SELA DEVELOPER PRACTICE

Spark Workshop With Python

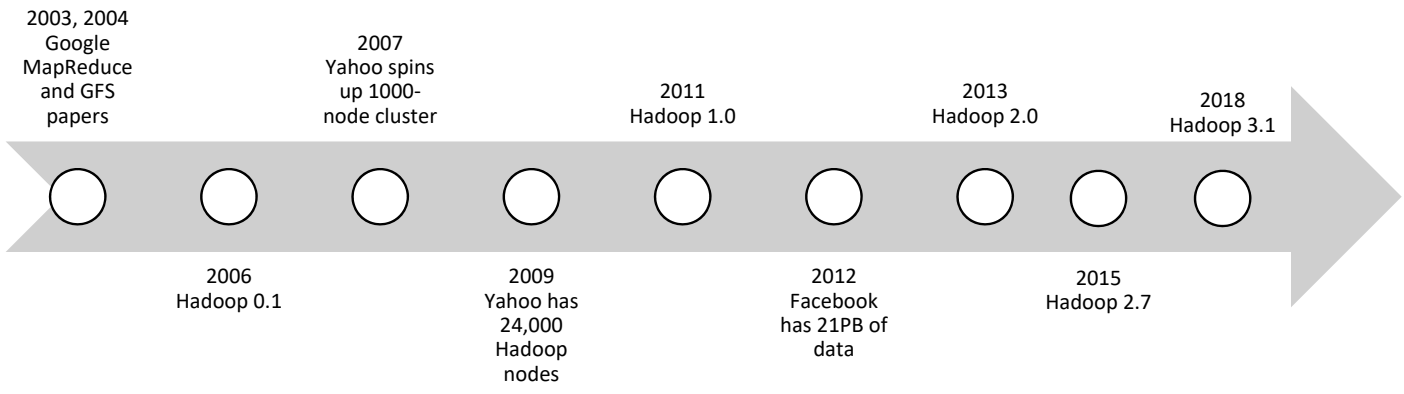
Netanel Malka netanelm@sela.co.il

Hadoop And Spark Introduction

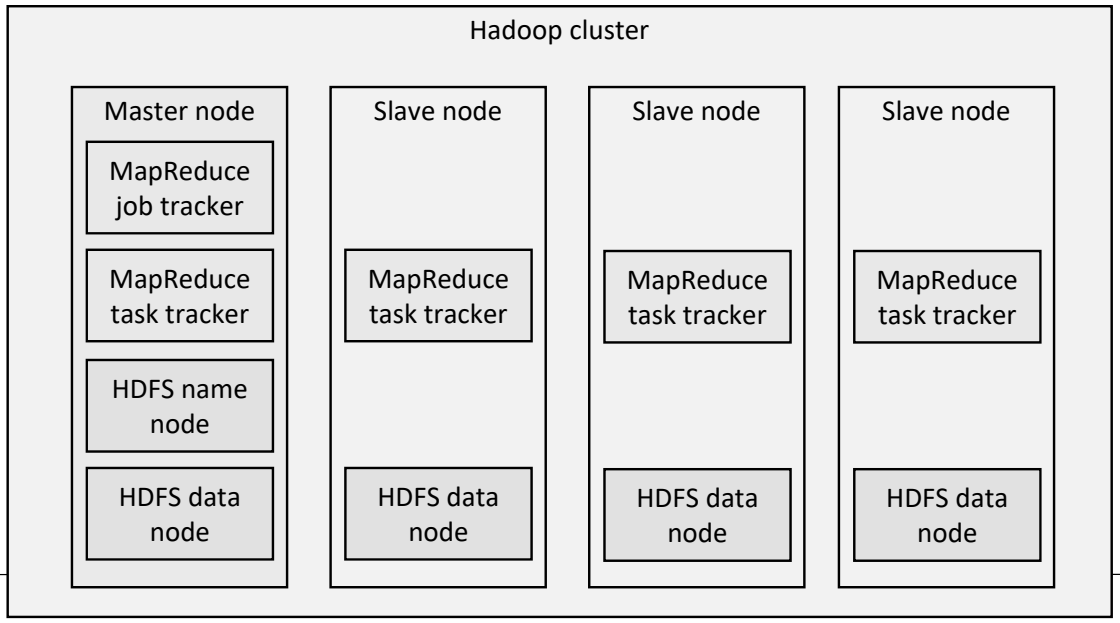
Hadoop

- Large-scale distributed data storage and processing framework
- Designed for horizontal scale on commodity hardware
- Open source (Apache 2.0 license)

Hadoop Timeline



Hadoop Architecture



Hadoop Distributed File System (HDFS)

- Distributed file system written in Java
- Single ı^Œ 'âłß¡' that serves metadata
  - High availability support in Hadoop 2.0
- Multiple ¡^÷^âłß¡'Ñthat serve file blocks
- Designed to cooperate with the job and task trackers – aware of where the data resides
- Hadoop can also work with local files, FTP, Amazon S3, and Azure Storage
- On *nixes, can be mounted (VFS)

MapReduce Platform

- A central –ßlâ±È^uÆ'È node receives jobs from clients and pushes work to ÷^ÑŒâ±È^uÆ'È nodes
- Multiple schedulers manage the distribution and priority of submitted jobs

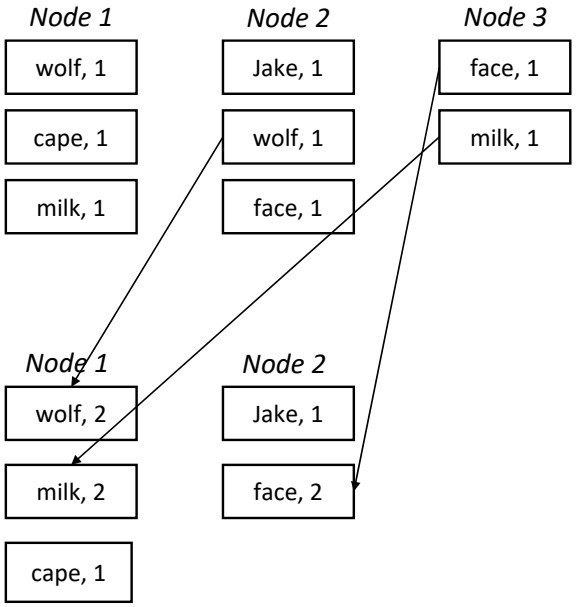
MapReduce

- ? ^½L'¡žu' is a programming model for parallel distributed computing
- ? ^½: each node applies a transformation to its local data é=öâVöíâ→ >Ñ÷é=Äâ VÄí
- For each key K2, all the generated values V2 are Ñ'ž,,ª'¡ to a specific node
- L'¡žu': each key K2 and its values are processed by a specific node to generate a final result
  é=Äâ>Ñ÷éVÄííâ→ >Ñ÷éVÀí

MapReduce Example

```
def map(id, doc):
  for w in doc.split():
    emit((w, 1))

def reduce(word, counts):
  total = 0
  for c in counts:
    total += c
  emit((word, total))
```



MapReduce Disadvantages

- All intermediate steps are persisted to disk
- The shuffle phase can induce a lot of network traffic
- Not every problem can be reduced to a large distributed sort
- Repeated operations on the same data tend to be expensive
- Writing MapReduce jobs can become tedious, fast

Spark

- Another distributed computing framework, not based on MapReduce
  - Revolves around RDDs (È'Ñ°ª°'ı÷â¡ Ñ÷È°ıž÷'¡â¡^÷^Ñ'÷Ñ), forming a distributed shared memory
  - Caching and sharing often improves performance compared to MapReduce, which writes even intermediate results to disk
- One monolithic (but extensible) framework for many needs
- Run workloads 100x faster than MapReduce(based on Logistic Regression test)

Sample Use Cases

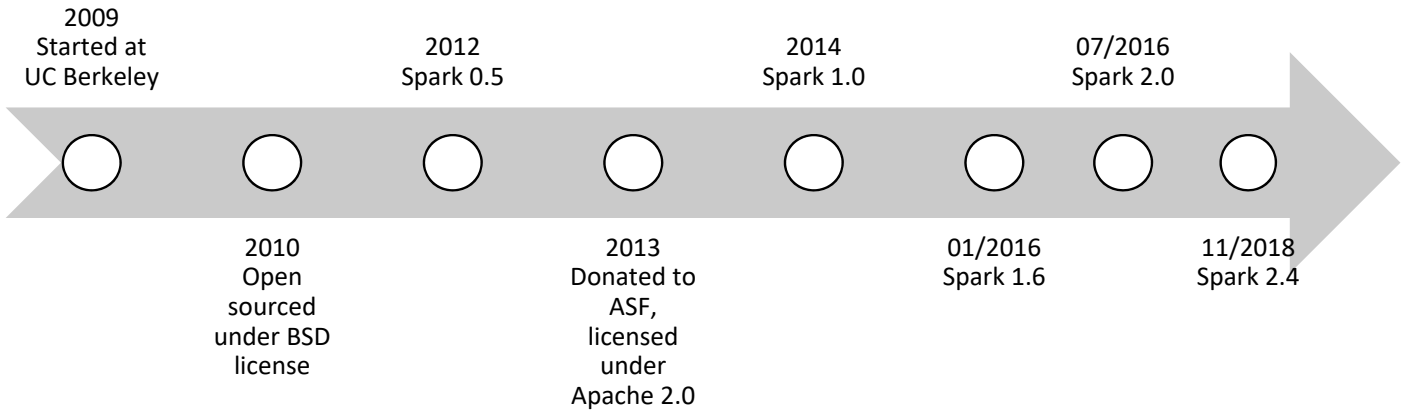
- Interactive analysis (fast!)
- Data streaming, enrichment, analysis
- Machine learning

Sample Case Studies

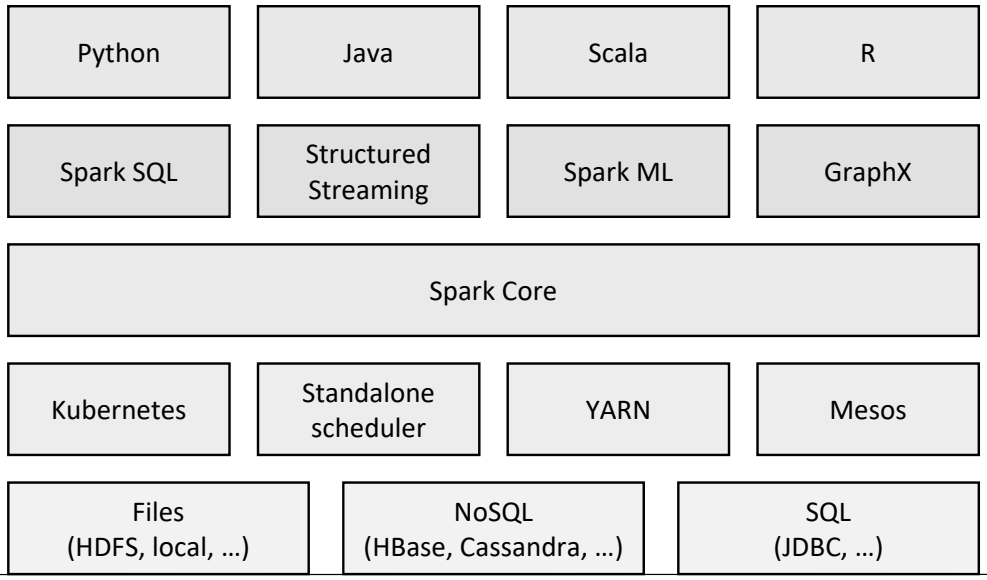
- Uber has a pipeline (Kafka, Spark Streaming, HDFS) for converting terabytes of data from mobile users to structured storage
- Telefonica uses Spark (Kafka, Storm, Spark, Cassandra) to monitor online sources and produce security alerts and reports in real-time
- Yahoo uses machine learning (Spark MLLib) for personalization of news pages

YAHOO! Telefónica

Spark Timeline



Spark Architecture



Spark in the “Big Data” Ecosystem

- L'^ªŽâ¡'Ñ‰ı'¡â±ßâ™ ßÈÆâ™ °÷'â'Ý'ÈŽ÷'°ı‰
- Spark can access all Hadoop file formats, and has built-in support for many of them
- Spark is better-suited for machine learning workloads, can be accessed directly from R
- Spark can query and join relational and non-relational databases
- Spark can ingest streaming data
- Spark can run side-by-side on Hadoop clusters, work with Mesos, YARN and Kubernetes

Major Vendors (Distributions)

- Cloudera (focus on administration and HA)
- MapR (focus on performance and scalability)
- Databricks (focus on management and hosting)

Assorted Sort Benchmarks Results

Year	Platform	Data Size	# Nodes	Time
2006	Hadoop	1.8TB	188	48h
2008	Hadoop	1TB	910	209s
2009	Hadoop	1TB		62s
2009	Hadoop	1PB		17h
2014	Hadoop	100TB	2100	72m
2014	Spark	100TB	206	23m
2014	Spark	1PB	190	234m

Getting Started With Spark

Installation

- Download .tgz from spark.apache.org or pre-packaged distribution (CDH, etc.)
- Bundled with Hadoop or integrates with an existing Hadoop installation

Spark in the Cloud

- The three big cloud providers have support for a fully managed Spark Cluster:
    - Microsoft Azure - HDinsight / Databricks
    - Google GCP - DataProc
    - Amazon AWS - EMR

- This approach reduces the need of maintain and configure a cluster, and also has advantages of security, scalability, global availability and so on.

Spark Distribution Contents

bin	03-May-19 15:43	File folder
conf	03-May-19 15:42	File folder
data	03-May-19 15:43	File folder
examples	03-May-19 15:43	File folder
jars	03-May-19 15:43	File folder
kubernetes	03-May-19 15:42	File folder
licenses	03-May-19 15:42	File folder
python	03-May-19 15:42	File folder
R	03-May-19 15:43	File folder
sbin	03-May-19 15:42	File folder
yarn	03-May-19 15:43	File folder
LICENSE	19-Apr-19 02:31	File
NOTICE	19-Apr-19 02:31	File
README.md	19-Apr-19 02:31	MD File
RELEASE	19-Apr-19 02:31	File

Interactive Shells

- Ñ½^ÈÆ"Ñ' ' ªªis a Scala interactive shell
- ½ŽÑ½^ÈÆ is a Python interactive shell
- Both can be configured to connect to a cluster or run in local mode

`help(x)` And `dir(x)`

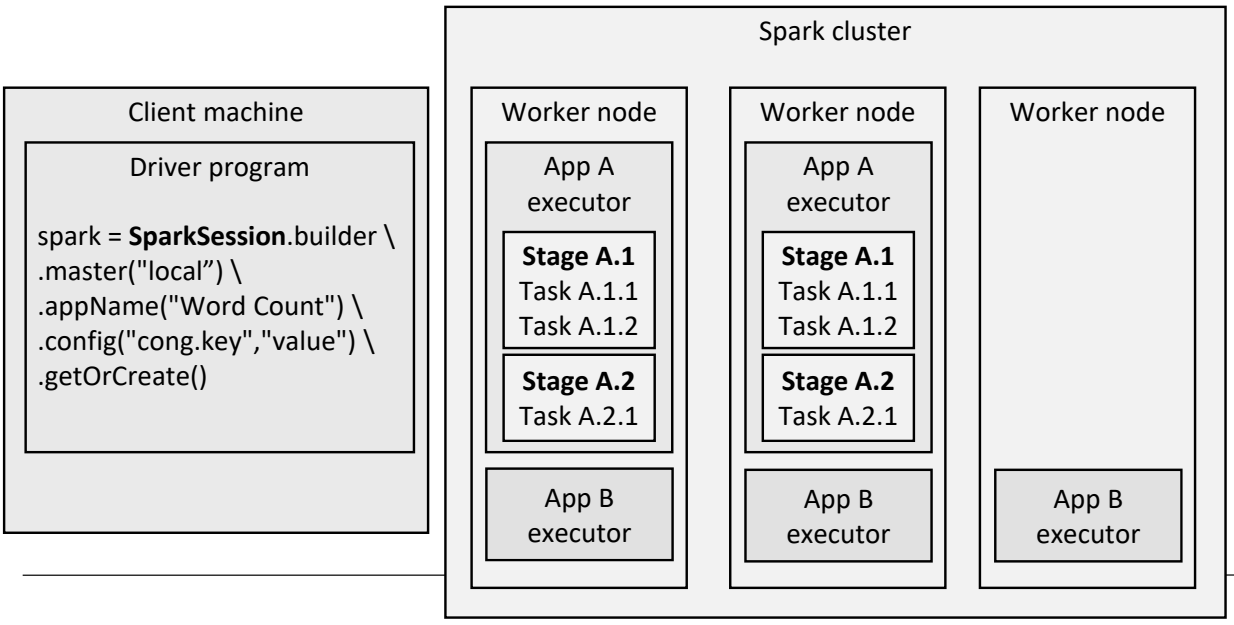
As you're working your way through, ' '½ and ¡ Èare good friends to have:

```
>>> dir(companies)
[…, 'collect', 'filter', 'cache', …]
>>> help(companies.filter)
Help on method filter in module pyspark.sql.dataframe:
filter(condition) method of pyspark.sql.dataframe.DataFrame instance
    Filters rows using the given condition.
    :func:`where` is an alias for :func:`filter`.
    :param condition: a :class:`Column` of :class:`types.BooleanType`
        or a string of SQL expression.
    >>> df.filter(df.age > 3).collect()
    [Row(age=5, name='Bob')]
```

Spark Hello, World

```
>>> rdd = sc.textFile("hdfs:///demo/data/alice.txt")
>>> wc = rdd.flatMap(lambda line: line.split()) \
...         .map(lambda word: (word, 1))        \
...         .reduceByKey(lambda x, y: x + y)    \
...         .sortBy(lambda w: -w[1])
>>> for w, c in wc.take(5):
...     print("%20s appears %-4d times" % (w, c))
                the appears 1664 times
                and appears 780  times
                 to appears 773  times
                  a appears 662  times
                 of appears 596  times
```

Spark Concepts



Spark Concepts cont’

- The , È˝Ý'Èás JVM which runs the application(main method)
- The Driver decide how to ½^È÷˚÷ß1 the data across the executors,
  by creating a Œ^½âover the real location of the data
- The Driver is assigning a partition of data to each ÷^ÑÆ
- Each Task will fetch from the ¡^÷^âÑßžÈı'åthe partition that assigned to it

Spark Standalone Applications

- Initialize your own M½^ÈÆM'ÑÑß1 and run it using the Ñ½^ÈÆ¨ ÑžlŒ ˚÷script
  - Takes command-line arguments that configure the desired cluster, cores, memory, etc.

```
from pyspark import SparkSession
spark = M½^ÈÆM'ÑÑß1().builder().Œ^Ñ÷'È("local") \
                        .^½½@ ^Œ '("wc") \
                        .getOrCreate()
rdd = spark.sparkContext.textFile(...)
...
```

Code and Data

- The data you operate on can be shipped to the executors, or they can access it directly (e.g. HDFS)
- The code of your program is also shipped to the executors

Lab: Multi-File Word Count

- https://github.com/netanel246/spark-workshop

  Lab 1 – Multi-File Word Count

RDD Transformations and Actions

Resilient Distributed Datasets

- An RDD is a distributed collection of elements
- Spark programs consist of uÈ'^÷ı‰ RDDs, ÷È^ıÑ‚ßÈŒ ı‰ RDDs, and computing È'ÑžªÑ from RDDs
- RDDs are distributed across the cluster
- RDDs are the core API of Spark

Creating RDDs

- We will discuss multiple data sources, but for now let's work with two
  - sc here is a M½^ÈÆ*ß1÷'³÷object, initialized by the shell
  - Can be access also by SparkSession.sparkContext
- Strings (lines) from a text file:

```
poem = sc.textFile("sonnet43.txt")
```

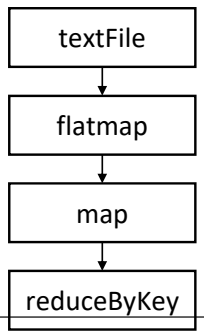
- In-memory collection:

```
nums = sc.parallelize(range(0, 10000))
```

Transformations

- OÈ^1Ñ,ßÈE ^÷ß1Ñconstruct a new RDD from an existing one
  - This forms a ,!5 of RDD transformations, also called the RDD's ª°1'^‰'

```
rdd = sc.textFile("hdfs:///demo/data/alice.txt")
wc  = rdd.flatmap(lambda l: l.split())
          .map(lambda w: (w, 1))
          .reduceByKey(lambda x, y: x + y)
```



Actions

- Transformations are evaluated ≪^ü°Ž; in the preceding example, nothing was run yet!
- !u÷ℬıÑforce evaluation of the job and return some data to the driver program

```
>>> words = sc.textFile(…).flatMap(lambda l: l.split())
>>> words.count()
29465
>>> words.first()
u'Project'
```

Common Transformations and Actions

Type	Syntax	Meaning
Trans	`rdd.map(f)`	Apply **f** to each element of **rdd** and produce a new RDD
Trans	`rdd.filter(f)`	Apply **f** to each element of **rdd** and produce a new RDD with only the elements for which **f** returned **true**
Trans	`rdd.flatMap(f)`	Apply **f** to each element of **rdd**, flatten the produced collections into a new RDD
Trans	`rdd.distinct()`	Return a new RDD with duplicate elements removed
Trans	`rdd.pipe(cmd)`	Pass each partition of **rdd** to a shell command and return a new RDD of strings with the command's stdout lines
Act	`rdd.collect()`	Return all elements in **rdd** to the driver program

Act	`rdd.count()`
	`rdd.take(n)`
	`rdd.first()`
		Return the number of elements in/first **n** elements of/the first element of **rdd**

Act	`rdd.saveAs...()`	Write the elements of **rdd** into a file (strings/serialized)

Pair RDDs

- A ½^ Ëä⊔, , is an RDD of tuples (K, V)
- Pair RDDs are extremely common in Spark, and provide a variety of additional operations
- To create a pair RDD, provide a collection of pairs to M½^ÈÆ*ßı÷'³÷½^È^ªª'ª"ü'éíor ÍE ^½éíover an existing RDD to produce tuples

```
sc.textFile(…).flatMap(lambda line: line.split())
              .map(lambda word: (len(word), word))
```

Common Transformations on Pair RDDs

Syntax	Meaning
`rdd.groupByKey()`	Group values in **rdd** that have the same key and attach them as a list to that key
`rdd.reduceByKey(f)`	Combine values in **rdd** that have the same key
`rdd.mapValues(f)`	Transform the values in **rdd** without changing the keys
`rdd.sortByKey()`	Return a new RDD sorted by key
`rdd.join(other)`	Join **rdd** and **other** on their keys and return a new RDD where each key has two values
`rdd.cogroup(other)`	Group values in **rdd** and **other** that have the same key and attach them as a list to that key
`rdd.subtractByKey(other)`	Return a new RDD that doesn't contain pairs whose keys are present in **other**
`rdd.countByKey()`	Return the count of elements for each key in **rdd** *this is an action, not a transformation*

Example: Pair RDDs

```
def parse(line):
  parts = line.split()
  return (parts[0], parts[1:])


interestingSites = sc.textFile(…)
                     .map(lambda s: (s, 0))    # dummy
sitesAndLinks = sc.textFile(…).map(parse)


# how many links from interesting sites?
interestingSites.join(sitesAndLinks)
                .mapValues(lambda (_, ls): len(ls))
                .collect()
```

Cache and Persistent

- When we cache an RDD, each ıß¡' stores any partitions of it that it computes in memory and reuses them in other actions on that dataset (or datasets derived from it)

  from pyspark import StorageLevel
  df.persist(M÷ßÈ‰'>'Ý'ª(useDisk=True, useMemory=True, useOffHeap=False, deserialized=False, replication=1))

- Cache are a syntactic sugar for ½'ÈÑÑ÷ÆM÷ßÈ‰'>'Ý'ª? -? B LYÓB @ >Yí

DEMO

http://localhost:8888/notebooks/GoogleDrive/Sela-workzone/Courses/SDP_05.19-Spark_Workshop/DEMO%20-%20Cache%20and%20persist.ipynb#

Example: uß Œ l ˚ı ') Ž = ' Žéí

```
>>> words = sc.textFile(…).flatMap(…)                  \
...           .filter(lambda w: w[0].isalpha())       \
...           .map(lambda w: (w[0].lower(), w))

>>> words.combineByKey(                                \
...        lambda w: (len(w), 1),                      \
...        lambda s, w: (s[0]+len(w), s[1]+1),         \
...        lambda s1, s2: (s1[0]+s2[0], s1[1]+s2[1])   \
...      )                                             \
...      .mapValues(lambda (s, c): s/float(c))         \
...      .sortBy(lambda (_, avg): -avg).take(10)
[(u'z', 7.0), (u'p', 6.71), (u'r', 6.68), (u'c', 6.66), (u'e', 6.64),
(u'q', 6.16), (u'g', 6.05), (u'd', 5.77), (u'l', 5.56), (u'f', 5.28)]
```

Loading and Saving RDDs

- There are multiple data sources supported out-of-the-box, and you can add additional providers

- 4°¹âÑŽÑ÷'Œ Ñ
  HDFS, Amazon S3, Azure Storage
- 4°¹â‚ßÈŒ ^÷Ñ
  Text, JSON, Hadoop sequence files, protobuf
- M÷ÈžU÷žÈ'¡â¡^÷^âÑßžÈu'Ñ (with schema)
- MK>â^ı¡â®ßMK>â¡^÷^l^Ñ'Ñ
- http://spark-packages.org is an extensions catalog for Spark

Text Files

- í÷'³÷4°¹éí will load lines from a single file, or all files from a directory
- í™ 'ß¹O'³÷4°¹Ñéí produces a pair RDD with file names as keys and file contents as values
- íÑ^Ý'!ÑO'³÷4°¹éí writes out an RDD to a file

```
>>> sc.parallelize([1,2,3], 4).saveAsTextFile(…)
```

```
$ ls saved_rdd
_SUCCESS  part-00000  part-00001  part-00002  part-00003
$ grep "" part-*          # print non-empty files
part-00000: 1
part-00002: 2
part-00003: 3
```

Unstructured JSON

- Simply load and dump JSON using the built-in –Ñ‰ı module (read file with í÷'³÷4°a'éí first)
- –Ñ‰ıí‰^¡Ñéí and –Ñ‰ıí¡žŒ½Ñéí

```
>>> json.loads('{"name": "Dave", "age": 42}')
{u'age': 42, u'name': u'Dave'}
```

Lab: Analyzing Flight Delays

- https://github.com/netanel246/spark-workshop

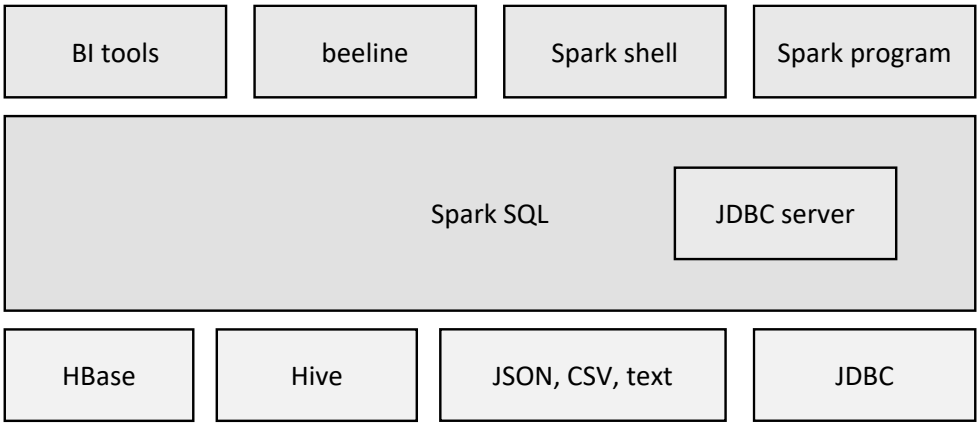
  Lab 2 – Analyzing Flight Delays

Spark SQL

Spark SQL

- So far, we have been working with (mostly) žıÑ÷ÈŽu÷žÈ'¡ data
- Spark SQL is a framework that makes it easier to work with Ñ÷ÈŽu÷žÈ'¡ data
- A DataFrame is a distributed, optimized, columnar collection of structured data and ½'È‚ßÈŒ â‹'÷÷'Èâ÷'^ıâ‹, ,
- Immediately familiar to any SQL user
- More efficient than unstructured storage
  - Partial queries can be satisfied without reading all fields
  - Parts of the query can be pushed down to the engine
  - Multiple aggregations easier to express

Spark SQL Components



Creating DataFrames

- Create M½^ÈÆM'ÑÑ℠ı:

```
spark = M½^ÈÆM'ÑÑ℠ı().builder().Œ ^Ñ÷'È("local") \
                .^½½@ ^Œ '("wc") \
                .getOrCreate()
```

- Load a , ^÷^4È^Œ ' from a data source directly:

```
df = spark .read.json("hdfs://…")
df = spark .read.jdbc("jdbc:postgresql://svr/db",
         "orders", numPartitions=8,
         properties=creds)

df = spark.read.option("inferSchema", "true") \
            .option("header", "false") \
            .csv(“hdfs://..”)
```

Working with Dataframes

- Inspect the DataFrame:

```
>>> df.dtypes #Return df column names and data types
>>> df.show() #Display the content of df
>>> df.first() #Return first row
>>> df.take(2) #Return the first n rows
>>> df.schema #Return the schema of df
>>> df.columns #Return the columns of df
>>> df.count() #Count the number of rows in df
>>> df.printSchema() #Print the schema of df
```

Running SQL Queries

- ítÈ'^÷'BÈL'½ª^u'O'Œ½V°'™ gives your ,^÷^4È^Œ' a temporary name that you can use in queries

```
>>> df.createOrReplaceTempView("flights")
>>> results = spark.sql(
... "select avg(ArrDelay) from flights")
>>> results
DataFrame[_c0: double]
>>> results.first()
42.33317
```

SQL Fluent API

- , ^÷^4Ĕ^Ȇ 's also support a fluent query API, similar to transformations and actions on RDDs

```
>>> flights.select("FlightNum").distinct().count()
6429
>>> flights.groupBy("OriginCityName",
...                 "DestCityName").count().take(3)
[Row(OriginCityName=u'Philadelphia, PA', DestCityName=u'Charlotte, NC',
count=284), Row(OriginCityName=u'Austin, TX', DestCityName=u'Charlotte,
NC', count=61), Row(OriginCityName=u'Juneau, AK', DestCityName=u'Yakutat,
AK', count=30)]
```

SQL Fluent API

- There are a lot of functions that supports the Fluent API: https://spark.apache.org/docs/2.4.0/api/sql/index.html

```
>>> from pyspark.sql.functions import sum,col
>>> companies.select(sum("acquisition.price_amount"))
    .withColumnRenamed("sum(price_amount)", "total_price_amount")
    .first()

>>> companies.select(col("acquisition.acquired_year")).take(10)
>>> companies.withColumn(“mul_price”, companies.acquisition.price_amount *
10)
```

User-Defined Functions

- Register a žÑ'È¨¡',˚ı'¡â˛žıu÷ß1 (UDF) and call it from the MK >*ß1÷'³÷

```
>>> from pyspark.sql.types import *
>>> spark.udf.register("compensation",
...    lambda delay: …, IntegerType())
>>> spark.sql("""select   Carrier,
                          sum(compensation(ArrDelay))
               from       flights
               group by Carrier""").collect()
[Row(Carrier=u'AA', _c1=350400),
 Row(Carrier=u'NK', _c1=89600),
 Row(Carrier=u'HA', _c1=10000)
…]
```

Joins Between Sources

- If you register tables from multiple sources, your queries can reference them

```
>>> spark.sql("""select INV.risk, ACT.name, ACT.balance
                  from INV
                  join ACT
                    on INV.ssid = ACT.ssid
                 where ACT.balance > 100000"""
           ).collect()
```

- NOTE when you tell Spark to load a JDBC URL, it fetches the whole dataset in parallel across nodes; beware of DoS-ing your database server

Write DataFrames

```
>>> df.select("firstName", "city") \
       .write \
       .csv("nameAndCity.csv")

>>> df.select("firstName", "age") \
       .write \
       .format("json") \
       .save("nameAndAge.csv")
```

Lab: Analyzing Startup Companies

- https://github.com/netanel246/spark-workshop

  Lab 3 - Analyzing Startup Companies

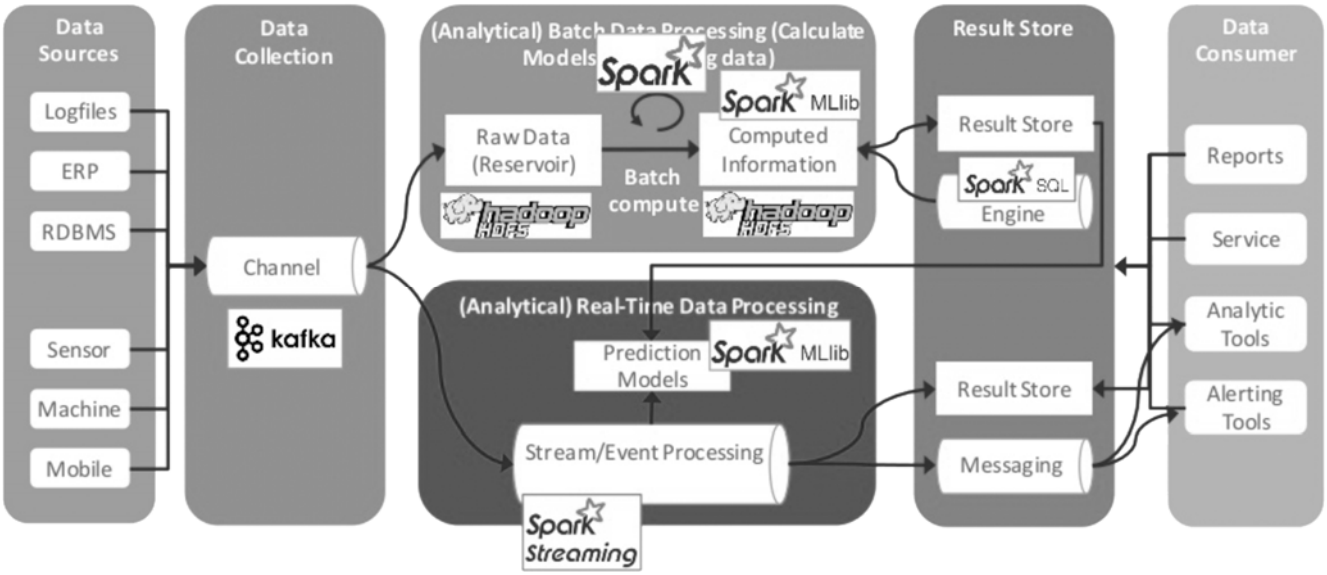
Lab: Analyzing UK Property Prices

- https://github.com/netanel246/spark-workshop

  Lab 4 - Analyzing UK Property Prices

Spark Structured Streaming

Streaming is integrated in your system



Typical use cases for stream processing

- Analyze packets from IoT devices to detect errors
- Perform security checks on data that came from log files
- Predict users in e-commerce and offer them discounts
- Display real-time dashboard
- Providing alerts on critical devices

Spark streaming is evolving

- Spark streaming started has a spin of to batch processing with micro batching.
- A the beginning we had Dstreams.
- Now we have structure streaming
- Spark 2.4 provides us continuous processing

Dstream in spark

- A DStream is a sequence of RDDs, one for each time interval
- Provides RDD operations and special window-related operations
- Simple, predictable set of APIs for both streaming and offline analysis scenarios

```
sctx = StreamingContext(sc, batchDuration=60)
ds = sctx.textFileStream("hdfs:///logs/www")
respCodes = ds.map(lambda line: line.split()[8])     \
              .map(lambda code: (code, 1))
respCodes.reduceByKeyAndWindow(
  lambda x, y: x + y,        # reducer
  lambda x, y: x - y,        # “anti-reducer”
  360,                              # window duration
  180                               # slide duration
).pprint()
sctx.start()
Sctx.awaitTermination()
```

The problems with DStream

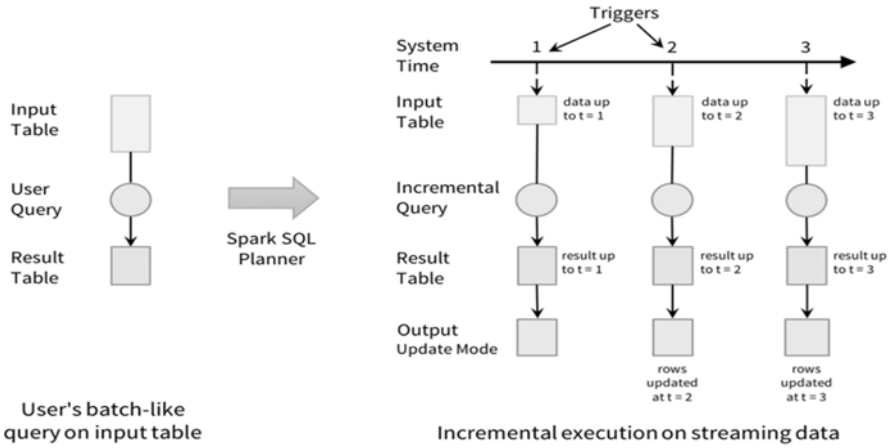
- Not the same code for stream and batch processing
  - You need to activate the stream
  - You need to manage state
  - You need to manage windows
  - You need to manage checkpoints
- Late data is not handled
- Working with data frame is much more optimized!

Structure streams

The simplest way to perform stream analytics is not having to reason about streaming at all

---

Structure streaming model



Structured Streaming Processing Model

Users express queries using a batch API; Spark incrementalizes them to run on streams

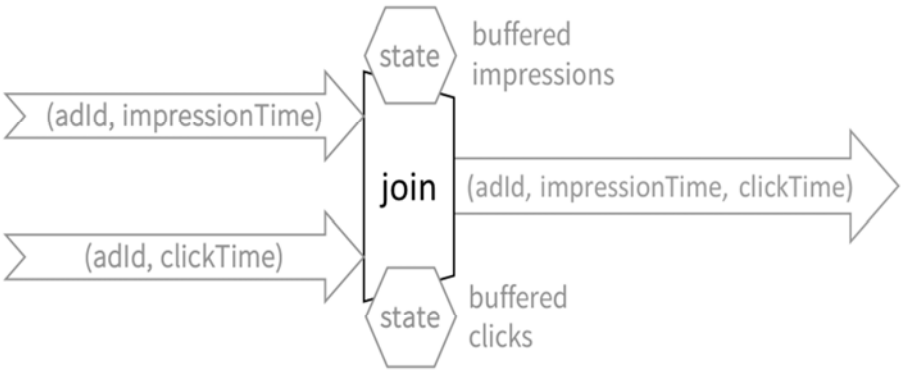
Output modes

- Complete mode – save all the data in external data to perform actions like distinct
- Append mode – perform calculation only on data that appended and not remembering the historical data – like count
- Update mode – saving all the data like complete and can update changes of historical data

Watermarking

- Let's say we wish to count all the events that came in the last hour, can we calculate the result right now?
- Like in any other place in life we need to handle late arrivals
- Spark structure streaming enables you to define the watermarks for closing the window for late arrivals in simple api.

Stream join

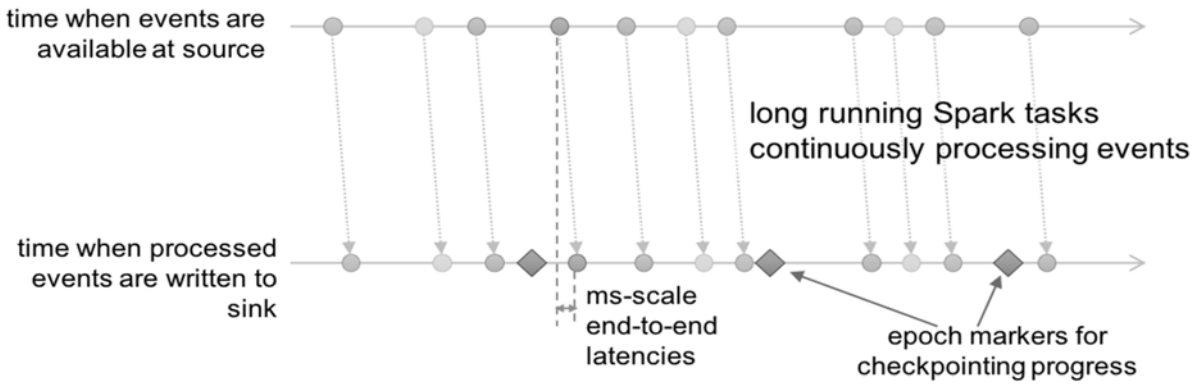


Stream-stream join use case: Ad-Monetization (joining ad clicks to impressions)

Spark streaming – continuous processing

Continuous processing

Experimental Mode



Millisecond-scale end-to-end latencies with Continuous Processing

Continuous processing in numbers

- Working with structure streaming micro batching has latency of 100ms.
- Working with structure streaming and continuous processing has latency of 1ms
- Only at least one guarantee.
- SQL and Map like data set operation are available

Spark Streaming

---

1 Convert to Structured Streaming
Netanel Malka, 03/05/2019

Spark Streaming

- So far, we worked with data that was relatively static; many applications need to process data when it arrives
- Spark Streaming provides discretized streams (, M÷È' ^Œ Ñ) that represent a series of batched data
- A DStream is a sequence of RDDs, one for each time interval
- Provides RDD operations and special window-related operations
- Simple, predictable set of APIs for both streaming and offline analysis scenarios

Creating DStreams

- First, create a M÷È'ˆŒ ˚ı‰*ßı÷'³÷ that specifies the batch duration (in seconds):

```
streaming = StreamingContext(sc)
```

- DStream from a socket receiving lines of text:

```
ds = streaming.socketTextStream("localhost", 99)
```

- DStream from an HDFS directory of text files:

```
ds = streaming.textFileStream("hdfs:///logs")
```

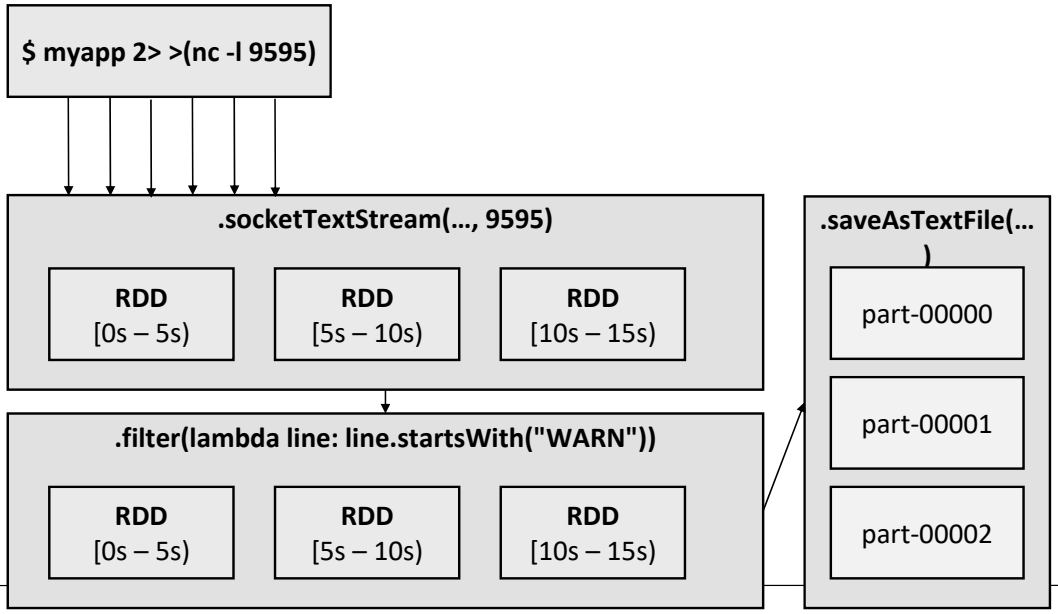
  - NOTE files must be atomically moved into the directory

Processing DStreams

- Transformations (such as Œ ˆ½, ,ª÷'È) on DStreams produce new DStreams
  - Use ı÷Èˆı Ñ,ßÈŒ éı to reuse operations on RDDs
- Actions produce output from a single batch

```
streaming.socketTextStream(…)                     \
         .flatMap(lambda line: line.split())     \
         .map(lambda w: (w, 1))                  \
         .reduceByKey(lambda x, y: x + y)        \
         .pprint()       # action!
streaming.start()
streaming.awaitTermination()
```

DStreams



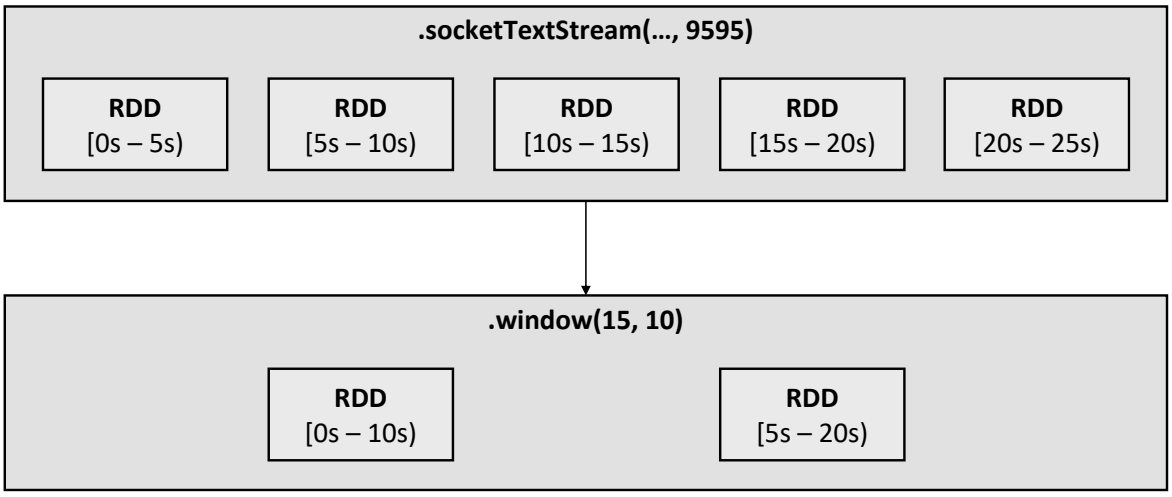
Processing and Checkpointing

- launches that create and replicate RDDs from the stream data
- When the interval elapses, they create short jobs to process the RDDs
- Stateful work (discussed next) requires checkpointing (or file)
- Restart from checkpoint with and recompute lost RDDs

Sliding Window

- Multiple batches from the stream can be combined to a sliding window
- Operations can be performed on the window and not the individual batches:
  - ÍÈ'¡žu')ŽW ˚ı¡ß™éí, ÍÈ'¡žu')Ž='Ž!ı¡W ˚ı¡ß™éí
  - Has an optimization for reductions where you provide a function for sliding a value into and out of the window
  - íußžı÷)ŽW ˚ı¡ß™éí, íußžı÷)ŽV^ªž'!ı¡W ˚ı¡ß™éí

Sliding Window Illustrated



Example: Windowed Visitor Log

```
# The input stream is a standard www server log
# 10.1.0.4 - - [1/Jan/2023:18:03:14 +0300] "GET /index.html HTTP/1.1" 200 89446

sctx = StreamingContext(sc, batchDuration=60)

ds = sctx.textFileStream("hdfs:///logs/www")

respCodes = ds.map(lambda line: line.split()[8])     \
              .map(lambda code: (code, 1))

respCodes.reduceByKeyAndWindow(
  lambda x, y: x + y,            # reducer
  lambda x, y: x - y,            # "anti-reducer"
  360,                           # window duration
  180                            # slide duration
).pprint()
```

Maintaining State

- updateStateByKey lets you maintain and update state per key
- You provide a function that updates the old state based on the new events and returns the new state

```
ds = streaming.socketTextStream(…)

ds.flatMap(lambda line: line.split())      \
  .map(lambda w: (w, 1))                   \
  .updateStateByKey(
       lambda events, count: (count or 0) + len(events)
 ).pprint()
```

Lab: Tweet Streaming

- https://github.com/goldshtn/spark-workshop
  Lab 5 – Streaming Tweet Analysis