SPARK Architecture

* Spark is a fast, distributed and general purpose cluster computing system

* Spark offers - lazy computations - optimize job before execution
  - In memory data caching
  - Efficient pipelining

* Spark Applications run as independent sets of processes on a cluster, coordinated by the SparkContext object of our main program (also called driver program)

TWO Main Abstractions in Spark

(1) Resilent Distributed Datasets (RDD)

* collection of data items split into partitions & stored in the memory of worker nodes of the cluster.
* its the interface for doing data transformations.
* RDDs can contain data from HDFS, HBase, Cassandra or in cache (memory, mem + disk) or another RDD.
* RDDs automatically recover from node failures.
* Partitions are recomputed on failure or cache eviction

* Meta data stored in RDDs consists of
  - Partitions - sets of data splits associated with this RDD
  - Dependencies - list of parent RDDs involved in the computation
  - Compute - function to compute partition of the RDD given parent partitions from the dependencies
  - Preferred Locations - where is the best place to put computations on this partition (data locality)

RDD Operations

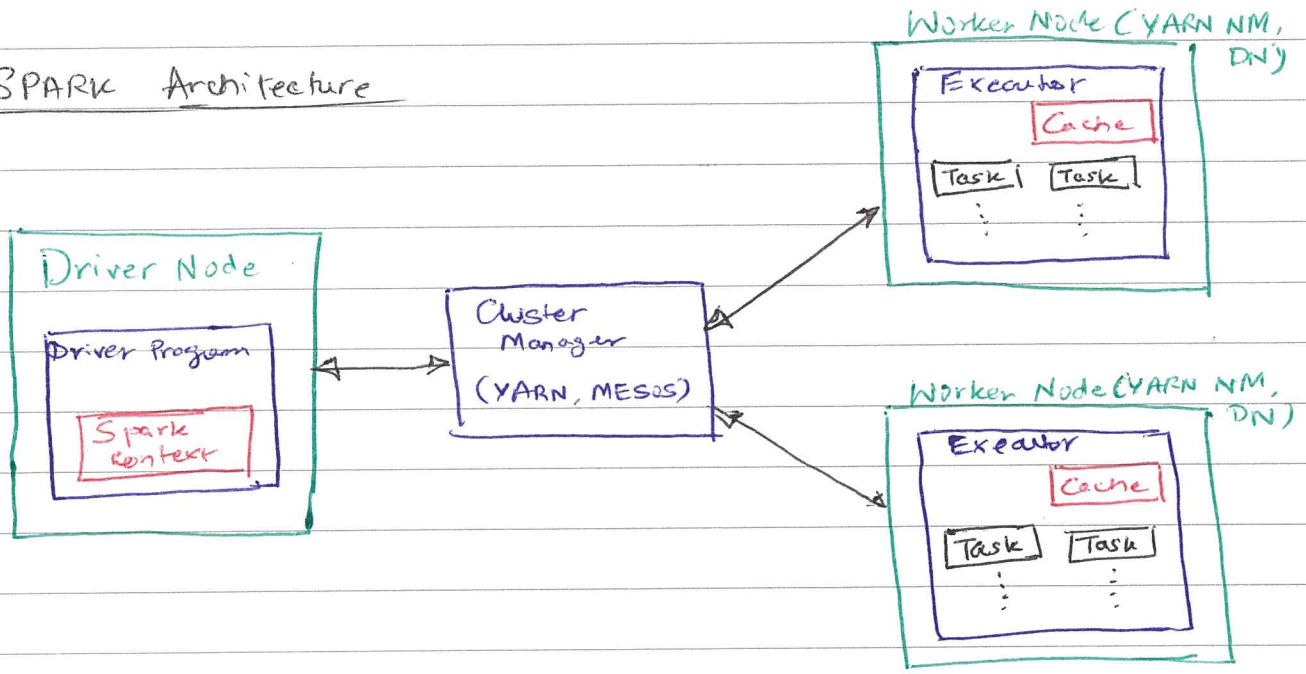
(1) Transformations - do not evaluate immediately (lazy evaluation)
- return new (transformed) RDD
- only cause change in metadata.

(2) Actions - trigger immediate execution of related previous transformation
- return result to the driver program.

(2) Direct Acyclic Graph (DAG)

* Sequence of computations performed on data.
* Node → RDD partition
* Edge → transformation
* Acyclic → graph cannot return to the older partition
* Direct → transformation is a transition from A to B for a data partition

SPARK Architecture



* Driver

- Entry point to the Spark shell (Scala, Python, R)
- Is the class that has the main function.
- Place where SparkContext is created
- Translates RDD into the execution graph.
- Schedule tasks & controls their execution. Due to this reason the driver should run close to worker nodes.
- Stores metadata about all RDD & partitions.

* SparkContext

- Main entry for spark functionality.
- A SparkContext represents the connection to Spark cluster & is used to create RDD, broadcast variables & accumulators.
- Only one SparkContext may be active per JVM.

* Executor

- Each Spark application will get its own executor process which stays up for the duration of the application and runs tasks in multiple threads.
  This results in isolation on both scheduling side (driver schedules its own tasks) and executor side (task from different application run in different JVMs)

* Stores data in cache in JVM heap or Hard disk
* Read data from external sources
* Writes data to external sources
* Performs all the data processing.

Application :

* Single instances of Spark context that stores some data processing logic and can schedule series of jobs, sequentially or in parallel

* Job is a complete set of transformations on RDD that finishes with an Action or data savings, triggered by the driver application

* Stage - set of transformations that can be pipelined and executed by single independent worker. Usually it is the transformations between "read", "shuffle", "action" "save".

* Task - execution of the stage on a single data partition.

* Spark Application is agnostic of the cluster manager. As long as it can acquire executor processes and these processes can communicate with each other it can run on a cluster which support other non-spark applications

RDD Operations

Transformations
- return RDD
- lazy evaluation

Actions
- return result to driver
- immediately cause the previous op (transformations) to be executed.

RDD Persistance:
- By default each transformed RDD will be evaluated everytime an action gets subsequently executed.
- We can persists an RDD in memory using persist() or cache() method in which case spark will keep the RDDs in memory around the cluster for much faster access the next time perform operations on this RDD.
- Caching is a key tool for iterative algorithms
- The cache() method allows only default storage level (memory_only)
- The persist() method allows to specify storage level

MEMORY_ONLY - Stored as serialized Java objects in JVM

MEMORY_AND_DISK - serialized Java objects in JVM. If an RDD does not fit in available memory store partitions that don't fit onto the disk.

MEMORY_ONLY_SER - stored as serialized Java objects such as array of byte (one byte array per partition)

MEMORY_AND_DISK_SER - Spill partitions that don't fit onto disk.

DISK_ONLY - only stored on Disk (not in mem)

MEMORY_ONLY_2

MEMORY_AND_DISK_2 - keep 2 copies of each partition across nodes.

SPARK PROGRAM EXECUTION (SCALA)

Command Line

> Spark-shell

parameters

--master	MASTE_VRL	Values ~~"client" or~~ "local" "yarn", spark://host:port
--class	CLASS_NAME	Application main class
--name	"Name"	Application name
--jars	JARS	CSV list of local jars to include in driver & executor class path.
--py-files	PYFILES	csv list of zip, egg or py files to place on PYTHONPATH for python apps
--files	FILES	CSV list of files to be placed in working directories of each executor
--conf	PROP=VALUE	Arbitrary Spark-config property
--properties-file	FILE	Path to file from which to load extra properties. By default uses /etc/conf/spark-default.conf

YARN only parameters

--archives	ARCHIVES	CSV list of archives to be extracted into working directories of each executor

Spark-shell in Interactive mode

- we don't specify --class & jar file.

We can type in scala statements for spark or in general on the scala prompt.

~~Spark-shell is~~ Batch mode

There are two choices in Batch mode.

1. Spark-shell
2. SBT (Scala build tool)

→ Common Steps

✳ ~~Create~~ Directory Structure

```
./
./simple.sbt
./src
./src/main/scala/Yourprogram.scala
```

simple.sbt

```
name := "Project Name"
version := "1.0"
scalaVersion := "2.10.4"
libraryDependencies += "org.apache.spark" %% "spark-core" % "1.2.1"
```

YourProgram.scala

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf
import org.apache.spark.rdd._
object YourProgram {

  def main (args : Array[String]) {
    Your Statement ... : val conf = new SparkConf()
                           .setAppName("...")
    val sc = new SparkContext(conf)
                           .setMaster("local")
  }
}
```

// Package the program into Jar
go to the folder that contains simple.sbt (*.sbt)

> ~~sbtpa~~ sbt package. - This will create a target folder

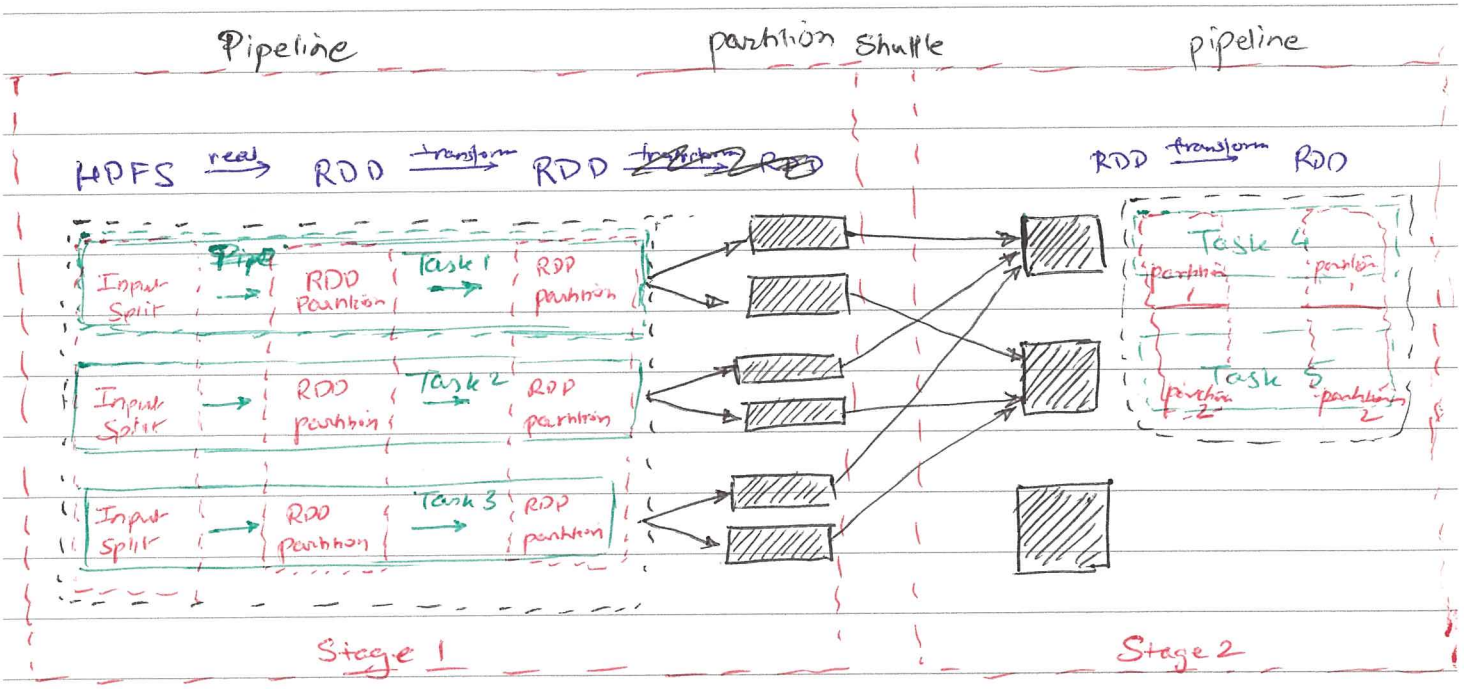
Run applications using Spark-submit

> spark-submit --class "Your main class"
--master "local[4]"
target/scala/*.jar
{~~your~~ location of jar)

Running application using Spark-shell

> spark-shell --class "Your main class"
--jars "path to jar"

~~Sh~~ S&B (
Developer → Architect (ACS → NS
Canada (Lianne)
US (Lianne)
Mobile solution (@reenlow)
Lies
New Contact (Improved with new Designation & salary revision)



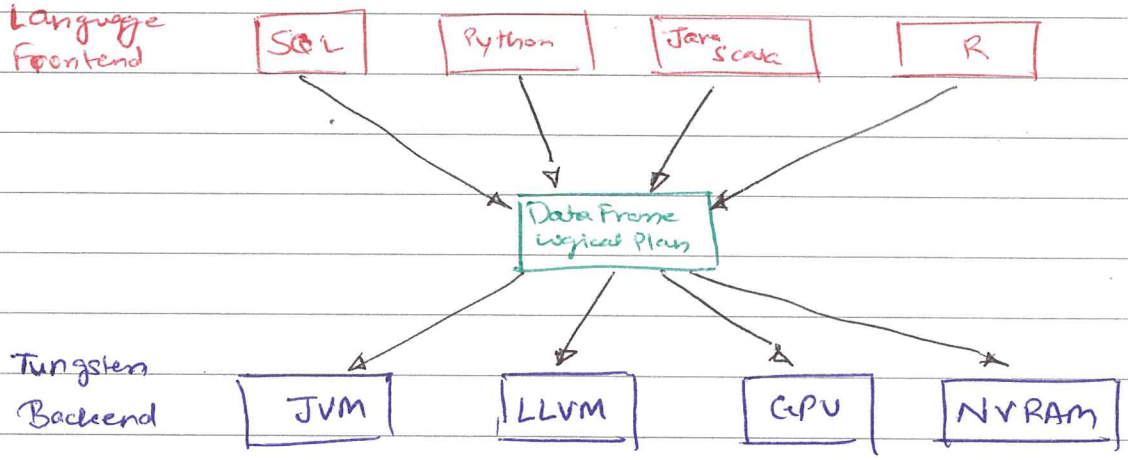
RDD persistence

* By default, each transformed RDD will be evaluated everytime an action gets subsequently executed.
* We can persist an RDD in memory using cache() or persist() method.
* Caching is key tool for iterative algorithms
* The cache() method only supports in-memory persistance.

1. MEMORY_ONLY - deserialized Java objects in JVM. if the RDD does not fit in mem, some partitions will not be cached & will be recomputed on the fly. This default.
2. MEMORY_AND_DISK - if RDD does not fit in memory, store partitions that don't fit on the disk & read back when needed
3. MEMORY_ONLY_SER - store as "serialized Java objects" (one byte array per partition)
4. MEMORY_AND_DISK_SER - same as above but uses disk
5. DISK_ONLY
6. MEMORY_ONLY_2 } Same as top and 2 but replicate each
7. MEMORY_AND_DISK_2 } partition on two cluster nodes

* Spark considers memory as cache with LRU eviction rules
* if disk is involved, data is evicted to disk.

SPARK Dataframes :



* Data frame is an RDD with schema - field names, field data types and statistics
* Unified transformation interface in all languages
* Can be accessed as RDD, in this case transformed to RDDs of Row objects.
* Data frames allow better compression ratio than standard RDD.
* Each column in each partition stores min - max values for partition pruning.

Shared Variables

* can be used for parallel operations
* By default when Spark function runs in parallel across nodes it ships a copy of each variable used in the function to each task.
* Sometimes however we need to share variables across tasks. hence we use shared variables
* Two Types

(1) Broadcast Variables

* Used to cache a value in memory of all nodes
* created by calling broadcast function on SparkContext

```
val bcVar = sc.broadcast (v);
```

* After creation it should be used instead of original variable 'v'

(2) Accumulator :

* variables that are only added such as counters or sums.
* Created with initial value (v)

```
val acc = sc.accumulator (v);
```

* Tasks running on cluster can ~~added~~ call add() or += operator on the acc.

SPARK API (Scala)

Important Namespaces

* org.apache.spark.{SparkContext, SparkConf}
* org.apache.spark.rdd._ (RDDs)
* org.apache.io._ (Compression)
* org.apache.spark.sql._ (SQLContext)
* org.apache.spark.sql.hive._ (HiveContext)

Important classes & their methods

SparkConf - configuration of Spark application
- created using new SparkConf()

getAll(): gets all config settings as K, V

get(String): String - gets the value of specified config property

getDouble(s), getLong(c) - typed methods for config properties

setAppName(String) - set name of our Spark Application

setJars(Array[String]) - set jar files to distribute to cluster

setMaster(String) - master URL to connect to

ex: "local" to run locally in one thread

"local[4]" to run locally in 4 cores

"yarn" to run on YARN cluster

SparkContext :

* Main entry point for Spark functionality.
* represents the connection to Spark cluster
* used to create RDD, Acc, & broadcast var.
* only one spark context per JVM.
* created using new SparkContext (appName, SparkConf)

* addFile(path) - add file to be downloaded with this Spark Job on every node

* addJar(path) - adds a dependency JAR for all tasks to be executed on this context in the future

* appName - get App name
* broadcast(obj) - create broadcast variables
* accumulator(obj) - Creati accumulator.
* getConf() - returns configuration (SparkConf) associated with this context.

* newAPIHadoopFile(path, class[InputFormat], class[K], class[V])

ex: import org.apache.~~spark~~hadoop.mapreduce.lib.input._

```
val depseq = SC.newAPIHadoopFile[IntWritable, Text, SequenceFileInputFormat]
              (outputpath,
               classOf[SequenceFileInputFormat[IntWritable, Text]],
               classOf[IntWritable],
               classOf[Text]);
```

* sequenceFile(path, class[K], class[V]) - reads sequence file

eg: val depseq = SC.SequenceFile[Int, String]
("/user/cloudera/dataFile/dep.seq");

* textFile (path) - reads textfile from HDFS/LFS

* wholeTextFiles (path) - reads directory of text files from HDFS/LFS

returns RDD [String, String]

filepath, file content

Important hadoop namespaces

org.apache.hadoop.mapreduce.lib.input.xxx InputFormat

org.apache.hadoop.mapreduce.lib.output.xxx OutputFormat

org.apache.hadoop.io.{all Writable classes & interfaces}

org.apache.hadoop.io.compression {all compression codecs}

RDD Operations

cache() - persists an RDD in with default storage level (MEM_ONLY)

coalesce(n) - returns new RDD with specified number of partitions. Usually used to reduce number of partitions after filter operation.

collect(f), collect() - returns new RDD by applying specified function
↳ returns Array RDD containing all RDD elements

context - returns SparkContext that this RDD is created on

Count() - number of elements in this RDD.

countByValue() (Ordering = null)
- returns count of each unique value in this RDD as a map.
- For single fields if a unique value
- For tuples its every unique combination of fields.

distinct(n) : n - num of partitions
- return RDD containing distinct elements

filter(f: (T) ⇒ Boolean) : returns RDD containing elements that satisfy the specified predicate

first() - return first element

flatMap[U] (f : (T) ⇒ U)

- Returns an RDD by first applying the function to all elements of the RDD & then flattening the results.
- This requires the predicate to return a collection instead of single element.

ex : wordcount = buck.flatMap(line ⇒ line.split(" "));

fold (zeroValue : T) (op : (T, T) ⇒ T)

- Aggregates the elements ~~of each~~ each partition & results of each partition
- Similar to reduce but allows specifying zero value which gets passed a first Argument to predicate during first call in each partition

foreach (f : (T) ⇒ Unit)

- Applies predicate to all elements of RDD.
- This is an action & not transformation
- it does not return anything
- used for side-effects like printing elements of RDD

groupBy (f: (T) ⇒ K) returns RDD[(K, Iterable(T))]

- groups a tuple by the specified ~~field~~ predicate (the predicate returns a field which becomes to grouping key)
- this function creates a nested structure
- After this function, we can used mapValues for doing aggregations (max, min, sum).
- After this function, we can use flatMapValues for doing sorting / ranking.

intersection (other: RDD)

- returns an RDD which contain common elements to this and other RDD

keyBy [K] (f: (T) ⇒ K) returns RDD[(K, T)]

- creates a tuple of K, V where K is based on field or fields that is returned by the predicate

ex: depRDD.keyBy(x ⇒ x.dep_id)

will return RDD[(dep_id, dep_Record)]

* map[U](f: (T) ⇒ U) returns RDD(U)

- returns a new RDD by applying the predicate to elements of this RDD.
- Used for doing transformations on whole RDD (not by key)
- This is a PUSH MODEL where elements in source RDD are pushed one by one into the predicate function call.

mapPartitions[U](f: Iterator(T) ⇒ Iterator(U))

mapPartitionsWithIndex[U](f: (Int, Iterator(T)) ⇒ Iterator(U))

- Similar to map but applies the function at partition level instead of each element.
- All the elements of a partition are passed in as an Iterator through which we can traverse
- mapPartitionsWithIndex() additionally passes the partition number.
- This functions are useful for doing any initialization at partition level (instead of each element, can enhance performance)
- Since this functions pass in an Iterator (instead of element as in map()) and we have to pull elements within the predicate this is a PULL MODEL
- These functions convert all elements from U to V and return an Iterator(V). (which means we can also have logic to drop certain elements)

max() (ordering func optional)
min() (ordering func optional)

persists (newLevel) - sets RDDs storage level to persist value across operations

unpersist() - make RDD non-persistent & remove all blocks from memory & disk

* reduce (f : (T, T) => T)
  * used to perform overall / global aggregations (sum, min, max)
  * can be applied to single elements, tuples etc.
  * cannot be used for "per key" aggregations

ex: Find max price of product
products.reduce((x, y) => if (x.price > y.price) x else y));

repartition (numPartitions) - return new RDD that contain exactly numPartitions

saveAsObjectFile (path)
saveAsTextFile (path)

saveAsNewAPIHadoopFile (path, class[key], class[value]
class[outputFormat[key, value]]);

ex: import org.apache.~~spark~~hadoop.mapreduce.lib.output._

depRDD.saveAsNewAPIHadoopFile (outputPath,
classOf[IntWritable], classOf[Text],
classOf[SequenceFileOutputFormat[IntWritable, Text]])

* sortBy(f : (T) => K, asc = true, numPartitions = default)

* used for global sorting of RDD by any field.
* can be used with single elements, tuples etc
* predicate returns key to sort on)
* can be used to sort on multiple fields

ex: Sort products by category-id in Asc (2nd field) and Price in DESC (5th field)

products.sortBy(x => (x._2, -x._5)

subtract(other: RDD)
- returns RDD that contains elements in this, which are not present in other,

RANKING

take(n) - take first n elements from RDD
- this is an action that returns an Array.

takeOrdered(N)(ordering func)
- first sorts the elements (default Asc on first field) or according to the specified ordering function. and then takes only the first N elements

ex: products.takeOrdered(3); take 5 smallest ~~bottom~~ 3 based on first field.

products.takeOrdered(3)(Ordering[Int].on(x => x.-1))

The above two func calls are equivalent.

// Top 3 based on third field in tuple
products.takeOrdered(3)(Ordering[Int].reverse.on(x => x._3))

// Top 3 based on ordering with multiple columns
first column is Int which is 2nd field (Asc order
second column is Float which is 5th field (DESC order)

products.takeOrdered(3)
(Ordering[Int, Float].on(x => (x._2, -x._5)))

* top(N)(Ordering function)

- works in exact opposite way of takeOrdered by sorting elements in DESC order by default & based on first field.
- This can be overridden by specifying Ordering function (either change to Asc order or choose other fields)

// top 3 records based on sorting on field 2
products.top(3)(Ordering[Float].on(x => x._2)

Union(other: RDD)

- returns the RDD which is union of this with other (Simply appends other to this)

PAIR RDD Functions

* Only available for tuples with K, V pairs.

* reduceByKey (func : (V, V) => V)
  ← reducer function

  * passes in an accumulator & value (one per predicate call) and the predicate should return one element (tuple or single value) but not a collection. (not suitable for sorting, ranking)
  * ~~for~~ This is similar to reduce but the values will be first grouped by the key and then the values for each key will be passed one-by-one into the predicate function.
  * This method is used to do aggregations (sum, max, min per key basis.
  * is a transformation
  * Uses a combiner whose logic is same as the predicate (reducer function) and hence this method is efficient way of doing per key aggregation.
  * The source RDD could be a flat structure of K, V
  * Since values per key are passed to predicate one-by-one this is a PUSH MODEL

ex : totalOrderAmtPerOrder

```
= orders.reduceByKey
    ((acc, value) => acc + value);
```

Assuming orders is RDD[Int, Float]
↓ orderId ↓ subtotal.

C V C

combineByKey [C] (combinerFunc : (acc, val) ⇒ Value),
func : (acc, val) ⇒ Value)
(reducer) C C C

* Similar to reduceByKey but use when the combiner logic is different than reducer function.
* The input ~~and output~~ to the combiner & output of combine & reduce function should be of same datatype

* **aggregateByKey** (zeroVal : U)
(combineOp : (U, V) ⇒ U,
reduceOp : (U, U) ⇒ U)

* more flexible than reduceByKey & combineByKey
* used when we need not only a different combiner & reducer logic but also the input type to combiner & reducer to be different.
* Also allows to specify a zero value which gets passed into combineOp (1st param) per key.
* This zero value should be of the same type that we expect out of the combiner predicate & reducer predicate

ex: Calculate Avg revenue per day

```
val revenueAggPerDay =
    orderRevMap.aggregateByKey((0.0, 0))
        ((acc, val) => {(acc._1 + value, acc._2 + 1)}
        ((acc, value) => {(acc._1 + value._1, acc._2 + value._2)}
```

zeroValue we are expecting tuple of (Float, Int)

↑ combineOp

↑ we are accumulating the total in first field & count in 2nd field

↑ reducerOp both input parameters match combiner output ie Tuple (Float, Int) bc acc & value are both tuples

↑ we are adding up as intermediate totals from first field & intermediate counts from 2nd field

foldByKey (zeroVal : V) (func : (V, V) => V)

* Similar to reduceByKey but allows to specify a zero value
* the predicate logic is used for both combiner & reducer.

* groupByKey (numPartitions)
  - groups a K,V flat structure & returns a key with Iterable (values) i.e nested structure
  - normally used for sorting & ranking of values per key.
  - does not use combiner hence should not be used for performing subsequent aggregations.

ex : val productsGroupByCat = products.groupByKey();
Assuming products is K,V pair where K is category_id.

mapValues [U] (func : (V) => U)

* used to perform a function over all values per key.
* can perform transformation (aggregate) to return single value or collection. In case of returning collection this function results in a nested structure. K, multiple values) [V1, V2 V3]
* For each key the values are passed in as is into the predicate.
* In case the value V is iterable, its responsibility of predicate to PULL the values.
* Normally used after grouping of data

ex: val grouped = orders.groupByKey() OR orders.groupBy(_._1)

Assuming K, V where K is order_id &
V is order subtotal

val revenuePerOrder = grouped.mapValues(v => v.sum);

flatMapValues [U] (func : (V) => U);

V could be iterable or single value

* Similar to mapValues but the predicate should return a collection since this method flattens the result
* used mostly for sorting & ranking values within a key.

ex: // Top 3 products per category by price.

val prodGroup = products.groupByKey() OR groupBy(_._1)

// Assume products is K, V where K is prod.cat
V is product tuple

NOTE takeO is with flatMapValues

val top3 productGroup.flatMapValues(x => x.toList.sortBy(-_._4).take(3))

This will result in a collection of top 3 records sorted by price (4th field) DESC

flatMapValues will convert this nested structure K, Iterable (V) into flat structure

K, V1
K, V2
K, V3

≠ Nested Tuples (Tuple inside Tuple) is not considered to be a nested structure. It's still a flat structure.

keys() - returns RDD containing keys of each tuple

values() - returns RDD containing values of each tuple

lookup(key:K) - returns a list of values for a given RDD key
- this is an action

JOINS

join [W] (other : RDD [K, W]) returns RDD [(K, (V, W))]

* joins two RDDs and returns a tuple containing the key and the corresponding records from this and other as a tuple
* generates a flat structure
* performs an inner join (filters out records from this & other that don't exist).

ex: val joined = orders.join(orderItems)

Assume orders is K, V where K is order_id (int)
V is order record String

orderItems is K, V where K is order_id (int)
V order Items record String

then join will be

RDD : [(Int, (String, String))]

- Int → key (order_id)
- String → order record
- String → order item record

leftOuterJoin (other : RDD)

* Same as join but performs left outer join
* which means the 2nd field in the value tuple of the result can be empty for missing records in other
* generates flat structure.

rightOuterJoin (other : RDD)

* same as join but performs right outer join.
* the 1st field in the value tuple of the result could be empty for missing records in this.
* generates flat structure.

fullOuterJoin (other : RDD)

* performs a full outer join
* the 1st or 2nd field in the value tuple of the result will be empty for missing records in this OR other respectively.
* generates flat structure

* **cogroup** (also called groupWith)

[W] (other : RDD[K, V]) : RDD[(K, (Iterable(V), Iterable(W))]

- performs a full outer join but generates a nested structure because it also groups the result after the join by the key.

ex: val stocks — Assume K, V (K is stock symbol (String) & V is stock record (String))

val companies — Assume K, V (K is stock symbol (String) & V is company record (String))

val cogrouped = stocks.cogroup(companies)

Cogrouped will be RDD[(String, (Iterable(String), Iterable(String)))]

- String → key (stocksymbol)
- Iterable(String) → iterator for all stock records for this symbol
- Iterable(String) → iterator for all company record for this key.

We can remove nesting by using flatMapValues to combine the two iterables into one Iterable

```
cogrouped.flatMapValues(
    v => for(v1 <- v._1, v2 <- v._2)
         yield (v1, v2)
)
```

- v._1 : value from 1st iterable
- v._2 : value from 2nd iterable
- (v1, v2) : combine them into one tuple.