Introduction to Apache Spark

Architecture, RDD, DataFrames, SQL & MLlib

Prasanth Kothuri, CERN

IT-Database Group

Outline

What is Apache Spark?

Spark Abstractions

Spark Architecture

Spark Data APIs

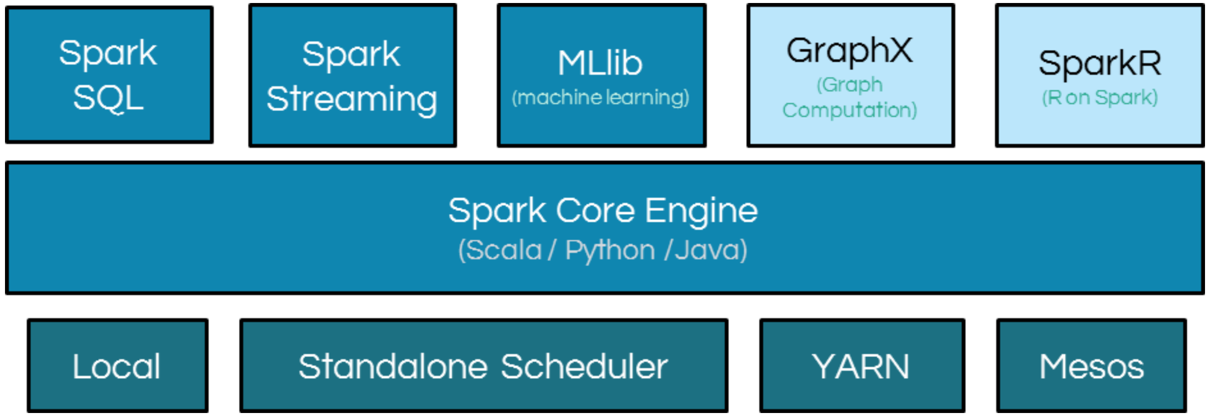
- RDD
- Dataframe
- SQL

Spark Mlib

What is Apache Spark?

- Apache Spark is an open-source parallel processing framework with expressive development APIs (in multiple languages) that allows for sophisticated analytics, real-time streaming and machine learning on large datasets

- Spark ecosystem



Spark Abstractions

Two main abstractions of Spark

- RDD – Resilient Distributed Dataset

- DAG – Direct Acyclic Graph

RDD

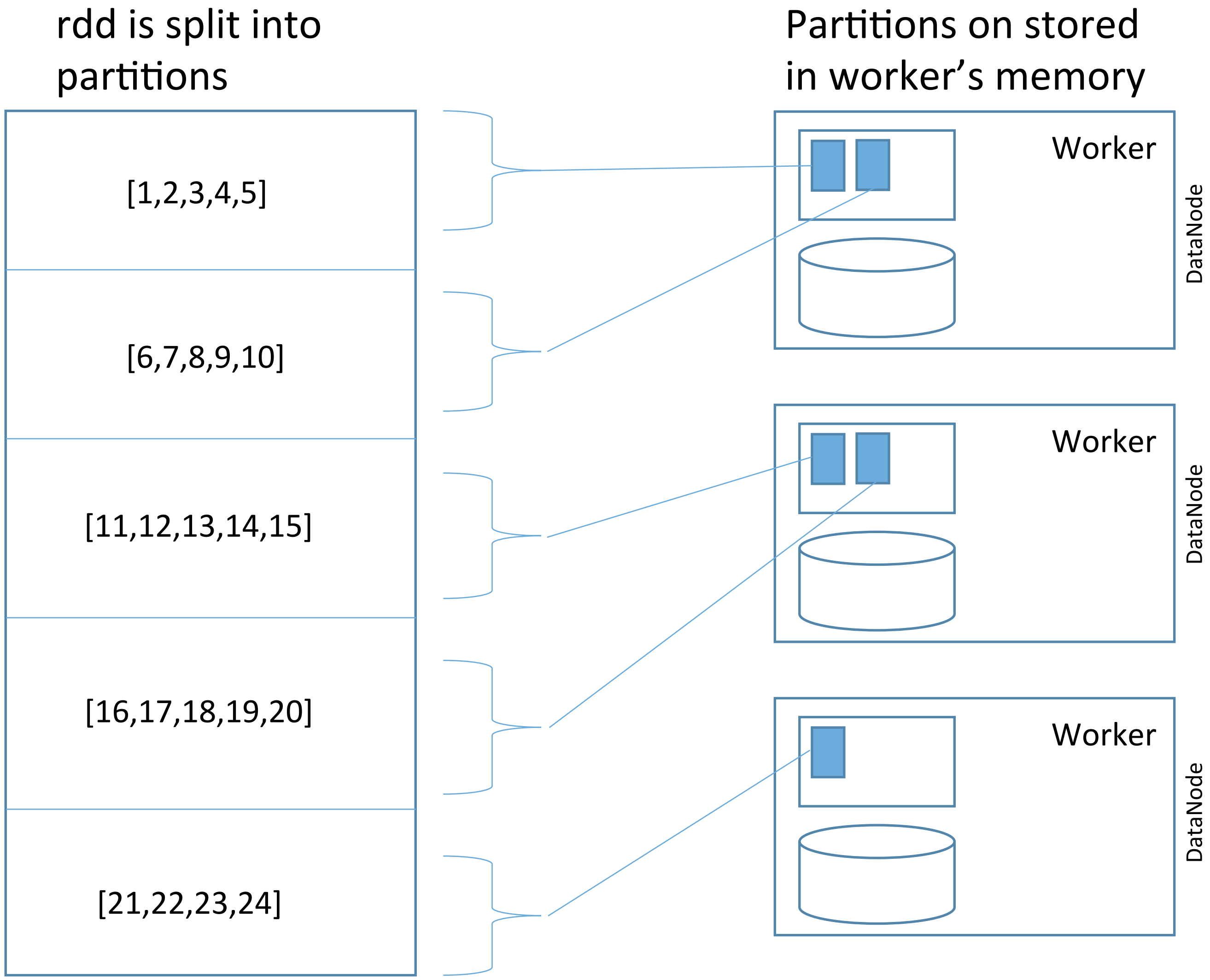
Resilient Distributed Datasets (RDDs) are the primary abstraction in Spark – a immutable distributed collection of records that can be operated on in parallel

There are currently two ways to create them:

- **parallelized collections** – take an existing python/scala collection and run functions on it in parallel
- **Hadoop datasets** – run functions on each record of a file(s) in Hadoop distributed file system

RDD: Example

```
rdd = sc.parallelize(range(1,25),5)
```



Partitioned: RDD is partitioned and distributed across worker nodes of the cluster

In-Memory : RDD is stored in memory as much (size) and long (time) as possible

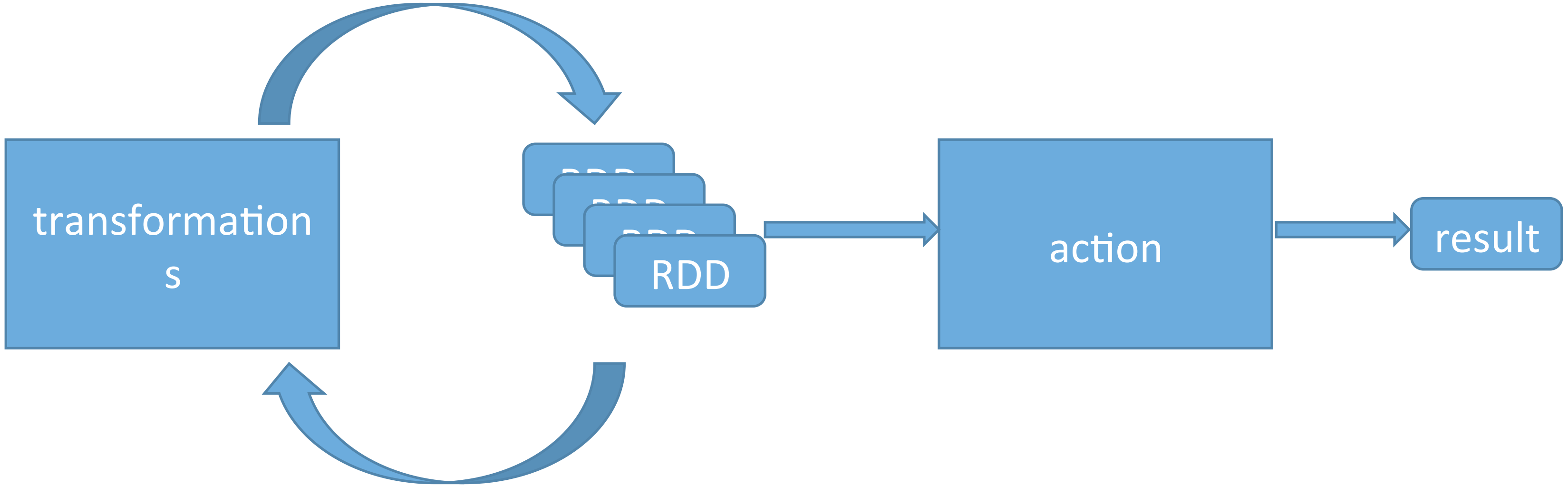
Immutable: does not change once created, can only be transformed into new RDDs

Typed: RDDs have types

Cacheable: hold all the data in a persistent storage like memory (preferrable) or disk

RDD: Actions and Transformations

- Two types of operations on RDDs:
  transformations and actions
- transformations – lazy operations that return another RDD
- actions – operations that trigger computation and return value



DAG

Direct Acyclic Graph – sequence of computations performed on data

- Node – RDD partition
- Edge – transformation on top of data
- Acyclic – graph cannot return to the older partition
- Direct – transformation is an action that transistions data partition state (from X to Y)

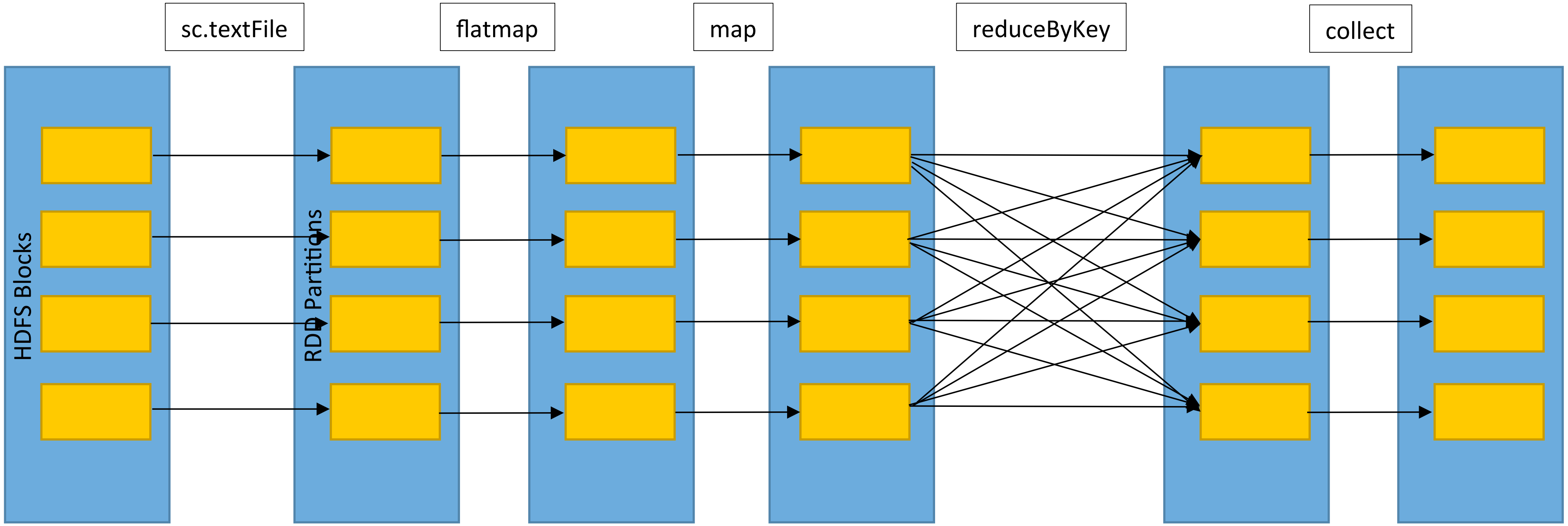
DAG

WordCount example

```
text_file = sc.textFile("hdfs:///user/pkothuri/sparkTraining/
LICENSE.txt")
counts = text_file.flatMap(lambda line: line.split(" ")) \
             .map(lambda word: (word, 1)) \
             .reduceByKey(lambda a, b: a + b)
for x in counts.collect():
     print x
```

DAG

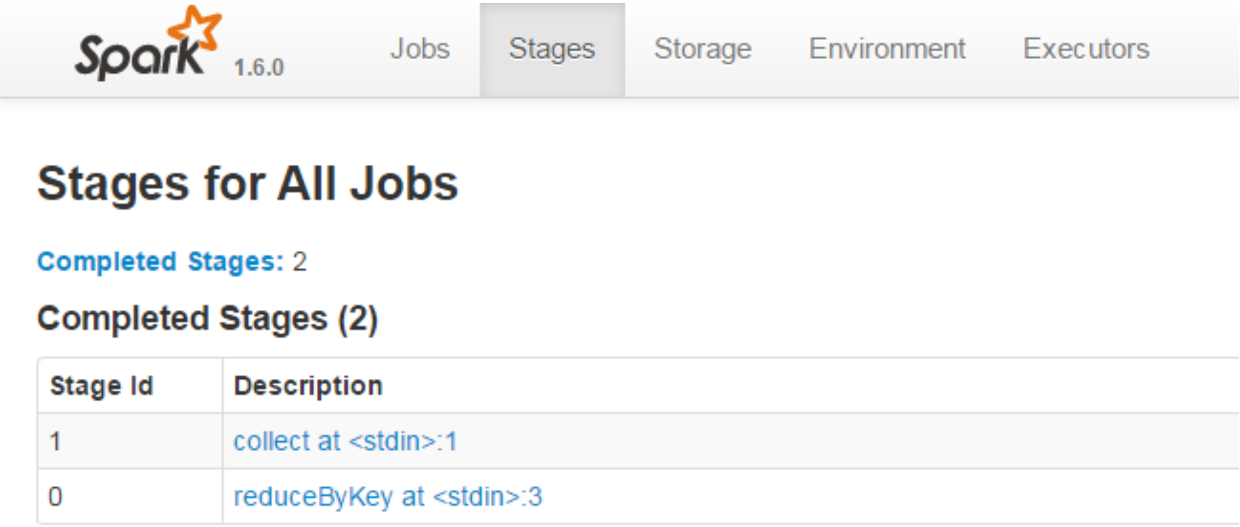
WordCount example

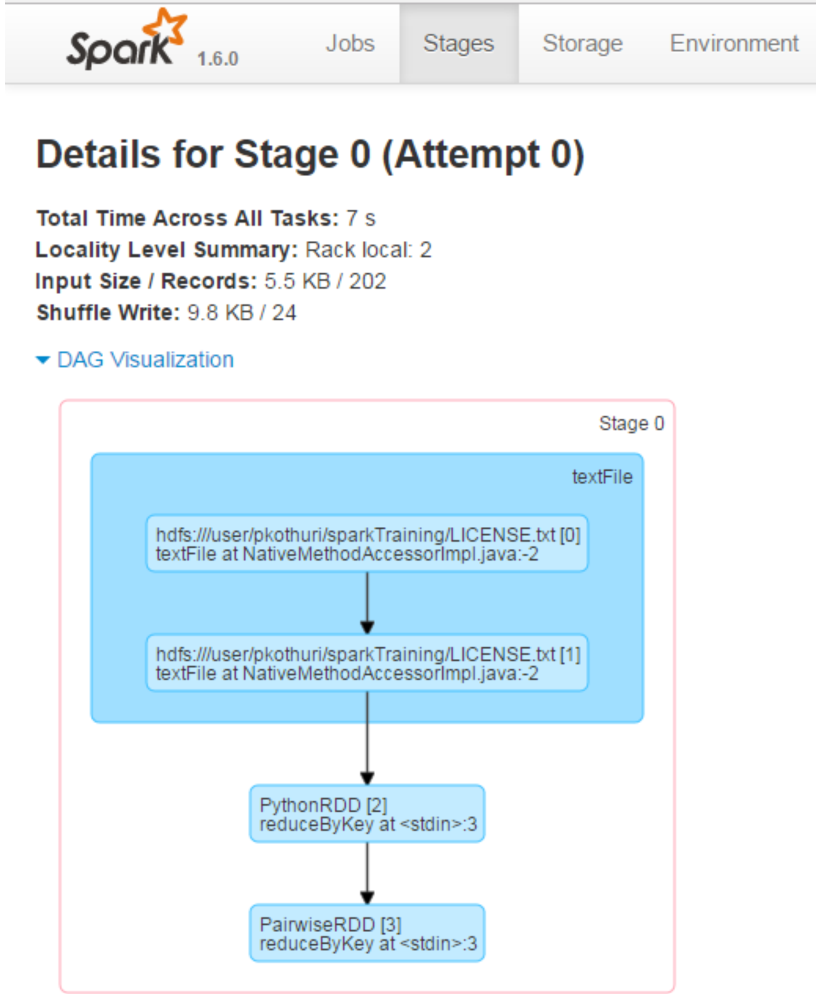


Spark WEB UI

Every SparkContext launches a web UI, by default on port 4040, that displays useful information about the application. This includes:

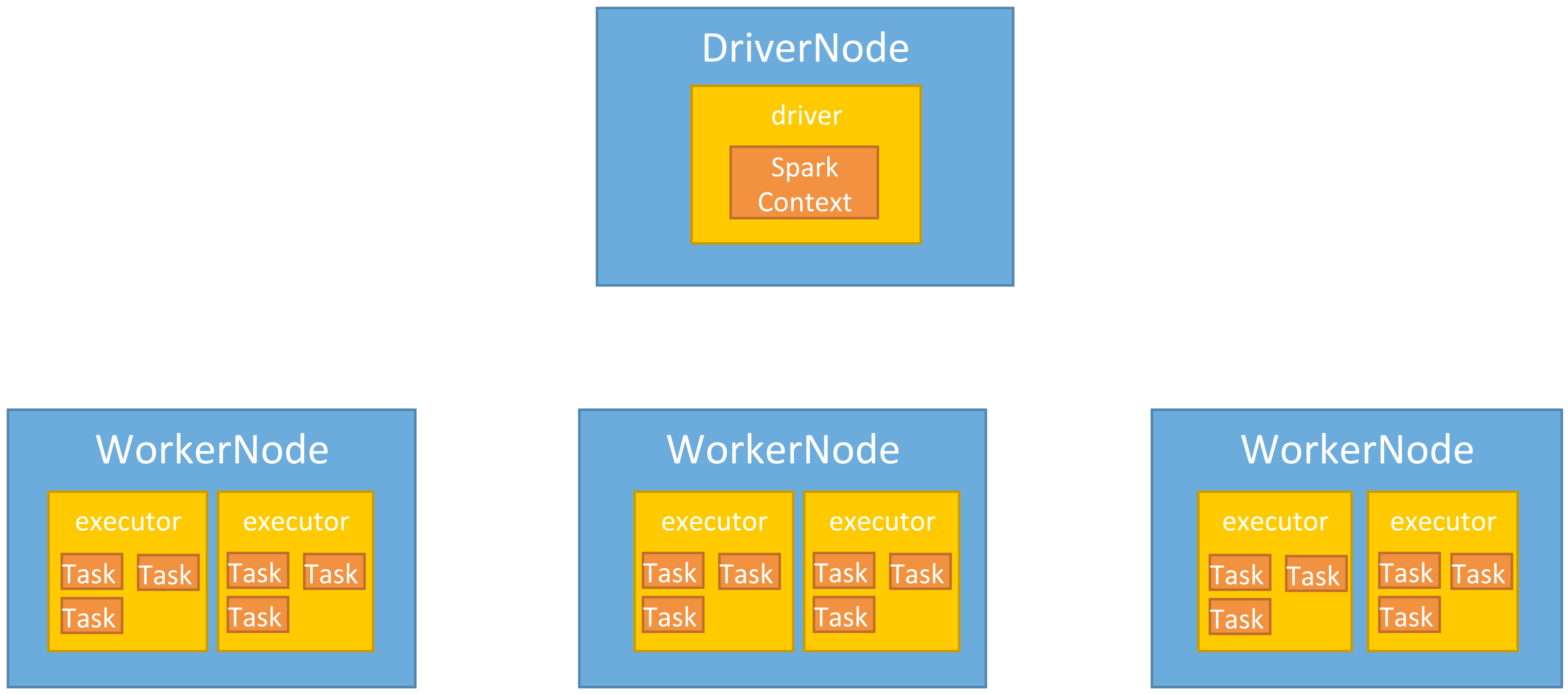
- A list of scheduler stages and tasks
- A summary of RDD sizes and memory usage
- Environmental information.
- Information about the running executors





Spark Architecture

Spark Cluster



Spark Architecture

Driver

- Entry point for Spark Shell (Scala, Python)
- SparkContext is created here and resides here
- Graph is built and submitted to DAGScheduler
- DAGScheduler divides it into stages and tasks
- Schedules tasks and controls their execution

Spark Architecture

Executor

- Reads data from HDFS (or external sources)
- Stores the data in cache in JVM heap or on HDDs
- Performs all data processing
- Writes data to HDFS (or external sources)

Spark Architecture

Application Decomposition

- Application
  - Single instance of SparkContext that stores some data processing logic and can schedule series of jobs, sequentially or in parallel

- Job
  - Complete set of transformations on RDD that finishes with action or data saving, triggered by the driver application

Spark Architecture

Application Decomposition

- Stage
  - Set of transformations that can be pipelined and executed by a single independent worker. Usually it is app the transformations between «read», «shuffle», «action» and «save»
- Task
  - Execution of the stage on a single data partition. Basic unit of scheduling

DAG scheduler

- When an action is called on the RDD, Spark creates DAG and submits to the DAG scheduler

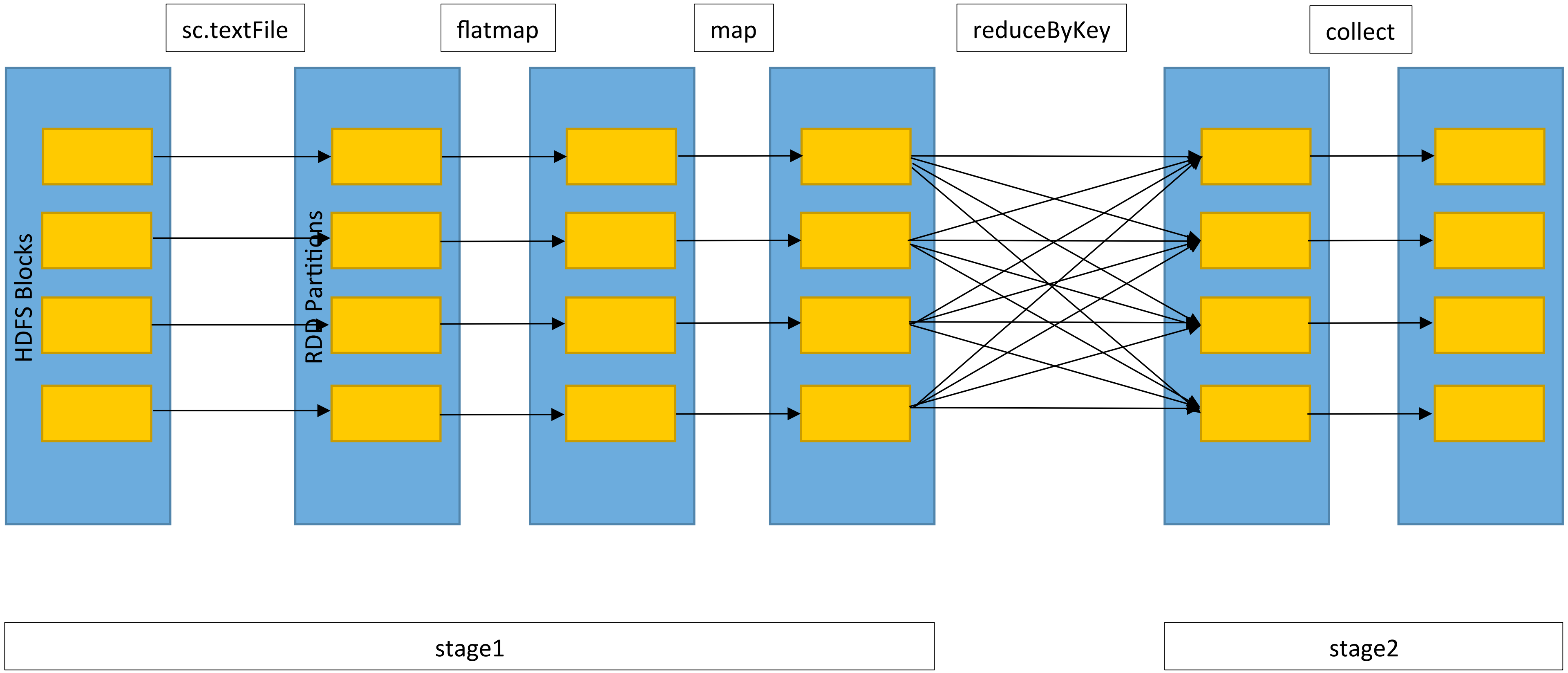
- The DAG scheduler divides operators into stages of tasks

- The stages are created based on the transformations, the narrow transformations are grouped (pipelined) into a single stage

- The DAG scheduler submits the stages to the task scheduler
  - The number of tasks depend on number of partitions
  - The number of tasks submitted depends on number of available executors

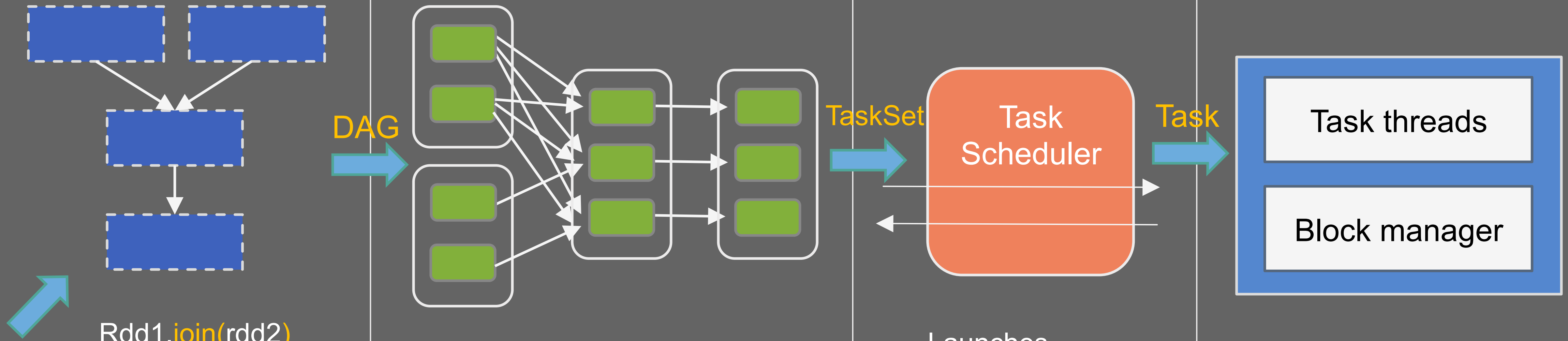
DAG

WordCount example



SCHEDULING PROCESS

RDD Objects | DAG Scheduler | Task Scheduler | Executor



RDD Objects

Rdd1.join(rdd2)
.groupBy(…)
.filter(…)

- Build operator DAG

DAG Scheduler

- Split graph into *stages* of tasks
- Stage boundaries are at wide transformations (shuffle)
- Submit each stage as ready

Task Scheduler

- Launches individual tasks
- Retry failed or straggling tasks

Executor

- Execute tasks
- Store and serve blocks

Essential Spark Operations

TRANSFORMATIONS

General

- map
- filter
- flatMap
- mapPartitions
- groupBy
- sortBy
- flatMapValues
- groupByKey
- reduceByKey
- foldByKey
- sortByKey
- combineByKey

Math / Stats

- sample
- sampleByKey
- randomSplit

Set Theory

- union
- intersection
- subtract
- distinct
- cartesian
- zip
- join
- fullOuterJoin
- leftOuterJoin
- rightOuterJoin

Data Structure / I/O

- keyBy
- zipWithIndex
- zipWithUniqueID
- zipPartitions
- coalesce
- repartition
- repartitionAndSortWithinPartitions
- pipe
- partitionBy

ACTIONS

General

- reduce
- collect
- aggregate
- fold
- first
- take
- forEach
- top
- treeAggregate
- treeReduce
- forEachPartition
- collectAsMap
- keys
- values

Math / Stats

- count
- takeSample
- max
- min
- sum
- histogram
- mean
- variance
- stdev
- sampleVariance
- countApprox
- countApproxDistinct

Set Theory

- takeOrdered

Data Structure / I/O

- saveAsTextFile
- saveAsSequenceFile
- saveAsObjectFile
- saveAsHadoopDataset
- saveAsHadoopFile
- saveAsNewAPIHadoopDataset
- saveAsNewAPIHadoopFile

Transformations: **map** and **filter**

MAP

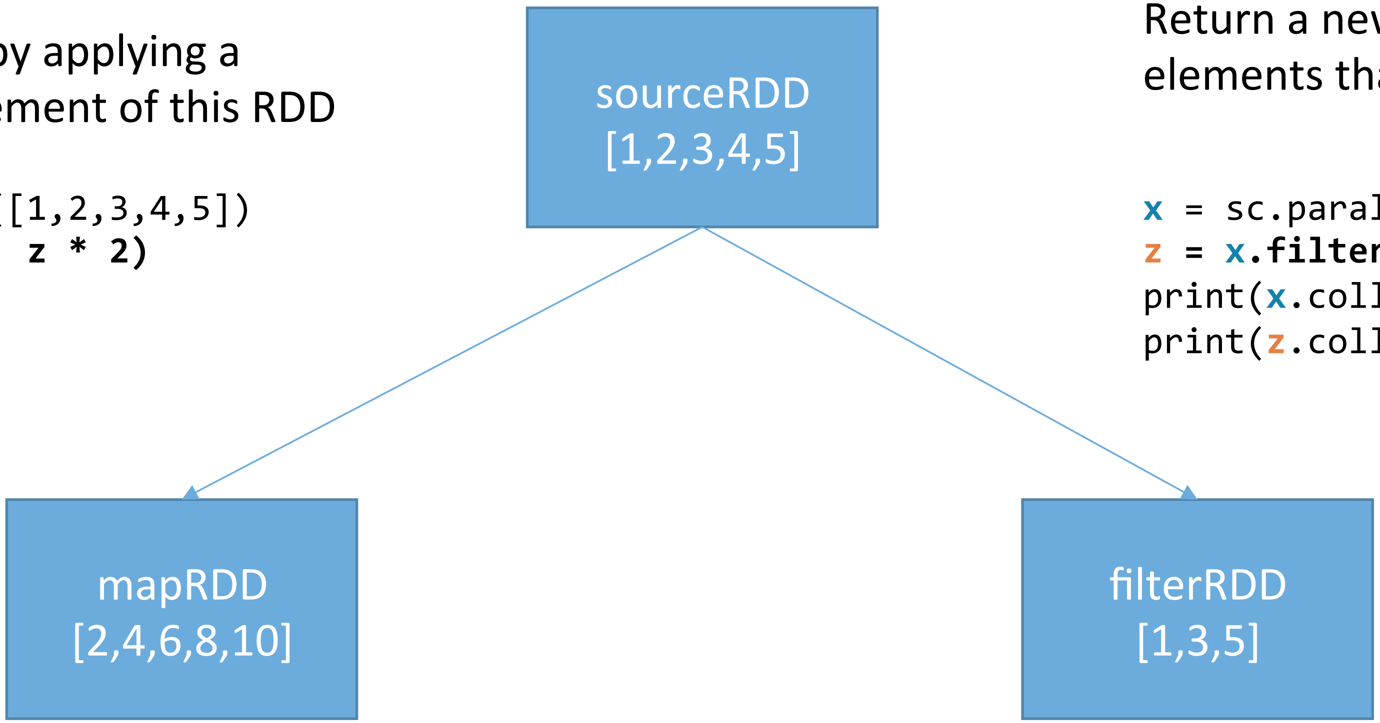
Return a new RDD by applying a function to each element of this RDD

```
x = sc.parallelize([1,2,3,4,5])
y = x.map(lambda z: z * 2)
print(x.collect())
print(y.collect())
```

FILTER

Return a new RDD that only has elements that pass the filter() function

```
x = sc.parallelize([1,2,3,4,5])
z = x.filter(lambda z: z % 2 != 0)
print(x.collect())
print(z.collect())
```



Transformations: **map** and **flatmap**

MAP

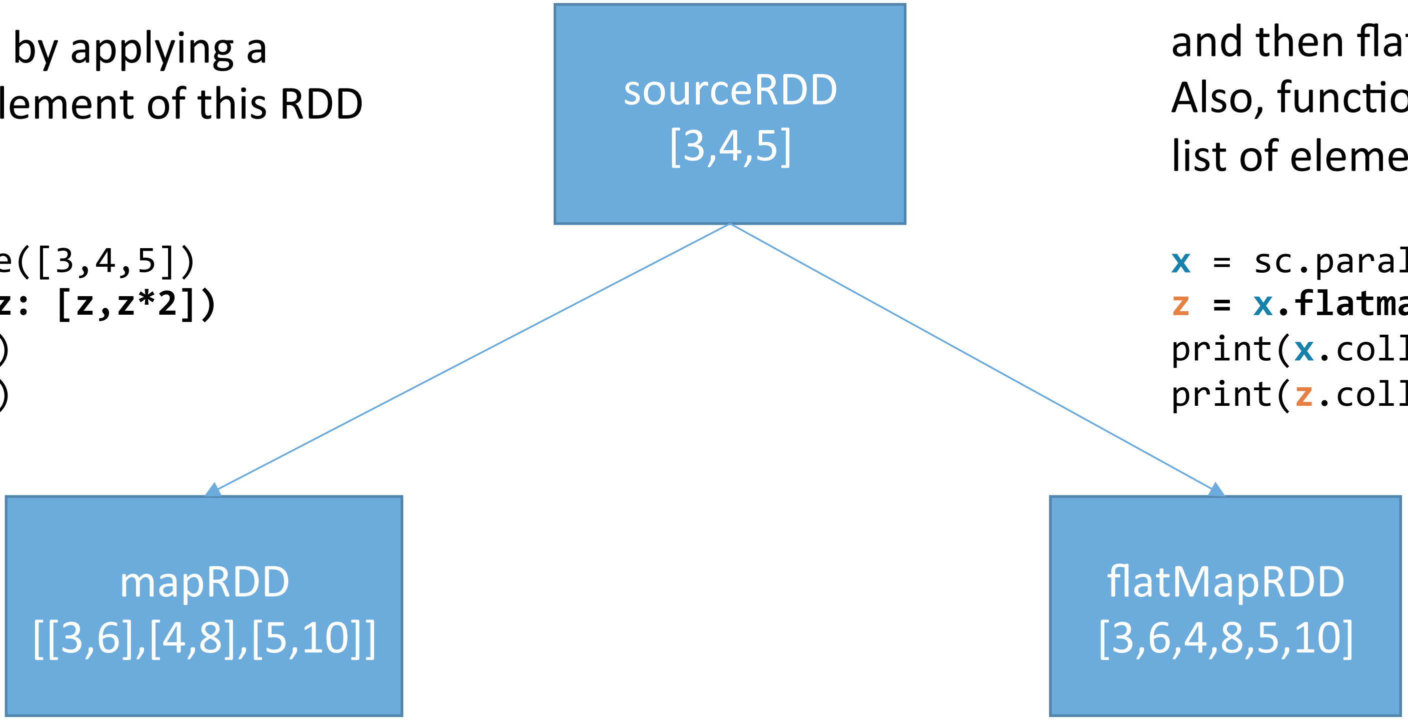
Return a new RDD by applying a function to each element of this RDD

```
x = sc.parallelize([3,4,5])
y = x.map(lambda z: [z,z*2])
print(x.collect())
print(y.collect())
```

FLATMAP

Return a new RDD by applying a function to each element of the RDD, and then flattening the results.
Also, function in flapMap can return a list of elements (0 or more)

```
x = sc.parallelize([3,4,5])
z = x.flatmap(lambda z: [z,z*2])
print(x.collect())
print(z.collect())
```



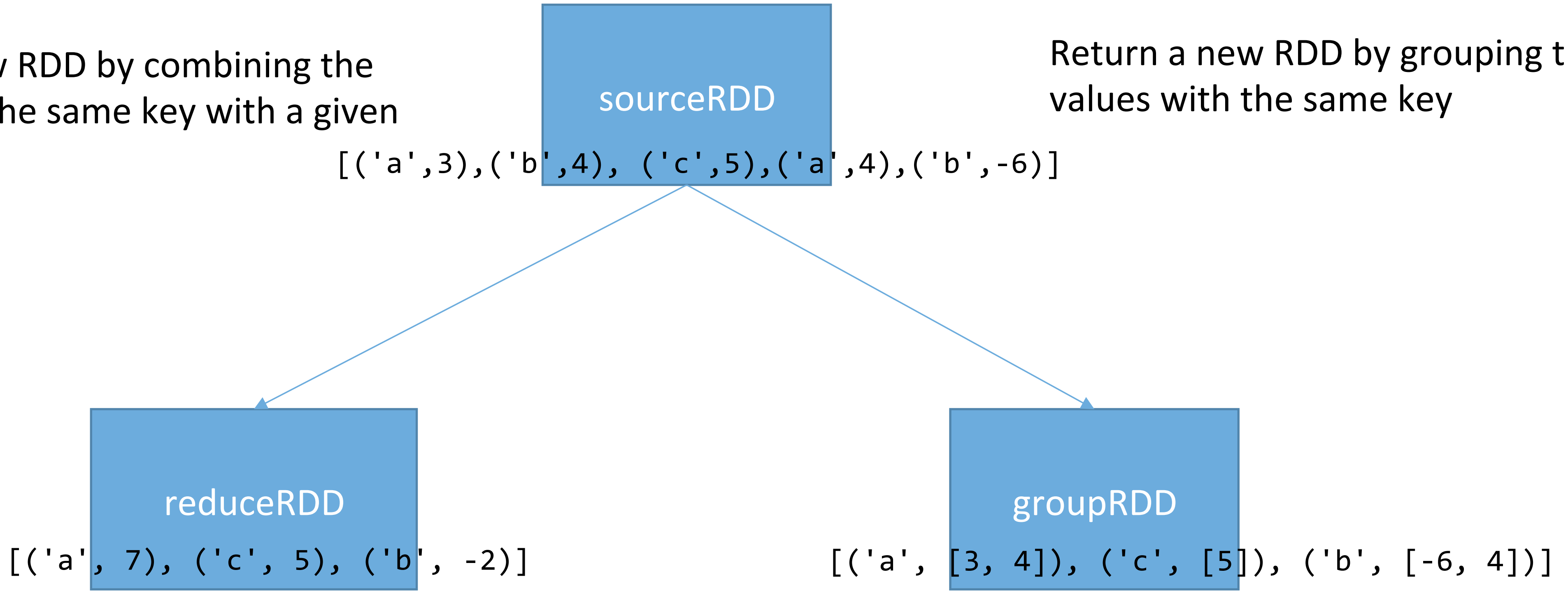
Transformations: **reduceByKey** and **groupByKey**

REDUCEBYKEY

Return a new RDD by combining the values with the same key with a given function

GROUPBYKEY

Return a new RDD by grouping the values with the same key




```
x = sc.parallelize([('a',3),('b',4), ('c',5),('a',4),('b',-6)])
y = x.reduceByKey(add)
print(x.collect())
print(y.collect())
```

```
x = sc.parallelize([('a',3),('b',4), ('c',5),('a',4),('b',-6)])
I = x.groupByKey()
y = I.map(lambda z: (z[0],sum(z[1])))
print(x.collect())
print(y.collect())
```

Actions

Actions trigger the computations and resulting result must fit in the driver JVM

collect():	returns all the elements of the RDD as an array to the driver, should be done after a filter or other operation
count()	returns the number of elements in the RDD
countByKey()	for pair RDDs returns (K, Int) pairs with the count of each key
first()	returns the first element of the RDD
take(n)	returns an array with first n elements
saveAsTextFile()	writes the elements of the RDD as a text file to HDFS or local filesystem
getNumPartitions()	returns the number of partitions of the RDD

RDD: Demo

Login to lxplus

cd /eos/user/p/pkothuri
git clone https://github.com/prasanthkothuri/sparkTraining.git

Login to SWAN

https://swan.cern.ch/
open sparkTraining->notebooks->Tutorial_RDD_Final.ipynb

Lifecycle of a Spark program

Create some input RDDs from external data or parallelize a collection in your driver program

Lazily transform them to define new RDDs using transformations like filter() or map()

Ask Spark to cache() any intermediate RDDs that will need to be reused

Launch actions such as count() and collect() to kick off a parallel computation, which is then optimized and executed by Spark

Persistence and Cache

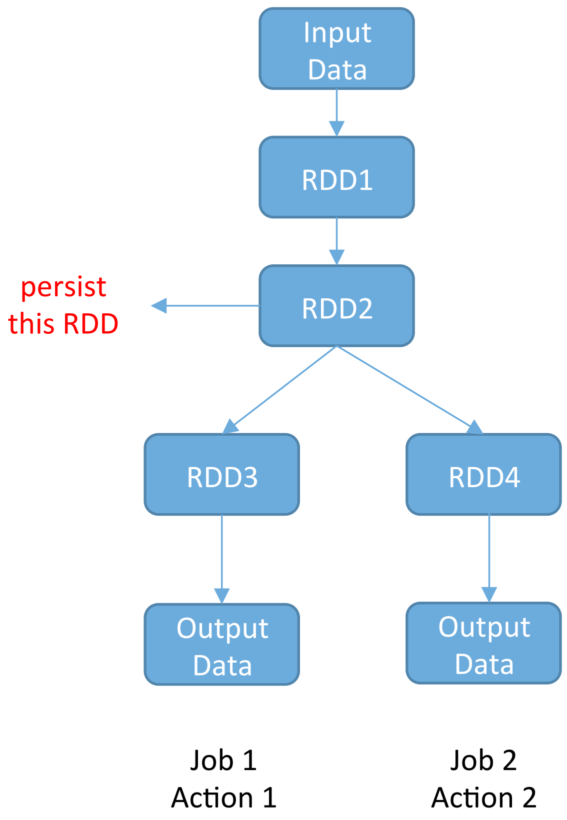
persist() is an action which triggers computation and persists a dataset in memory across operations

You can persist with different storage level options (e.g MEMORY_ONLY or MEMORY_AND_DISK)

persist() or cache() is key for iterative algorithms

cache() is same as persist() with MEMORY_ONLY storage option

Example DAG



DataFrames

- An extension to the existing RDD API
- DataFrame is an RDD with schema
- DataFrames have numerous optimizations that make them much faster than RDD (predicate pushdown, bloom-filter)
- Write less code – solve problems concisely using dataframe functions
- Inspired by data frames in Python (pandas) and R

Write Less Code: DataFrame vs RDD

Using RDDs

```
w_rdd.map(lambda record: (record.NAMELAST, 1) ) \
     .reduceByKey(add) \
     .map(lambda (x,y):(y,x)) \
     .sortByKey(False) \
     .collect()
```

Using DataFrames

```
w_df.groupBy("NAMELAST") \
    .count() \
    .orderBy("count")
    .show()
```

Construct a DataFrame

```
# Create a DataFrame from json file
df = sqlContext.read.json("/tmp/read_my_shiny.json")
```

```
# Create a DataFrame by loading a parquet file
df = sqlContext.read.parquet("/tmp/path_to_the_parquet_file")
```

```
df = sqlContext.read.format('jdbc')
\ .options(driver='oracle.jdbc.driver.OracleDriver',url='jdbc:oracle:thin:username/
password@host:port:servicename',dbtable='table_name') \
.load()
```

```
# Convert a DataFrame
df = rdd.toDF()
```

Schema Inference

DataFrames have schemas and can infer the schema from the type of the data being read

```
>>> df.printSchema()
root
 |-- LocID: string (nullable = true)
 |-- Location: string (nullable = true)
 |-- VarID: string (nullable = true)
 |-- Variant: string (nullable = true)
 |-- Time: string (nullable = true)
 |-- MidPeriod: string (nullable = true)
 |-- SexID: string (nullable = true)
 |-- Sex: string (nullable = true)
 |-- AgeGrp: string (nullable = true)
 |-- AgeGrpStart: string (nullable = true)
 |-- AgeGrpSpan: string (nullable = true)
 |-- Value: string (nullable = true)
```

A Parquet file has a schema (column names and types) that DataFrames can use.

What if the data doesn't have schema (e.g csv)

create an RDD of particular type using python namedtuple, dict and convert RDD to DataFrame

You can also specify the column names in .toDF() function

DataFrame: Transformations and Actions

DataFrames are *lazy*. *Transformations* contribute to the query plan, but they don't execute anything.

Actions cause the execution of the query.

Transformation examples

- filter
- select
- drop
- intersect
- join

Action examples

- count
- collect
- show
- head
- take

Execution of the query means:

- Spark initiates a distributed read of the data source
- The data flows through the transformations
- The result of the action is pulled back into the driver JVM.

Transformations: select(), filter() and show()

filter()

The filter() method allows you to filter out rows from you results

select()

similar to SQL SELECT, allows you to limit the results to specific columns

show()

displays the first n elements in the DataFrame (n defaults to 20)

```
>>>w_df.filter(w_df.NAMELAST ==
'KOTHURI').select(w_df.NAMELAST,w_df.NAMEFIRST,w_df.APPT_START_DATE).show()
+--------+----------------+---------------+
|NAMELAST|       NAMEFIRST|APPT_START_DATE|
+--------+----------------+---------------+
| KOTHURI|         KRISHNA|  9/27/14 11:30|
| KOTHURI|           SARYU|  9/27/14 11:30|
| KOTHURI|VENKATAGOPALARAO|  9/27/14 11:30|
+--------+----------------+---------------+
```

Transformations: orderBy(), groupBy()

orderBy()

The orderBy() method allows you sort the results

groupBy()

groupBy() groups the elements by a specific column value, often used with count

```
>>>w_df.groupBy(w_df.NAMELAST) \
    .count() \
    .orderBy("count",ascending=False) \
    .show(10)

+--------+-----+
|NAMELAST|count|
+--------+-----+
|   Smith|25908|
| Johnson|21491|
|Williams|18228|
|   Brown|16804|
|   Jones|16023|
|   SMITH|14565|
|  Miller|12942|
|   Davis|12263|
| JOHNSON|12157|
|     Lee|10151|
+--------+-----+
only showing top 10 rows
```

Transformations: Joins

r_DF – is a DataFrame holding movie ratings [userId,movieId,rating,timestamp]
m_DF – is a DataFrame holding movie information [movieId,title,genres]

These DataFrames can be joined as below to obtain number of reviews per genre

```
>>>r_DF.join(m_DF,r_DF.movieId == m_DF.movieId) \
    .groupBy(m_DF.genres) \
    .count() \
    .orderBy("count",ascending =False) \
    .show(5)

+--------------------+-----+
|              genres|count|
+--------------------+-----+
|               Drama| 5832|
|              Comedy| 5648|
|      Comedy|Romance| 3194|
|       Drama|Romance| 2649|
|Comedy|Drama|Romance| 2486|
+--------------------+-----+
only showing top 5 rows
```

Spark SQL and DataFrames

DataFrames and Spark SQL are essentially tied to each other

- The DataFrames API provides a programmatic interface for interacting with data
- Spark SQL provides a SQL-like interface
- Whatever you can do in DataFrames, you can do in Spark SQL and vice versa

Spark SQL contd.

Spark SQL allows you to manipulate distributed data with SQL queries. Currently, two SQL dialects are supported.

- If you're using a Spark SQLContext, the only supported dialect is "sql", a rich subset of SQL 92.
- If you're using a HiveContext, the default dialect is "hiveql", corresponding to Hive's SQL dialect. "sql" is also available, but "hiveql" is a richer dialect.

Spark SQL contd.

- You issue SQL queries through a SQLContext or HiveContext, using the sql() method.
- The sql() method returns a DataFrame.
- You can mix DataFrame methods and SQL queries in the same code.
- To use SQL, you must either:
  - query a persisted Impala or Hive table, or
  - make a table alias for a DataFrame, using registerTempTable()

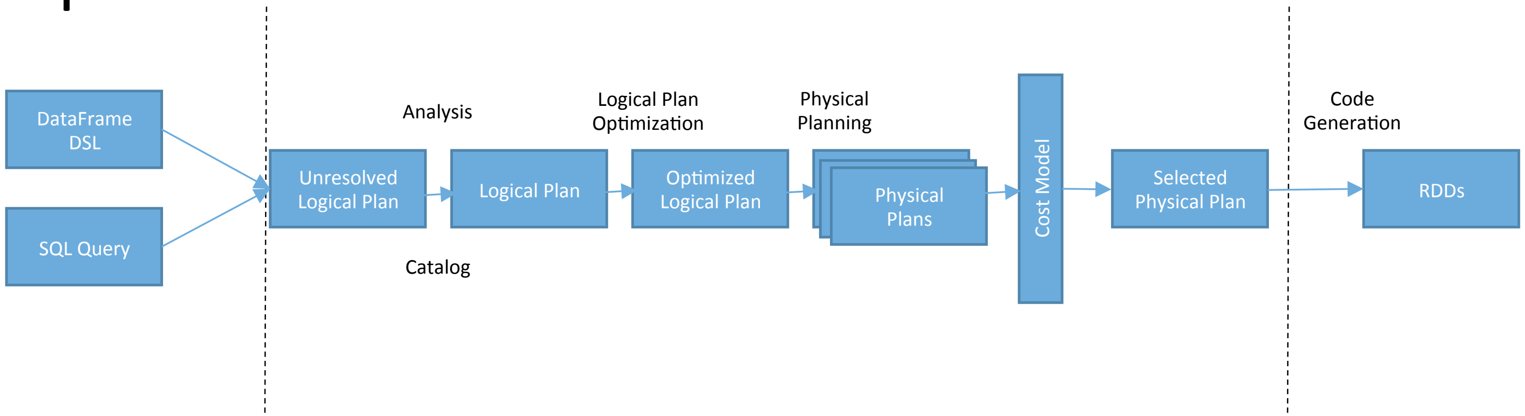
Spark SQL - Example

To issue SQL against an existing DataFrame, create a temporary table, which essentially gives the DataFrame a *name* that's usable within a query.

```
>>> df = sqlContext.read.parquet("/tmp/WH_VRecord/part-r-00000-5396c70a-
ff5b-4dda-9306-3a5e8bd9167a.gz.parquet")

>>> df.registerTempTable("VistorRecords")

>>> sql_df = sqlContext.sql("SELECT NAMELAST,NAMEFIRST,APPT_START_DATE,APPT_END_DATE FROM
VistorRecords")

>>> sql_df.show(5)
+-----------+---------+---------------+-------------+
|   NAMELAST|NAMEFIRST|APPT_START_DATE|APPT_END_DATE|
+-----------+---------+---------------+-------------+
|Adamopoulos|   Stella|         5/1/15|       5/1/15|
|    Brosman|   Muriel|         5/1/15|       5/1/15|
|  Brumfield|    Avery|         5/1/15|       5/1/15|
|    Chipman|Catherine|         5/1/15|       5/1/15|
|      Chubb|   Steven|         5/1/15|       5/1/15|
+-----------+---------+---------------+-------------+
only showing top 5 rows
```

Catalyst : Spark's Optimizer

- Spark SQL uses catalyst to optimize all the queries written both in spark sql and dataframe dsl



Analysis

- phase where attribute references or relations are resolved
- e.g: column validity, column type
- catalog object tracks the tables in all data sources

Catalyst : Spark's Optimizer

Logical Optimizations

- Standard rule-based optimizations
- e.g: predicate pushdown, project prunning, null propagation etc

Physical Planning

- generated one or more physical plans
- cost model is used to select a plan

Code Generation

- generate java bytecode to speed up execution

Further Reading

- https://databricks.com/blog/2015/04/13/deep-dive-into-spark-sqls-catalyst-optimizer.html

Dataframe and SPARK SQL: Demo

Login to lxplus

cd /eos/user/p/pkothuri
git clone https://github.com/prasanthkothuri/sparkTraining.git

Login to SWAN

https://swan.cern.ch/
open sparkTraining->notebooks->Tutorial_DataFrame_Final.ipynb

Spark MLlib

Why Apache Spark for Machine Learning?

- Bigger than memory datasets
  - Able to train a model on large scale dataset
- General Purpose
  - Apart from the libraries for commonly used algorithms, libs for advanced data preparation, feature engineering etc
- Compatibility
  - Support for multiple languages and integrate well with python libs like pandas, scikit-learn etc

Dataframe and RDD based API

DataFrame-based API (spark.ml)

- Easier to construct a machine learning pipeline
- Flexible and versatile API compared to spark.mllib
- Will reach feature parity with spark.mllib in the next releases

RDD-based (spark.mllib)

- Original Machine Learning API
- No new features, only bugfixes
- Will be removed in Spark 3.0

Spark MLlib – Main Concepts

DataFrame

- Same as the Dataframe from Spark SQL / Dataframe API
- Used to hold ML dataset

Transformer

- Transforms one DataFrame into another DataFrame
- Examples
  - Feature transformer appends new column (features) to DataFrame
  - Learning model transforms a DataFrame with features into a DataFrame with predictions

Estimator

- Abstracts the learning algorithm; accepts a DataFrame and produces a Model

Pipeline

- A sequence of transformers and estimators to process and learn from data

Parameter

- Specifying parameters for transformers and estimators

Spark MLlib utilities and algorithms

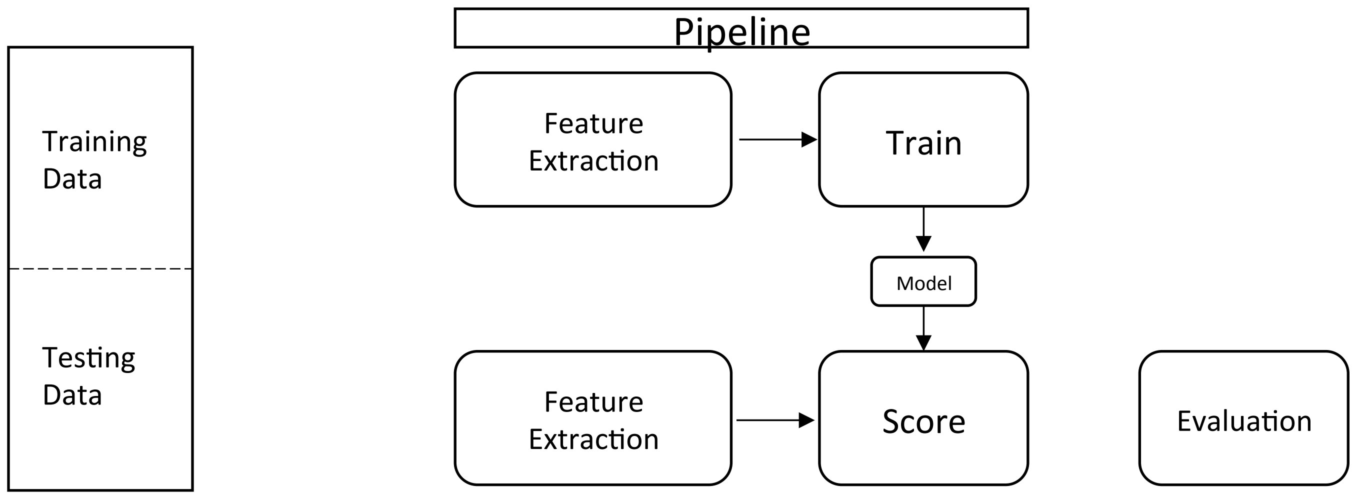
Distributed pre-processing workflow utilities

- Feature Engineering
  - Extraction:- Extracting features from raw data
    - Word2vec
    - countvectorizer
  - Transformations:- modifying, converting or scaling features
    - Tokenizer
    - StringIndexer
    - Standardization
    - Normalization
    - VectorAssembler
  - Selectors:- selecting a subset from a larger set of features

High performant ML algorithms (classification, regression, clustering etc)

- Full list - http://spark.apache.org/mllib/

Model Lifecycle



SPARK MLlib: Demo

Login to lxplus

cd /eos/user/p/pkothuri
git clone https://github.com/prasanthkothuri/sparkTraining.git

Login to SWAN

https://swan.cern.ch/
open sparkTraining->notebooks->Tutorial-ML-Final.ipynb

Conclusion

- We have covered the spark concepts; abstraction and architecture
- Introduced to Spark data APIs – RDD, DataFrame and Spark SQL
- Demonstration of using Spark data APIs for exploratory data analysis and data analytics
- Introduced to Spark Mllib for scalable machine learning
- Understood how spark can aid in distributed computing of VERY large datasets
- Several ways to interact with spark – spark-shell, pyspark, spark-submit and jupyter notebooks

Further study

- Slides and Notebooks for Hands On
  - https://github.com/prasanthkothuri/sparkTraining

- 2016 IT DB Hadoop tutorials
  - https://github.com/prasanthkothuri/hadoop-tutorials-2016

- Coursera course – Big Data Analysis with Scala and Spark
  - https://www.coursera.org/learn/scala-spark-big-data

- CERN Technical Training on Apache Spark in June