Learning Goals

- Identify Spark Unified Stack Components
- List Benefits of Apache Spark over Hadoop Ecosystem
- Describe Spark Data Pipeline Use Cases

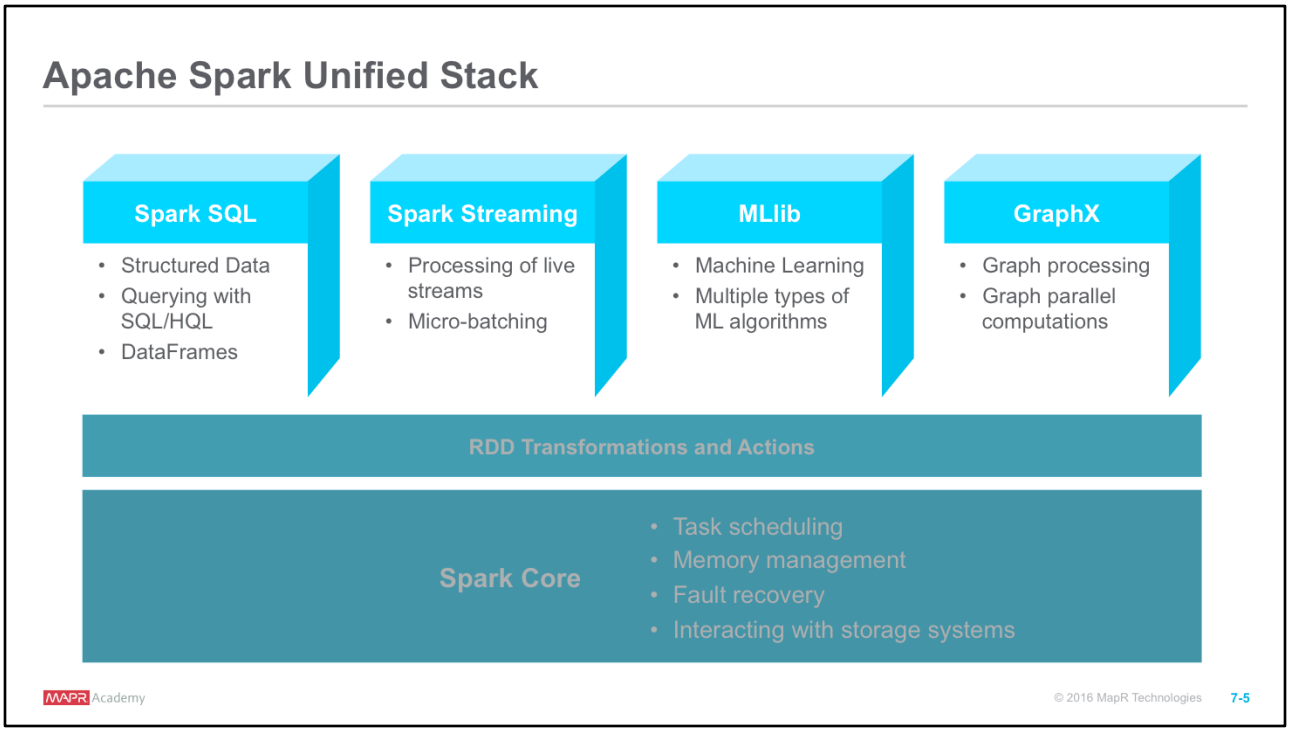
At the end of this lesson, you will be able to:

- Identify the components of the Spark unified stack
- List the benefits of Apache Spark over the Hadoop ecosystem
- Describe Apache Spark data pipeline use cases

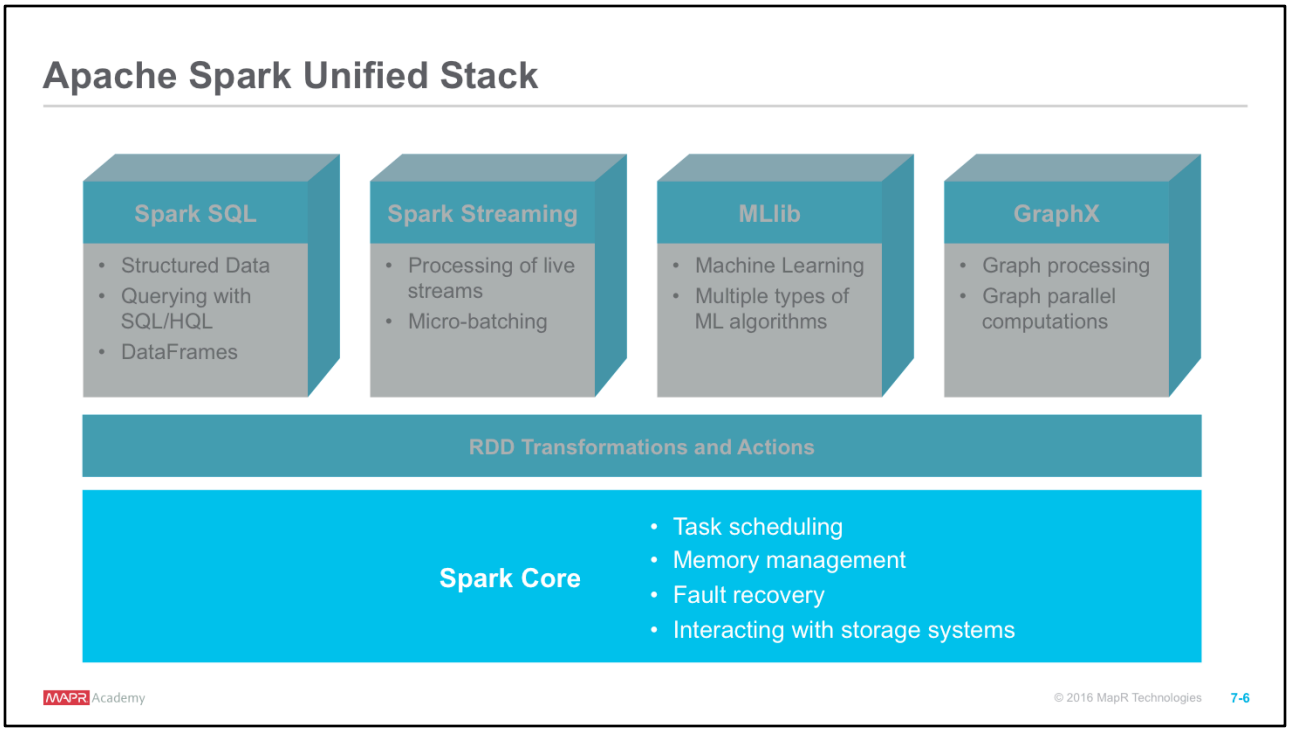
Learning Goals

- **Identify Spark Unified Stack Components**
- List Benefits of Apache Spark over Hadoop Ecosystem
- Describe Spark Data Pipeline Use Cases

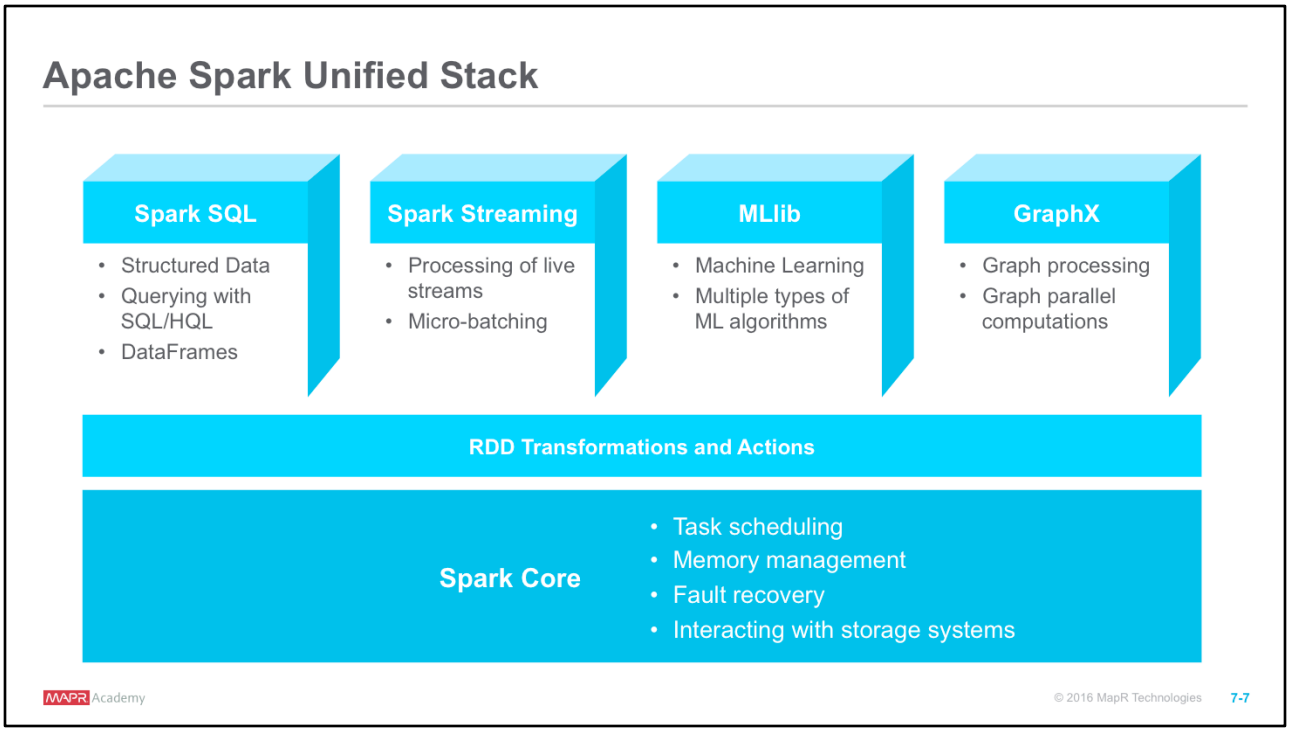
In this first section, we will discuss the components of the Apache Spark unified stack.



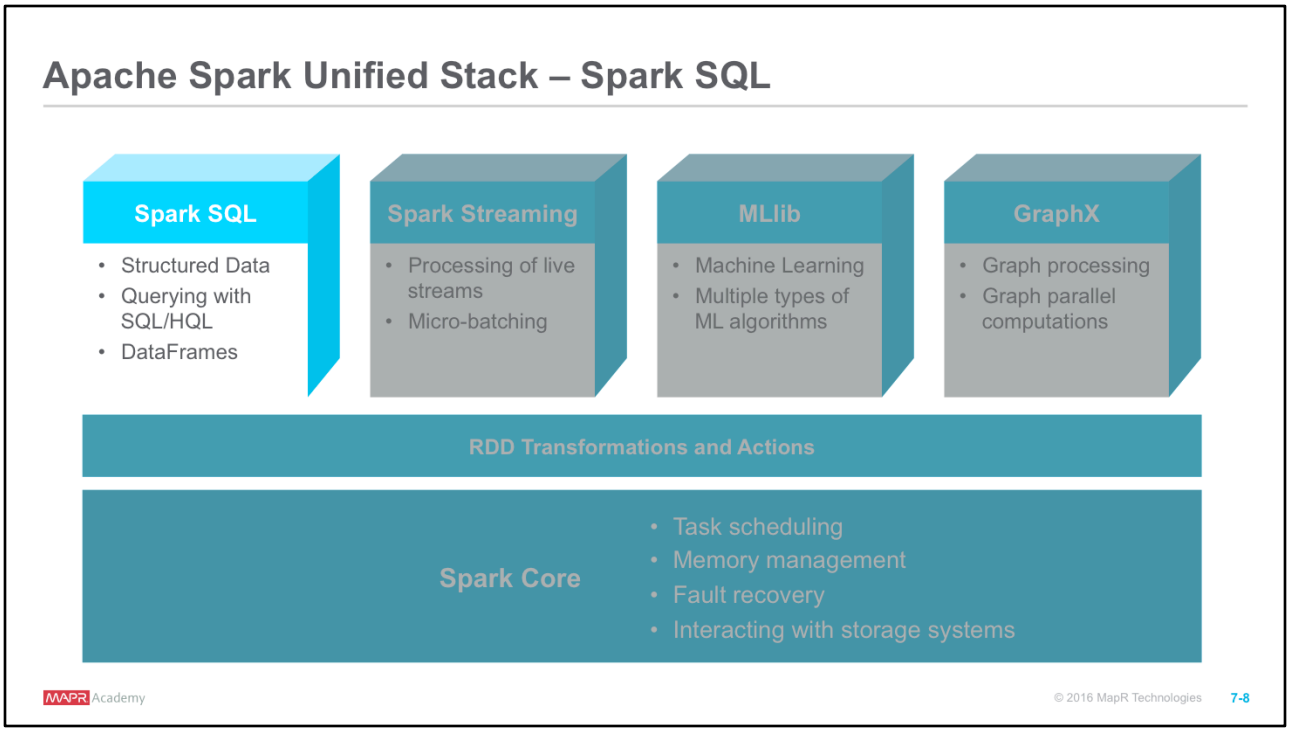
Apache Spark has an integrated framework for advanced analytics like graph processing, advanced queries, stream processing and machine learning. You can combine these libraries into the same application and use a single programming language through the entire workflow.



The Spark core is a computational engine that is responsible for task scheduling, memory management, fault recovery and interacting with storage systems. The Spark core contains the functionality of Spark. It also contains the APIs that are used to define RDDs and manipulate them.

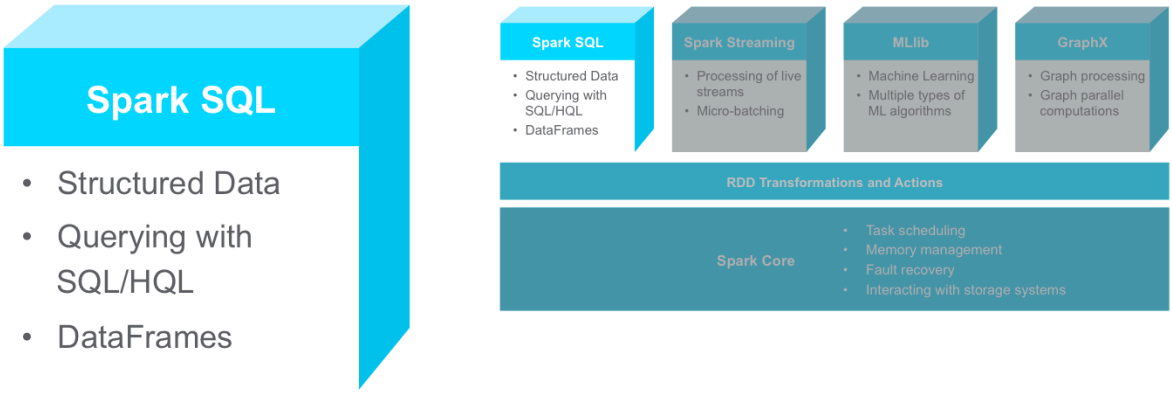


The other Spark modules shown here are tightly integrated with the Spark core.



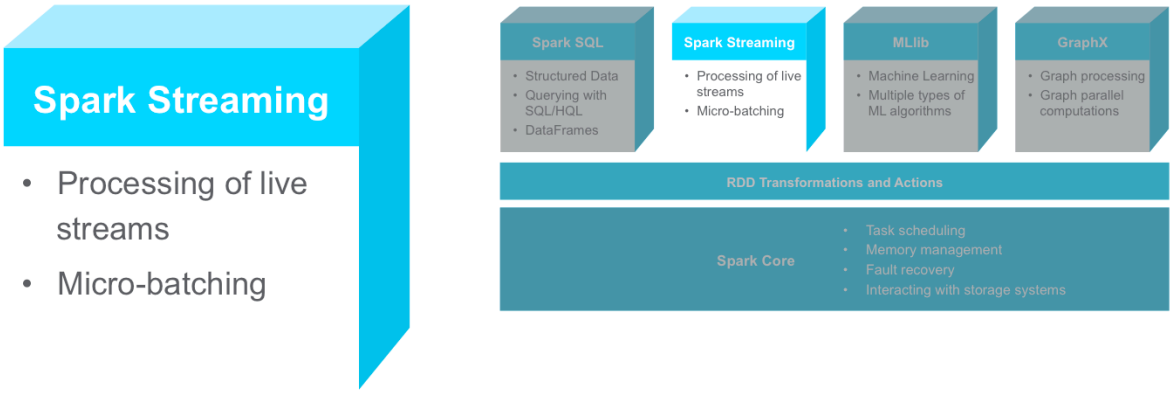
Spark SQL is used to process structured data.

Apache Spark Unified Stack – Spark SQL



Spark provides native support for SQL queries. You can query data via SQL or HiveQL. Spark SQL supports many types of data sources, such as structured Hive tables and complex JSON data. The primary abstraction of Spark SQL is Spark DataFrames.

Apache Spark Unified Stack – Spark Streaming

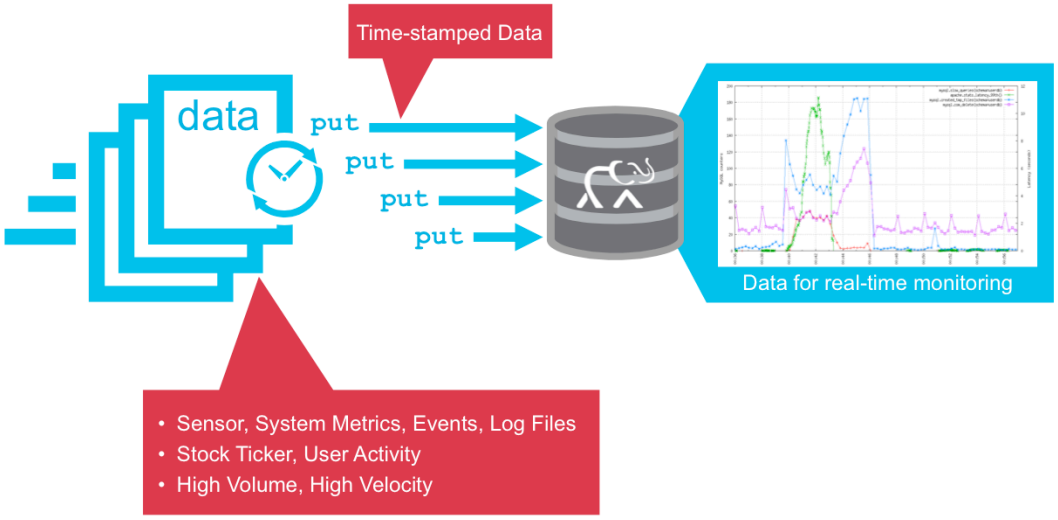


 7-10

Spark streaming enables processing of live streams of data.

When performing analytics on the live data streams, Spark streaming uses a micro-batch execution model.

Apache Spark Unified Stack – Spark Streaming



Many applications need to process streaming data with the following requirements:

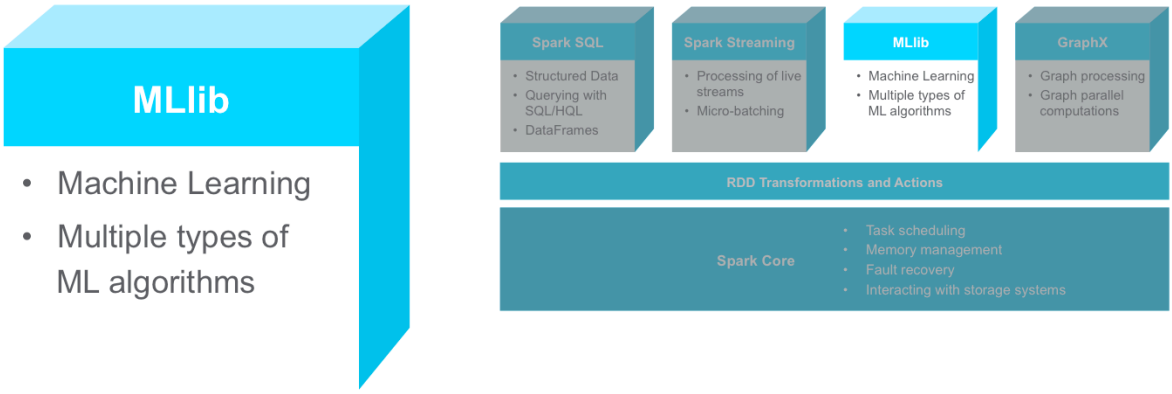
- The results should be in near-real-time
- Be able to handle large workloads and
- Have latencies of few seconds

Batch processing on the other hand processes large volumes of data collected over a period of time, where the latency is in terms of minutes.

Common examples of streaming data include activity stream data from a web or mobile application, time stamped log data, transactional data and event streams from sensor device networks.

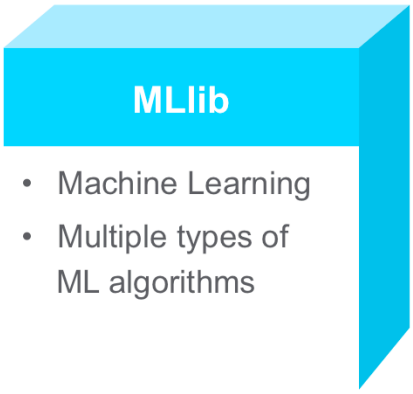
Real-time applications of stream processing include website monitoring, network monitoring, fraud detection, web clicks and advertising.

Apache Spark Unified Stack – Spark MLlib



MLlib is a Spark machine learning library that provides multiple types of machine learning algorithms such as classification, regression, clustering, collaborative filtering and optimization primitives.

Apache Spark Unified Stack – Spark MLlib

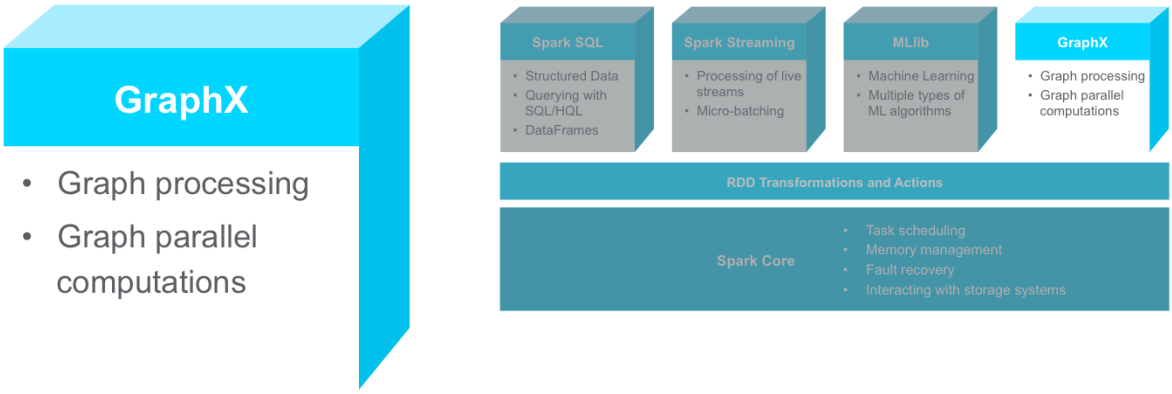


- Spark Machine Learning Library
- Common learning algorithms
  - Classification and Regressions
  - Clustering
  - Collaborative filtering
  - Dimensionality Reduction
- Two packages
  - spark.mllib
  - spark.ml

MLlib is the Spark machine learning library. It supports common learning algorithms such as classification, regressions, clustering, collaborative filtering and dimensionality reduction.

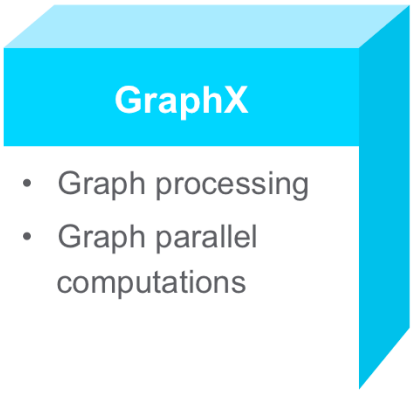
There are two packages available:
- spark.mllib – original API that can be used on RDDs
- spark.ml – higher level API that can be used on DataFrames

Apache Spark Unified Stack – Spark GraphX



GraphX is a library for manipulating graphs and performing graph-parallel computations.

Apache Spark Unified Stack – Spark GraphX



- Graph abstraction extends Spark RDD
- Can work seamlessly with both graphs and collections
- Operations for graph computation
  - Includes optimized version of Pregel
- Provides graph algorithms & builders

The primary abstraction is a graph that extends the Spark RDD. A Graph is a directed multigraph wherein each vertex and edge has properties attached. GraphX has a number of operators that can be used for graph computations including an optimized version of Pregel. GraphX also provides graph algorithms and builders for graph analytics.

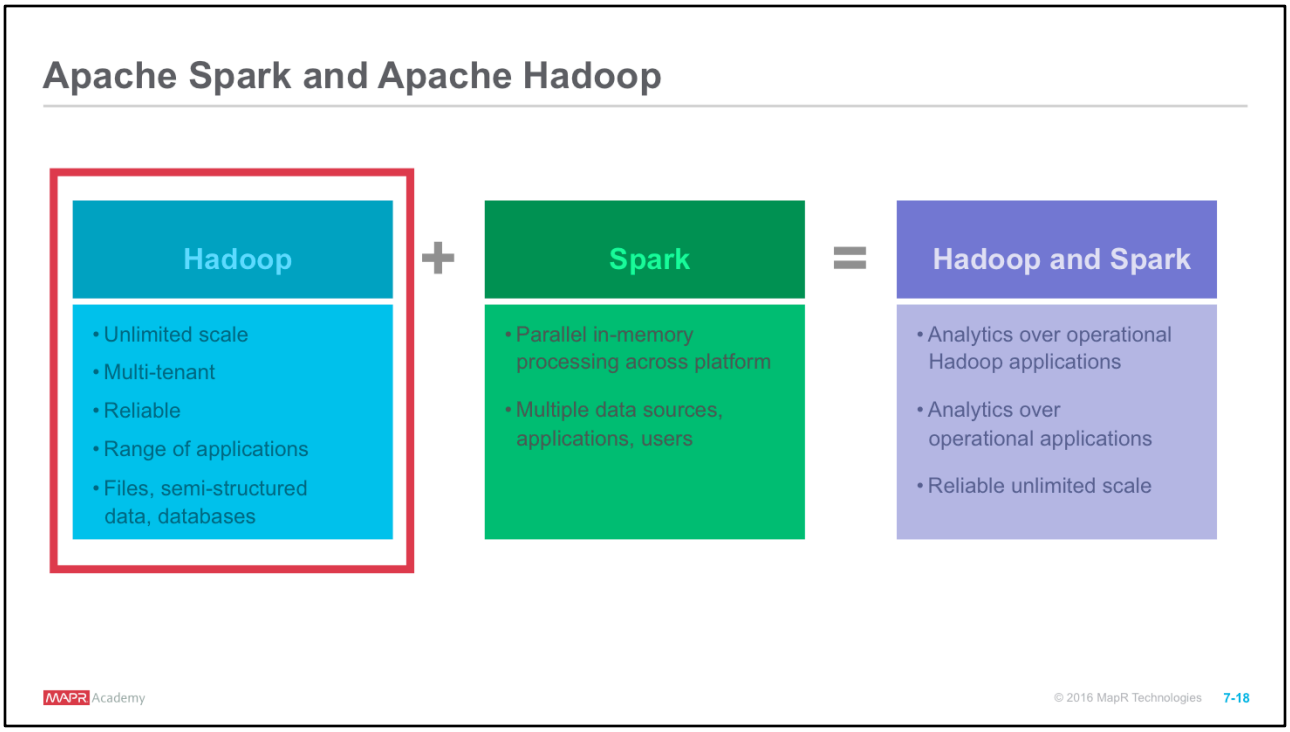
We can work seamlessly with both graphs and collections; view same data as both graphs and collections without duplication or movement of data.
We can also write custom iterative graph algorithms

Learning Goals

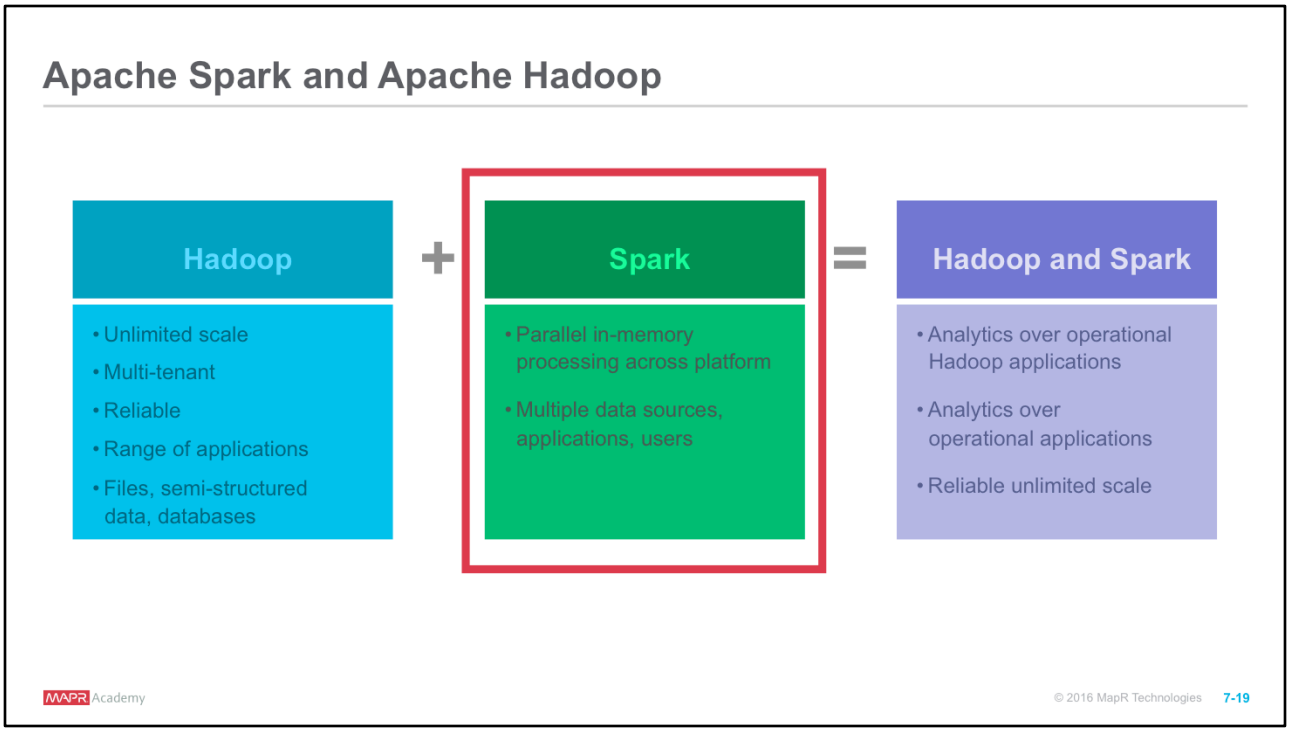
Learning Goals

- Identify Spark Unified Stack Components
- **List Benefits of Apache Spark over Hadoop Ecosystem**
- Describe Spark Data Pipeline Use Cases

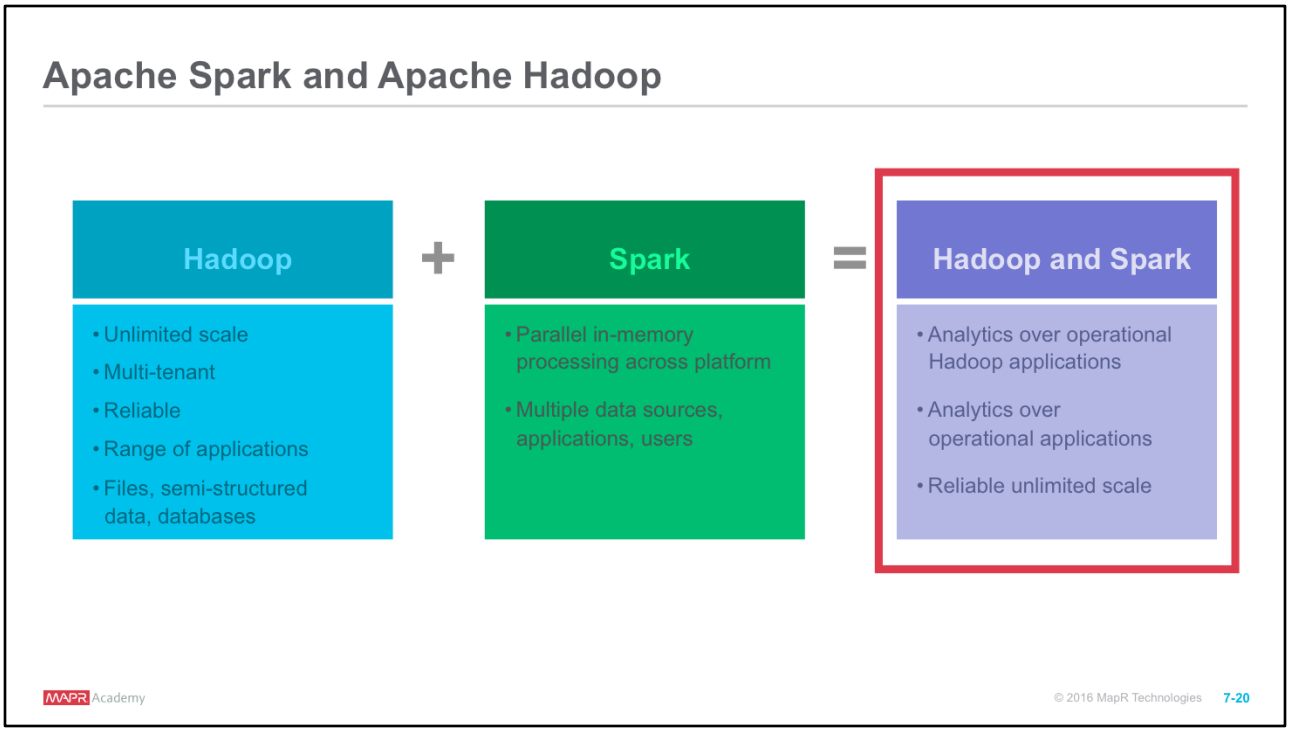
Now let us take a look a how at Spark fits in with Hadoop, and the benefits of the Spark unified stack over the Hadoop Ecosystem.



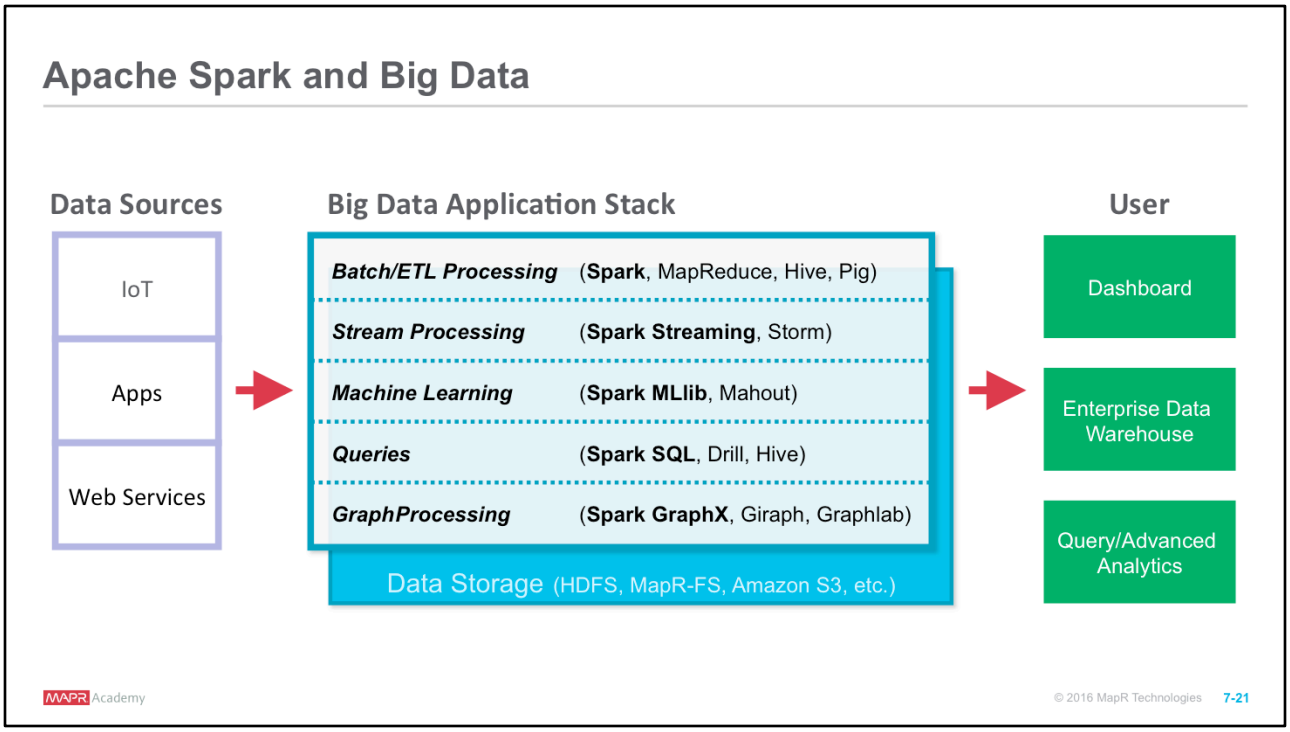
Hadoop has grown into a multi-tenant, reliable, enterprise-grade platform with a wide range of applications that can handle files, databases, and semi-structured data.



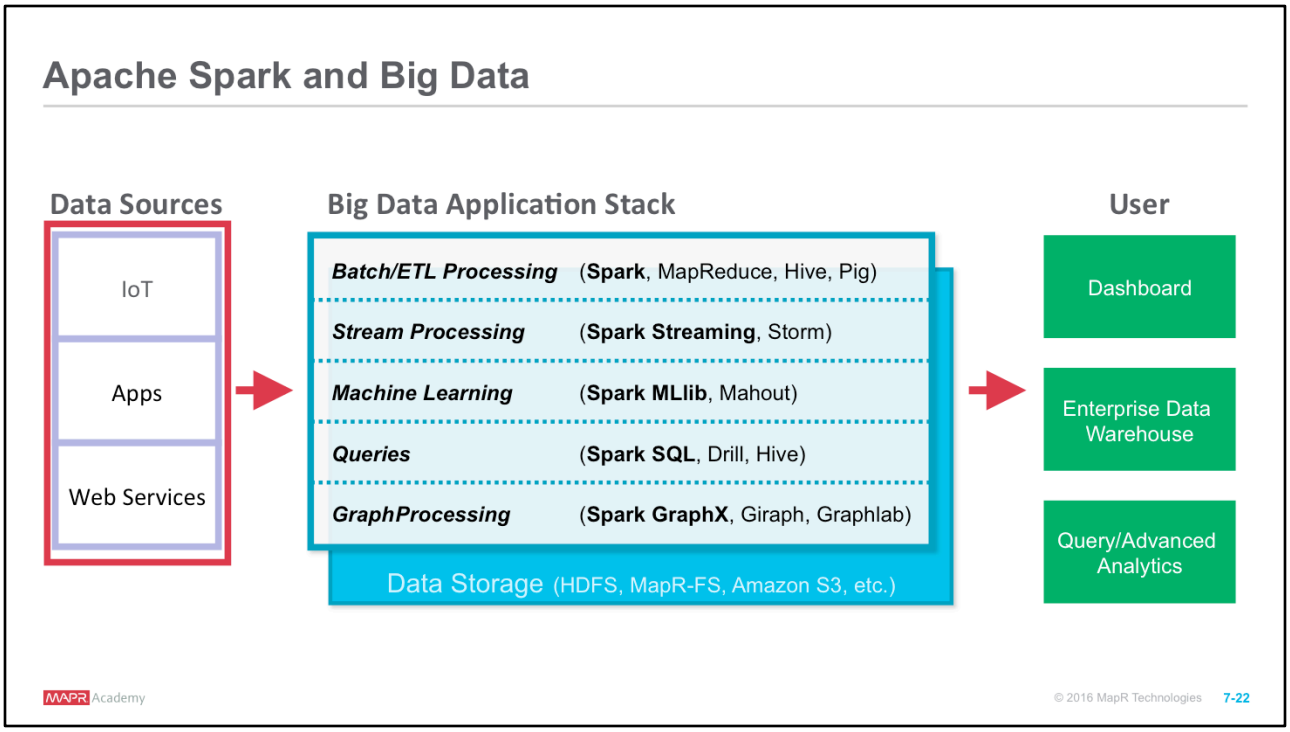
Spark provides parallel, in-memory processing across this platform, and can accommodate multiple data sources, applications and users.



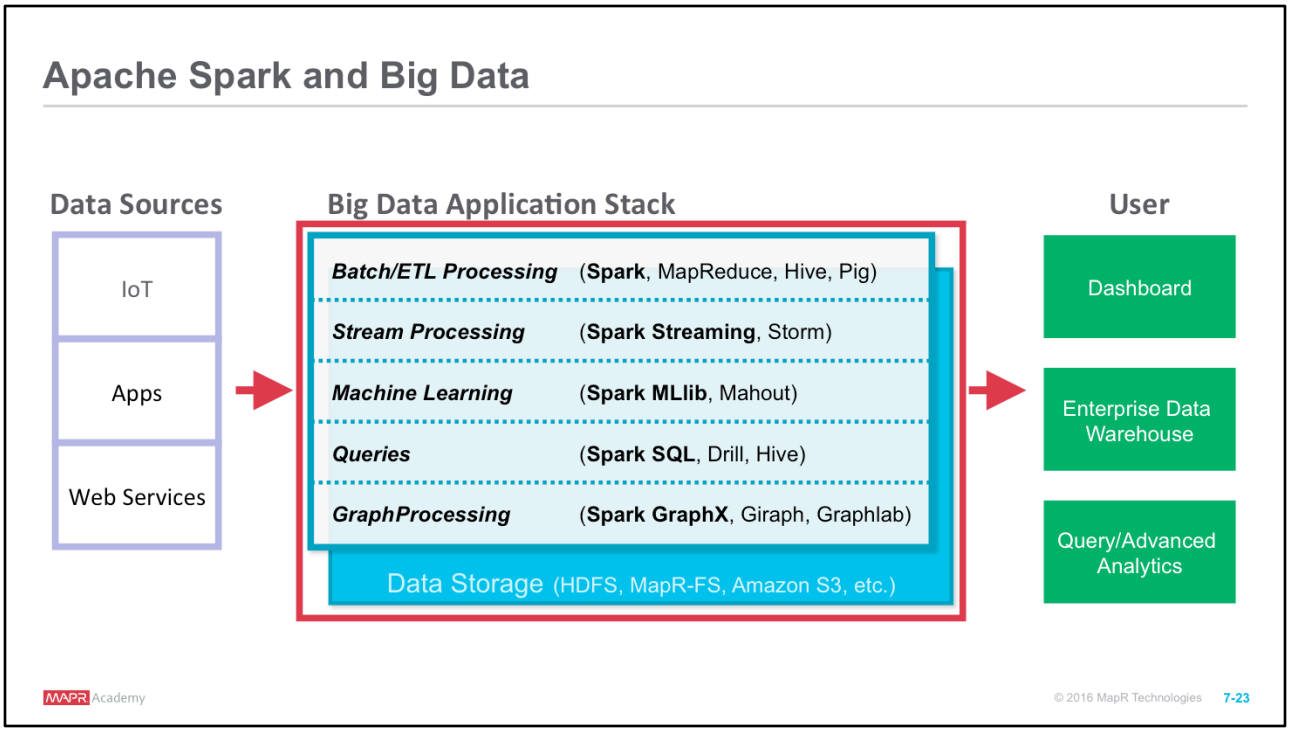
The combination of Spark and Hadoop takes advantage of both platforms, providing reliable, scalable, and fast parallel, in-memory processing. Additionally, you can easily combine different kinds of workflows to provide analytics over Hadoop and other operational applications.



This graphic depicts where Spark fits in the big data application stack.

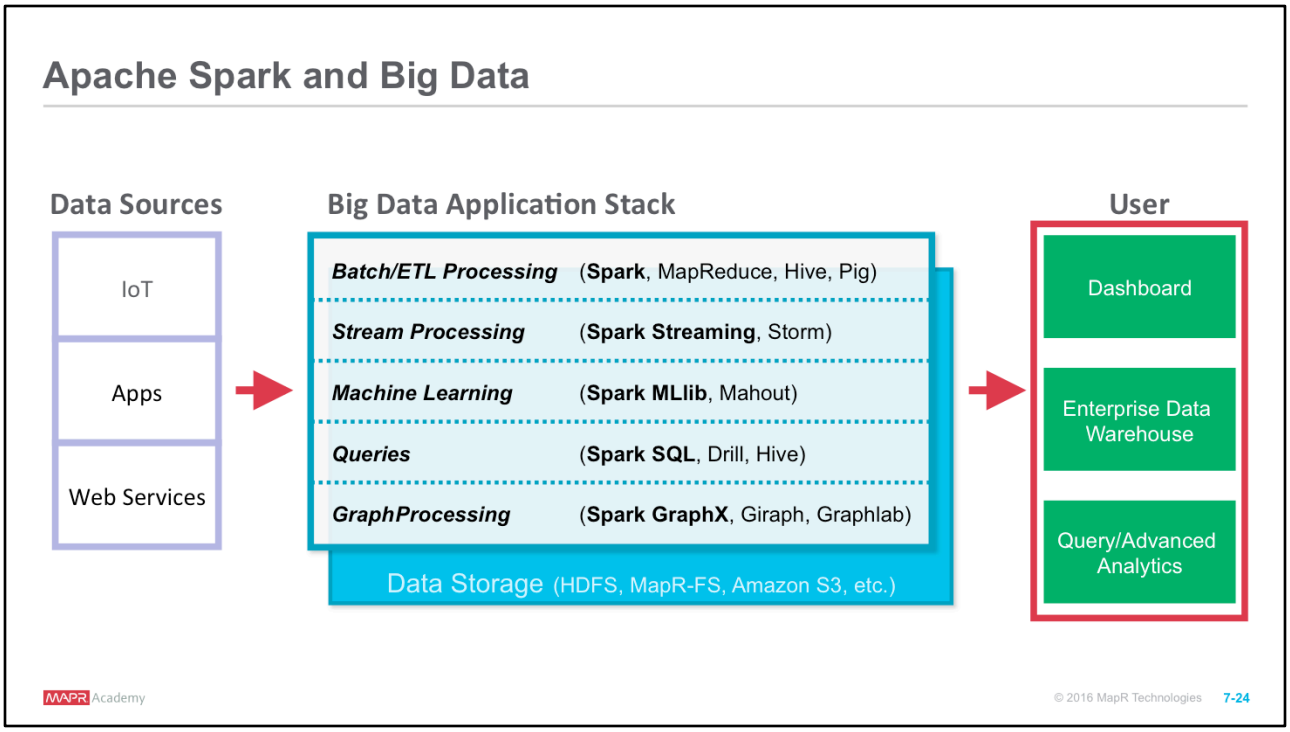


On the left we see different data sources. There are multiple ways of ingesting data, using different industry standards such as NFS or existing Hadoop tools.



The stack in the middle represents various big data processing workflows and tools that are commonly used. You may have just one of these workflows in your application, or a combination of many. Any of these workflows could read/write to or from the storage layer.

While you can combine various workflows in Hadoop, it requires using different languages and different tools. As you can see here, with Spark, you can use Spark for any of the workflows. You can build applications that ingest streaming data and apply machine learning and graph analysis to the live streams. All this can be done in the same application using one language.



The output can then be used to create real-time dashboards and alerting systems for querying and advanced analytics.

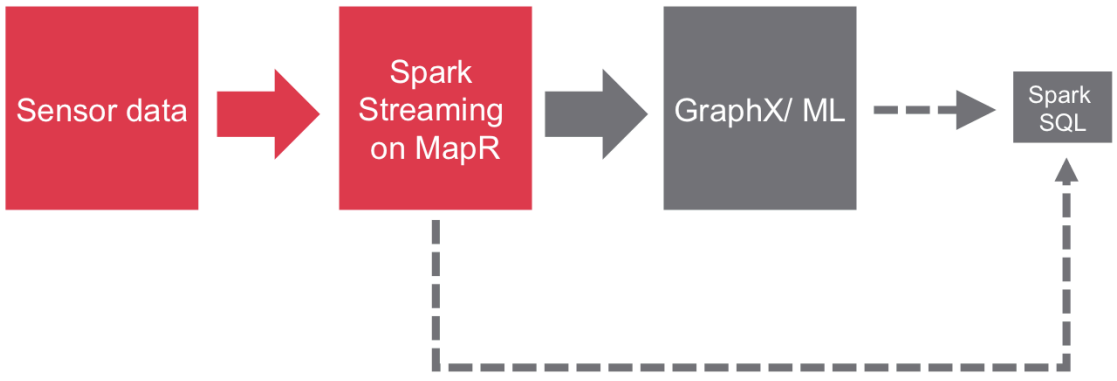
Learning Goals

Learning Goals

- Identify Spark Unified Stack Components
- List Benefits of Apache Spark over Hadoop Ecosystem
- **Describe Spark Data Pipeline Use Cases**

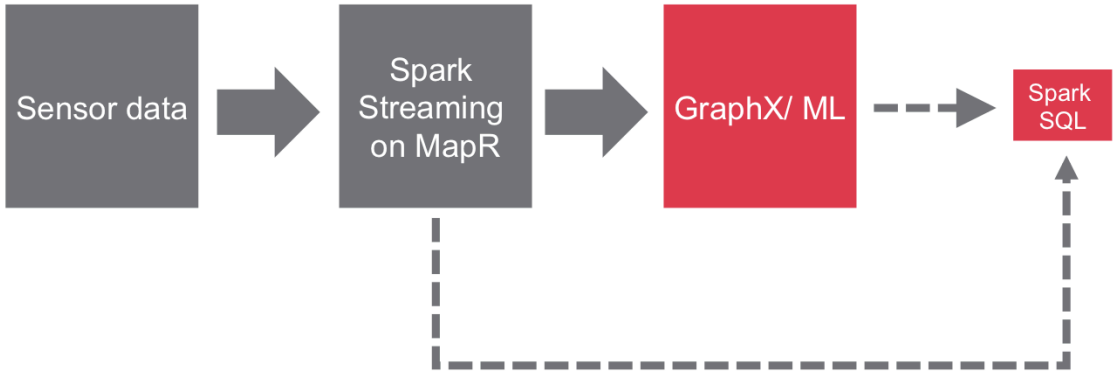
This section provides some examples of data pipeline applications using Apache Spark.

Use Case: Managed Security Services Provider



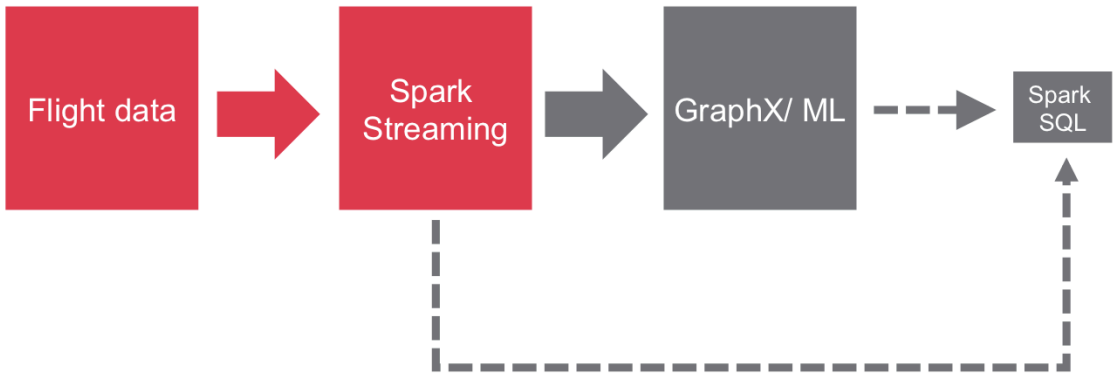
In our first use case, sensor data is streamed in using Spark Streaming on MapR and is checked for first known threats.

Use Case: Managed Security Services Provider

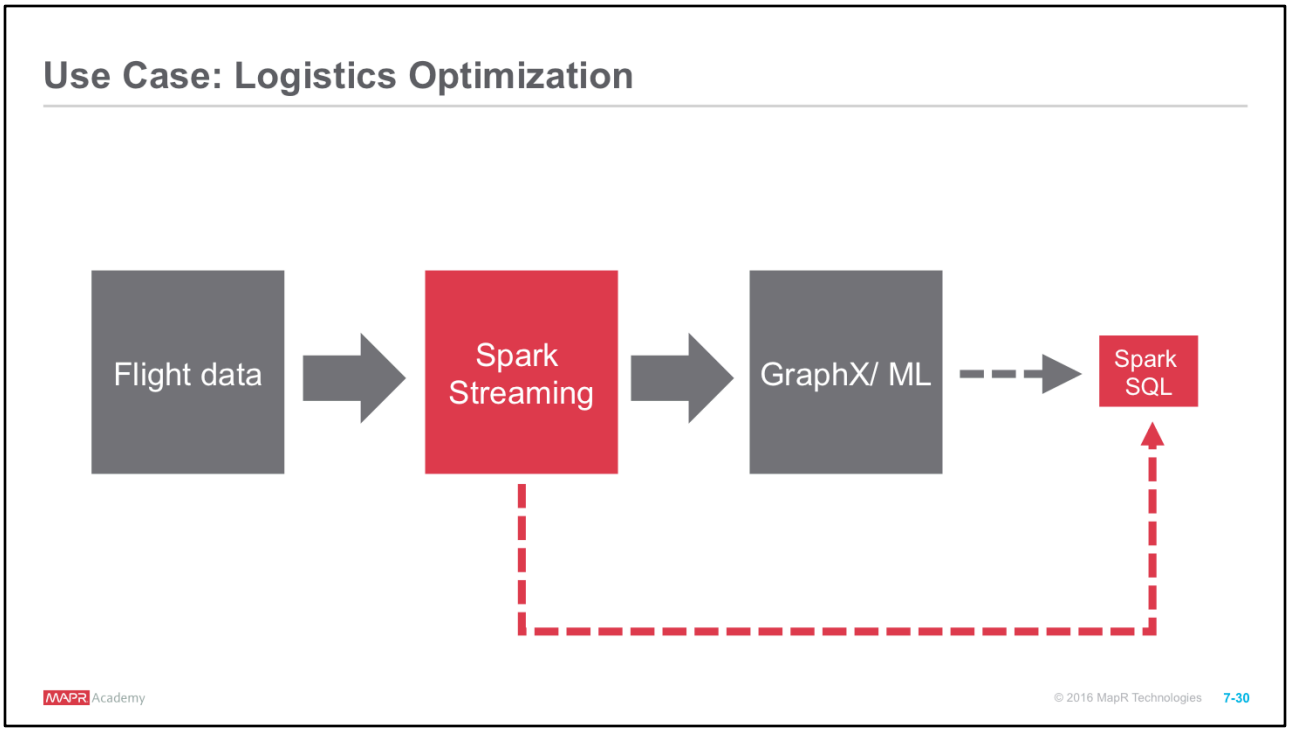


The data then goes through graph processing and machine learning for predictions. Additional querying such as results of graph algorithms, predictive models and summary/aggregate data is done using Spark SQL.

Use Case: Logistics Optimization

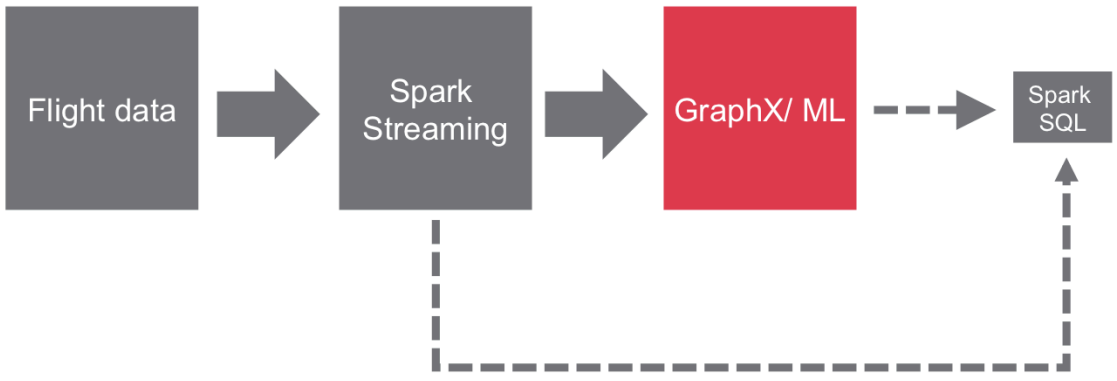


In a second use case, we begin with flight data being streamed in.



We use Spark SQL to do some analytics on the streaming data.

Use Case: Logistics Optimization



We use GraphX to analyze airports and routes that serve them. Machine learning is used to predict delays using a classification or decision tree algorithms.

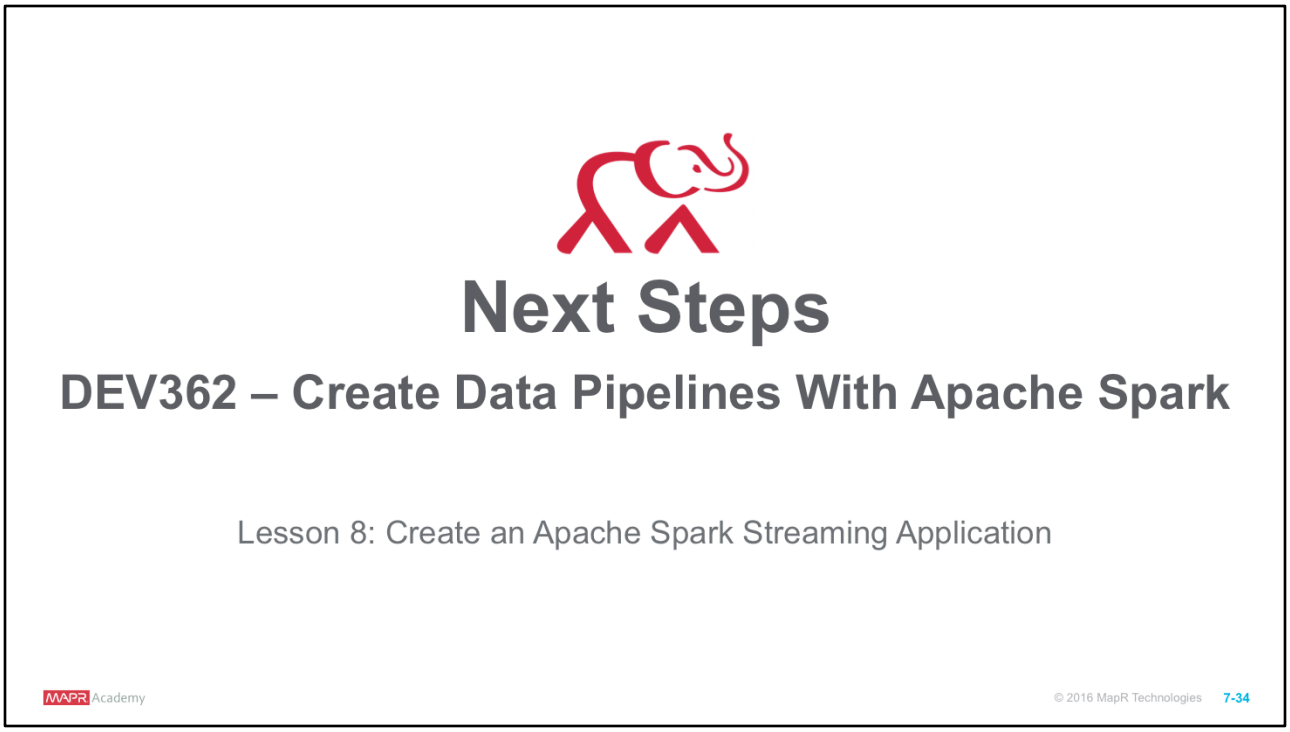
Knowledge Check

Knowledge Check

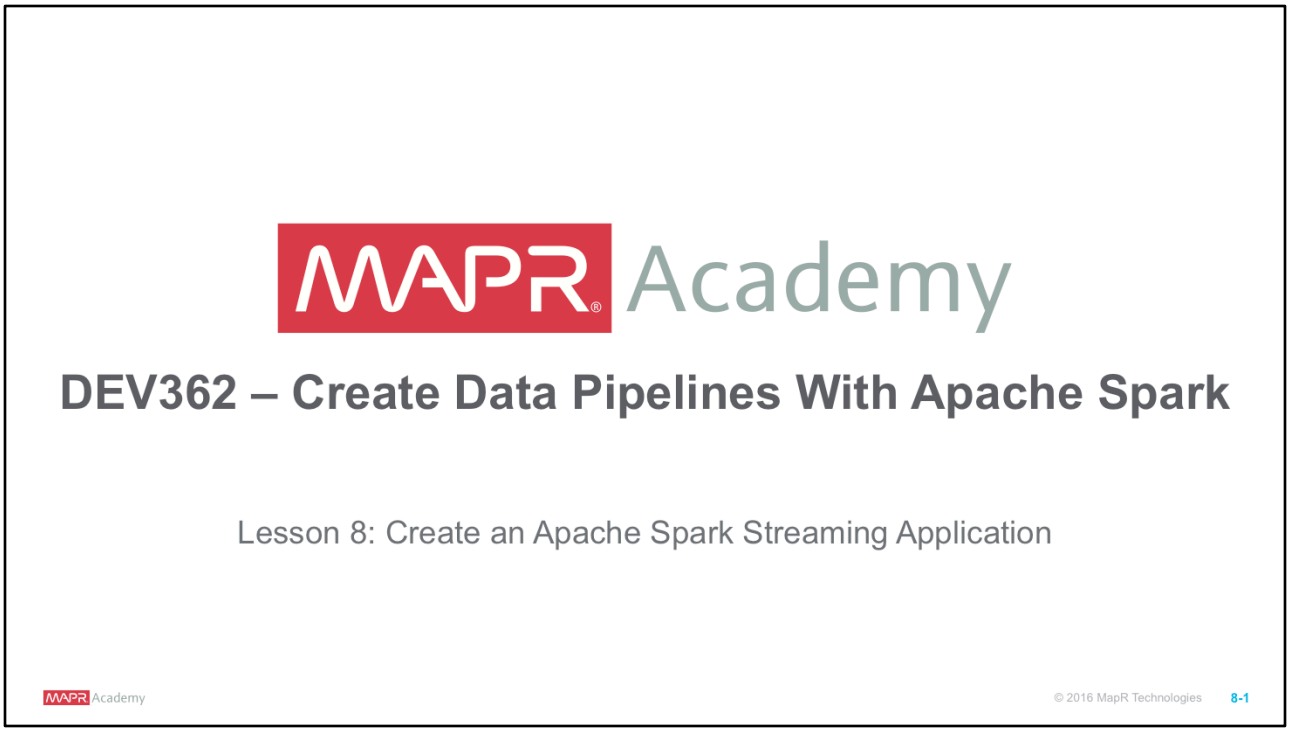
We have real-time twitter feed. We need to build an application that is near real-time and classifies the twitter feeds based on relevant and not relevant, where "relevant" means that it contains the words "FIFA", "Women's", and "World Cup". Which of the following Apache Spark libraries could we use in the application?

A. Spark SQL
B. Spark Streaming
C. Spark MLlib
D. Spark GraphX

Use Spark SQL to query the data in DataFrames, Spark Streaming to ingest the live feeds and Spark MLlib to do the classification.



Congratulations, you have completed DEV 362 Lesson 7. Continue on to lesson 8 to learn about how to create an Apache Spark Streaming application.



Welcome to DEV 362 lesson 8, create an Apache Spark Streaming application.

Learning Goals

Learning Goals

- Describe Spark Streaming Architecture
- Create DStreams and a Spark Streaming Application
- Apply Operations on DStreams
- Define Windowed Operations
- Describe How Streaming Applications are Fault Tolerant

At the end of this lesson, you will be able to:

- Describe the Apache Spark Streaming architecture
- Create DStreams and a Spark Streaming application
- Apply operations on DStreams
- Define windowed operations and
- Describe how streaming applications are fault tolerant

Learning Goals

- **Describe Spark Streaming Architecture**
- Create DStreams and a Spark Streaming Application
- Apply Operations on DStreams
- Define Windowed Operations
- Describe How Streaming Applications are Fault Tolerant

In the first section we will describe the Spark streaming architecture.

Stream Processing Architecture

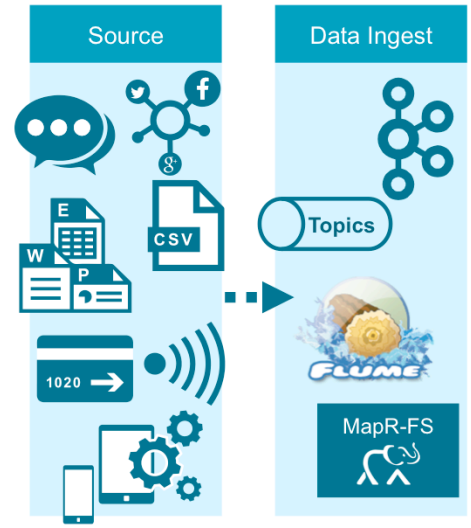
MAPR Academy

 8-5

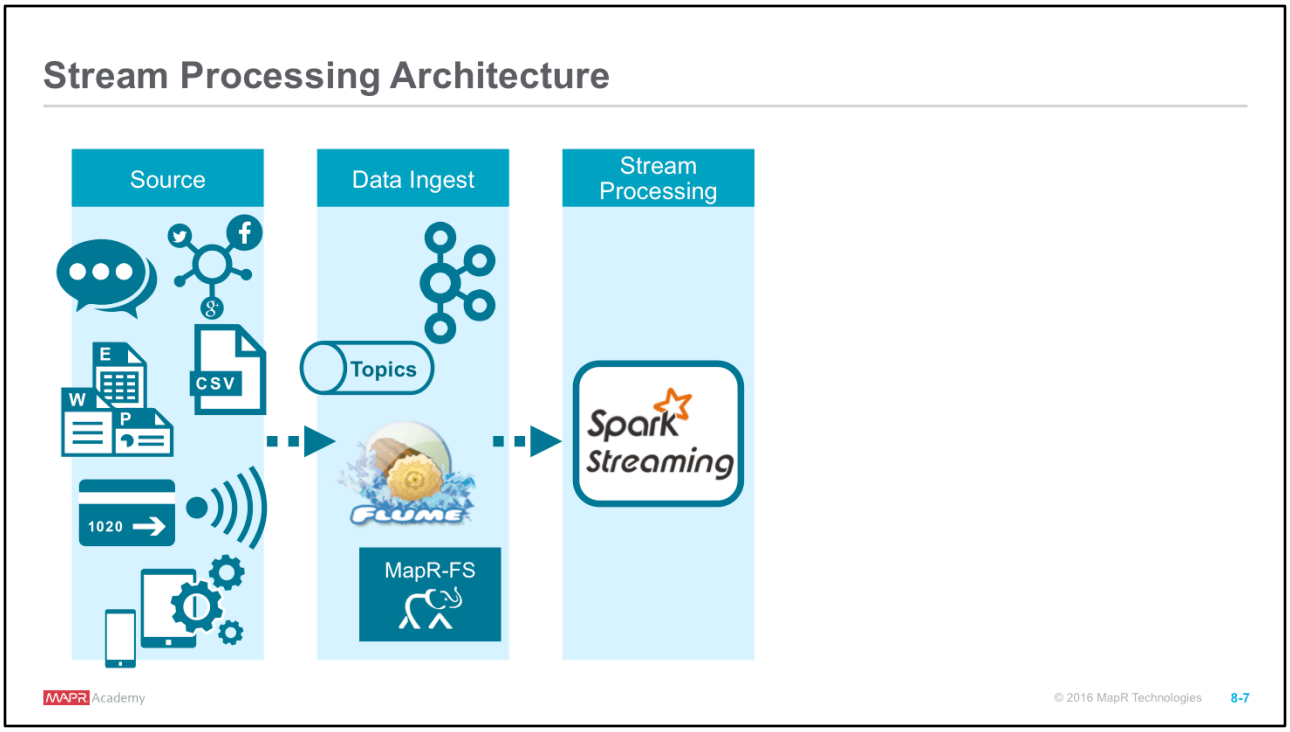
A stream processing architecture is typically made of the following components. First, the data we want to process must come from somewhere.

Sources refers to the source of data streams. Examples include sensor networks, mobile applications, web clients, logs from a server, or even a “Thing” from the Internet of Things.

Stream Processing Architecture

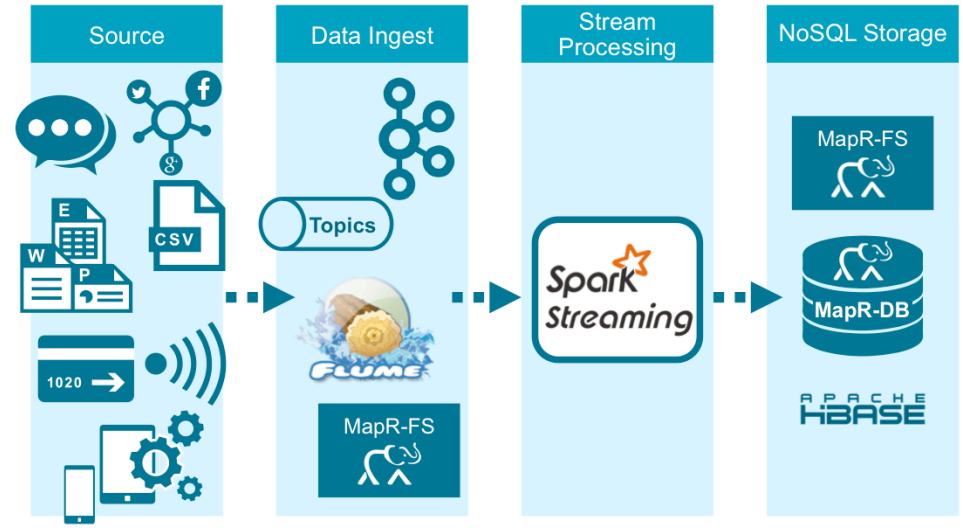


This data is delivered through messaging systems such as MapR Streams, Kafka, Flume, or deposited in a file system.

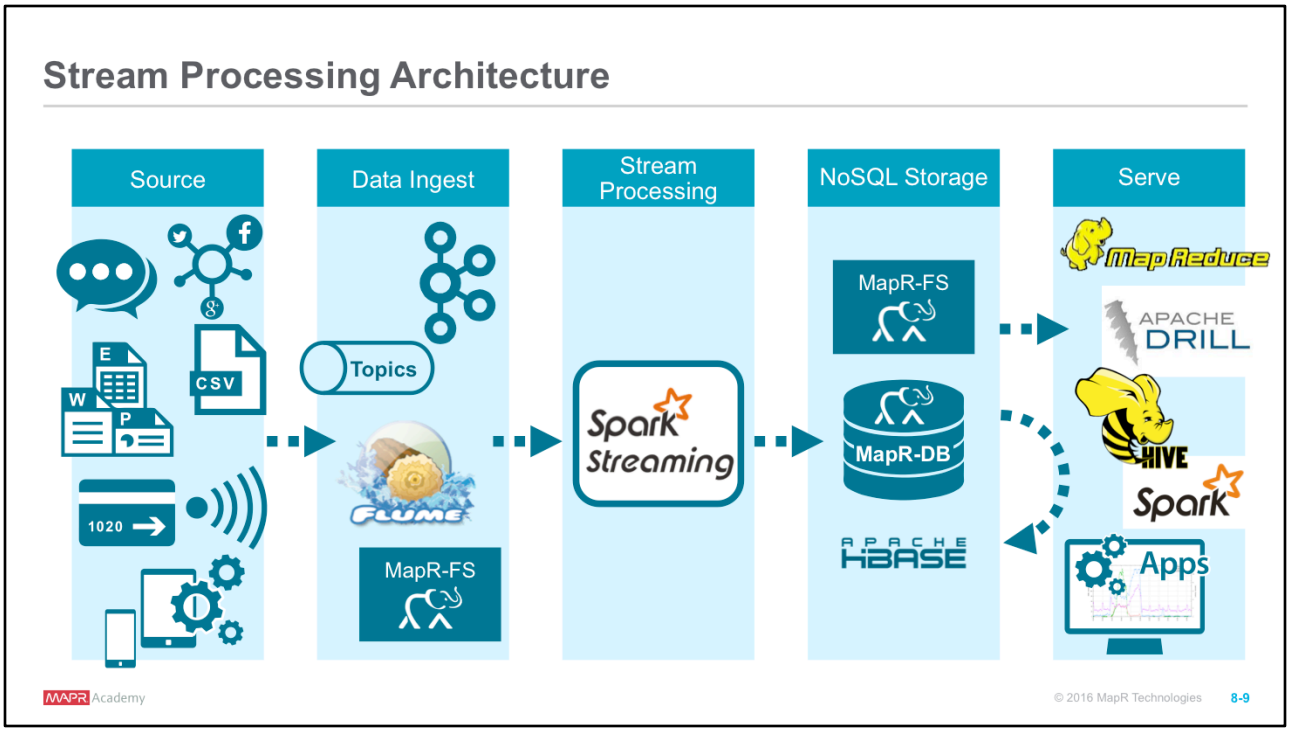


The data is then processed by a stream processing system like Spark Streaming, which is a framework for processing data streams.

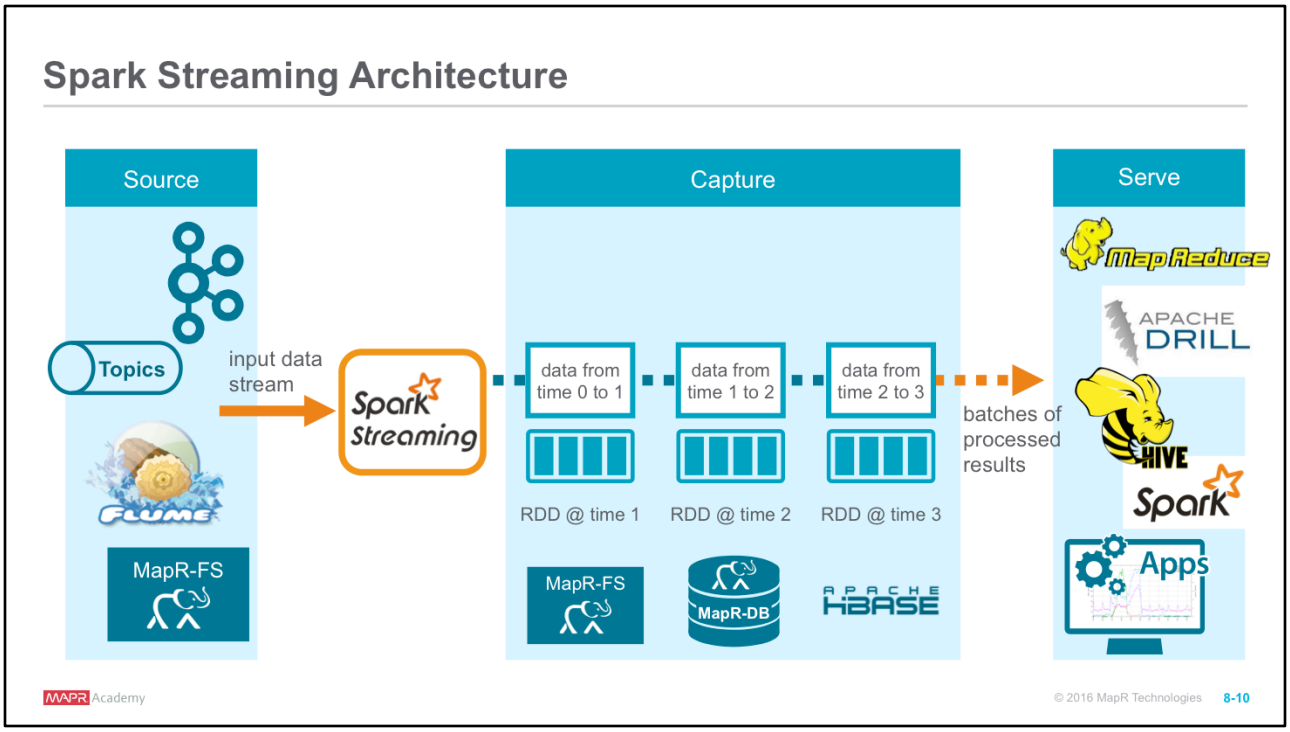
Stream Processing Architecture



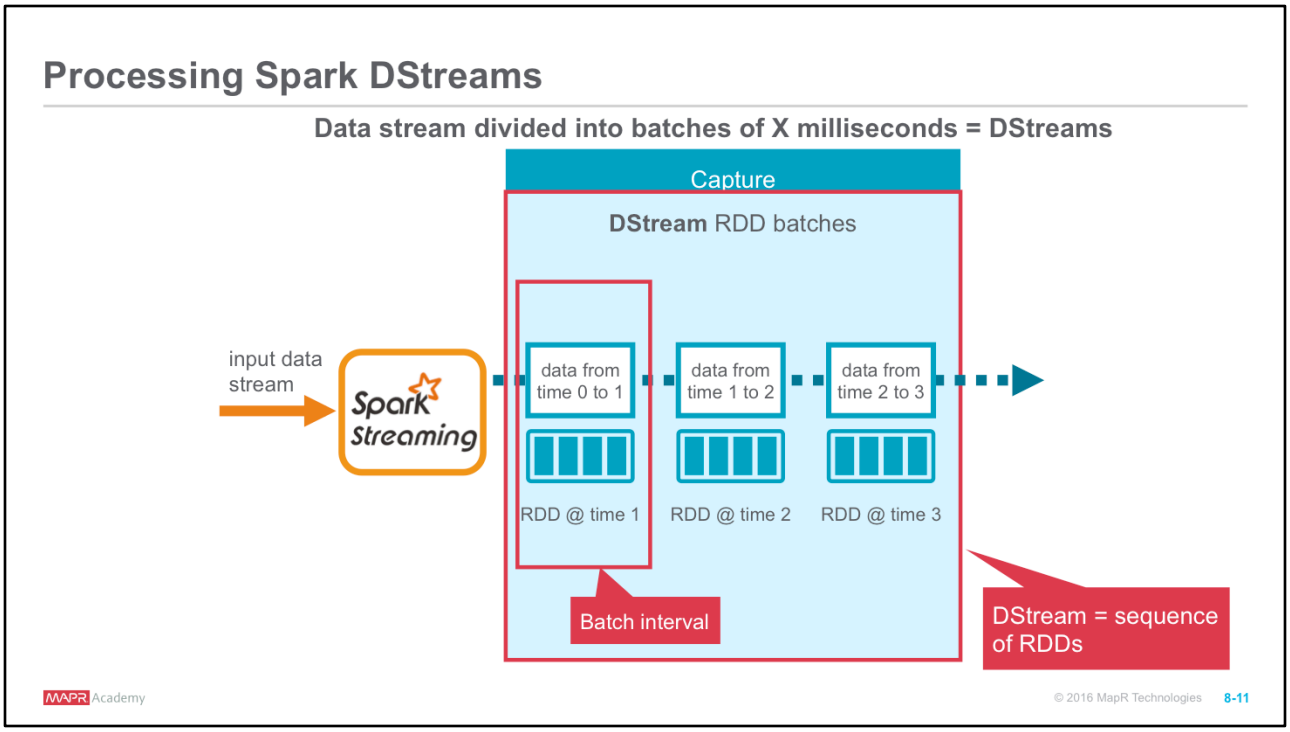
A NoSql database, such as HBase, MapR-DB, or MapR-FS is used for storing processed data. This system must be capable of low latency, fast read and writes.



End applications like dashboards, business intelligence tools and other applications use the processed data. The output can also be stored back in our database for further processing later.



Possible input sources include Flume, MapR Streams, and HDFS. Streaming data is continuous, but to process the data stream, it needs to be batched.



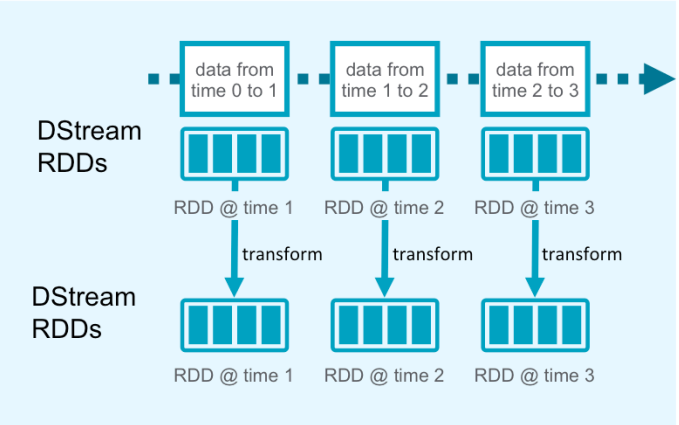
Spark Streaming divides the data stream into batches of X milliseconds called discretized streams, or DStreams.

A DStream is a sequence of mini-batches, where each mini-batch is represented as a Spark RDD. The stream is broken up into time periods equal to the batch interval.

Each RDD in the stream will contain the records that are received by the Spark Streaming application during the batch interval.

Processing Spark DStreams

Process using transformations → creates new RDDs

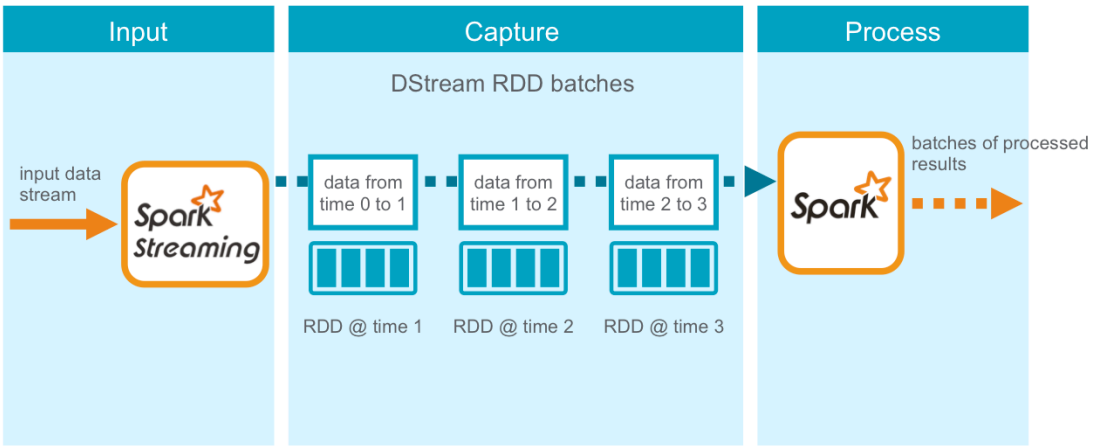


There are two types of operations on DStreams: *transformations* and *output operations*.

Your Spark application processes the DStream RDDs using Spark transformations like map, reduce, and join, which create new RDDs. Any operation applied on a DStream translates to operations on the underlying RDDs, which in turn, applies the transformation to the elements of the RDD.

Processing Spark DStreams

Processed results are pushed out in batches



Output operations are similar to RDD actions in that they write data to an external system, but in Spark Streaming they run periodically on each time step, producing output in batches.

Spark `StreamingContext`

- Entry point to all streaming functionality
- Can create new `StreamingContext` from:
  - Existing `SparkContext sc`

```
val ssc = new StreamingContext(sc, Seconds(5))
```

  - Existing `SparkConf conf`

```
val ssc = new StreamingContext(sparkConf, Seconds(5))
```


Spark StreamingContext is the entry point to all streaming functionality. It gives access to methods for creating DStreams from various input sources.

The StreamingContext can be created from an existing SparkContext or SparkConf, as shown in these examples. It can also be created by specifying a Spark master URL, and an app name.

Spark `StreamingContext`

- Entry point to all streaming functionality
- Can create new `StreamingContext` from:
  - Existing `SparkContext sc`

```
val ssc = new StreamingContext(sc, Seconds(5))
```

`batchDuration`
Time interval at which streaming data divided into batches

  - Existing `SparkConf conf`

```
val ssc = new StreamingContext(sparkConf, Seconds(5))
```


The second parameter here is the batchDuration. This is the time interval at which the streaming data is divided into batches.

Whether using the Spark Interactive Shell or creating a standalone application, we need to create a new StreamingContext.

Knowledge Check

Knowledge Check

Spark Streaming converts streaming data into DStreams. Which the statements below about DStreams are true?

- A DStream is a sequence of mini batches of streamed content
- Each mini-batch in a DStream is represented as a Spark RDD
- You can run Spark transformations and output operations on a DStream
- Spark `StreamingContext` gives access to methods for creating DStreams from various input sources
- `StreamingContext` can be created from an existing `SparkContext` or `SparkConf`

True
False – each complete DStream, sequence of mini-batches, is represented by an RDD
True
True
True

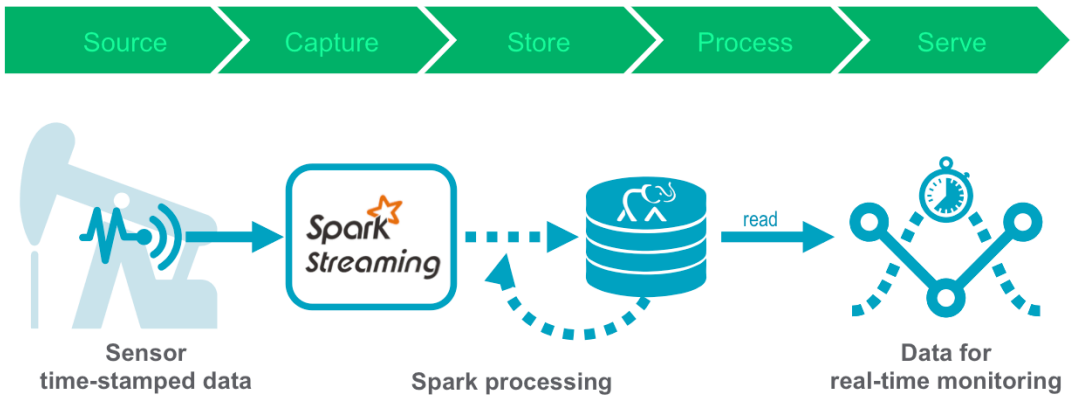
Learning Goals

Learning Goals

- Describe Spark Streaming Architecture
- **Create DStreams and a Spark Streaming Application**
  - **Define Use Case**
  - **Basic Steps**
  - **Save Data to HBase**
- Apply Operations on DStreams
- Define Windowed Operations
- Describe How Streaming Applications are Fault Tolerant

In this section, you will create DStreams and save the output to HBase tables. First, we will define our use case and application needs.

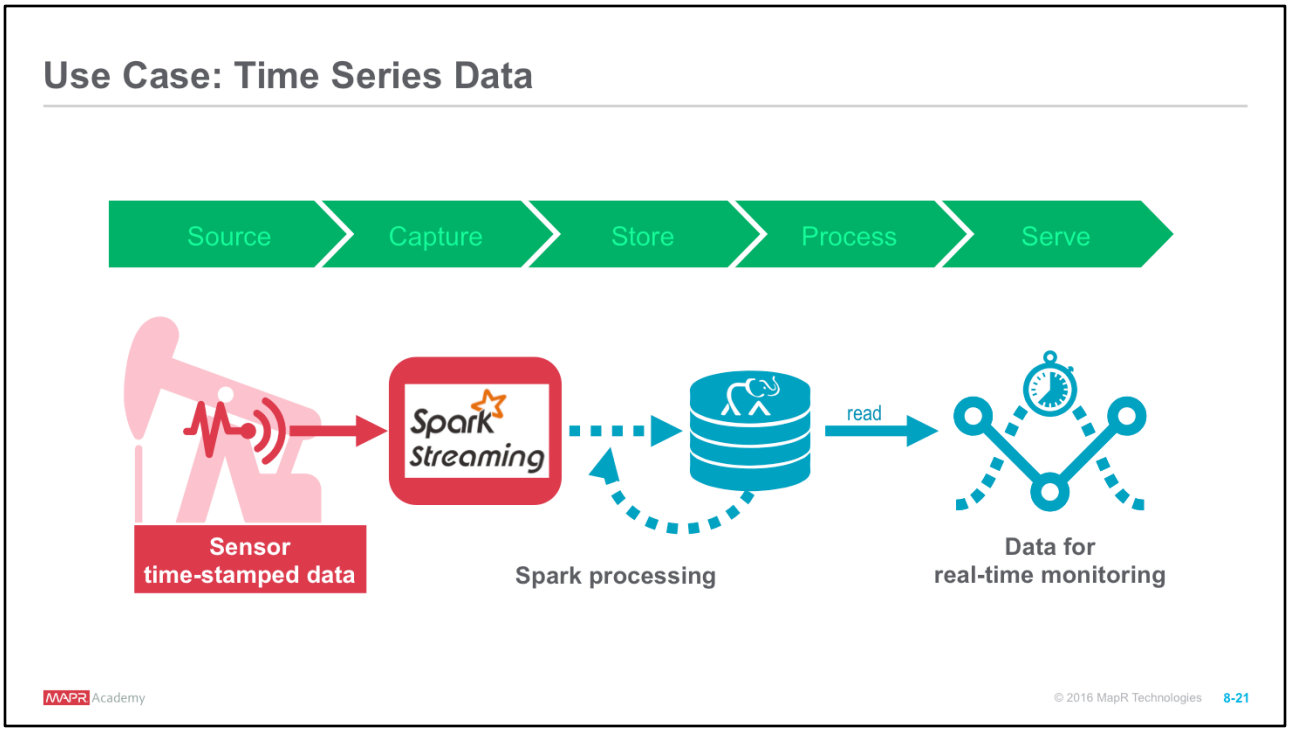
Use Case: Time Series Data



The example use case we will look at here is an application that monitors oil wells.

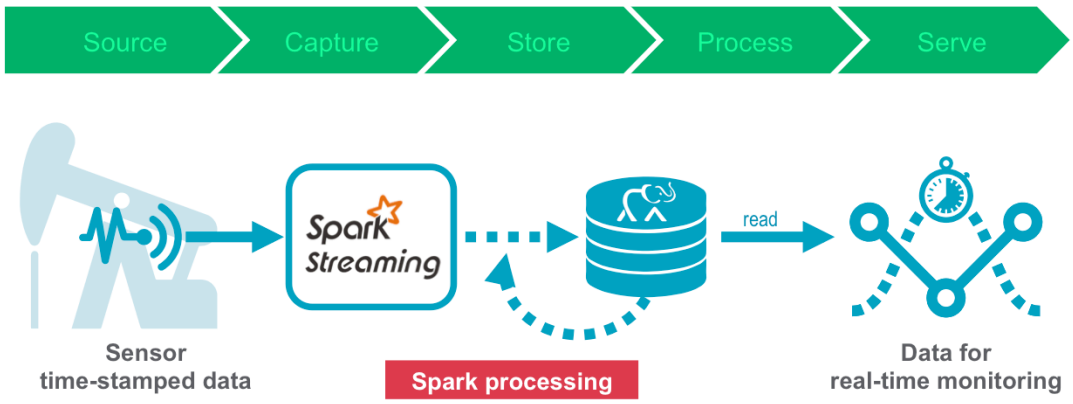
Sensors in oil rigs generate streaming data, which is processed by Spark and stored in HBase, for use by various analytical and reporting tools.

We want to store every single event in HBase as it streams in. We also want to filter for, and store alarms. Daily Spark processing will store aggregated summary statistics.



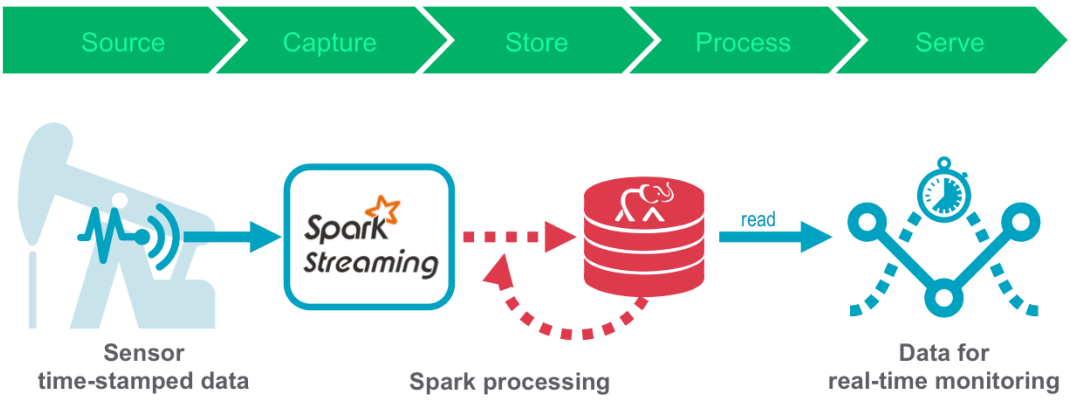
Our Spark Streaming example flow then, first reads streaming data…

Use Case: Time Series Data



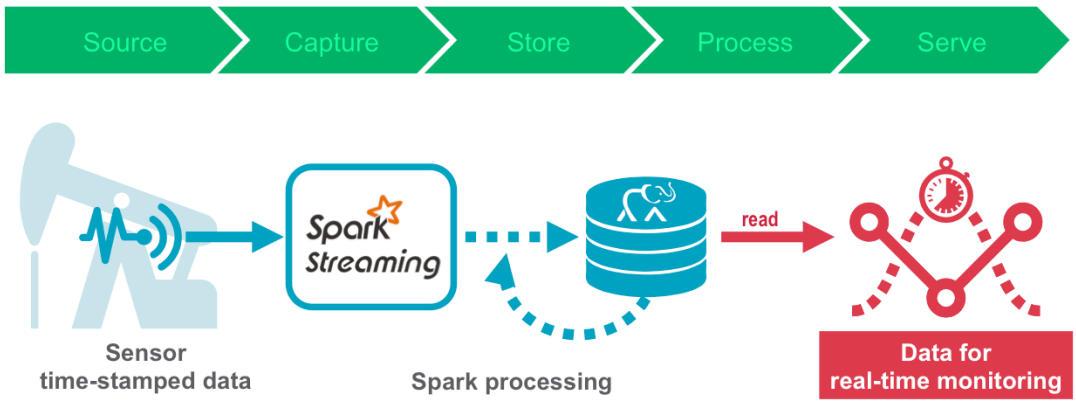
Processes the streaming data…

Use Case: Time Series Data



And writes the processed data to an HBase table.

Use Case: Time Series Data



Our non Streaming Spark code:

- Reads the HBase table data written by the streaming code
- Calculates daily summary statistics, and
- Writes summary statistics to the HBase table, Column Family stats

Convert Line of CSV Data to Sensor Object

sensordata.csv

```
1  ResourceID,Date,Time,HZ,Displace,Flow,SedimentPPM,PressureLbs,ChlorinePPM
2  COHUTTA,3/10/14,1:01,10.27,1.73,881,1.56,85,1.94
3  COHUTTA,3/10/14,1:02,9.67,1.731,882,0.52,87,1.79
4  COHUTTA,3/10/14,1:03,10.47,1.732,882,1.7,92,0.66
```

```
case class Sensor(resid: String, date: String, time: String,
   hz: Double, disp: Double, flo: Double, sedPPM: Double,
   psi: Double, chlPPM: Double)

def parseSensor(str: String): Sensor = {
   val p = str.split(",")
   Sensor(p(0), p(1), p(2), p(3).toDouble, p(4).toDouble, p(5).toDouble,
       p(6).toDouble, p(7).toDouble, p(8).toDouble)
}
```

MAPR Academy

 8-25

The oil pump sensor data comes in as comma separated value files, or CSV, saved to a directory. Spark Streaming will monitor the directory and process any files created into that directory. Spark Streaming supports many different streaming data sources. For simplicity, in this example we will use the MapR file system.

Unlike other Hadoop distributions that only allow cluster data import, or import as a batch operation, MapR lets you mount the cluster itself via NFS, so that your applications can read and write data directly. MapR allows direct file modification with multiple concurrent reads and writes via POSIX semantics. With an NFS-mounted cluster, you can read and write data directly with standard tools, applications, and scripts.

This means that MapR-FS is really easy to use with Spark streaming by putting data files into a directory.

Convert Line of CSV Data to Sensor Object

sensordata.csv

```
1  ResourceID,Date,Time,HZ,Displace,Flow,SedimentPPM,PressureLbs,ChlorinePPM
2  COHUTTA,3/10/14,1:01,10.27,1.73,881,1.56,85,1.94
3  COHUTTA,3/10/14,1:02,9.67,1.731,882,0.52,87,1.79
4  COHUTTA,3/10/14,1:03,10.47,1.732,882,1.7,92,0.66
```

```
case class Sensor(resid: String, date: String, time: String,
   hz: Double, disp: Double, flo: Double, sedPPM: Double,
   psi: Double, chlPPM: Double)

def parseSensor(str: String): Sensor = {
   val p = str.split(",")
   Sensor(p(0), p(1), p(2), p(3).toDouble, p(4).toDouble, p(5).toDouble,
       p(6).toDouble, p(7).toDouble, p(8).toDouble)
}
```


We use a Scala case class to define the Sensor schema corresponding to the sensor data CSV files…

Convert Line of CSV Data to Sensor Object

sensordata.csv

```
1  ResourceID,Date,Time,HZ,Displace,Flow,SedimentPPM,PressureLbs,ChlorinePPM
2  COHUTTA,3/10/14,1:01,10.27,1.73,881,1.56,85,1.94
3  COHUTTA,3/10/14,1:02,9.67,1.731,882,0.52,87,1.79
4  COHUTTA,3/10/14,1:03,10.47,1.732,882,1.7,92,0.66
```

```
case class Sensor(resid: String, date: String, time: String,
   hz: Double, disp: Double, flo: Double, sedPPM: Double,
   psi: Double, chlPPM: Double)

def parseSensor(str: String): Sensor = {
   val p = str.split(",")
   Sensor(p(0), p(1), p(2), p(3).toDouble, p(4).toDouble, p(5).toDouble,
       p(6).toDouble, p(7).toDouble, p(8).toDouble)
}
```


…and a parseSensor function to parse the comma separated values into the sensor case class.

Learning Goals

Learning Goals

- Describe Spark Streaming Architecture
- **Create DStreams and a Spark Streaming Application**
  - **Define Use Case**
  - **Basic Steps**
  - **Save Data to HBase**
- Apply Operations on DStreams
- Define Windowed Operations
- Describe How Streaming Applications are Fault Tolerant

In this section, you will create the DStreams.

Basic Steps for Spark Streaming Code

1. Initialize a Spark `StreamingContext` object
2. Using context, create a DStream
   - Represents streaming data from a source
   1. Apply transformations
      - Creates new DStreams
   2. And/or apply output operations
      - Persists or outputs data
3. Start receiving and processing data
   - Using `streamingContext.start()`
4. Wait for the processing to be stopped
   - Using `streamingContext.awaitTermination()`

Shown here are the basic steps for Spark Streaming code:

First, initialize a Spark StreamingContext object.

Using this context, create a Dstream, which represents streaming data from a source. Apply transformations and/or output operations to the Dstreams.

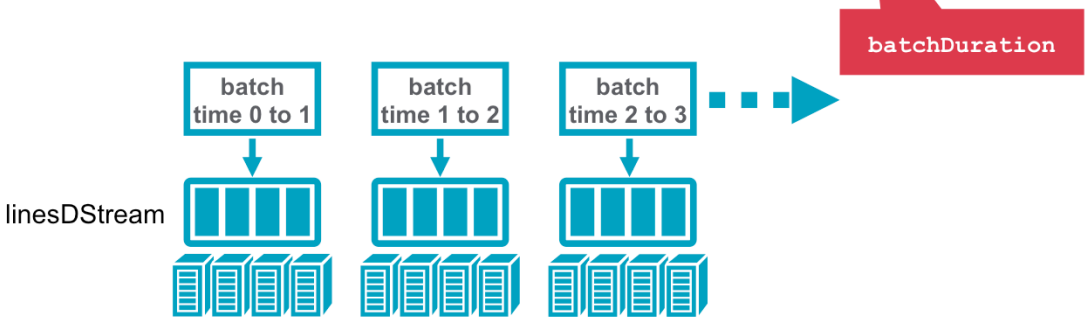
We can then start receiving data and processing it using streamingContext.start().

And finally we wait for the processing to be stopped using streamingContext.awaitTermination()

We will go through these steps showing code from our use case example.
Spark Streaming programs are best run as standalone applications built using Maven or sbt. In the lab you will use the shell for simplicity, and build a streaming application.

Create a DStream

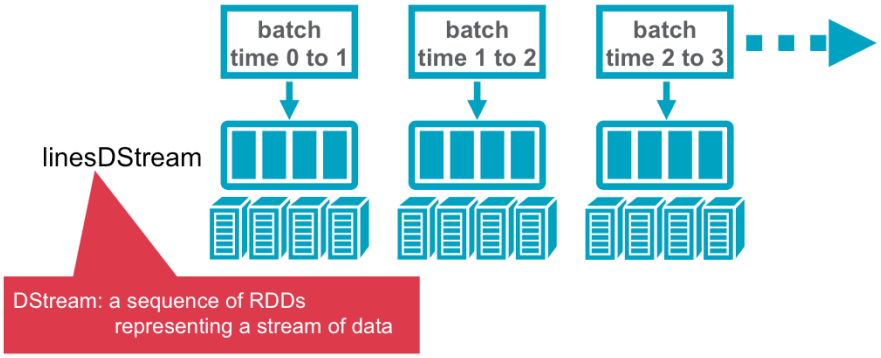
```
val ssc = new StreamingContext(sparkConf, Seconds(2))
val linesDStream = ssc.textFileStream("/mapr/stream")
```



The first step is to create a StreamingContext, which is the main entry point for streaming functionality. In this example we will use a 2 second batch interval.

Create a DStream

```
val ssc = new StreamingContext(sparkConf, Seconds(2))
val linesDStream = ssc.textFileStream("/mapr/stream")
```



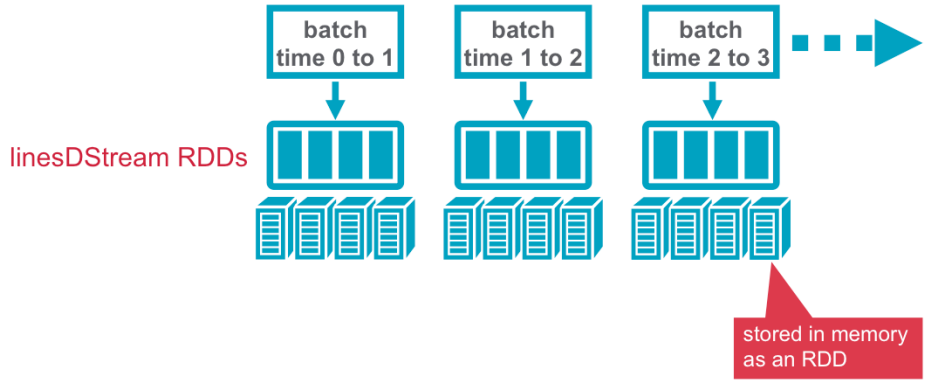
Using this context, we can create a DStream that represents streaming data from a source.

In this example we use the StreamingContext textFileStream method to create an input stream that monitors a Hadoop-compatible file system for new files, and processes any files created in that directory.

This ingestion type supports a workflow where new files are written to a landing directory and Spark Streaming is used to detect them, ingest them, and process the data. Only use this ingestion type with files that are moved or copied into a directory.

Create a DStream

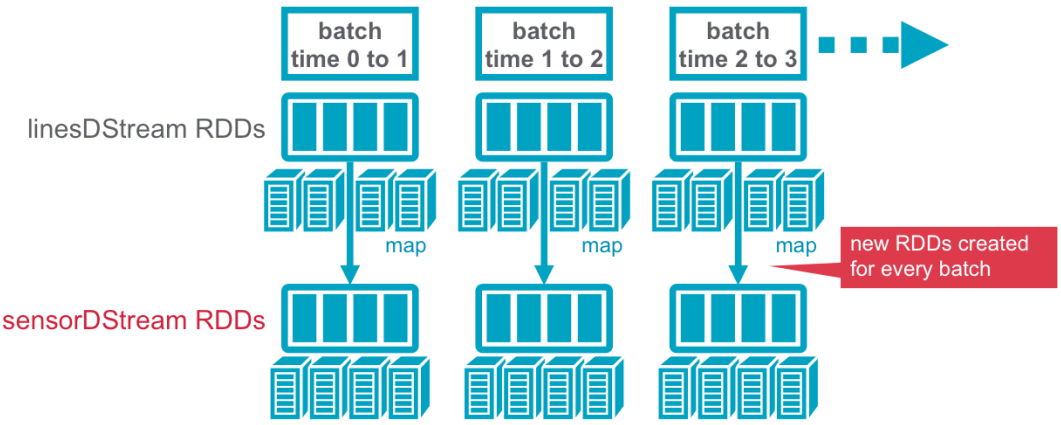
```
val ssc = new StreamingContext(sparkConf, Seconds(2))
val linesDStream = ssc.textFileStream("/mapr/stream")
```



linesDStream represents the stream of incoming data, where each record is a line of text. Internally a DStream is a sequence of RDDs, one RDD per batch interval.

Process DStream

```
val linesDStream = ssc.textFileStream("directory path")
val sensorDStream = linesDStream.map(parseSensor)
```



Next we parse the lines of data into Sensor objects, with the map operation on the linesDStream.

The map operation applies the Sensor.parseSensor function on the RDDs in the linesDStream, resulting in RDDs of Sensor objects.

Any operation applied on a DStream translates to operations on the underlying RDDs. the map operation is applied on each RDD in the linesDStream to generate the sensorDStream RDDs.

Process DStream

```
// for Each RDD
sensorDStream.foreachRDD { rdd =>
   // filter sensor data for low psi
   val alertRDD = sensorRDD.filter(sensor => sensor.psi < 5.0)
   . . .
}
```


Next we use the DStream foreachRDD method to apply processing to each RDD in this DStream. We filter the sensor objects for low PSI to create an RDD of alert sensor objects.

Start Receiving Data

```
sensorDStream.foreachRDD {  rdd =>
    . . .
}
// Start the computation
ssc.start()
// Wait for the computation to terminate
ssc.awaitTermination()
```


To start receiving data, we must explicitly call start() on the StreamingContext, then call awaitTermination to wait for the streaming computation to finish.

Streaming Application Output

```
-------------------------------------------
Time: 1452886488000 ms
-------------------------------------------
COHUTTA,3/10/14,1:01,10.27,1.73,881,1.56,85,1.94
COHUTTA,3/10/14,1:02,9.67,1.731,882,0.52,87,1.79
COHUTTA,3/10/14,1:03,10.47,1.732,882,1.7,92,0.66
COHUTTA,3/10/14,1:05,9.56,1.734,883,1.35,99,0.68
COHUTTA,3/10/14,1:06,9.74,1.736,884,1.27,92,0.73
COHUTTA,3/10/14,1:08,10.44,1.737,885,1.34,93,1.54
COHUTTA,3/10/14,1:09,9.83,1.738,885,0.06,76,1.44
COHUTTA,3/10/14,1:11,10.49,1.739,886,1.51,81,1.83
COHUTTA,3/10/14,1:12,9.79,1.739,886,1.74,82,1.91
COHUTTA,3/10/14,1:13,10.02,1.739,886,1.24,86,1.79
...

low pressure alert
Sensor(NANTAHALLA,3/13/14,2:05,0.0,0.0,0.0,1.73,0.0,1.51)
Sensor(NANTAHALLA,3/13/14,2:07,0.0,0.0,0.0,1.21,0.0,1.51)
-------------------------------------------
Time: 1452886490000 ms
-------------------------------------------
```


The output from our application will show the name of the oil rig, and streamed data.

Knowledge Check

Knowledge Check

Using the Spark Streaming code here:

```
val ssc = new StreamingContext(sparkConf, Seconds(30))
val linesDStream = ssc.textFileStream("/mapr/source/webstream")
```

- **What is the batch interval?**
- **What method is used to create an input stream?**
- **Where is the streaming data saved?**

Batch interval = 30 seconds

ssc.textFileStream is the method used to create the input stream

Streaming data is saved into the directory: /mapr/source/webstream

Learning Goals

Learning Goals

- Describe Spark Streaming Architecture
- **Create DStreams and a Spark Streaming Application**
  - **Define Use Case**
  - **Basic Steps**
  - **Save Data to HBase**
- Apply Operations on DStreams
- Define Windowed Operations
- Describe How Streaming Applications are Fault Tolerant

In this section, you will save the process stream data into an HBase table.

Configure to Write to HBase

```
// set JobConfiguration variables for writing to HBase
val jobConfig: JobConf = new JobConf(conf, this.getClass)
jobConfig.setOutputFormat(classOf[TableOutputFormat])
jobConfig.set(TableOutputFormat.OUTPUT_TABLE, tableName)
```


We register the DataFrame as a table, which allows us to use it in subsequent SQL statements.

Now we can inspect the data.

Schema

- All events stored, CF data could be set to expire data
- Filtered alerts put in CF alerts
- Daily summaries put in CF stats

Row key	CF data			CF alerts		CF stats		
	hz	…	psi	psi	…	hz_avg	…	psi_min
COHUTTA_3/10/14_1:01	10.37		84	0				
COHUTTA_3/10/14						10		0

Row Key contains oil pump name, date, and a time stamp

The HBase table schema for the streaming data is as follows:

The row key is a composite of the oil pump name, date, and a time stamp

Schema

- All events stored, CF data could be set to expire data
- Filtered alerts put in CF alerts
- Daily summaries put in CF stats

Row key	CF data			CF alerts		CF stats		
	hz	…	psi	psi	…	hz_avg	…	psi_min
COHUTTA_3/10/14_1:01	10.37		84	0				
COHUTTA_3/10/14						10		0

The Column Family “data” contains columns corresponding to the input data fields.

The Column Family “alerts” contains columns corresponding to any filters for alarm values.

Note that the data and alert column families could be set to expire values after a certain amount of time.

Schema

- All events stored, CF data could be set to expire data
- Filtered alerts put in CF alerts
- Daily summaries put in CF stats

Row key	CF data			CF alerts		CF stats		
	hz	…	psi	psi	…	hz_avg	…	psi_min
COHUTTA_3/10/14_1:01	10.37		84	0				
COHUTTA_3/10/14						10		0

The Schema for the daily statistics summary rollups is as: the composite row key of the pump name and date, and then the Column Family “stats”

Function to Convert Sensor Data to HBase Put Object

```
//  Convert a row of sensor object data to an HBase put object
def convertToPut(sensor: Sensor): (ImmutableBytesWritable, Put) = {
    val put = new Put(Bytes.toBytes(rowkey))
    // add column  values to put object
    . . .
    put.add(Bytes.toBytes("data"), Bytes.toBytes("psi"),
        Bytes.toBytes(sensor.psi))
    return (new ImmutableBytesWritable(Bytes.toBytes(rowkey)), put)
}
```


This function converts a Sensor object into an HBase Put object, which is used to insert a row into HBase.

Save to HBase

```
// for Each RDD parse into a sensor object filter
sensorDStream.foreachRDD { rdd =>
   . . .
   // convert alert to put object write to HBase alerts
   rdd.map(Sensor.convertToPutAlert)
      .saveAsHadoopDataset(jobConfig)
}
```


We use the DStream foreachRDD method to apply processing to each RDD in the DStream.

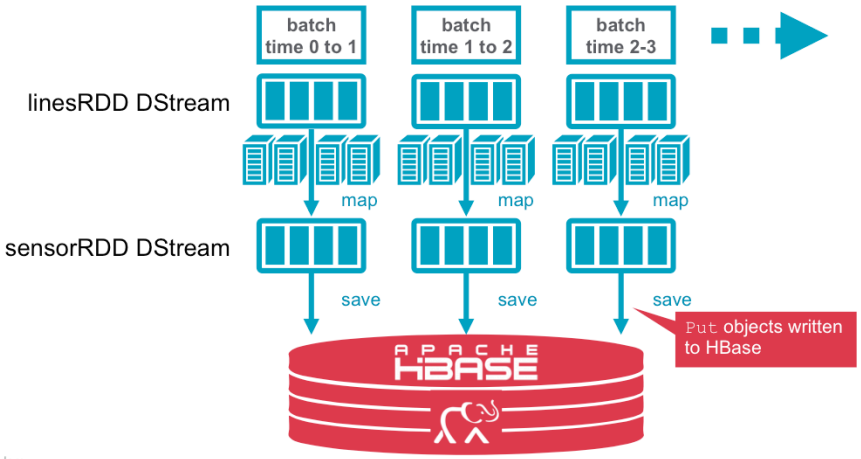
We filter the sensor objects for low PSI values, to create alerts, and then convert the data to HBase Put objects using the convertToPutAlert fuction.

Next we write the sensor and alert data to HBase using the PairRDDFunctions saveAsHadoopDataset method. This outputs the RDD to any Hadoop-supported storage system using a Hadoop Configuration object for that storage system.

Save to HBase

```
rdd.map(Sensor.convertToPut).saveAsHadoopDataset(jobConfig)
```

output operation: persist data to external storage



The sensorRDD objects are converted to Put objects, and then written to HBase, using the saveAsHadoopDataset method.

Learning Goals

Learning Goals

- Describe Spark Streaming Architecture
- Create DStreams and a Spark Streaming Application
- **Apply Operations on DStreams**
- Define Windowed Operations
- Describe How Streaming Applications are Fault Tolerant

This section you will learn how to apply operations to your DStreams.

Operations on DStreams

- What is the pump vendor and maintenance information for sensors with low pressure alerts?
- What is the Max, Min, and Average for sensor attributes?

Now that we have our input data stream, we would like to answer some questions such as:

What is the pump vendor and maintenance information for sensors with low pressure alerts?

What is the max, min, and average for sensor attributes?

To answer these questions, we will use operations on the DStream we just created.

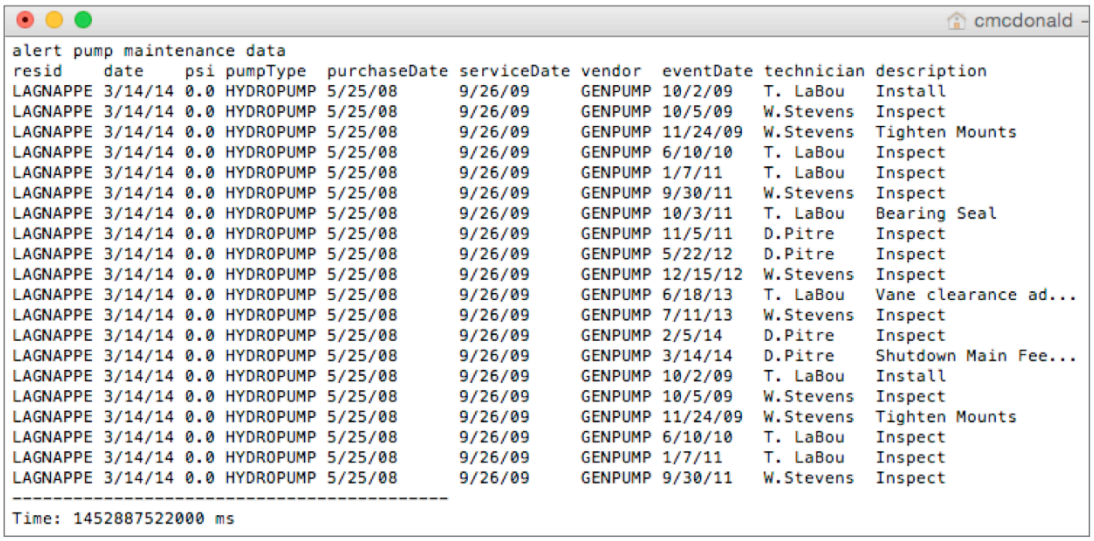
DataFrame and SQL Operations

```
// put pump vendor and maintenance data in temp table
sc.textFile("vendor.csv").map(parsePump).toDF().registerTempTable("pump")
sc.textFile("maint.csv").map(parseMaint).toDF().registerTempTable("maint")
// for Each RDD
sensorDStream.foreachRDD { rdd =>
  val sqlContext = SQLContext.getOrCreate(rdd.sparkContext)
  rdd.filter(sensor => sensor.psi < 5.0).toDF.registerTempTable("alert")
  val alertpumpmaint = sqlContext.sql("select s.resid, s.date, s.psi,
     p.pumpType, p.vendor, m.eventDate, m.technician from alert s join
     pump p on s.resid = p.resid join maint m on p.resid=m.resid")
  alertpumpmaint.show
}
```


Here we join the filtered alert data with pump vendor and maintenance information, which was read in and cached before streaming. Each RDD is converted into a DataFrame, registered as a temporary table and then queried using SQL.

This Query answers our first question:
What is the pump vendor and maintenance information for sensors with low pressure alerts?

Streaming Application Output



Here we see some sample output, answering our first question.

DataFrame and SQL Operations

```
// for Each RDD
sensorDStream.foreachRDD { rdd =>
  val sqlContext = SQLContext.getOrCreate(rdd.sparkContext)
  rdd.toDF().registerTempTable("sensor")
  val res = sqlContext.sql( "SELECT resid, date,
   max(hz) as maxhz, min(hz) as minhz, avg(hz) as avghz,
   max(disp) as maxdisp, min(disp) as mindisp, avg(disp) as avgdisp,
   max(flo) as maxflo, min(flo) as minflo, avg(flo) as avgflo,
   max(psi) as maxpsi, min(psi) as minpsi, avg(psi) as avgpsi
   FROM sensor GROUP BY resid,date")

   res.show()
}
```


Our second question:

What is the max, min, and average for sensor attributes?

Is addressed by the query shown here.

Streaming Application Output

sensor max, min, averages

resid	date	maxhz	minhz	avghz	maxdisp	mindisp	avgdisp	maxflo	minflo	avgflo
LAGNAPPE	3/12/14	10.5	9.5	9.988799582463459	3.235	1.623	2.4714206680584563	1570.0	788.0	1199.1022964509395
CHER	3/13/14	10.5	9.5	10.005271398747391	3.552	1.857	2.732156576200417	1670.0	873.0	1284.660751565762
BBKING	3/13/14	10.5	9.5	9.996033402922755	1.58	0.902	1.26099164926931	1822.0	1041.0	1454.0240083507306
MOJO	3/12/14	10.5	9.5	10.006482254697294	3.589	2.143	2.888004175365341	1899.0	1134.0	1528.2724425887266
CARGO	3/10/14	10.5	9.5	9.998507306889353	3.752	1.903	2.8525417536534423	1533.0	778.0	1165.5302713987473
LAGNAPPE	3/11/14	10.5	9.5	10.009540709812102	3.092	1.454	2.307491649269313	1500.0	706.0	1119.5647181628392
CHER	3/12/14	10.5	9.5	10.008256784968692	3.443	1.728	2.6184937369519825	1619.0	812.0	1231.230688935282
BBKING	3/12/14	10.5	9.5	10.006795407098123	1.556	0.862	1.2138110647181624	1794.0	994.0	1399.608559498956
MOJO	3/11/14	10.5	9.5	10.0029331941545	3.513	1.979	2.8009843423799587	1859.0	1047.0	1482.2160751565762
LAGNAPPE	3/10/14	10.5	9.5	10.00329853862213	3.116	1.453	2.349840292275576	1512.0	705.0	1140.1022964509395
CHER	3/11/14	10.5	9.5	9.990396659707717	3.325	1.691	2.545035490605431	1564.0	795.0	1196.6837160751566
BBKING	3/11/14	10.5	9.5	9.997348643006282	1.49	0.821	1.1701722338204592	1718.0	947.0	1349.2849686847599
MOJO	3/10/14	10.5	9.5	9.999457202505194	3.345	1.828	2.6188089770354868	1770.0	967.0	1385.8131524008352
CHER	3/10/14	10.5	9.5	9.998726513569954	3.172	1.653	2.4233079331941516	1492.0	777.0	1139.4488517745303
BBKING	3/10/14	10.5	9.5	10.001409185803748	1.502	0.821	1.18567223382046	1732.0	947.0	1367.1711899791233
THERMALITO	3/14/14	10.5	9.5	9.983862212943635	3.784	2.135	3.0065960334029254	1680.0	948.0	1335.1002087682673
THERMALITO	3/13/14	10.5	9.5	9.999311064718169	3.896	2.116	3.069195198329852	1730.0	940.0	1362.910229645094
ANDOUILLE	3/14/14	10.5	9.5	10.011388308977041	2.146	1.139	1.6609373695198306	1592.0	845.0	1232.2296450939457
THERMALITO	3/12/14	10.5	9.5	10.007526096033398	3.823	1.986	2.9294561586638808	1697.0	882.0	1300.8622129436326
ANDOUILLE	3/13/14	10.5	9.5	9.99043841336118	2.174	1.104	1.656331941544886	1613.0	819.0	1228.8371607515658

And output from that query shows the max, min, and average output from our sensors.

Key Concepts

- **Data sources:**
  - File based: HDFS
  - Network based: TCP sockets
  - Twitter, Kafka, Flume, ZeroMQ, Akka Actor
- Transformations: create new DStream
  - Standard RDD operations: `map, countByValue, reduce, join, ...`
  - Stateful operations: `UpdateStateByKey(function), countByValueAndWindow, ...`
- Output operations: trigger computation
  - `print` – prints first 10 elements
  - `saveAsObjectFile, saveAsTextFiles, saveAsHadoopFiles` – save to HDFS
  - `forEachRDD` – do anything with each batch of RDDs

Spark Streaming supports various input sources, including file-based sources and network-based sources such as socket-based sources, the Twitter API stream, Akka actors, or message queues and distributed stream and log transfer frameworks, such Flume, Kafka, and Amazon Kinesis.

Key Concepts

- Data sources:
  - File based: HDFS
  - Network based: TCP sockets
  - Twitter, Kafka, Flume, ZeroMQ, Akka Actor

- **Transformations: create new DStream**
  - Standard RDD operations: `map, countByValue, reduce, join, ...`
  - Stateful operations: `UpdateStateByKey(function), countByValueAndWindow, ...`

- Output operations: trigger computation
  - `print` – prints first 10 elements
  - `saveAsObjectFile, saveAsTextFiles, saveAsHadoopFiles` – save to HDFS
  - `foreachRDD` – do anything with each batch of RDDs

Spark Streaming provides a set of transformations available on DStreams. These transformations are similar to those available on RDDs. They include map, flatMap, filter, join, and reduceByKey. Streaming also provides operators such as reduce and count.

These operators return a DStream made up of a single element. Unlike the reduce and count operators on RDDs, these do not trigger computation on DStreams. They are not actions, they return another DStream.

Stateful transformations maintain state across batches, they use data or intermediate results from previous batches to compute the results of the current batch. They include transformations based on sliding windows and on tracking state across time. updateStateByKey() is used to track state across events for each key. For example this could be used to maintain stats (state) by a key accessLogDStream.reduceByKey(SUM_REDUCER).

Streaming transformations are applied to each RDD in the DStream, which, in turn, applies the transformation to the elements of the RDD.

Want to see more? Find a full list of transformations at: http://spark.apache.org/docs/latest/streaming-programming-guide.html

Key Concepts

- Data sources:
  - File based: HDFS
  - Network based: TCP sockets
  - Twitter, Kafka, Flume, ZeroMQ, Akka Actor
- Transformations: create new DStream
  - Standard RDD operations: `map, countByValue, reduce, join, ...`
  - Stateful operations: `UpdateStateByKey(function), countByValueAndWindow, ...`
- **Output operations: trigger computation**
  - `print` – prints first 10 elements
  - `saveAsObjectFile, saveAsTextFiles, saveAsHadoopFiles` – save to HDFS
  - `forEachRDD` – do anything with each batch of RDDs

Actions are output operators that, when invoked, trigger computation on the DStream. They include:

print, which prints the first 10 elements of each batch to the console, and is typically used for debugging and testing.

The saveAsObjectFile, saveAsTextFiles, and saveAsHadoopFiles functions output each batch to a Hadoop-compatible file system.

And the forEachRDD operator applies any arbitrary processing to the RDDs within each batch of a DStream.

Knowledge Check

Knowledge Check

Indicate whether the statements below are TRUE or FALSE:

- **Transformations on a DStream return another DStream**
- **Stateful transformations maintain their state across batches**
- **Transformations trigger computations**
- **`forEachRDD` will apply processing to the RDDs within each batch of a DStream**
- **DStreams can only be made from streaming sources**

True
True
False - output actions trigger actions. Transformations return another DStream.
True
False – DStreams can be made from file-based sources and network-based sources such as socket-based sources.

Learning Goals

Learning Goals

- Describe Spark Streaming Architecture
- Create DStreams and a Spark Streaming Application
- Apply Operations on DStreams
- **Define Windowed Operations**
- Describe How Streaming Applications are Fault Tolerant

This section discusses widowed operations.

Sliding Windows

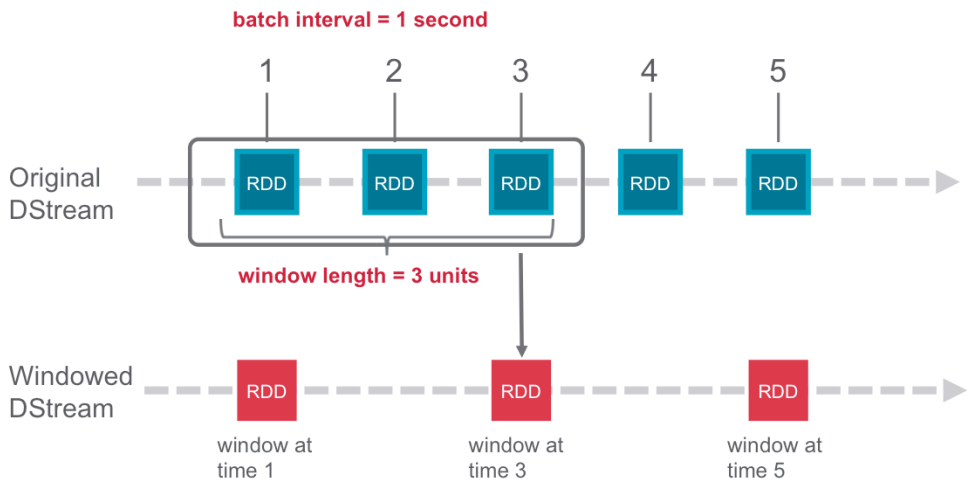
- Can compute results across longer time period
- Example – want oil pressure data across last 12 intervals (60 seconds) computed after every 6 intervals (30 seconds)
  - Create a sliding window of the last 60 seconds
  - Computed every 30 seconds

With window operations you can apply transformations over a sliding window of data. This allows us to perform computations over a longer time period than that specified in the StreamingContext batch Interval, by combining results from multiple batches.

For example, our data is coming in at 5-second intervals. The DStream computations are performed every 5 seconds across the data in that 5-second interval.

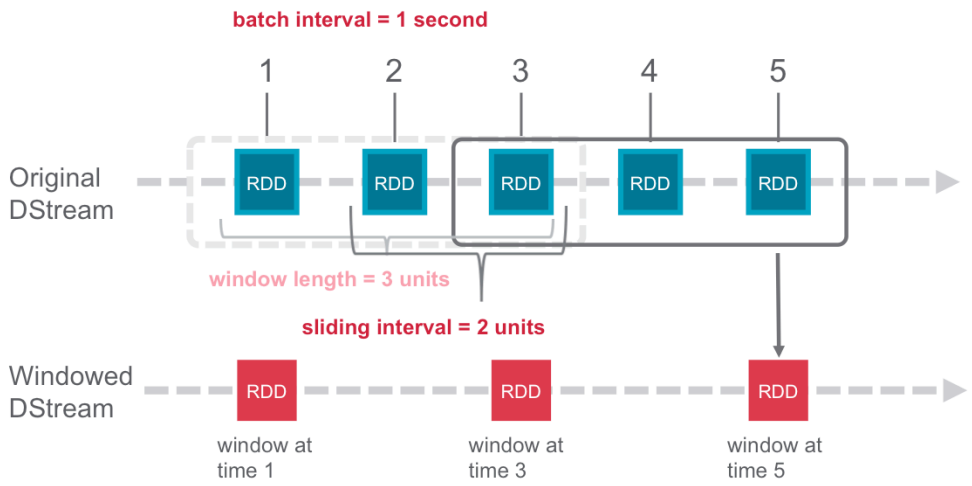
Say, we want to compute the oil pressure at a specific station every 30 seconds over the last 60 seconds. We create a sliding window of the last 60 seconds and compute every 30 seconds.

Windowed Computations



In this illustration, the original stream is coming in at one second intervals. The length of the sliding window is specified by the window length, in this case 3 units.

Windowed Computations



The window slides over the source DStream at the specified sliding interval, in this case 2 units.

Both the window length and the sliding interval must be multiples of the batch interval of the source DStream, which in this case is 1 second. When the window slides over the source DStream, all the RDDs that fall in that window are combined. The operation is applied to the combined RDDs resulting in RDDs in the windowed stream.

All window operations require two parameters:
The window length, is the duration of the window. In this example the window length is 3 units.
And the sliding interval, which is the interval at which the operation window is performed. In this example to sliding interval is 2 units.

Again, both of these parameters must be multiples of the batch interval of the original DStream.

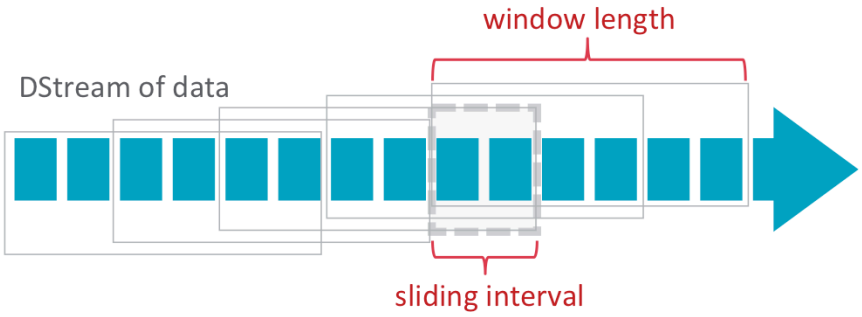
Window-based Transformations

```
val windowedWordCounts = pairs.reduceByKeyAndWindow(
    (a:Int,b:Int) => (a + b), Seconds(12), Seconds(4))
```

sliding window operation

window length

sliding interval



For example we want to generate word counts every 4 seconds, over the last 6 seconds of data. To do this, we apply the reduceByKey operation on the DStream of paired 2-tuples (word and the number 1), over the last 6 seconds of data using the operation reduceByKeyAndWindow.

Window Operations

Window Operation	Description
`window(windowLength,slideInterval)`	Returns new DStream computed based on windowed batches of source DStream.
`countByWindow(windowLength, slideInterval)`	Returns a sliding window count of elements in the stream
`reduceByWindow(func,windowLength,slideInterval)`	Returns a new single-element stream created by aggregating elements over sliding interval using `func`.
`reduceByKeyAndWindow(func, windowLength,slideInterval, [numTasks])`	Returns a new DStream of (K,V) pairs from DStream of (K,V) pairs; aggregates using given reduce function `func` over batches of sliding window.
`countByValueAndWindow(windowLength,slideInterval, [numTasks])`	Returns new DStream of (K,V) pairs where value of each key is its frequency within a sliding window; it acts on DStreams of (K,V) pairs

Output operations are used to push the data in DStreams to an external system such as a database or file system. Some commonly used output operations are listed here.

Window Operations on DStreams

With a windowed stream :

- What is the count of sensor events by pump ID?
- What is the Max, Min, and Average for PSI?

We would like to answer some questions such as:

With a windowed stream of 6 seconds of data, every 2 seconds:

What is the count of sensor events by pump ID?

What is the Max, Min, and Average pump PSI?

To answer these questions, we will use operations on a windowed stream.

DataFrame and SQL Operations

```
// Windowed 6 seconds of data, every 2 seconds
sensorDStream.window(Seconds(6), Seconds(2))
// for Each RDD
sensorDStream.foreachRDD { rdd =>
  val sqlContext = SQLContext.getOrCreate(rdd.sparkContext)
  rdd.toDF().registerTempTable("sensor")
  val res = sqlContext.sql("SELECT resid, date, count(resid) as total
    FROM sensor GROUP BY resid, date")

  res.show
}
```


The code here performs operations on 6 seconds worth of data, on a 2 second window.

DataFrame and SQL Operations

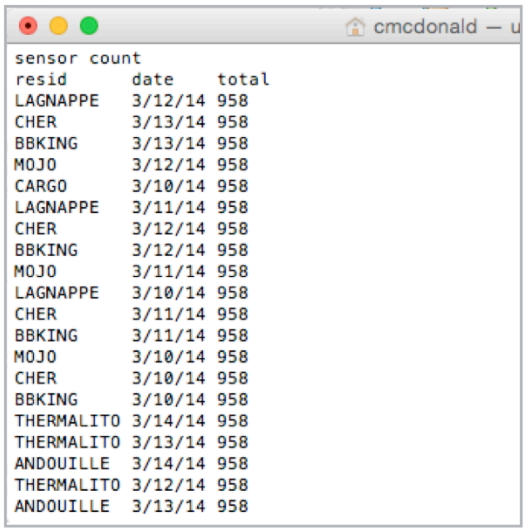
```
// Windowed 6 seconds of data, every 2 seconds
sensorDStream.window(Seconds(6), Seconds(2))
// for Each RDD
sensorDStream.foreachRDD { rdd =>
  val sqlContext = SQLContext.getOrCreate(rdd.sparkContext)
  rdd.toDF().registerTempTable("sensor")
  val res = sqlContext.sql("SELECT resid, date, count(resid) as total
    FROM sensor GROUP BY resid, date")

  res.show
}
```


We answer our first question:

What is the count of sensor events by pump ID?

By counting the ID on each sensor DStream RDD.

Streaming Application Output



This output shows the answer for our first question.

DataFrame and SQL Operations

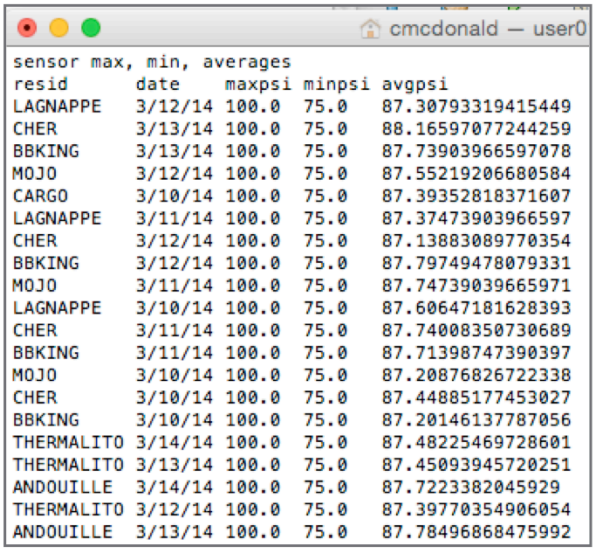
```
// Windowed 6 seconds of data, every 2 seconds
sensorDStream.window(Seconds(6), Seconds(2))
// for Each RDD
sensorDStream.foreachRDD { rdd =>
  val sqlContext = SQLContext.getOrCreate(rdd.sparkContext)
  rdd.toDF().registerTempTable("sensor")
  val res = sqlContext.sql("SELECT resid, date, max(psi) as maxpsi,
    min(psi) as minpsi, avg(psi) as avgpsi
    FROM sensor GROUP BY resid,date")
  res.show
}
```


To answer our second question:

What is the Max, Min, and Average pump PSI?

We use the same data window, and then collect the PSI data on each sensor RDD.

Streaming Application Output



The output from our application gives us the requested PSI data for each pump.

Class Discussion

Class Discussion

What are some scenarios where you would find it useful to use windowed operations?

Learning Goals

Learning Goals

- Describe Spark Streaming Architecture
- Create DStreams and a Spark Streaming Application
- Apply Operations on DStreams
- Define Windowed Operations
- **Describe How Streaming Applications are Fault Tolerant**

This sections describes what makes Spark streaming applications fault tolerant.

Fault Tolerance in Spark RDDs

- RDD is immutable
- Each RDD remembers lineage
- If RDD partition is lost due to worker node failure, partition recomputed
- Data in final transformed RDD always the same
- Data comes from fault-tolerant systems → RDDs from fault-tolerant data are also fault tolerant

As a review, an RDD is an immutable, distributed dataset that is deterministically re-computable.

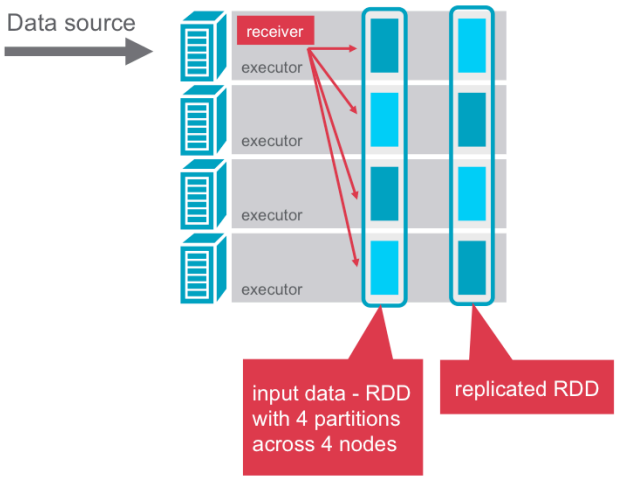
Each RDD remembers it's lineage of operations.

If any RDD partition is lost due to worker node failure, the partition can be recomputed from the original data using the lineage. The data in the final transformed RDD is always the same provided the RDD transformation are deterministic.

Since RDDs are operating on data from fault tolerant systems such as HDFS, S3 and MapR, the RDDs generated from fault tolerant data are also fault tolerant.

This is not the case for Spark Streaming, however, since data is received over the network, except when using fileStream. Next we will see how Spark achieves the same fault-tolerance with streaming.

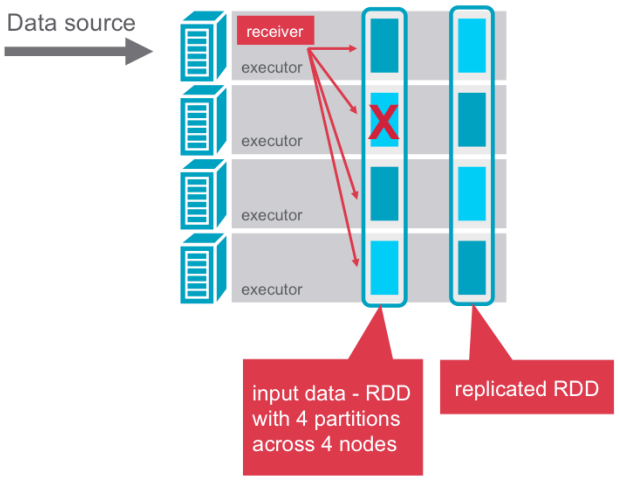
Fault Tolerance in Spark Streaming



Spark Streaming launches receivers within an executor for each input source. The receiver receives input data that is saved as RDDs and then replicated to other executors for fault tolerance. The default replication factor is 2.

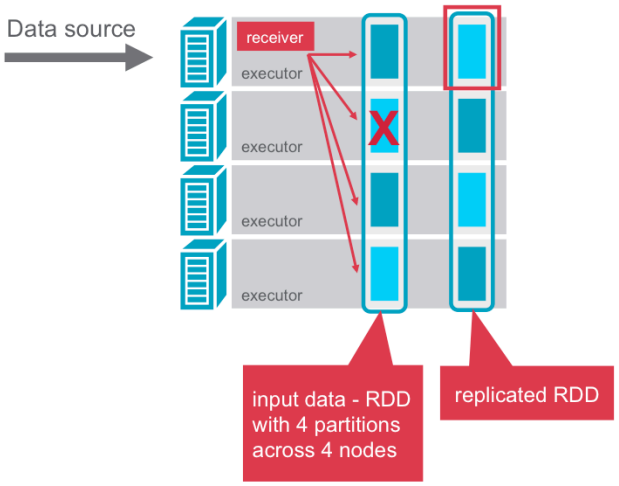
The data is stored in memory of the executors as cached RDDs. In this example, the input data source is partitioned by the Spark receiver into a DStream of four RDDs and the RDD partitions are replicated to different executors.

Fault Tolerance in Spark Streaming



When receiving data from network-based sources, you can have a worker node fail.

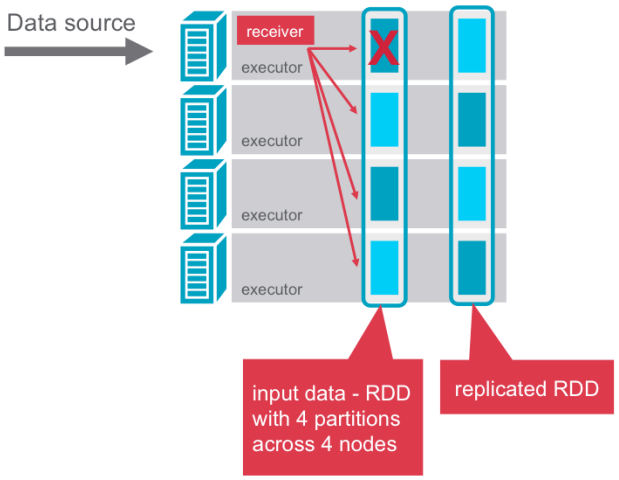
Fault Tolerance in Spark Streaming



Since the data is replicated in memory on the worker node, in the event of failure of a single worker node, the data exists on another node.

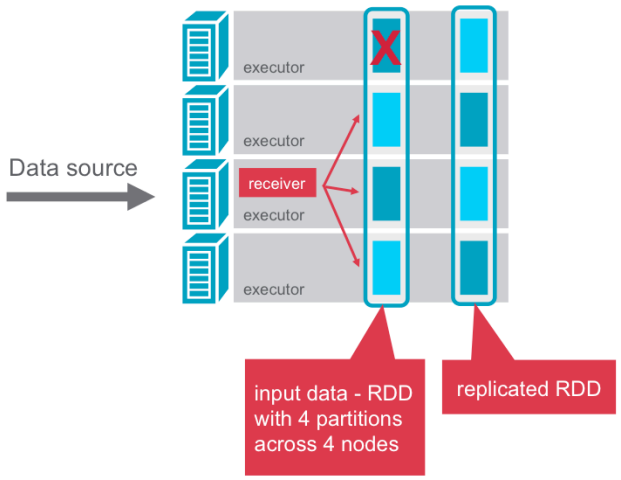
In this example, if the second node fails, there is a replica of the data on node 1.

Fault Tolerance in Spark Streaming



If instead the node with the receiver fails, then there may be a loss of the data that was received by the system but not yet replicated to other nodes.

Fault Tolerance in Spark Streaming



In this case the receiver will be started on a different node and it will continue to receive data.

Fault Tolerance in Spark RDDs – Write Ahead Logs

- Flume, MapR Streams, and Kafka use receivers
- Store received data in executor memory
- Driver runs tasks on executors

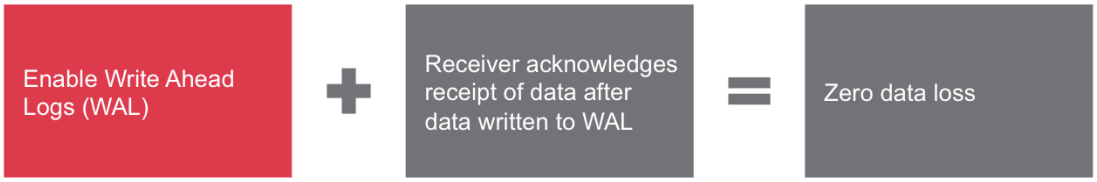
Spark also supports write ahead logs for Spark Streaming, to improve the recovery mechanism.

Sources like Flume, MapR Streams and Kafka use receivers to receive data.

The received data is stored in the executor memory, and the driver runs tasks on executors.

Fault Tolerance in Spark RDDs – Write Ahead Logs

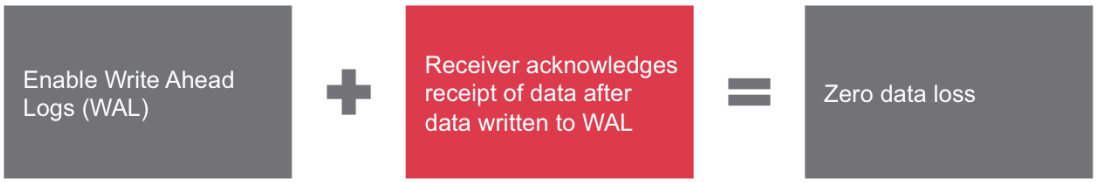
- Flume, MapR Streams, and Kafka use receivers
- Store received data in executor memory
- Driver runs tasks on executors



When write ahead logs is enabled, all of the received data is saved to log files in a fault tolerant system. This makes the received data durable in the event of any failure in Spark Streaming.

Fault Tolerance in Spark RDDs – Write Ahead Logs

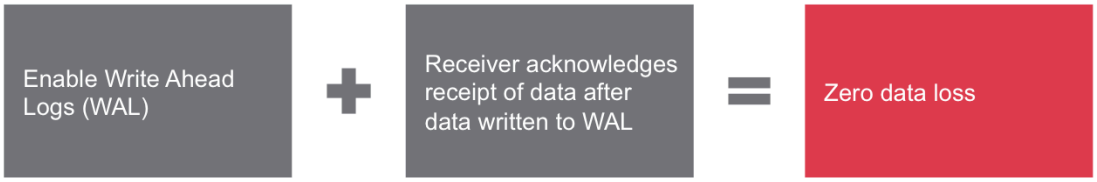
- Flume, MapR Streams, and Kafka use receivers
- Store received data in executor memory
- Driver runs tasks on executors



Furthermore, if the receiver acknowledges the receipt of the data after it has been written to the WAL, then in the event of a failure, the data that is buffered but not saved can be resent by the source once the driver is restarted.

Fault Tolerance in Spark RDDs – Write Ahead Logs

- Flume, MapR Streams, and Kafka use receivers
- Store received data in executor memory
- Driver runs tasks on executors



These two factors provide zero data loss.

Refer to the Spark documentation on how to enable and configure write ahead logs for Spark Streaming.

Checkpointing

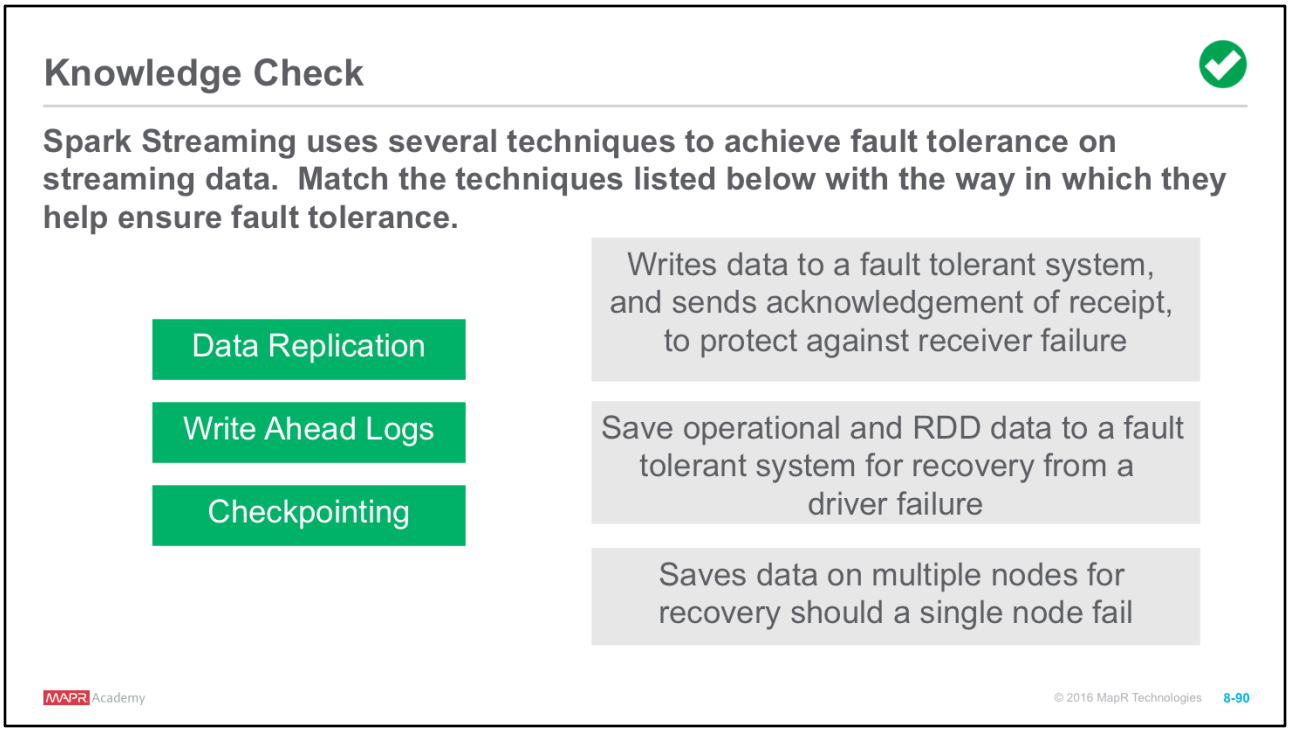
- Provides fault tolerance for driver
- Periodically saves data to fault tolerant system
- Two types of checkpointing:
  - Metadata – for recovery from driver failures
  - Data checkpointing – for basic functioning if using stateful transformations
- Enable checkpointing
  - `ssc.checkpoint("hdfs://...")`

For a streaming application to be available 24/7, and be resilient to failures, Spark Streaming needs to checkpoint enough information to a fault-tolerant storage system for recovery.

There are two types of checkpointing- metadata checkpointing and data checkpointing.
Metadata checkpointing is for recovery from driver failures.
Data checkpointing is needed for basic functioning if using stateful transformations.

You can enable checkpointing by calling the checkpoint method on the StreamingContext.
If you use sliding windows, you need to enable checkpointing.

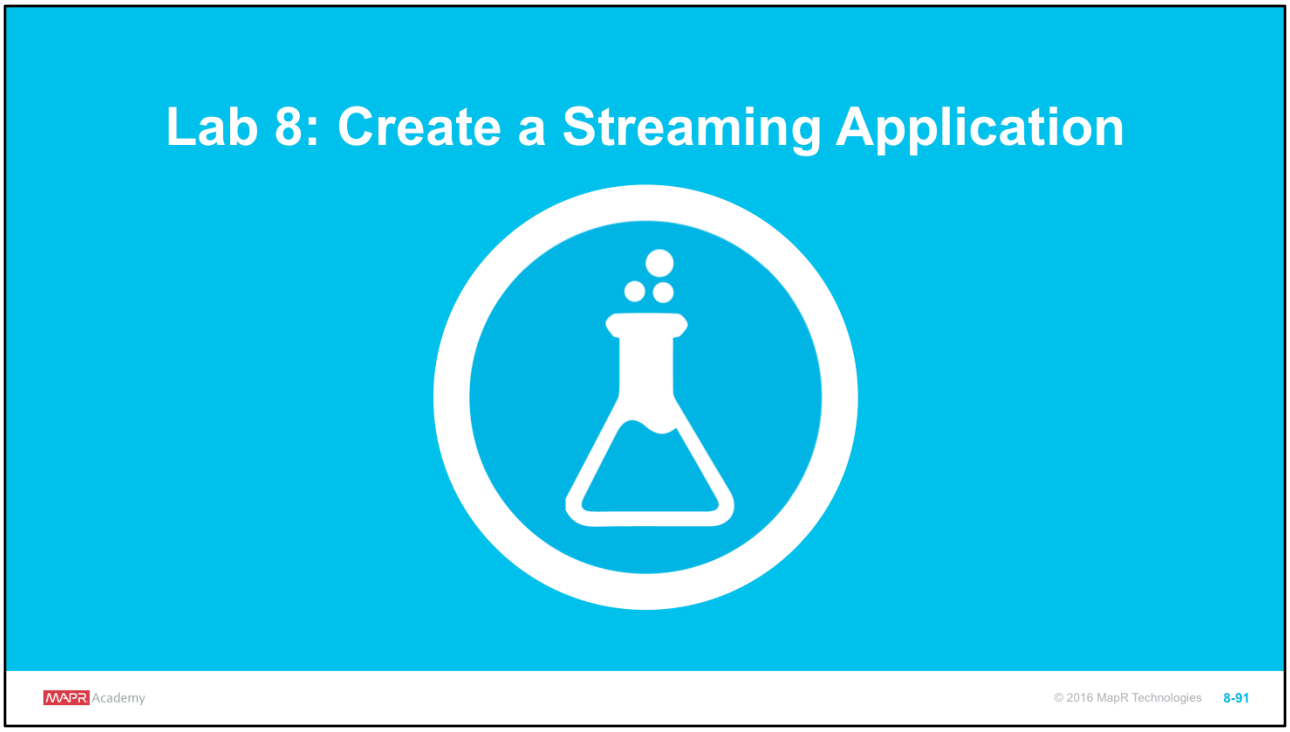
Knowledge Check



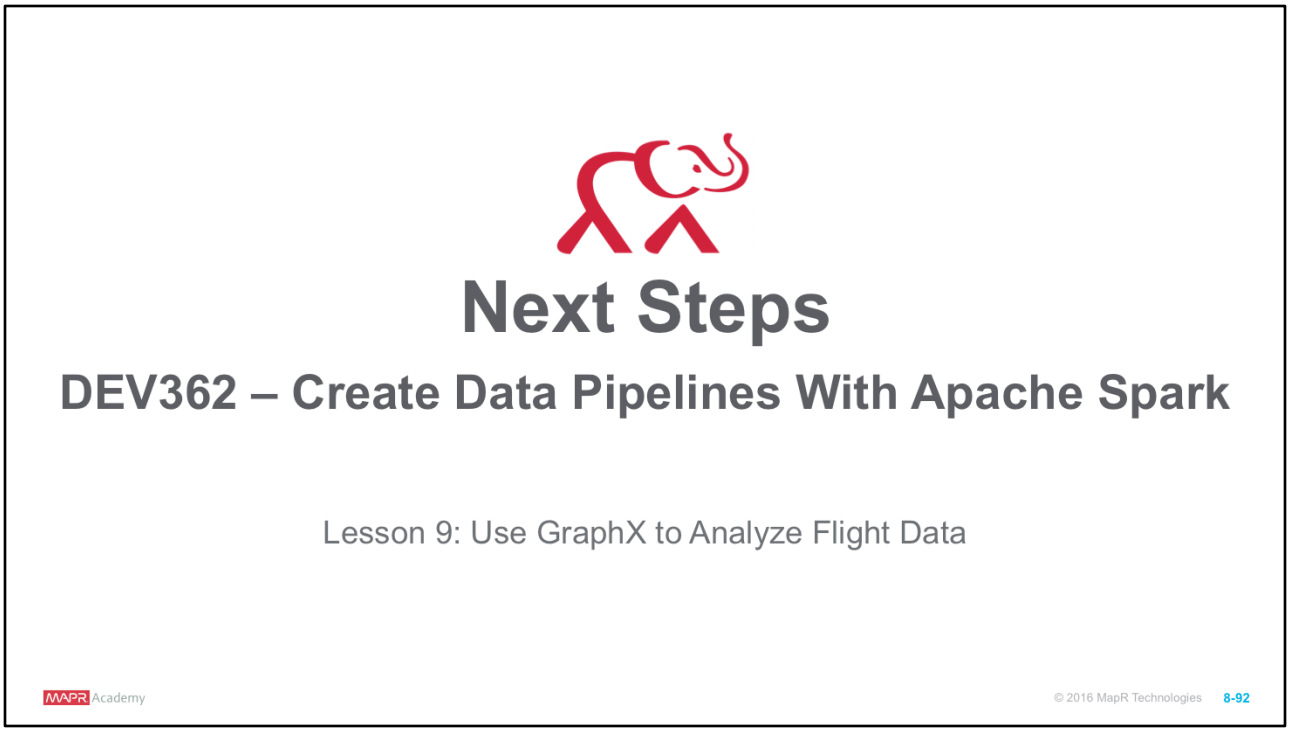
Data Replication - Saves data on multiple nodes for recovery should a single node fail

Write Ahead Logs - Writes data to a fault tolerant system, and sends acknowledgement of receipt, to protect against receiver failure

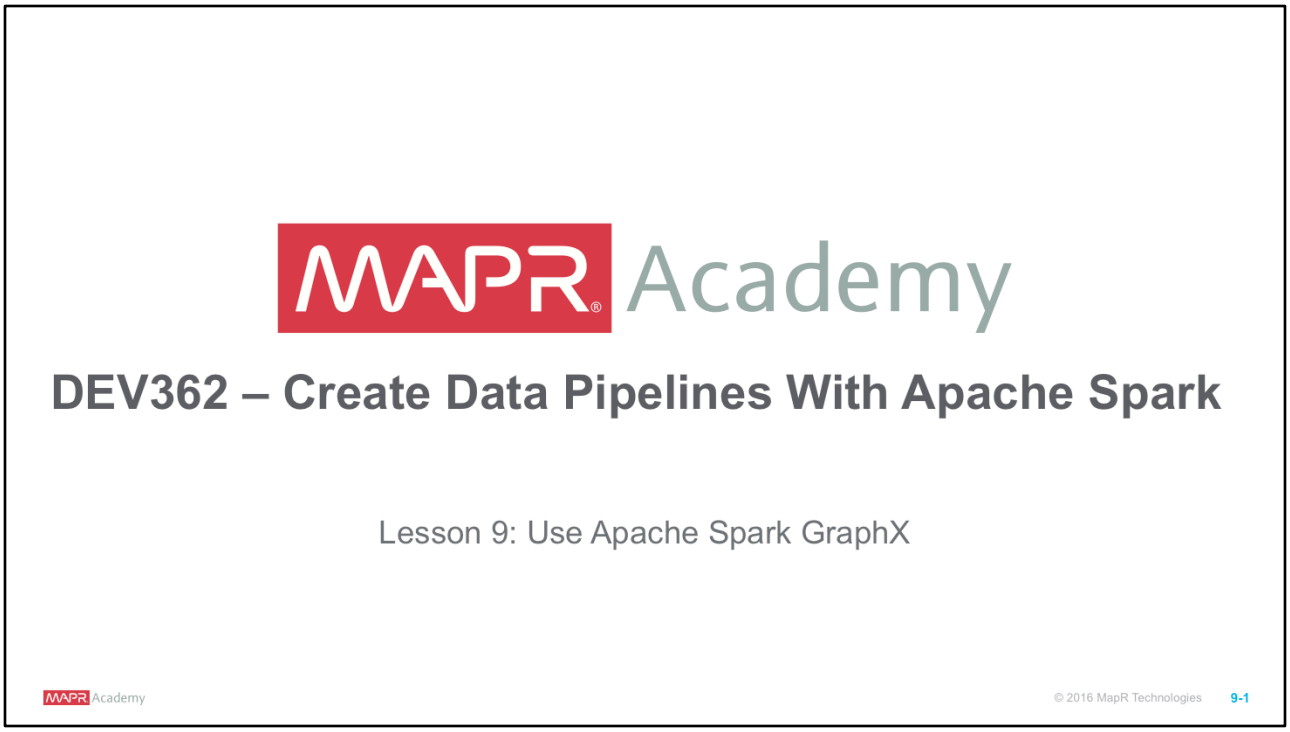
Checkpointing - Save operational and RDD data to a fault tolerant system for recovery from a driver failure



In these activities, you will create a Spark Streaming application. There is also a follow up lab where you save Spark Streaming data to an HBase table.



Congratulations! You have completed Lesson 8.



Welcome to DEV 362 lesson 9, Use Apache Spark GraphX.