TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA

Macroarea di Ingegneria
Dipartimento di Ingegneria Civile e Ingegneria Informatica

Apache Spark

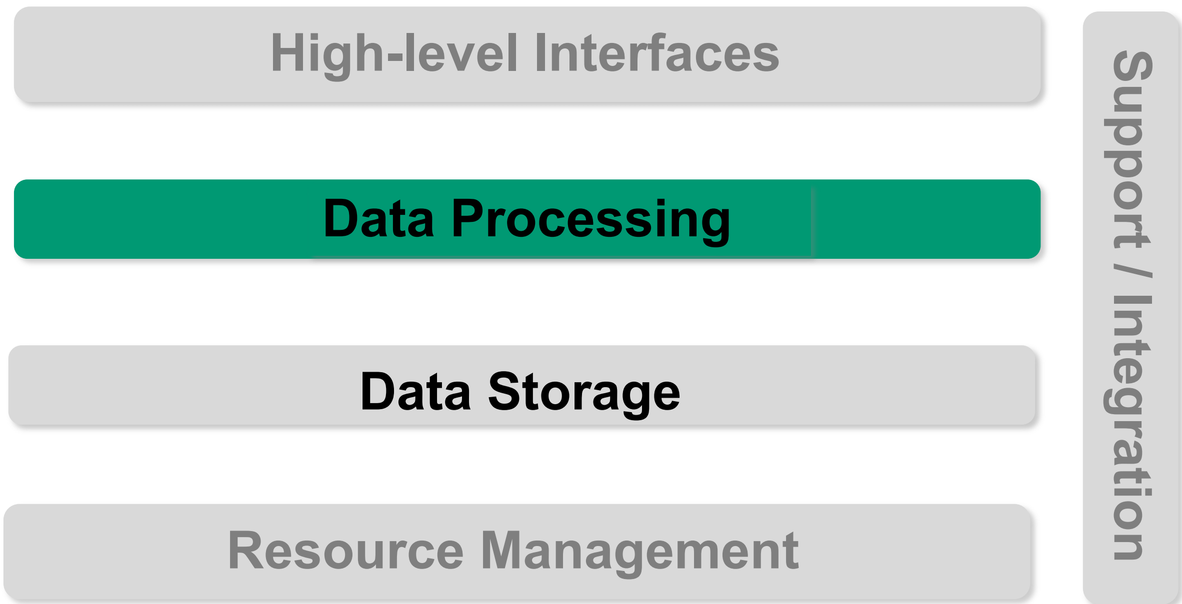
Corso di Sistemi e Architetture per Big Data

A.A. 2022/23

Valeria Cardellini

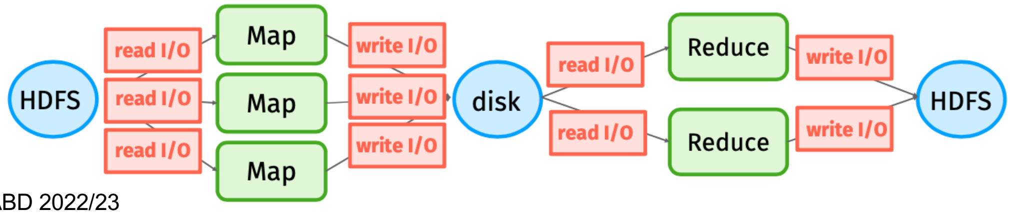
Laurea Magistrale in Ingegneria Informatica

The reference Big Data stack



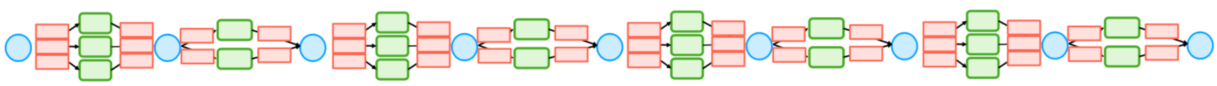
MapReduce (MR): limitations

- Programming model
  - Hard to implement everything as a MR program
  - Multiple MR steps even for simple tasks
    - E.g., sorting words by their frequency requires two MR steps
  - Lack of control, structures and data types
- Efficiency (recall HDFS)
  - High communication cost: compute (map), communicate (shuffle), compute (reduce)
  - Read input and store output from/on disk
  - Limited exploitation of main memory



MapReduce: limitations

- Lack of native support for iteration
  - Each step writes/reads data from disk: I/O overhead
  - But real-world applications (e.g., ML algorithms) require iterating MR steps
    - Partial solution: design algorithms that minimize the number of iterations



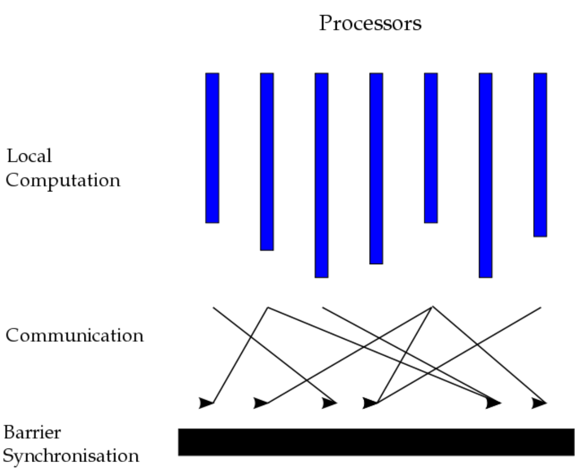
- Not feasible for real-time data stream processing
  - MR job requires to scan entire input before processing it

Alternative programming models

- Based on *directed acyclic graphs* (**DAGs**)
  - Spark, Spark Streaming, Storm, Flink, …
  - Each vertex is an operation and edges represent dependencies (data flow) of each operation
- SQL-based
  - Hive, Spark SQL, Vertica, …
- NoSQL data stores
  - HBase, MongoDB, Cassandra, …
- Based on *Bulk Synchronous Parallel* (**BSP**)

Alternative programming models: BSP

- Bulk Synchronous Parallel (BSP)
  - Developed by Leslie Valiant during 1980s
  - Considers communication actions *en masse*
  - Suitable for graph analytics at massive scale and massive scientific computations (e.g., matrix, graph and network algorithms)
  - Examples: Google's Pregel, Apache Giraph to perform graph processing on big data



Apache Spark

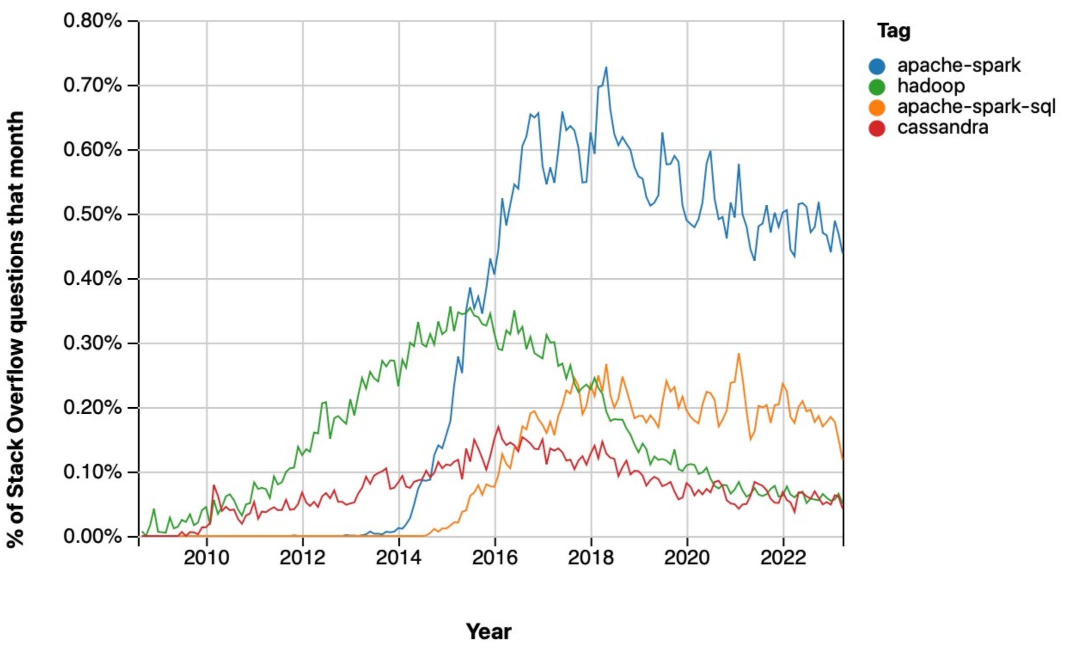
- Unified engine for large-scale data analytics
  – Not a modified version of Hadoop
  – Leading platform for batch/stream processing, large-scale SQL, and machine learning on single-node machines or clusters
  – Multi-language: Scala, Python, Java and R
- In-memory data storage for fast iterative processing
  – At least 10x faster than Hadoop
- Suitable for execution of DAGs and powerful optimization
- Compatible with Hadoop's storage APIs
  – Can read/write to any Hadoop-supported system, including HDFS and HBase

Spark milestones

- Spark project started in 2009
- Developed originally at UC Berkeley's AMPLab by Matei Zaharia for his PhD thesis
- Open sourced in 2010, Apache project from 2013
- In 2014, Zaharia founded Databricks
- Current release: 3.4.0
- The most active open source project for Big Data processing, see next slide

Spark popularity

- Based on Stack Overflow Trends



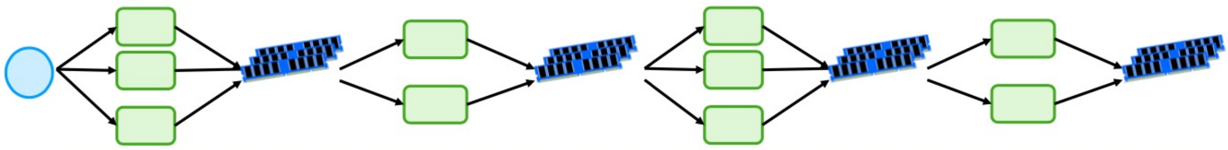
Spark: why a new programming model?

- MapReduce simplified Big Data analysis
  - But executes jobs in a simple but rigid structure
    - Step to process or transform data (map)
    - Step to synchronize (shuffle)
    - Step to combine results (reduce)

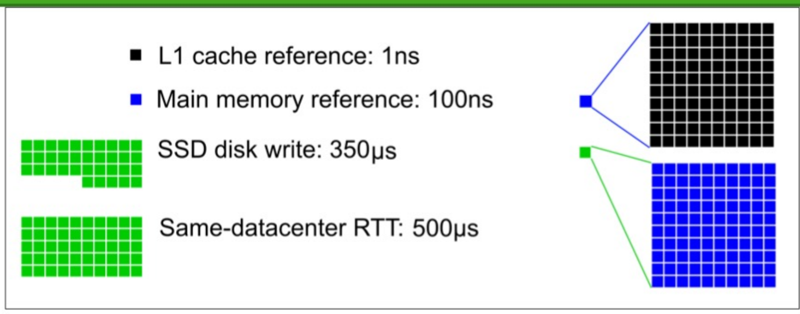
- As soon as MapReduce got popular, users wanted:
  - Iterative computations, e.g., graph and ML algorithms
  - Interactive ad-hoc queries
  - More efficiency
  - Faster **in-memory data sharing** across parallel jobs (required by both iterative and interactive applications)

Spark: In-memory computation

- Key idea: keep and share datasets in main memory
- Distributed in-memory: 10x-100x faster than disk and network



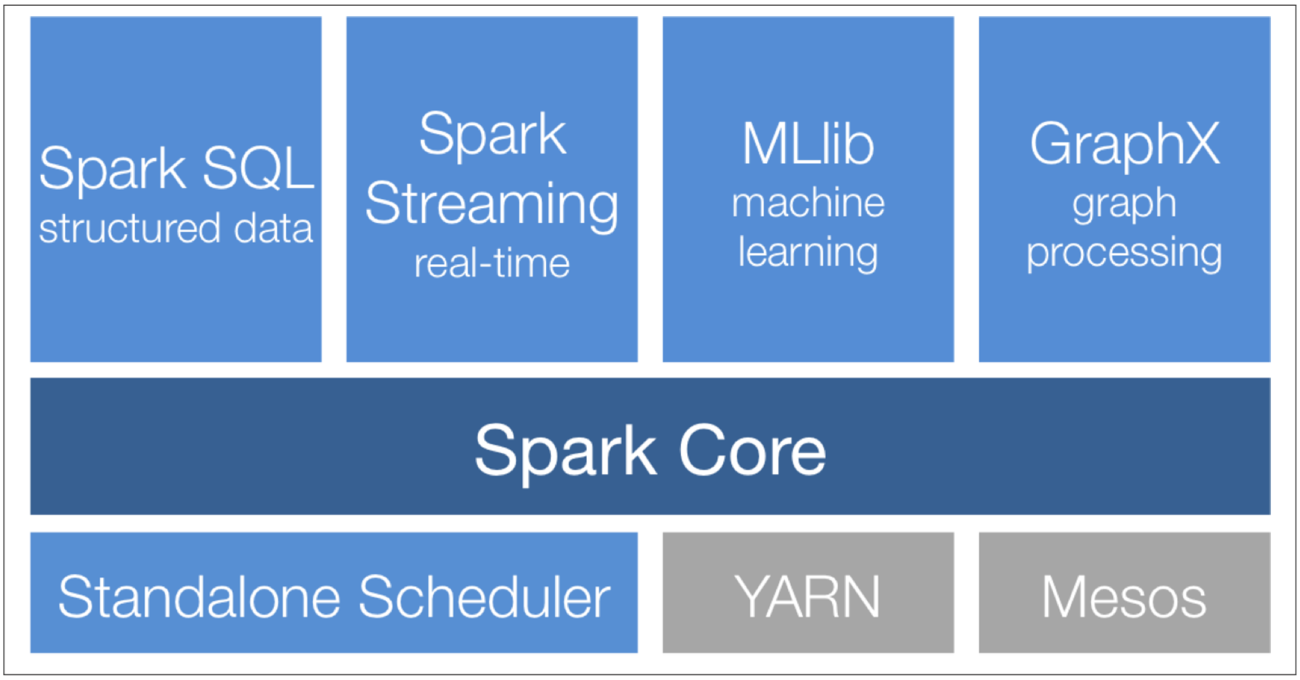
Much faster response time (in practice: 10x-100x)



Spark vs Hadoop MapReduce

- Underlying programming paradigm similar to MapReduce
  – Basically "scatter-gather": scatter data and computation on multiple cluster nodes that run in parallel processing on data portions; gather final results
- Spark offers a more general data model
  – RDDs, DataSets, DataFrames
- Spark offers a more general and developer-friendly programming model
  – Map -> **Transformations** in Spark
  – Reduce -> **Actions** in Spark
- Spark is storage agnostic
  – Not only HDFS, but also Cassandra, S3, Parquet files, …

Spark stack



Spark core

- Provides basic functionalities (including task scheduling, memory management, fault recovery, interacting with storage systems) used by other components
- Provides a data abstraction called **resilient distributed dataset (RDD)**, a collection of items distributed across many compute nodes that can be manipulated in parallel
  – Spark Core provides APIs for building and manipulating these collections
- Written in Scala but APIs for Java, Python and R

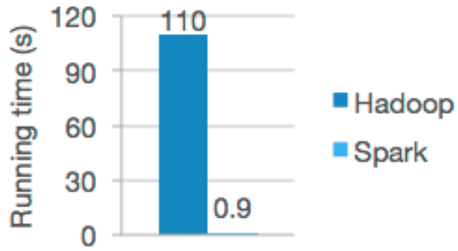
Spark as unified analytics engine

- A number of integrated higher-level modules built on top of Spark
  – Can be combined seamlessly in the same application
- **Spark SQL**
  – To work with structured data
  – Allows querying data via SQL
  – Supports many data sources (Hive tables, Parquet, JSON, …)
  – Extends Spark RDD API
- **Spark Streaming**
  – To process live streams of data
  – Extends Spark RDD API

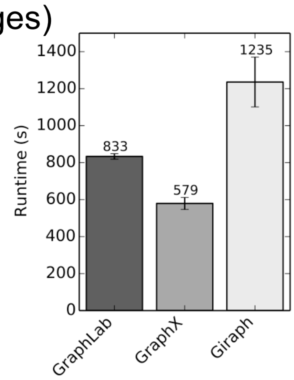
Spark as unified analytics engine

- **MLlib**
  – Scalable ML library
  – Many distributed algorithms: feature extraction, classification, regression, clustering, recommendation, …

Logistic regression performance



PageRank performance (20 iterations, 3.7B edges)



- **GraphX**
  – API for manipulating graphs and performing graph-parallel computations
  – Includes also common graph algorithms (e.g., PageRank)
  – Extends Spark RDD API

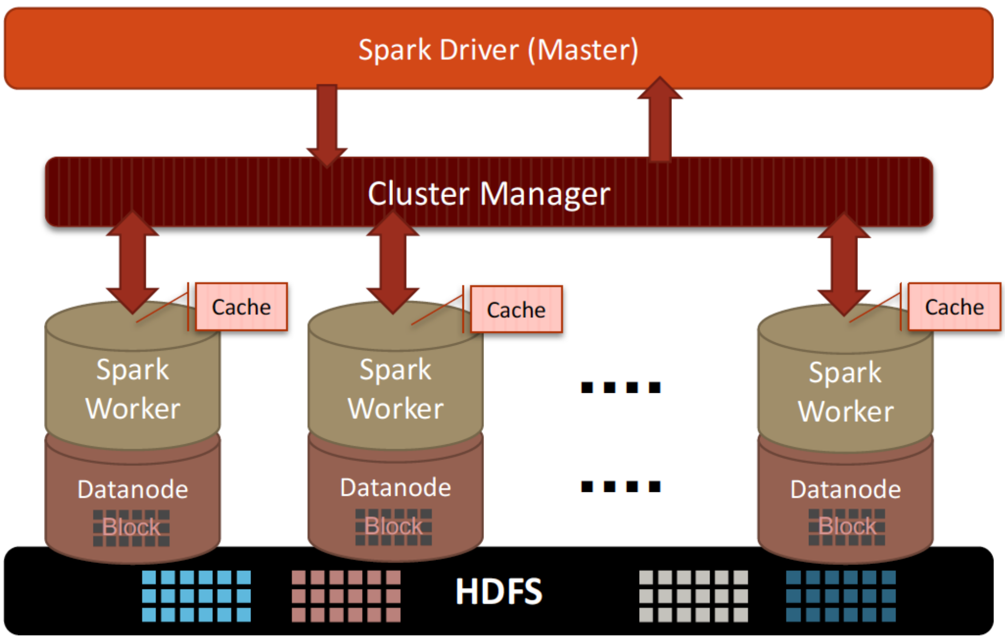
Spark on top of cluster managers

- Spark can exploit many cluster resource managers which allocate cluster resources to run the applications

1. Standalone
   - Simple cluster manager included with Spark that makes it easy to set up a cluster
2. Hadoop YARN
   - Resource manager in Hadoop 2
3. Mesos
   - General cluster manager from AMPLab
4. Kubernetes

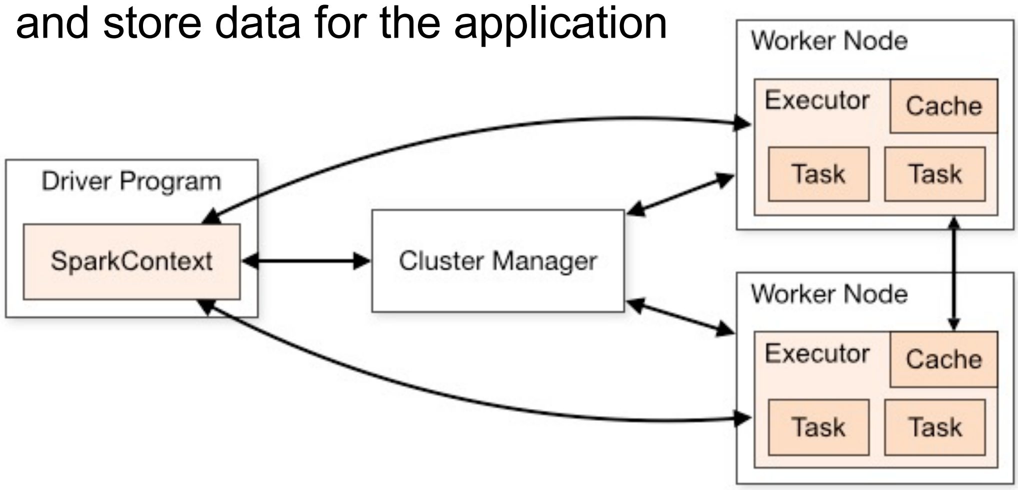
Spark architecture

- Master/worker architecture



Spark architecture

- Main program (called driver program) connects to cluster manager, which allocates resources
- Worker nodes in which executors run
- Executors are processes that run computations and store data for the application

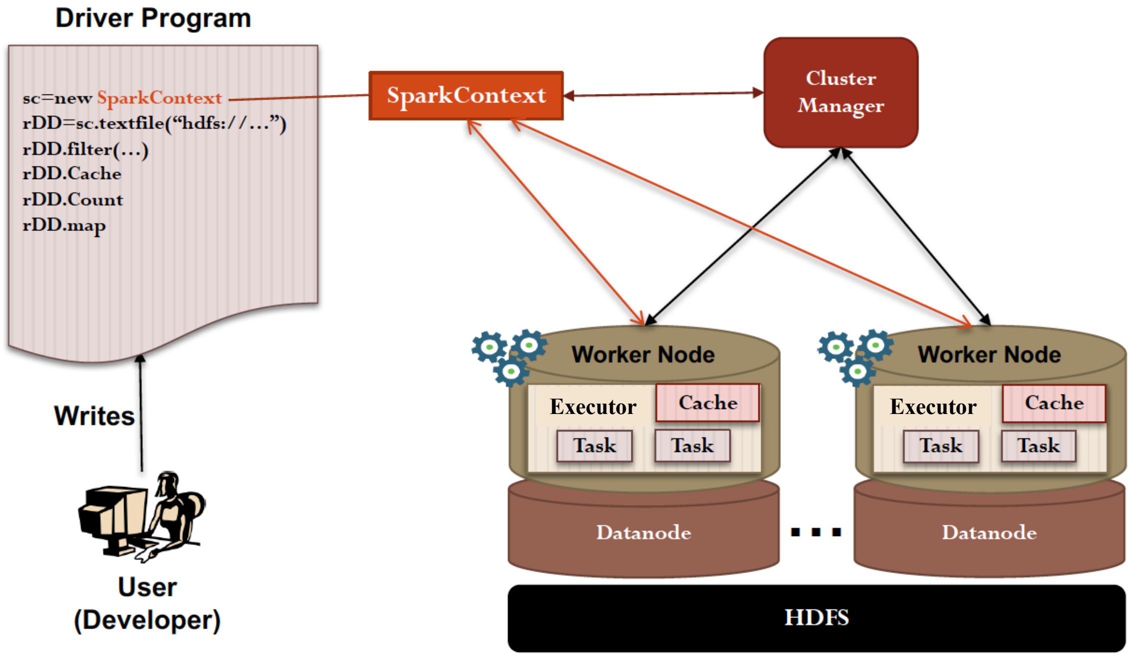


spark.apache.org/docs/latest/cluster-overview.html

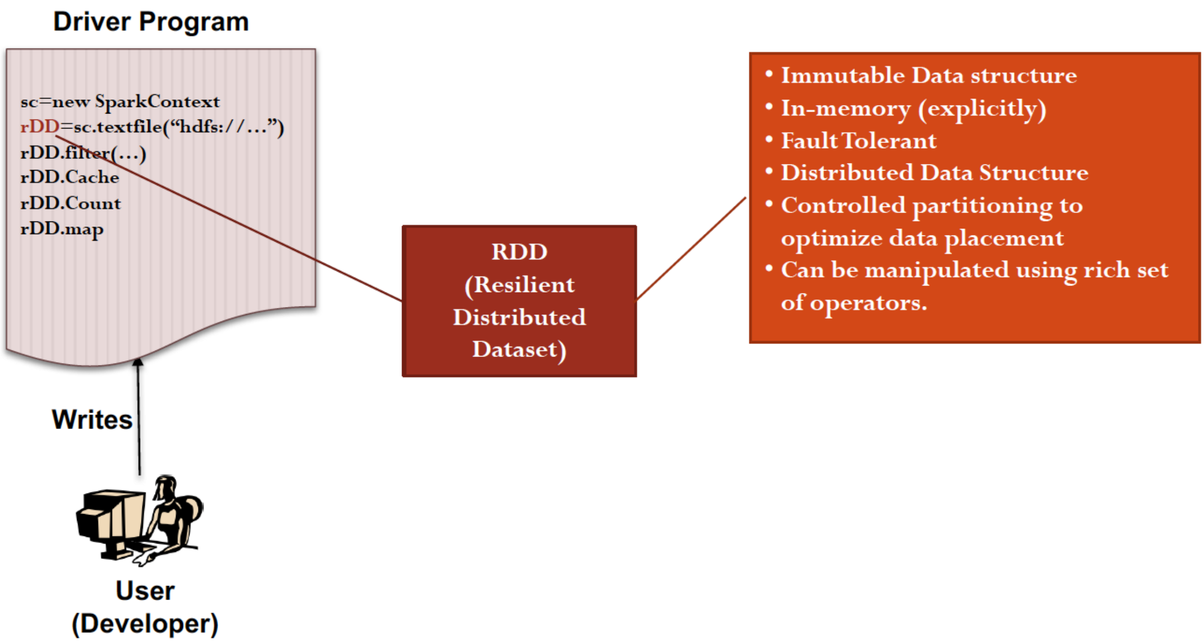
Spark architecture

- Each application consists of a driver program and executors on the cluster
  - Driver program: process which runs application `main()` and creates SparkContext object
- Each application gets its own executors, which are processes which stay up for the duration of the whole application and run tasks in multiple threads
  - Isolation of concurrent applications
- To run on a cluster, SparkContext connects to cluster manager, which allocates cluster resources
- Once connected, Spark acquires executors on cluster nodes and sends the application code (e.g., jar) to executors
- Finally, SparkContext sends tasks to executors to run

Spark programming model



Spark programming model

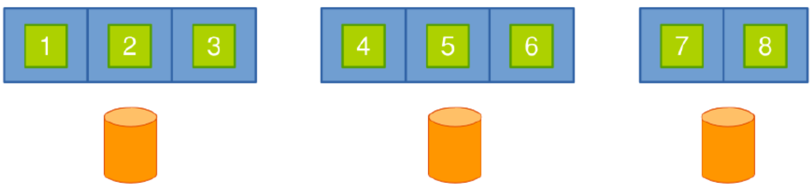


Resilient Distributed Datasets (RDDs)

- RDDs are the key programming abstraction in Spark: a distributed memory abstraction
- Immutable, partitioned and fault-tolerant collection of elements that can be manipulated in parallel
  - Like a LinkedList <MyObjects>
  - Stored in main memory across the cluster nodes
    - Each worker node that is used to run an application contains at least one partition of the RDD(s) that is (are) defined in the application

RDDs: distributed and partitioned

- Stored in main memory of the executors running in the worker nodes (when it is possible) or on node local disk (if not enough main memory)
- Allow executing in parallel the code invoked on them
  - Each executor of a worker node runs the specified code on its partition of the RDD
  - Partition: atomic chunk of data (a logical division of data) and basic unit of parallelism
  - Partitions of an RDD can be stored on different cluster nodes



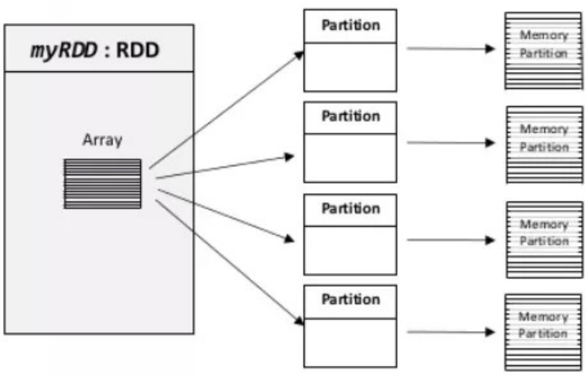
RDDs: immutable and fault-tolerant

- Immutable once constructed
  - RDD content cannot be modified
  - New RDD is created from existing RDD(s)

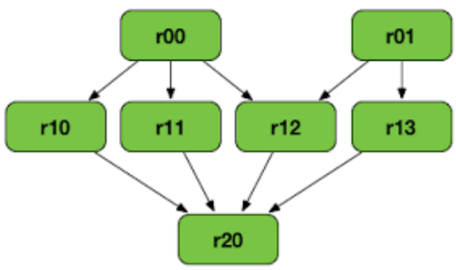
- Automatically rebuilt on failure (without replication)
  - Track lineage information so to efficiently recompute missing or lost data due to (node) failures
  - For each RDD, Spark knows how it has been constructed and can rebuild it if a failure occurs
  - This information is represented by means of RDD lineage DAG which keeps track of one or more operations that lead to the creation of that RDD

RDD: Spark management

- Spark manages the split of RDDs in partitions and allocates RDDs' partitions to cluster nodes



- Spark hides complexity of fault tolerance
  - RDDs are automatically rebuilt in case of failure using the RDD lineage DAG, that defines the logical execution plan



RDD: API and suitability

- RDD API
  - Clean language-integrated API for Scala, Python, Java, and R
  - Can be used interactively from console (Scala and PySpark)
- RDD suitability
  - Best suited for apps that apply the same operation to all the elements in dataset
  - Provides fine-grained control over physical distribution of data
  - Not a good fit for apps with fine-grained updates to shared state
- Also higher-level APIs: DataFrame and DataSet

Python Spark (PySpark)

- PySpark: Python API for Spark supporting the collaboration of Spark and Python
- Using PySpark, you can work with RDDs in Python
- PySpark shell for interactive analysis

PySpark: SparkContext

- SparkContext: entry point for low-level API functionalities, the connection to a Spark cluster

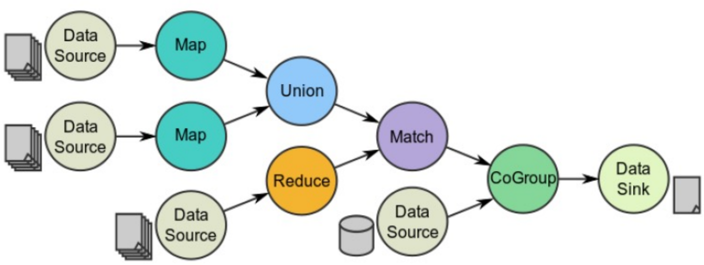
```
conf = SparkConf().setAppName(appName).setMaster(master)
sc = SparkContext(conf=conf)
```

- Used to set various Spark parameters, among which
  - `master`: URL of cluster to connect to
  - `appName`: name of job to run
- When using shell, it is created as `sc` variable

See spark.apache.org/docs/latest/api/python

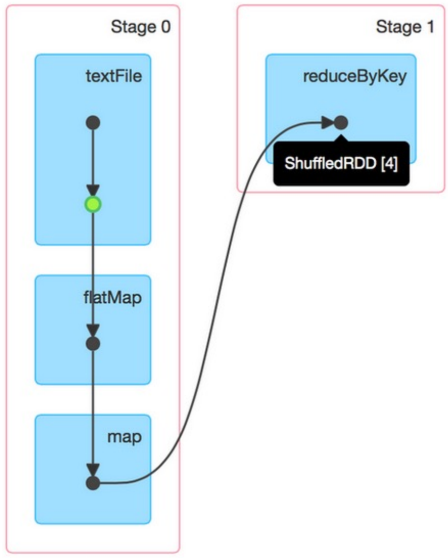
Spark programming model: DAG

- Data flow is composed of any number of data sources, operators, and data sinks by connecting their inputs and outputs
- A Directed Acyclic Graph (DAG) in Spark is a set of vertices and edges, where vertices represent the RDDs and edges represent the operations to be applied on RDDs
  - Generalization of MapReduce model, which has only two operations (Map and Reduce)



Spark programming model: DAG

- DAG can be visualized using Spark Web UI
  – See figure: DAG for WordCount
- DAG is divided into stages
- A stage is a set of operations that do not involve a shuffle of data
- As soon as a shuffle of data is needed (when a wide transformation is performed), the DAG will yield a new stage



Operations in RDD API

- Spark programs are written in terms of operations on RDDs
- Programming model based on parallelizable operations
  – Higher-order functions that execute user-defined functions in parallel
- RDDs are created from external data or other RDDs
- RDDs are created and manipulated through operators

See spark.apache.org/docs/latest/rdd-programming-guide.html

RDD operations

- RDD operations: higher-order functions
- Two types of RDD operations: transformations and actions
- **Transformations**: coarse-grained and lazy operations that define new RDD based on previous one(s)
  - map, filter, join, union, distinct, …
  - **lazy**: the new RDD representing the result of a computation is not immediately computed but is materialized on demand when an action is called
- **Actions**: operations that kick off a job to execute on a cluster and return a value to the driver program after running a computation on RDD or write data to external storage
  - count, collect, save, …

Transformations and actions on RDDs

- Common transformations and actions on RDDs
  - Seq[T]: sequence of elements of type T

spark.apache.org/docs/latest/rdd-programming-guide.html#transformations

spark.apache.org/docs/latest/rdd-programming-guide.html#actions

Transformations	$map(f : \mathrm{T} \Rightarrow \mathrm{U})$	: RDD[T] ⇒ RDD[U]
	$filter(f : \mathrm{T} \Rightarrow \mathrm{Bool})$	: RDD[T] ⇒ RDD[T]
	$flatMap(f : \mathrm{T} \Rightarrow \mathrm{Seq[U]})$	: RDD[T] ⇒ RDD[U]
	$sample(fraction : \mathrm{Float})$	: RDD[T] ⇒ RDD[T] (Deterministic sampling)
	$groupByKey()$	: RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
	$reduceByKey(f : (\mathrm{V},\mathrm{V}) \Rightarrow \mathrm{V})$	: RDD[(K, V)] ⇒ RDD[(K, V)]
	$union()$	: (RDD[T], RDD[T]) ⇒ RDD[T]
	$join()$	: (RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]
	$cogroup()$	: (RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]
	$crossProduct()$	: (RDD[T], RDD[U]) ⇒ RDD[(T, U)]
	$mapValues(f : \mathrm{V} \Rightarrow \mathrm{W})$	: RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)
	$sort(c : \mathrm{Comparator[K]})$	: RDD[(K, V)] ⇒ RDD[(K, V)]
	$partitionBy(p : \mathrm{Partitioner[K]})$	: RDD[(K, V)] ⇒ RDD[(K, V)]
Actions	$count()$	: RDD[T] ⇒ Long
	$collect()$	: RDD[T] ⇒ Seq[T]
	$reduce(f : (\mathrm{T},\mathrm{T}) \Rightarrow \mathrm{T})$	: RDD[T] ⇒ T
	$lookup(k : \mathrm{K})$	: RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs)
	$save(path : \mathrm{String})$	: Outputs RDD to a storage system, *e.g.*, HDFS

How to create RDD

- RDD can be created by:
  - Parallelizing existing data collections of the hosting programming language (e.g., collections and lists of Scala, Java, Python, or R)
    - Number of partitions specified by user
    - RDD API: `parallelize`
  - From (large) files stored in HDFS or any other file system
    - One partition per HDFS block
    - RDD API: `textFile`
  - Transforming an existing RDD
    - Number of partitions depends on transformation type
    - RDD API: transformation operations (`map`, `filter`, `flatMap`)

How to create RDD

- Turn an existing collection into an RDD

```
lines = sc.parallelize(["pandas", "i like pandas"])
```

  - sc is Spark context variable
  - Important parameter: number of partitions to cut the dataset into
  - Spark will run one task for each partition of the cluster (typical setting: 2-4 partitions for each CPU in the cluster)
  - Spark tries to set the number of partitions automatically
  - You can also set it manually by passing it as a second parameter to parallelize, e.g., `sc.parallelize(data, 10)`

- Load data from storage (local file system, HDFS, or S3)

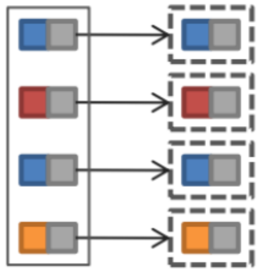
```
lines = sc.textFile("/path/input.txt")
```

Examples in Python

RDD transformations: map and filter

- `map`: takes as input a function which is applied to each element of the RDD and maps each input element to another element

```
# transform each element through a function
nums = sc.parallelize([1, 2, 3, 4])
squares = nums.map(lambda x: x * x)  # [1,4,9,16]
```



- `filter`: generates a new RDD by filtering the source dataset using the specified function

```
# select those elements that func returns true
even = squares.filter(lambda num: num % 2 == 0)  # [4,16]
```


RDD transformations: flatMap

- `flatMap`: takes as input a function which is applied to each element of the RDD; can map each input item to zero or more output items

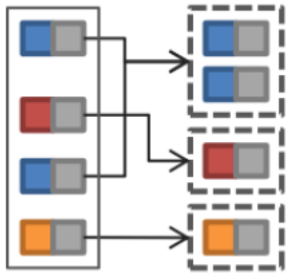
```
# map each element to zero or more others
ranges = nums.flatMap(lambda x: range(0, x, 1))
# [0, 0, 1, 0, 1, 2, 0, 1, 2, 3]
```

range function in Python: ordered sequence of integer values in range [start;end) with non-zero step

```
# split input lines into words
lines = sc.parallelize(["hello world", "hi"])
words = lines.flatMap(lambda line: line.split(" "))
#['hello', 'world', 'hi']
```

RDD transformations: reduceByKey

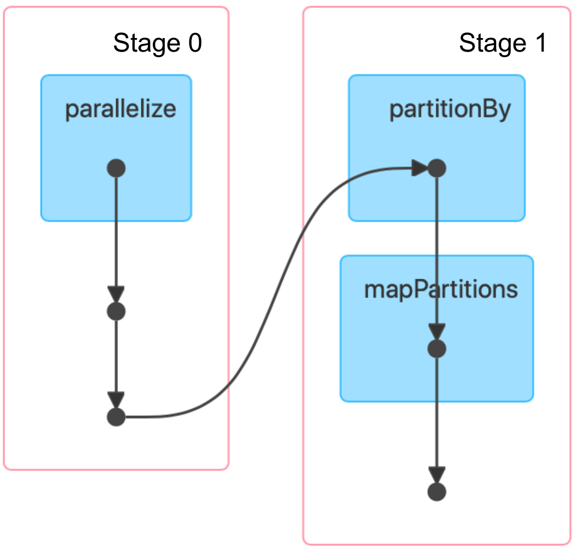
- `reduceByKey`: aggregates values with identical key using the specified function
- Runs several parallel reduce operations, one for each key in the dataset, where each operation combines values that have the same key



```
x = sc.parallelize([("a", 1), ("b", 1), ("a", 1), ("a", 1),
... ("b", 1), ("b", 1), ("b", 1), ("b", 1)], 3)

# apply reduceByKey operation
y = x.reduceByKey(lambda accum, n: accum + n)
# [('b', 5), ('a', 3)]
```

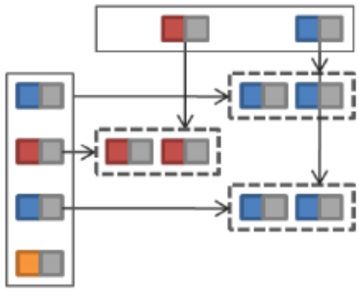

RDD transformations: reduceByKey

- Let's see the corresponding DAG



RDD transformations: join

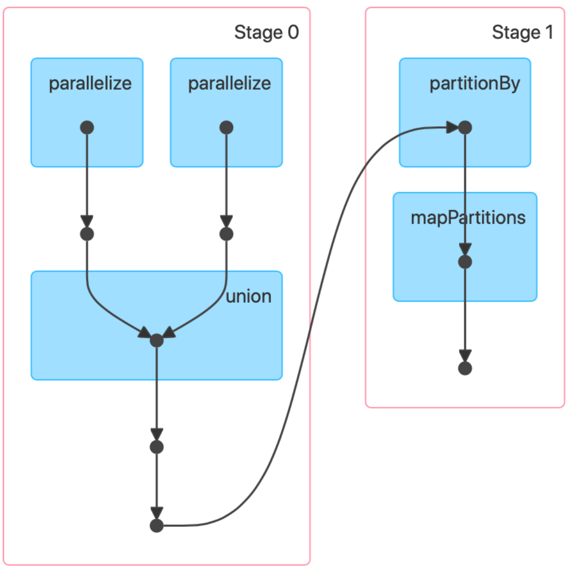
- `join`: performs an inner-join on the keys of two RDDs
- Only keys that are present in both RDDs are output
- Join candidates are independently processed



```
users = sc.parallelize([(0, "Alex"), (1, "Bert"), (2, "Curt"),
(3, "Don")])
hobbies = sc.parallelize([(0, "writing"), (0, "gym"), (1,
"swimming")])
users.join(hobbies).collect()
# [(0, ('Alex', 'writing')), (0, ('Alex', 'gym')), (1,
('Bert', 'swimming'))]
```


RDD transformations: join

- Let's see the corresponding DAG



Other useful transformations

- `union`: returns a new RDD that contains the union of the elements in the source RDD and the argument
- `partitionBy`: returns a copy of the RDD partitioned using the specified partitioner
- `mapPartitions`: similar to `map`, but runs separately on each partition
- `distinct`: returns a new RDD that contains the distinct elements of the source RDD
- `groupByKey`: when called on a key-value paio RDD, groups the values for each key in the RDD into a single sequence
- `mapValues`: passes each value in the key-value pair RDD through a map function without changing the keys

Transformations and actions

- Transformations are **lazy**
  - Are not computed till an action requires a result to be returned to the driver program
  - Spark can build up the logical transformation plan
- This design enables Spark to perform operations more efficiently as they can be grouped together
  - E.g., if there were multiple filter or map operations, Spark can fuse them into one pass
  - E.g., if Sparks knows that data is partitioned, it can avoid moving it over the network for groupBy
- We run an action to trigger the computation
  - Instructs Spark to compute a result from a series of transformations

Some RDD actions

- `collect`: returns all the elements of the RDD as a list

```
nums = sc.parallelize([1, 2, 3, 4])
nums.collect()      # [1, 2, 3, 4]
```

- `take`: returns an array with the first *n* elements in the RDD

```
nums.take(3) # [1, 2, 3]
```

- `count`: returns the number of elements in the RDD

```
nums.count() # 4
```


Some RDD actions

- `reduce`: aggregates the elements in the RDD using the specified function

```
sum = nums.reduce(lambda x, y: x + y)
```

- `saveAsTextFile`: writes the elements of the RDD as a text file either to the local file system or HDFS

```
nums.saveAsTextFile("hdfs://file.txt")
```

Your very first examples in Spark

- After having installed Spark (e.g., Bitnami image), you can run the fragments of code using PySpark by a terminal window
  - `sc` is Spark context variable

```
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 3.2.1
      /_/

Using Python version 3.8.13 (default, Apr 11 2022 12:27:15)
Spark context Web UI available at http://4fec1336ba80:4040
Spark context available as 'sc' (master = local[*], app id = local-1651133501036).
SparkSession available as 'spark'.
>>> nums = sc.parallelize([1, 2, 3, 4])
>>> squares = nums.map(lambda x: x * x)
>>> nums.collect()
[1, 2, 3, 4]
>>> squares.collect()
[1, 4, 9, 16]
>>> 
```


First examples

- Let’s first analyze two simple examples using RDD API spark.apache.org/examples.html
  - Pi estimation
  - WordCount

- More examples: see those distributed with Spark, e.g.,
  - Java
    github.com/apache/spark/tree/master/examples/src/main/java/org/apache/spark/examples
  - Python
    github.com/apache/spark/tree/master/examples/src/main/python

Example: Pi estimation in Python

```
def inside(p):
  x, y = random.random(), random.random()
  return x*x + y*y < 1

samples = sc.parallelize(range(0, NUM_SAMPLES))
within_circle = samples.filter(inside)
count = within_circle.count()
print("Pi is roughly %f" % (4.0 * count / NUM_SAMPLES))
```


Example: Pi estimation in Python with chaining

- Transformations and actions can be chained together

```
def inside(p):
  x, y = random.random(), random.random()
  return x*x + y*y < 1
count = sc.parallelize(range(0, NUM_SAMPLES)) \
      .filter(inside).count()
print("Pi is roughly %f" % (4.0 * count / NUM_SAMPLES))
```

Example: Pi estimation in Scala

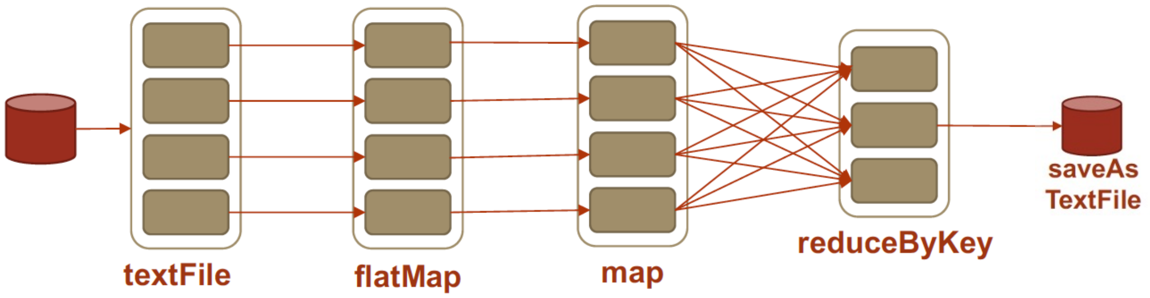
```
val count = sc.parallelize(1 to NUM_SAMPLES).filter { _ =>
  val x = math.random
  val y = math.random
  x*x + y*y < 1
}.count()
println(s"Pi is roughly ${4.0 * count / NUM_SAMPLES}")
```

- To run Spark shell in Scala

```
$ spark-shell
```


Example: WordCount in Python

```
text_file = sc.textFile("hdfs://inputfile")

counts = text_file.flatMap(lambda line: line.split(" ")) \
       .map(lambda word: (word, 1)) \
       .reduceByKey(lambda a, b: a + b)

counts.saveAsTextFile("hdfs://output")
```



Example: WordCount in Python

- Alternative solution: use `countByValue`
  - Action that returns the count of each unique value in the RDD as a dictionary of (value, count) pairs
  - Implemented with reduce: the driver will collect the partial results of the partitions and does the merge itself

```
text_file = sc.textFile("hdfs://inputfile")
counts = text_file.flatMap(lambda line: line.split(" "))
wordCount = words.countByValue()
print(wordCount)
```

- Which solution is better? Depends on dataset size
  - Large dataset: use `map`, `reduceByKey` and `collect` to exploit parallelism of `reduceByKey`
  - Small dataset: `countByValue` may introduce less network traffic (one stage less)

Lambda expressions in Java

- Lambda expressions are short blocks of code which take in parameters and return a value
  - Enable to treat functionality as method argument, or code as data
- Similar to methods (anonymous methods, i.e., methods without names), but do not need a name and can be implemented in the body itself
- Usually passed as parameters to a function
- Arrow operator `->` divides the lambda expressions in two parts
  - Left side: parameters required by lambda expression
  - Right side: actions of lambda expression

Example: Pi estimation in Java

- Transformations and actions can be chained together

```
List<Integer> l = new ArrayList<>(NUM_SAMPLES);
for (int i = 0; i < NUM_SAMPLES; i++) {
    l.add(i);
}
long count = sc.parallelize(l).filter(i -> {
   double x = Math.random();
   double y = Math.random();
   return x*x + y*y < 1;
}).count();
System.out.println("Pi is roughly " + 4.0 * count / NUM_SAMPLES);
```


Example: WordCount in Java

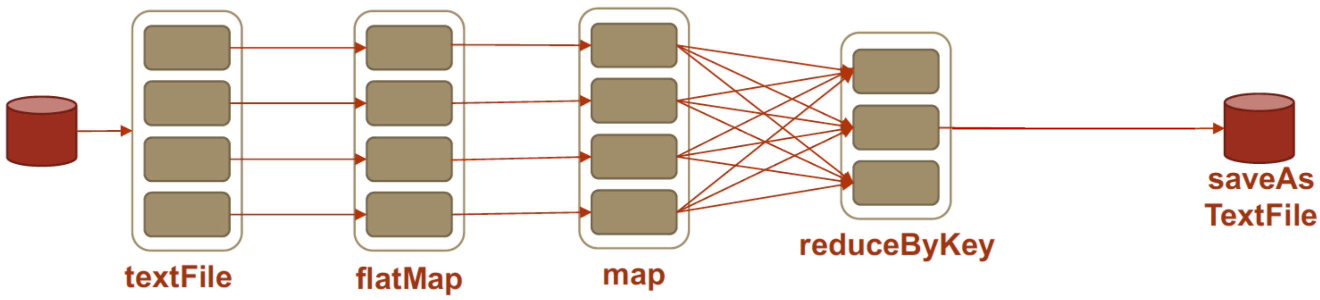
- `JavaPairRDD`: RDD containing key/value pairs
- Spark's Java API allows to create tuples using `scala.Tuple2` class

```
JavaRDD<String> lines = sc.textFile("hdfs://inputfile");
JavaRDD<String> words = lines.flatMap(line ->
       Arrays.asList(SPACE.split(line).iterator());
JavaPairRDD<String, Integer> ones = words.mapToPair(w ->
       new Tuple2<>(w, 1));
JavaPairRDD<String, Integer> counts = ones.reduceByKey((x, y) ->
       x+y);
counts.saveAsTextFile("output");
```

Example: WordCount in Java

- Same code but with chaining

```
JavaRDD<String> lines = sc.textFile("hdfs://inputfile");
JavaPairRDD<String, Integer> counts = lines
    .flatMap(s -> Arrays.asList(SPACE.split(line)).iterator())
    .mapToPair(w -> new Tuple2<>(w, 1))
    .reduceByKey((x, y) -> x + y);
counts.saveAsTextFile("output");
```



Initializing Spark: SparkContext

- First step in Spark program using RDD API: create SparkContext object
  - Represents connection to Spark cluster, can be used to create RDDs on that cluster
- SparkConf object: configuration for a Spark application
  - Used to set various Spark parameters as key-value pairs

```
SparkConf().setMaster("local").setAppName("My app")
```

- Only one SparkContext may be active per JVM
  - `stop()` active SparkContext before creating a new one

WordCount in Java: using SparkContext

Full example in Java using SparkContext and API RDD

```java
package org.apache.spark.examples;

import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaSparkContext;
import scala.Tuple2;

import java.util.Arrays;
import java.util.List;
import java.util.regex.Pattern;

public final class WordCount {
        private static final Pattern SPACE = Pattern.compile(" ");

        public static void main(String[] args) throws Exception {
                if (args.length < 1) {
                        System.err.println("Usage: WordCount <file>");
                        System.exit(1);
                }

                final SparkConf sparkConf = new SparkConf().setAppName("WordCount");
                final JavaSparkContext ctx = new JavaSparkContext(sparkConf);
                final JavaRDD<String> lines = ctx.textFile(args[0], 1);
```


WordCount in Java: using SparkContext

```java
                final JavaRDD<String> words = lines.flatMap(s -> Arrays.asList(SPACE.split(s)));
                final JavaPairRDD<String, Integer> ones = words.mapToPair(s -> new Tuple2<>(s, 1));
                final JavaPairRDD<String, Integer> counts = ones.reduceByKey((i1, i2) -> i1 + i2);

                final List<Tuple2<String, Integer>> output = counts.collect();
                for (Tuple2 tuple : output) {
                        System.out.println(tuple._1() + ": " + tuple._2());
                }
                ctx.stop();
        }
}
```

SparkSession

- From Spark 2.0, SparkSession unifies the different contexts from different APIs and represents the entry point into all functionalities in Spark
- Available from Spark shell as variable `spark`
- Within application: use `builder` to create and configure SparkSession

Python

```python
from pyspark.sql import SparkSession

spark = SparkSession \
    .builder \
    .appName("Python Spark SQL basic example") \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()
```

Java

```java
import org.apache.spark.sql.SparkSession;

SparkSession spark = SparkSession
  .builder()
  .appName("Java Spark SQL basic example")
  .config("spark.some.config.option", "some-value")
  .getOrCreate();
```


Pi estimation in Python: using SparkSession

Full example in Python using SparkSession and API RDD

```python
import sys
from random import random
from operator import add

from pyspark.sql import SparkSession


if __name__ == "__main__":
    """
        Usage: pi [partitions]
    """
    spark = SparkSession\
        .builder\
        .appName("PythonPi")\
        .getOrCreate()

    partitions = int(sys.argv[1]) if len(sys.argv) > 1 else 2
    n = 100000 * partitions

    def f(_: int) -> float:
        x = random() * 2 - 1
        y = random() * 2 - 1
        return 1 if x ** 2 + y ** 2 <= 1 else 0

    count = spark.sparkContext.parallelize(range(1, n + 1), partitions).map(f).reduce(add)
    print("Pi is roughly %f" % (4.0 * count / n))

    spark.stop()
```

Create and configure SparkSession

Differs slightly from slide 50: map and reduce rather than filter and count

Access SparkContext from SparkSession and operate on RDD

WordCount in Java: using SparkSession

```java
package org.apache.spark.examples;

import scala.Tuple2;

import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.sql.SparkSession;

import java.util.Arrays;
import java.util.List;
import java.util.regex.Pattern;

public final class JavaWordCount {
  private static final Pattern SPACE = Pattern.compile(" ");

  public static void main(String[] args) throws Exception {

    if (args.length < 1) {
      System.err.println("Usage: JavaWordCount <file>");
      System.exit(1);
    }

    SparkSession spark = SparkSession
      .builder()
      .appName("JavaWordCount")
      .getOrCreate();
```

Full example in Java using SparkSession and API RDD

Create and configure SparkSession

WordCount in Java: using SparkSession

Use Dataset

Use RDD

```java
    JavaRDD<String> lines = spark.read().textFile(args[0]).javaRDD();

    JavaRDD<String> words = lines.flatMap(s -> Arrays.asList(SPACE.split(s)).iterator());

    JavaPairRDD<String, Integer> ones = words.mapToPair(s -> new Tuple2<>(s, 1));

    JavaPairRDD<String, Integer> counts = ones.reduceByKey((i1, i2) -> i1 + i2);

    List<Tuple2<String, Integer>> output = counts.collect();
    for (Tuple2<?,?> tuple : output) {
      System.out.println(tuple._1() + ": " + tuple._2());
    }
    spark.stop();
  }
}
```

Submit applications to Spark

- Submit applications using `bin/spark-submit` script

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


Submit applications: main options

- `--class`: app entry point (e.g., org.apache.spark.examples.SparkPi)
- `--master`: master URL for cluster (e.g., spark://23.195.26.187:7077) (`local`, default)
- `--deploy-mode`: whether to deploy driver on worker nodes (`cluster`) or locally as external client (`client`, default)
- `--conf`: Spark configuration property in key=value format
- `application-jar`: path to jar including app and all dependencies. Be careful: URL must be globally visible, e.g., `hdfs://` path or a `file://` path that is present on all nodes
- For Python app: pass a .py file in place of `application-jar` and add Python .zip, .egg or .py files to the search path using `--py-files`
- `application-arguments`: arguments passed to the main method of the main class, if any

Submit applications: example

```
./bin/spark-submit --class
org.apache.spark.examples.SparkPi \
    --master local \
    --deploy-mode client \
    --num-executors 2 \
    --driver-memory 512m \
    --executor-memory 512m \
    --executor-cores 1 \
    examples/jars/spark-examples*.jar 10
```


Deploy modes and cluster managers

- Spark supports different deploy modes and cluster managers, so it can run in different configurations and environments

Mode	Spark driver	Spark executor	Cluster manager
Local	Runs on a single JVM, like a laptop or single node	Runs on the same JVM as the driver	Runs on the same host
Standalone	Can run on any node in the cluster	Each node in the cluster will launch its own executor JVM	Can be allocated arbitrarily to any host in the cluster
YARN (client)	Runs on a client, not part of the cluster	YARN's NodeManager's container	YARN's Resource Manager works with YARN's Application Master to allocate the containers on NodeManagers for executors
YARN (cluster)	Runs with the YARN Application Master	Same as YARN client mode	Same as YARN client mode
Kubernetes	Runs in a Kubernetes pod	Each worker runs within its own pod	Kubernetes Master

Caching and persistence

- By default, RDDs are recomputed each time you run an action on them
  - This can be *expensive* (in time) if you need to use the RDD more than once (e.g., iterative algorithms)
- To avoid computing an RDD more than once, ask Spark to ***persist*** (or ***cache***) data for rapid reuse
  - To persist RDD, use `persist()` or `cache()` methods on it
  - When RDD is persisted, each node stores in memory any partitions of it and reuses them in other actions on that RDD (or RDDs derived from it): future actions are much faster (100x)
- Key tool for iterative algorithms and fast interactive use
- Cache and persist also DataFrame and Dataset

Caching and persistence: storage level

- Using `persist()` you can specify the storage level for persisting an RDD
  - `cache()` is equivalent to `persist()` with default storage level (`MEMORY_ONLY`)
- Main storage levels for persist():
  - `MEMORY_ONLY`
  - `MEMORY_AND_DISK`
  - `MEMORY_ONLY_SER, MEMORY_AND_DISK_SER`
    - `MEMORY_ONLY_SER`: data is serialized as compact byte array representation and stored only in memory; to use it, it has to be deserialized at a cost
    - `MEMORY_AND_DISK_SER:` like MEMORY_AND_DISK, but data is serialized when stored in memory (data is always serialized when stored on disk)
  - `DISK_ONLY`

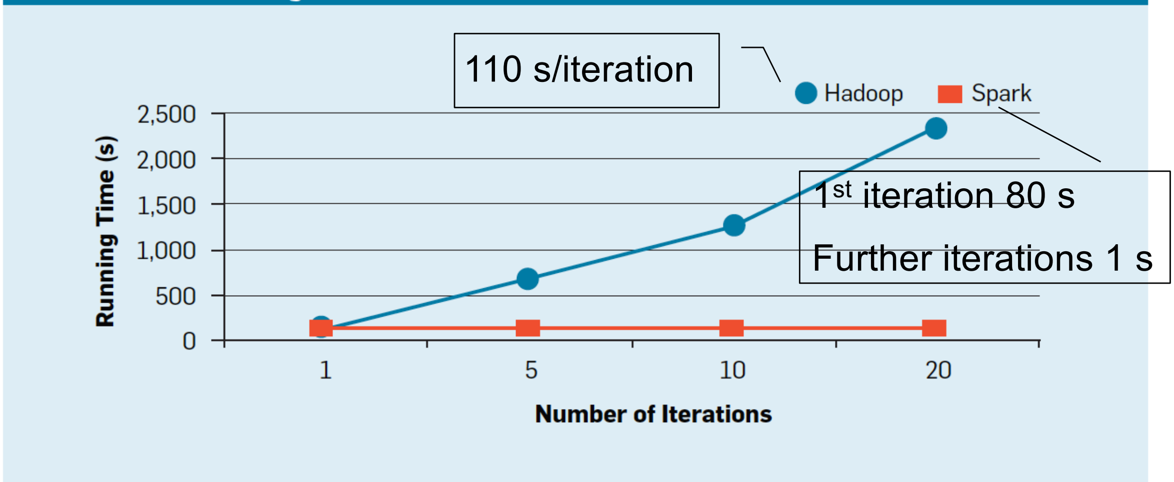
Caching and persistence: storage level

- Which storage level is best? Few things to consider:
  - Try to keep in-memory as much as possible
  - Serialization makes objects much more space-efficient
    - But select a fast serialization library (e.g., Kryo for Java) to not incur in overhead
  - Try not to spill to disk unless the functions that computed your datasets are expensive (e.g., filter a large amount of data)
  - Use replicated storage levels only if you want fast fault recovery

Caching and persistence: performance speedup

- Spark outperforms Hadoop by up to 100x in iterative ML
  - Speedup comes from avoiding I/O and deserialization costs by storing data in memory

Figure 4. Performance of logistic regression in Hadoop MapReduce vs. Spark for 100GB of data on 50 `m2.4xlarge` EC2 nodes.



Source: “Apache Spark: A Unified Engine for Big Data Processing”

Caching and persistence: example

- Let's analyze how persistence is used in iterative algorithms
- Naïve implementation of K-means algorithm
  github.com/apache/spark/blob/master/examples/src/main/python/kmeans.py
  – We need to use at each iteration the RDD containing the data points to be clustered
  – Let's cache this RDD

```
data = lines.map(parseVector).cache()
```


K-means in Spark

- Name of data file, number of clusters K and convergence threshold are read from command line

```
Usage: kmeans <file> <k> <convergeDist>
$ spark-submit --master local
  $SPARK_HOME/examples/src/main/python/kmeans.py
  $SPARK_HOME/data/mllib/kmeans_data.txt 2 0.1
```

- Code uses NumPy, the fundamental package for scientific computing with Python

K-means in Spark

- Let's first define two utility functions: `parseVector` and `closestPoint`

```
def parseVector(line):
    return np.array([float(x) for x in line.split(' ')])
```

Return the index of the nearest centroid for point p. `centers` contains sets of centroids, where `centers[i]` is the i-th set of centroids

```
def closestPoint(p, centers):
    bestIndex = 0
    closest = float("+inf")
    for i in range(len(centers)):
        tempDist = np.sum((p - centers[i]) ** 2)
        if tempDist < closest:
            closest = tempDist
            bestIndex = i
    return bestIndex
```


K-means in Spark

- Read data to be clustered from file, convert data into float numbers and then set `K` and `convergeDist`
- Cache the RDD data to improve performance
- Inizialize randomly the cluster centroids `kPoints`

```
lines = spark.read.text(sys.argv[1]).rdd.map(lambda r: r[0])
data = lines.map(parseVector).cache()
K = int(sys.argv[2])
convergeDist = float(sys.argv[3])

kPoints = data.takeSample(False, K, 1)
tempDist = 1.0
```

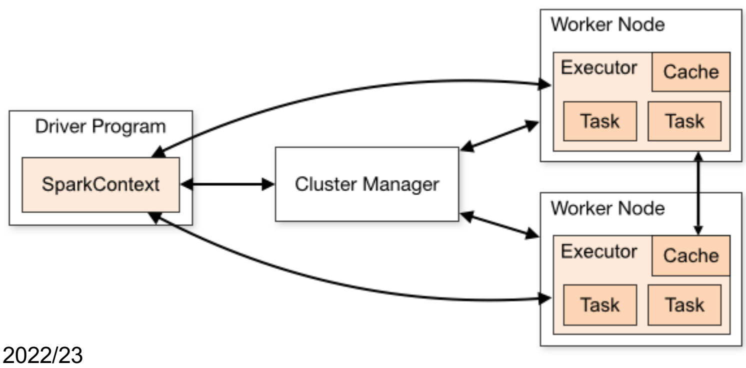
K-means in Spark

- Repeat in a loop until convergence
  - Map each data point to its closest centroid
  - Calculate new cluster centroids

```python
while tempDist > convergeDist:
    closest = data.map(
        lambda p: (closestPoint(p, kPoints), (p, 1)))
    pointStats = closest.reduceByKey(
        lambda p1_c1, p2_c2: (p1_c1[0] + p2_c2[0], p1_c1[1] + p2_c2[1]))
    newPoints = pointStats.map(
        lambda st: (st[0], st[1][0] / st[1][1])).collect()

    tempDist = sum(np.sum((kPoints[iK] - p) ** 2) for (iK, p) in newPoints)

    for (iK, p) in newPoints:
        kPoints[iK] = p

print("Final centers: " + str(kPoints))
```

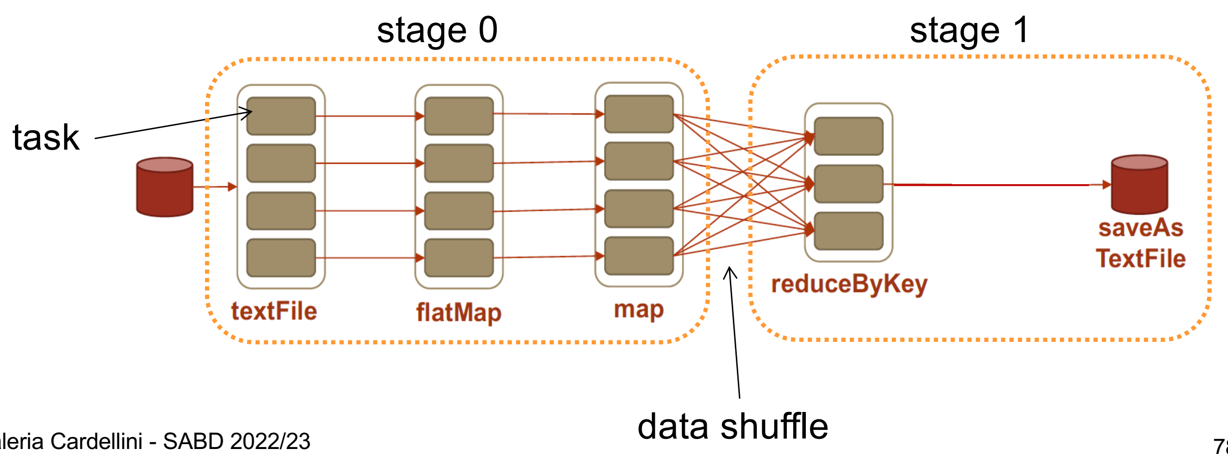

How Spark works on clusters

- A Spark application runs as a set of processes (executors) on the cluster, coordinated by SparkContext object in main() function (called *driver program*) of the application
- Executor: process launched for an application on a worker node, that runs tasks and keeps data in memory or disk storage
  - Each application has its own executors



How Spark works at runtime

- Application creates RDDs, transforms them, and runs actions: this results in a DAG of operations
- DAG is compiled into stages
  - Stage: set of tasks without a shuffle in between
  - Each task is a unit of execution that is sent to one executor and works on a single partition of data
- Actions drive the execution



Stage execution

- Spark:
  - Creates a task for each partition in RDD
  - Schedules and assigns tasks to worker nodes
- All this happens internally (you need to do anything)

Summary of Spark components

Coarse grain

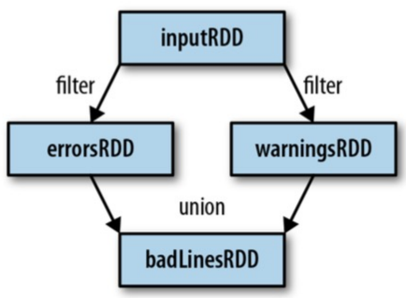
- RDD: parallel dataset with partitions
- DAG: logical graph of RDD operations
- Stage: set of tasks that run in parallel
- Task: fundamental unit of execution in Spark

Fine grain

Fault tolerance

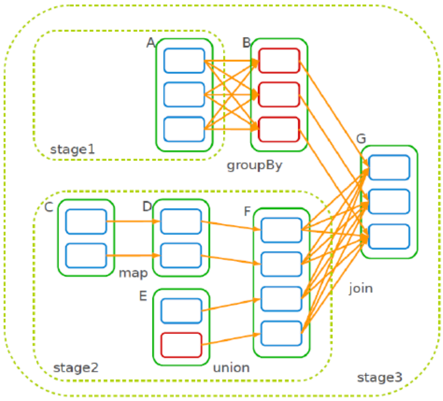
- Spark keeps track of the transformations used to build RDDs (their lineage DAG)
- Lineage information *plus* RDD immutability provide fault tolerance
  – Lineage is used to recover lost data of a RDD by replaying transformations on RDDs

Example: RDD lineage DAG created during log analysis



Application scheduling

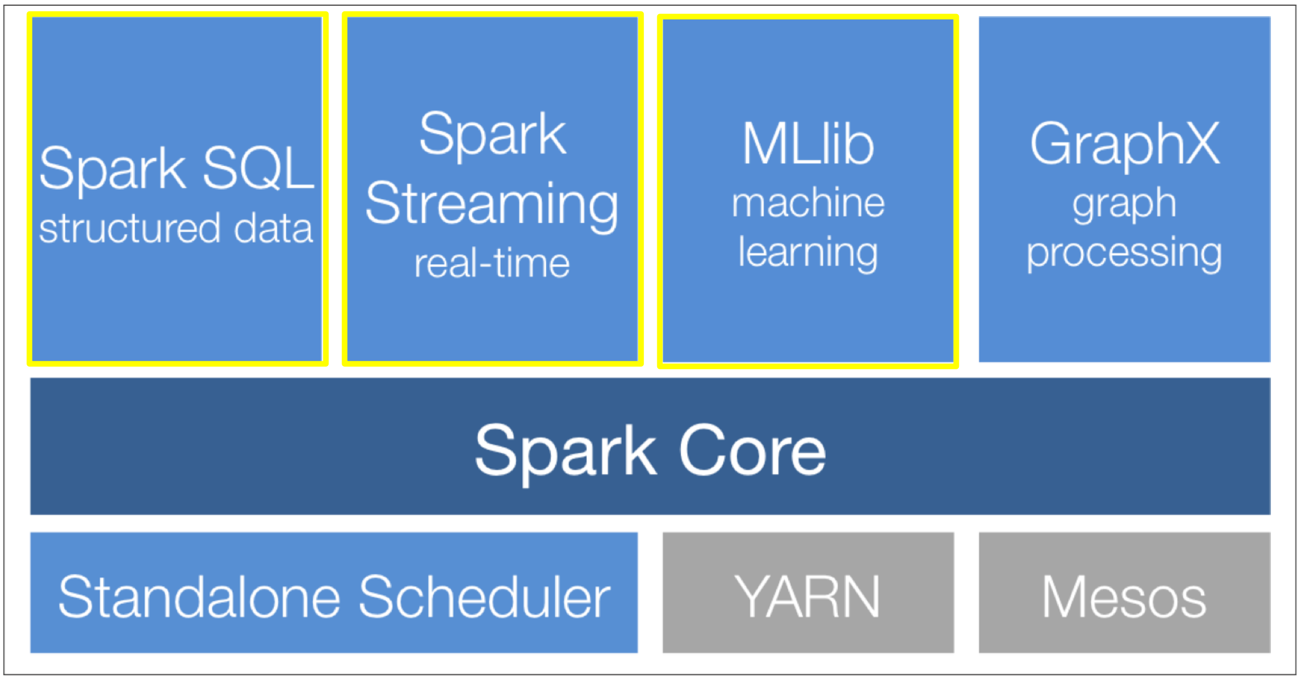
- DAG scheduler takes tasks from Spark app and sends them out to executors to get processed
- When app runs a Spark action (e.g., `collect`), scheduler builds a DAG of stages from the RDD lineage DAG
  - A stage contains pipelined transformations with narrow dependencies
  - Stage boundary:
    - Shuffles for wide dependencies
    - Already computed partitions



Application scheduling

- The scheduler launches tasks to compute missing partitions from each stage until it computes the target RDD
- Tasks are assigned to worker nodes based on data locality
  - If a task needs a partition which is available in a node's memory, the task is sent to that node

Spark stack



Spark SQL

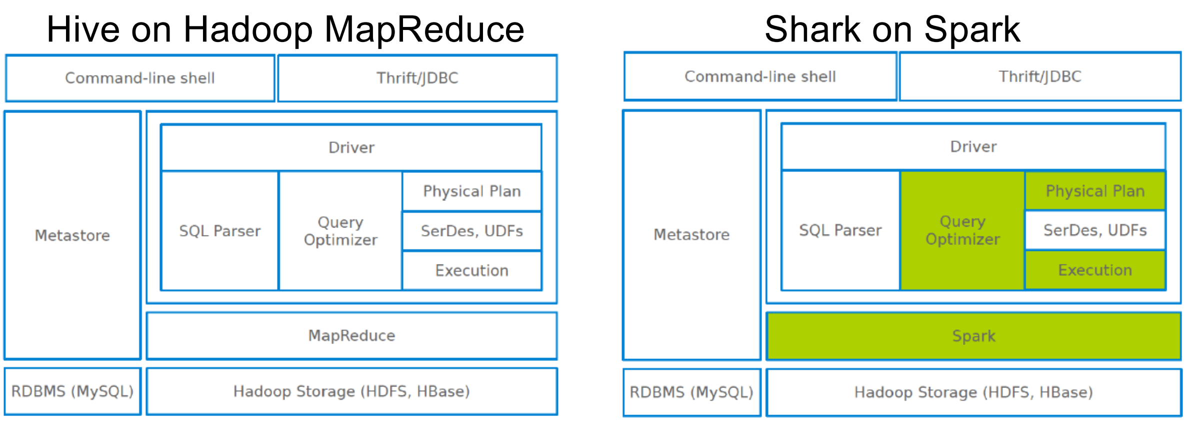
- Spark module for structured data processing
- Run SQL queries on top of Spark
- Integrated with Spark ecosystem
  - Seamlessly mix SQL queries with Spark programs, using either SQL or DataFrame API
  - Apply functions to results of SQL queries, e.g.,

```
results = spark.sql(
  "SELECT * FROM people")
names = results.map(lambda
p: p.name)
```

- Compatible with Hive, speedup up to 100x
  - Hive: data warehouse built on top of Hadoop that provides data summarization, query, and analysis with SQL-like interface

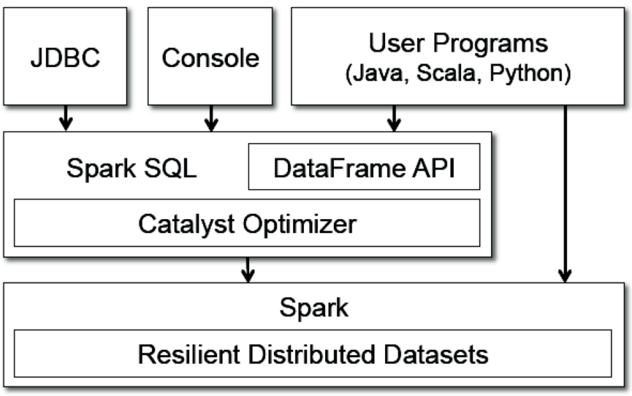
Spark SQL: the beginning

- How to extend Hive to run on Spark? Shark
  - Shark modified Hive’s backend to run over Spark, employing in-memory columnar storage
  - Limitations
    - Only Hive data model
    - Query optimizer tied to Hadoop



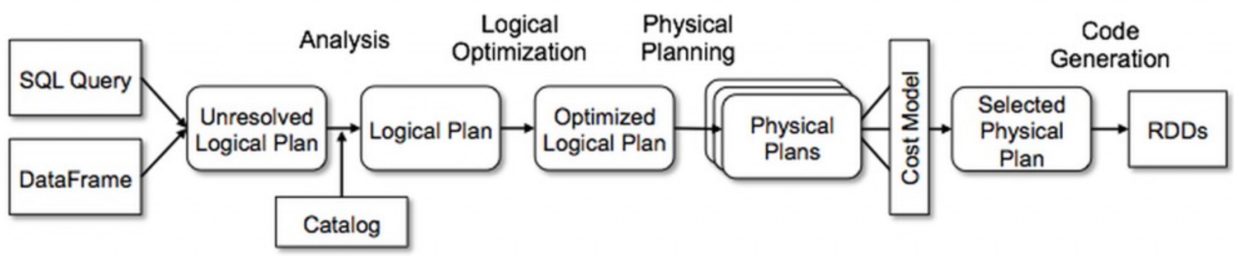
Spark SQL: Features

- Borrows from Shark
  - Hive data loading, in-memory columnar storage
- Adds:
  - RDD-aware query optimizer (Catalyst Optimizer)
  - Schema to RDD (DataFrame and Dataset APIs)
  - Rich language interfaces



Spark SQL: Catalyst optimizer

- Catalyst is based on functional programming constructs in Scala and designed for
  - Easily adding new optimization techniques and features to Spark SQL
  - Enabling developers to extend the optimizer (e.g., adding data source specific rules, support for new data types)
- Phases of query execution: analysis, logical optimization, physical planning, and code generation



DataFrame and Dataset APIs

- **Higher-level APIs** than RDD API
- DataFrames and Datasets have in common with RDDs:
  - Distributed in-memory collection of data
  - Immutable
  - Can be manipulated in similar ways to RDDs
  - Are evaluated lazily
  - Can be persisted in memory
  - Spark keeps a lineage of transformations

DataFrame and Dataset APIs

- **DataFrame** (from Spark 1.3) adds to RDD a schema to describe data
  - Unlike RDD, data is organized into a distributed in-memory table with named columns and schema
  - Works only on structured and semi-structured data
  - Since Spark 2.0 DataFrame is implemented as special case of Dataset
  - Spark SQL provides APIs to run SQL queries on DataFrame with a simple SQL-like syntax
- Table-like format of a DataFrame

```
+------------------+--------+--------------------+--------------------+-----------+------+
|              Name| Surname|             Address|                City|      State|   ZIP|
+------------------+--------+--------------------+--------------------+-----------+------+
|              John|     Doe|   120 jefferson st.|           Riverside|         NJ| 08075|
|              Jack|McGinnis|        220 hobo Av.|               Phila|         PA| 09119|
| "John ""Da Man"""|  Repici|   120 Jefferson St.|           Riverside|         NJ| 08075|
|           Stephen|   Tyler|"7452 Terrace ""A...|            SomeTown|         SD| 91234|
|              null|Blankman|                null|            SomeTown|         SD| 00298|
|"Joan ""the bone""|   Anne"|                 Jet|9th, at Terrace plc|Desert City|    CO|
+------------------+--------+--------------------+--------------------+-----------+------+
```


DataFrame and Dataset APIs

- **Dataset** (from Spark 1.6) extends DataFrame providing type-safe, OO programming interface
  - Structured but typed collection of data
  - Dataset is a collection of strongly-typed JVM objects in Scala or a class in Java
- In Scala DataFrame can be seen as a collection of generic objects, `Dataset[Row]`, where `Row` is a generic untyped JVM object
- Spark 2.0 unified DataFrame and Dataset APIs as Structured APIs with similar interfaces so that developers only have to learn a single set of APIs
- SparkSession: entry point for both APIs

RDDs vs DataFrames vs Datasets

	RDDs	Dataframes	Datasets
Data Representation	RDD is a distributed collection of data elements without any schema.	It is also the distributed collection organized into the named columns	It is an extension of Dataframes with more features like type-safety and object-oriented interface.
Optimization	No in-built optimization engine for RDDs. Developers need to write the optimized code themselves.	It uses a catalyst optimizer for optimization.	It also uses a catalyst optimizer for optimization purposes.
Projection of Schema	Here, we need to define the schema manually.	It will automatically find out the schema of the dataset.	It will also automatically find out the schema of the dataset by using the SQL Engine.
Aggregation Operation	RDD is slower than both Dataframes and Datasets to perform simple operations like grouping the data.	It provides an easy API to perform aggregation operations. It performs aggregation faster than both RDDs and Datasets.	Dataset is faster than RDDs but a bit slower than Dataframes.

Dataset API

- Provides the benefits of RDDs (strong typing, ability to use lambda functions) with those of Spark SQL's optimized execution engine
- Available only in Scala and Java
- Can be constructed from JVM objects
- Can be manipulated using transformations (`map`, `flatMap`, `filter`, `groupBy`, ...) and actions
- Lazy, i.e. computation is only triggered when an action is invoked
  – Internally, a logical plan describes the computation required to produce data. When an action is invoked, Spark query optimizer optimizes the logical plan and generates a physical plan for efficient execution in a parallel and distributed manner

Dataset API

- How to create a Dataset?
  – From a file using `read` function
  – From an existing RDD by converting it
  – Through transformations applied on existing Datasets

- When creating a Dataset you have to know the schema (i.e., the data types)
  - With JSON and CSV files it is possible to infer the schema

DataFrame API

- DataFrame: a *Dataset* organized into named columns
- Conceptually equivalent to a table in a relational database but with richer optimizations
  – Like Dataset, it exploits Catalyst optimizer
- Available in Scala, Java, Python, and R
  – In Scala and Java, a DataFrame is represented by a Dataset of Rows
- Can be manipulated in similar ways to RDDs
- Can be constructed from:
  – Structured data files (JSON, CSV, Parquet, Avro)
  – Existing RDDs, either inferring the schema using reflection or programmatically specifying the schema
  – Tables in Hive

DataFrame API: constructing data frames

```
>>> data_df = spark.createDataFrame([("Brooke", 20), ("Denny", 31), ("Jules", 30),
... ("TD", 35), ("Brooke", 25)], ["name", "age"])
>>> data_df.show()
+------+---+
|  name|age|
+------+---+
|Brooke| 20|
| Denny| 31|
| Jules| 30|
|    TD| 35|
|Brooke| 25|
+------+---+
```

- Can infer schema from CSV file
  - Can specify the separator ("," as default)

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

See example
github.com/apache/spark/blob/master/examples/src/main/python/sql/datasource.py

DataFrame API: loading a CSV file

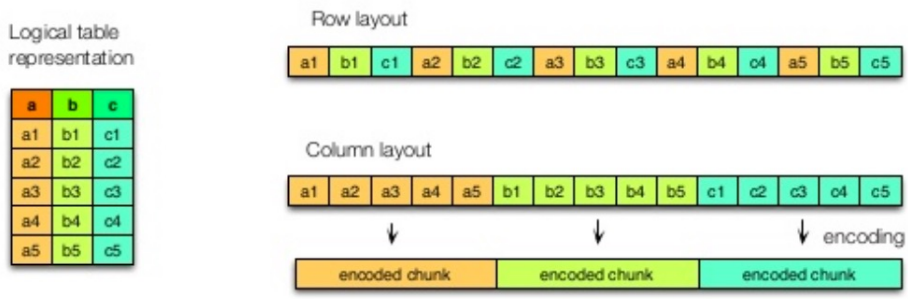
```
>>> df = spark.read.csv("address.csv", header=True)
>>> df.show()
+------------------+--------+--------------------+--------------------+-----------+------+
|              Name| Surname|             Address|                City|      State|   ZIP|
+------------------+--------+--------------------+--------------------+-----------+------+
|              John|     Doe|  120 jefferson st.|           Riverside|         NJ| 08075|
|              Jack|McGinnis|        220 hobo Av.|               Phila|         PA| 09119|
| "John ""Da Man"""|  Repici|  120 Jefferson St.|           Riverside|         NJ| 08075|
|           Stephen|   Tyler|"7452 Terrace ""A...|            SomeTown|         SD| 91234|
|              null|Blankman|                null|            SomeTown|         SD| 00298|
|"Joan ""the bone""|   Anne"|                 Jet|9th, at Terrace plc|Desert City|    CO|
+------------------+--------+--------------------+--------------------+-----------+------+
```

- Can infer schema from CSV file

```
>>> df.printSchema()
root
 |-- Name: string (nullable = true)
 |-- Surname: string (nullable = true)
 |-- Address: string (nullable = true)
 |-- City: string (nullable = true)
 |-- State: string (nullable = true)
 |-- ZIP: string (nullable = true)
```

Parquet file format

- An efficient columnar data storage format
  - Default data source in Spark
- Also supported by other data processing frameworks
  - Hive, Impala, Pig, ...
- Interoperable with other data storage formats
  - Avro, Thrift, Protocol Buffers, ...



Parquet file format

- Supports efficient compression and encoding schemes
- Example: Parquet vs. CSV

Dataset	Size on Amazon S3	Query Run time	Data Scanned	Cost
Data stored as CSV files	1 TB	236 seconds	1.15 TB	$5.75
Data stored in Apache Parquet format*	130 GB	6.78 seconds	2.51 GB	$0.01
Savings / Speedup	87% less with Parquet	34x faster	99% less data scanned	99.7% savings

- Spark SQL provides support for reading and writing Parquet files
- Schema of original data is automatically preserved

DataFrame API: using Parquet

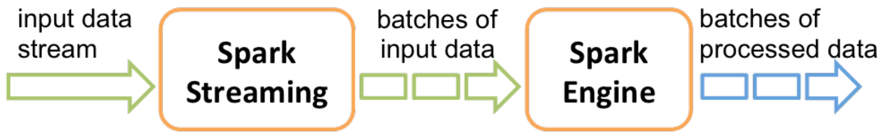
```
peopleDF = spark.read.json("examples/src/main/resources/people.json")

# DataFrames can be saved as Parquet files, maintaining the schema information.
peopleDF.write.parquet("people.parquet")

# Read in the Parquet file created above.
# Parquet files are self-describing so the schema is preserved.
# The result of loading a parquet file is also a DataFrame.
parquetFile = spark.read.parquet("people.parquet")

# Parquet files can also be used to create a temporary view and then used in SQL statements.
parquetFile.createOrReplaceTempView("parquetFile")
teenagers = spark.sql("SELECT name FROM parquetFile WHERE age >= 13 AND age <= 19")
teenagers.show()
# +------+
# |  name|
# +------+
# |Justin|
# +------+
```

Spark SQL can automatically infer the schema of a JSON dataset and load it as a `Dataset[Row]`. This conversion can be done using `SparkSession.read.json()`

See spark.apache.org/docs/latest/sql-data-sources-parquet.html

Spark Streaming

- To analyze streaming data
  - Ingested and analyzed in micro-batches
- Uses a high-level abstraction called Dstream (discretized stream) which represents a continuous stream of data
  - A sequence of RDDs
- Internally, it works as:



- We will study Spark Streaming later

Spark MLlib

- Spark MLlib is Spark library for machine learning (ML)
  - Two packages:
    - `spark.mllib` package: MLlib RDD-based API in maintanance mode
    - `spark.ml` package: MLlib DataFrame-based API to support a variety of data types
- Provides many distributed ML algorithms
  - Classification (e.g., logistic regression), regression, clustering (e.g., K-means), recommendation, decision trees, random forests, and more
- Provides also utilities
  - For ML: feature transformations, model evaluation and hyper-parameter tuning
  - For distributed linear algebra (e.g., PCA) and statistics (e.g., summary statistics, hypothesis testing)

Spark MLlib: logistic regression example

- Logistic regression: popular method to predict a categorical response
  - Binomial and multinomial
- Dataset of labels and features
- Load training data and fit model using binomial logistic regression

ML package

```
from pyspark.ml.classification import LogisticRegression

# Load training data
training = spark.read.format("libsvm").load("data/mllib/sample_libsvm_data.txt")

lr = LogisticRegression(maxIter=10, regParam=0.3, elasticNetParam=0.8)

# Fit the model
lrModel = lr.fit(training)

# Print the coefficients and intercept for logistic regression
print("Coefficients: " + str(lrModel.coefficients))
print("Intercept: " + str(lrModel.intercept))
```

LIBSVM format

Spark MLlib: K-means

- MLlib implementation includes a parallelized variant of K-means++ called kmeans||
  - K-means++ goal: inizialize *K* initial cluster centroids by spreading them out so as to improve solution quality and convergence
    - 1st cluster centroid is chosen uniformly from data points
    - Each subsequent centroid is chosen from the *remaining* data points with probability proportional to its squared distance from the point's closest existing cluster centroid
  - Con: since k-means++ is sequential, we need to use its parallel version kmeans|| to exploit Spark parallelism
- Input is feature vector
- Output is predicted cluster centers

Spark ML: k-means example

```
from pyspark.ml.clustering import KMeans
from pyspark.ml.evaluation import ClusteringEvaluator

# Loads data.
dataset = spark.read.format("libsvm").load("data/mllib/sample_kmeans_data.txt")

# Trains a k-means model.
kmeans = KMeans().setK(2).setSeed(1)
model = kmeans.fit(dataset)

# Make predictions
predictions = model.transform(dataset)

# Evaluate clustering by computing Silhouette score
evaluator = ClusteringEvaluator()

silhouette = evaluator.evaluate(predictions)
print("Silhouette with squared euclidean distance = " + str(silhouette))

# Shows the result.
centers = model.clusterCenters()
print("Cluster Centers: ")
for center in centers:
    print(center)
```

Silhouette is used to study the separation distance between resulting clusters.

Silhouette plot displays a measure of how close each point in one cluster is to points in the neighboring clusters and provides a way to assess the number of clusters visually.

Combining processing tasks with Spark

- It is easy to seamlessly combine different Spark libraries in the same application

- Example in Scala combining SQL, ML and streaming libraries in Spark
  - Read historical Twitter data using Spark SQL
  - Train a K-means clustering model using MLlib
  - Apply the model to a new stream of tweets in order to predict language from location

Combining processing tasks with Spark

```
// Load historical data as an RDD using Spark SQL
val trainingData = sql(
"SELECT location, language FROM old_tweets")

// Train a K-means model using MLlib
val model = new KMeans()
.setFeaturesCol("location")
.setPredictionCol("language")
.fit(trainingData)

// Apply the model to new tweets in a stream
TwitterUtils.createStream(...)
.map(tweet => model.predict(tweet.location))
```

References

- Zaharia et al., Spark: Cluster Computing with Working Sets, HotCloud'10.
- Zaharia et al., Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing, NSDI'12.
- Zaharia et al., Apache Spark: A Unified Engine For Big Data Processing", Commun. ACM, 2016.
- Ambrust et al., Spark SQL: Relational Data Processing in Spark, ACM SIGMOD'15.
- Damji et al., Learning Spark - Lightning-Fast Big Data Analysis, 2nd edition, O'Reilly, 2020.