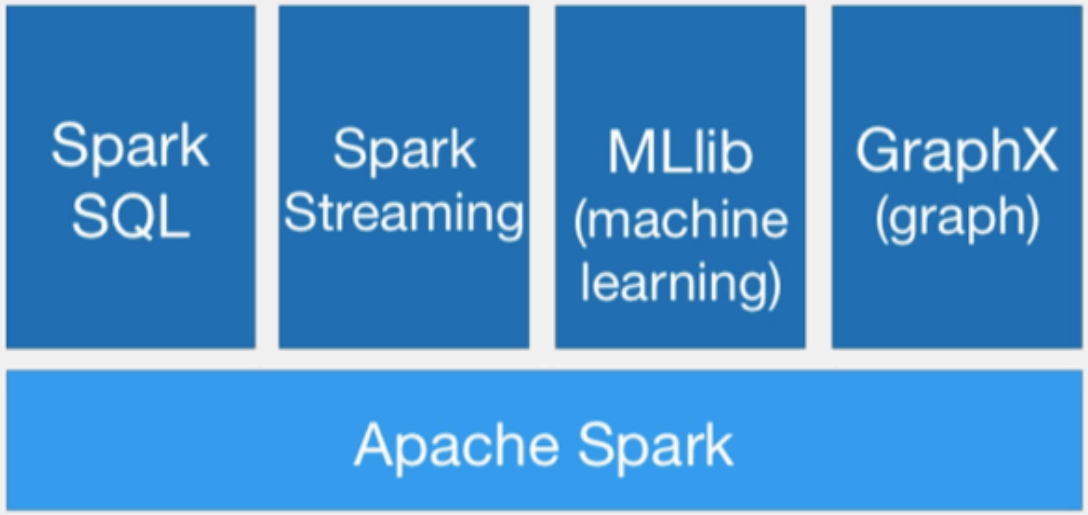
ТЕХНОСФЕРА

Лекция 13

Spark Streaming. Structured Streaming. Spark SQL. GraphX. MLLib.

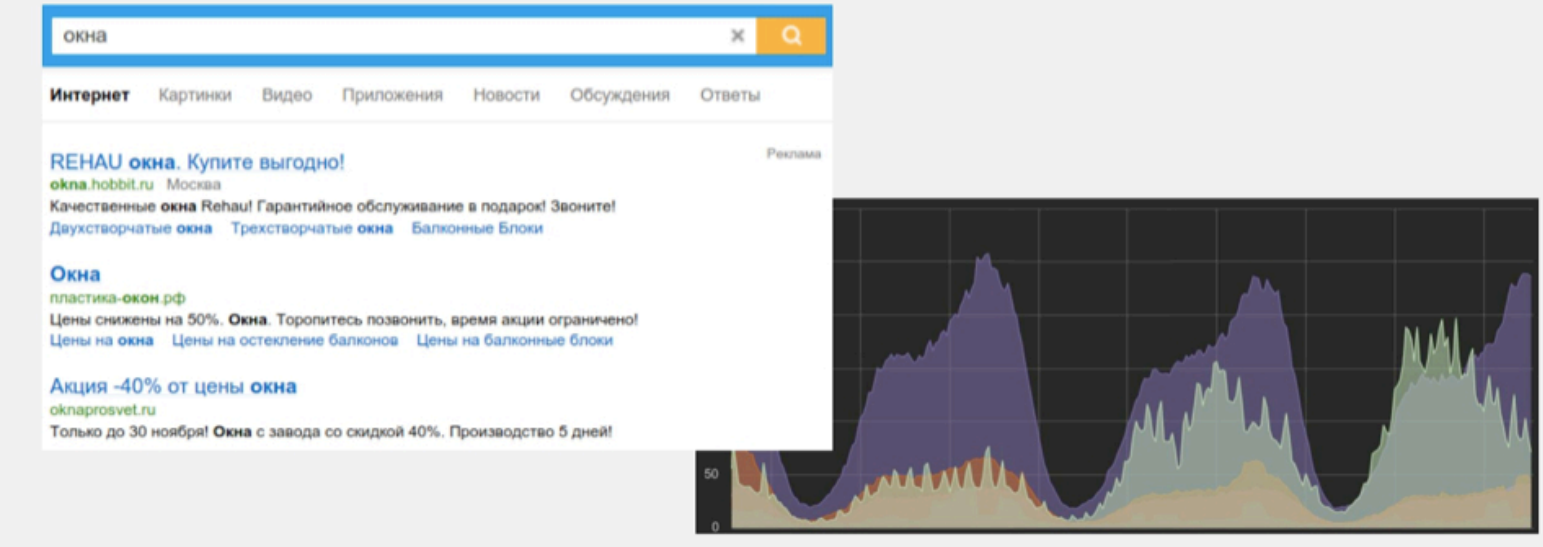
Смирнов Даниил

Apache Spark



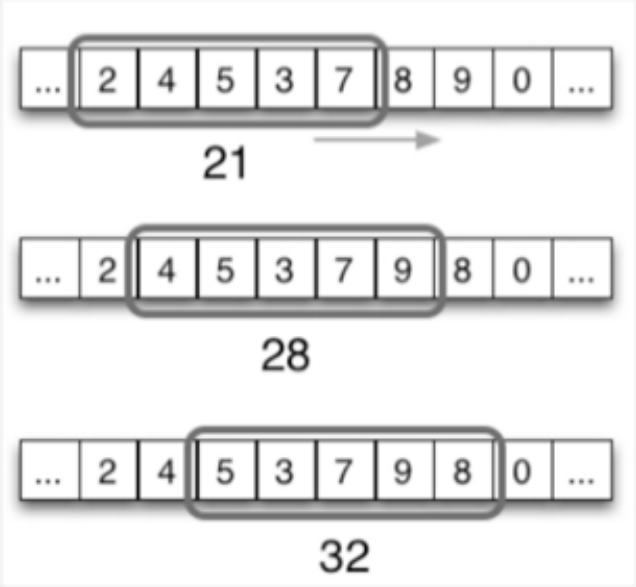
Batch vs Stream processing

- MapReduce и Spark хорошо подходят для пакетной обработки данных.
- Иногда бывает потребность обрабатывать бесконечные потоки данных.

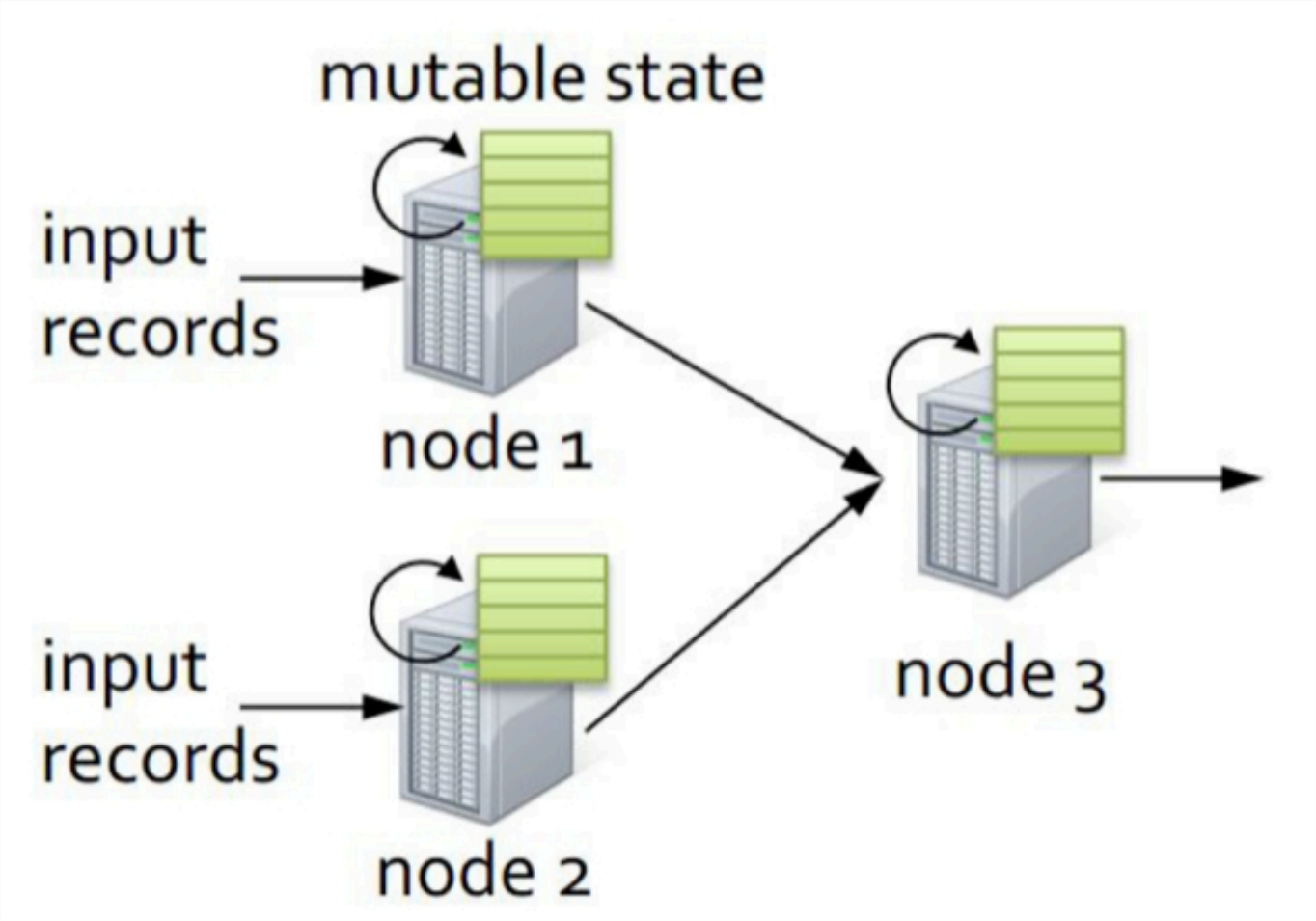


Stream processing используя batch

- Можно использовать метод скользящего окна
- Hadoop MapReduce плохо подходит для real time



Stateful processing



Spark Streaming

- Масштабируемый
- Подходит для real time обработки
- Встроенная fault tolerance
- Можно комбинировать batch & streaming

Micro batch модель

- Делим входной поток на фрагменты (1 сек)
- Рассматриваем каждый фрагмент как RDD
- Выход – поток обработанных RDD

- Используем общий код для обработки данных
- Легко организовать fault tolerance
- Можно комбинировать с batch обработкой

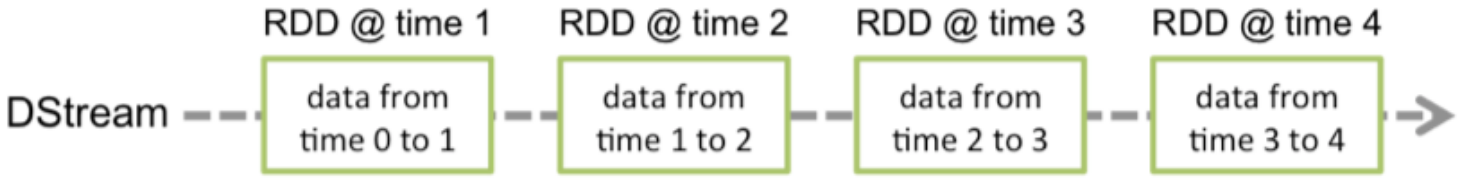
Discretized Streams (DStreams)

2 способа создать DStream :

- Из входного источника
- Применяя трансформации к другим DStream

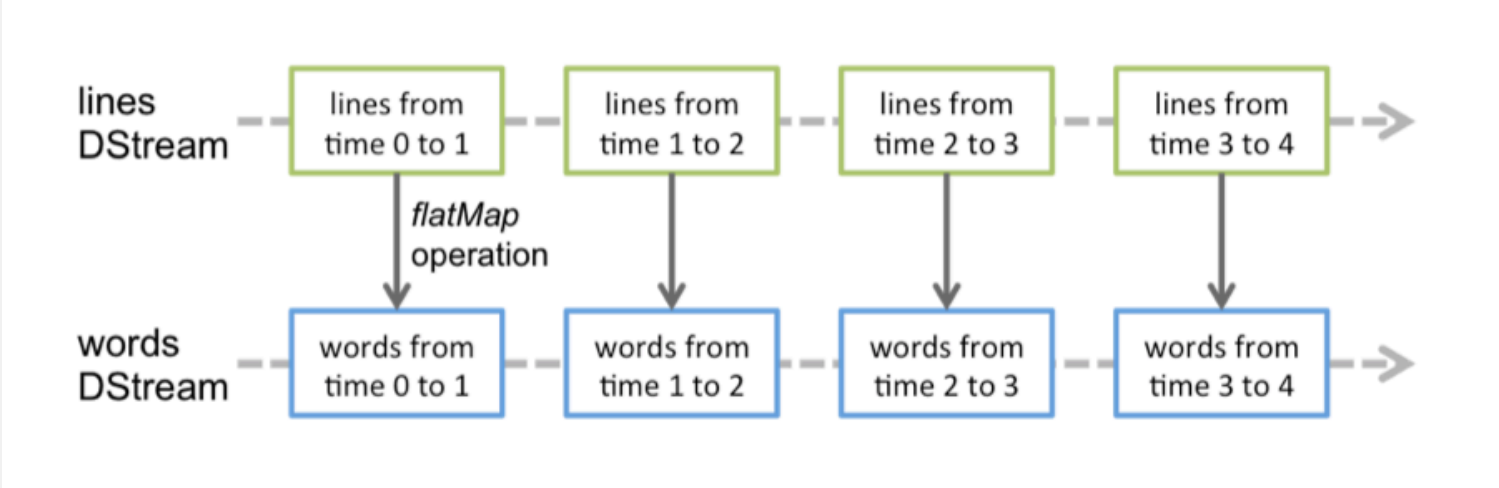
DStreams

DStream - это последовательность RDD, набранных за batch interval



Transformations

- Классические spark преобразования (map, reduceByKey, и т.д.) Применяются к каждому RDD индивидуально.



- Специальные операции
  - Скользящее окно
  - Обновление состояния

Spark Streaming Context

- Streaming запускается командой streamingContext.**start**().
- streamingContext.**awaitTermination**() - streaming будет работать до прерывания или ошибки
- Может быть остановлен командой streamingContext.**stop**()
- Только **один** активный streaming context на JVM
- Если контекст остановлен его **нельзя** перезапустить
- StreamingContext.stop() также останавливает SparkContext. С помощью параметра можно остановить только StreamingContext
- С помощью одного SparkContext можно создавать несколько StreamingContexts, не более одного в любой момент времени

Пример

Запустим пример WordCount

```
1.  ./bin/run-example streaming.JavaNetworkWordCount localhost 9999 // в первой
    консоли
2.  ……….
3.  // nc -lk 9999 // во второй консоли
4.  ……. .
5.  ……..
6.  // WordCount code
7.  SparkConf sparkConf = new SparkConf().setAppName("JavaNetworkWordCount");
8.  JavaStreamingContext ssc = new JavaStreamingContext(sparkConf,
    Durations.seconds(1));
9.
10. JavaReceiverInputDStream<String> lines = ssc.socketTextStream(
11.             args[0], Integer.parseInt(args[1]),
    StorageLevels.MEMORY_AND_DISK_SER);
12. JavaDStream<String> words = lines.flatMap(x ->
    Arrays.asList(SPACE.split(x)).iterator());
13. JavaPairDStream<String, Integer> wordCounts = words.mapToPair(s -> new
    Tuple2<>(s, 1)).reduceByKey((i1, i2) -> i1 + i2);

15. wordCounts.print();
16. ssc.start();
17. ssc.awaitTermination();
```

Receiver

Каждому входному DStream'у (кроме DStream'а из файла) нужен особенный объект **Receiver**, который будет читать контент из источника и сохранять его в памяти.

Receiver

- *Базовые источники:* файловая система, сокеты

```
streamingContext.textFileStream ( dataDirectory )
```

- *Расширенные источники* : Kafka, Flume, Twitter...

```
KafkaUtils.createStream ( ssc , zk , “name", {topic: 1})
```

- *Кастомные источники*

Спарку должно быть выделено слотов больше чем число receivers, т.к. **один приёмник расходует 1 слот!**
Посмотрим на пример в консоли.

Кастомный Receiver

- Компактное API
- Бывают 2х видов: надежные и ненадежные (В чем разница?)
- ssc.receiverStream(new MyReceiver(StotageLevel.MEMORY_ONLY))

```
class MyReceiver(storageLevel: StorageLevel) extends NetworkReceiver[String](storageLevel) {
def onStart() {
    // Setup stuff (start threads, open sockets, etc.) to start receiving data.
    // Must start new thread to receive data, as onStart() must be non-blocking.

    // Call store(...) in those threads to store received data into Spark's memory.

    // Call stop(...), restart(...) or reportError(...) on any thread based on how
    // different errors needs to be handled.

    // See corresponding method documentation for more details
}

def onStop() {
    // Cleanup stuff (stop threads, close sockets, etc.) to stop receiving data.
}
}
```

Reliable & Unreliable Receivers

Reliable Receivers

- Нет потери данных.
- Генерация блоков, сохранение данных в памяти спарка ложится на разработчика
- Сложность имплементации логики зависит от acknowledgement механизма источника.

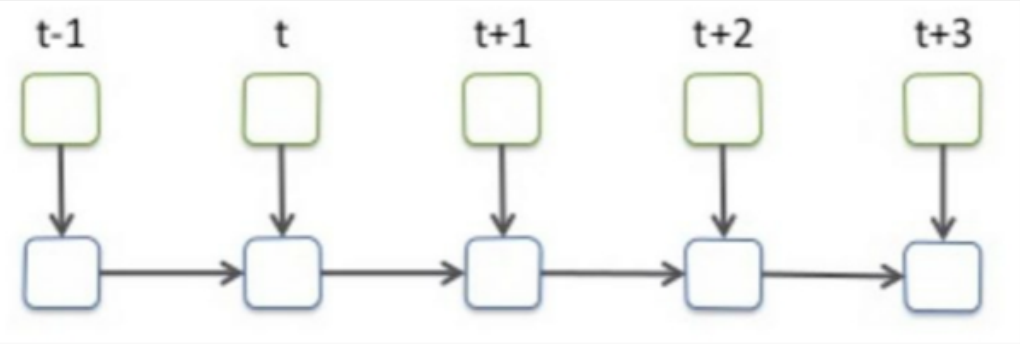
Unreliable Receivers

- Прост в реализации.
- Спарк сам контролирует размер блока и batch interval.
- Возможна потеря данных при падении.

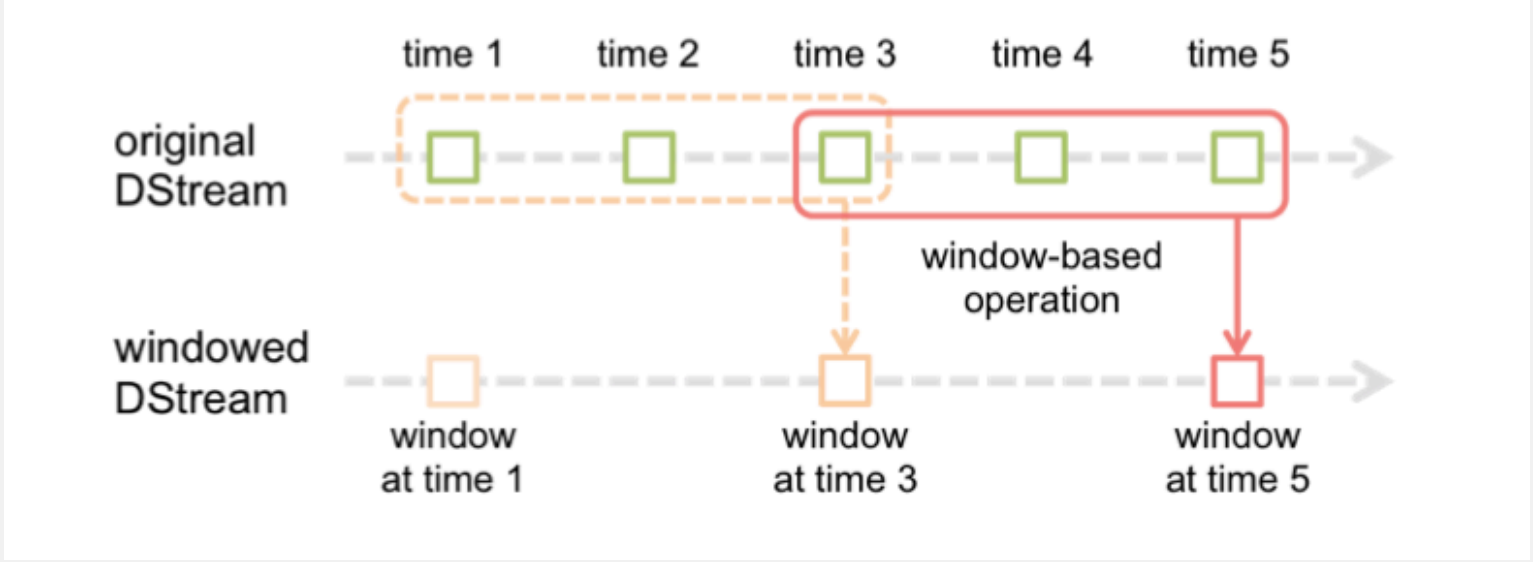
UpdateStateByKey

- Возможность работать с состояниями
- Необходима директория для checkpoints

```
1.  def updateFunction(newValues, runningCount):
2.      if runningCount is None:
3.          runningCount = 0
4.      return sum(newValues, runningCount)  # add the new values
    with the previous running count to get the new count
5.
6.  runningCounts = pairs.updateStateByKey(updateFunction)
```



Window operations



Window operations

- window(windowLength, slideInterval)
- countByWindow (windowLength , slideInterval)
- reduceByWindow(func, windowLength, slideInterval)
- reduceByKeyAndWindow (func, windowLength, slideInterval, [numTasks])
- reduceByKeyAndWindow (func, invFunc, windowLength, slideInterval, [numTasks])
- countByValueAndWindow(windowLength, slideInterval, [numTasks])

Скользящее окно

2 параметра - ширина окна и сдвиг окна

Хотим вычислять word count за последние 30 секунд каждые 10 секунд

```
1.  // Reduce last 30 seconds of data, every 10 seconds
2.  val windowedWordCounts =
    pairs.reduceByKeyAndWindow((a:Int,b:Int) => (a + b),
    Seconds(30), Seconds(10))

// Какую invFunc тут можно указать?
```

Output operations

- print () - **pprint()** in Python API
- saveAsTextFiles (prefix , [suffix])
- saveAsObjectFiles (prefix , [suffix]) - **not available** in the Python API
- saveAsHadoopFiles (prefix , [suffix]) - **not available** in the Python API
- foreachRDD (func)
  - Не путать с transform(func)

Без сохранения данных spark streaming не запустит обработку данных!

Важно

- DStreams запускаются только при выполнении output operations
- Внутри output operations должны быть RDD **actions** для запуска вычислений
- Для запуска вычислений внутри foreachRDD должны быть actions
- Output operation выполняются по одному в указанном порядке

foreachRDD

Что не так?

```
def sendRecord(rdd):
    connection = createNewConnection()  # executed at the driver
    rdd.foreach(lambda record: connection.send(record))
    connection.close()

dstream.foreachRDD(sendRecord)
```

foreachRDD

Что не так?

```
def sendRecord(record):
    connection = createNewConnection()
    connection.send(record)
    connection.close()

dstream.foreachRDD(lambda rdd: rdd.foreach(sendRecord))
```

foreachRDD

Что не так?

```
def sendPartition(iter):
    connection = createNewConnection()
    for record in iter:
        connection.send(record)
    connection.close()

dstream.foreachRDD(lambda rdd: rdd.foreachPartition(sendPartition))
```

foreachRDD

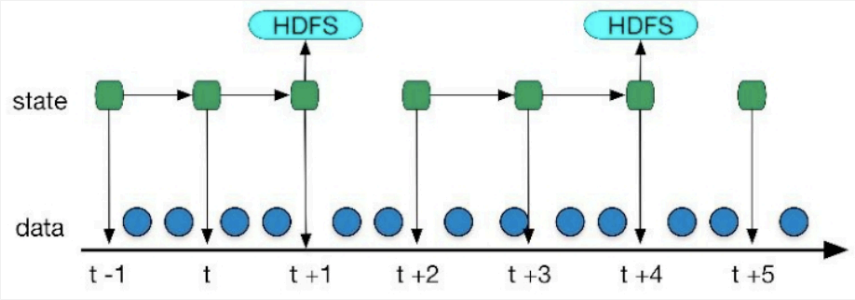
```
def sendPartition(iter):
    # ConnectionPool is a static, lazily initialized pool of connections
    connection = ConnectionPool.getConnection()
    for record in iter:
        connection.send(record)
    # return to the pool for future reuse
    ConnectionPool.returnConnection(connection)

dstream.foreachRDD(lambda rdd: rdd.foreachPartition(sendPartition))
```

Checkpoint

- Метаданные
  - Параметры конфигурации streaming context.
  - Преобразования DStream’a.
  - Информация о незавершенных батчах.
- Некоторые RDD

Когда применять:

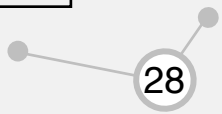
  - updateStateByKey или reduceByKeyAndWindow
  - Для защиты от падения драйвера
  - Работа 24/7



Checkpoints

- Устанавливаем директорию для checkpoints в отказоустойчивом хранилище (hdfs, s3, etc)
- **Для драйвера** - при первом **старте** создаем новый StreamingContext с директорией для checkpoints
- При **рестарте** пересоздаем StreamingContext, используя созданную ранее директорию для checkpoints

```
1.  # Function to create and setup a new StreamingContext
2.  def functionToCreateContext():
3.      sc = SparkContext(...)  # new context
4.      ssc = StreamingContext(...)
5.      lines = ssc.socketTextStream(...)  # create DStreams
6.      ...
7.      ssc.checkpoint(checkpointDirectory)  # set checkpoint directory
8.      return ssc

10. # Get StreamingContext from checkpoint data or create a new one
11. context = StreamingContext.getOrCreate(checkpointDirectory, functionToCreateContext)

12. # Do additional setup on context that needs to be done,
13. # irrespective of whether it is being started or restarted
14. context. …
15. # Аккуратно настраиваем частоту checkpoints - не слишком часто, но и не редко
16. dstream.checkpoint(checkpointInterval)

18. # Start the context
19. context.start()
20. context.awaitTermination()
```



Shared variables

Shared variables не могут быть восстановлены из checkpoints, поэтому надо позаботиться об их пересоздании при рестарте драйвера

```
1.  def getWordBlacklist(sparkContext):
2.      if ("wordBlacklist" not in globals()):
3.          globals()["wordBlacklist"] = sparkContext.broadcast(["a", "b", "c"])
4.      return globals()["wordBlacklist"]

6.  def getDroppedWordsCounter(sparkContext):
7.      if ("droppedWordsCounter" not in globals()):
8.          globals()["droppedWordsCounter"] = sparkContext.accumulator(0)
9.      return globals()["droppedWordsCounter"]

11. def echo(time, rdd):
12.     # Get or register the blacklist Broadcast
13.     blacklist = getWordBlacklist(rdd.context)
14.     # Get or register the droppedWordsCounter Accumulator
15.     droppedWordsCounter = getDroppedWordsCounter(rdd.context)

17.     # Use blacklist to drop words and use droppedWordsCounter to count them
18.     def filterFunc(wordCount):
19.         if wordCount[0] in blacklist.value:
20.             droppedWordsCounter.add(wordCount[1])
21.             False
22.         else:
23.             True

25.     counts = "Counts at time %s %s" % (time, rdd.filter(filterFunc).collect())

27. wordCounts.foreachRDD(echo)
```

Мониторинг

В Web UI на вкладке Streaming можно смотреть за параметрами стриминга

- **Processing Time** - время обработки батча
- **Scheduling delay** - время ожидания батчом своей очереди
- Batch interval - установленный при запуске задачи размер батча

Если очередь растет или processingTime > batchInterval => спарк не справляется с обработкой потока

Performance tuning

- Настраиваем batch interval, чтобы обработка шла со скоростью чтения данных receiver’ом
- Уменьшаем processing time - рефракторам код, эффективно используем Spark
- Регулируем уровень параллелизма в Data Receiving
- Регулируем уровень параллелизма в Data Processing - spark.default.parallelism
- Частота checkpoints
- Репартиционирование (учитываем shuffle)
- Тип сериализации (MEMORY_ONLY и т.д.)
- Увеличиваем число executors

Tuning Data receiving - чтение

Если чтение превратилось в bottle neck, распараллелим чтение.

Например имея несколько топиков Кафки для чтения, создадим по Receiver'у на каждый

```
1.  numStreams = 5
2.  kafkaStreams = [KafkaUtils.createStream(...) for _ in range
    (numStreams)]
3.  unifiedStream = streamingContext.union(*kafkaStreams)
4.  unifiedStream.pprint()
```

Tuning Data receiving - блоки

- Receivers собирают прочитанные данные в блоки перед сохранением в памяти Спарка
- Tuning spark.streaming.blockInterval
- Batch состоит из блоков
- Один task на каждый block
- Зная batchInterval, можем регулировать уровень параллелизма с помощью blockInterval
- Например, batchInterval = 2s, при blockInterval = 0.2s tasks = 10, при blockInterval = 0.1s tasks = 20

Spark 2. Structured Streaming

- Spark streaming для структурированных данных на Spark SQL движке
- Быстрый
- Масштабируемый
- Отказоустойчивый
- Гарантия end-to-end exactly-once на checkpoints и WAL
- Micro-batch модель (Continuous Processing since Spark 2.3)

Пример

Запустим пример WordCount

Бесконечная таблица с колонкой value

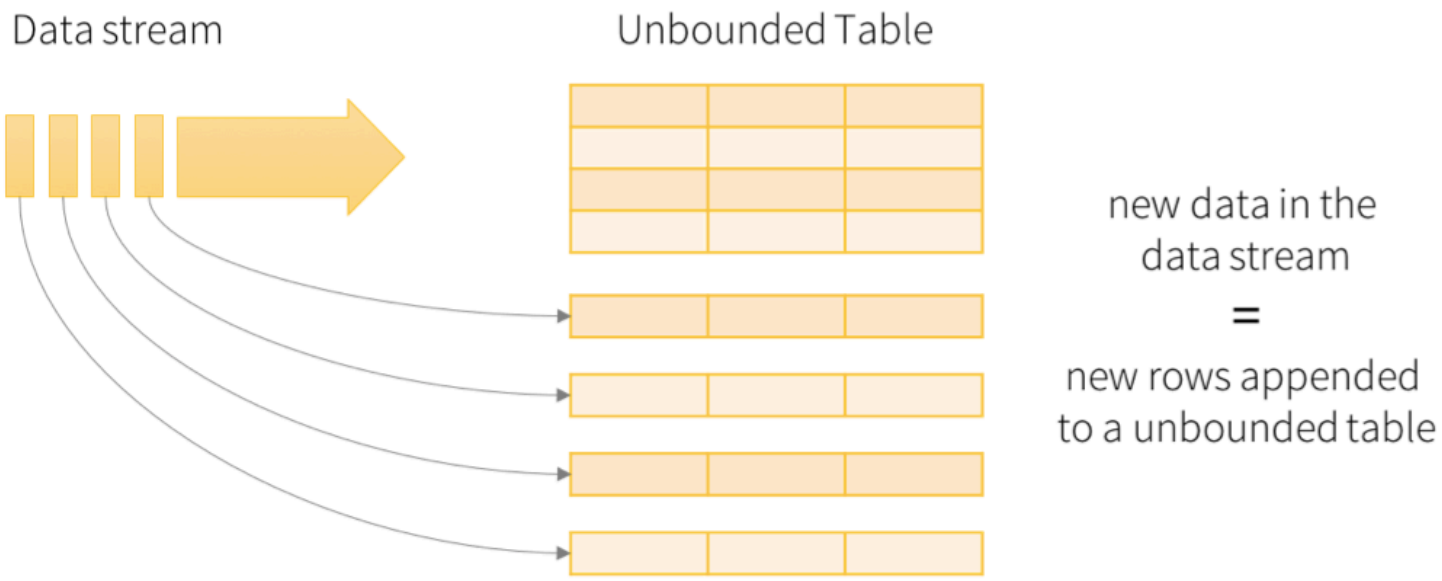
«Бегущий» word count

```
1.  ./bin/run-example org.apache.spark.examples.sql.streaming.JavaStructuredNetworkWordCount
    localhost 9999 // в первой консоли
2.  ……….
3.  // nc -lk 9999 // во второй консоли
4.  …….. .
5.  …….
6.  // WordCount code
7.  SparkSession spark = SparkSession.builder()
8.      .appName("JavaStructuredNetworkWordCount")
9.      .getOrCreate();
10. // Create DataFrame representing the stream of input lines from connection to localhost:9999
11. Dataset<Row> lines = spark
12.   .readStream()
13.   .format("socket")
14.   .option("host", "localhost")
15.   .option("port", 9999)
16.   .load();

18. // Split the lines into words
19. Dataset<String> words = lines
20.   .as(Encoders.STRING())
21.   .flatMap((FlatMapFunction<String, String>) x -> Arrays.asList(x.split(" ")).iterator(),
    Encoders.STRING());

23. // Generate running word count
24. Dataset<Row> wordCounts = words.groupBy("value").count();
```

Пример

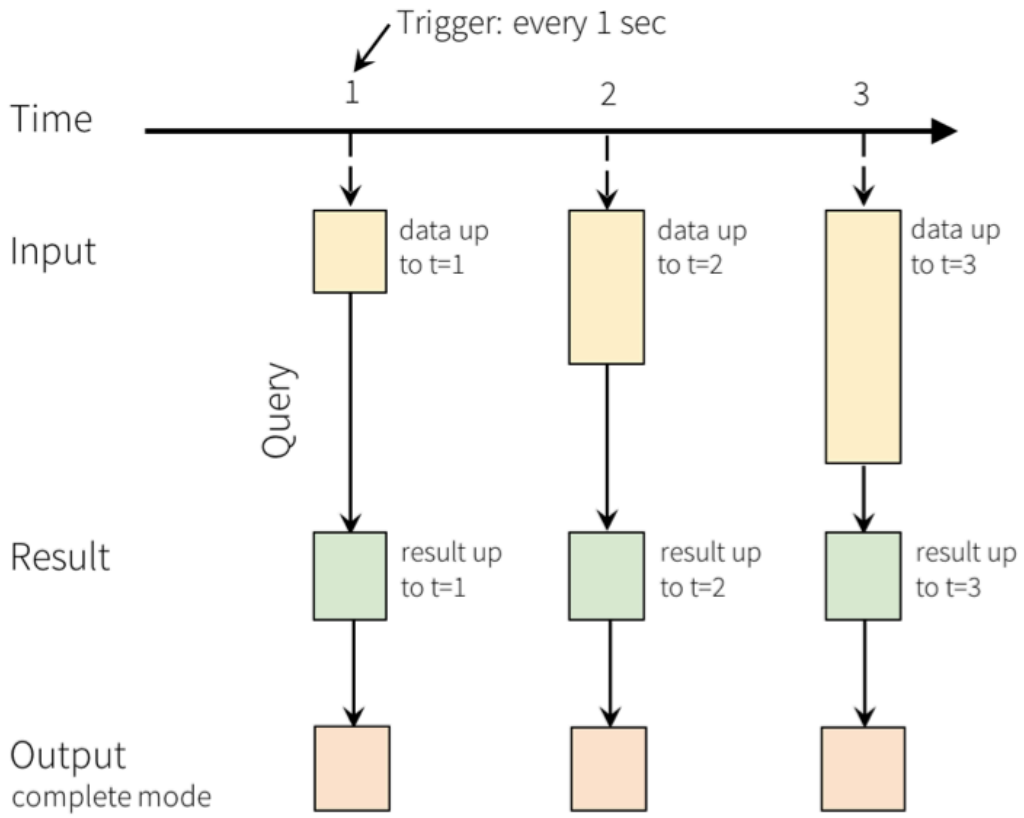
```
1.  // Start running the query that prints the running counts to
    the console

3.  StreamingQuery query = wordCounts.writeStream()
4.    .outputMode("complete") // на каждый апдейт общий результат
5.    .format("console")
6.    .start();

8.  query.awaitTermination(); // работать до прерывания/ошибки
```

Programming model



Data stream as an unbounded table

Programming model

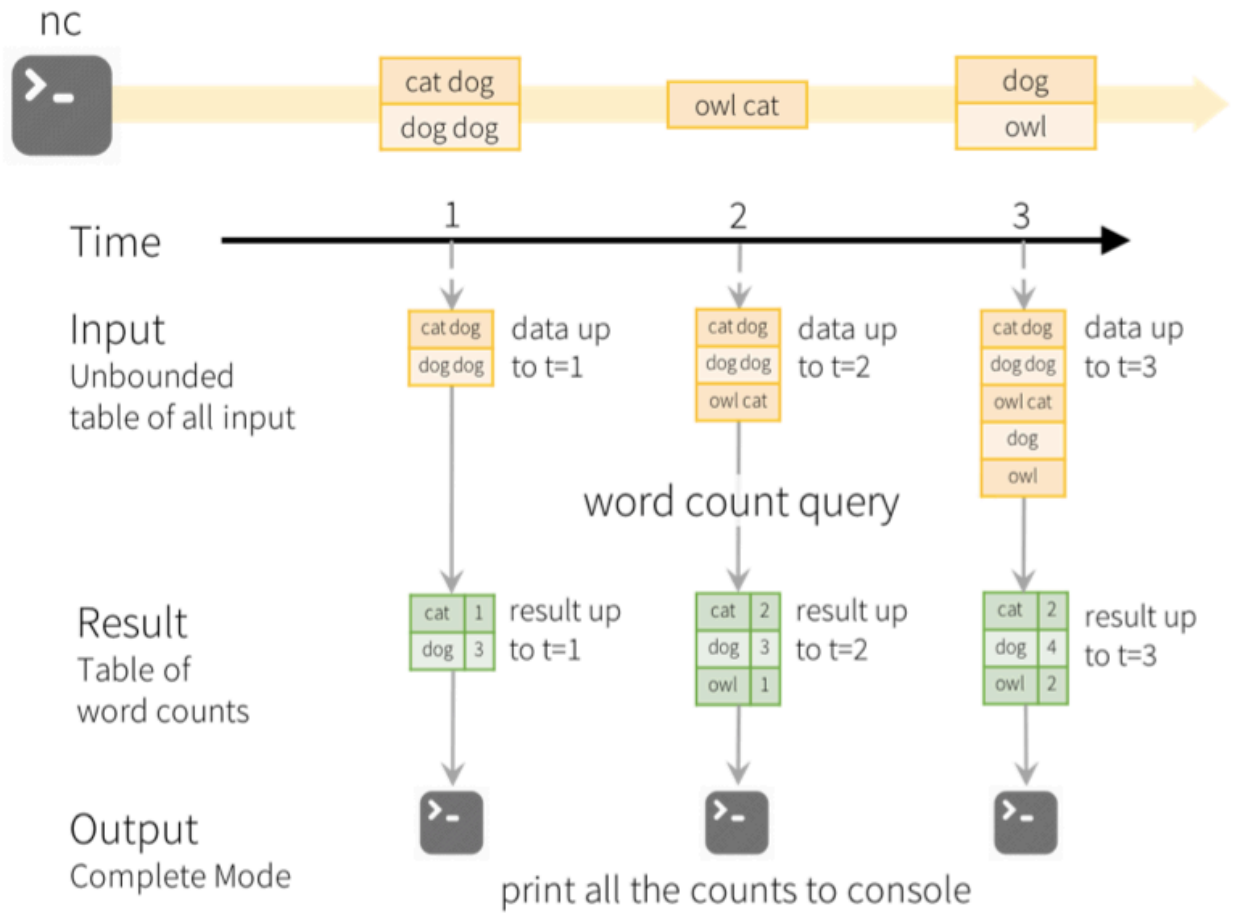


Programming Model for Structured Streaming

Output mode

- Complete Mode - Записывается в результат вся таблица. Storage connector решает как ему обрабатывать целую таблицу.
- Append Mode - Записываются только новые записи с последнего изменения таблицы. Применяется там где записанные строки таблицы не будут изменяться в будущем
- Update Mode - Записываются только измененные записи с последнего изменения таблицы (со Spark 2.1.1).

Programming model на примере



Model of the Quick Example

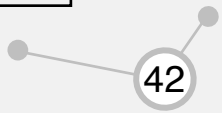
Structured Streaming

- 2 неограниченные таблицы - Input & Result
- Естественным образом доступен настоящий event time события
- end-to-end exactly-once semantics
- Доступно DataSet/DataFrame API

Пример

Чтение данных в csv формате

```
spark = SparkSession. ...

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

socketDF.isStreaming()    # Returns True for DataFrames that have streaming sources

socketDF.printSchema()

# Read all the csv files written atomically in a directory
userSchema = StructType().add("name", "string").add("age", "integer")
csvDF = spark \
    .readStream \
    .option("sep", ";") \
    .schema(userSchema) \ # Схема проверяется в runtime
    .csv("/path/to/directory")  # Equivalent to format("csv").load("/path/to/directory")
```



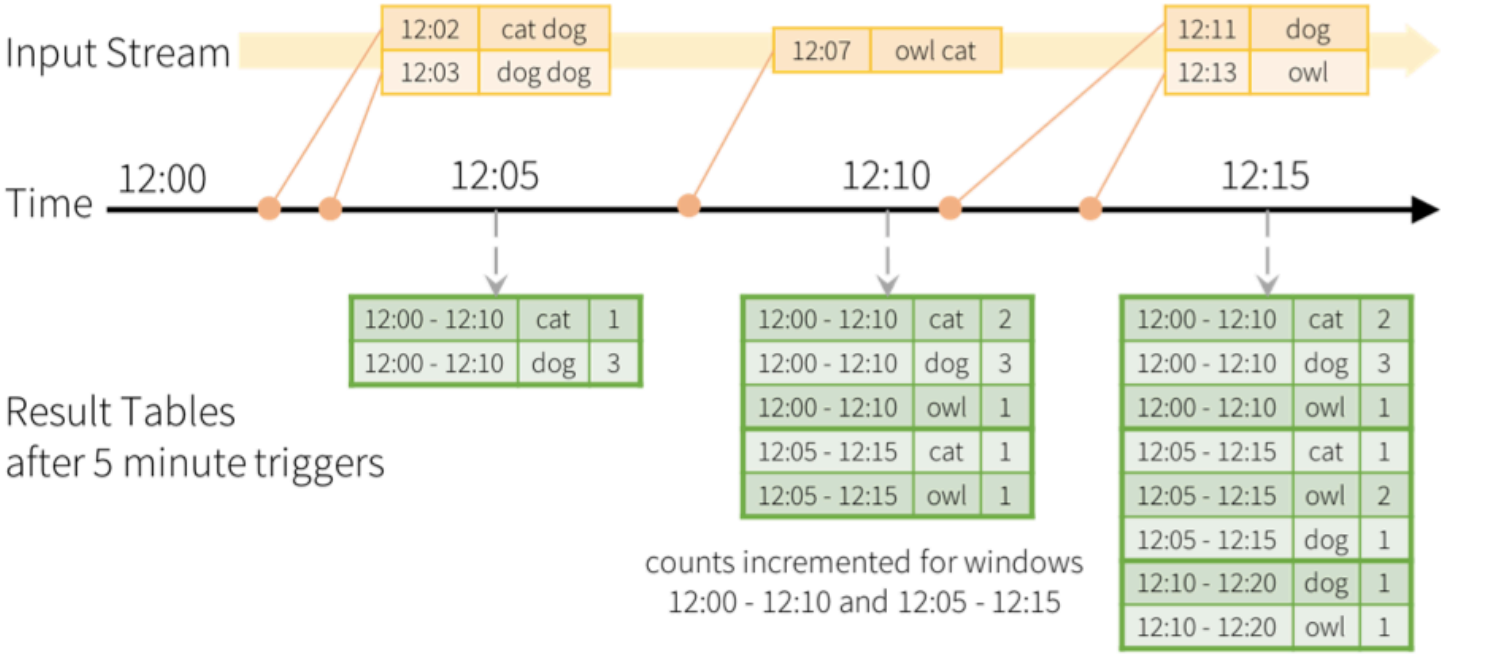
Пример

```
df = ...  # streaming DataFrame with IOT device data with schema { device:
string, deviceType: string, signal: double, time: DateType }

# Select the devices which have signal more than 10
df.select("device").where("signal > 10")

# Running count of the number of updates for each device type
df.groupBy("deviceType").count()
```

Пример

```
df = ...  # streaming DataFrame with IOT device data with schema { device:
string, deviceType: string, signal: double, time: DateType }

# Select the devices which have signal more than 10
df.select("device").where("signal > 10")

# Running count of the number of updates for each device type
df.groupBy("deviceType").count()


……….

df.createOrReplaceTempView("updates")
spark.sql("select count(*) from updates")  # returns another streaming DF

……..

df.isStreaming()
```

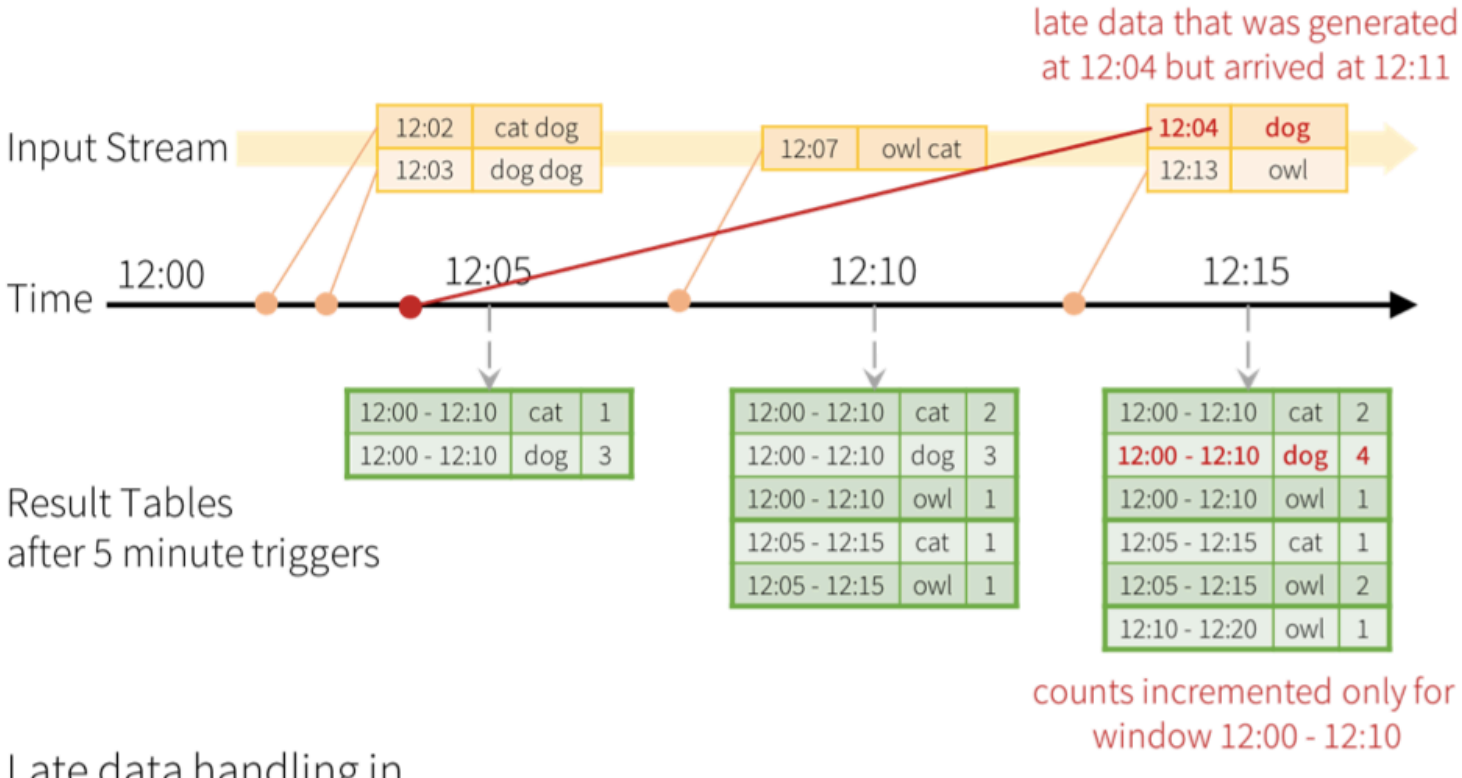
Window Operations



Пример Word Count with Sliding Window

```
words = ...  # streaming DataFrame of schema { timestamp: Timestamp, word:
String }

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

Handling Late Data

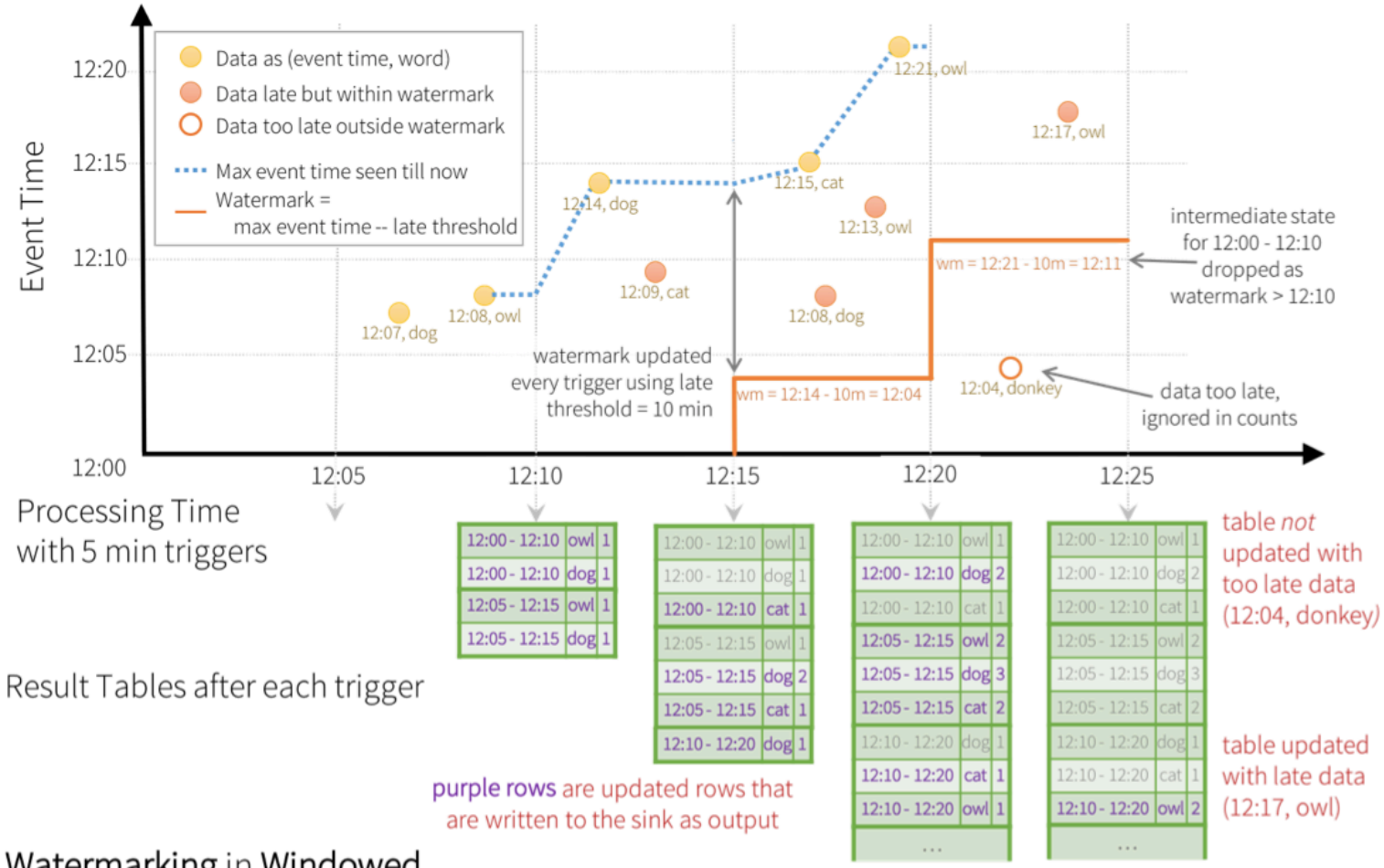


Watermarking (Spark 2.1)

- **Не можем долго держать в памяти старое состояние для опоздавших данных**
- **Только для потоков**
- **Только для Append и Update Modes**

```
words = ...  # streaming DataFrame of schema { timestamp: Timestamp, word:
String }

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

Watermarking



Watermarking

- Не можем долго держать в памяти старое состояние для опоздавших данных
- Имеет смысл только для потоков
- Только для Append и Update Modes
- withWaterMark вызывает по той же колонке что и groupBy
- df.groupBy("time").count().withWatermark("time", "1 min") - неверно, т.к. waterMark должен идти перед groupBy

Structured Streaming пока не содержит

- Multiple streaming aggregations (i.e. a chain of aggregations on a streaming DF) are not yet supported on streaming Datasets.
- Limit and take first N rows are not supported on streaming Datasets.
- Distinct operations on streaming Datasets are not supported.
- Sorting operations are supported on streaming Datasets only after an aggregation and in Complete Output Mode.
- Few types of outer joins on streaming Datasets are not supported.

Spark Streaming vs Structured Streaming

	Spark Streaming	Structured Streaming
Что в основе?	RDD	DataSets/DataFrames
Streaming	DStream	DataSets/DataFrames API
Programming model	Micro batch	Unbounded tables
Поддержка Late Data	Нет	Watermarks

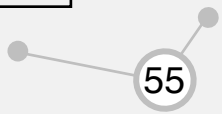
Spark SQL

Generic Load/Save Functions

```
df = spark.read.load("examples/src/main/resources/users.parquet")
df.select("name", "favorite_color").write.save("namesAndFavColors.parquet"

……..

df = spark.read.load("examples/src/main/resources/people.json", format="json")
df.select("name", "age").write.save("namesAndAges.parquet", format="parquet")

…….

df = spark.read.load("examples/src/main/resources/people.csv",
                     format="csv", sep=":", inferSchema="true", header="true")

……

df = spark.sql("SELECT * FROM parquet.`examples/src/main/resources/
users.parquet`")
```

JSON

```python
# spark is from the previous example.
sc = spark.sparkContext

# A JSON dataset is pointed to by path.
# The path can be either a single text file or a directory storing text files
path = "examples/src/main/resources/people.json"
peopleDF = spark.read.json(path)

# The inferred schema can be visualized using the printSchema() method
peopleDF.printSchema()
# root
#  |-- age: long (nullable = true)
#  |-- name: string (nullable = true)

# Creates a temporary view using the DataFrame
peopleDF.createOrReplaceTempView("people")

# SQL statements can be run by using the sql methods provided by spark
teenagerNamesDF = spark.sql("SELECT name FROM people WHERE age BETWEEN 13 AND 19")
teenagerNamesDF.show()
# +------+
# |  name|
# +------+
# |Justin|
# +------+

# Alternatively, a DataFrame can be created for a JSON dataset represented by
# an RDD[String] storing one JSON object per string
jsonStrings = ['{"name":"Yin","address":{"city":"Columbus","state":"Ohio"}}']
otherPeopleRDD = sc.parallelize(jsonStrings)
otherPeople = spark.read.json(otherPeopleRDD)
otherPeople.show()
# +---------------+----+
# |        address|name|
# +---------------+----+
# |[Columbus,Ohio]| Yin|
# +---------------+----+
```

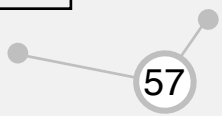


Hive tables

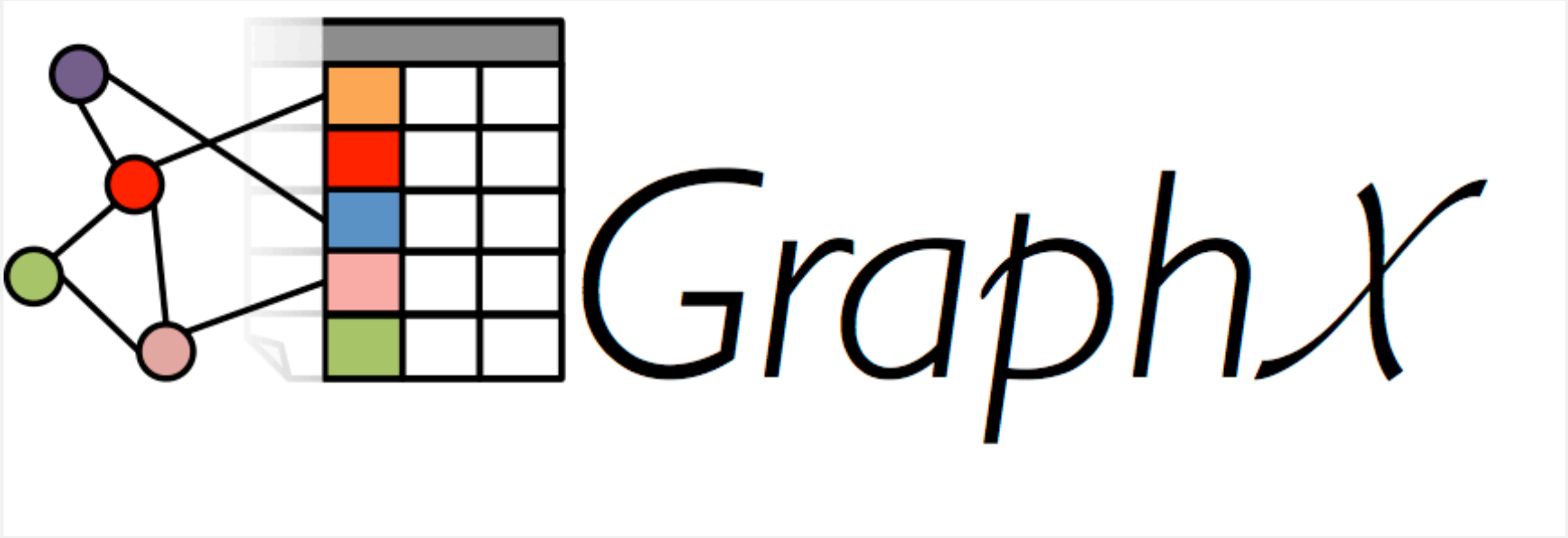
```
from os.path import expanduser, join, abspath

from pyspark.sql import SparkSession
from pyspark.sql import Row

# warehouse_location points to the default location for managed databases and tables
warehouse_location = abspath('spark-warehouse')

spark = SparkSession \
    .builder \
    .appName("Python Spark SQL Hive integration example") \
    .config("spark.sql.warehouse.dir", warehouse_location) \
    .enableHiveSupport() \
    .getOrCreate()

# spark is an existing SparkSession
spark.sql("CREATE TABLE IF NOT EXISTS src (key INT, value STRING) USING hive")
spark.sql("LOAD DATA LOCAL INPATH 'examples/src/main/resources/kv1.txt' INTO TABLE src")

# Queries are expressed in HiveQL
spark.sql("SELECT * FROM src").show()
# +---+-------+
# |key|  value|
# +---+-------+
# |238|val_238|
# | 86| val_86|
# |311|val_311|
# ...
```

Hive tables

```
# Aggregation queries are also supported.
spark.sql("SELECT COUNT(*) FROM src").show()
# +--------+
# |count(1)|
# +--------+
# |    500 |
# +--------+

# The results of SQL queries are themselves DataFrames and support all normal functions.
sqlDF = spark.sql("SELECT key, value FROM src WHERE key < 10 ORDER BY key")

# The items in DataFrames are of type Row, which allows you to access each column by ordinal.
stringsDS = sqlDF.rdd.map(lambda row: "Key: %d, Value: %s" % (row.key, row.value))
for record in stringsDS.collect():
    print(record)
# Key: 0, Value: val_0
# Key: 0, Value: val_0
# Key: 0, Value: val_0
# ...

# You can also use DataFrames to create temporary views within a SparkSession.
Record = Row("key", "value")
recordsDF = spark.createDataFrame([Record(i, "val_" + str(i)) for i in range(1, 101)])
recordsDF.createOrReplaceTempView("records")

# Queries can then join DataFrame data with data stored in Hive.
spark.sql("SELECT * FROM records r JOIN src s ON r.key = s.key").show()
# +---+------+---+------+
# |key| value|key| value|
# +---+------+---+------+
# |  2| val_2|  2| val_2|
# |  4| val_4|  4| val_4|
# |  5| val_5|  5| val_5|
# ...
```



GraphX



GraphX

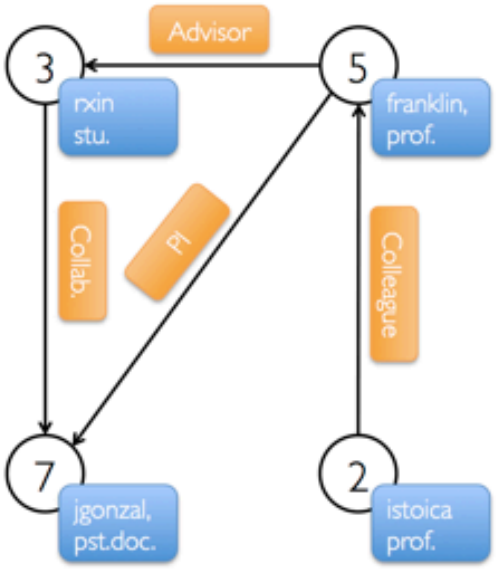
- Позволяет работать с графами на кластере
- Основан на RDD
- Предоставляет множество методов для манипуляции графами - subgraph, joinVertices, and aggregateMessages и тд
- Реализованы некоторые алгоритмы на графах - PageRank, Connected Components, Triangle Counting
- Инструмент для построения графа различными способами
- Нет python

GraphX

VertexRDD[VD] & EdgeRDD[ED] оптимизированные наследники класса RDD[(VertexId, VD)] and RDD[Edge[ED]]

```
import org.apache.spark._
import org.apache.spark.graphx._
// To make some of the examples work we will also need RDD
import org.apache.spark.rdd.RDD


.....


class Graph[VD, ED] {
  val vertices: VertexRDD[VD]
  val edges: EdgeRDD[ED]
}
```

GraphX



Vertex Table

Id	Property (V)
3	(rxin, student)
7	(jgonzal, postdoc)
5	(franklin, professor)
2	(istoica, professor)

Edge Table

SrcId	DstId	Property (E)
3	7	Collaborator
5	3	Advisor
2	5	Colleague
5	7	PI

Пример

```scala
// Assume the SparkContext has already been constructed
val sc: SparkContext
// Create an RDD for the vertices
val users: RDD[(VertexId, (String, String))] =
  sc.parallelize(Array((3L, ("rxin", "student")), (7L, ("jgonzal",
"postdoc")),
                       (5L, ("franklin", "prof")), (2L, ("istoica", "prof"))))
// Create an RDD for edges
val relationships: RDD[Edge[String]] =
  sc.parallelize(Array(Edge(3L, 7L, "collab"),    Edge(5L, 3L, "advisor"),
                       Edge(2L, 5L, "colleague"), Edge(5L, 7L, "pi")))
// Define a default user in case there are relationship with missing user
val defaultUser = ("John Doe", "Missing")
// Build the initial Graph
val graph = Graph(users, relationships, defaultUser)
```

Пример

```scala
val graph: Graph[(String, String), String] // Constructed from above
// Count all users which are postdocs
graph.vertices.filter { case (id, (name, pos)) => pos == "postdoc" }.count
// Count all the edges where src > dst
graph.edges.filter(e => e.srcId > e.dstId).count
```

Пример

Используем Триплет

```
val graph: Graph[(String, String), String] // Constructed from above
// Use the triplets view to create an RDD of facts.
val facts: RDD[String] =
  graph.triplets.map(triplet =>
    triplet.srcAttr._1 + " is the " + triplet.attr + " of " +
triplet.dstAttr._1)
facts.collect.foreach(println(_))

# franklin is the advisor of rxin
```

Property Operators

```
class Graph[VD, ED] {
  def mapVertices[VD2](map: (VertexId, VD) => VD2): Graph[VD2, ED]
  def mapEdges[ED2](map: Edge[ED] => ED2): Graph[VD, ED2]
  def mapTriplets[ED2](map: EdgeTriplet[VD, ED] => ED2): Graph[VD, ED2]
}
```

```
……
val newVertices = graph.vertices.map { case (id, attr) => (id, mapUdf(id,
attr)) }
val newGraph = Graph(newVertices, graph.edges)

val newGraph = graph.mapVertices((id, attr) => mapUdf(id, attr))
……
```

```
// Given a graph where the vertex property is the out degree
val inputGraph: Graph[Int, String] =
  graph.outerJoinVertices(graph.outDegrees)((vid, _, degOpt) =>
degOpt.getOrElse(0))
// Construct a graph where each edge contains the weight
// and each vertex is the initial PageRank
val outputGraph: Graph[Double, Double] =
  inputGraph.mapTriplets(triplet => 1.0 / triplet.srcAttr).mapVertices((id, _)
=> 1.0)
```

Structural Operators

```
class Graph[VD, ED] {
  def reverse: Graph[VD, ED]
  def subgraph(epred: EdgeTriplet[VD,ED] => Boolean,
               vpred: (VertexId, VD) => Boolean): Graph[VD, ED]
  def mask[VD2, ED2](other: Graph[VD2, ED2]): Graph[VD, ED]
  def groupEdges(merge: (ED, ED) => ED): Graph[VD,ED]
}
```

……

```
// Run Connected Components
val ccGraph = graph.connectedComponents() // No longer contains missing field
// Remove missing vertices as well as the edges to connected to them
val validGraph = graph.subgraph(vpred = (id, attr) => attr._2 != "Missing")
// Restrict the answer to the valid subgraph
val validCCGraph = ccGraph.mask(validGraph)
```

Join Operators

```
class Graph[VD, ED] {
  def joinVertices[U](table: RDD[(VertexId, U)])(map: (VertexId, VD, U) => VD)
    : Graph[VD, ED]
  def outerJoinVertices[U, VD2](table: RDD[(VertexId, U)])(map: (VertexId, VD,
Option[U]) => VD2)
    : Graph[VD2, ED]
}
```

Computing Degree Information

```
// Define a reduce operation to compute the highest degree vertex
def max(a: (VertexId, Int), b: (VertexId, Int)): (VertexId, Int) = {
  if (a._2 > b._2) a else b
}
// Compute the max degrees
val maxInDegree: (VertexId, Int)  = graph.inDegrees.reduce(max)
val maxOutDegree: (VertexId, Int) = graph.outDegrees.reduce(max)
val maxDegrees: (VertexId, Int)   = graph.degrees.reduce(max)
```

Graph Builders

```
object GraphLoader {
  def edgeListFile(
      sc: SparkContext,
      path: String,
      canonicalOrientation: Boolean = false,
      minEdgePartitions: Int = 1)
    : Graph[Int, Int]
}

…….

object Graph {
  def apply[VD, ED](
      vertices: RDD[(VertexId, VD)],
      edges: RDD[Edge[ED]],
      defaultVertexAttr: VD = null)
    : Graph[VD, ED]

  def fromEdges[VD, ED](
      edges: RDD[Edge[ED]],
      defaultValue: VD): Graph[VD, ED]

  def fromEdgeTuples[VD](
      rawEdges: RDD[(VertexId, VertexId)],
      defaultValue: VD,
      uniqueEdges: Option[PartitionStrategy] = None): Graph[VD, Int]

}
```

ValueRDD

```
class VertexRDD[VD] extends RDD[(VertexId, VD)] {
  // Filter the vertex set but preserves the internal index
  def filter(pred: Tuple2[VertexId, VD] => Boolean): VertexRDD[VD]
  // Transform the values without changing the ids (preserves the internal
index)
  def mapValues[VD2](map: VD => VD2): VertexRDD[VD2]
  def mapValues[VD2](map: (VertexId, VD) => VD2): VertexRDD[VD2]
  // Show only vertices unique to this set based on their VertexId's
  def minus(other: RDD[(VertexId, VD)])
  // Remove vertices from this set that appear in the other set
  def diff(other: VertexRDD[VD]): VertexRDD[VD]
  // Join operators that take advantage of the internal indexing to accelerate
joins (substantially)
  def leftJoin[VD2, VD3](other: RDD[(VertexId, VD2)])(f: (VertexId, VD,
Option[VD2]) => VD3): VertexRDD[VD3]
  def innerJoin[U, VD2](other: RDD[(VertexId, U)])(f: (VertexId, VD, U) =>
VD2): VertexRDD[VD2]
  // Use the index on this RDD to accelerate a `reduceByKey` operation on the
input RDD.
  def aggregateUsingIndex[VD2](other: RDD[(VertexId, VD2)], reduceFunc: (VD2,
VD2) => VD2): VertexRDD[VD2]
}
```

EdgeRDD

```
// Transform the edge attributes while preserving the structure
def mapValues[ED2](f: Edge[ED] => ED2): EdgeRDD[ED2]
// Reverse the edges reusing both attributes and structure
def reverse: EdgeRDD[ED]
// Join two `EdgeRDD`s partitioned using the same partitioning strategy.
def innerJoin[ED2, ED3](other: EdgeRDD[ED2])(f: (VertexId, VertexId, ED, ED2)
=> ED3): EdgeRDD[ED3]
```

Connected Components

```
import org.apache.spark.graphx.GraphLoader

// Load the graph as in the PageRank example
val graph = GraphLoader.edgeListFile(sc, "data/graphx/followers.txt")
// Find the connected components
val cc = graph.connectedComponents().vertices
// Join the connected components with the usernames
val users = sc.textFile("data/graphx/users.txt").map { line =>
  val fields = line.split(",")
  (fields(0).toLong, fields(1))
}
val ccByUsername = users.join(cc).map {
  case (id, (username, cc)) => (username, cc)
}
// Print the result
println(ccByUsername.collect().mkString("\n"))
```

Spark ML

Data type

- **Vectors**

```
import numpy as np
import scipy.sparse as sps
from pyspark.mllib.linalg import Vectors

# Use a NumPy array as a dense vector.
dv1 = np.array([1.0, 0.0, 3.0])

# Use a Python list as a dense vector.
dv2 = [1.0, 0.0, 3.0]

# Create a SparseVector.
sv1 = Vectors.sparse(3, [0, 2], [1.0, 3.0])

# Use a single-column SciPy csc_matrix as a sparse vector.
sv2 = sps.csc_matrix((np.array([1.0, 3.0]), np.array([0, 2]), np.array([0,
2])), shape=(3, 1))
```

Data type

- **LabeledPoint**

```
from pyspark.mllib.linalg import SparseVector
from pyspark.mllib.regression import LabeledPoint

# Create a labeled point with a positive label and a dense feature vector.
pos = LabeledPoint(1.0, [1.0, 0.0, 3.0])

# Create a labeled point with a negative label and a sparse feature vector.
neg = LabeledPoint(0.0, SparseVector(3, [0, 2], [1.0, 3.0]))
```

Data type

- **LocalMatrix**

```
from pyspark.mllib.linalg import Matrix, Matrices

# Create a dense matrix ((1.0, 2.0), (3.0, 4.0), (5.0, 6.0))
dm2 = Matrices.dense(3, 2, [1, 3, 5, 2, 4, 6])

# Create a sparse matrix ((9.0, 0.0), (0.0, 8.0), (0.0, 6.0))
sm = Matrices.sparse(3, 2, [0, 1, 3], [0, 2, 1], [9, 6, 8])
```

Data type

- **RowMatrix**

```scala
from pyspark.mllib.linalg.distributed import RowMatrix

# Create an RDD of vectors.
rows = sc.parallelize([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]])

# Create a RowMatrix from an RDD of vectors.
mat = RowMatrix(rows)

# Get its size.
m = mat.numRows()  # 4
n = mat.numCols()  # 3

# Get the rows as an RDD of vectors again.
rowsRDD = mat.rows
```

Data type

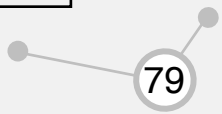
- **CoordinateMatrix**
  - **Each entry is (i: Long, j: Long, value: Double)**

```
from pyspark.mllib.linalg.distributed import CoordinateMatrix, MatrixEntry

# Create an RDD of coordinate entries.
#   - This can be done explicitly with the MatrixEntry class:
entries = sc.parallelize([MatrixEntry(0, 0, 1.2), MatrixEntry(1, 0, 2.1), MatrixEntry(2, 1, 3.7)])
#   - or using (long, long, float) tuples:
entries = sc.parallelize([(0, 0, 1.2), (1, 0, 2.1), (2, 1, 3.7)])

# Create an CoordinateMatrix from an RDD of MatrixEntries.
mat = CoordinateMatrix(entries)

# Get its size.
m = mat.numRows()  # 3
n = mat.numCols()  # 2

# Get the entries as an RDD of MatrixEntries.
entriesRDD = mat.entries

# Convert to a RowMatrix.
rowMat = mat.toRowMatrix()

# Convert to an IndexedRowMatrix.
indexedRowMat = mat.toIndexedRowMatrix()

# Convert to a BlockMatrix.
blockMat = mat.toBlockMatrix()
```

Data type

- **BlockMatrix**
  - **Each entry is (i: Long, j: Long, value: Double)**

```
from pyspark.mllib.linalg import Matrices
from pyspark.mllib.linalg.distributed import BlockMatrix

# Create an RDD of sub-matrix blocks.
blocks = sc.parallelize([((0, 0), Matrices.dense(3, 2, [1, 2, 3, 4, 5, 6])),
                         ((1, 0), Matrices.dense(3, 2, [7, 8, 9, 10, 11, 12]))])

# Create a BlockMatrix from an RDD of sub-matrix blocks.
mat = BlockMatrix(blocks, 3, 2)

# Get its size.
m = mat.numRows()  # 6
n = mat.numCols()  # 2

# Get the blocks as an RDD of sub-matrix blocks.
blocksRDD = mat.blocks

# Convert to a LocalMatrix.
localMat = mat.toLocalMatrix()

# Convert to an IndexedRowMatrix.
indexedRowMat = mat.toIndexedRowMatrix()

# Convert to a CoordinateMatrix.
coordinateMat = mat.toCoordinateMatrix()
```

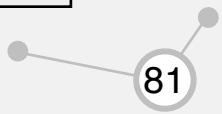


MLLib

- Classification: logistic regression, naive Bayes,...
- Regression: generalized linear regression, isotonic regression,...
- Decision trees, random forests, and gradient-boosted trees
- Recommendation: alternating least squares (ALS)
- Clustering: K -means, Gaussian mixtures (GMMs),...
- Topic modeling: latent Dirichlet allocation (LDA)
- Feature transformations: standardization, normalization, hashing,...
- Model evaluation and hyper-parameter tuning
- Frequent itemset and sequential pattern mining: FP -growth, association rules, PrefixSpan
- Distributed linear algebra: singular value decomposition (SVD), principal component analysis (PCA),...
- Statistics: summary statistics, hypothesis testing,...

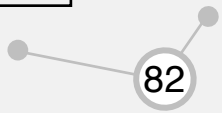
Classification

```
from pyspark.mllib.classification import SVMWithSGD, SVMModel
from pyspark.mllib.regression import LabeledPoint

# Load and parse the data
def parsePoint(line):
    values = [float(x) for x in line.split(' ')]
    return LabeledPoint(values[0], values[1:])

data = sc.textFile("data/mllib/sample_svm_data.txt")
parsedData = data.map(parsePoint)

# Build the model
model = SVMWithSGD.train(parsedData, iterations=100)

# Evaluating the model on training data
labelsAndPreds = parsedData.map(lambda p: (p.label,
model.predict(p.features)))
trainErr = labelsAndPreds.filter(lambda lp: lp[0] != lp[1]).count() /
float(parsedData.count())
print("Training Error = " + str(trainErr))

# Save and load model
model.save(sc, "target/tmp/pythonSVMWithSGDModel")
sameModel = SVMModel.load(sc, "target/tmp/pythonSVMWithSGDModel")
```

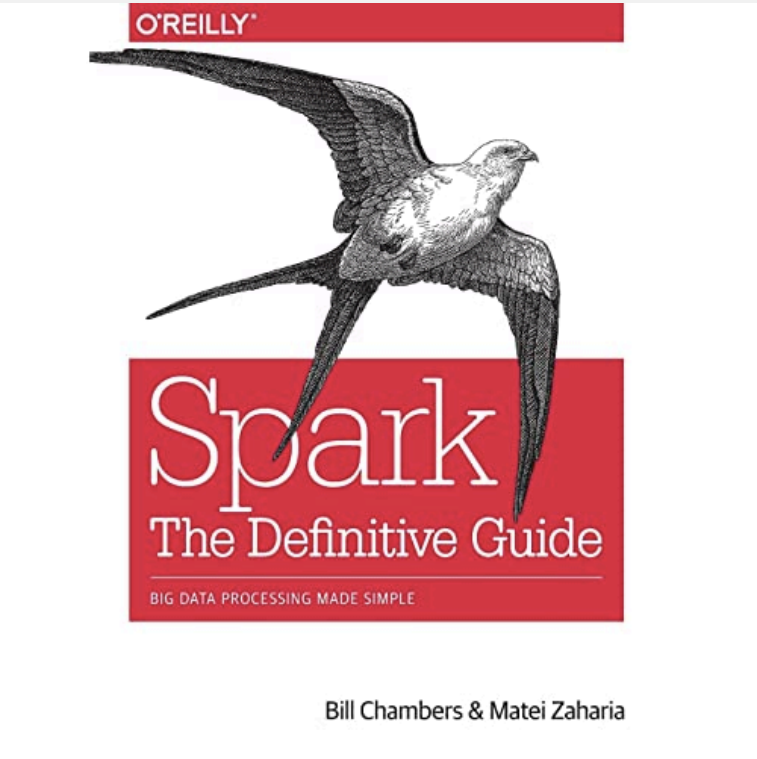


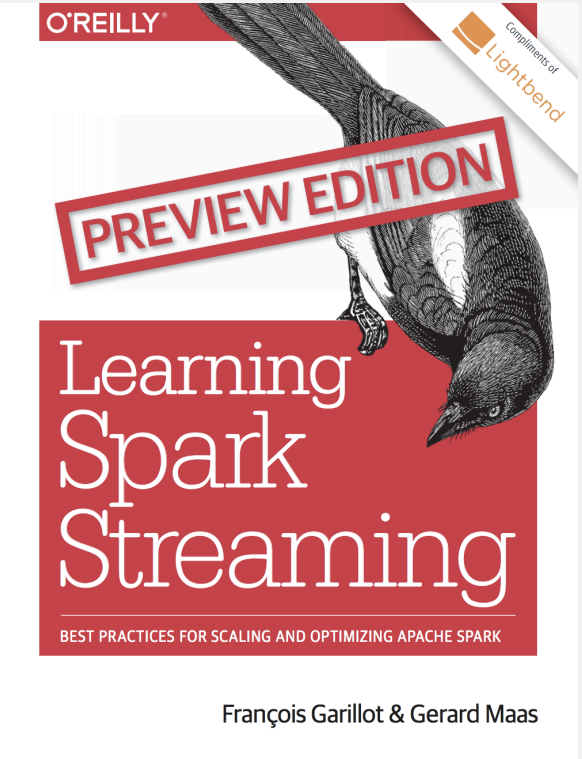
Collaborative filtering

```
from pyspark.mllib.recommendation import ALS, MatrixFactorizationModel, Rating

# Load and parse the data
data = sc.textFile("data/mllib/als/test.data")
ratings = data.map(lambda l: l.split(','))\
    .map(lambda l: Rating(int(l[0]), int(l[1]), float(l[2])))

# Build the recommendation model using Alternating Least Squares
rank = 10
numIterations = 10
model = ALS.train(ratings, rank, numIterations)

# Evaluate the model on training data
testdata = ratings.map(lambda p: (p[0], p[1]))
predictions = model.predictAll(testdata).map(lambda r: ((r[0], r[1]), r[2]))
ratesAndPreds = ratings.map(lambda r: ((r[0], r[1]), r[2])).join(predictions)
MSE = ratesAndPreds.map(lambda r: (r[1][0] - r[1][1])**2).mean()
print("Mean Squared Error = " + str(MSE))

# Save and load model
model.save(sc, "target/tmp/myCollaborativeFilter")
sameModel = MatrixFactorizationModel.load(sc, "target/tmp/
myCollaborativeFilter")
```



Литература


O'REILLY
Spark
The Definitive Guide
BIG DATA PROCESSING MADE SIMPLE
Bill Chambers & Matei Zaharia


O'REILLY
Compliments of
Lightbend
PREVIEW EDITION
Learning
Spark
Streaming
BEST PRACTICES FOR SCALING AND OPTIMIZING APACHE SPARK
François Garillot & Gerard Maas

Семинар

1. Выведем самые популярные хэштеги в твитах за последние 10 минут - выводить топ 10 каждые 30 секунд.
2. Имея поток твитов из кафки и файл со всеми пользователями Twitter необходимо среди пользователей, у которых более 10000 подписчиков, посчитать самых активных по числу твитов за последний час. Считаем каждые 15 минут.
3. Написать PageRank на GraphX, имея 2 файла - followers и users, сджойнить результат
4. Имеется HBase таблица в пабликами и профилями vk, у каждой записи есть приоритет от 0 до 100. Необходимо выбрать для обкатки топ 100к пабликов и топ 50к профилей по приоритету. Далее записать их в HDFS таким образом, чтобы сначала шли паблики и профили, у которых более 2000 подписчиков, отсортированные по времени последней скачки (давно не качавшиеся на вершину списка), затем меньше или равно 2000, отсортированные по такому же принципу.

Домашнее задание №4

Реализуем алгоритм PageRank

$$PR(p_i; t+1) = \frac{1-d}{N} + d \sum_{p_j \in M(p_i)} \frac{PR(p_j; t)}{L(p_j)},$$

Данные: LiveJournal social network

Описание https://goo.gl/ghdIrU

Скачать https://goo.gl/PUhTZb

Hadoop /data/voropaev/soc-LiveJournal1.txt

Срок сдачи

28.05.2020

Максимум

15 баллов

Домашнее задание

Сравниваем скорость работы алгоритма PageRank на Spark, MapReduce и GraphX

Отчет:

- Скорость работы трех реализаций алгоритма PageRank
- Топ 10 вершин графа
- Исходные коды

ТЕХНОСФЕРА

Спасибо за внимание!

Смирнов Даниил

dr.smirnov@corp.mail.ru

Не забудьте отметиться