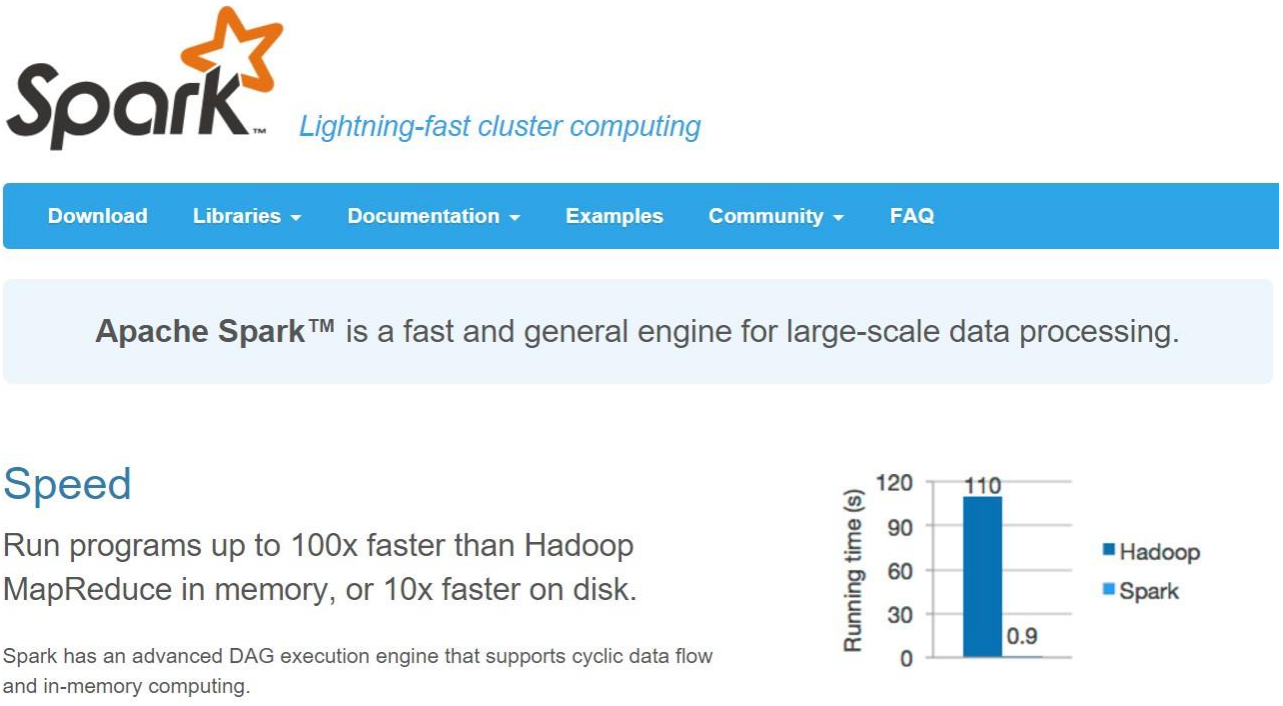
What is Spark?

- Apache Spark is a fast and general engine for large-scale data processing
- Written in Scala – Functional programming language that runs in a JVM
- Spark shell – Interactive—for learning, data exploration, or ad hoc analytics – Python or Scala
- Spark applications – For large scale data processing – Python, Scala, or Java



Reference: http://spark.apache.org

MapReduce VS Spark

How does MapReduce work and what is its constraint?

- In MapReduce, all the data is written back to the physical storage medium after each operation. This means that it reads data from the disk and once this is completed, it writes the data back to the disk.
- This process becomes inefficient when we need low latency. Since it has to read all the data from disk at the beginning of each stage of the process, it is very time consuming. Data Sharing is Slow in MapReduce
- Unfortunately, in most current frameworks, the only way to reuse data between computations (Ex – between two MapReduce jobs) is to write it to an external table storage system (Ex – HDFS). Although this framework provides numerous abstractions for accessing a cluster's computational resources, users still want more.
- Both Iterative and Interactive applications require faster data sharing across parallel jobs. Data sharing is slow in MapReduce due to replication, serialization, and disk IO. Regarding storage system, most of the Hadoop applications, they spend more than 90% of the time doing HDFS read-write operations

Apache Spark Stack

Spark provides a stack of libraries built on core Spark

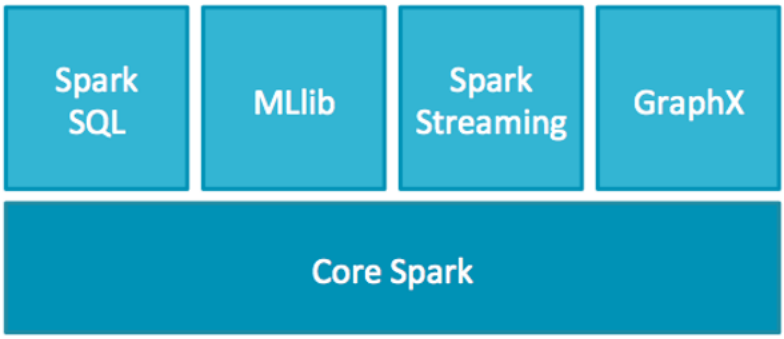
— Core Spark provides the fundamental Spark abstraction: Resilient Distributed Datasets (RDDs)

— Spark SQL works with structured data

— MLlib supports scalable machine learning

— Spark Streaming applications process data in real time

— GraphX works with graphs and graph-parallel computation



Spark's Language APIs

Spark's language APIs allow you to run Spark code from other languages

Scala

Spark is primarily written in Scala, making it Spark's "default" language.

Java

Even though Spark is written in Scala, Spark's authors have been careful to ensure that you can write Spark code in Java.

Python

Python supports nearly all constructs that Scala supports.

SQL

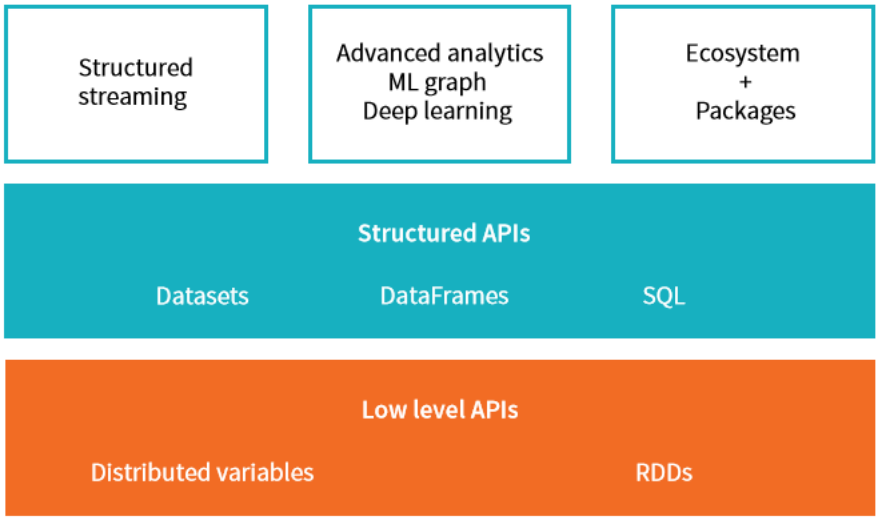
Spark supports ANSI SQL 2003 standard. This makes it easy for analysts and non-programmers to leverage the big data powers of Spark.

R

Spark has two commonly used R libraries, one as a part of Spark core (SparkR) and another as a R community driven package (sparklyr).

A Tour of Spark's Tool sets and Spark's APIs:

Spark is composed of the simple primitives, the lower level APIs and the Structured APIs, then a series of "standard libraries" included in Spark.



While Spark is available from a variety of languages, what Spark makes available in those languages is worth mentioning.

- RDD Programming Guide: overview of Spark basics - RDDs (core but old API), accumulators, and broadcast variables
- Spark SQL, Datasets, and DataFrames: processing structured data with relational queries (newer API than RDDs)
- Structured Streaming: processing structured data streams with relation queries (using Datasets and DataFrames, newer API than DStreams)
- Spark Streaming: processing data streams using DStreams (old API)
- MLlib: applying machine learning algorithms
- GraphX: processing graphs

Spark has two fundamental sets of APIs: the low level "Unstructured" APIs and the higher level Structured APIs.

Unstructured API: RDD was the primary user-facing API in Spark since its inception. At the core, an RDD is an immutable distributed collection of elements of your data, partitioned across nodes in your cluster that can be operated in parallel with a low-level API that offers transformations and actions.

Structured API: DataFrames, Datasets & SparkSql

Like an RDD, a **DataFrame** is an immutable distributed collection of data. Unlike an RDD, data is organized into named columns, like a table in a relational database. Designed to make large data sets processing even easier, DataFrame allows developers to impose a structure onto a distributed collection of data, allowing higher-level abstraction; it provides a domain specific language API to manipulate your distributed data; and makes Spark accessible to a wider audience, beyond specialized data engineers.

Datasets:

Dataset takes on two distinct APIs characteristics: a strongly-typed API and an untyped API. Conceptually, consider DataFrame as an alias for a collection of generic objects Dataset[Row], where a Row is a generic untyped JVM object.

Dataset, by contrast, is a collection **of strongly-typed JVM objects**, dictated by a **case class** you define in Scala or a class in Java.

Spark Shell and Spark Applications

The Spark Shell

- **The Spark shell provides an interactive Spark environment**
  - Often called a *REPL*, or Read/Evaluate/Print Loop
  - For learning, testing, data exploration, or ad hoc analytics
  - You can run the Spark shell using either Python or Scala
- **You typically run the Spark shell on a gateway node**

Starting the Spark Shell

- **On a Cloudera cluster, the command to start the Spark 2 shell is**
  - `pyspark2` for Python
  - `spark2-shell` for Scala
- **The Spark shell has a number of different start-up options, including**
  - `master`: specify the cluster to connect to
  - `jars`: Additional JAR files (Scala only)
  - `py-files`: Additional Python files (Python only)
  - `name`: the name the Spark Application UI uses for this application
    - Defaults to `PySparkShell` (Python) or `Spark shell` (Scala)
  - `help`: Show all the available shell options

```
$ pyspark2 --name "My Application"
```

```
C:\spark\bin>spark-shell2.cmd
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 setLogLevel(newLevel).
18/04/07 20:52:38 WARN NativeCodeLoader: Unable to load native-hadoop library for your platfor
m... using builtin-java classes where applicable
Spark context Web UI available at http://192.168.56.1:4040
Spark context available as 'sc' (master = local[*], app id = local-1523114559504).
Spark session available as 'spark'.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 2.2.1
      /_/

Using Scala version 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_161)
Type in expressions to have them evaluated.
Type :help for more information.
```

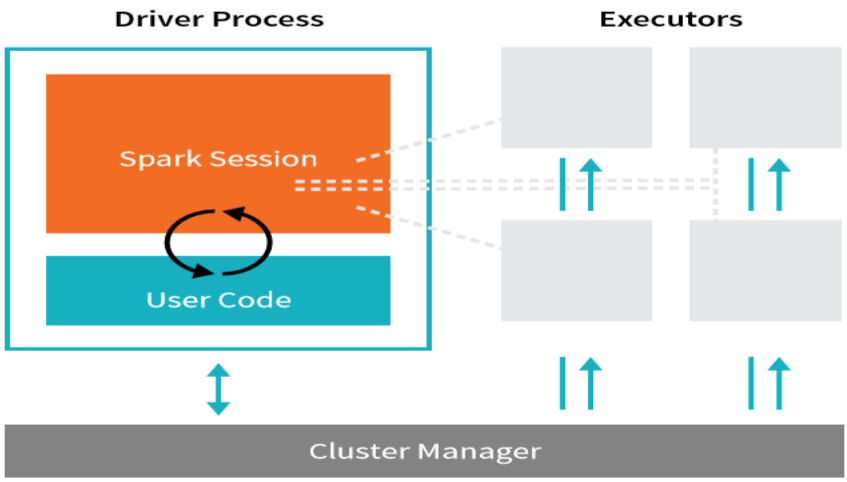
```
C:\spark\bin>pyspark2.cmd
Python 3.6.1 (v3.6.1:69c0db5, Mar 21 2017, 17:54:52) [MSC v.1900 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
18/04/07 22:05:16 WARN NativeCodeLoader: Unable to load native-hadoop library for your platfor
m... using builtin-java classes where applicable
18/04/07 22:05:26 WARN ObjectStore: Failed to get database global_temp, returning NoSuchObject
Exception
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 2.2.1
      /_/

Using Python version 3.6.1 (v3.6.1:69c0db5, Mar 21 2017 17:54:52)
SparkSession available as 'spark'.
>>>
```

```
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
18/04/07 22:09:26 WARN NativeCodeLoader: Unable to load native-hadoop library for your platfo
rm... using builtin-java classes where applicable
18/04/07 22:09:34 WARN ObjectStore: Failed to get database global_temp, returning NoSuchObjec
tException

 Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version  2.2.1
      /_/

 SparkSession available as 'spark'.
>
```

Spark Applications

Spark Applications consist of a driver process and a set of executor processes. The driver process runs your main() function, sits on a node in the cluster, and is responsible for three things: maintaining information about the Spark Application; responding to a user's program or input; and analyzing, distributing, and scheduling work across the executors (defined momentarily). The driver process is absolutely essential - it's the heart of a Spark Application and maintains all relevant information during the lifetime of the application.

The executors are responsible for actually executing the work that the driver assigns them. This means, each executor is responsible for only two things: executing code assigned to it by the driver and reporting the state of the computation, on that executor, back to the driver node.



The cluster manager controls physical machines and allocates resources to Spark Applications. This can be one of several core cluster managers: Spark's standalone cluster manager, YARN, or Mesos.

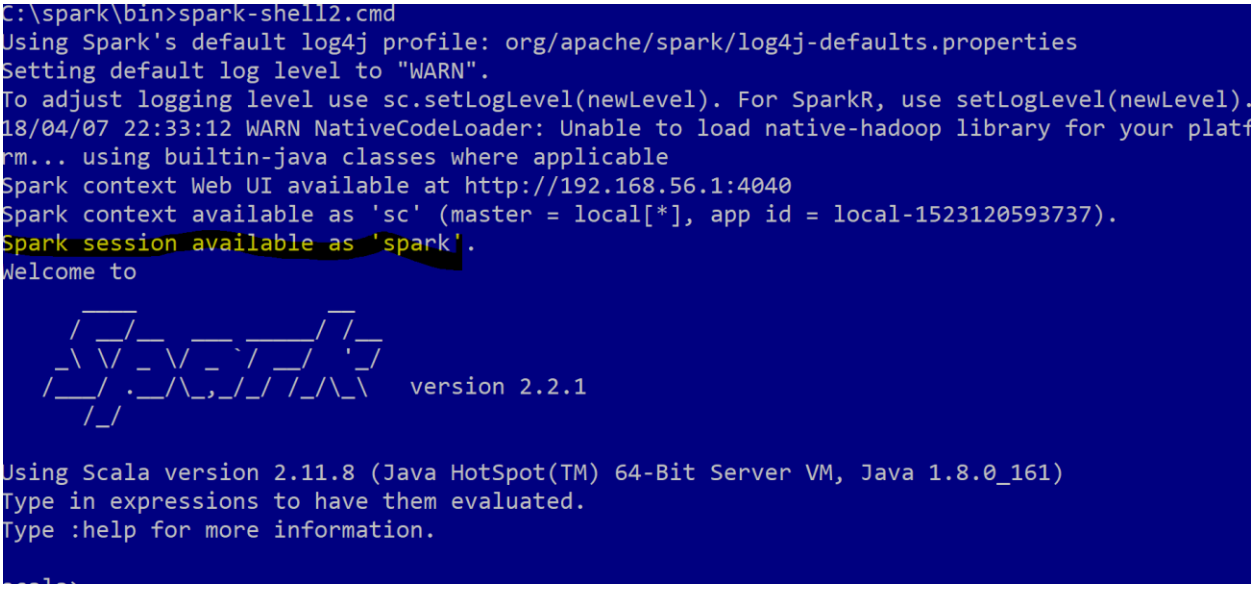
As a short review of Spark Applications, the key points to understand at this point are that:

• Spark has some cluster manager that maintains an understanding of the resources available.

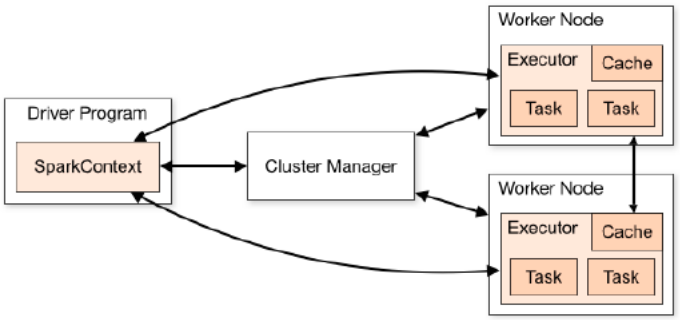
• The driver process is responsible for executing our driver program's commands across the executors in order to complete our task.

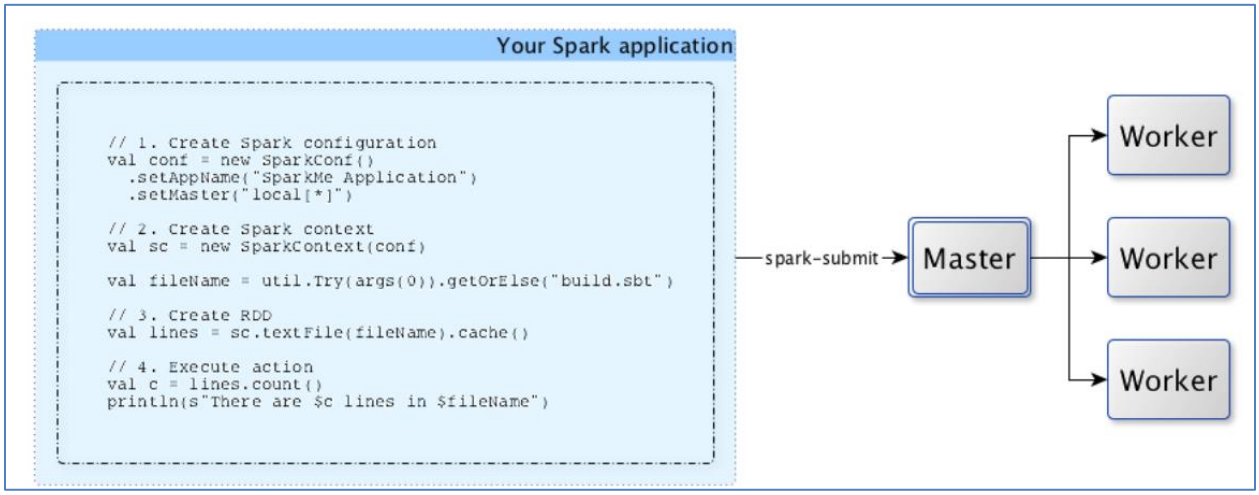
Working with Spark Session

▪ The main entry point for the Spark API is a Spark session
▪ The Spark shell provides a preconfigured SparkSession object called spark
▪ The SparkSession class provides functions and attributes to access all of Spark functionality
▪ Examples include
— sql: execute a Spark SQL query
— catalog: entry point for the Catalog API for managing tables
— read: function to read data from a file or other data source
— conf: object to manage Spark configuration settings
— SparkContext : entry point for core Spark API



SparkContext & SparkSession





As shown in the diagram, a SparkContext is a conduit to access all Spark functionality; only a single SparkContext exists per JVM. The Spark driver program uses it to connect to the cluster manager to communicate, submit Spark jobs and knows what resource manager (YARN, Mesos or Standalone) to communicate to. It allows you to configure Spark configuration parameters. And through SparkContext, the driver can access other contexts such as SQLContext, HiveContext, and StreamingContext to program Spark.

Example: Refer SparkSession.rtf

SparkSession Encapsulates SparkContext

Each language API will maintain the same core concepts that we described above. There is a **SparkSession** available to the user, the SparkSession will be the entrance point to running Spark code. When using Spark from a Python or R, the user never writes explicit JVM instructions, but instead writes Python and R code that Spark will translate into code that Spark can then run on the executor JVMs

```
import org.apache.spark.sql.SparkSession
val sparkSession = SparkSession.builder
  .master("local")
  .appName("my-spark-app")
  .config("spark.some.config.option", "config-value")
  .getOrCreate()
import org.apache.spark.sql.SparkSession sparkSession: org.apache.spark.sql.S
parkSession = org.apache.spark.sql.SparkSession@46d6b87c
```

Anatomy of Spark Job

Job Submission

Spark job is submitted automatically as soon as action is performed on RDD. Internally it call runJob() on SparkContext which will pass the call to scheduler. Scheduler is made of two parts:

DAG scheduler: that breaks down the job into DAG of stages and

Task scheduler: that is responsible for submitting the task from each stage to the cluster

DAG scheduler construct:

Shuffle map tasks:

Each shuffle map task run on one RDD partition. Based on the partition function it writes output to new set of partitions

Result task:

Each results task runs a computation on its RDD partition then send results back to the driver, and the driver assembles the results from each partition into a final results

DAG scheduler is responsible for splitting stages into tasks for submission to the task scheduler

Task Scheduling

When the task scheduler is sent a set of tasks, it uses its list of executors that are running for the application and construct mapping of task to executor

Executor first assign task to local process then node-level then rack level and finally cluster level

Task Execution:

Executor first makes sure that jar and file dependencies are copied to local

Executor and Cluster Manager

Local

In the local mode there is single executor running in the same JVM as driver. The Master url for local mode is local [n]

Standalone

Simple distributed implementation that runs a single Spark Master and one or more workers. When a Spark Application starts, the master will ask workers to spawn executor process on behalf of the application

Master url is spark://localhost:7077

Spark on YARN

Running Spark on YARN provides the tightest integration with other Hadoop components. YARN has resource manager used in HADOOP

Each running Spark application corresponds to an instance of a YARN application and each executor runs in its own YARN container. The Master URL is yarn-client or yarn-cluster

Yarn client mode is required for programs that have any interactive component, such as spark-shell or pyspark

Yarn cluster mode is for batch mode and useful for production deployment

Syntax for submitting of Spark Job in batch mode

```
./bin/spark-submit \
  --class <main-class>
  --master <master-url> \
  --deploy-mode <deploy-mode> \
  --conf <key>=<value> \
  ... # other options
```

```
  <application-jar> \
  [application-arguments]
```

Some of the commonly used options are:

--class: The entry point for your application (e.g. org.apache.spark.examples.SparkPi)

--master: The master URL for the cluster (e.g. spark://23.195.26.187:7077)

--deploy-mode: Whether to deploy your driver on the worker nodes (cluster) or locally as an external client (client) (default: client) †

--conf: Arbitrary Spark configuration property in key=value format. For values that contain spaces wrap “key=value” in quotes (as shown).

application-jar: Path to a bundled jar including your application and all dependencies. The URL must be globally visible inside of your cluster, for instance, an hdfs:// path or a file:// path that is present on all nodes.

application-arguments: Arguments passed to the main method of your main class, if any

Examples:

 # Run application locally on 8 cores

```
./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master local[8] \
  /path/to/examples.jar \
  100
```

 # Run on a Spark standalone cluster in client deploy mode

```
./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master spark://207.184.161.138:7077 \
  --executor-memory 20G \
  --total-executor-cores 100 \
  /path/to/examples.jar \
  1000
```

 # Run on a Spark standalone cluster in cluster deploy mode with supervise

```
./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master spark://207.184.161.138:7077 \
  --deploy-mode cluster
  --supervise
  --executor-memory 20G \
  --total-executor-cores 100 \
  /path/to/examples.jar \
  1000
```

Note: Spark standalone cluster with cluster deploy mode, you can also specify --supervise to make sure that the driver is automatically restarted if it fails with non-zero exit code

 # Run on a YARN cluster

```
export HADOOP_CONF_DIR=XXX

./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master yarn \
  --deploy-mode cluster \  # can be client for client mode
  --executor-memory 20G \
  --num-executors 50 \
  /path/to/examples.jar \
  1000
```

 # Run a Python application on a Spark standalone cluster

```
./bin/spark-submit \
  --master spark://207.184.161.138:7077 \
  examples/src/main/python/pi.py \
  1000
```

The master URL passed to Spark can be in one of the following formats:

Master URL: Meaning

Local: Run Spark locally with one worker thread (i.e. no parallelism at all).

local[K]: Run Spark locally with K worker threads (ideally, set this to the number of cores on your machine).

local[*]: Run Spark locally with as many worker threads as logical cores on your machine.

spark://HOST:PORT: Connect to the given Spark standalone cluster master. The port must be whichever one your master is configured to use, which is 7077 by default.

Yarn: Connect to a YARN cluster in client or cluster mode depending on the value of --deploy-mode. The cluster location will be found based on the HADOOP_CONF_DIR or YARN_CONF_DIR variable.

yarn-client: Equivalent to yarn with --deploy-mode client, which is preferred to `yarn-client`

yarn-cluster: Equivalent to yarn with --deploy-mode cluster, which is preferred to `yarn-cluster`

Yarn-client

In YARN client mode, the interaction with YARN starts when a new SparkContext instance is constructed by the driver programs.

- The context submits a YARN application to the YARN resource manager

- YARN RM start a YARN container on a Node Manager in the cluster and runs SparkExecutorLauncher application Master in it.

- The job of ExecutorLauncher is to start executors in YARN container, which it does by requesting resources from the RM

Then launching ExecutorBackend process as the container are allocated to it

Yarn-cluster

In YARN cluster mode, the users' driver program runs in a YARN application master process.

- The spark-submit client will launch the YARN application but it does not run any user code.

- Application Master starts the driver program before allocating resources for the executor

Spark Cluster Options (1)

- **Spark applications can run on these types of clusters**
  - Apache Hadoop YARN
  - Apache Mesos
  - Spark Standalone
- **They can also run locally instead of on a cluster**
- **The default cluster type for the Spark 2 Cloudera add-on service is YARN**
- **Specify the type or URL of the cluster using the `master` option**

Spark Cluster Options (2)

- **The possible values for the `master` option include**
  - `yarn`
  - `spark://masternode:port` (Spark Standalone)
  - `mesos://masternode:port` (Mesos)
  - `local[*]` runs locally with as many threads as cores (default)
  - `local[n]` runs locally with *n* threads
  - `local` runs locally with a single thread

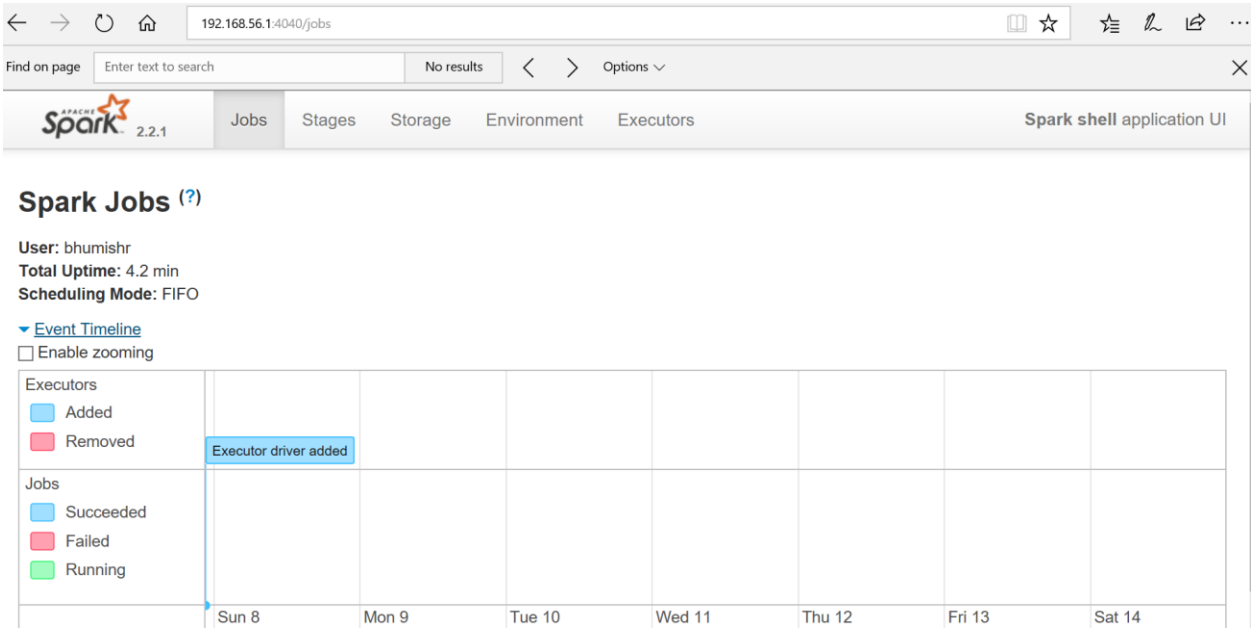
```
$ pyspark2 --master yarn
```

Language: *Python*

```
$ spark2-shell --master yarn
```

Language: *Scala*

Debugging and Monitoring Spark Application:



Scalable language

Scala is a **modern multi-paradigm*** programming language designed to express **common** programming **patterns** in a **concise**, **elegant**, and **type-safe** way.

**In the context of programming, "paradigm" often refers to object-oriented, procedural, functional, logical, etc. So "multi paradigm" would be referring to something as relating to features of several paradigms*

Introduction to Scala

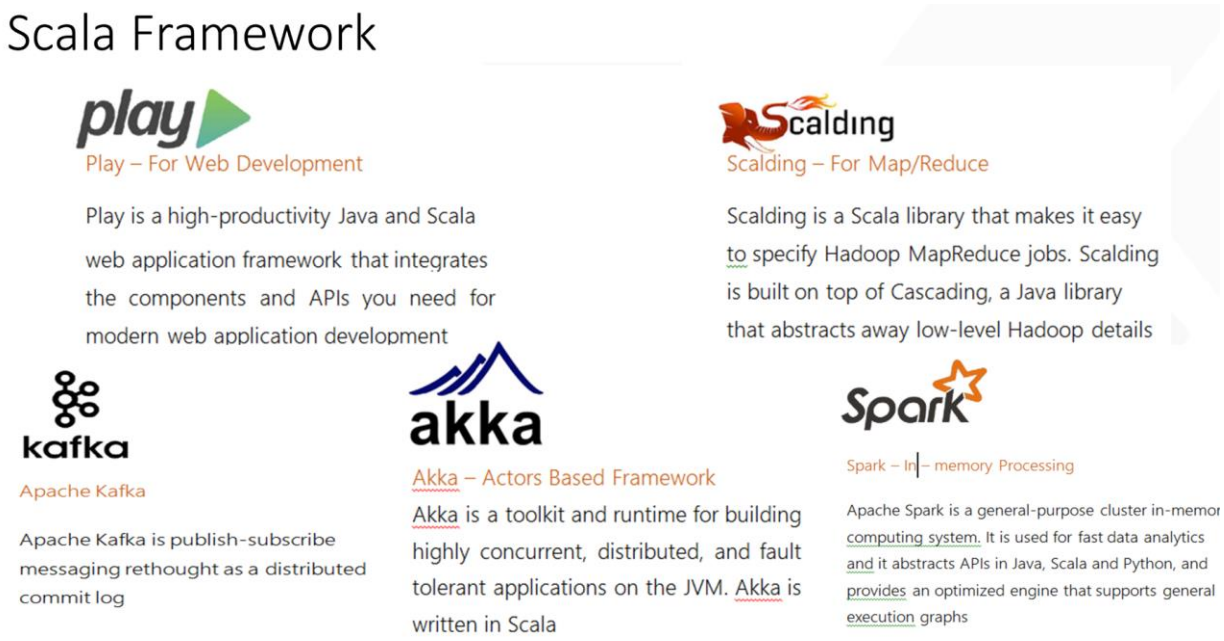
- ✓ Martin Odersky and his team started developing Scala in 2001
- ✓ Scala is a general purpose programming language, multi paradigm object oriented, functional, scalable
- ✓ Aimed to implement common programming patterns in a concise, elegant, and type-safe way
- ✓ Supports both object-oriented and functional programming styles, thus helping programmers to be more productive
- ✓ Publicly released in January 2004 on the JVM platform and a few months later on the .NET platform
- ✓ Scala is Statically Typed (Statically typed language binds the type to a variable for its entire scope)
- ✓ Dynamically typed languages bind the type to the actual value referenced by a variable .Example : python
- ✓ Fully supports Object Oriented Programming i.e. everything is an object in Scala
- ✓ Unlike Java, Scala does not have primitives
- ✓ Supports “static” class members through Singleton Object Concept
- ✓ Improved support for OOP through Traits

Why Scala?

- ✓ Scala is pure object-oriented language. Conceptually, every value is an object and every operation is a method-call
- ✓ Scala is also a functional language and supports immutable data structures
- ✓ Many big data technologies use Scala like Spark, Kafka, Storm, Akka, Scalding and web frameworks like Play

Java Code	Scala Code

`List<String> list = new ArrayList<String>();`
`list.add("1");`
`list.add("2");`
`list.add("3");`
	`val list = List("1", "2", "3")`



Download and Install Scala

Latest version can be downloaded from: http://www.scala-lang.org/download/

Install the Scala and Set the Scala Path in Machine

For Scala Crash Course refer following materials:

LearningScala1.sc
LearningScala2.sc
LearningScala3.sc
LearningScala4.sc

Spark RDD

Resilient Distributed Datasets (RDDs)

- **RDDs are part of core Spark**
- ***Resilient Distributed Dataset* (RDD)**
  - *Resilient*: If data in memory is lost, it can be recreated
  - *Distributed*: Processed across the cluster
  - *Dataset*: Initial data can come from a source such as a file, or it can be created programmatically
- **Despite the name, RDDs are *not* Spark SQL `Dataset` objects**
  - RDDs predate Spark SQL and the DataFrame/Dataset API

RDD Data Types

- **RDDs can hold any serializable type of element**
  - Primitive types such as integers, characters, and booleans
  - Collections such as strings, lists, arrays, tuples, and dictionaries (including nested collection types)
  - Scala/Java Objects (if serializable)
  - Mixed types
- **Some RDDs are specialized and have additional functionality**
  - Pair RDDs
    - RDDs consisting of key-value pairs
  - Double RDDs
    - RDDs consisting of numeric data

Creating RDDs from Files

- **Use `SparkContext` object, not `SparkSession`**
  - `SparkContext` is part of the core Spark library
  - `SparkSession` is part of the Spark SQL library
  - One Spark context per application
  - Use `SparkSession.sparkContext` to access the Spark context
    - Called `sc` in the Spark shell
- **Create file-based RDDs using the Spark context**
  - Use `textFile` or `wholeTextFiles` to read text files
  - Use `hadoopFile` or `newAPIHadoopFile` to read other formats
    - The Hadoop "new API" was introduced in Hadoop .20
    - Spark supports both for backward compatibility

Creating RDDs from Text Files (1)

- **`SparkContext.textFile` reads newline-terminated text files**
  - Accepts a single file, a directory of files, a wildcard list of files, or a comma-separated list of files
  - Examples
    - `textFile("myfile.txt")`
    - `textFile("mydata/")`
    - `textFile("mydata/*.log")`
    - `textFile("myfile1.txt,myfile2.txt")`

```
myRDD = spark.sparkContext.textFile("mydata/")
```

Language: *Python*

Creating RDDs from Text Files (2)

- `textFile` maps each line in a file to a separate RDD element
  - Only supports newline-terminated text

```
myRDD = spark. \
  sparkContext. \
  textFile("purplecow.txt")
```

Language: *Python*

```
I've never seen a purple cow.\n
I never hope to see one;\n
But I can tell you, anyhow,\n
I'd rather see than be one.\n
```

myRDD

I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.

Multi-line Text Elements

- `textFile` maps each line in a file to a separate RDD element
  - What about files with a multi-line input format, such as XML or JSON?
- Use `wholeTextFiles`
  - Maps entire contents of each file in a directory to a single RDD element
  - Works only for small files (element must fit in memory)

user1.json

```
{
 "firstName":"Fred",
 "lastName":"Flintstone",
 "userid":"123"
}
```

user2.json

```
{
 "firstName":"Barney",
 "lastName":"Rubble",
 "userid":"234"
}
```

Example: Using wholeTextFiles

```
userRDD = spark.sparkContext. \
  wholeTextFiles("userFiles")
```

Language: *Python*

userRDD

("user1.json",{"firstName":"Fred", "lastName":"Flintstone","userid":"123"})
("user2.json",{"firstName":"Barney", "lastName":"Rubble","userid":"234"})
("user3.json",…)
("user4.json",…)

Creating RDDs from Collections

- **You can create RDDs from collections instead of files**
  - `SparkContext.parallelize(`*collection*`)`
- **Useful when**
  - Testing
  - Generating data programmatically
  - Integrating with other systems or libraries
  - Learning

```
myData = ["Alice","Carlos","Frank","Barbara"]
myRDD = sc.parallelize(myData)
```

Language: *Python*

Saving RDDs

- **You can save RDDs to the same data source types supported for reading RDDs**
  - Use `RDD.saveAsTextFile` to save as plain text files in the specified directory
  - Use `RDD.saveAsHadoopFile` or `saveAsNewAPIHadoopFile` with a specified Hadoop `OutputFormat` to save using other formats
- **The specified save directory cannot already exist**

```
myRDD.saveAsTextFile("mydata/")
```

RDD Operations

- **Two general types of RDD operations**
  - *Actions* return a value to the Spark driver or save data to a data source
  - *Transformations* define a new RDD based on the current one(s)
- **RDDs operations are performed lazily**
  - Actions trigger execution of the base RDD transformations

RDD Action Operations

- **Some common actions**
  - `count` returns the number of elements
  - `first` returns the first element
  - `take(n)` returns an array (Scala) or list (Python) of the first *n* elements
  - `collect` returns an array (Scala) or list (Python) of all elements
  - `saveAsTextFile(dir)` saves to text files

```
myRDD = sc. \
  textFile("purplecow.txt")

for line in myRDD.take(2):
  print line
I've never seen a purple cow.
I never hope to see one;
```

Language: *Python*

```
val myRDD = sc.
  textFile("purplecow.txt")

for (line <- myRDD.take(2))
  println(line)
I've never seen a purple cow.
I never hope to see one;
```

Language: *Scala*

RDD Transformation Operations (1)

- **Transformations create a new RDD from an existing one**
- **RDDs are immutable**
  - Data in an RDD is never changed
  - Transform data to create a new RDD
- **A transformation operation executes a *transformation function***
  - The function transforms elements of an RDD into new elements
  - Some transformations implement their own transformation logic
  - For many, you must provide the function to perform the transformation

RDD Transformation Operations (2)

- **Transformation operations include**
  - `distinct` creates a new RDD with duplicate elements in the base RDD removed
  - `union(`*`rdd`*`)` creates a new RDD by appending the data in one RDD to another
  - `map(`*`function`*`)` creates a new RDD by performing a function on each record in the base RDD
  - `filter(`*`function`*`)` creates a new RDD by including or excluding each record in the base RDD according to a Boolean function

Example: `distinct` and `union` Transformations

cities1.csv

```
Boston,MA
Palo Alto,CA
Santa Fe,NM
Palo Alto,CA
```

cities2.csv

```
Calgary,AB
Chicago,IL
Palo Alto,CA
```

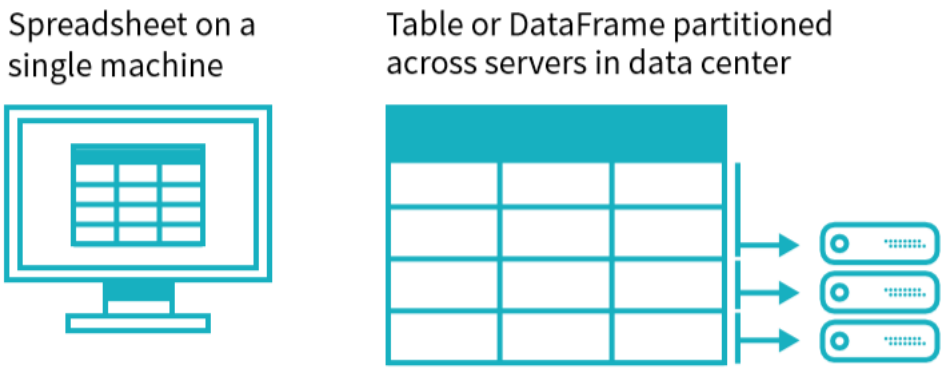
```
distinctRDD = sc.\
  textFile("cities1.csv").distinct()
for city in distinctRDD.collect(): \
  print city
Boston,MA
Palo Alto,CA
Santa Fe,NM

unionRDD = sc.textFile("cities2.csv"). \
  union(distinctRDD)
for city in unionRDD.collect():
  print city
Calgary,AB
Chicago,IL
Palo Alto,CA
Boston,MA
Palo Alto,CA
Santa Fe,NM
```

Language: *Python*

f

DataFrames

A DataFrame is the most common Structured API and simply represents a table of data with rows and columns. A simple analogy would be a spreadsheet with named columns. The fundamental difference is that while a spreadsheet sits on one computer in one specific location, a Spark DataFrame can span thousands of computers. The reason for putting the data on more than one computer should be intuitive: either the data is too large to fit on one machine or it would simply take too long to perform that computation on one machine.



The DataFrame concept is not unique to Spark. R and Python both have similar concepts. However, Python/R DataFrames (with some exceptions) exist on one machine rather than multiple machines. This limits what you can do with a given DataFrame in python and R to the resources that exist on that specific machine. However, since Spark has language interfaces for both Python and R, it's quite easy to convert to Pandas (Python) DataFrames to Spark DataFrames and R DataFrames to Spark DataFrames (in R).

NOTE: Spark has several core abstractions: Datasets, DataFrames, SQL Tables, and Resilient Distributed Datasets (RDDs). These abstractions all represent distributed collections of data however they have different interfaces for working with that data. The easiest and most efficient are DataFrames, which are available in all languages

Partitions

In order to allow every executor to perform work in parallel, Spark breaks up the data into chunks, called partitions. A partition is a collection of rows that sit on one physical machine in our cluster. A DataFrame's partitions represent how the data is physically distributed across your cluster of machines during execution. If you have one partition, Spark will only have a parallelism of one even if you have thousands of executors. If you have many partitions, but only one executor Spark will still only have a parallelism of one because there is only one computation resource.

An important thing to note, is that with DataFrames, we do not (for the most part) manipulate partitions manually (on an individual basis). We simply specify high level transformations of data in the physical partitions and Spark determines how this work will actually execute on the cluster. Lower level APIs do exist (via the Resilient Distributed Datasets interface)

Interoperating between DataFrames, Datasets, and RDDs

There may be times when you need to drop down to a RDD in order to perform some very specific sampling, operation, or specific MLlib algorithm not available in the DataFrame API. Doing this is simple, simply leverage the RDD property on any structured data types. You'll notice that if we do a conversion from a Dataset to an RDD, you'll get the appropriate native type back.

```
%scala
spark.range( 10)          //DataSets

spark.range( 10).rdd     // RDD
```

However if we convert from a DataFrame to a RDD, we will get an RDD of type Row.

```
spark.range( 10).toDF().rdd
```

In order to operate on this data, you will have to convert this Row object to the correct data type or extract values out of it like you see below.

```
spark.range(10).toDF().rdd.map( rowObject => rowObject.getLong(0))
```

This same methodology allows us to create a DataFrame or Dataset from a RDD. All we have to do is call the toDF method on the RDD.

```
spark.range( 10).rdd.toDF()

%python

spark.range( 10). rdd.toDF()
```

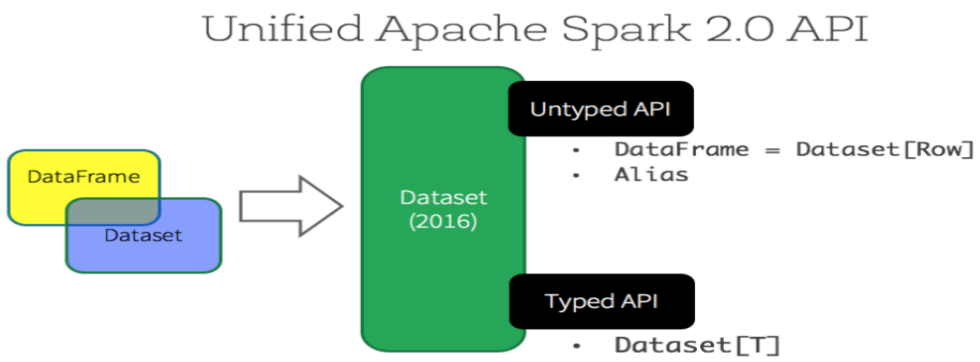
Datasets:

Dataset takes on two distinct APIs characteristics: a strongly-typed API and an untyped API. Conceptually, consider DataFrame as an alias for a collection of generic objects Dataset[Row], where a Row is a generic untyped JVM object.

Dataset, by contrast, is a collection **of strongly-typed JVM objects**, dictated by a **case class** you define in Scala or a class in Java.

Please note:

As of Apache Spark 2.0, the DataFrame and Dataset APIs are merged together; a DataFrame is the Dataset Untyped API while what was known as a Dataset is the Dataset Typed API.

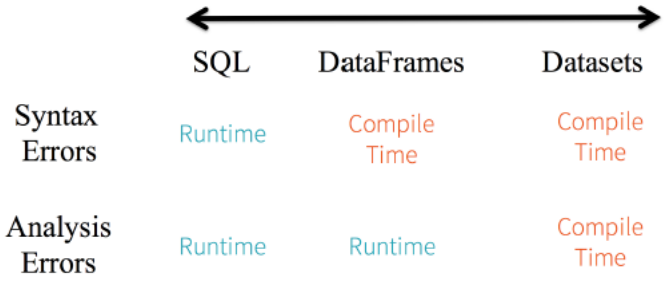


Typed and Un-typed APIs

Language	Main Abstraction
Scala	Dataset[T] & DataFrame (alias for Dataset[Row])
Java	Dataset[T]
Python*	DataFrame
R*	DataFrame

Data Organization (Row Format VS Column Format)

Benefits of APIs



A Tele of Three Apache Spark API: RDD, DataFrames and DataSets API

When to use RDDs?

Consider these scenarios or common use cases for using RDDs when:

• you want low-level transformation and actions and control on your dataset;

• your data is unstructured, such as media streams or streams of text;

• you want to manipulate your data with functional programming constructs than domain specific expressions;

• you don't care about imposing a schema, such as columnar format, while processing or accessing data attributes by name or column; and

• you can forgo some optimization and performance benefits available with DataFrames and Datasets for structured and semi-structured data.

When should I use DataFrames or Datasets?

• If you want rich semantics, high-level abstractions, and domain specific APIs, use DataFrame or Dataset.

• If your processing demands high-level expressions, filters, maps, aggregation, averages, sum, SQL queries, columnar access and use of lambda functions on semi-structured data, use DataFrame or Dataset.

• If you want higher degree of type-safety at compile time, want typed JVM objects, take advantage of Catalyst optimization, and benefit from Tungsten's efficient code generation, use Dataset.

If you want unification and simplification of APIs across Spark Libraries, use DataFrame or Dataset.

• If you are a R user, use DataFrames.

• If you are a Python user, use DataFrames and resort back to RDDs if you need more control.

Distributed Data Persistence

RDD Persistence

One of the most important capabilities in Spark is *persisting* (or *caching*) a dataset in memory across operations. When you persist an RDD, each node stores any partitions of it that it computes in memory and reuses them in other actions on that dataset (or datasets derived from it). This allows future actions to be much faster (often by more than 10x). Caching is a key tool for iterative algorithms and fast interactive use.

You can mark an RDD to be persisted using the `persist()` or `cache()` methods on it. The first time it is computed in an action, it will be kept in memory on the nodes. Spark's cache is fault-tolerant – if any partition of an RDD is lost, it will automatically be recomputed using the transformations that originally created it.

In addition, each persisted RDD can be stored using a different *storage level*, allowing you, for example, to persist the dataset on disk, persist it in memory but as serialized Java objects (to save space), replicate it across nodes. These levels are set by passing a `StorageLevel` object (Scala,Java, Python) to `persist()`. The `cache()` method is a shorthand for using the default storage level, which is `StorageLevel.MEMORY_ONLY` (store deserialized objects in memory). The full set of storage levels is:

Storage Level	Meaning
MEMORY_ONLY	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.
MEMORY_AND_DISK	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
MEMORY_ONLY_SER (Java and Scala)	Store RDD as *serialized* Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer, but more CPU-intensive to read.
MEMORY_AND_DISK_SER	Similar to MEMORY_ONLY_SER, but spill partitions that don't

(Java and Scala)	fit in memory to disk instead of recomputing them on the fly each time they're needed.
DISK_ONLY	Store the RDD partitions only on disk.
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.	Same as the levels above, but replicate each partition on two cluster nodes.
OFF_HEAP (experimental)	Similar to MEMORY_ONLY_SER, but store the data in off-heap memory. This requires off-heap memory to be enabled.

Note: *In Python, stored objects will always be serialized with the Pickle library, so it does not matter whether you choose a serialized level. The available storage levels in Python include* `MEMORY_ONLY`, `MEMORY_ONLY_2`, `MEMORY_AND_DISK`, `MEMORY_AND_DISK_2`, `DISK_ONLY`, *and* `DISK_ONLY_2`.

Spark also automatically persists some intermediate data in shuffle operations (e.g. `reduceByKey`), even without users calling `persist`. This is done to avoid recomputing the entire input if a node fails during the shuffle. We still recommend users call `persist` on the resulting RDD if they plan to reuse it.

Which Storage Level to Choose?

Spark's storage levels are meant to provide different trade-offs between memory usage and CPU efficiency. We recommend going through the following process to select one:

- If your RDDs fit comfortably with the default storage level (`MEMORY_ONLY`), leave them that way. This is the most CPU-efficient option, allowing operations on the RDDs to run as fast as possible.
- If not, try using `MEMORY_ONLY_SER` and selecting a fast serialization library to make the objects much more space-efficient, but still reasonably fast to access. (Java and Scala)
- Don't spill to disk unless the functions that computed your datasets are expensive, or they filter a large amount of the data. Otherwise, recomputing a partition may be as fast as reading it from disk.
- Use the replicated storage levels if you want fast fault recovery (e.g. if using Spark to serve requests from a web application). *All* the storage levels provide full fault tolerance by recomputing lost data, but the replicated ones let you continue running tasks on the RDD without waiting to recompute a lost partition.

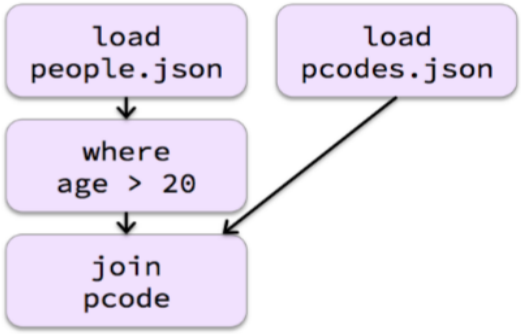
Persistence

- **You can *persist* a DataFrame, Dataset, or RDD**
  - Also called *caching*
  - Data is temporarily saved to memory and/or disk
- **Persistence can improve performance and fault-tolerance**
- **Use persistence when**
  - Query results will be used repeatedly
  - Executing the query again in case of failure would be very expensive
- **Persisted data cannot be shared between applications**

Example: DataFrame Persistence (1)

```
over20DF = spark.read. \
  json("people.json"). \
  where("age > 20")
pcodesDF = spark.read. \
  json("pcodes.json")
joinedDF = over20DF. \
  join(pcodesDF, "pcode")
```

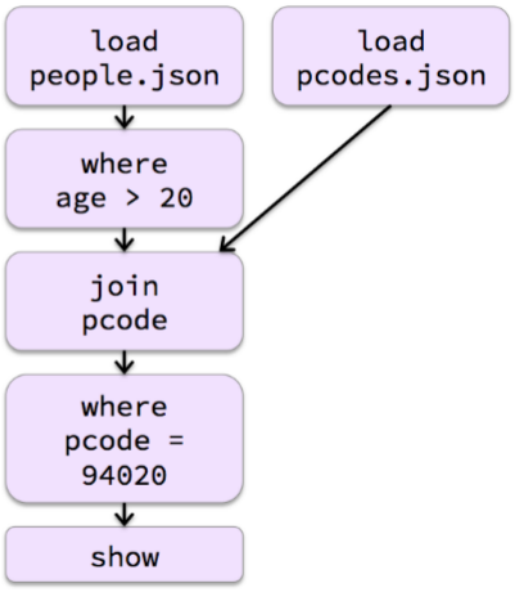
Language: *Python*



Example: DataFrame Persistence (2)

```
over20DF = spark.read. \
  json("people.json"). \
  where("age > 20")
pcodesDF = spark.read. \
  json("pcodes.json")
joinedDF = over20DF. \
  join(pcodesDF, "pcode")
joinedDF. \
  where("pcode = 94020"). \
  show()
```

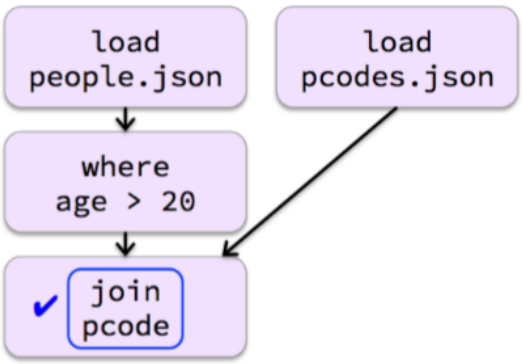
Language: *Python*



Example: DataFrame Persistence (3)

```
over20DF = spark.read. \
  json("people.json"). \
  where("age > 20")
pcodesDF = spark.read. \
  json("pcodes.json")
joinedDF = over20DF. \
  join(pcodesDF, "pcode"). \
  persist()
```

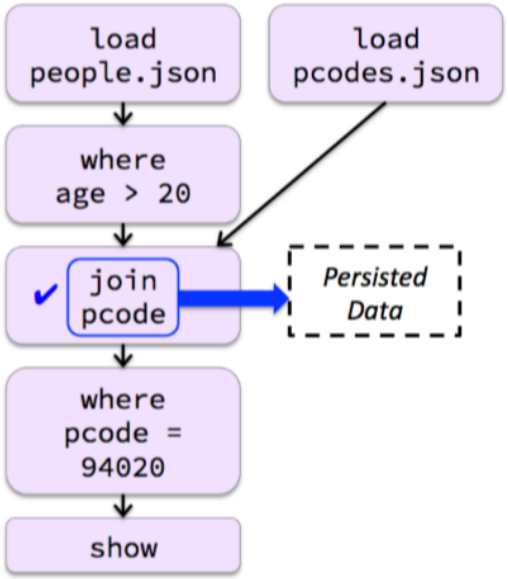
Language: *Python*



Example: DataFrame Persistence (4)

```
over20DF = spark.read. \
  json("people.json"). \
  where("age > 20")
pcodesDF = spark.read. \
  json("pcodes.json")
joinedDF = over20DF. \
  join(pcodesDF, "pcode"). \
  persist()
joinedDF. \
  where("pcode = 94020"). \
  show()
```

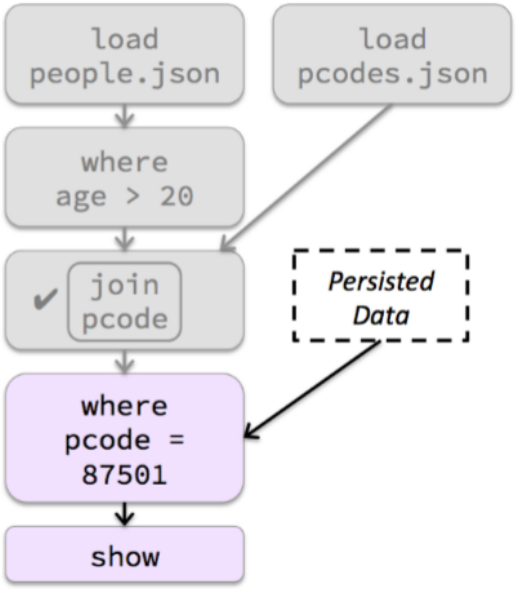
Language: *Python*



Example: DataFrame Persistence (5)

```
over20DF = spark.read. \
  json("people.json"). \
  where("age > 20")
pcodesDF = spark.read. \
  json("pcodes.json")
joinedDF = over20DF. \
  join(pcodesDF, "pcode"). \
  persist()
joinedDF. \
  where("pcode = 94020"). \
  show()
joinedDF. \
  where("pcode = 87501"). \
  show()
```

Language: *Python*



Storage Levels

- ***Storage levels* provide several options to manage how data is persisted**
  - Storage location (memory and/or disk)
  - Serialization of data in memory
  - Replication
- **Specify storage level when persisting a DataFrame, Dataset, or RDD**
  - Tables and views do not use storage levels
    - Always persisted in memory
- **Data is persisted based on partitions of the underlying RDDs**
  - Executors persist partitions in JVM memory or temporary local files
  - The application driver keeps track of the location of each persisted partition's data

Table and View Persistence

- **Tables and views can be persisted in memory using CACHE TABLE**

```
spark.sql("CACHE TABLE people")
```

- **CACHE TABLE can create a view based on a SQL query and cache it at the same time**

```
spark.sql("CACHE TABLE over_20 AS SELECT *
  FROM people WHERE age > 20")
```

- **Queries on cached tables work the same as on persisted DataFrames, Datasets, and RDDs**
  - The first query caches the data
  - Subsequent queries use the cached data

Storage Levels: Location

- **Storage location—where is the data stored?**
  - `MEMORY_ONLY`: Store data in memory if it fits
  - `DISK_ONLY`: Store all partitions on disk
  - `MEMORY_AND_DISK`: Store any partition that does not fit in memory on disk
    - Called *spilling*

```
from pyspark import StorageLevel
myDF.persist(StorageLevel.DISK_ONLY)
```

Language: Python

```
import org.apache.spark.storage.StorageLevel
myDF.persist(StorageLevel.DISK_ONLY)
```

Language: Scala

Storage Levels: Memory Serialization

- **In Python, data in memory is always serialized**
- **In Scala, you can choose to serialize data in memory**
  - By default, in Scala and Java, data in memory is stored objects
  - Use `MEMORY_ONLY_SER` and `MEMORY_AND_DISK_SER` to serialize the objects into a sequence of bytes instead
  - Much more space efficient but less time efficient
- **Datasets are serialized by Spark SQL encoders, which are very efficient**
  - Plain RDDs use native Java/Scala serialization by default
  - Use Kryo instead for better performance
- **Serialization options do not apply to disk persistence**
  - Files are always in serialized form by definition

Storage Levels: Partition Replication

- **Replication—store partitions on two nodes**
  - `DISK_ONLY_2`
  - `MEMORY_AND_DISK_2`
  - `MEMORY_ONLY_2`
  - `MEMORY_AND_DISK_SER_2` (Scala and Java only)
  - `MEMORY_ONLY_SER_2` (Scala and Java only)
  - You can also define custom storage levels for additional replication

Default Storage Levels

- **The `storageLevel` parameter for the DataFrame, Dataset, or RDD `persist` operation is optional**
  - The default for DataFrames and Datasets is `MEMORY_AND_DISK`
  - The default for RDDs is `MEMORY_ONLY`
- **`persist` with no storage level specified is a synonym for `cache`**

```
myDF.persist()
```

is equivalent to

```
myDF.cache()
```

- **Table and view storage level is always `MEMORY_ONLY`**

When and Where to Persist

- **When should you persist a DataFrame, Dataset, or RDD?**
  - When the data is likely to be reused
    - Such as in iterative algorithms and machine learning
  - When it would be very expensive to recreate the data if a job or node fails
- **How to choose a storage level**
  - **Memory**—use when possible for best performance
    - Save space by serializing the data if necessary
  - **Disk**—use when re-executing the query is more expensive than disk read
    - Such as expensive functions or filtering large datasets
  - **Replication**—use when re-execution is more expensive than bandwidth

Changing Storage Levels

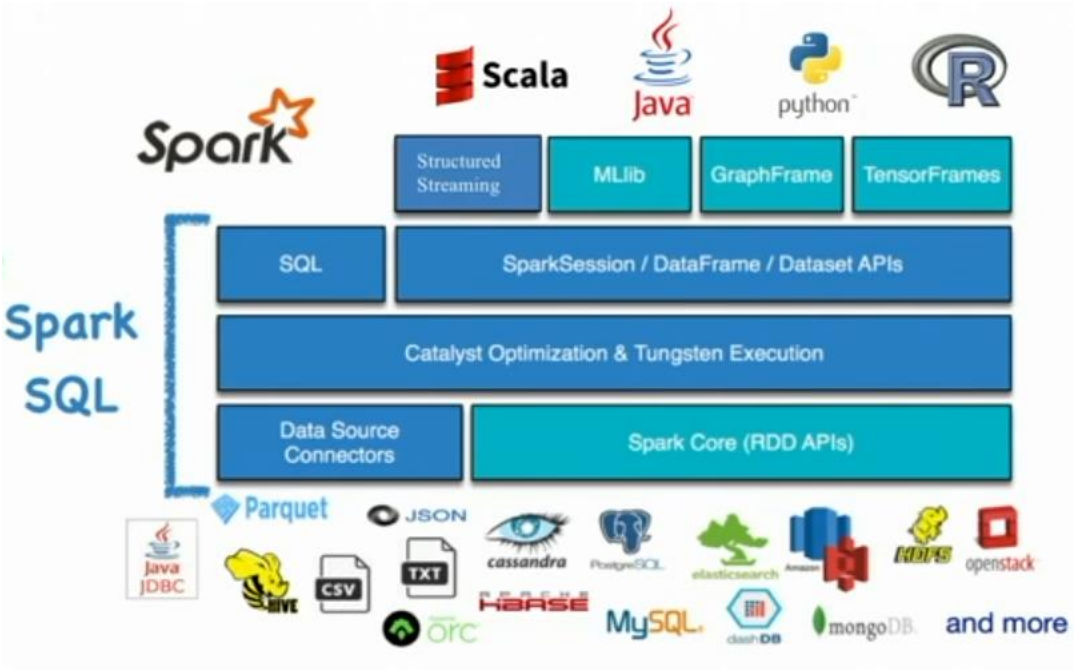
- **You can remove persisted data from memory and disk**
  - Use `unpersist` for Datasets, DataFrames, and RDDs
  - Use `Catalog.uncacheTable(`*`table-name`*`)` for tables and views
- **Unpersist before changing to a different storage level**
  - Re-persisting already-persisted data results in an exception

```
myDF.unpersist()
myDF.persist(new-level)
```

SparkSql

Spark SQL is a Spark module for structured data processing. Unlike the basic Spark RDD API, the interfaces provided by Spark SQL provide Spark with more information about the structure of both the data and the computation being performed.

One use of Spark SQL is to execute SQL queries. Spark SQL can also be used to read data from an existing Hive installation. For more on how to configure this feature, please refer to the Hive Tables section.

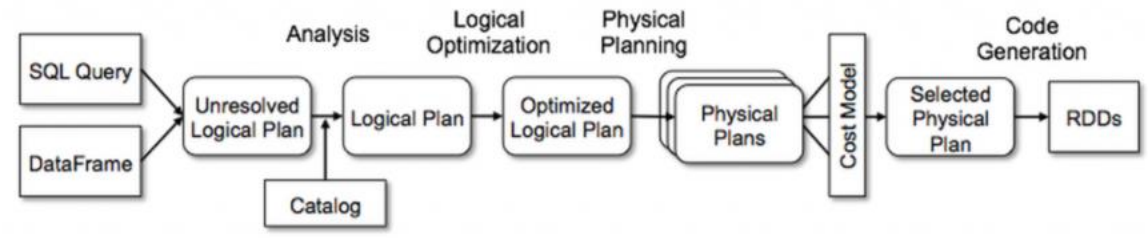


Catalyst Optimizer

Spark SQL is one of the most technically involved components of Apache Spark. It powers both SQL queries and the DataFrame API. At the core of Spark SQL is the Catalyst optimizer, which leverages advanced programming language features (e.g. Scala's pattern matching and quasiquotes) in a novel way to build an extensible query optimizer.

Catalyst is based on functional programming constructs in Scala and designed with these key two purposes:

- Easily add new optimization techniques and features to Spark SQL
- Enable external developers to extend the optimizer (e.g. adding data source specific rules, support for new data types, etc.)



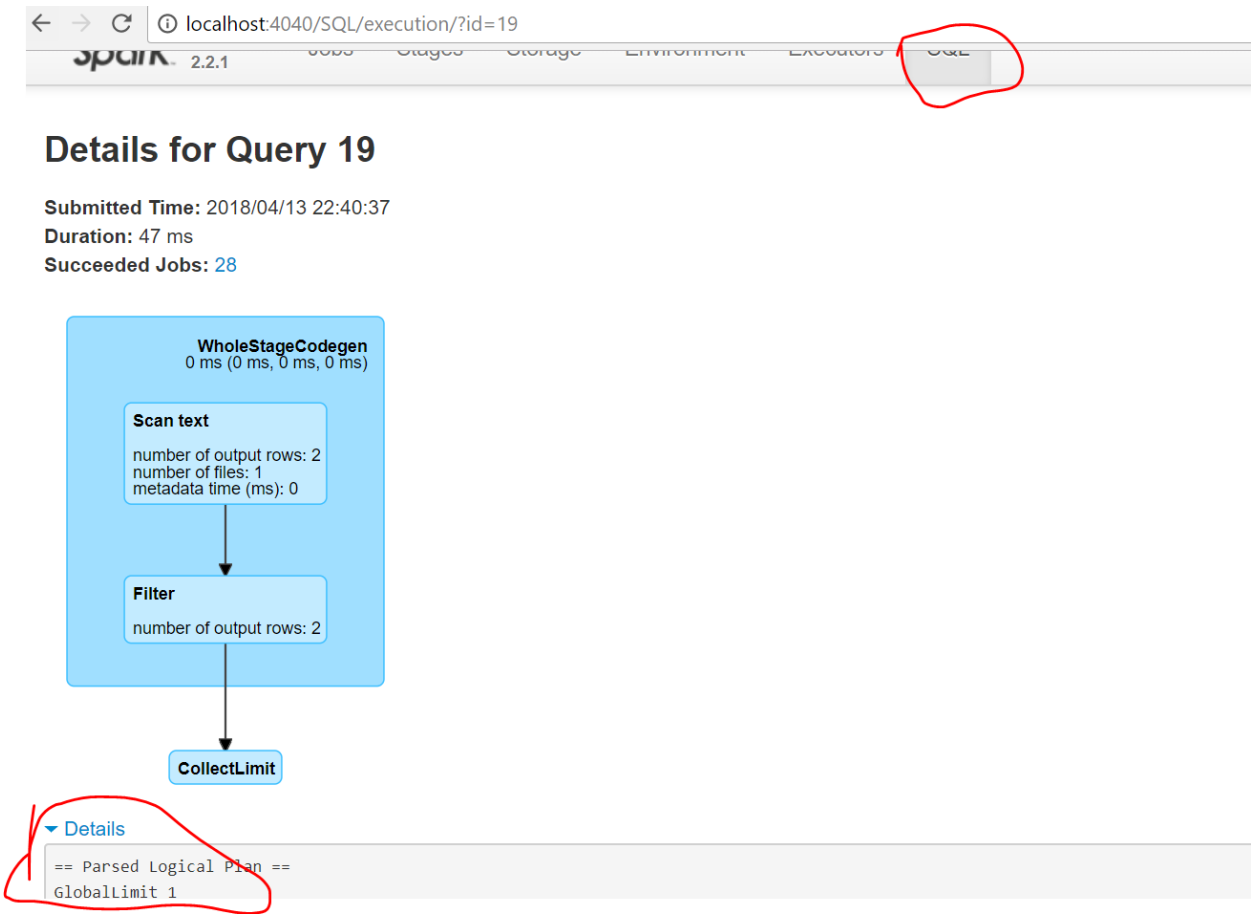
Tungsten

Tungsten is the codename for the umbrella project to make changes to Apache Spark's execution engine that focuses on substantially improving the efficiency of memory and CPU for Spark applications, to push performance closer to the limits of modern hardware. This effort includes the following initiatives:

- *Memory Management and Binary Processing:* leveraging application semantics to manage memory explicitly and eliminate the overhead of JVM object model and garbage collection
- *Cache-aware computation:* algorithms and data structures to exploit memory hierarchy
- *Code generation:* using code generation to exploit modern compilers and CPUs
- *No virtual function dispatches:* this reduces multiple CPU calls which can have a profound impact on performance when dispatching billions of times.
- *Intermediate data in memory vs CPU registers:* Tungsten Phase 2 places intermediate data into CPU registers. This is an order of magnitudes reduction in the number of cycles to obtain data from the CPU registers instead of from memory
- *Loop unrolling and SIMD:* Optimize Apache Spark's execution engine to take advantage of modern compilers and CPUs' ability to efficiently compile and execute simple for loops (as opposed to complex function call graphs).

Example:

```
val peopleDF = spark.read.option("header","true").csv("D:/Spark_Scala_Training/data/people.csv")

val pcodesDF =
spark.read.option("header","true").csv("D:/Spark_Scala_Training/data/pcodes.csv")

val joinedDF = peopleDF.join(pcodesDF, "pcode")

joinedDF.explain()
```


localhost:4040/SQL/execution/?id=19
2.2.1
Details for Query 19
Submitted Time: 2018/04/13 22:40:37
Duration: 47 ms
Succeeded Jobs: 28
WholeStageCodegen
0 ms (0 ms, 0 ms, 0 ms)
Scan text
number of output rows: 2
number of files: 1
metadata time (ms): 0
Filter
number of output rows: 2
CollectLimit
Details
== Parsed Logical Plan ==
GlobalLimit 1

Apache Spark: MLlib

Spark's Packages for Advanced Analytics Spark includes several core packages and many external packages for performing advanced analytics.

The primary package is MLlib which provides an interface for building machine learning pipelines. We elaborate on other packages in later chapters.

What is MLlib?

MLlib is a package, built on and included in Spark, that provides interfaces for.

1. gathering and cleaning data,
2. generating and selecting features,
3. training and tuning large scale supervised and unsupervised machine learning models, and using those models in production.

This means that it helps with all three steps of the process although it really shines in steps one and two for reason that we will touch on shortly.

MLlib consists of two packages that leverage different core data structures. The package **org.apache.spark.ml** maintains an interface for use with Spark DataFrames. This package also maintains a high level interface for building machine learning pipelines that help standardize the way in which you perform the above steps. The lower level package, org.apache.spark.mllib, maintains interfaces for Spark's Low-Level, RDD APIs.

We will focus on the DataFrame API because the RDD API is both well documented and is currently in maintenance mode (meaning it will only receive bug fixes, not new features) at the time of this writing.

When and why should you use MLlib (vs scikit learn or another package)?

Now, at a high level, this sounds like a lot of other machine learning packages you have probably heard of like scikit-learn for Python or the variety of R packages for performing similar tasks. So why should you bother MLlib at all? The answer is simple, scale. There are numerous tools for performing machine learning on a single machine. They do quite well at this and will continue to be great tools. However they reach a limit, either in data size or processing time. This is where Spark excels. The fact that they hit a limit in terms of scale makes them complementary tools, not competitive ones. When your input data or model size become too difficult or inconvenient to put on one machine, use Spark to do the heavy lifting. Spark makes big data machine learning simple.

Advance Analytics and Machine Learning:

Gartner defines advanced analytics as follows:

Advanced Analytics is the autonomous or semi-autonomous examination of data or content using sophisticated techniques and tools, typically beyond those of traditional business intelligence (BI), to discover deeper insights, make predictions, or generate recommendations. Advanced analytic techniques include those such as data/ text mining, machine learning, pattern matching, forecasting, visualization, semantic analysis, sentiment analysis, network and cluster analysis, multivariate statistics, graph analysis, simulation, complex event processing, neural networks.

Reference:

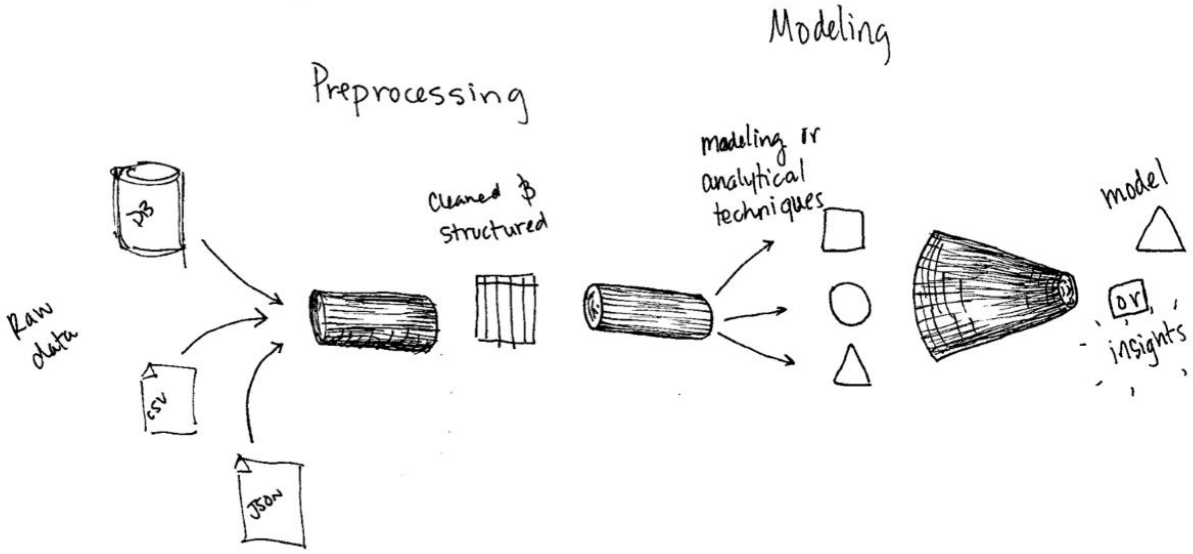
http://www.deeplearningbook.org/

http://www.saedsayad.com/data_mining_map.htm

Spark can be leverage the do difference advance Analytics tasks:

- Preprocessing data(Cleansing data)
- Feature Engineering
- Supervised Learning
- Unsupervised Learning
- Recommendation Engines
- Graph Analysis

The Advanced Analytics Workflow



Supervised Learning

Supervised learning occurs when you train a model to predict a particular outcome based on historical information.

Some examples of supervised learning include:

Spam email detection - Spam detection systems leverage supervised learning to predict whether or not a message is spam or not. It does like by analyzing the contents of a given email.

Classifying handwritten digits - The United States Postal Service had a use case where they wanted to be able to read handwritten addresses on letters. To do this they leverage machine learning to train a classifier to them the value of a given digit.

Predicting heart disease - A doctor or hospital might want to predict the likelihood of a person's body characteristics or lifestyle leading to heart disease later in life.

Anomaly Detection - Given some standard event type often occurring over time, we might want to report when a non-standard type of event occurs (non-standard being a potentially difficult term to define generally.) An example of this might be that a security officer would like to receive a notification when a strange object (think vehicle, skater or bicyclist) is observed on a pathway.

Recommendation

The task of recommendation is likely one of the most intuitive. By studying what people either explicitly state that they like and dislike (through ratings) or by what they implicitly state that they like and dislike (through observed behavior) you can make recommendations on what one user may like by drawing similarities between those individuals and other individuals.

Some examples of recommendations are:

Movie Recommendations - Netflix uses Spark to make large scale movie recommendations to their users. More generally, movies can be recommended based on what you watch as well as what you rated previously. Product Recommendations - In order to promote high purchases, companies use product recommendations to suggest new products to buy to their customers. This can be based on previous purchases or simply viewing behavior. Unsupervised Learning

https://www.youtube.com/watch?v=-Jcq1a__MBQ&list=PL-x35fyliRwiyjnk7JLUYm3Gj231x1z3x

Product Recommendations - In order to promote high purchases, companies use product recommendations to suggest new products to buy to their customers. This can be based on previous purchases or simply viewing behavior.

Unsupervised Learning

Unsupervised learning occurs when you train a model on data that does not have a a specific outcome variable.

The goal is to discover and describe some underlying structure or clusters in the data.

Some examples of unsupervised learning include:

Clustering - Given some traits in plant types, we might want to cluster them by these attributes in order to try and find similarities (or differences) between them.

Anomaly Detection - Given some standard event type often occuring over time, we might want to report when a non-standard type of event occurs (non-standard being a potentially difficult term to define

Graph Analysis

Graph analysis is a bit more of a sophisticated analytical tool that can absorb aspects of all of the above. Graph analysis is effectively the study of relationships where we specify "vertices" which are objects and "edges" which represent relationships between those objects.

Some examples of graph analysis include:

Fraud Prediction - Capital One uses Spark's graph analytics capabilities to better understand fraud networks. This includes assigning probabilities to certain bits of information to make a decision about whether or not a given piece of information suggests that a charge it fraudulent.

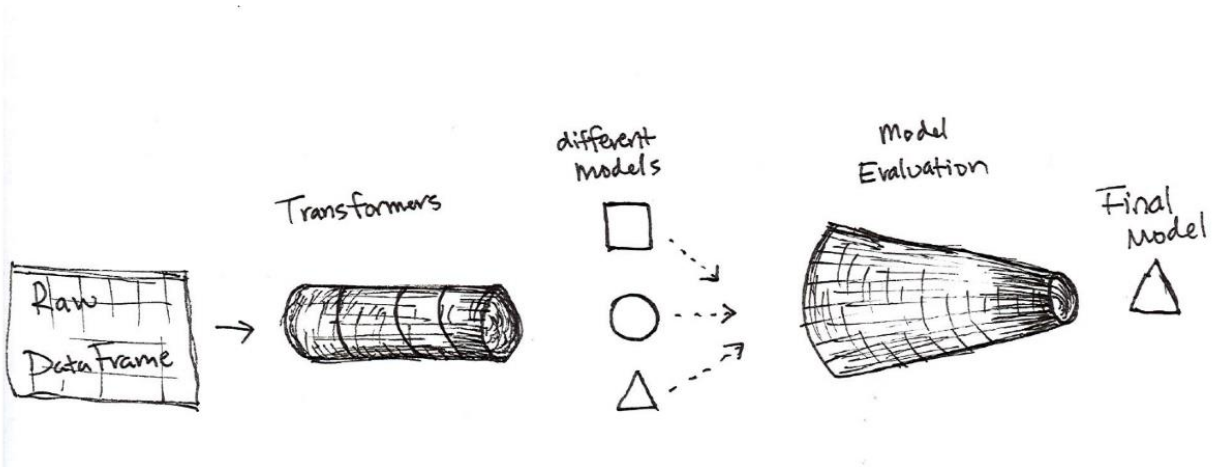
Anomaly Detection - By looking at how networks of individuals connect with one another, outliers and anomalies can be flagged for manual analysis.

Classification - Given some facts about certain vertices in the network, you can classify other vertices according to their connection to that original node. An example might be looking at classifying influencers in friend groups.

High Level MLlib Concepts

In MLlib there are several fundamental architectural types: transformers, estimators, evaluator and pipelines.

The following is a diagram of the overall workflow.



Transformers are just functions that convert raw data into another, usually more structured representation. Additionally they allow you to create new features from your data like interactions between variables. An example of a transformer is one converts string categorical variables into a better representation for our algorithms. Transformers are primarily used in the first step of the machine learning process we described previously.

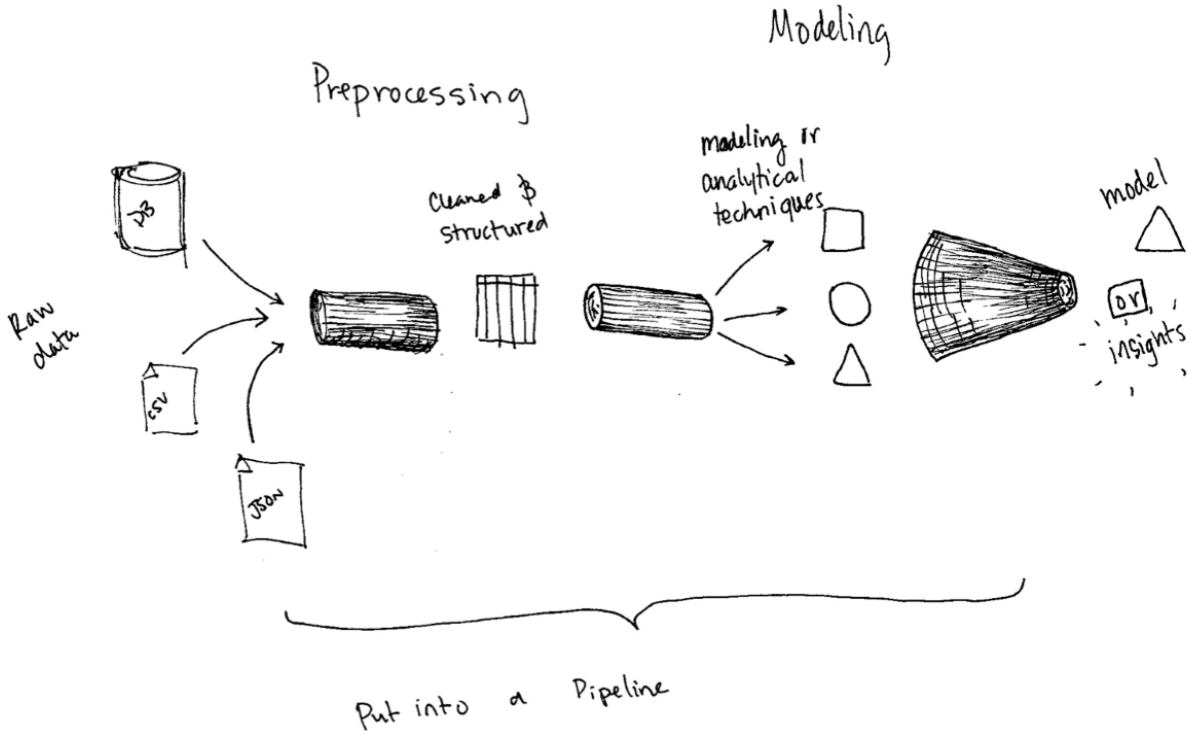
Estimators represent different models (or variations of the same model) that are trained and then tested using an evaluation.

An **evaluator** allows us to see how a given estimator performs according to some criteria that we specify like a ROC curve. Once we select the best model from the ones that we tested, we can then use it to make predictions.

Pipelining our Workflow

Pipelining our Workflow As you likely noticed above, if you are performing a lot of transformations, writing all the steps and keeping track of DataFrames ends up being quite tedious. That's why Spark includes the concept of a Pipeline. A pipeline allows you to set up a dataflow of the relevant transformations, ending with an estimator that is automatically tuned according to your specifications resulting a tuned model ready for a production use case.

The following diagram illustrates this process.



Example:

```
import org.apache.spark.ml.classification.LogisticRegression
import org.apache.spark.ml.linalg.{Vector, Vectors}
import org.apache.spark.ml.param.ParamMap
import org.apache.spark.sql.Row

// Prepare training data from a list of (label, features) tuples.
val training = spark.createDataFrame(Seq(
  (1.0, Vectors.dense(0.0, 1.1, 0.1)),
  (0.0, Vectors.dense(2.0, 1.0, -1.0)),
  (0.0, Vectors.dense(2.0, 1.3, 1.0)),
  (1.0, Vectors.dense(0.0, 1.2, -0.5))
)).toDF("label", "features")

// Create a LogisticRegression instance. This instance is an Estimator.
val lr = new LogisticRegression()
// Print out the parameters, documentation, and any default values.
println(s"LogisticRegression parameters:\n ${lr.explainParams()}\n")
```

```scala
// We may set parameters using setter methods.
lr.setMaxIter(10)
  .setRegParam(0.01)

// Learn a LogisticRegression model. This uses the parameters stored in lr.
val model1 = lr.fit(training)
// Since model1 is a Model (i.e., a Transformer produced by an Estimator),
// we can view the parameters it used during fit().
// This prints the parameter (name: value) pairs, where names are unique IDs for this
// LogisticRegression instance.
println(s"Model 1 was fit using parameters: ${model1.parent.extractParamMap}")

// We may alternatively specify parameters using a ParamMap,
// which supports several methods for specifying parameters.
val paramMap = ParamMap(lr.maxIter -> 20)
  .put(lr.maxIter, 30)  // Specify 1 Param. This overwrites the original maxIter.
  .put(lr.regParam -> 0.1, lr.threshold -> 0.55)  // Specify multiple Params.

// One can also combine ParamMaps.
val paramMap2 = ParamMap(lr.probabilityCol -> "myProbability")  // Change output column name.
val paramMapCombined = paramMap ++ paramMap2

// Now learn a new model using the paramMapCombined parameters.
// paramMapCombined overrides all parameters set earlier via lr.set* methods.
val model2 = lr.fit(training, paramMapCombined)
println(s"Model 2 was fit using parameters: ${model2.parent.extractParamMap}")

// Prepare test data.
val test = spark.createDataFrame(Seq(
  (1.0, Vectors.dense(-1.0, 1.5, 1.3)),
  (0.0, Vectors.dense(3.0, 2.0, -0.1)),
  (1.0, Vectors.dense(0.0, 2.2, -1.5))
)).toDF("label", "features")

// Make predictions on test data using the Transformer.transform() method.
// LogisticRegression.transform will only use the 'features' column.
// Note that model2.transform() outputs a 'myProbability' column instead of the usual
// 'probability' column since we renamed the lr.probabilityCol parameter previously.
model2.transform(test)
  .select("features", "label", "myProbability", "prediction")
  .collect()
  .foreach { case Row(features: Vector, label: Double, prob: Vector, prediction: Double) =>
    println(s"($features, $label) -> prob=$prob, prediction=$prediction")
```

```
}
```

Pipeline Example:

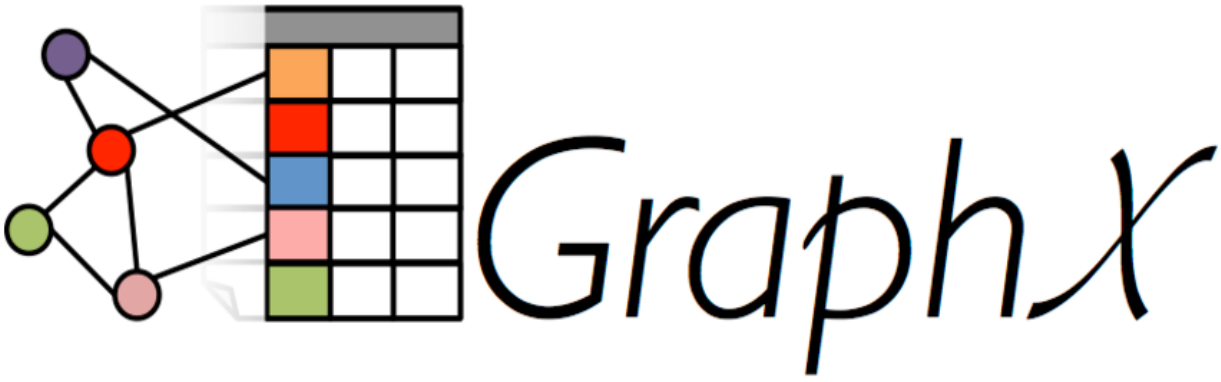
```
import org.apache.spark.ml.{Pipeline, PipelineModel}
import org.apache.spark.ml.classification.LogisticRegression
import org.apache.spark.ml.feature.{HashingTF, Tokenizer}
import org.apache.spark.ml.linalg.Vector
import org.apache.spark.sql.Row

// Prepare training documents from a list of (id, text, label) tuples.
val training = spark.createDataFrame(Seq(
  (0L, "a b c d e spark", 1.0),
  (1L, "b d", 0.0),
  (2L, "spark f g h", 1.0),
  (3L, "hadoop mapreduce", 0.0)
)).toDF("id", "text", "label")

// Configure an ML pipeline, which consists of three stages: tokenizer, hashingTF, and lr.
val tokenizer = new Tokenizer()
  .setInputCol("text")
  .setOutputCol("words")
val hashingTF = new HashingTF()
  .setNumFeatures(1000)
  .setInputCol(tokenizer.getOutputCol)
  .setOutputCol("features")
val lr = new LogisticRegression()
  .setMaxIter(10)
  .setRegParam(0.001)
val pipeline = new Pipeline()
  .setStages(Array(tokenizer, hashingTF, lr))

// Fit the pipeline to training documents.
val model = pipeline.fit(training)

// Now we can optionally save the fitted pipeline to disk
model.write.overwrite().save("/tmp/spark-logistic-regression-model")

// We can also save this unfit pipeline to disk
pipeline.write.overwrite().save("/tmp/unfit-lr-model")
```

```
// And load it back in during production
val sameModel = PipelineModel.load("/tmp/spark-logistic-regression-model")

// Prepare test documents, which are unlabeled (id, text) tuples.
val test = spark.createDataFrame(Seq(
  (4L, "spark i j k"),
  (5L, "l m n"),
  (6L, "spark hadoop spark"),
  (7L, "apache hadoop")
)).toDF("id", "text")

// Make predictions on test documents.
model.transform(test)
  .select("id", "text", "probability", "prediction")
  .collect()
  .foreach { case Row(id: Long, text: String, prob: Vector, prediction: Double) =>
    println(s"($id, $text) --> prob=$prob, prediction=$prediction")
  }
```

Example:

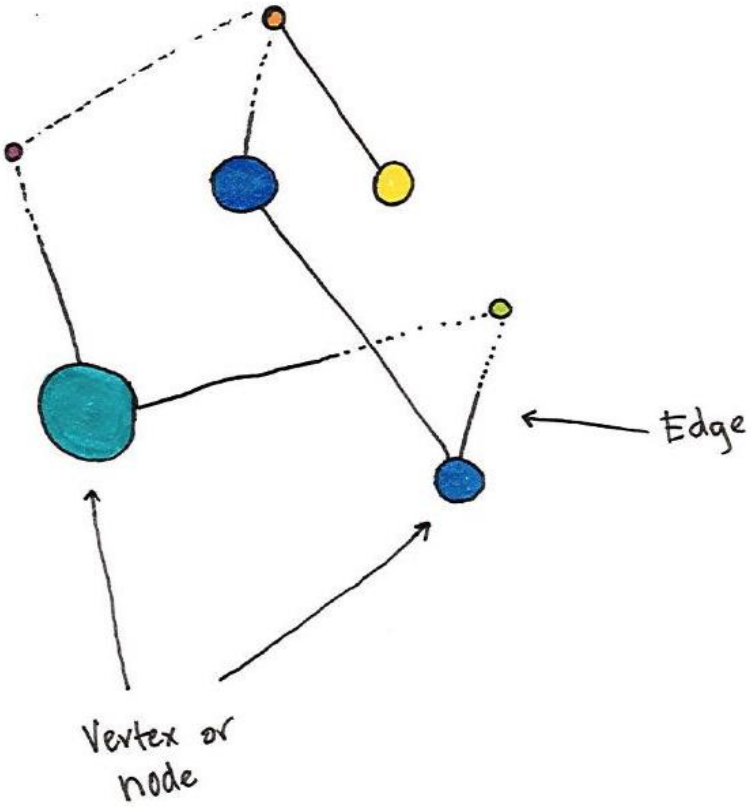
Recommendation Engine using MLlib

Apache Spark: GraphX



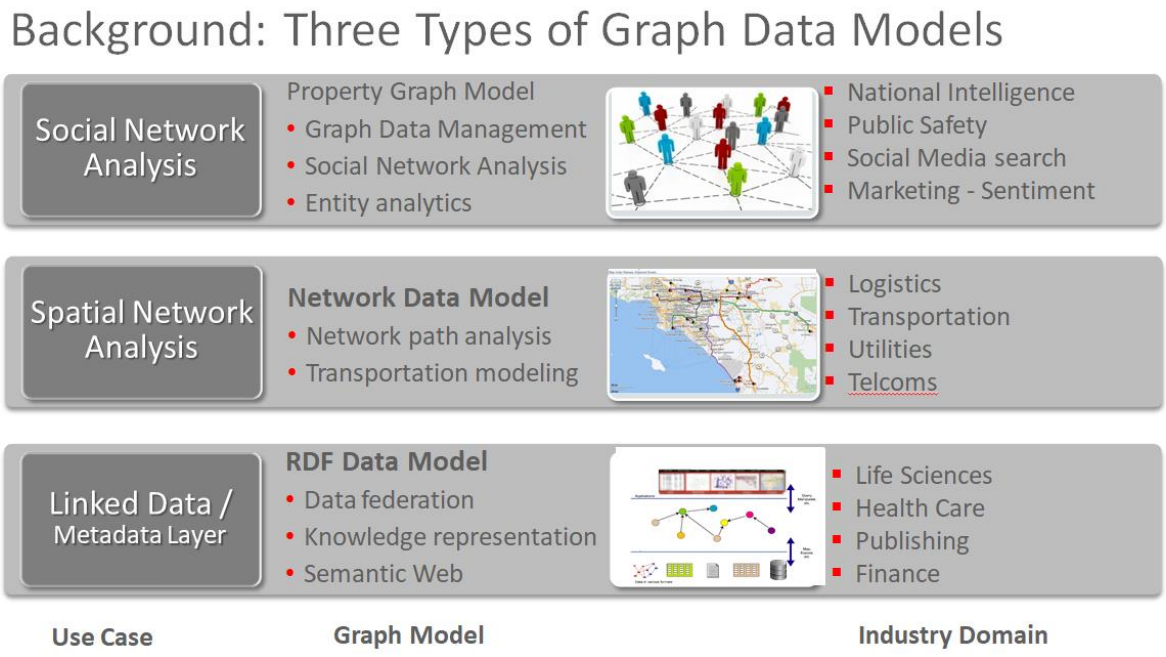
Graph

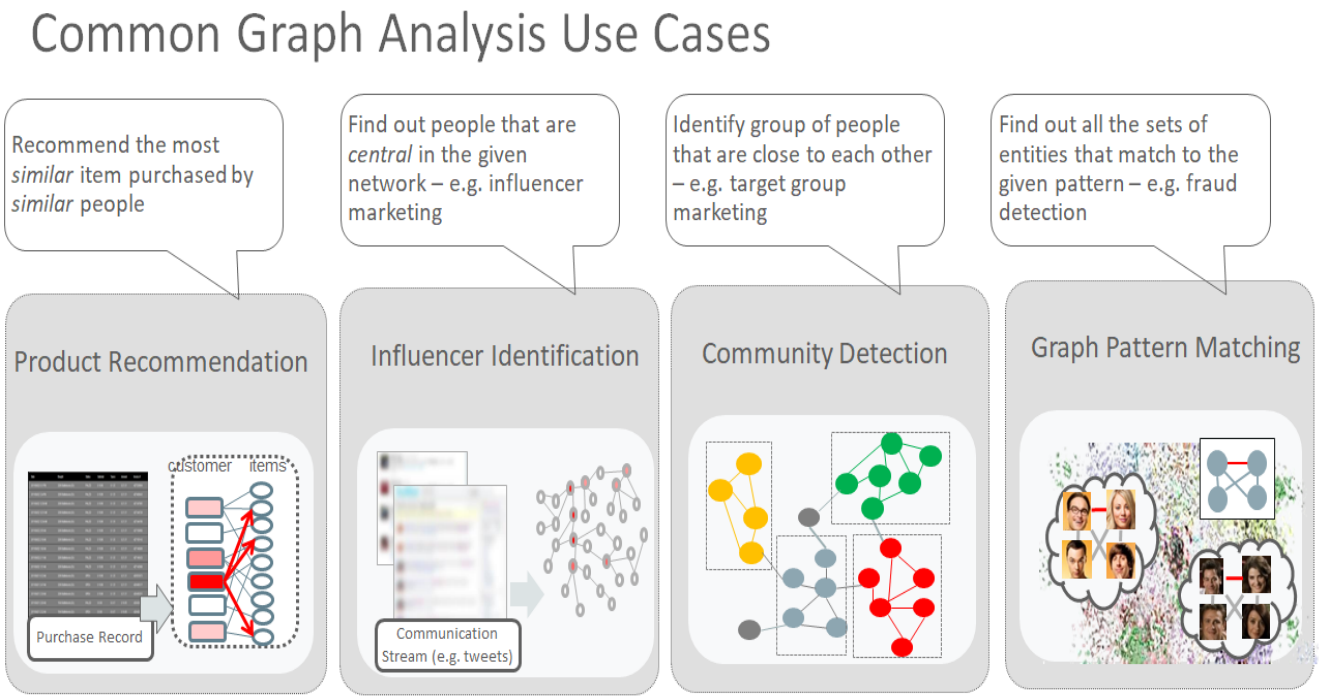
Graphs are an intuitive and natural way of describing relationships between objects. In the context of graphs, nodes or vertices are the units while edges define the relationships between those nodes. The process of graph analysis is the process of analyzing these relationships. An example analysis might be your friend group, in the context of graph analysis each vertex or node would represent a person and each edge would represent a relationship.



Why do we care?

- Graphs are everywhere
  - Social networks/Social Web (Facebook, Linkedin, Twitter, Google+,…)
  - Cyber networks, power grids, protein interaction graphs
  - Knowledge graphs (IBM Watson, Apple SIRI, Google Knowledge Graph)
- Graphs are intuitive and flexible
  - Easy to navigate, easy to form a path, natural to visualize
  - Do not require a predefined schema





Graph Algorithms

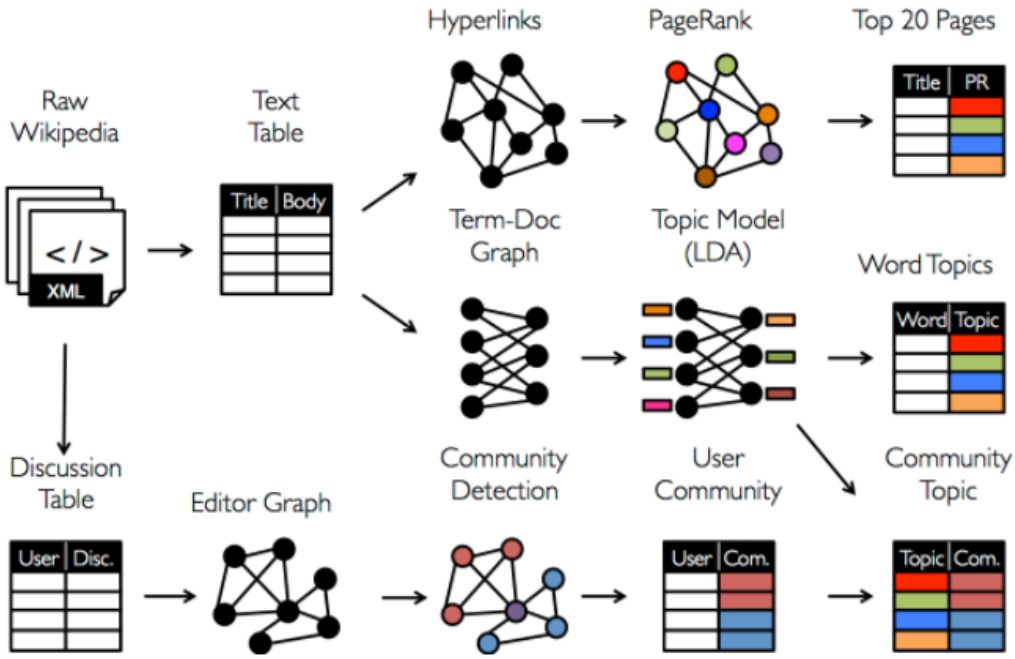
A graph is just a logical representation of data. Graph theory provides numerous algorithms for describing data in this format and GraphFrames allows us to leverage many algorithms out of the box.

PageRank arguably one of the most prolific graph algorithm.

https://en.wikipedia.org/wiki/PageRank

PageRank, Larry Page, founder of Google, created PageRank as a research project for how to rank webpages.

PageRank works by counting the number and quality of links to a page to determine a rough estimate of how important the website is. The underlying assumption is that more important websites are likely to receive more links from other websites. However PageRank generalizes quite well outside of the web domain. We can apply this right to our own data and get a sense for important bike stations.



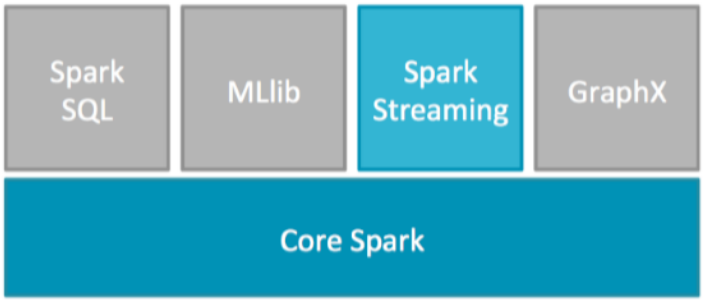
Reference: Graphframes package

https://spark-packages.org/package/graphframes/graphframes

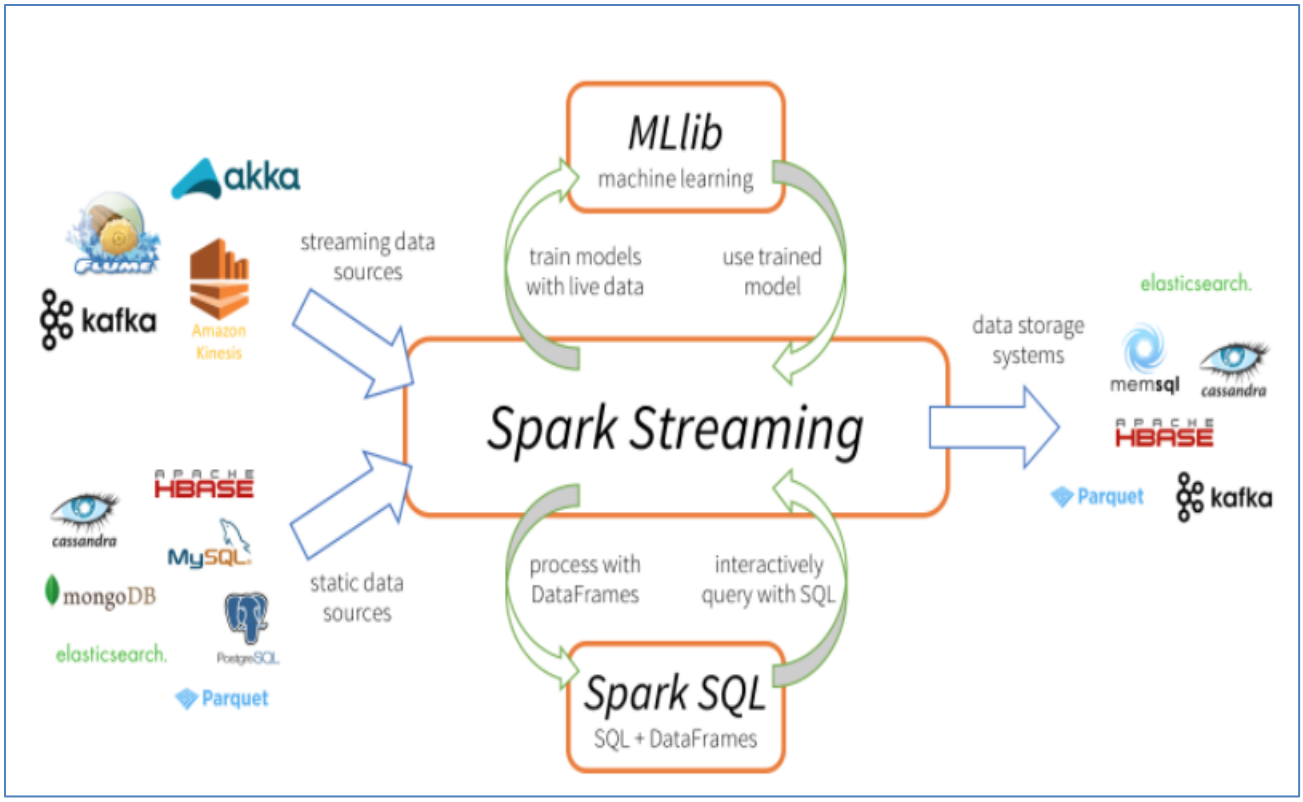
Apache Spark Streaming

What Is Spark Streaming?

- **An extension of core Spark**
- **Provides real-time data processing**
- **Segments an incoming stream of data into micro-batches**



- Spark Streaming is an extension of the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams
- Data can be ingested from many sources like Kafka, Flume, Kinesis, or TCP sockets, and can be processed using complex algorithms expressed with high-level functions like map, reduce, join and window.
- Finally, processed data can be pushed out to filesystems, databases, and live dashboards. In fact, you can apply Spark's machine learning and graph processing algorithms on data streams



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

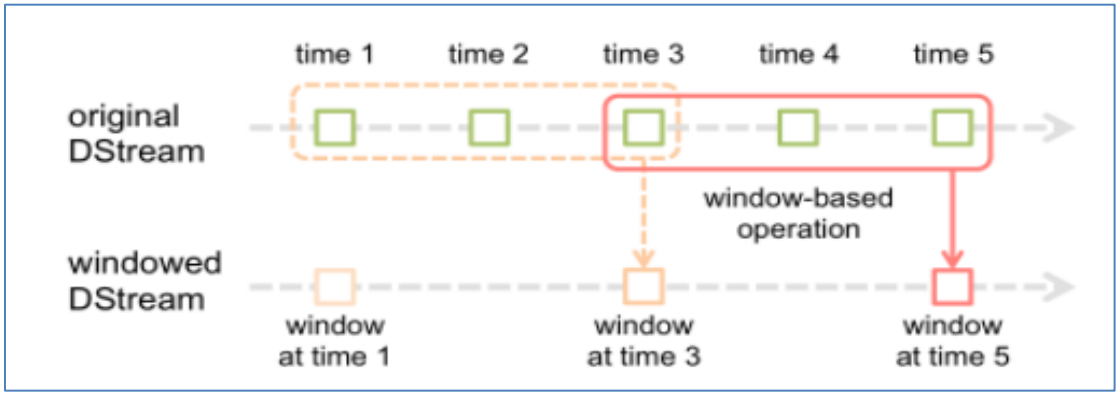
- Spark Streaming provides a high-level abstraction called *discretized stream* or *DStream*, which represents a continuous stream of data. DStreams can be created either from input data streams from sources such as Kafka, Flume, and Kinesis, or by applying high-level operations on other DStreams. Internally, a DStream is represented as a sequence of RDDs.

Discretized Streams (DStreams)

Discretized Stream or **DStream** is the basic abstraction provided by Spark Streaming. It represents a continuous stream of data, either the input data stream received from source, or the processed data stream generated by transforming the input stream. Internally, a DStream is represented by a continuous series of RDDs,

Window Operations

Spark Streaming also provides *windowed computations*, which allow you to apply transformations over a sliding window of data. The following figure illustrates this sliding window.



As shown in the figure, every time the window *slides* over a source DStream, the source RDDs that fall within the window are combined and operated upon to produce the RDDs of the windowed DStream. In this specific case, the operation is applied over the last 3 time units of data, and slides by 2 time units. This shows that any window operation needs to specify two parameters.

- *window length* - The duration of the window.
- *sliding interval* - The interval at which the window operation is performed

Checkpointing

A streaming application must operate 24/7 and hence must be resilient to failures unrelated to the application logic (e.g., system failures, JVM crashes, etc.). For this to be possible, Spark Streaming needs to *checkpoint* enough information to a fault- tolerant storage system such that it can recover from failures. There are two types of data that are checkpointed.

- *Metadata checkpointing* - Saving of the information defining the streaming computation to fault-tolerant storage like HDFS. This is used to recover from failure of the node running the driver of the streaming application (discussed in detail later). Metadata includes:
  - *Configuration* - The configuration that was used to create the streaming application.
  - *DStream operations* - The set of DStream operations that define the streaming application.
  - *Incomplete batches* - Batches whose jobs are queued but have not completed yet.
- *Data checkpointing* - Saving of the generated RDDs to reliable storage. This is necessary in some *stateful* transformations that combine data across multiple batches. In such transformations, the generated RDDs depend on RDDs of previous batches, which causes the length of the dependency chain to keep increasing with time. To avoid such unbounded increases in recovery time (proportional to dependency chain), intermediate RDDs of stateful transformations are periodically *checkpointed* to reliable storage (e.g. HDFS) to cut off the dependency chains.

Apache Spark: Structured Streaming

Overview

Structured Streaming is a scalable and fault-tolerant stream processing engine built on the Spark SQL engine. You can express your streaming computation the same way you would express a batch computation on static data. The Spark SQL engine will take care of running it incrementally and continuously and updating the final result as streaming data continues to arrive

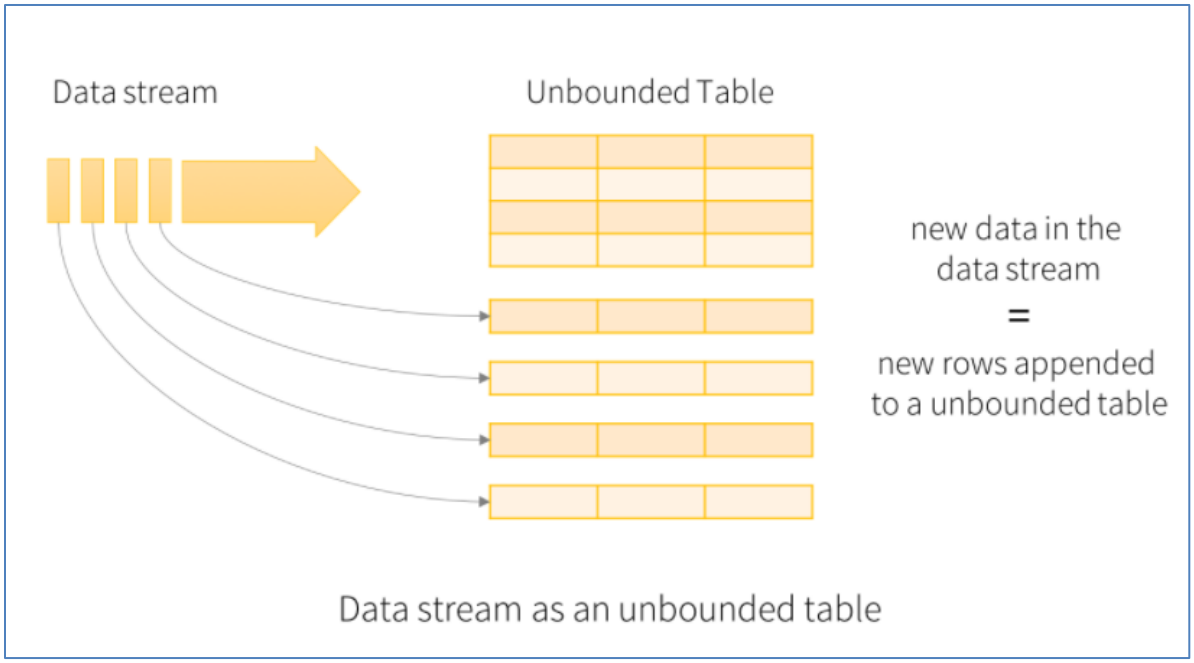
Internally, by default, Structured Streaming queries are processed using a *micro-batch processing* engine, which processes data streams as a series of small batch jobs thereby achieving end-to-end latencies as low as 100 milliseconds and exactly-once fault-tolerance guarantees

Programming Model

The key idea in Structured Streaming is to treat a live data stream as a table that is being continuously appended. This leads to a new stream processing model that is very similar to a batch processing model. You will express your streaming computation as standard batch-like query as on a static table, and Spark runs it as an *incremental* query on the *unbounded* input table.

Basic Concepts

Consider the input data stream as the “Input Table”. Every data item that is arriving on the stream is like a new row being appended to the Input Table.



Data stream as an unbounded table

Apache Spark: SparkR

Overview

SparkR is an R package that provides a light-weight frontend to use Apache Spark from R. In Spark 2.3.0, SparkR provides a distributed data frame implementation that supports operations like selection, filtering, aggregation etc. (similar to R data frames, dplyr) but on large datasets. SparkR also supports distributed machine learning using MLlib.

Starting Up: SparkSession

The entry point into SparkR is the SparkSession which connects your R program to a Spark cluster. You can create a SparkSession using sparkR.session and pass in options such as the application name, any spark packages depended on, etc. Further, you can also work with SparkDataFrames via SparkSession. If you are working from the sparkR shell, the SparkSession should already be created for you, and you would not need to call sparkR.session.

Starting Up from RStudio

You can also start SparkR from RStudio. You can connect your R program to a Spark cluster from RStudio, R shell, Rscript or other R IDEs. To start, make sure SPARK_HOME is set in environment (you can check Sys.getenv), load the SparkR package, and call sparkR.session as below. It will check for the Spark installation, and, if not found, it will be downloaded and cached automatically. Alternatively, you can also run install spark manually.

In addition to calling sparkR.session, you could also specify certain Spark driver properties. Normally these Application properties and Runtime Environment cannot be set programmatically, as the driver JVM process would have been started, in this case SparkR takes care of this for you. To set them, pass them as you would other configuration properties in the sparkConfig argument to sparkR.session().

```
if (nchar(Sys.getenv("SPARK_HOME")) < 1) {

 Sys.setenv(SPARK_HOME = "/home/spark")

}

library(SparkR, lib.loc = c(file.path(Sys.getenv("SPARK_HOME"), "R", "lib")))

sparkR.session(master = "local[*]", sparkConfig = list(spark.driver.memory = "2g"))
```

Examples

```
> people <-
read.df("C:/Users/bhumishr/Desktop/Spark_Scala_Training/data/people.json", "json")
> head(people)

> printSchema(people)

> df <- as.DataFrame(faithful)

> head(select(df, df$eruptions))

> head(select(df, "eruptions"))

> head(filter(df, df$waiting < 50))

> createOrReplaceTempView(people, "people")

> teenagers <- sql("SELECT firstname FROM people WHERE age >= 13 AND age <= 19")

> head(teenagers)
```

Spark Application Deployment

Spark deployment

- Executing inside Scala ID
- Executing Spark application by exporting as jar file
- Using sbt tool to build the jar

Anatomy of Spark Job

Job Submission

Spark job is submitted automatically as soon as action is performed on RDD. Internally it call runJob() on SparkContext which will pass the call to scheduler. Scheduler is made of two parts:

1. DAG scheduler: that breaks down the job into DAG of stages
2. Task scheduler: that is responsible for submitting the task from each stage to the cluster

DAG scheduler construct:

- Shuffle map tasks
    - Each shuffle map task run on one RDD partition. Based on the partition function it writes output to new set of partitions
- Result task
    - Each results task runs a computation on its RDD partition then send results back to the driver, and the driver assembles the results from each partition into a final results

DAG scheduler is responsible for splitting a Stages into tasks for submission to the task scheduler

Task Scheduling
When the task scheduler is sent a set of tasks, it uses its lists of executors that are running for the application and construct mapping of task to executor. Executor first assign task to local process then node-level then rack level and finally cluster level

Task Execution:

Executor first makes sure that jar and file dependencies are copied to local

Executor and Cluster Manager

Local: In the local mode there is single executor running in the same JVM as driver. the Master url for local mode is local [n]

Standalone: Simple distributed implementation that runs a single Spark Master and one or more workers. When a Spark Application starts, the master will ask workers to spawn executor process on behalf of the application

Master url is spark://localhost:7077

Spark on YARN: Running Spark on YARN provides the tightest integration with other Hadoop components. YARN is resource manager used in HADOOP

Each running Spark application corresponds to an instance of a YARN application and each executor runs in its own YARN container

The Master URL is yarn-client or yarn-cluster. Yarn client mode is required for programs that have any interactive component, such as spark-shell or pyspark

Yarn cluster mode is for batch mode and useful for production deployment

Syntax for submitting of Spark Job in batch mode

```
./bin/spark-submit \
        --class <main-class>
        --master <master-url> \
        --deploy-mode <deploy-mode> \
        --conf <key>=<value> \
        ... # other options
        <application-jar> \
        [application-arguments]
```

Some of the commonly used options are:

--class: The entry point for your application (e.g. org.apache.spark.examples.SparkPi)

--master: The master URL for the cluster (e.g. spark://23.195.26.187:7077)

--deploy-mode: Whether to deploy your driver on the worker nodes (cluster) or locally as an external client (client) (default: client) †

--conf: Arbitrary Spark configuration property in key=value format. For values that contain spaces wrap “key=value” in quotes (as shown).

application-jar: Path to a bundled jar including your application and all dependencies. The URL must be globally visible inside of your cluster, for instance, an hdfs:// path or a file:// path that is present on all nodes.

application-arguments: Arguments passed to the main method of your main class, if any

Examples:

Run application locally on 8 cores

```
./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master local[*] \
  /path/to/examples.jar \
  100
```

Run on a Spark standalone cluster in client deploy mode

```
./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master spark://207.184.161.138:7077 \
  --executor-memory 20G \
  --total-executor-cores 100 \
  /path/to/examples.jar \
  1000
```

Run on a Spark standalone cluster in cluster deploy mode with supervise

```
./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master spark://207.184.161.138:7077 \
  --deploy-mode cluster
  --supervise
  --executor-memory 20G \
  --total-executor-cores 100 \
  /path/to/examples.jar \
  1000
```

Note: Spark standalone cluster with cluster deploy mode, you can also specify --supervise to make sure that the driver is automatically restarted if it fails with non-zero exit code

Run on a YARN cluster

```
export HADOOP_CONF_DIR=XXX
./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master yarn \
  --deploy-mode cluster \  # can be client for client mode
  --executor-memory 20G \
  --num-executors 50 \
  /path/to/examples.jar \
  1000
```

Run a Python application on a Spark standalone cluster

```
./bin/spark-submit \
  --master spark://207.184.161.138:7077 \
  examples/src/main/python/pi.py \
  1000
```

The master URL passed to Spark can be in one of the following formats:

Master URL	Meaning
local	Run Spark locally with one worker thread (i.e. no parallelism at all).
local[K]	Run Spark locally with K worker threads (ideally, set this to the number of cores on your machine).
local[*]	Run Spark locally with as many worker threads as logical cores on your machine.
spark://HOST:PORT	Connect to the given Spark standalone cluster master. The port must be whichever one your master is configured to use, which is 7077 by default.
yarn	Connect to a YARN cluster in client or cluster mode depending on the value of --deploy-mode. The cluster location will be found based on the HADOOP_CONF_DIR or YARN_CONF_DIR variable.
yarn-client	Equivalent to yarn with --deploy-mode client, which is preferred to `yarn-client`
yarn-cluster	Equivalent to yarn with --deploy-mode cluster, which is preferred to `yarn-cluster`

yarn-client: In YARN client mode, the interaction with YARN starts when a new SparkContext instance is constructed by the driver programs.

- The context submits a YARN application to the YARN resource manager

- YARN RM start a YARN container on a Node Manager in the cluster and runs SparkExecutorLauncher application Master in it.

- The job of ExecutorLauncher is to start executors in YARN container, which it does by requesting resources from the RM. Then launching Executor Backend process as the container are allocated to it

Yarn-cluster: In YARN cluster mode, the users' driver program runs in a YARN application master process.

- The spark-submit client will launch the YARN application but it does not run any user code.

- Application Master starts the driver program before allocating resources for the executor