TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA
Macroarea di Ingegneria
Dipartimento di Ingegneria Civile e Ingegneria Informatica

Apache Spark

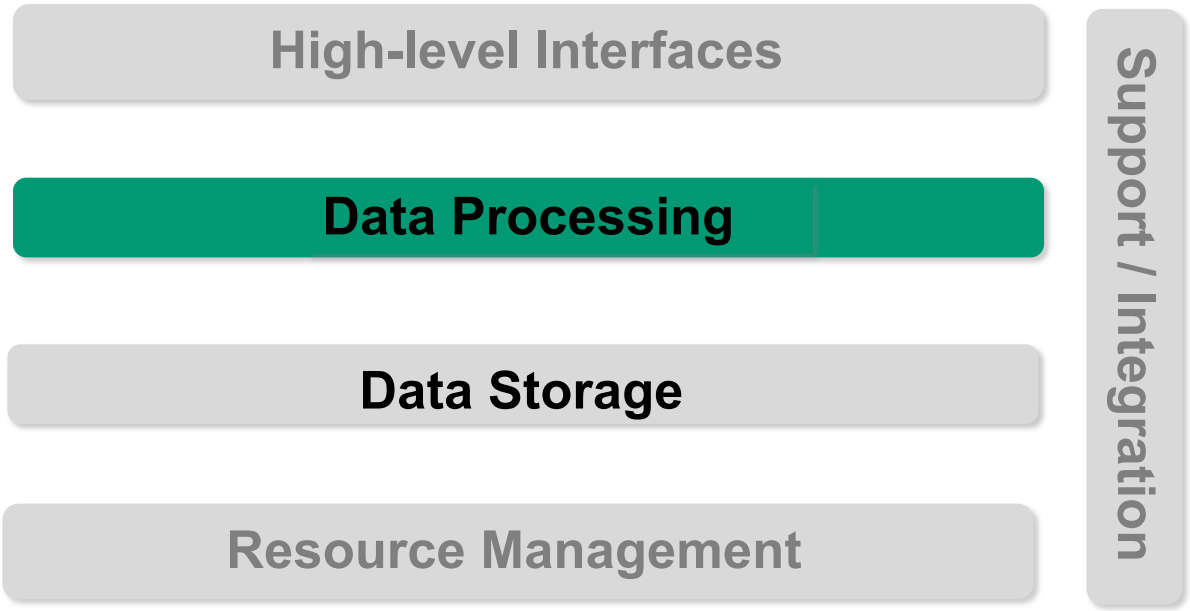
Corso di Sistemi e Architetture per Big Data

A.A. 2025/26

Valeria Cardellini

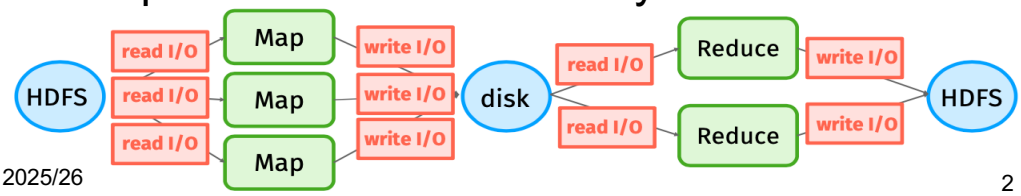
Laurea Magistrale in Ingegneria Informatica

The reference Big Data stack



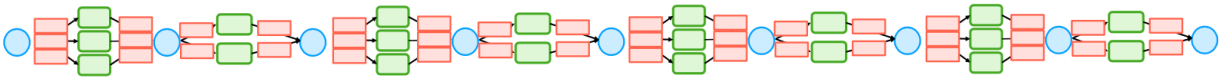
MapReduce (MR): limitations

- Programming model
  - Not all problems fit naturally into MR paradigm
  - Even simple tasks may require multiple MR jobs
    - E.g., sorting words by their frequency requires 2 jobs
  - Limited control over execution flow, data structures, and iteration
- Efficiency
  - High overhead due to data movement: compute (map) → communicate (shuffle) → compute (reduce)
  - Input and output from/to disks (HDFS and local disks)
  - Limited exploitation of main memory



MapReduce: limitations

- No native support for iterative processing
  - High I/O overhead: each iteration reads/write data from/to disk
  - Poor fit for iterative algorithms (e.g., ML)
    - Partial solution: design algorithms that minimize the number of iterations



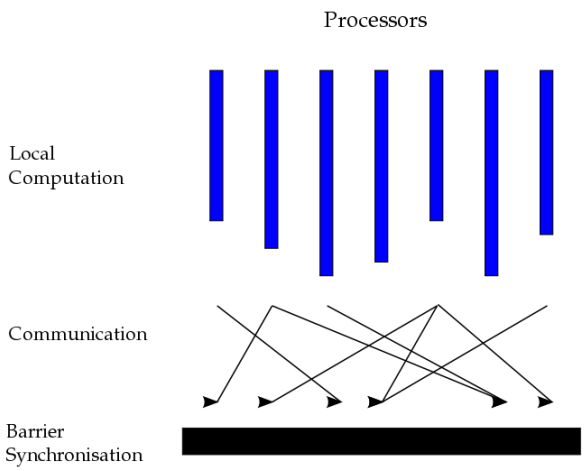
- Not suitable for real-time stream processing
  - A MR job typically requires scanning the full input dataset before producing results

Alternative programming models

- DAG-based execution frameworks
  - Application represented as directed acyclic graphs (DAGs)
    - Node: operation/task
    - Edge: dependency (data flow between operations)
  - Examples: Spark, Flink, Storm, Airflow, TensorFlow
- SQL-based data processing
  - Hive, Spark SQL, Trino, Vertica, …
- NoSQL and NewSQL data stores
  - HBase, MongoDB, Cassandra, Spanner, …
- *Bulk Synchronous Parallel* model

Alternative programming models: BSP

- Bulk Synchronous Parallel (BSP)
  - Developed by Leslie Valiant during 1980s
  - Computation proceeds in super-steps:
    - Local computation
    - Communication
    - Barrier synchronization
  - Well-suited for iterative algorithms (e.g., graph processing)
  - Examples: Google Pregel, Apache Giraph



Apache Spark

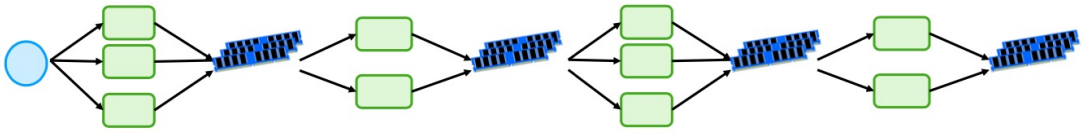
- Unified engine for large-scale data analytics
  - Leading platform for batch/stream processing, SQL analytics, data science, and machine learning on clusters of nodes
  - Multi-language support: Scala, Python, Java, R
- In-memory data processing for efficient iterative computation
  - Often 10x faster or more than Hadoop MapReduce
- Execution model based on DAGs with advanced optimization
- Compatible with the Hadoop ecosystem
  - Can read/write data from systems like HDFS and HBase

Spark milestones

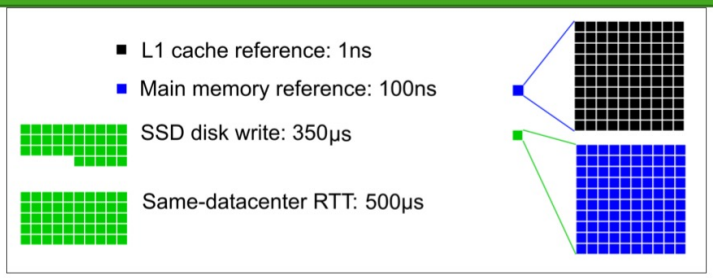
- Project started in 2009
- Originally developed at UC Berkeley AMPLab by Matei Zaharia during his PhD
- Open sourced in 2010, Apache project since 2013
- Zaharia co-founded Databricks in 2014
- One of the most widely used open-source frameworks for big data processing
- Current release (Jan. 2026): 4.1.1

Spark: In-memory computation

- Key idea: keep datasets in main memory across operations
- Share data efficiently across multiple tasks/stages
- Distributed in-memory processing: 10x-100x faster than disk and network



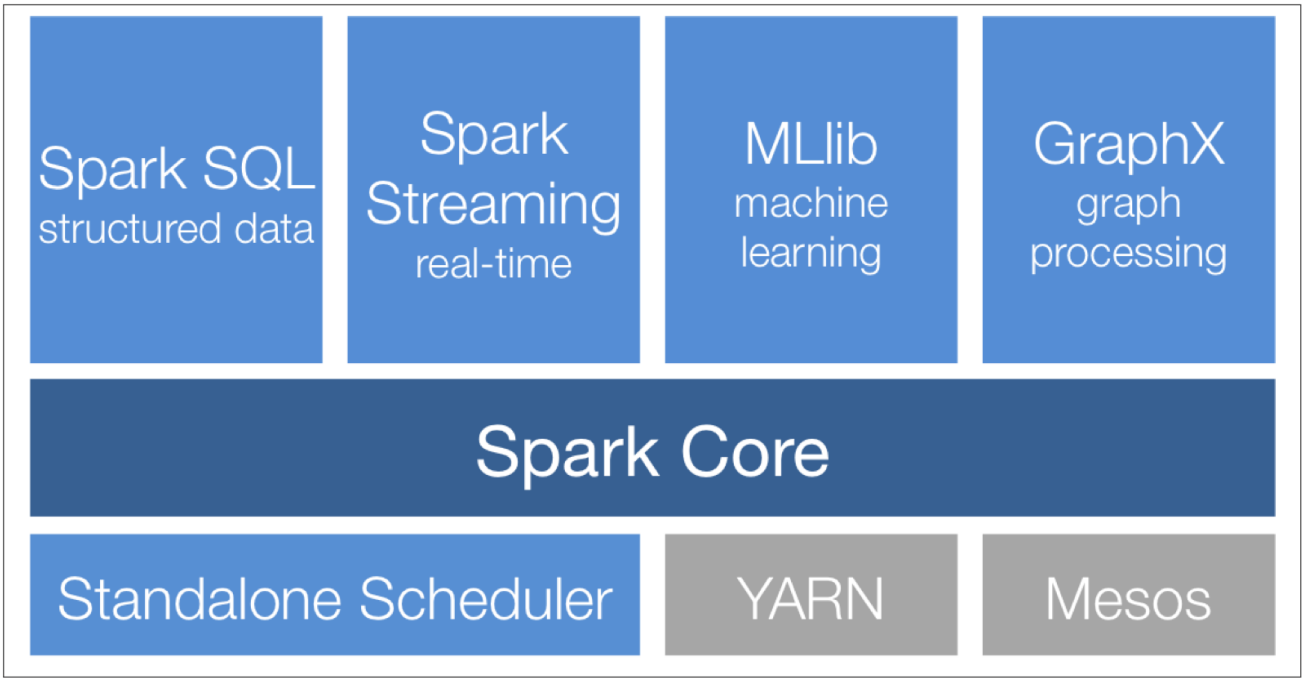
Much faster response time (in practice: 10x-100x)



Spark vs Hadoop MapReduce

- Programming paradigm conceptually similar to MapReduce
  - Scatter-gather pattern: distribute (scatter) data and computation across cluster nodes that process partitions in parallel; aggregate (gather) results
- More general and flexible data model
  - RDDs, DataSets, DataFrames
- More expressive and developer-friendly programming model
  - Transformations: define data processing steps
  - Actions: trigger execution and produce results
  - Generalizes Map and Reduce into many transformations and actions
- Storage-agnostic
  - Not only HDFS, but also Cassandra, S3, Parquet files, …

Spark stack



Spark core

- Provides basic functionalities
  - Task scheduling, memory management, fault tolerance and recovery, interaction with storage systems
- Provides the core data abstraction: **resilient distributed dataset (RDD)**
  - Collection of items distributed across cluster nodes and processed in parallel
  - APIs to create and manipulate RDDs
- Implemented in Scala, with APIs also in Java, Python, R

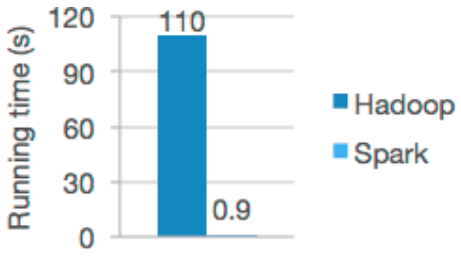
Spark as unified analytics engine

- Provides a rich set of integrated higher-level libraries built on top of Spark core
  - Can be seamlessly combined within same app
- **Spark SQL**
  - Supports SQL and structured data processing
  - Mutiple data sources (Parquet, JSON, Avro, …)
- **Structured Streaming**
  - Supports stream processing as continuous, incremental computation

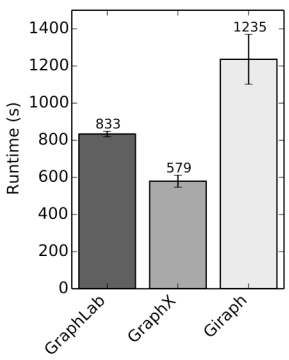
Spark as unified analytics engine

- **MLlib**
  - Scalable ML library
  - Distributed ML algorithms: feature extraction, classification, regression, clustering, recommendation, …
- **GraphX**
  - Graph-parallel computation
  - Provides graph algorithms (e.g., PageRank)
- **Pandas API on Spark**
  - Enables scaling pandas workloads to distributed environments

Logistic regression performance



PageRank performance (20 iterations, 3.7B edges)



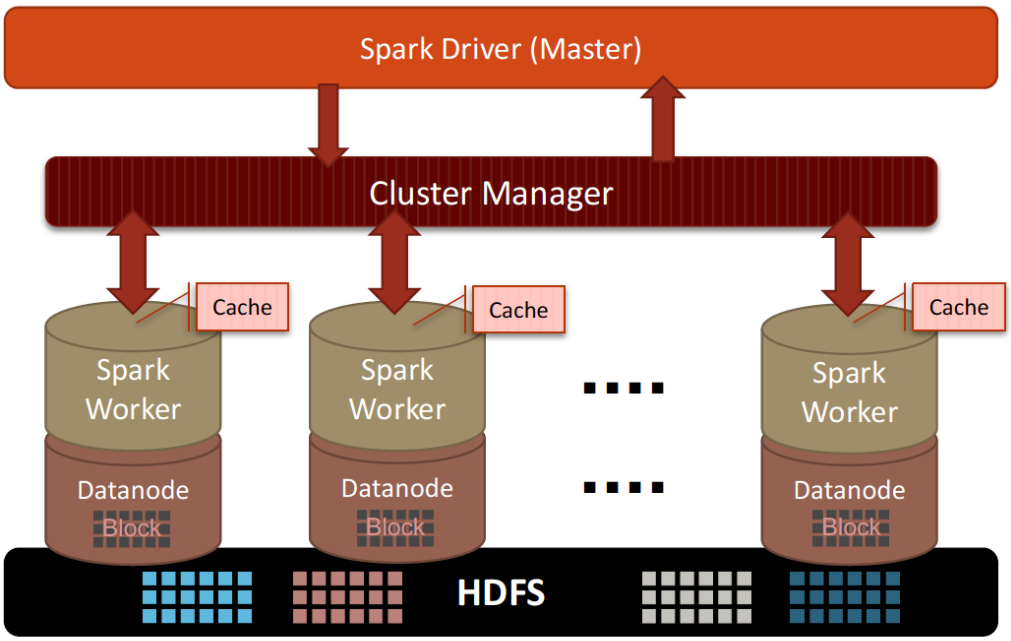
Spark and cluster managers

- Spark can run on top of multiple cluster resource managers which allocate cluster resources to run applications

1. Standalone
   - Built-in cluster manager included with Spark, simple to deploy and execute; mainly used for small clusters or testing
2. Hadoop YARN
   - Hadoop cluster manager
3. Mesos
   - General-purpose cluster manager originally developed at UC Berkeley AMPLab
4. Kubernetes

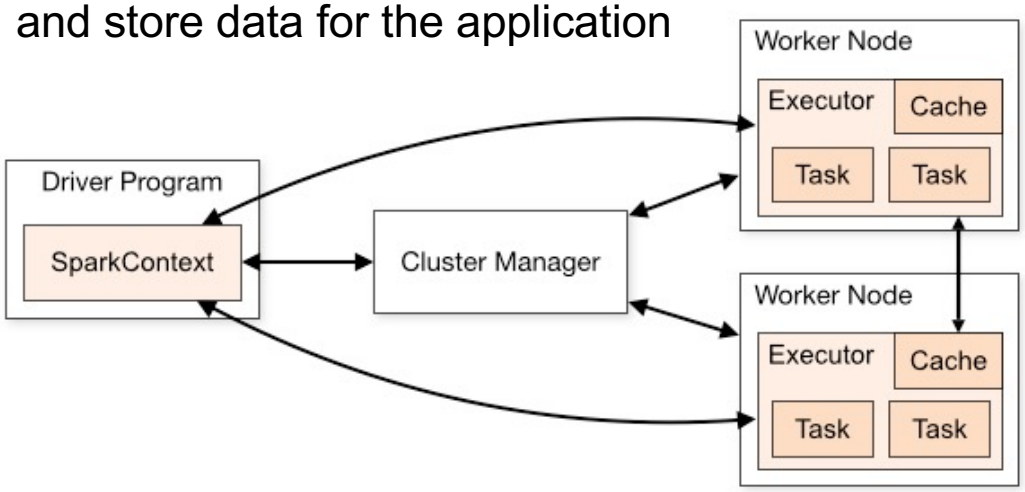
Spark architecture

- 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

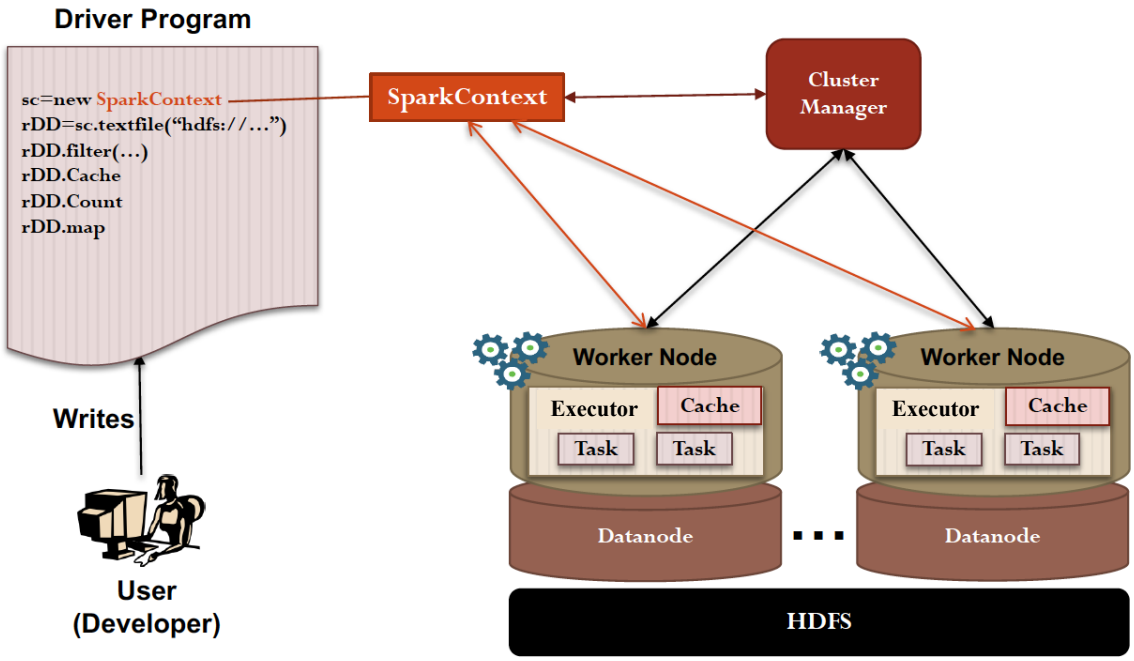


https://spark.apache.org/docs/latest/cluster-overview.html

Spark architecture

- Each application consists of a driver program and executors running on the cluster
  - Driver program: process which runs user's `main` function and creates the SparkContext
  - SparkContext: main entry point for Spark functionality, connects to the cluster and manages resource access
- Each application gets its own executors
  - Executors: processes launched on cluster nodes, run tasks in multiple threads
  - Isolation between concurrent applications
- Execution flow on a cluster:
  - SparkContext connects to cluster manager, which allocates cluster resources
  - Spark launches executors on worker nodes and sends application code (e.g., jar) to executors
  - SparkContext sends tasks to executors to run

Spark architecture



Resilient Distributed Datasets (RDDs)

- Core programming abstraction in Spark: a distributed memory abstraction
- Immutable, partitioned and fault-tolerant **collection of elements**
- Can be processed in parallel across a cluster
- Distributed memory
  - Stored in the main memory of executors on worker nodes (when possible), or spilled to local disk if memory is insufficient

RDD: distributed and partitioned

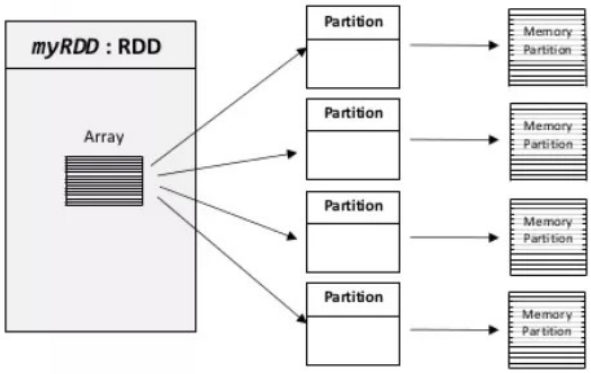
- Enable parallel execution of operations on data
  - Each executor runs the code on its assigned RDD partitions
- Partition
  - Logical division of dataset
  - Smallest unit of parallelism in Spark: each task operates on one partition
  - RDD partitions can be distributed across different nodes in the cluster

RDD: immutable and fault-tolerant

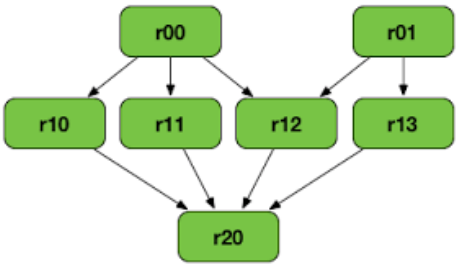
- Immutable once constructed
  - RDD content cannot be modified
  - New RDD is created from existing RDD(s)
- Automatically rebuilt on failure (without replication)
  - Track lineage information so to efficiently recompute missing or lost data due to (node) failures
  - For each RDD, Spark knows how 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 **lineage DAG**, that defines the logical execution plan and represents the dependencies between RDDs (or DataFrames)



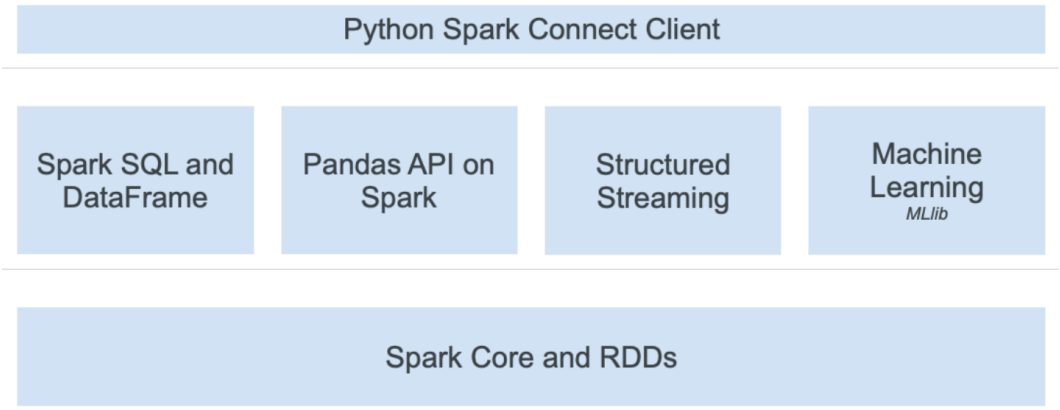
RDD API

- Clean language-integrated API for Scala, Python, Java, and R
- Supports interactive use via PySpark and Scala shell
- Why use RDDs?
  - Best suited for unstructured data or low-level data processing
  - Provides fine-grained control over data distribution and execution
- Other higher-level APIs
  - DataFrame: optimized, structured data
  - DataSet: type-safe API (mainly in Scala/Java)

PySpark

- Python API for Spark supporting the integration between Spark and Python
- Provides PySpark shell for interactive analysis
- Full support for core Spark capabilities: RDD, Spark SQL, DataFrames, Structured Streaming, MLlib

https://spark.apache.org/docs/latest/api/python/



PySpark: SparkContext

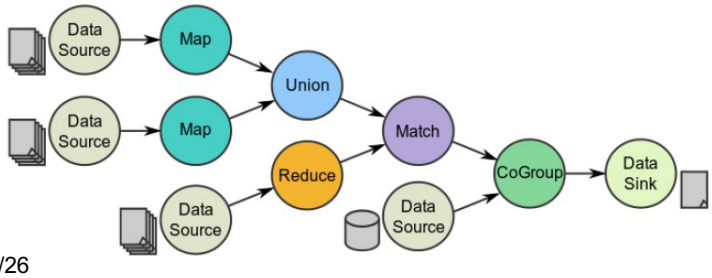
- SparkContext: entry point for low-level RDD API, connection to Spark cluster
- To create a SparkContext, you first need to build a SparkConf object that contains information about application

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

- SparkConf allows to set various Spark parameters, among which
  - `master`: URL of cluster to connect to
  - `appName`: name of job to run
- In the shell, SparkContext is already available as `sc`

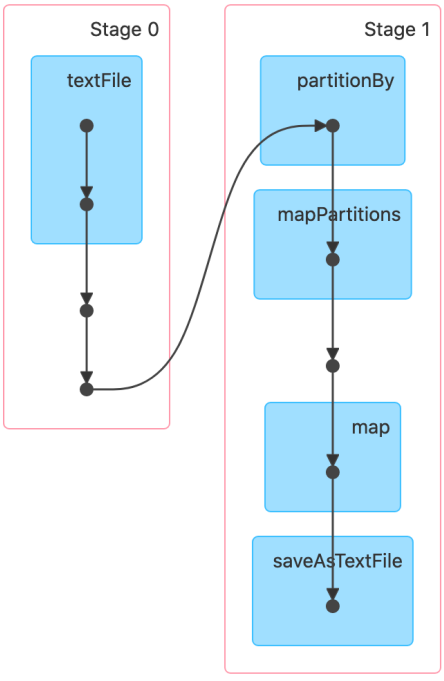
Spark programming model: DAG

- 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 nodes and links, where nodes represent the operations on RDDs and directed links represent the data dependencies between operations
  - Acyclic graph: no cycles or loops in the graph
  - Generalization of MapReduce model, which has only two operations (Map and Reduce)



Spark programming model: DAG

- DAG can be visualized using Spark Web UI
  - In figure: WordCount DAG
- DAG is divided into stages
- Stage: set of operations that do not involve a shuffle of data, resulting in a more efficient computation
- As soon as a shuffle of data is needed (i.e., when a *wide transformation* is performed), the DAG will yield a new stage



RDD API: Operations

- 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

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

RDD API: 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

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

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

Transformations	*map*(*f* : T ⇒ U)	:	RDD[T] ⇒ RDD[U]
	filter(*f* : T ⇒ Bool)	:	RDD[T] ⇒ RDD[T]
	flatMap(*f* : T ⇒ Seq[U])	:	RDD[T] ⇒ RDD[U]
	sample(*fraction* : Float)	:	RDD[T] ⇒ RDD[T] (Deterministic sampling)
	groupByKey()	:	RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
	reduceByKey(*f* : (V, V) ⇒ 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* : V ⇒ W)	:	RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)
	sort(*c* : Comparator[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	partitionBy(*p* : Partitioner[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
Actions	*count*()	:	RDD[T] ⇒ Long
	collect()	:	RDD[T] ⇒ Seq[T]
	reduce(*f* : (T, T) ⇒ T)	:	RDD[T] ⇒ T
	lookup(*k* : K)	:	RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs)
	save(*path* : 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
  - 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)

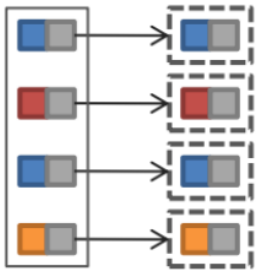
```
lines = sc.textFile("/path/input.txt")
```

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`**: takes as input a function which is applied as filter to each element of the RDD, selecting only those elements on which the function returns true

```
# 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: union and mapPartitions

- **union**: returns a new RDD that contains the union of the elements in the source RDD and the argument

```
rdd1 = sc.parallelize([2, 4, 7, 9])
rdd2 = sc.parallelize([1, 4, 5, 8, 9])
rdd3 = rdd1.union(rdd2)
# [2, 4, 7, 9, 1, 4, 5, 8, 9]
```

- **mapPartitions**: similar to `map`, but runs separately on each partition

```
rdd1 = sc.parallelize([1, 2, 3, 4, 5, 6, 7, 8, 9, 10], 3)
# Define a function to process each partition
def f(iterator): yield sum(iterator)
rdd2 = mapPartitions(f).collect()  # [6, 15, 34]
```

RDD transformations: partitionBy

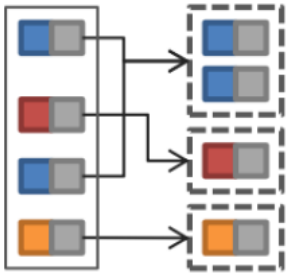
- **partitionBy**: returns a new RDD that contains the RDD partitioned using the specified partitioner and number of partitions

```
rdd = sc.parallelize([("apple", 1), ("banana", 2),
... ("orange", 3), ("apple", 4), ("banana", 1)])
partitioned_rdd = rdd.partitionBy(3)
# Use glom to see data in each partition
partitioned_rdd.glom().collect()
# [[('orange', 3)], [], [('apple', 1), ('banana', 2),
('apple', 4), ('banana', 1)]]
```

- **glom**: returns a new RDD by coalescing all elements within each partition into a list
  - useful for inspecting how data is distributed across partitions or for debugging purposes

RDD transformations: reduceByKey

- **reduceByKey**: when called on a RDD of key-value pairs, aggregates values with the same key using the specified function
- Runs parallel reduce operations, one for each key in the RDD



```
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)]
```

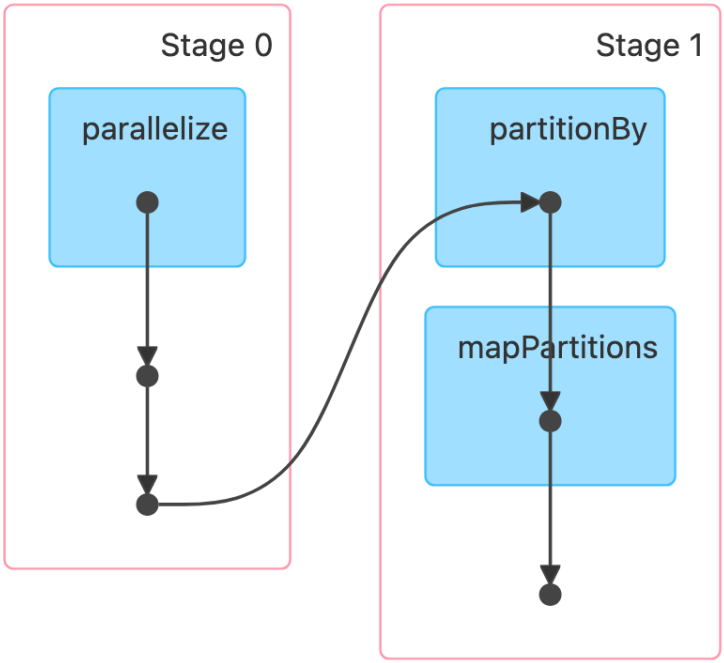
RDD transformations: reduceByKey

- Corresponding DAG (Spark Web UI)

Two stages: wide transformation (data shuffling)

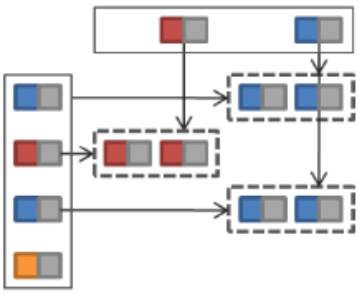
How Spark implements `reduceByKey`

- `partitionBy` partitions data by key
- `mapPartitions` operates after the shuffle and performs local (partition-level) aggregation



RDD transformations: join

- **`join`**: performs an inner-join on the keys of two RDDs
- Only keys present in both RDDs are included in the output
- Join matching is performed independently for each key after shuffle



```
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'))]
```

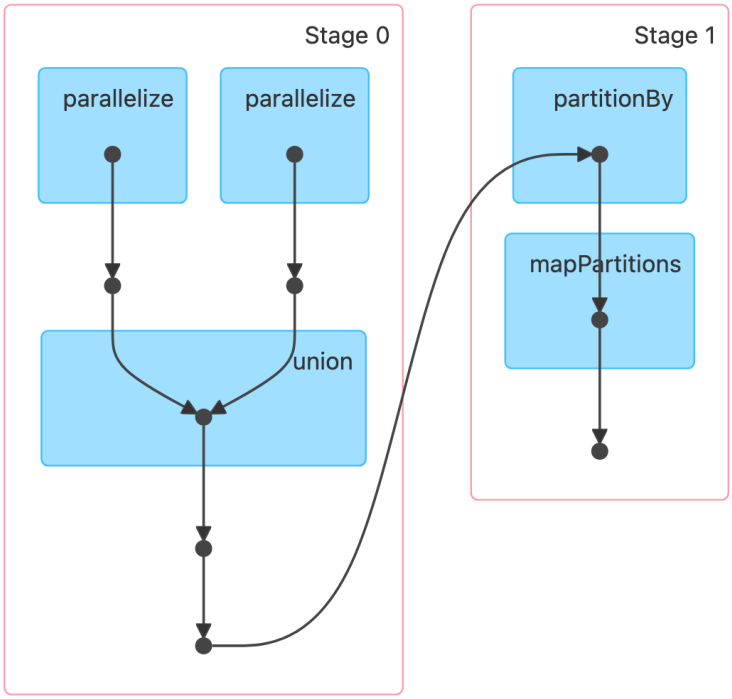
RDD transformations: join

- Corresponding DAG (Spark Web UI)

Two stages: wide transformation (data shuffling)

How Spark implements `join`

- After Stage 0, data is shuffled and partitioned by key
- `partitionBy` defines how keys are assigned to partitions
- `mapPartitions` operates on each partition after shuffle



Additional RDD transformations

- **`distinct`**: returns a new RDD that contains only distinct elements of the source RDD
- **`groupByKey`**: called on key-value pair RDD, groups all values for each key into a single sequence
- **`mapValues`**: applies a function to each value in a key-value pair RDD, keeping keys unchanged
- **`sample`**: returns a sampled fraction of the data (with or without replacement)
- **`repartition`**: changes the number of partitions (increases or decreases parallelism)
- **`coalesce`**: reduces the number of partitions

How to pass functions to transformations

- Lambda expressions
  - Used for simple functions written as a single expression
  - Do not support multi-statement logic or statements without a return value
- Local defs
  - Useful for more complex logic
  - Defined inside the function that interacts with Spark
- Top-level functions
  - Defined outside of other functions, at module scope
  - Preferred for reuse in Spark jobs

Transformations and actions

- Transformations are **lazy**
  - Are not executed immediately
  - Are computed only when an action requires a result to be returned to the driver
  - Spark builds a logical transformation plan (DAG)
  - This allows Spark to optimize execution by grouping operations
- Optimizations enabled by laziness
  - Multiple `map` / `filter` operations can be fused into a single stage
  - If data is already partitioned, Spark can avoid unnecessary shuffling (e.g., for `groupBy`)
- Computation is triggered only when an action is called
  - Actions instruct Spark to execute the transformation pipeline and return a result

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
```


RDD actions

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

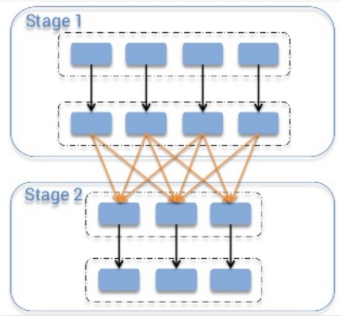
```
sum = nums.reduce(lambda x, y: x + y)
```

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

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

Shuffle in Spark

- Spark mechanism for re-distributing data across partitions
  - Data is regrouped so that related records (e.g., same key) are co-located
  - Involves data transfer across executors and machines, making it expensive
  - Example: in `reduceByKey` values for the same key may be on different partitions or machines
    - A shuffle (all-to-all communication) is required to bring them together for aggregation



- Operations that trigger a shuffle
  - Repartitioning: `repartition`, `coalesce`
  - ByKey operations: `reduceByKey`, `groupByKey`
  - Joins: `join`, `cogroup`

RDD API: First examples

- Running the examples with PySpark
  - After installing Spark (e.g., Docker official image https://hub.docker.com/_/spark), you can use PySpark from terminal to run examples interactively
  - `sc` is SparkContext variable

```
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 4.1.1
      /_/

Using Python version 3.10.12 (main, Mar  3 2026 11:56:32)
Spark context Web UI available at http://aa3eb8ba6cd3:4040
Spark context available as 'sc' (master = local[*], app id = local-1777283389575
).
SparkSession available as 'spark'.
>>> lines = sc.textFile("/sabd-data/input.txt")
>>> lines.collect()
['hello world goodbye', 'hello valeria', 'hello students', 'hello SABD', 'goodby
e mapreduce']
>>>
```

RDD API: Simple examples

- Let's analyze a few simple examples using the RDD API
  - Pi estimation
  - WordCount
  - Computing average

- Additional examples: see Spark distribution
  - Java
    https://github.com/apache/spark/tree/master/examples/src/main/java/org/apache/spark/examples
  - Python
    https://github.com/apache/spark/tree/master/examples/src/main/python

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))
```

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))
```


Pi estimation in Scala

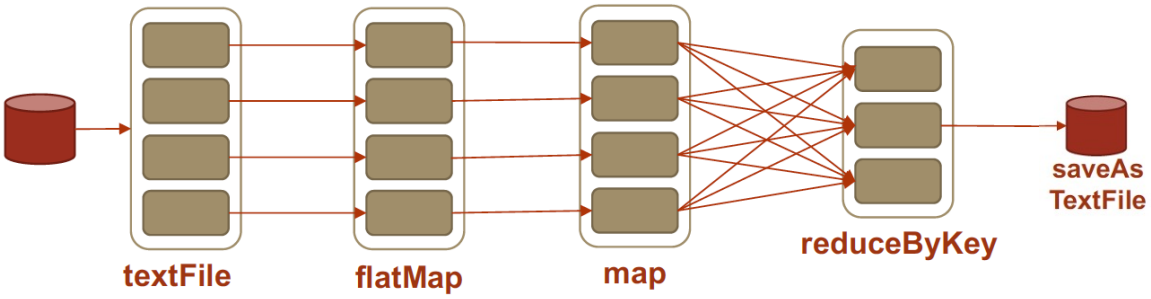
- Running the Spark shell in Scala

```
$ spark-shell
```

```
var NUM_SAMPLES = 100000
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}")
```

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")
```



WordCount in Python

- Alternative: use `countByValue`
  - Action that returns the count of each unique value in an
  - Output: dictionary of (value, count) pairs
  - The driver collects all partitions and performs the merge

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

- Which solution is better? Depends on dataset size
  - Large dataset: prefer `map` + `reduceByKey` + `collect` to exploit parallelism of `reduceByKey` during aggregation
  - Small dataset: `countByValue` can be more efficient, reduces network traffic (one stage less)

Computing average in Python

- Common pattern in data analysis
- Aggregate all ages for each name, group data by name, compute the average age per group

```
# Create an RDD of tuples (name, age)
dataRDD = sc.parallelize([("Brooke", 20), ("Denny", 31),
   ("Bob", 40), ("Bob", 35), ("Brooke", 25)])
# Use map and reduceByKey transformations with their lambda
# expressions to aggregate and then compute average
agesRDD = (dataRDD
.map(lambda x: (x[0], (x[1], 1)))
.reduceByKey(lambda x, y: (x[0] + y[0], x[1] + y[1]))
.map(lambda x: (x[0], x[1][0]/x[1][1])))
```


Java: Lambda expressions

- 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 expression in two parts
  - Left side: parameters required by lambda expression
  - Right side: actions of lambda expression

Pi estimation in Java with chaining

```
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);
```

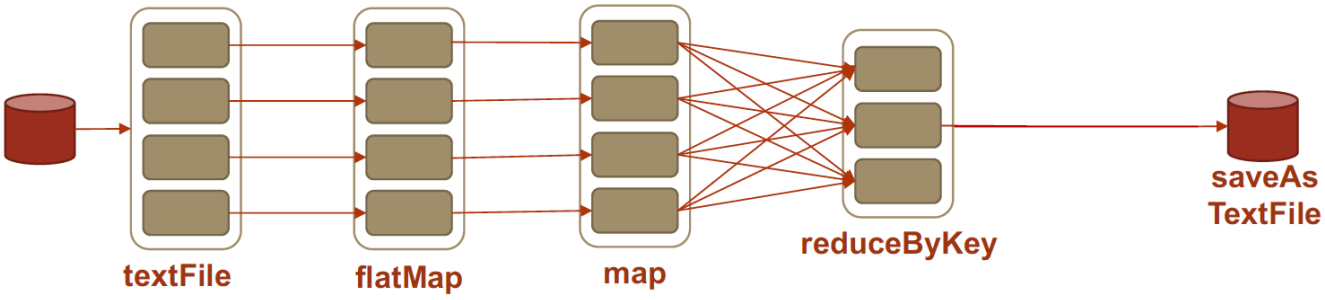

WordCount in Java

- `JavaPairRDD`: an RDD that stores key-value pairs
- In Spark Java API, pairs are created 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");
```

WordCount in Java 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 job using RDD API: create a Spark configuration object `SparkConf` and use it to inizialize a `SparkContext` object
- `SparkConf`: holds configuration for Spark application
  - Parameters are set as key-value pairs

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

- SparkContext: entry point to Spark core functionalities, used to connect to Spark cluster, load data, and apply transformations
- Only one SparkContext per JVM can be active
  - Stop the existing SparkContext before creating a new one using `stop()`

SparkSession

- `SparkSession` (since Spark 2.0): unified entry point for all Spark functionalities
  - Replaces multiple contexts used in earlier APIs
  - Still provides access to `SparkContext`, required for working directly with RDDs
- Within interactive shell: available as `spark` variable
- Within application: use `builder` API to create and configure it

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

```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()
```

Full example using `SparkSession` and RDD API

Create and configure `SparkSession`

Variant of slide 49: use `map` and `reduce` instead of `filter` and `count`

Obtain `SparkContext` from `SparkSession` and work with RDDs

https://github.com/apache/spark/blob/master/examples/src/main/python/pi.py

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 using `SparkSession` and RDD API

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();
  }
}
```

https://github.com/apache/spark/blob/master/examples/src/main/java/org/apache/spark/examples/JavaWordCount.java

Launch applications

- To launch Spark application, use `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]
```

https://spark.apache.org/docs/latest/submitting-applications.html

`spark-submit` options

- `--class`: application entry point (e.g., `org.apache.spark.examples.SparkPi`)
- `--master`: cluster master URL (e.g., spark://23.195.26.187:7077) (default: `local`)
- `--deploy-mode`: where to run the driver; locally as client (default: `client`) or on worker nodes (`cluster`)
- `--conf`: Spark configuration property in key=value format
- `application-jar`: path to jar with application and dependencies; must be globally accessible, e.g., `hdfs://` path or `file://` path available on all nodes
- Python applications: pass a .py file instead of `application-jar` and use `--py-files` to include dependencies (.zip, .egg, .py)
- `application-arguments`: passed to the main method of the application

Launch applications: examples

- PageRank in Python passing arguments

```
./bin/spark-submit \
examples/src/main/python/pagerank.py \
data/mllib/pagerank_data.txt 10
```

- Pi estimation in Java configuring Spark and passing arguments

```
./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

- Deployment flexibility
  - Spark supports different deploy modes and cluster managers
  - 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

RDD persistence (caching)

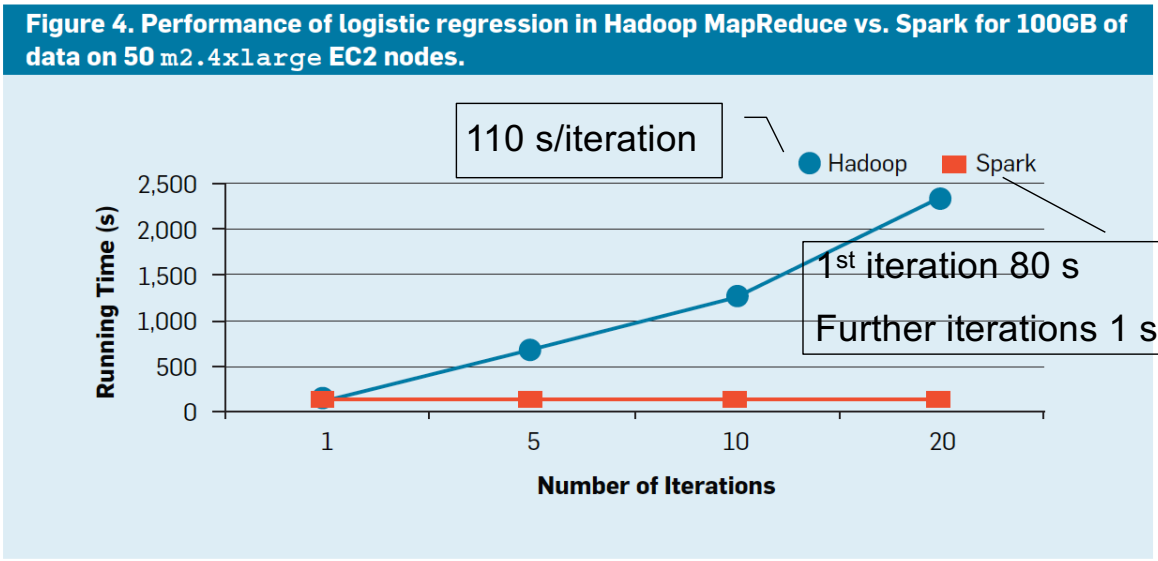
- By default, RDDs are recomputed every time an action is executed on them
  - This can be expensive, especially if the RDD is used more than once (e.g., iterative algorithms)
- To avoid recomputation, use persist (or cache)
  - Methods: `persist()` or `cache()`
- When an RDD is persisted:
  - Partitions are stored (typically in memory) on each node
  - Reused in subsequent actions
  - Leads to significant speedups (often >10x)
- Key tool for iterative algorithms and interactive data analysis
- Also available for DataFrames and Datasets APIs

RDD persistence: storage levels

- With `persist()` you can specify the storage level
  - `cache()` = `persist()` with default storage level (`MEMORY_ONLY`)
- Main storage levels:
  - `MEMORY_ONLY`: stores RDD in memory; if it does not fit, missing partitions are recomputed on demand
  - `MEMORY_AND_DISK`: data is stored in memory first; if memory is insufficient, Spark moves excess partitions to disk
  - `DISK_ONLY`: stores all partitions on disk
- Which storage level to choose?
  - Prefer in-memory storage whenever possible
  - Avoid disk unless recomputation is very expensive (e.g., filter a large amount of data)
  - Use replicated levels only when fast fault recovery is needed

RDD persistence: performance speedup

- Spark is faster than Hadoop up to 100x for ML workloads
  - Speedup comes from avoiding repeated disk I/O and serialization/deserialization overheads



Source: "Apache Spark: A Unified Engine for Big Data Processing"

RDD persistence: example

- We analyze how persistence is used in ML iterative algorithms
  - Iterative algorithms repeatedly reuse the same dataset
- We consider a naïve implementation of K-means
  - To avoid the input RDD (containing the data points to be clustered) is recomputed every iteration, exploit persistence by caching it

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

  - `cache()` stores the RDD in memory after the first computation
  - Subsequent iterations reuse the cached data directly
  - Avoids recomputation of `map(parseVector)` each iteration

https://github.com/apache/spark/blob/master/examples/src/main/python/kmeans.py

K-means in Spark

- Data file name, number of clusters K, and convergence threshold are provided from command line arguments

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

- Implementation uses NumPy

K-means in Spark

- First step: define utility functions `parseVector` and `closestPoint`

Convert raw input data into a float vector

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

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

```
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 the input file, convert into float numbers, set `K` and `convergeDist`
- Cache the RDD containing the data points
- Initialize the centroids `kPoints` by randomly selecting K data points

```
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
```

takeSample is an action used to retrieve a random sample from the RDD (False means without replacement)

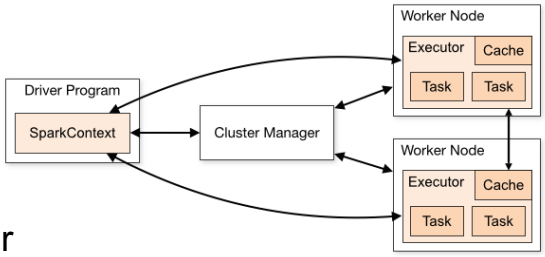
K-means in Spark

- Repeat until convergence
  - Assignment: map each data point to its closest centroid
  - Update: compute the new cluster centroids (average pattern)

```
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))
```

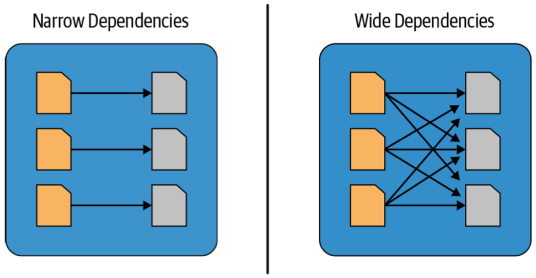
Spark execution model

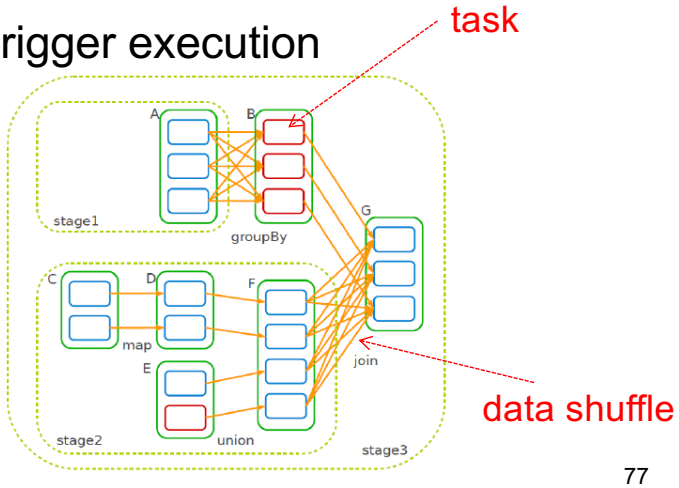
- A Spark application runs as a set of processes (executors) across the cluster, coordinated by the *driver program*
- Driver program
  - Main process of the application
  - Builds the execution plan
  - Schedules tasks across the cluster
- Executors
  - Processes launched on worker nodes
  - Execute tasks assigned by driver and store data in memory and/or disk
- Isolation: each application has its own executors



Spark execution model

- Application creates RDDs, applies transformation, and triggers actions: this builds a DAG of operations
- DAG is divided into stages
  - Stage: set of tasks without data shuffle, contains pipelined transformations with narrow dependencies
  - Task: smallest unit of execution that runs on executor and processes a single partition of data
- Actions (e.g., `collect`) trigger execution





Spark task execution

- Spark:
  - Creates one task per RDD partition
  - Schedules and assigns tasks to worker nodes
- Fully managed by Spark internally
- Cluster configuration
  - Number of executors
  - Amount of resources per executor (cores and memory)
  - By default, each executor runs one task per core

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: smallest unit of execution

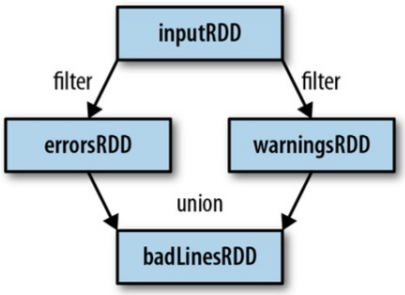
Fine grain

Fault tolerance

- Spark tracks all transformations used to build RDDs (their lineage DAG)

  Lineage information + RDD immutability = fault tolerance
- Recovery mechanism
  - If a partition is lost (e.g., executor fails), Spark recomputes the partition by replaying lineage transformations
  - No need for data replication

Example: RDD lineage DAG created during log analysis

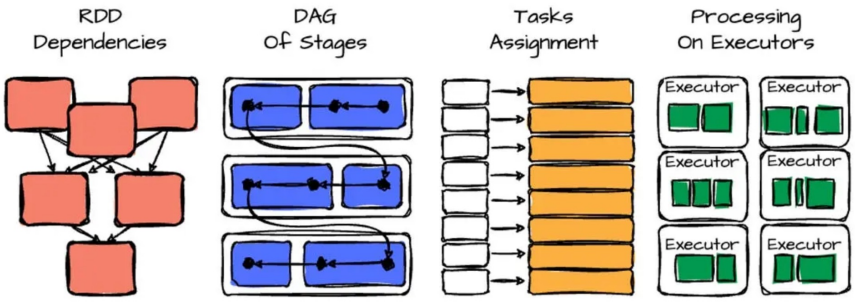


Application scheduling

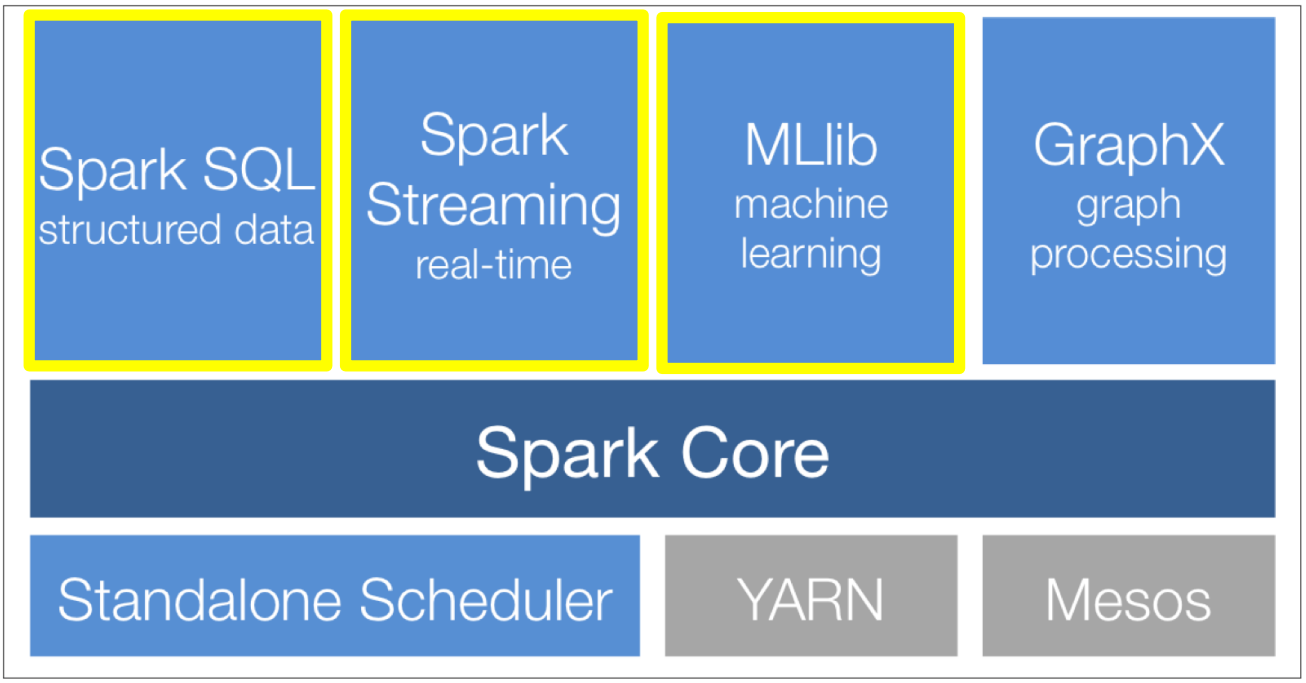
- When the application is launched, the driver program creates a DAG scheduler
- When an action is triggered, the DAG scheduler
  - Builds a physical execution plan (DAG of stages) from the RDD lineage DAG (e.g., `map`, `filter`, `join`), splitting the computation into stages based on shuffles
- Then, the Task scheduler
  - Receives a TaskSet from the DAG scheduler
    - TaskSet: set of tasks from a single stage, which execute the same code but on different data partitions
  - Assigns tasks to executors on workers based on data locality
    - If partition is in worker memory, assign task to that worker
- Failure handling
  - Task scheduler handles task failures
  - DAG scheduler handles stage-level failures by recomputing missing partitions via RDD lineage

Application scheduling

- Note: per-application scheduler
  - 1 application → 1 driver program → 1 DAG scheduler
  - No global optimization across multiple applications
- But global resource management by Cluster manager
  - E.g., Standalone, YARN, or Kubernetes, in charge of allocate cluster resources (executors) with a global view
- Cluster manager provides
  - Executor allocation
  - Resource sharing across applications
  - Resource fairness



Spark's high-level modules



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.,

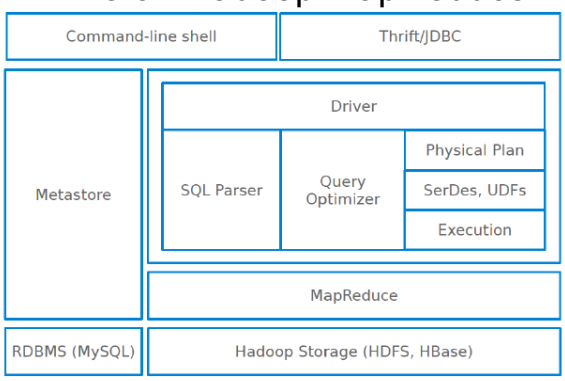
```
results = spark.sql(
  "SELECT * FROM people")
names = results.map(lambda p: p.name)
```

- 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

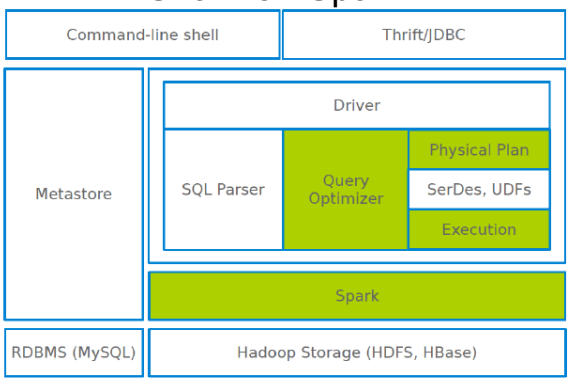
Spark SQL: how it began

- Goal was to extend Hive to run on Spark
  - Shark: modified Hive's backend to run over Spark, employing in-memory columnar storage
  - Shark limits
    - Only Hive data model
    - Query optimizer tied to Hadoop

Hive on Hadoop MapReduce

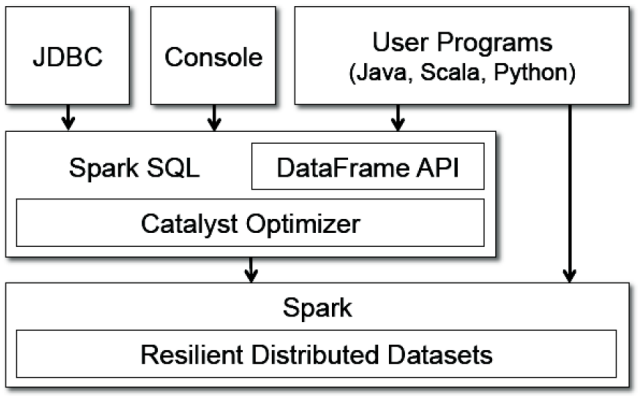


Shark on Spark



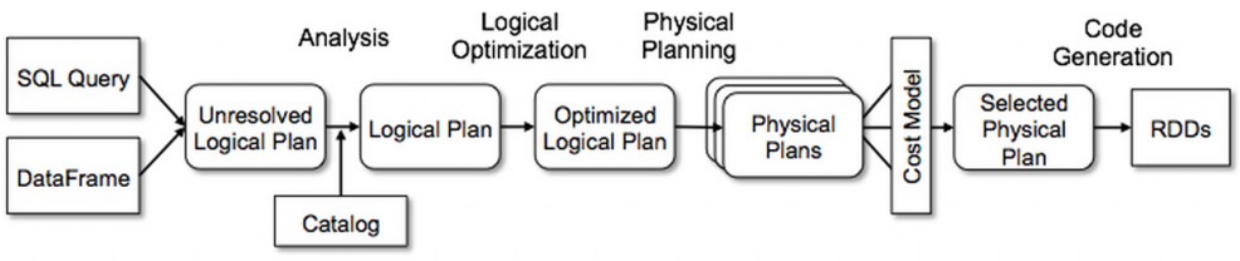
Spark SQL: Features

- 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 than RDD
- Best suited for structured and semi-structured data
- SparkSession: entry point
- Exploit Catalyst optimizer
- In common with RDDs:
  - Distributed in-memory collection of data
  - Immutable
  - Manipulated in similar ways to RDDs
  - Evaluated lazily
  - Persisted in memory
  - Spark keeps lineage of transformations

DataFrame and Dataset APIs

- **DataFrame** extends RDD by adding a schema (structured metadata about columns)
  - Data is organized as a distributed in-memory table with named columns
  - Spark SQL provides API to run SQL queries on DataFrames using SQL syntax
- Table-like structure 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|
+------------------+--------+--------------------+--------------------+-----------+------+
```

https://spark.apache.org/docs/latest/sql-programming-guide.html

DataFrame and Dataset APIs

- **Dataset** extends DataFrame by adding a type-safe, object-oriented interface
  - Represents a structured, strongly typed collection of data
  - A Dataset is a collection of typed JVM objects (e.g., Scala case classes or Java classes)
- DataFrame vs Dataset
  - DataFrame
    - More flexible and generally optimized for performance
    - Untyped (rows represented as generic objects)
  - Dataset
    - More type-safe and expressive, supports compile-time checks
    - May use more memory and has a slightly more limited API
- DataFrame and Dataset APIs have similar interfaces

RDDs vs DataFrames vs Datasets

Feature	RDD	DataFrame	Dataset
Abstraction level	Low-level	High-level	High-level
Data structure	Distributed collection of objects	Distributed table (rows & named columns)	Distributed collection of typed objects
Schema	No schema	Schema defined	Schema defined
Type safety	Yes (JVM-level)	No (generic Row)	Yes (compile-time)
API style	Functional	SQL + functional	SQL + functional + object-oriented
Optimization	Limited	Optimized (Catalyst, Tungsten)	Optimized (Catalyst, Tungsten)
Performance	Lower (manual optimization)	High	High (slightly less than DataFrame)
Ease of use	Harder	Easier	Moderate
Use case	Low-level control, unstructured	Structured data, analytics, SQL queries	Type-safe structured processing

Dataset API

- Combines benefits of RDDs (strong typing, lambda functions) with Spark SQL's optimized engine
- Available in Scala and Java (not Python)
- Can be created from JVM objects (Scala case classes or Java classes)
- Supports transformations (e.g., `map`, `flatMap`, `filter`, `groupBy`) and actions
- Lazy evaluation: computation runs only when an action is invoked
  - Internally uses a logical plan, which is optimized into a physical execution plan by Spark's optimizer

Dataset API

- How to create a Dataset
  - From file using `read` (e.g., JSON, CSV, Parquet)
  - From an existing RDD by converting it
  - Through transformations applied on existing Datasets

- Schema requirements
  - A Dataset requires a known schema (data types must be defined)
  - Schema can be:
    - Explicitly defined by user
    - Inferred automatically (e.g., from JSON or CSV files)

DataFrame API

- DataFrame: a *Dataset* organized into named columns
- Conceptually similar to a table in a relational DB, with advanced optimizations
- API available in Scala, Java, Python, and R
  - Can be used in PySpark shell
  - In Scala and Java: a DataFrame is a Dataset of Row objects
- Can be manipulated using functional transformations (similar to RDDs)
- How to create a DataFrame:
  - From tructured data files: JSON, CSV, Parquet, Avro, ORC, protobuf
  - From existing RDDs, either inferring the schema (reflection) or programmatically specifying the schema
  - Tables (e.g., Hive)

DataFrame API: constructing data frames

- Create DataFrame from RDD, list, or pandas DataFrame

```
>>> 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|
+------+---+
```

DataFrame API: constructing data frames

- Spark supports many file formats: text, CSV, JSON, Parquet, ORC, Avro
- Create DataFrame from file
  - Use generic `read.load()` method to load data into DataFrame
    - Default file format: Parquet
    - Additional options can be specified (e.g., delimiter, header)
    - File format can also be specified explicitly: `read.format("csv").load(...)`
  - Format-specific methods are available: `read.csv()`, `read.json()`, `read.parquet()`
  - Schema inference is supported for CSV and JSON files
  - Example: loading a CSV file (default separator is ",")

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

https://github.com/apache/spark/blob/master/examples/src/main/python/sql/datasource.py

DataFrame API: load CSV file

- Example: load CSV file using `read.csv`

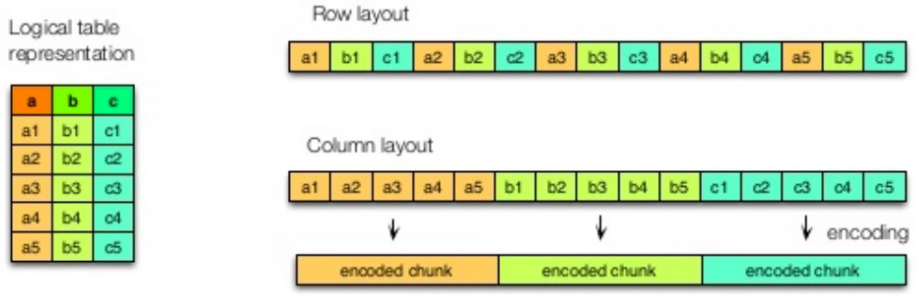
```
>>> df = spark.read.csv("/data/address.csv", header=True)
>>> df.show()
+------------------+--------+--------------------+------------------+-----------+------+
|              Name| Surname|             Address|              City|      State|   ZIP|
+------------------+--------+--------------------+------------------+-----------+------+
|              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|
+------------------+--------+--------------------+------------------+-----------+------+
```

- Spark can infer schema of each column 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

- Columnar data file format designed for efficient storage and retrieval https://parquet.apache.org
- Supported by Spark for both reading and writing
  - Widely supported by data processing frameworks, independent of programming language or data model
- Interoperable with other data storage formats
  - Avro, Thrift, Protocol Buffers, ...
- Unlike row-based formats (CSV), Parquet stores data by columns instead of rows



Parquet file format: performance

- Why Parquet is faster and storage-saving?
- Reads only required columns (column pruning)
- Provides efficient data compression and encoding schemes
  - Several compression codecs (e.g., gzip, snappy) with different compression ratio and processing cost
- 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**

Parquet file format: other features

- Schema of original data is automatically preserved
- Supports schema evolution
  - E.g., adding/removing columns
- Supports predicate pushdown
  - Applies filtering at the data scan level, reducing the amount of data read from disk

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 file using `SparkSession.read.json()`

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

DataFrame - RDD conversion

- Convert DataFrame to RDD when lower-level control is needed
  - Use `.rdd` method on a DataFrame
- Convert RDD to DataFRame
  - Use `spark.createDataFrame()` on an existing RDD
  - Schema can be inferred or explicitly defined

DataFrame API: benefits

- Let's consider expressivity and simplicity
- Example: aggregate all the ages for each name, group by name, and then average the ages
  - With RDDs (see slide 54), we instruct Spark *how to* aggregate keys and compute averages using lambda functions: hard to read and cryptic
  - With DataFrames, we instruct Spark *what to do*

```
from pyspark.sql.functions import avg
# Create a DataFrame
data_df = spark.createDataFrame([("Brooke", 20), ("Denny",
31), ("Jules", 30), ("TD", 35), ("Brooke", 25)], ["name",
"age"])
# Group the same names together, aggregate their ages,
# and compute an average
avg_df = data_df.groupBy("name").agg(avg("age"))
# Show the results of the final execution
avg_df.show()
```

Spark Streaming and Structured Streaming

- Spark Streaming (legacy): streaming engine
  - Processes data streams using micro-batches
    - Data is ingested and analyzed in small time intervals
  - Uses a high-level abstraction called DStream (Discretized Stream)
    - Dstream: continuous stream represented as a sequence of RDDs
  - Internally, it works as:

- Spark Structured Streaming
  - Built on Spark SQL engine
  - Uses DataFrames/Datasets instead of DStreams
  - Treats streaming data as an unbounded table

See hands-on lesson

Spark MLlib

- Spark library for machine learning
  - Includes 2 packages:
    - `spark.ml`: MLlib DataFrame-based API to support a variety of data types
    - `spark.mllib`: MLlib RDD-based API (maintenance mode)
- Provides many ML algorithms, including:
  - Classification (e.g., logistic regression), regression, clustering (e.g., K-means), recommendation (e.g., collaborative filtering), decision trees, random forests
- 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

```
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))
```

ML package

LIBSVM format

Create a LogisticRegression instance

Learn a LogisticRegression model, which uses the parameters stored in `lr`

Spark MLlib: K-means

- Input: feature vector
- Output: predicted cluster assignments and cluster centroids
- How to choose good initial centroids
  - Critical for solution quality and convergence
- K-means++ (initialization method):
  - Goal: select K initial centroids that are well spread out
  - Pick the first centroid uniformly at random from the data points
  - Pick each subsequent centroid with probability proportional to the squared distance from the nearest existing centroid
- Limitation: K-means++ is sequential
  - Requires K passes over the data

https://en.wikipedia.org/wiki/K-means%2B%2B

Spark MLlib: K-means

- MLlib implements a parallelized variant: K-means||
  - Based on distributed oversampling: pick multiple candidate centroids in each pass (instead of one)
- K-means||
  - Start with one randomly chosen centroid
  - In each pass, sample multiple new candidate points (usually 2K)
    - Probability proportional to squared distance from nearest existing centroid
  - Repeat for a small number of rounds (logarithmic in data size)
  - Reduce candidates to K final centroids (via clustering step)

Bahmani et al, Scalable k-means++, Proc. VLDB Endow., 2012.
theory.stanford.edu/~sergei/papers/vldb12-kmpar.pdf

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 measures how well clusters are separated

It evaluates how close each point is to its own cluster compared to neighboring clusters

Spark ML: Pipeline API

- Structured sequence of steps for building ML workflow
  - Goal: build scalable, reproducible, and production-grade ML app in Spark https://spark.apache.org/docs/latest/ml-pipeline.html
- Core components:
  - DataFrame: input dataset
  - Transformer: transforms DataFrames (e.g., scaling, feature extraction); implements `transform` method
  - Estimator: learns from preprocessed data and outputs a model; implements `fit` method
  - Pipeline: chains multiple stages into a single workflow
  - PipelineModel: the trained pipeline (after fitting), a serialized file containing transformation logic and trained weights
- ML workflow in Spark:
  - Raw data → preprocessing (transformers)
  - Estimators → trained models
  - PipelineModel: applied to new data for predictions

Spark ML: Pipeline API and MLOps

- Spark's unified engine allows to handle each stage of a ML pipeline using the same API both for batch or real-time stream data

Example: retail_clustering.py

- Types of ML pipelines
  - Feature pipelines (*data engineering* stage)
    - Turn raw data into a clean, feature-ready dataset
    - In Spark, often built using `Transformer`
  - Training pipelines (*data science* stage)
    - Pass them through an Estimator to find patterns
    - Spark tool: `pyspark.ml.Pipeline`
  - Inference pipelines (*production* stage)
    - Load the PipelineModel to generate predictions on new, "unseen" data
    - Spark tool: Spark batch job or Structured Streaming (for real-time); load the PipelineModel and call `transform`

Combining processing tasks with Spark

- Easy to seamlessly combine different Spark libraries within the same application
- Example: ML streaming pipeline using K-means to group data by geographic location
  - Read historical data using Spark SQL
  - Train a K-means clustering model using MLlib
  - Apply the model to a real-time data stream in order to assigns each live point to a geographic cluster

Combining processing tasks with Spark

Example: tweet_process.py

```
from pyspark.sql import SparkSession
from pyspark.ml.clustering import KMeans
from pyspark.ml.feature import VectorAssembler
import pyspark.sql.functions as F

# Initialize Spark
spark = SparkSession.builder \
    .appName("TweetClustering") \
    .getOrCreate()
# Set logging level to WARN
spark.sparkContext.setLogLevel("WARN")

# Mock data for training (location is double)
data = [(10.5, "en"), (22.1, "es"), (15.3, "fr"), (11.0, "en"), (25.0, "es")]
columns = ["location", "language"]
spark.createDataFrame(data, columns) \
    .createOrReplaceTempView("old_tweets")
```

Combining processing tasks with Spark

```
# Training pipeline
trainingData = spark.sql("SELECT location, language FROM old_tweets")
assembler = VectorAssembler(inputCols=["location"], outputCol="features")
trainingDataTransformed = assembler.transform(trainingData)

kmeans = KMeans().setFeaturesCol("features") \
    .setPredictionCol("prediction_cluster").setK(3)
model = kmeans.fit(trainingDataTransformed)

# Streaming: tuple contains timestamp and value
raw_stream = spark.readStream \
    .format("rate") \
    .option("rowsPerSecond", 1) \
    .load() \
    .withColumn("location", F.col("value").cast("double"))

# Transform stream using the same assembler
stream_features = assembler.transform(raw_stream)
```


Combining processing tasks with Spark

```
# Predict using the trained model
predictions = model.transform(stream_features)

# Output to console
query = predictions.select("timestamp", "location", "prediction_cluster") \
    .writeStream \
    .outputMode("append") \
    .format("console") \
    .start()

query.awaitTermination()
```

References

- Zaharia et al., Apache Spark: A Unified Engine For Big Data Processing, Commun. ACM, 2016. https://dl.acm.org/doi/pdf/10.1145/2934664
- Zaharia et al., Spark: Cluster Computing with Working Sets, HotCloud'10. https://static.usenix.org/events/hotcloud10/tech/full_papers/Zaharia.pdf
- Zaharia et al., Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing, NSDI'12 https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf
- Ambrust et al., Spark SQL: Relational Data Processing in Spark, ACM SIGMOD'15 https://dl.acm.org/doi/pdf/10.1145/2723372.2742797
- Damji et al., Learning Spark - Lightning-Fast Big Data Analysis, 2nd edition, O'Reilly, 2020 https://www.oreilly.com/library/view/learning-spark-2nd/9781492050032