Data 603 – Big Data Platforms

UMBC

Lecture 4

Introduction to Apache Spark

Spark Resilient Distributed Dataset (RDD)

Today’s Lecture Outline

- Homework Review
- Project and Final Paper Rubrics
- Apache Spark Introduction
- Spark SQL
- Resilient Distributed Dataset (RDD)
- RDD Transformations
- RDD Actions
- RDD Lab

Homework Review

Research Paper Grading Rubric

Integration of Knowledge (30%)

- Ability to demonstrate the understanding of the research subject
- Ability to integrate the paper concepts into the subjects covered in the class
- Keep it focused. I am not looking for a survey of big data tools

Depth of Discussion (30%)

- Research != Googling/Binging/Wikipedia
- Formulation of original thoughts
- Demonstration of understanding: Which big data issue are you addressing? What is the technical solution?
- Code examples are a good way of diving deep

Sources (30%)

- 5 or more current (< 5 years old) sources

Spelling and Grammar (10%)

Technical Paper Schedule

#	Date	Activity	Expected Outcome
1	09/01/21	Present the technical research paper assignment to students	Start thinking about proposals for the paper
4	09/22/21	Technical paper proposal ready for defense	Every student will submit his paper proposal
9	10/27/21	Present near complete paper and share progress	Every student will prepare and submit a paper progress report
13	11/24/21	Deliver Final paper	Final paper deliver (due 11:59PM)

Final Project Grading Rubric

Originality (20%)

- Bring new ideas.

Project Execution (30%)

- Big data techniques covered in the class
- Code execution (20%)
- Performance Optimization (20%)

Presentation (30%)

- Use of visualization & story telling (20%)
- What are interesting insights you gained? (20%)
- Don’t read the code
- Sell your story: You are pitching to potential investors and your executive board

Data Research (20%)

Project Schedule

#	Date	Activity	Expected Outcome
1	9/01/2021	Present the project assignment to students	Start thinking about big data project
5	9/29/2021	Project idea ready	Prepare a slide deck for presenting the project idea
10	11/03/2021	Present project progress report	Every student will prepare and submit a project progress status report
14 15	12/01/2021 12/08/2021	Project presentations	Prepare a slide deck for presenting your project to students
15	12/08/2021	Project report	Final slide deck and 1-page summary due

Apache Spark Introduction

What is Apache Spark?

- Unified engine for large-scale distributed data processing
  - https://spark.apache.org/
- In-memory storage for intermediate computations
- Incorporates libraries with composable APIs
  - Machine learning (MLlib)
  - SQL for interactive queries (SparkSQL)
  - Structured Streaming
  - GraphX for graph processing

Spark's Design Philosophy

- Speed
  - Use of commodity servers with multi-core CPUs, large memory, efficient multithreading and parallel processing
  - Query computations as a directed acyclic graph (DAG)
    - DAG scheduler and query optimizer construct an efficient computational graph
    - Graph is decomposed into **tasks executed in parallel** across workers on the cluster.
  - **Tungsten**, Spark's physical execution engine, uses whole-stage code generation to **generate compact code for execution**.

Spark's Design Philosophy

- Ease of Use
  - Fundamental abstraction of a simple logical data structure called a **Resilient Distributed Dataset (RDD)**
    - Higher-level structured data abstractions DataFrames and Datasets are constructed.
  - Set of transformations and actions as operations
    - Simple programming model for building big data applications in familiar languages.

Spark's Design Philosophy

- Modularity
  - Polyglot
    - Support many languages: Scala, Java, Python, SQL, and R
  - Application that can do it all!
    - Core components:
      - Spark SQL
      - Spark Structured Streaming
      - Spark Mllib
      - GraphX

Spark's Design Philosophy

- Extensibility
  - Decoupling of compute and storage
    - Different from Apache Hadoop
  - Connectors to read/write
    - https://docs.databricks.com/data/data-sources/index.html
  - Spark Packages
    - https://spark-packages.org/

Optimization with Tungsten

Apache Spark 2.0 shipped with the second generation Tungsten engine.

Built upon ideas from modern compilers and MPP databases and applied to data processing queries, Tungsten emits (SPARK-12795) **optimized bytecode at runtime that collapses the entire query into a single function**, eliminating virtual function calls and leveraging CPU registers for intermediate data. As a result of this streamlined strategy, called "whole-stage code generation," CPU efficiency is significantly improved.

- https://databricks.com/blog/2016/05/23/apache-spark-as-a-compiler-joining-a-billion-rows-per-second-on-a-laptop.html
- https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html

Spark SQL

- Integrates relational processing (tabular data abstraction) with Spark's functional programming API.
- Spark module for structured data (records with a known schema) processing
  - Offers tighter integration between relational and procedural processing through declarative DataFrame API that integrates with procedural Spark code
  - Includes Catalyst, a highly extensible optimizer built using features of the Scala programming.

Spark MLlib (1/2)

- Spark's machine learning (ML) library
  - ML Algorithms: classification, regression, clustering, and collaborative filtering
  - Featurization: feature extraction, transformation, dimensionality reduction, and selection
  - Pipelines: tools for constructing, evaluating, and tuning ML Pipelines
  - Persistence: saving and load algorithms, models, and Pipelines
  - Utilities: linear algebra, statistics, data handling, etc.

Spark MLlib (2/2)

- spark.mllib package
  - RDD-based APIs
  - Maintenance mode since 2.0
- spark.ml package (>= 1.6)
  - DataFrame-based API
  - More friendly API with benefits of Tungsten and Catalyst optimization

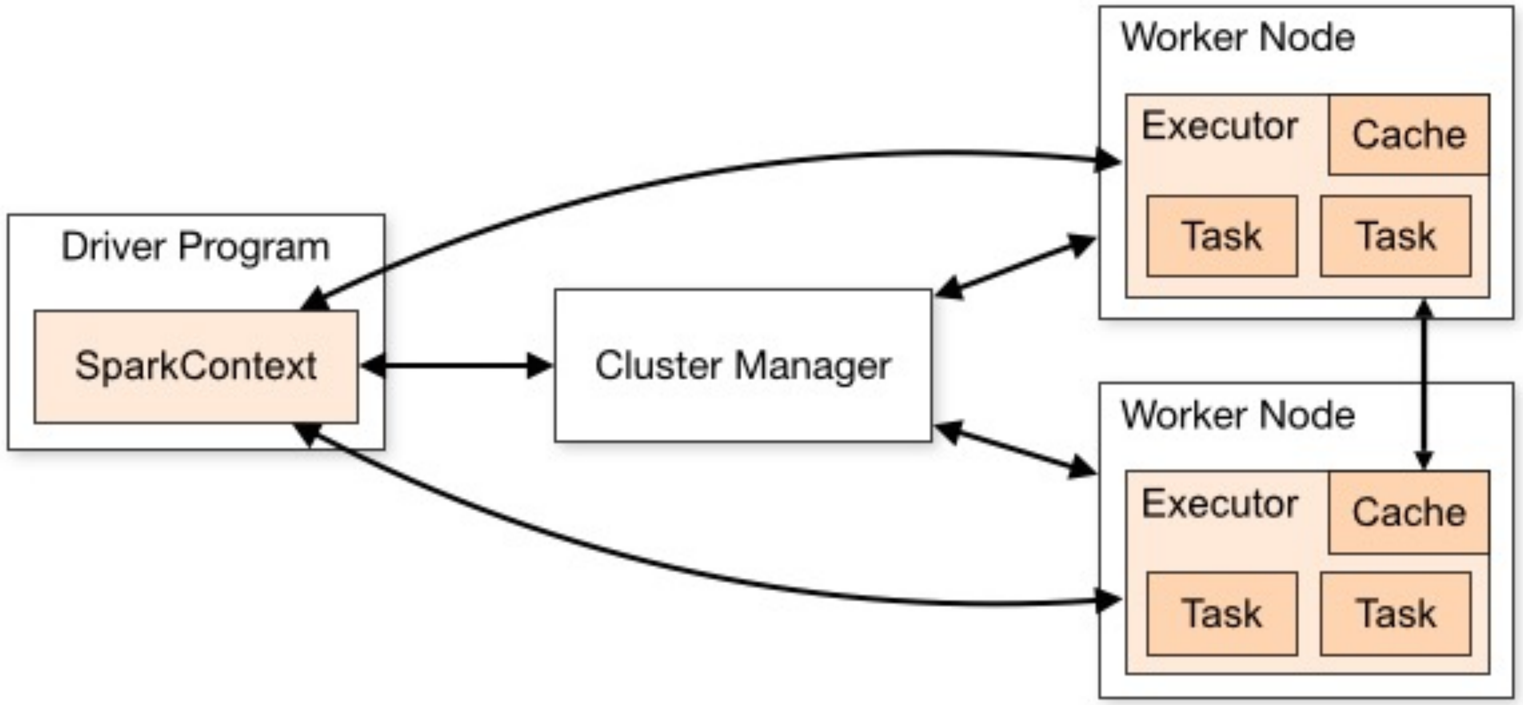
Structured Streaming

- Scalable and fault-tolerant stream processing engine built on the Spark SQL engine.
- Dataset/DataFrame based API in Scala, Java, Python or R
- Supports Streaming aggregations, event-time windows, stream-to-batch joins

GraphX

- Library for manipulating graphs (e.g. social network, routes and connected points, or network topology graphs)
- Graph-parallel computations
- Provides standard graph algorithms for analysis, connections, and traversals

Spark Architecture



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

Spark Application Concepts

- Spark Application
  - Consist of a driver process and a set of executor processes
- Spark Executor
  - Runs on each work node in the cluster
  - Responsible for carrying out the work assigned by the driver
- Cluster Manager
  - Responsible for managing and allocating resources for the cluster nodes
    - Currently supported managers: standalone, YARN, Mesos, Kubernetes.

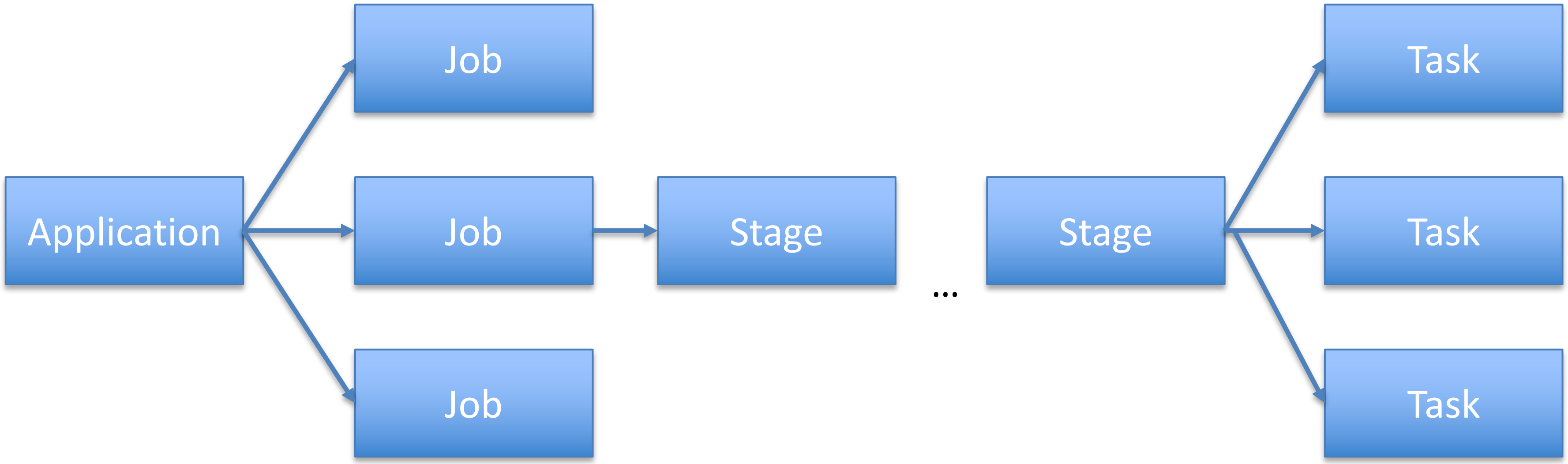
Spark Application Concepts

- Spark Driver
  - Runs main() function
  - Responsible for:
    - Instantiating SparkSession
    - Requesting resources (CPU, memory) from the cluster manager for executors
    - Transforming Spark application into one or more Spark Jobs.
      - Each job is transformed into a DAG computations
    - Scheduling and distributing work across the executors

Spark Job, Stages, Tasks

- Spark driver
  - Transforms Spark application into one or more Spark Jobs.
  - Each job is transformed into a DAG computations
  - Stages make up DAG nodes and are comprised of tasks (units of execution)
    - They are federated across each Spark executor
    - Each task map to a single core and works on a single partition of data => parallel execution

Spark Job, Stages, Tasks



https://spark.apache.org/docs/latest/job-scheduling.html#scheduling-within-an-application

Spark Application Concepts

- SparkSession
  - Unified conduit to all Spark operations and data
    - Subsuming of all previous entry points to Spark including SparkContext, SQLContext, HiveContext, SparkConf, and StreamingContext
  - Single unified entry point to all of Spark's functionality

Glossary

Term	Meaning
Application	User program built on Spark. Consists of a *driver program* and *executors* on the cluster.
Application jar	A jar containing the user's Spark application. In some cases users will want to create an "uber jar" containing their application along with its dependencies. The user's jar should never include Hadoop or Spark libraries, however, these will be added at runtime.
Driver program	The process running the main() function of the application and creating the SparkContext
Cluster manager	An external service for acquiring resources on the cluster (e.g. standalone manager, Mesos, YARN)
Deploy mode	Distinguishes where the driver process runs. In "cluster" mode, the framework launches the driver inside of the cluster. In "client" mode, the submitter launches the driver outside of the cluster.
Worker node	Any node that can run application code in the cluster
Executor	A process launched for an application on a worker node, that runs tasks and keeps data in memory or disk storage across them. Each application has its own executors.
Task	A unit of work that will be sent to one executor
Job	A parallel computation consisting of multiple tasks that gets spawned in response to a Spark action (e.g. save, collect); you'll see this term used in the driver's logs.
Stage	Each job gets divided into smaller sets of tasks called *stages* that depend on each other (similar to the map and reduce stages in MapReduce); you'll see this term used in the driver's logs.

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

Transformations and Actions

- Spark operations are either transformations or actions.
- Transformation
  - Transform a dataset into a new dataset without altering the original data (immutable).
  - Will not change the original dataset. A new dataset is returned.
  - All transformations are evaluated lazily
    - Results are not computed immediately, they are recorded as a lineage.
- Actions
  - Trigger the lazy evaluation of all recorded transformation.

Transformations and Actions

- Lazy evaluation allows Spark to optimize queries
  - Allow Spark to rearrange transformations, coalesce them, and optimize them into stages for more efficient executions.
- Lineage and data immutability provide fault tolerance
  - Spark records each transformation in its lineage, DataFrames (and RDDs) are immutable between transformations
  - Original state can be reproduced by replaying the recorded lineage.

Resilient Distributed Dataset (RDD)

Spark RDD

- Resilient Distributed Dataset (RDD)
  - Immutable, partitioned collection of records which can be operated on in parallel with the inbuilt capability of reliability and automated failure recovery.
  - Unlike DataFrames (structured with schema), RDD records are objects of the programming language used (Java, Scala, Python).
  - Cannot be optimized by Spark. Spark does not understand the inner structure of the object.
    - Optimizations need to be done by hand.
  - Low-level API, primary API in the Spark 1.x.

Spark RDD

Two types of RDD operations:

- *Transformations*: Create a new dataset from an existing one … e.g. map()
- *Actions*: Return a value to the driver program after running a computation on the dataset … e.g. reduce()

All transformations in Spark are *lazy*

- They do not compute their results right away.
- They just remember the transformations applied to some base dataset (e.g. a file).
- The transformations are only computed when an action requires a result to be returned to the driver program.
- By default, each transformed RDD may be recomputed each time you run an action on it.
- This design enables Spark to run more efficiently.
- RDDs maybe persisted in memory using the persist (or cache) method

Spark RDD

Two ways to create RDDs:

- Parallelizing an existing collection in the driver program
- Referencing a dataset in an external storage system, such as a shared filesystem, HDFS, HBase, or any data source offering a Hadoop InputFormat.

Sneak Preview: RDD vs. DataFrame vs. Datasets

https://databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html

Printing elements of an RDD

Attempt to print out the elements of an RDD using *rdd.foreach(println)* …

- On a single machine, this will generate the expected output and print all the RDD's elements

But …

- In cluster mode, the output to *stdout* being called by the executors is now writing to the executor's *stdout* instead. *stdout* on the driver won't show these!

So …

- To print all elements on the driver, one can use the collect() method to first bring the RDD to the driver node thus: *rdd.collect().foreach(println)*.

However …

- This can cause the driver to run out of memory, though, because *collect()* fetches the entire RDD to a single machine;

A safer approach is to use the *take()*: *rdd.take(100).foreach(println)*.

RDD Transformations

Single RDD Transformation

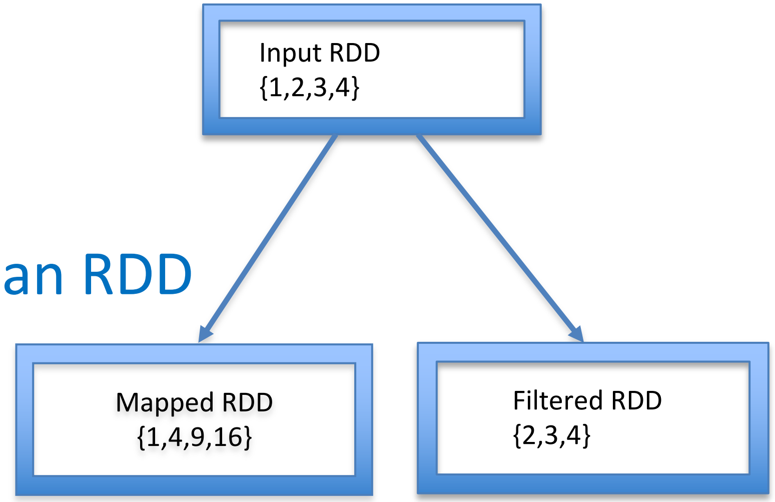
- Example – Basic RDD transformation:
    - RDD contains {1, 2, 3, 3}

Function Name	Example	Result
map	rdd.map(x => x + 1)	{2, 3, 4, 4}
flatMap*	rdd.flatMap(x => x.to(3))	{1, 2, 3, 2, 3, 3, 3}
filter	rdd.filter(x => x != 1)	{2, 3, 3}
distinct	rdd.distinct()	{1, 2, 3}
sample(withReplacement, fraction,[seed])	rdd.sample(false, 0.5)	non-deterministic (but containing ½ the elements of RDD)

* flatMap transforms an RDD of length *n* into a collection of *n* collections, then flattens these into a single RDD of results.

Element-wise transformations

- Two most common transformations:
  - Map()
    - Takes a function and applies it to each element in the RDD
    - Example: Map x=>x*x
  - Filter()
    - Takes in function and returns an RDD
    - Example: Filter x=>x!=1



Element-wise transformations Example [1]

- **Example :** {squaring values}

```
Nums = sc.parallelize([1,2,3,4])
Squared = nums.map(lambda x: x*x).collect()
For num in squared:
    Print "%i " %(num)
```

Element-wise transformations Example [2]

- **Example:** {String Splitting}
  - *Python flatMap example, splitting lines into words:*

```
lines = sc.parallelize(["hello world", "hi"])
words = lines.flatMap(lambda line: line.split(" "))
words.first()      # returns "hello”
```

Quick Overview
The Lambda (λ) Function

- In Python:
  - A lambda function is a small anonymous function
  - It takes any number of arguments, but can only have one expression
    - Lambda arguments : expression
    - Example:

```
x = lambda a : a + 10
print(x(5))
```

Multiple RDDs Transformation

- Example: Work on two RDDs
  - rdd1 contains {1, 2, 3}
  - rdd2 contains {3, 4, 5}

Function Name	Example	Result
union	rdd1.union(rdd2)	{1, 2, 3, 3, 4, 5}
intersection	rdd1.intersection(rdd2)	{3}
subtract	rdd1.subtract(rdd2)	{1, 2}
cartesian	rdd1.cartesian(rdd2)	{(1, 3), (1, 4), ... (3,5)}

Transformations on Pair RDDs

- Transformations on one pair RDD (example: {(1, 2), (3, 4), (3, 6)}):

Function name	Purpose	Example	Result
reduceByKey(func)	Combine values with the same key.	rdd.reduceByKey ((x, y) => x + y)	{(1, 2), (3, 10)}
groupByKey()	Group values with the same key.	rdd.groupByKey()	{(1, [2]), (3, [4, 6])}
combineByKey (createCombiner, mergeValue, mergeCombiners, partitioner)	Combine values with the same key using a different result type.		
mapValues(func)	Apply a function to each value of a pair RDD without changing the key.	rdd.mapValues (x=> x+1)	{(1, 3), (3, 5), (3, 7)}
flatMapValues(func)	Apply a function that returns an iterator to each value of a pair RDD, and for each element returned, produce a key/value entry with the old key. Often used for tokenization.	rdd.flatMapValues (x => (x to 5))	{(1, 2), (1, 3), (1, 4), (1,5), (3, 4), (3, 5)}
keys()	Return an RDD of just the keys.	rdd.keys()	{1, 3, 3}
values()	Return an RDD of just the values.	rdd.values()	{2, 4, 6}
sortByKey()	Return an RDD sorted by the key.	rdd.sortByKey()	{(1, 2), (3, 4), (3, 6)}

Transformations on two Pair RDDs

- Transformations on two pair RDDs
  (rdd = {(1, 2), (3, 4), (3, 6)}   other = {(3, 9)})

Function name	Purpose	Example	Result
subtractByKey	Remove elements with a key present in the other RDD.	rdd.subtractByKey (other)	{(1, 2)}
join	Perform an inner join between two RDDs.	rdd.join(other)	{(3, (4, 9)), (3, (6, 9))}
rightOuterJoin	Perform a join between two RDDs where the key must be present in the first RDD.	rdd.rightOuterJoin (other)	{(3,(4,9)), (3,(6,9))}
leftOuterJoin	Perform a join between two RDDs where the key must be present in the other RDD.	rdd.leftOuterJoin (other)	{(1,(2,None)), (3,(4,9)), (3,(6,9))}
cogroup	Group data from both RDDs sharing the same key.	rdd.cogroup(other)	{(1,([2],[])), (3,([4, 6],[9]))}

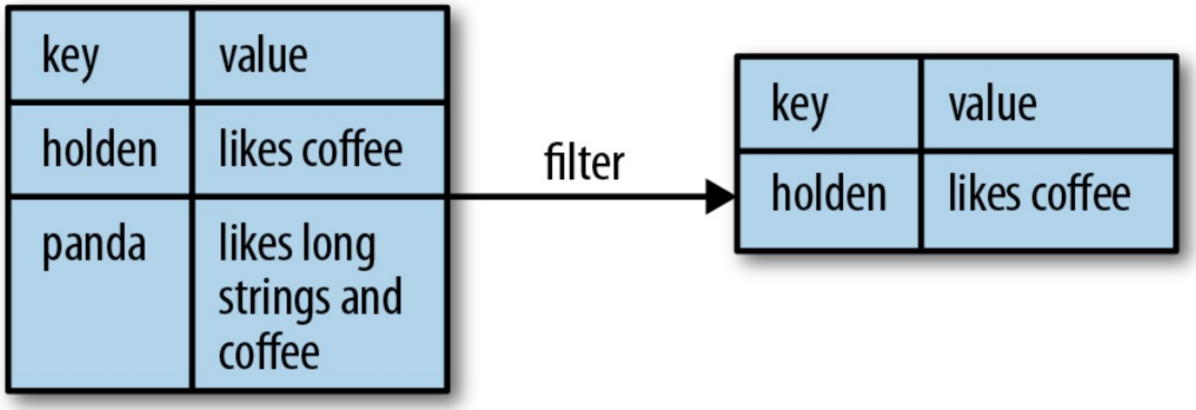
Creating Pair (Key/Value) RDDs

- Done by running a map() function that returns key/value pairs
  - Example:

```
pairs = lines.map(lambda x: x.split(" "))
```

- **Hint:** When creating a pair RDD from an in-memory collection in Python, we only need to call SparkContext.parallelize() on a collection of pairs

Filter on Value



Example:

result = pairs.filter(lambda keyValue: len(keyValue[1]) < 20)

{Remove the key/value pairs where length of string is longer than or equal 20}

RDDs Join Example

- //Data Example – Scala code shown
- rdd1 = sc.parallelize(Seq(("math", 55), ("math", 56), ("English", 57), ("English", 58), ("science", 59), ("science", 54)))
- rdd2 = sc.parallelize(Seq(("math", 60), ("math", 65), ("science", 61), ("science", 62), ("history", 63), ("history", 64)))
- //join() Example
- joined = rdd1.join(rdd2)
- joined.collect()
- Result:
  - Array[(String, (Int, Int))] = Array((math,(55,60)), (math,(55,65)), (math,(56,60)), (math,(56,65)), (science,(59,61)), (science,(59,62)), (science,(54,61)), (science,(54,62)))

Another simple example, consider an application that keeps a large table of user information in memory—say, an RDD of (UserID, UserInfo) pairs, where UserInfo contains a list of topics the user is subscribed to. The application periodically combines this table with a smaller file representing events that happened in the past five minutes—say, a table of (UserID, LinkInfo)pairs for users who have clicked a link on a website in those five minutes. For example, we may wish to count how many users visited a link that was not to one of their subscribed topics. We can perform this combination with Spark's join() operation, which can be used to group the UserInfo and LinkInfo pairs for each UserID by key

RDD Actions

Sorting Data

- Custom sort order :
  - Example:

    ```
    rdd.sortByKey (ascending=True, numPartitions=None,
    keyfunc = lambda x: str(x))
    ```

RDDs Actions

- Basic actions on an RDD containing {1,2,3,3}

Function Name	Example	Result
collect()	rdd.collect()	{1,2,3,3}
count()	rdd.count()	4
take(num)	rdd.take(2)	{1,2}
top(num)	rdd.top(2)	{3,3}
takeOrdered(num)(ordering)	rdd.takeOrdered(2)(myOrdering)	{3,3}
takeSample(withReplacement, num, [seed])	rdd.takeSample(false,1)	non-deterministic
reduce(func)	rdd.reduce((x,y)=>x+y)	9
fold(zero)(func)	rdd.fold(0)((x,y)=>x+y)	9
aggregate(zeroValue)(seqOp, combOp)	rdd.aggregate(0,0)({case(x,y)=>(y._1()+x., y._2() + 1)}, {case(x,y) => (y._1()+x._1(), y._2()+x._2())})	(9,4)
foreach(func)	rdd.foreach(func)	nothing

Pair RDDs Actions

- Actions on pair RDDs – example ({(1, 2), (3, 4), (3, 6)}):

Function	Description	Example	Result
countByKey()	Count the number of elements for each key	rdd.countByKey()	{(1, 1), (3, 2)}
collectAsMap()	Collect the result as a map to provide easy lookup	rdd.collectAsMap()	Map {(1, 2), (3,4), (3, 6)}
lookup(key)	Return all values associated with the provided key	rdd.lookup(3)	[4, 6]

RDD Lab

More Reading

- Spark SQL: Relational Data Processing in Spark: http://people.csail.mit.edu/matei/papers/2015/sigmod_spark_sql.pdf
- Apache Spark as a Compiler: Joining a Billion Rows per Second on a Laptop: https://databricks.com/blog/2016/05/23/apache-spark-as-a-compiler-joining-a-billion-rows-per-second-on-a-laptop.html
- Project Tungsten: Bringing Apache Spark Closer to Bare Metal: https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html
- Partitioning:
  - https://medium.com/parrot-prediction/partitioning-in-apache-spark-8134ad840b0
  - https://techmagie.wordpress.com/2015/12/19/understanding-spark-partitioning/
  - http://ethen8181.github.io/machine-learning/big_data/spark_partitions.html

Questions

Homework

- No RDD homework
- Get Spark+PySpark installed on your computer
  - Optional: use Docker or get familiar with PySpark on Google Colab (instructions to be sent out)
- Finish research paper proposal and project proposals this week– there will be a homework next week