectFiles(prefix, [suffix])</code>	Save this DStream's contents as a SequenceFile of serialized objects. The file name at each batch interval is generated based on <i>prefix</i> and <i>suffix</i> : " <i>prefix-TIME_IN_MS[.suffix]</i> ".
<code>saveAsTextFiles(prefix, [suffix])</code>	Save this DStream's contents as a text files. The file name at each batch interval is generated based on <i>prefix</i> and <i>suffix</i> : " <i>prefix-TIME_IN_MS[.suffix]</i> ".
<code>saveAsHadoopFiles(prefix, [suffix])</code>	Save this DStream's contents as a Hadoop file. The file name at each batch interval is generated based on <i>prefix</i> and <i>suffix</i> : " <i>prefix-TIME_IN_MS[.suffix]</i> ".

Persistence

- Similar to `RDDs`, `DStreams` also allow developers to persist the stream's data in memory.
- `persist()` method applied on a `DStream` would automatically persist every `RDD` of that `DStream` in memory.
- Persisting a `DStream` is useful if the data in the `DStream` will be computed multiple times (e.g., multiple operations on the same data).
- 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()`.
- For input streams that receive data over the network (such as, Kafka, Flume, sockets, etc.), the default persistence level is set to replicate the data to two nodes for fault-tolerance.
- Unlike `RDDs`, the default persistence level of `DStreams` keeps the data serialized in memory.

RDD Checkpointing

- A stateful operation is one which operates over multiple batches of data. This includes all window-based operations and the `updateStateByKey` operation. Since stateful operations have a dependency on previous batches of data, they continuously accumulate metadata over time. To clear this metadata, streaming supports periodic checkpointing by saving intermediate data to HDFS.
- The checkpointing incurs the cost of saving to HDFS which may cause the corresponding batch to take longer to process. The checkpointing interval needs to be set carefully.
- At small batch sizes (1 second), checkpointing every batch may significantly reduce operation throughput. Conversely, checkpointing too infrequently causes the lineage and task sizes to grow which may have detrimental effects.
- Typically, a checkpoint interval of 5 - 10 times of sliding interval of a `DStream` is good setting to try.
- To enable checkpointing, the developer has to provide the HDFS path to which RDD will be saved. This is done by using

```
ssc.checkpoint(hdfsPath)
```

- `ssc` is the `StreamingContext` or `JavaStreamingContext`
- The interval of checkpointing of a `DStream` can be set by using

```
dstream.checkpoint(checkpointInterval)
```

- For `DStreams` that must be checkpointed (that is, `DStreams` created by `updateStateByKey` and `reduceByKeyAndWindow` with inverse function), the checkpoint interval of the `DStream` is by default set to a multiple of the `DStream`'s sliding interval such that its at least 10 seconds.

Checkpointing is mandatory for stateful streaming

- From the Spark documentation:
 - *A streaming application must operate 24/7 and hence must be resilient to failures unrelated to the application logic (e.g., system failures, JVM crashes, etc.). For this to be possible, Spark Streaming needs to checkpoint enough information to a fault- tolerant storage system such that it can recover from failures.*
- Spark's mechanism of checkpointing is the way of guaranteeing fault tolerance through the lifetime of our spark job. When we're operating 24/7, things will fail that might not be directly under our control, such as a network failure or datacenter crashes. To promise a clean way of recovery, Spark can checkpoint our data every interval of our choosing to a persistent data store, such as Amazon S3, HDFS or Azure Blob Storage, if we tell it to do so.
- Checkpointing is a *feature* for any non-stateful transformation, but **it is mandatory that you provide a checkpointing directory for stateful streams**, otherwise your application won't be able to start.
- Providing a checkpoint directory is as easy as calling the `StreamingContext` with the directory location:

```
val sparkContext = new SparkContext()
val ssc = new StreamingContext(sparkContext, Duration(4000))
ssc.checkpoint("path/to/persistent/storage")
```

- One important thing to be noted is that **checkpointed data is only useable as long as you haven't modified existing code**, and is mainly suitable to recover from job failure. Once you've modified your code (i.e uploaded a new version to the spark cluster), the checkpointed data is no longer compatible and must be deleted in order for your job to be able to start.

Setting the Right Batch Size

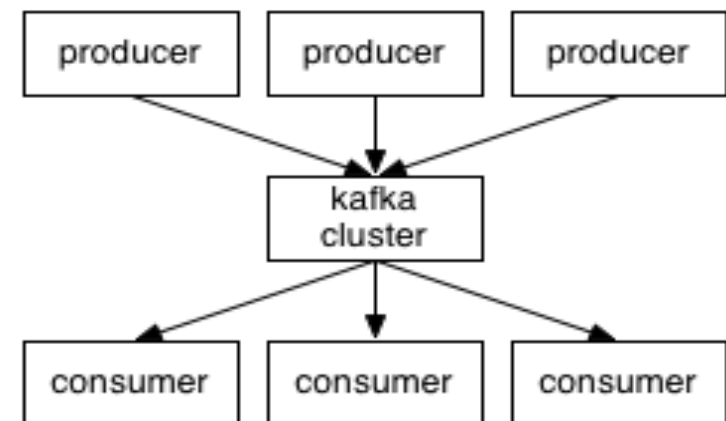
- For a Spark Streaming application running on a cluster to be stable, the processing of the data streams must keep up with the rate of ingestion of the data streams. Depending on the type of computation, the batch size used may have significant impact on the rate of ingestion that can be sustained by the Spark Streaming application on a fixed cluster resources.
- For example, let us consider the earlier WordCountNetwork example. For a particular data rate, the system may be able to keep up with reporting word counts every 2 seconds (i.e., batch size of 2 seconds), but not every 500 milliseconds.
- A good approach to figure out the right batch size for your application is to test it with a conservative batch size (say, 5-10 seconds) and a low data rate. To verify whether the system is able to keep up with data rate, you can check the value of the end-to-end delay experienced by each processed batch (either look for "Total delay" in Spark driver log4j logs, or use the `StreamingListener` interface).
- If the delay is maintained to be comparable to the batch size, then system is stable. Otherwise, if the delay is continuously increasing, it means that the system is unable to keep up and it therefore unstable. Once you have an idea of a stable configuration, you can try increasing the data rate and/or reducing the batch size.
- The momentary increase in the delay due to temporary data rate increases maybe fine as long as the delay reduces back to a low value (i.e., less than batch size).

24/7 Operations

- By default, Spark does not forget any of the metadata (RDDs generated, stages processed, etc.).
- For Spark Streaming application to operate 24/7, it is necessary for Spark to do periodic cleanup of its metadata.
- This can be enabled by setting the configuration property `spark.cleaner.ttl` to the number of seconds you want any metadata to persist.
- For example, setting `spark.cleaner.ttl` to 600 would cause Spark to periodically cleanup all metadata and persisted RDDs that are older than 10 minutes.
- This property needs to be set before the `SparkContext` is created.
- This value is closely tied with any window operation that is being used. Any window operation would require the input data to be persisted in memory for at least the duration of the window. Hence it is necessary to set the delay to at least the value of the largest window operation used in the Spark Streaming application. If this delay is set too low, the application will throw an exception saying so.

Call in Kafka

- Spark Streaming might have difficulties keeping up with incoming messages. We need some kind of buffer, to keep messages in place while Spark catches its breath. Apache Kafka provides that service.
- Kafka is a distributed, partitioned, replicated commit log service. It provides the functionality of a messaging system, but with a unique design.
- Kafka maintains feeds of messages in categories called *topics*.
- Processes that publish messages to a Kafka topic are called *producers*.
- Processes that subscribe to topics and process the feed of published messages are called *consumers*.
- Kafka is run as a cluster comprised of one or more servers each called a *broker*.
- For several Kafka machines (processes, servers) to run in coordination we need a "Cluster coordination software" called Zookeeper.

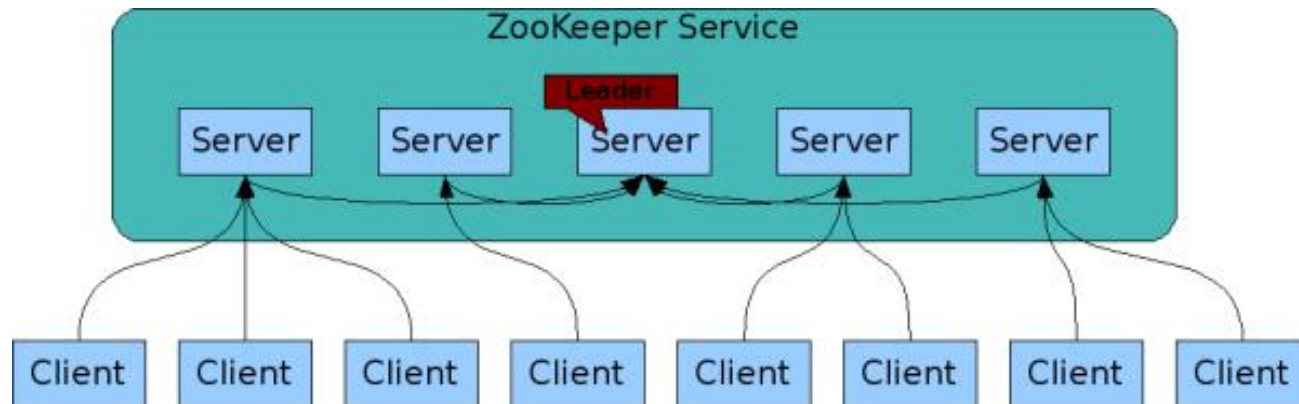


Zookeeper What is Alternative

- ZooKeeper is a high-performance coordination service for distributed applications. It exposes common services - such as naming, configuration management, synchronization, and group services - in a simple interface so you don't have to write them from scratch.
- We can use Zookeeper off-the-shelf to implement consensus, group management, leader election, and presence protocols.
- Many frameworks, like Kafka, are built to use Zookeeper.
- ZooKeeper allows distributed processes to coordinate with each other through a shared hierarchical namespace which is organized similarly to a standard file system. The name space consists of data registers - called znodes, in ZooKeeper parlance - and these are similar to files and directories. Unlike a typical file system, which is designed for storage, ZooKeeper data is kept in-memory, which means ZooKeeper can achieve high throughput and low latency numbers.
- The ZooKeeper implementation puts a premium on high performance, highly available, strictly ordered access. The performance aspects of ZooKeeper means it can be used in large, distributed systems. The reliability aspects keep it from being a single point of failure. The strict ordering means that sophisticated synchronization primitives can be implemented at the client.
- **ZooKeeper is replicated.** Like the distributed processes it coordinates, ZooKeeper itself is intended to be replicated over a sets of hosts called an ensemble.

Zookeeper

- The servers that make up the ZooKeeper service must all know about each other. They maintain an in-memory image of state, along with a transaction logs and snapshots in a persistent store. As long as a majority of the servers are available, the ZooKeeper service will be available.
- Clients connect to a single ZooKeeper server. The client maintains a TCP connection through which it sends requests, gets responses, gets watch events, and sends heart beats. If the TCP connection to the server breaks, the client will connect to a different server.
- **ZooKeeper is fast.** It is especially fast in "read-dominant" workloads. ZooKeeper applications run on thousands of machines, and it performs best where reads are more common than writes, at ratios of around 10:1.



Dependencies

- To write Spark Streaming programs using Spark Streaming you must add the following to SBT or Maven project:

```
groupId = org.apache.spark  
artifactId = spark-streaming_2.10
```

- For ingesting data from sources like Kafka and Flume that are not present in the Spark Streaming core API, you have to add the corresponding artifact `spark-streaming-xyz_2.10` to the dependencies.
- For example, some of the common ones are as follows.

Source	Artifact
Kafka	spark-streaming-kafka_2.10
Flume	spark-streaming-flume_2.10
Twitter	spark-streaming-twitter_2.10
ZeroMQ	spark-streaming-zeromq_2.10
MQTT	spark-streaming-mqtt_2.10

Install Kafka, Start the Server

- **Step 1. Download the 0.9.0.0 release and un-tar it.**

```
> tar -xzf kafka_2.11-0.9.0.0.tgz
```

```
> cd kafka_2.11-0.9.0.0
```

- **Step 2: Start the server**

- Kafka uses ZooKeeper so you need to first start a ZooKeeper server if you don't already have one. You can use the convenience script packaged with Kafka to get a quick-and-dirty single-node ZooKeeper instance.

```
> bin/zookeeper-server-start.sh config/zookeeper.properties
```

- [2016-03-22 15:01:37,495] INFO Reading configuration from:
config/zookeeper.properties
(org.apache.zookeeper.server.quorum.QuorumPeerConfig)

- **If you get a message that the port is in use, run `zookeeper-server-stop.sh` or go to directory `/etc/init.d` and run**

```
$ sudo service zookeeper-server stop
```

- **Now start the Kafka server:**

```
> bin/kafka-server-start.sh config/server.properties
```

```
[2016-0-22 15:01:47,028] INFO Verifying properties  
(kafka.utils.VerifiableProperties)
```

```
[2016-03-22 15:01:47,051] INFO Property socket.send.buffer.bytes is  
overridden to 1048576 (kafka.utils.VerifiableProperties)
```

Create Topic, Start Producer, Consumer

- **Step 3:** Create a topic

- Let's create a topic named "test" with a single partition and only one replica:

```
> bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic test
```

- We can now see that topic if we run the list topic command:

```
bin/kafka-topics.sh --list --zookeeper localhost:2181
test
```

- Alternatively, instead of manually creating topics you can also configure your brokers to auto-create topics when a non-existent topic is published to.

- **Step 4:** Send some messages

- Kafka comes with a command line client that will take input from a file or from standard input and send it out as messages to the Kafka cluster. By default each line will be sent as a separate message.

- Run the producer and then type a few messages into the console to send to the server.

```
> bin/kafka-console-producer.sh --broker-list localhost:9092 --topic test
```

This is a message

This is another message

- **Step 5:** Start a consumer

- Kafka also has a command line consumer that will dump out messages to standard output.

```
➤ bin/kafka-console-consumer.sh --zookeeper localhost:2181 --topic test --from-beginning
```

This is a message

This is another message

- If you have each of the above commands running in a different terminal then you should now be able to type messages into the producer terminal and see them appear in the consumer terminal.

Spark Streaming & Kafka Integration

- We want to configure Spark Streaming to receive data from Kafka.
- There are two approaches to this - the old approach using Receivers and Kafka's high-level API, and a new approach (introduced in Spark 1.3) without using Receivers.
- They have different programming models, performance characteristics, and semantics guarantees.

Receiver Approach, Dependencies

- The Receiver is implemented using the Kafka high-level consumer API.
- The data received from Kafka through a Receiver is stored in Spark executors, and then jobs launched by Spark Streaming processes the data.
- Under the default configuration, this approach can lose data under failures. To ensure zero-data loss, you have to additionally enable Write Ahead Logs in Spark Streaming. These logs synchronously save all the received Kafka data into write ahead logs on a distributed file system (e.g HDFS), so that all the data can be recovered on failure.

- For Scala/Java applications using SBT/Maven project definitions, link your streaming application with the following artifact.

```
groupId = org.apache.spark  
artifactId = spark-streaming-kafka_2.10  
version = 1.6.1
```

- For Python applications, you will also have to add the above library and its dependencies when deploying the application.

Receiver Approach, Programming

- In the streaming application code, import `KafkaUtils` and create an input `DStream`:

```
import org.apache.spark.streaming.kafka.*;
JavaPairReceiverInputDStream<String, String> kafkaStream =
    KafkaUtils.createStream(streamingContext,
        [ZK quorum], [consumer group id], [per-topic number of Kafka
partitions to consume]);
```

- You can also specify the key and value classes and their corresponding decoder classes using variations of `createStream`.
- Topic partitions in Kafka do not correlate to partitions of RDDs generated in Spark Streaming. 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. It does not increase the parallelism of Spark in processing the data.
- Multiple Kafka input `DStreams` can be created with different groups and topics for parallel receiving of data using multiple receivers.
- If you have enabled Write Ahead Logs with a replicated file system like HDFS, the received data is already being replicated in the log. Hence, the storage level in storage level for the input stream to `StorageLevel.MEMORY_AND_DISK_SER` (that is, use `KafkaUtils.createStream(..., StorageLevel.MEMORY_AND_DISK_SER)`).

Deploying

- As with any Spark applications, `spark-submit` is used to launch your application. However, the details are slightly different for Scala/Java applications and Python applications.
- For Scala and Java applications, if you are using SBT or Maven for project management, then package `spark-streaming-kafka_2.10` and its dependencies into the application JAR.
- For Python applications which lack SBT/Maven project management, `spark-streaming-kafka_2.10` and its dependencies can be directly added to `spark-submit` using `--packages`.

```
./bin/spark-submit --packages org.apache.spark:spark-streaming-kafka_2.10:1.6.1 ...
```

- Alternatively, you can also download the JAR of the Maven `artifact` `spark-streaming-kafka-assembly` from the Maven repository and add it to `spark-submit` with `--jars`

Direct Approach

- This new receiver-less "direct" 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 (similar to read files from a file system).

This approach has the following advantages over the receiver-based approach:

- **Simplified Parallelism:** 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, which will all read data from Kafka in parallel. So there is a one-to-one mapping between Kafka and RDD partitions, which is easier to understand and tune.
- **Efficiency:** Achieving zero-data loss in the first approach required the data to be stored in a Write Ahead Log, which further replicated the data. This is actually inefficient as the data effectively gets replicated twice - once by Kafka, and a second time by the Write Ahead Log. This second approach eliminates the problem as there is no receiver, and hence no need for Write Ahead Logs. As long as you have sufficient Kafka retention, messages can be recovered from Kafka.
- **Exactly-once semantics:** The first approach uses Kafka's high level API to store consumed offsets in Zookeeper. This is traditionally the way to consume data from Kafka. While this approach (in combination with write ahead logs) can ensure zero data loss (i.e. at-least once semantics), there is a small chance some records may get consumed twice under some failures. This occurs because of inconsistencies between data reliably received by Spark Streaming and offsets tracked by Zookeeper. Hence, in this second approach, we use simple Kafka API that does not use Zookeeper. Offsets are tracked by Spark Streaming within its checkpoints. This eliminates inconsistencies between Spark Streaming and Zookeeper/Kafka, and so each record is received by Spark Streaming effectively exactly once despite failures.
- **One disadvantage** of this approach is that it does not update offsets in Zookeeper, hence Zookeeper-based Kafka monitoring tools will not show progress.

Programming

- This approach is supported only in Scala/Java application. Link your SBT/Maven project with the following artifact

```
groupId = org.apache.spark
artifactId = spark-streaming-kafka_2.10
version = 1.6.1
```

- In the streaming application code, import `KafkaUtils` and create an input `DStream` as:

```
import org.apache.spark.streaming.kafka.*;
```

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

- You can also pass a `messageHandler` to `createDirectStream` to access `MessageAndMetadata` that contains metadata about the current message and transform it to any desired type. See the API docs and the example.
- In the Kafka parameters, you must specify either `metadata.broker.list` or `bootstrap.servers`. By default, it will start consuming from the latest offset of each Kafka partition. If you set configuration `auto.offset.reset` in Kafka parameters to `smallest`, then it will start consuming from the smallest offset.
- You can also start consuming from any arbitrary offset using other variations of `KafkaUtils.createDirectStream`. Furthermore, if you want to access the Kafka offsets consumed in each batch, you can do the following

Code Snippet

```
// Hold a reference to the current offset ranges, so it can be used downstream
final AtomicReference<OffsetRange[]> offsetRanges = new AtomicReference<>();

directKafkaStream.transformToPair(
    new Function<JavaPairRDD<String, String>, JavaPairRDD<String, String>>() {
        @Override
        public JavaPairRDD<String, String> call(JavaPairRDD<String, String> rdd) throws
Exception {
            OffsetRange[] offsets = ((HasOffsetRanges) rdd.rdd()).offsetRanges();
            offsetRanges.set(offsets);
            return rdd;
        }
    }
).map(
    ...
).foreachRDD(
    new Function<JavaPairRDD<String, String>, Void>() {
        @Override
        public Void call(JavaPairRDD<String, String> rdd) throws IOException {
            for (OffsetRange o : offsetRanges.get()) {
                System.out.println(
                    o.topic() + " " + o.partition() + " " + o.fromOffset() + " " +
o.untilOffset()
                );
            }
            ...
            return null;
        }
    }
);
```

Near Real-Time Stack

- For data scientists and developers working with real-time data pipelines, Spark Streaming and Kafka are not all that is needed. Often a large volume of data needs to be stored or retrieved, either before or after processing with Spark Stream. **Spark Streaming-Kafka-Cassandra** has recently emerged as the stack of choice for such applications,.
- This stack satisfies the key requirements for real-time data analytics:
 - **Spark Streaming** is an extension of the core Spark API; it allows integration of near real-time data from disparate event streams.
 - **Kafka**: a messaging system to capture and publish streams of data. With Spark you can ingest data from Kafka, filter that stream down to a smaller data set, augment the data, and then push that refined data set to a persistent data store.
 - **Cassandra**: appears to be an excellent choice when data needs to be written to a scalable and resilient operational database for persistence, easy application development, and real-time analytics.
- These open source frameworks are powerful and well-suited for the requirements of building real-time data pipelines.

References

- <https://databricks.com/blog/2016/07/28/structured-streaming-in-apache-spark.html>
- <http://spark.apache.org/docs/latest/streaming-programming-guide.html>
- Chapter 6, Spark in Action by Petar Zecevic & Marko Bonaci, Manning Publishing. 2nd Edition is apparently available.