COMP5349 – Cloud Computing

Week 5: Spark Framework

Dr. Ying Zhou
School of Computer Science

Last Week

- Last week we cover MapReduce framework
- MapReduce is the first kind of big data processing framework
  - Many other frameworks are either built on top of it, or follow its design principle
- MapReduce borrows many concepts from functional programming
  - Two higher order functions: ***map*** and ***reduce***
- It divides a data analytic workload into a sequence of jobs consisting of map phase and reduce phase.
- Both Map and Reduce phase can be parallelized
- *Key value pair* is the prominent data structure

Outline

- **Data Sharing and Job Chaining in MapReduce**
- **Spark Basic Concepts**
- **A complete Spark Application Example**

Data Sharing in MapReduce

- The mappers and reducers are designed to run independently on ***different partitions*** of the input data on different nodes
- The API for mapper and reducer only takes input key value pairs as arguments
- How do we pass *additional* program data?
- E.g. for a simple word count program, we may want to
  - Get rid of all stop words, the list of stop words is stored in a text file
  - Remove words that occurs less than a threshold, the threshold value can be given as command line argument or in a property file
  - How do we share the stop word file among all mappers?
  - How do we tell all reducers the threshold values?

Sharing parameters

- Sharing parameters of simple type can be achieved using the **`Configuration`** object
  - E.g. a threshold value (int), a timestampe value, some string value
- In Java application
  - the driver program will set the property with the configuration

    conf.set("mapper.placeFilter.country", “Australia“)
  - Both mapper and reducer can read it out

    context.getConfiguration().get("mapper.placeFilter.country", countryName);
- In Python application
  - Simple parameter can be specified as argument of the mapper or reducer script

    mapper "place_filter_mapper.py Australia"

Distributing Auxiliary Job data

- Auxiliary job data
  - In general a small file contains common background knowledge for map and/reduce functions
    - E.g. the stop word list for word counting, the dictionary for spelling check
  - All mappers/reducers need to read it
  - The file is small enough to fit in the memory of mappers/reducers
- Hadoop provides a mechanism for this purpose called the **`distributed cache`**.
  - Files put in the distributed cache is accessible by both mappers and reducers.
- Distributed cache can be used to provide an efficient join if one join table is small enough to fit in the memory

Chaining Jobs

- Most of the time, an analytic workload cannot be implemented in a single MapReduce job, we need to chain multiple jobs
- E.g. If we want to sort words descendingly based on their occurrence in a cohort
  - We need two jobs, the first job does the *word counting*, the second one does the *sorting*
  - The second job uses the output of the first job as input
- In Java API
  - The chaining happens in the driver program, define jobs one by one and submitting them in the correct order
- In Python API
  - We use streaming API multiple times to submit jobs one by one

A simple `select-join` example

- Two input files
  - **`Photo.csv:`**

```
phoeo_id \t owner \t tags \t date_taken \t place_id \t accuracy
```

  - **`Place.csv:`**

```
place_id \t woeid \t lat \t longi \t place_name \t place_url
```

- We want to find all photos taken in “Australia”
- Assuming **place.csv** is large, but its subset containing “Australia” is quite small
- Assuming photo.csv is very large
- Two jobs:
  - The first job reads the **place.csv** data, filters only rows containing word Australia in **place_url** and saves the output in a file
  - The second job uses distributed cache to distribute the result of the first job and joins it with **photos.csv**

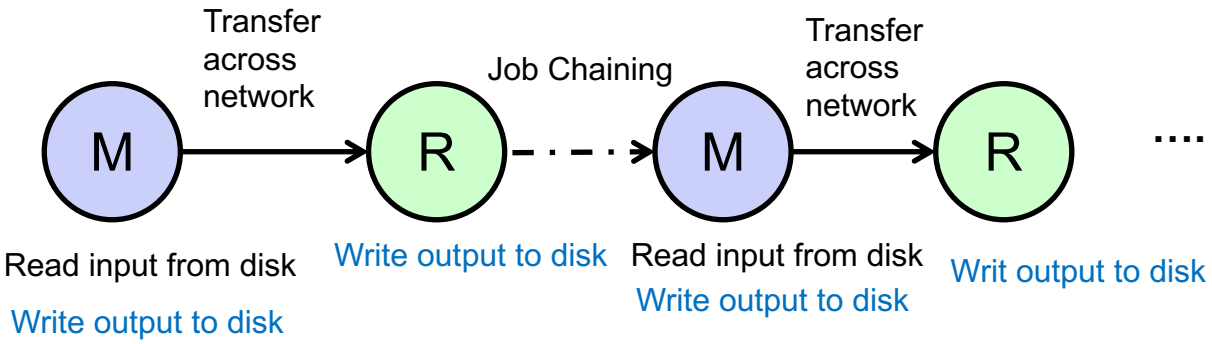
See course git repos for example in Python code

Outline

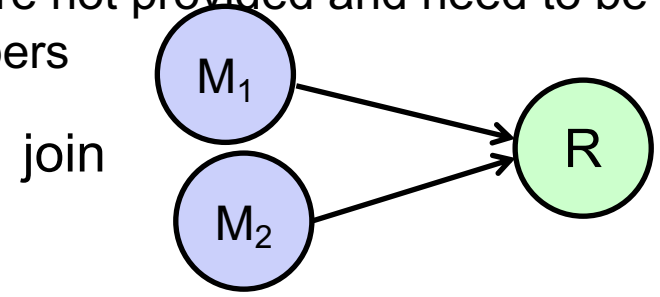
- Data Sharing and Job Chaining in MapReduce
- **Spark Basic Concepts**
  - **Motivation**
  - **RDD and its operations**
  - **Functional Programming Revisit**
  - **General Structure of Spark Program**
- A complete Spark Application Example

Two major issues with MapReduce

- Map/Reduce are storage based
  - Pro: parallelism, fault-tolerance, runtime can decide where to run tasks
  - Con: simple processing model that assumes data flowing from stable storage to stable storage + materialization of intermediate results



- Provide a simple set of API: **map** and **reduce**
  - Pro: very flexible, you can implement various processing in the map and reduce function.
  - Con: means common processing tasks are not provided and need to be implemented again and again by developers



Data Analytics Workload

- Exploratory
  - Summarizing the main characteristics of the data set
  - Comparing across various columns
  - Such exploration may reveal high level features of the data set and help to make decisions
    - E.g. we may need to set up a data center in Australia, it seems we have a lot of customers there
- Predictive
  - Build either statistic or machine learning models to make prediction
    - E.g. which movie this customer may like;
  - Most machine learning algorithms are iterative,
    - The data set will be scanned and processed multiple times until some stop criterion is reached
    - The results of pervious iteration will be used in next iteration
  - Storage based MapReduce is not suitable for such algorithm

Beyond MapReduce

- One type of attempt, which deals with mainly exploratory data analysis, built a SQL, or SQL like layer on top of MapReduce
  - Pig, HIVE and others
  - SQL like data analytic expressions are automatically converted into MapReduce programs
    - common processing like filtering, projection, joining are implemented
  - Various optimization techniques has been proposed to achieve performance similar to or better than hand coded version
  - Backend engine is still storage heavy MapReduce
- Another type of attempt, which covers both exploratory and predictive analysis, is the data flow based analysis system
  - Memory based + rich set of APIs
  - Spark and Flink

Apache Spark

- In-***memory*** framework for interactive and iterative computations
- Goals:
  - Distributed memory abstractions for clusters to support apps that needs to repeatedly reuse working sets of data
  - Retain the attractive properties of MapReduce:
    - Fault tolerance (for crashes & stragglers)
    - Data locality
    - Scalability
    - Functional programming flavour
- Approach:
  - Augment data flow model with **Resilient Distributed Dataset (RDD)**
    - RDD: fault-tolerance, in-memory storage abstraction
  - New data structure abstraction such as DataFrame is proposed and becomes the main data structure in later versions.

Spark RDD Programming Model

- **Resilient distributed datasets (RDDs)**
  - Immutable **collections** partitioned across cluster that can be rebuilt if a partition is lost
  - Created by transforming data in stable storage using data flow operators (map, filter, group-by, …)
  - Can be *cached* across parallel operations
- Parallel operations on RDDs
  - Transformations and actions
- Restricted shared variables
  - Accumulators, broadcast variables

Resilient Distributed Datasets

- RDDs are created by
  - Parallelizing an existing collection
  - Referencing a dataset in an external storage system

```
//parallelizing existing collection
    data = [1, 2, 3, 4, 5]
    distData = sc.parallelize(data)

//referencing a data set in HDFS
    lines = sc.textFile("data.txt")
```

- RDDs may contain key value pair as record
  - If the data in an RDD is of tuple type, the first element would be treated as ***key*** automatically, the rest will be ***value***s
  - The ***value*** can be of simple type, or of tuples

```
kvData = [('a',1), ('b',2), ('c',3,) ('d',4), ('e',5)]
kvDistData = sc.parallelize(data)
```

RDD operations

- Transformation
  - create a new dataset from an existing one
  - Eg. map(func), flatMap(func), reduceByKey(func)
- Action
  - return a value to the driver program after running a computation on the data set
  - Eg. count(), first(), collect(), saveAsTextFile(path)
- Most RDD operations take *one* or *more* functions as parameter
  - Most of them can be viewed as higher order functions
- Spark has strong functional programming flavour!

Spark Program

- A Spark program is just a regular **main** program/script that creates **`SparkContext`** object, which is used to access a cluster.

- The spark context provides methods for
  - Data input
  - Data output
  - ...

- The data processing steps are defined using **RDD** transformations and actions

WordCount in Spark (Python)

```
from pyspark import SparkConf, SparkContext

myconf = (SparkConf().setMaster(”…")
                     .setAppName("WordcountExample")
                     .set("spark.executor.memory", "1g")
sc = SparkContext(myconf)

text_file = sc.textFile("hdfs://...")
counts = text_file.flatMap(lambda line: line.strip().split(" ")) \
                  .map(lambda word: (word, 1)) \
                  .reduceByKey(lambda a, b: a + b)
counts.saveAsTextFile("hdfs://...")
```

Create RDD from input file

One transformation

See demo of word count in Ed workspace

[Cf.: http://spark.apache.org/examples.html]

Revisit: functional programming

- Higher order functions can take other functions as parameter
- MapReduce provides two such higher order functions
  - **map** and **reduce**
  - MapReduce API defines the signature of functions supplied to map and reduce
    - Functions supplied to map should take a key value pair as input and returns a list of key value pairs
    - Functions supplied to reduce take a key and value list as input and returns a key value pair
  - The signature is general enough to handle many types of processing
  - MapReduce framework is responsible for calling those user supplied "functions"
    - They can be implemented as method, script, etc..
    - Actual programming does not involve a lot functioning flavour

Revisit: functional programming

- In Spark, most RDD transformations are higher order functions, they take one or more functions as argument(s)
- Spark API defines and restricts the *signature of functions* supplied to each transformation
- The actual invocation is user specified
  - Calling a transformation on an RDD with a given function

A transformation

```
text_file.flatMap(lambda line: line.strip().split(" "))
```

an RDD

A function argument

Anonymous Functions

- Functional argument can be compactly expressed using anonymous functions
  - **Python: map (lambda a: 2*a)**
- Lambda expression is a way to express anonymous functions in programming language
- Nearly all functional languages and most script languages support anonymous function
- Anonymous function cannot be reused efficiently

Lambda Expression

- Lambda expression is quite hard to understand when the function logic gets complicated
  - Function body is embedded in the main program logic
  - The argument(s) and return types are not declared explicitly
- **`Function`** is a language concept in Python, lambda expression is only used for very simple functions
- It is much easier to define a function then call it using the function name.

RDD Operation functional argument

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

flatMap(*f*, *preservesPartitioning=False*) [source]

Return a new RDD by first applying a function to all elements of this RDD, and then flattening the results.

```
>>> rdd = sc.parallelize([2, 3, 4])
>>> sorted(rdd.flatMap(lambda x: range(1, x)).collect())
[1, 1, 1, 2, 2, 3]
>>> sorted(rdd.flatMa
[(2, 2), (2, 2), (3,
```

map(*f*, *preservesPartitioning=False*)

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

```
>>> rdd = sc.parallelize(["b", "a", "c"])
>>> sorted(rdd.map(lambda x: (x, 1)).collect())
[('a', 1), ('b', 1), ('c', 1)]
```

reduceByKey(*func*, *numPartitions=None*, *partitionFunc=<function portable_hash>*)

Merge the values for each key using an associative and commutative reduce function.

reduce(*f*) [sourc

Reduces the elements of this RDD using the specified commutative and associative binary operator. Currently reduces partitions locally.

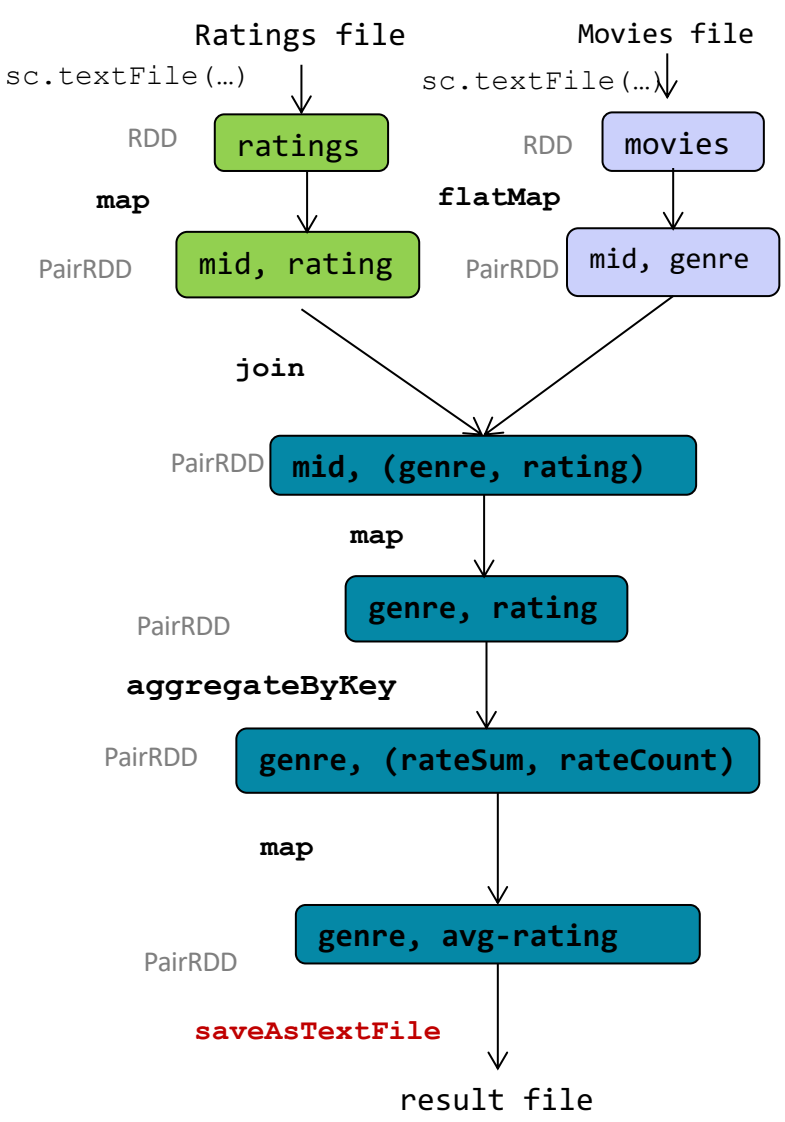
Outline

- Data Sharing and Job Chaining in MapReduce
- Spark Basic Concepts
  - RDD and its operations
  - Functional Programming Revisit
  - General Structure of Spark Program
  - Lambda Expression in Java
- **A Complete Spark Application Example**

A sample program

- Two data sets stored as txt files
- Movies (mid, title, genres)
  - Sample data:
  - `1,Toy Story (1995),Adventure|Animation|Children|Comedy|Fantasy`
- Ratings (uid, mid, rating, timestamp)
  - Sample data:
  - `1,253,3.0,900660748`
- We want to find out the average rating for each genre
  - We would join the two data sets on movie id (**`mid`**) and keep only the **genre** and **rating** data, we then group the rating data based on genre and find the average for each genre.

Spark RDD operation design



uid, mid, rating, timestamp

```
1,1,3.0,900660748
1,2,4.0, 932640588
```

mid, title, genres

```
1,Toy Story (1995), Animation|Children
2,Babe(1995), Children
```

```
"1,1,3.0,900660748"
"1,2,4.0,932640588"
```

```
"1,Toy Story (1995),
Animation|Children",
"2,Babe(1995), Children"
```

```
(1,3.0)
(2,4.0)
```

```
(1,Animation)
(1,Children)
(2,Children)
```

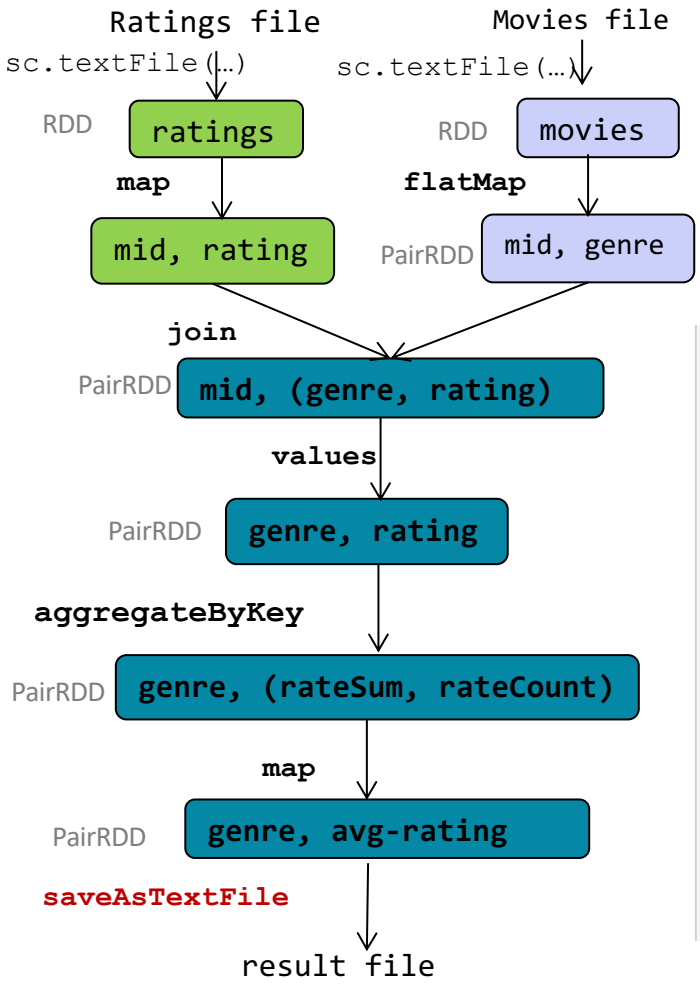
```
(1,(Animation,3.0))
(1,(Children,3.0))
(2,(Children,4.0))
```

```
(Animation,3.0)
(Children,3.0)
(Children,4.0)
```

```
(Animation,(3.0,1))
(Children,(7.0,2))
```

```
(Animation,3.0)
(Children,3.5,)
```

Spark Program Skeleton




```
sc = SparkContext(appName="Average Rating per Genre")

#You can change the input path pointing to your own HDFS
#If spark is able to read hadoop configuration, you can use relative path
input_path = 'hdfs://soit-hdp-pro-1.ucc.usyd.edu.au/share/movie/small/'

#Relative path is used to specify the output directory
#The relative path is always relative to your home directory in HDFS: /user/<yourUserName>
output_path = 'ratingOut'

ratings = sc.textFile(input_path + "ratings.csv")
movieData = sc.textFile(input_path + "movies.csv")

movieRatings = ratings.map(extractRating)
movieGenre = movieData.flatMap(pairMovieToGenre) # we use flatMap as there are multiple genre

genreRatings = movieGenre.join(movieRatings).values()
genreRatingsAverage = genreRatings.aggregateByKey((0.0,0),
                                                  mergeRating,
                                                  mergeCombiners, 1).map(mapAverageRating)

genreRatingsAverage.saveAsTextFile(output_path)
```

Design Functions

```
movieRatings = ratings.map(extractRating)
movieGenre = movieData.flatMap(pairMovieToGenre)
```

```
21 def extractRating(record):
22     """ This function converts entries of ratings.csv into key,value pair of the following format
23     (movieID, rating)
24     Args:
25         record (str): A row of CSV file, with four columns separated by comma
26     Returns:
27         The return value is a tuple (movieID, genre)
28     """
29     try:
30         userID, movieID, rating, timestamp = record.split(",")
31         rating = float(rating)
32         return (movieID, rating)
33     except:
34         return ()
```

```
4  def pairMovieToGenre(record):
5      """This function converts entries of movies.csv into key,value pair of the following format
6      (movieID, genre)
7      since there may be multiple genre per movie, this function returns a list of tuples
8      Args:
9          record (str): A row of CSV file, with three columns separated by comma
10     Returns:
11         The return value is a list of tuples, each tuple contains (movieID, genre)
12     """
13     try:
14         movieID, name, genreList = record.split(",")
15         genres = genreList.split("|")
16         return [(movieID, genre) for genre in genres]
17     except:
18         return []
```

https://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD

Design Functions

- AggregateByKey transformation

```
genreRatings.aggregateByKey((0.0,0),
                            mergeRating,
                            mergeCombiners, 1)
```

There are a few transformations similar to the **`reducer`** in MapReduc framework

`groupByKey` groups the values for each key. This is like the step of preparing input for reducer in MapReduce framework

`reduceByKey` merge the values for each key using a given reduce function. This will also perform the merging locally on each “mapper” before sending results to a reducer, similarly to a "combiner" in MapReduce. But the type of the merged value should be the same as the type of the input value!

`foldByKey` is similar to **`reduceByKey`** except that you can supply a natural zero value.

`aggregateByKey` is more general than **`reduceByKey`**. It allows the merged value to have different type of the input value. It takes at least a natural zero value and two functions as parameter.

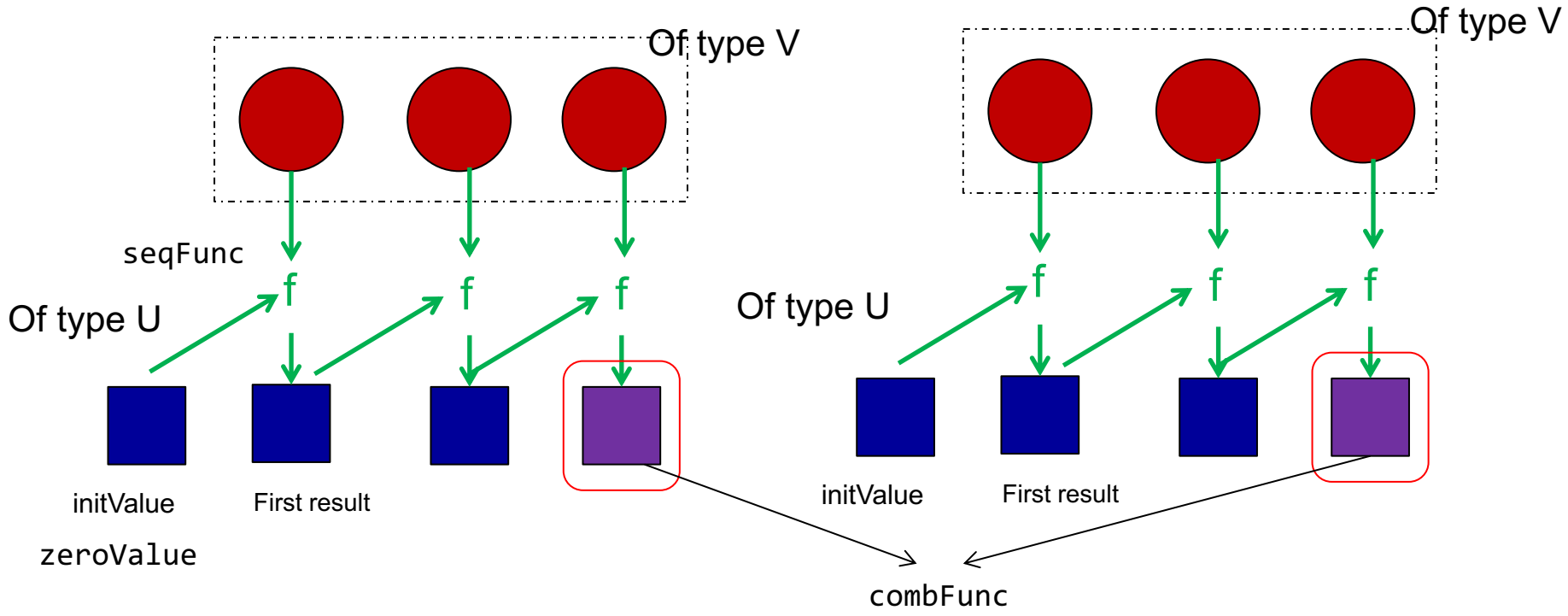
All above transformations can take extra parameter to indicate the number of partition, or a partitioner object. This is like specifying the number of **reducers** in MapReduce, or specifying a customized partitioner.

Design Functions

- aggregateByKey transformation

aggregateByKey(*zeroValue*, *seqFunc*, *combFunc*, *numPartitions=None*, *partitionFunc=<function portable_hash>*) [source]

Aggregate the values of each key, using given combine functions and a neutral "zero value". This function can return a different result type, U, than the type of the values in this RDD, V. Thus, we need one operation for merging a V into a U and one operation for merging two U's, The former operation is used for merging values within a partition, and the latter is used for merging values between partitions. To avoid memory allocation, both of these functions are allowed to modify and return their first argument instead of creating a new U.



Design Functions

```python
def mergeRating(accumulatedPair, currentRating):
    """This funtion update a current  summary (ratingTotal, ratingCount) with a new rating value.

    Args:
        accumulatedPair (tuple): a tuple of (ratingTotal, ratingCount)
        currentRating (float):a new rating value,
    Returns:
        The return value is an updated tuple of (ratingTotal, ratingCount)

    """
    ratingTotal, ratingCount = accumulatedPair
    ratingTotal += currentRating
    ratingCount += 1
    return (ratingTotal, ratingCount)
```

```python
genreRatings.aggregateByKey((0.0,0),
                            mergeRating,
                            mergeCombiners, 1)
```

```python
def mergeCombiners(accumulatedPair1, accumulatedPair2):
    """This function merges two intermedate summaries of the format (ratingTotal, ratingCount)

    Args:
        accumulatedPair1 (tuple): a tuple of (ratingTotal, ratingCount)
        accumulatedPair2 (fuple): a tuple of (ratingTotal, ratingCount)
    Returns:
        The return value is an updated tuple of (ratingTotal, ratingCount)
    """
    ratingTotal1, ratingCount1 = accumulatedPair1
    ratingTotal2, ratingCount2 = accumulatedPair2
    return (ratingTotal1+ratingTotal2, ratingCount1+ratingCount2)
```