CSE 424

Big Data

Apache Spark, Spark Streaming and SparkSQL

Slides 7

Instructor: Asst. Prof. Dr. Hüseyin ABACI

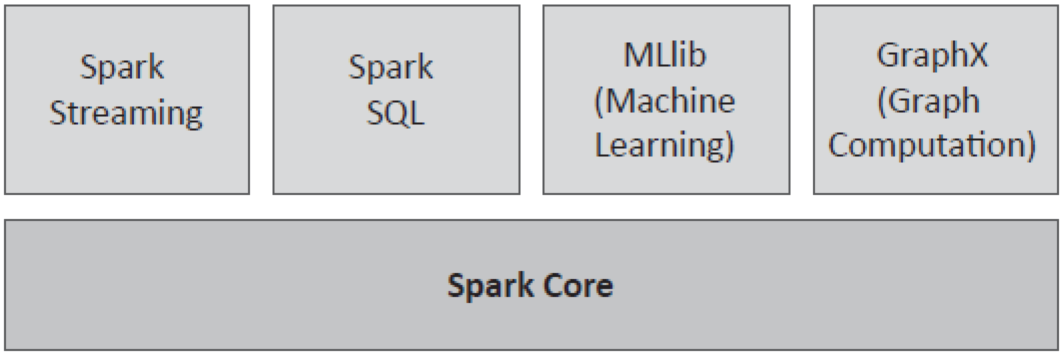
Outline

- Apache Spark
- Spark Cluster
- Running Spark
- RDD(Resilient Distributed Datasets)
- Spark Transformations
- Spark Actions
- Spark Streaming
- Window Operations
- SparkSQL

Bahga, Arshdeep, and Vijay Madisetti. Big data science & analytics: A hands-on approach. VPT, 2016.
Karau, Holden, et al. *Learning spark: lightning-fast big data analysis*. " O'Reilly Media, Inc.", 2015.

Apache Spark

- Apache Spark is an open source cluster computing framework for data analytics.
- Spark supports in-memory cluster computing and promises to be faster than Hadoop.
- Spark supports various high-level tools for data analysis such as Spark Streaming for streaming jobs, Spark SQL for analysis of structured data, MLlib machine learning library for Spark, and GraphX for graph processing.
- Spark allows real-time, batch and interactive queries and provides APIs for Scala, Java and Python languages.

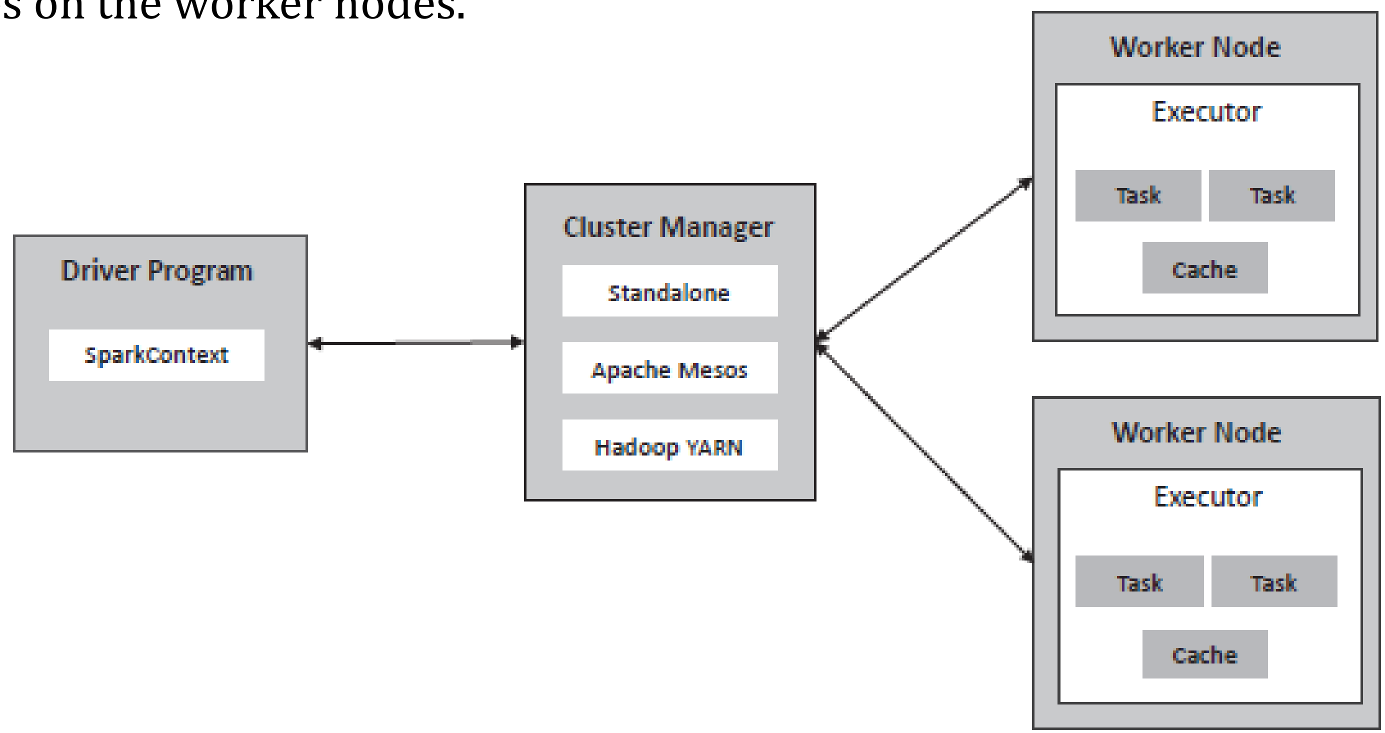


Apache Spark (cont'd)

- **Spark Core:** Spark Core provides common functionality (such as task scheduling and input/output), which is used by other Spark components.
  - Spark provides a data abstraction called resilient distributed dataset (RDD) which is a collection of elements partitioned across the nodes in a Spark cluster. The RDD elements can be operated on in parallel in the cluster. RDDs are immutable and distributed collection of objects.
- **Spark Streaming:** Spark Streaming is a Spark component for analysis of streaming data such as sensor data, click stream data, web server logs, etc.
- **Spark SQL:** Spark SQL is a Spark component that enables interactive querying of data using SQL queries.
- **Spark MLlib:** Spark MLlib is Spark's machine learning library that provides implementations of commonly used machine learning algorithms for clustering, classification, regression, collaborative filtering and dimensionality reduction.
- **Spark GraphX:** Spark GraphX is a component for performing graph computations. GraphX provides implementations of common graph algorithms such as PageRank, connected components, and triangle counting.

Apache Spark - Cluster

- Each Spark application consists of a driver program and is coordinated by a `SparkContext` object.
- Spark supports various cluster managers including Spark's standalone cluster manager, Apache Mesos and Hadoop YARN.
- The cluster manager allocates resources for applications on the worker nodes. The executors which are allocated on the worker nodes run the application code as multiple tasks.
- Applications are isolated from each other and run within their own executor processes on the worker nodes.



Apache Spark – Running Spark (shell)

```
C:\Users\NETLAB>pyspark
Python 3.7.2 (default, Feb 21 2019, 17:35:59) [MSC v.1915 64 bit (AMD64)] :: Ana
conda, Inc. on win32

Warning:
This Python interpreter is in a conda environment, but the environment has
not been activated. Libraries may fail to load.  To activate this environment
please see https://conda.io/activation

Type "help", "copyright", "credits" or "license" for more information.
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLeve
l(newLevel).
19/11/13 10:29:47 WARN NativeCodeLoader: Unable to load native-hadoop library fo
r your platform... using builtin-java classes where applicable
19/11/13 10:29:54 WARN ObjectStore: Failed to get database global_temp, returnin
g NoSuchObjectException
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 2.2.0
      /_/

Using Python version 3.7.2 (default, Feb 21 2019 17:35:59)
SparkSession available as 'spark'.
>>> sc.version
'2.2.0'
>>> help()

Welcome to Python 3.7's help utility!

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at https://docs.python.org/3.7/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules.  To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, symbols, or topics, type
"modules", "keywords", "symbols", or "topics".  Each module also comes
with a one-line summary of what it does; to list the modules whose name
or summary contain a given string such as "spam", type "modules spam".

help>
```

After you installed the spark and related binaries, write *"pyspark"* to cmd shell to see the prompt, in order to quit press *control+Z* or simple close the window.

Running Spark (Jupyter Notebook)

- Import necessary Spark classes into your program from pyspark library.
- Spark program must do is to create a ***SparkContext*** object.
- To create a SparkContext, there is need to build a ***SparkConf*** object that contains information about your application.
- We use Spark's readme file ("README.md" in jupyter notebook's work space) for initial observations.

```
In [1]: from pyspark import SparkConf
        from pyspark import SparkContext

In [2]: conf=SparkConf().setAppName("SparkOps")
        sc= SparkContext (conf=conf)

In [26]: rdd=sc.textFile("README.md")

In [27]: rdd.count()

Out[27]: 105
```

Apache Spark – README.md

```
# Apache Spark

Spark is a fast and general cluster computing system for Big Data. It
provides
high-level APIs in Scala, Java, Python, and R, and an optimized engine
that
supports general computation graphs for data analysis. It also supports a
rich set of higher-level tools including Spark SQL for SQL and
DataFrames,
MLlib for machine learning, GraphX for graph processing,
and Spark Streaming for stream processing.

<http://spark.apache.org/>


## Online Documentation

You can find the latest Spark documentation, including a programming
```

Apache Spark - RDD

- A Resilient Distributed Dataset (RDD) is the most fundamental data object used in Spark programming. RDDs are datasets within a Spark application, including the initial dataset(s) loaded, any intermediate dataset(s), and the final resultant dataset(s).
- Most Spark applications load an RDD (immutable) with external data and then create new RDDs by performing operations on the existing RDDs; these operations are ***transformations***.
- This process is repeated until an output (return a value to the driver program) operation is ultimately required—for instance, to write the results of an application to a filesystem; these types of operations are ***actions***.
- In the case of PySpark, RDDs consist of distributed Python objects, such as lists, tuples, and dictionaries.
- Although there are options for persisting RDDs to disk, RDDs are predominantly stored in memory, or at least they are intended to be stored in memory.
- RDDs can be created either by **parallelizing** existing collections or by loading an external dataset as shown in box below:

```
■ #Create RDD from a local file
lines = sc.textFile("file:///root/spark/README.md")

#Create RDD by parallelizing an existing collection
data = sc.parallelize([1, 2, 2, 3, 3, 4, 5])
```

Apache Spark - Transformations

- Let us look at some commonly used transformations (it shapes your dataset) with examples. For the examples, we will use the three datasets as shown below:

```
lines = sc.textFile("file:///root/spark/README.md")

lines.take(3)
[u'# Apache Spark', u'', u'Spark is a fast and general
cluster computing system for Big Data. It provides']

data1 = sc.parallelize([1, 2, 2, 3, 3, 4, 5])
data2 = sc.parallelize([3, 4, 5, 6, 7, 8])
```

- The *map* transformation takes as input a function which is applied to each element of the dataset and maps each input item to another item.

```
#map transformation example
lineLengths = lines.map(lambda s: len(s))
lineLengths.take(5)
[14, 0, 78, 72, 73]
```

- The *filter* transformation generates a new dataset by filtering the source dataset using the specified function.

```
#filter transformation example
filteredLines = lines.filter(lambda line: line.find('Spark')>0)
filteredLines.take(3)
[u'# Apache Spark', u'rich set of higher-level tools including Spark SQL
for SQL and structured', u'and Spark Streaming for stream processing.']
```

Apache Spark - Transformations

- The *reduceByKey* transformation when applied on dataset containing key-value pairs, aggregates values of each key using the function specified.

```
# reduceByKey transformation example
splitLines = lines.flatMap(lambda line: line.split())
```

```
In [28]: splitLines = lines.flatMap(lambda line: line.split())
```

```
In [39]: print(splitLines.take(10))

         ['#', 'Apache', 'Spark', 'Spark', 'is', 'a', 'fast', 'and', 'general', 'cluster']
```

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

```
In [30]: words=splitLines.map(lambda word: (word, 1))
```

```
In [40]: print(words.take(10))

         [('#', 1), ('Apache', 1), ('Spark', 1), ('Spark', 1), ('is', 1),
         1)]
```

Apache Spark - Transformations

```
counts=words.reduceByKey(lambda a, b: a+b)
counts.take(5)
```

```
In [32]: counts=words.reduceByKey(lambda a, b: a+b)
         counts.take(5)

Out[32]: [('#', 1), ('Apache', 1), ('Spark', 16), ('is', 7), ('It', 2)]
```

- The *union* transformation generates a new dataset from the union of two datasets.

```
#union transformation example
data = data1.union(data2)
data.collect()
[1, 2, 2, 3, 3, 4, 5, 3, 4, 5, 6, 7, 8]
```

- The *intersection* transformation generates a new dataset from the intersection of two datasets.

```
#intersection transformation example
data = data1.intersection(data2)
data.collect()
[4, 5, 3]
```

Apache Spark - Transformations

- The *join* transformation generates a new dataset by joining two datasets containing key-value pairs.

```
#join transformation example
a=sc.parallelize([('John', 1), ('Tom', 2), ('Ben', 3)])
b=sc.parallelize([('John', 'CA'), ('Tom', 'GA'), ('Ben', 'VA')])
c=a.join(b)
c.collect()
[('Ben', (3, 'VA')), ('John', (1, 'CA')), ('Tom', (2, 'GA'))]
```

- The *flatMap* transformation takes as input a function which is applied to each element of the dataset. The flatMap transformation can map each input item to zero or more output items.

```
#flatMap transformation example
splitLines = lines.flatMap(lambda line: line.split())
splitLines.take(10)
[u'#', u'Apache', u'Spark', u'Spark', u'is',
u'a', u'fast', u'and', u'general', u'cluster']
```

- Transformations are lazy and not computed till an action requires a result to be returned to the driver program. By computing transformations in a lazy manner, Spark is able to perform operations in a more efficient manner as the operations can be grouped together. Spark API allows chaining together transformations and actions.

Apache Spark – Transformations (Scala)

- Basic RDD transformations on an RDD containing {1, 2, 3, 3}

Function name	Purpose	Example	Result
`map()`	Apply a function to each element in the RDD and return an RDD of the result.	`rdd.map(x => x + 1)`	`{2, 3, 4, 4}`
`flatMap()`	Apply a function to each element in the RDD and return an RDD of the contents of the iterators returned. Often used to extract words.	`rdd.flatMap(x => x.to(3))`	`{1, 2, 3, 2, 3, 3, 3}`
`filter()`	Return an RDD consisting of only elements that pass the condition passed to `filter()`.	`rdd.filter(x => x != 1)`	`{2, 3, 3}`
`distinct()`	Remove duplicates.	`rdd.distinct()`	`{1, 2, 3}`
`sample(withReplacement, fraction, [seed])`	Sample an RDD, with or without replacement.	`rdd.sample(false, 0.5)`	Nondeterministic

Apache Spark – Transformations (Scala)

- Two-RDD transformations on RDDs containing {1, 2, 3} and {3, 4, 5}

Function name	Purpose	Example	Result
`union()`	Produce an RDD containing elements from both RDDs.	`rdd.union(other)`	`{1, 2, 3, 3, 4, 5}`
`intersection()`	RDD containing only elements found in both RDDs.	`rdd.intersection(other)`	`{3}`
`subtract()`	Remove the contents of one RDD (e.g., remove training data).	`rdd.subtract(other)`	`{1, 2}`
`cartesian()`	Cartesian product with the other RDD.	`rdd.cartesian(other)`	`{(1, 3), (1, 4), … (3,5)}`

Apache Spark - Actions

- Let us look at some commonly used actions with examples:
- The *reduce* action aggregates the elements in a dataset using the specified function.

```
#reduce action example
lineLengths = lines.map(lambda s: len(s))
totalLength = lineLengths.reduce(lambda a, b: a + b)
3526
```

- The *collect* action is used to return all the elements of the result as an array.

```
#collect action example
lineLengths = lines.map(lambda s: len(s))
lineLengths.collect()
[14, 0, 78, 72, 73, ... , 70]
```

- The *count* action returns the number of elements in a dataset.

```
#count action example
lines.count()
98
```

- The *first* action returns the first element in a dataset.

```
#first action example
lines.first()
u'# Apache Spark'
```

Apache Spark - Actions

- The *take* action returns the first n elements in a dataset.

```
#take action example
lines.take(3)
[u'# Apache Spark', u'', u'Spark is a fast and general
cluster computing system for Big Data. It provides']
```

- The *saveAsTextFile* action writes the elements in a dataset to a text file either on the local filesystem or HDFS.

```
#saveAsTextFile action example
lines.saveAsTextFile('/path/to/file')
```

Apache Spark – Actions (Scala)

- Basic actions on an RDD containing {1, 2, 3, 3}

Function name	Purpose	Example	Result
`collect()`	Return all elements from the RDD.	`rdd.collect()`	`{1, 2, 3, 3}`
`count()`	Number of elements in the RDD.	`rdd.count()`	`4`
`countByValue()`	Number of times each element	`rdd.countByValue()`	`{(1, 1), (2, 1),`
`take(num)`	Return `num` elements from the RDD.	`rdd.take(2)`	`{1, 2}`
`top(num)`	Return the top `num` elements the RDD.	`rdd.top(2)`	`{3, 3}`
`takeOrdered(num)(ordering)`	Return `num` elements based on provided ordering.	`rdd.takeOrdered(2)(myOrdering)`	`{3, 3}`

Apache Spark – Actions (Scala)

- Basic actions on an RDD containing {1, 2, 3, 3}

`takeSample(withReplacement, num, [seed])`	Return `num` elements at random.	`rdd.takeSample(false, 1)`	Nondeterministic
`reduce(func)`	Combine the elements of the RDD together in parallel (e.g., `sum`).	`rdd.reduce((x, y) => x + y)`	9
`fold(zero)(func)`	Same as `reduce()` but with the provided zero value.	`rdd.fold(0)((x, y) => x + y)`	9
`aggregate(zeroValue)(seqOp, combOp)`	Similar to `reduce()` but used to return a different type.	`rdd.aggregate((0, 0))((x, y) => (x._1 + y, x._2 + 1), (x, y) => (x._1 + y._1, x._2 + y._2))`	`(9, 4)`
`foreach(func)`	Apply the provided function to each element of the RDD.	`rdd.foreach(func)`	Nothing

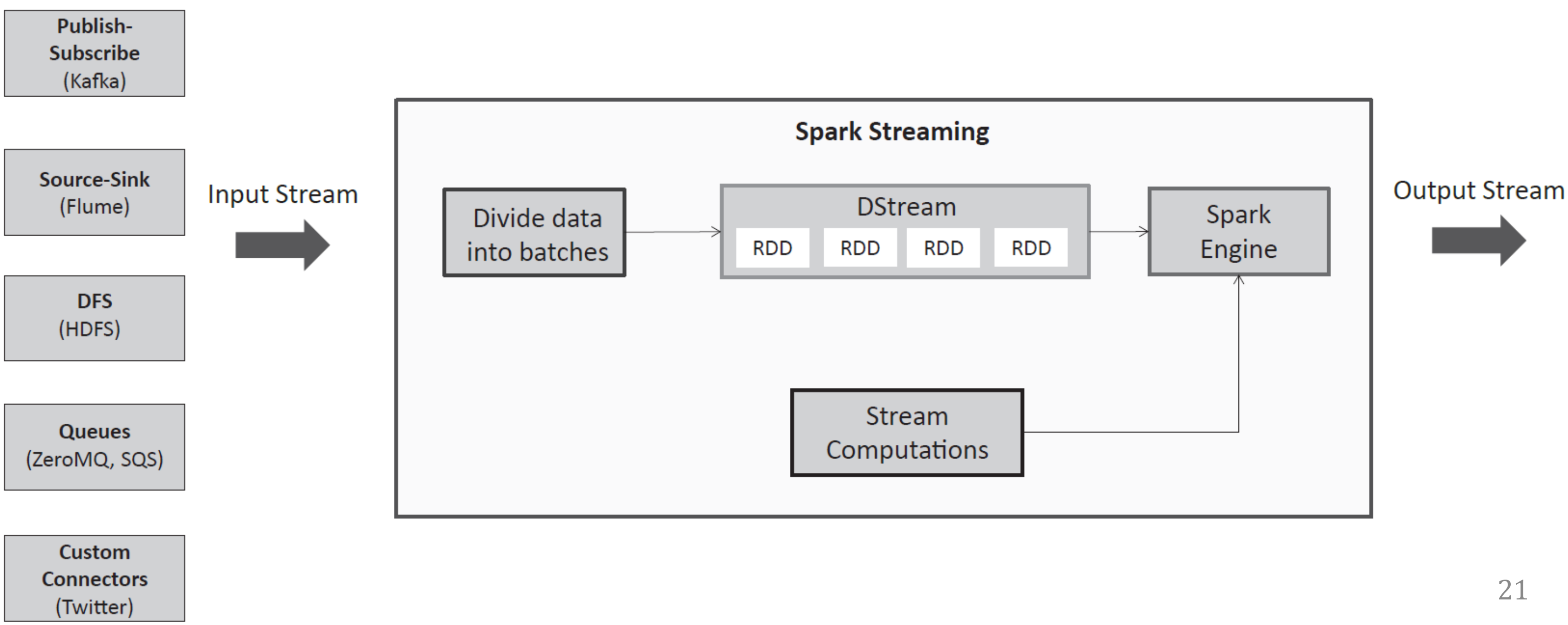
Apache Spark - Actions

- Let us now look at a standalone Spark application that computes word counts in a file.
- The following program uses the map and reduce functions. The flatMap and map transformation take as input a function which is applied to each element of the dataset. While the flatMap function can map each input item to zero or more output items, the map function maps each input item to another item.
- The transformations take as input, functions which are applied to the data elements. The input functions can be in the form of Python lambda expressions or local functions. In the word count example flatMap takes as input a lambda expression that splits each line of the file into words. The map transformation outputs key value pairs where the key is a word and value is 1.
- The reduceByKey transformation aggregates values of each key using the function specified (add function in this example). Finally, the collect action is used to return all the elements of the result as an array.

```
from operator import add
from pyspark import SparkContext

sc = SparkContext(appName="WordCountApp")
lines = sc.textFile("file.txt")
counts = lines.flatMap(lambda x: x.split(‘ ’)).map(lambda x: (x, 1)).reduceByKey(add)

output = counts.collect()

for (word, count) in output:
  print "%s: %i" % (word, count)
```

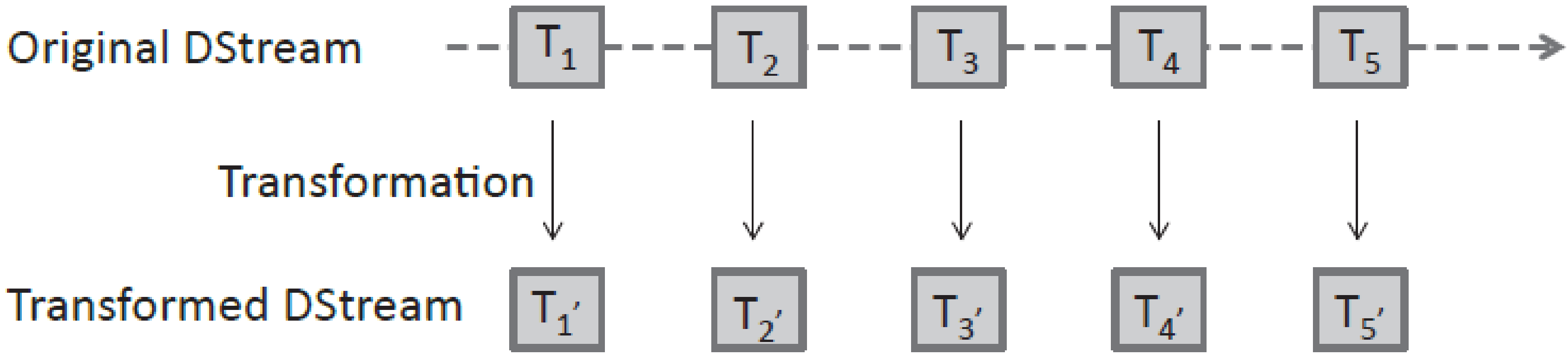
Apache Spark – Spark Streaming

- The streaming data is ingested and analyzed in micro-batches. Spark Streaming enables scalable, high throughput and fault-tolerant stream processing. Spark Streaming provides a high-level abstraction called DStream (discretized stream).
- Dstream is a sequence of RDDs. Spark can ingest data from various types of data sources such as publish-subscribe messaging frameworks, messaging queues, distributed file systems and custom connectors. The data ingested is converted into DStreams. Figure shows the Spark Streaming components.



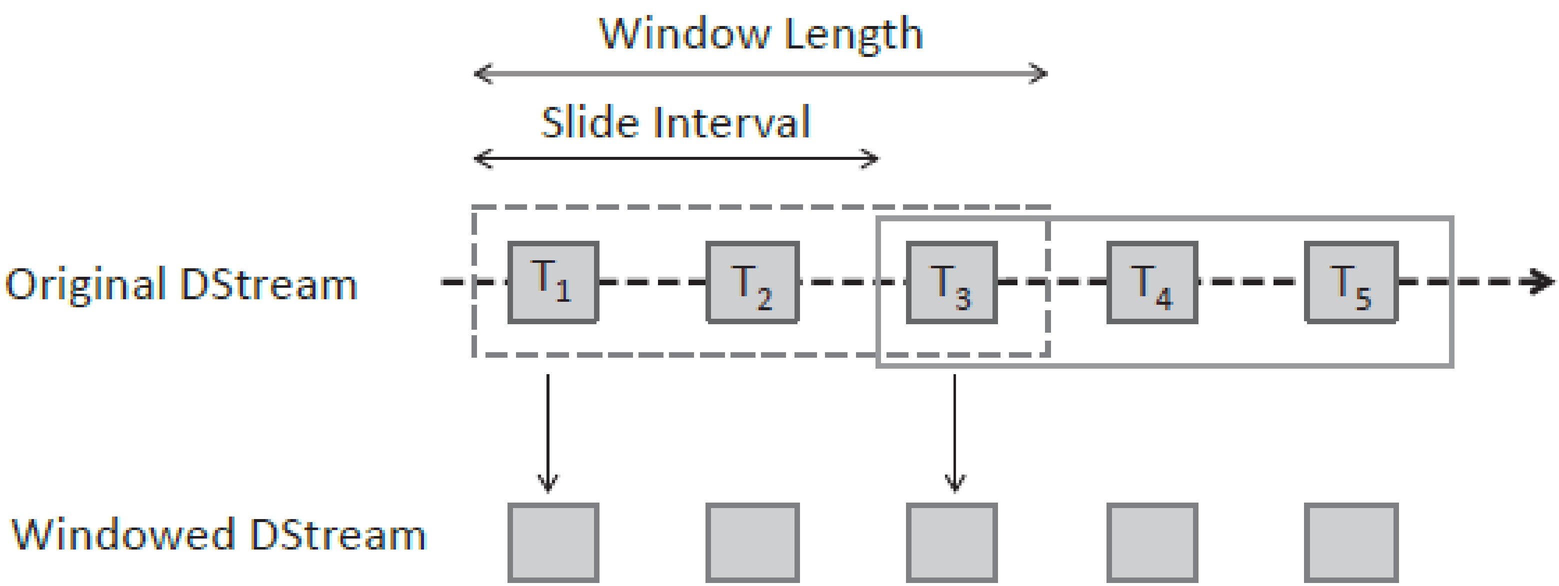
Apache Spark – Spark Streaming

- Figure shows a DStream, which is composed of RDDs, where each RDD contains data from a certain time interval. The DStream operations are translated into operations on the underlying RDDs. DStream transformations such as map, flatMap, filter, reduceByKey are **stateless** as the transformation are applied to the RDDs in the Dstream separately.



Apache Spark – Spark Streaming

- Spark also supports **stateful** operations such as windowed operations and updateStateByKey operation. Stateful operations require checkpointing for fault tolerance purposes. For stateful operations, a checkpoint directory is provided to which RDDs are checkpointed periodically.
- Window operations allow the computations to be done over a sliding window of data. For window operations, a window length and a slide interval in specified. In the Figure the window length is 3 and slide interval is 2.



Apache Spark – Window Operations

- Let us look at some commonly used window operations:
- The *window* operation returns a new DStream from a sliding window over the source DStream.

```
#Format: window(windowLength, slideInterval)
# Example: Return a new DStream with RDDs containing last
# 8 seconds of data, every 4 seconds
windowStream = sourceStream.window(8,4)
```

- The *countByWindow* operation counts the number of elements in a window over the DStream.

```
#Format: countByWindow(windowDuration, slideDuration)
# Example: Count the number of elements in a sliding window with
# window duration of 10 and slide interval of 4 seconds
count = sourceStream.countByWindow(10,4)
```

- The *reduceByWindow* operation aggregates the elements in a sliding window over a stream using the specified function.

```
#Format: reduceByWindow(func, windowLength, slideInterval)
# Example: In a text data stream compute the running line lenghts
# with window duration of 10 and slide interval of 4 seconds
totalLength = lineLengths.reduceByWindow(lambda a, b: a + b, 10, 4)
```

Apache Spark – Spark SQL

- Spark SQL is a component of Spark which enables interactive querying. Spark SQL can interactively query structured and semi-structured data using SQL-like queries.
- Spark SQL provides a programming abstraction called DataFrames.
- A DataFrame is a distributed collection of data organized into named columns. DataFrames can be created from existing RDDs, structured data files (such as text files, Parquet, JSON, Apache Avro), Hive tables and also from external databases. Spark provides a Data Sources API, which allows accessing structured data though Spark SQL.

- Spark SQL provides an SQLContext, which is the entry point for Spark SQL. SQLContext provides functionality for creating a DataFrame, registering DataFrame as a table and executing SQL statements over a table. SQLContext can be created from a SparkContext as shown in the box below.

```
from pyspark.sql import SQLContext, Row
sqlContext = SQLContext(sc)
from pyspark.sql.types import *
```

pyspark.sql module - https://spark.apache.org/docs/2.4.0/api/python/pyspark.sql.html

Apache Spark – Spark SQL

- We will use the Google N-Gram dataset [1] in this example. The dataset file is in CSV format and contains data on bigrams with the following columns :

  Bigram (2-Gram), Year, Count, Pages, Books

! $17.95	1996	4	2	2
! $17.95	1997	6	4	4
! $17.95	1998	3	2	1
! $17.95	1999	8	8	8
! $17.95	2000	10	8	8
! $17.95	2001	2	2	2
! $17.95	2002	3	3	3
! $17.95	2003	1	1	1
! $17.95	2004	14	14	14
! $17.95	2005	12	12	12
! $17.95	2006	4	4	4
! $17.95	2007	2	2	2
! $17.95	2008	1	1	1
! 09	1807	1	1	1
! 09	1810	1	1	1
! 09	1817	1	1	1

[1] http://storage.googleapis.com/books/ngrams/books/datasetsv2.html
http://storage.googleapis.com/books/ngrams/books/googlebooks-eng-us-all-2gram-20090715-50.csv.zip

Apache Spark – Spark SQL

- In this example, an RDD is first created by loading the dataset file. The lines in the file are split to obtain the individual columns which are then converted into Row objects by passing the list of key-value pairs to the ***Row*** class.
- SQLContext provides a *createDataFrame* function to convert the RDD of *Row* objects to a DataFrame by inferring the data types.

```
lines = sc.textFile("file:///home/hadoop/
googlebooks-eng-us-all-2gram-20090715-50.csv")

parts = lines.map(lambda l: l.split(" "))

ngrams = parts.map(lambda x: Row(ngram=x[0], year=int(x[1]),
ngramcount=int(x[2]), pages=int(x[3]), books=int(x[4])))

schemaNGrams = sqlContext.createDataFrame(ngrams)
```

Apache Spark – Spark SQL

- To view the rows in the created DataFrame, the ***show*** function can be used which prints the first N rows to the console (default N=20).

```
■ »> schemaNGrams.show()

+---+----+-------+---+---+
|books| ngram|ngramcount|pages|year|
+---+----+-------+---+---+
|   1|   !  09|       1|   1|1829|
|   3|   !  09|       3|   3|1879|
|   2|   !  09|       2|   2|1911|
|   4|   !  09|       4|   4|1941|
|   4|   !  09|       4|   4|1969|
|  12|   !  09|      17|  17|1994|
|   1|!  13.5|       1|   1|1936|
|   1|!  1430|       1|   1|1861|
|   1|!  1430|       1|   1|1959|
|   2|!  16th|       3|   3|1854|
|   1|!  16th|       1|   1|1959|
|   2|!  1791|       2|   2|1856|
|   1|!  1791|       1|   1|1968|
|   1|!  1847|       1|   1|1859|
|   2|!  1847|       2|   2|1909|
|   2|!  1847|       2|   2|1962|
|   2|!  1944|       2|   2|1945|
|   2|!  1944|       8|   8|1977|
|   1|!  1944|       1|   1|2007|
|   2|!  23rd|       2|   2|1957|
+---+----+-------+---+---+
```

Apache Spark – Spark SQL

- To view the schema of the DataFrame the *printSchema* function can be used as shown below:

```
»> schemaNGrams.printSchema()
root
|- books: int (nullable = true)
|- ngram: string (nullable = true)
|- ngramcount: int (nullable = true)
|- pages: int (nullable = true)
|- year: int (nullable = true)
```

Apache Spark – Spark SQL

```
■ schemaNGrams.filter(schemaNGrams['ngramcount'] > 5).show()

+---+-----+-------+---+---+
|books| ngram|ngramcount|pages|year|
+---+-----+-------+---+---+
|  12|  !   09|   17|  17|1994|
| 2| !  1944|    8| 8|1977|
|  11|  !   28|   15|  15|1866|
|  10|  !   28|   10|  10|1891|
|  32|  !   28|   37|  37|1916|
|  14|  !   28|   14|  14|1941|
|  41|  !   28|   48|  47|1966|
|  57|  !   28|   76|  76|1991|
|  15|  !   56|   15|  15|1979|
|  54|  !   56|   61|  61|2004|
| 3| !   936|   16|  15|1943|
| 6| !   936|    9| 9|1973|
|  14| !   ANNE|   108|  95|1916|
| 4|  !  ANNE|   35|  26|1941|
| 6|  !  ANNE|   28|  26|1969|
| 6|   !   AS|    6| 6|1892|
| 6|   !   AS|    6| 6|1943|
| 6|   !   AS|    7| 7|1968|
|  10|  !   AS|   17|  15|1993|
|  17|!  Abort|   24|  21|2004|
+---+-----+-------+---+---+
```

- The box shows an example of the *filter* function which filters rows using the given condition. In this example, we filter all N-grams which have a count greater than five.

Apache Spark – Spark SQL

- The *groupBy* function can be used to group the DataFrame using the specified columns. Aggregations (such as avg, max, min, sum, count) can then be applied to the grouped DataFrame. The box below shows an example of grouping the N-Grams by year and then applying the count aggregations to find the total number of N-Grams in each year.

```
■ schemaNGrams.groupBy("year").count().show()

+---+---+
|year|count|
+---+---+
|1831|  79|
|1832|  57|
|1833|  56|
|1834|  47|
|1835|  71|
|1836|  66|
|1837|  74|
|1838|  56|
|1839|  63|
|1840|  66|
|1841|  71|
|1842|  54|
|1843|  87|
|1844|  81|
|1845|  91|
|1846|  95|
|1847|  72|
|1848|  76|
|1849| 101|
|1850| 104|
+---+---+
```

Apache Spark – Spark SQL

```
schemaNGrams.registerTempTable("ngrams")

result = sqlContext.sql("SELECT ngram, ngramcount
FROM ngrams WHERE ngramcount >= 5").show()
+----+--------+
| ngram|ngramcount|
+----+--------+
| !   09|   17|
|!   1944|    8|
| !   28|   15|
| !   28|   10|
| !   28|   37|
| !   28|   14|
| !   28|   48|
| !   28|   76|
| !   56|    5|
| !   56|    5|
| !   56|   15|
| !   56|   61|
|  !   936|   16|
|  !   936|    9|
|!   ANNE|    108|
|!   ANNE|   35|
|!   ANNE|   28|
| !   AS|    5|
| !   AS|    6|
| !   AS|    6|
+----+--------+
```

- Spark SQL allows registering a DataFrame as a temporary table for querying the data using SQL-like queries.
- With the created DataFrame (schemaNGrams) a temporary table (ngrams) is created using *registerTempTable* function.
- An SQL query for filtering all N-grams which have count greater than five is shown in box.

Apache Spark – Spark SQL

```
result = sqlContext.sql("SELECT year, COUNT(*) AS cnt FROM ngrams GROUP BY year ORDER BY cnt DESC").show()

+---+--+
|year|cnt|
+---+--+
|2007|470|
|2002|450|
|2000|447|
|2003|446|
|2006|445|
|2001|445|
|1997|441|
|2004|440|
|1988|437|
|1999|436|
|2005|435|
|1991|432|
|1998|421|
|1995|415|
|1996|410|
|1987|408|
|1994|402|
|1990|397|
|1978|390|
|1986|390|
+---+--+
```

- The box shows an example of an SQL query that uses the GROUP BY clause to group the N-Grams by the year column and COUNT statement to count the number of N-Grams in each year. The results are ordered by the count of N-Grams.