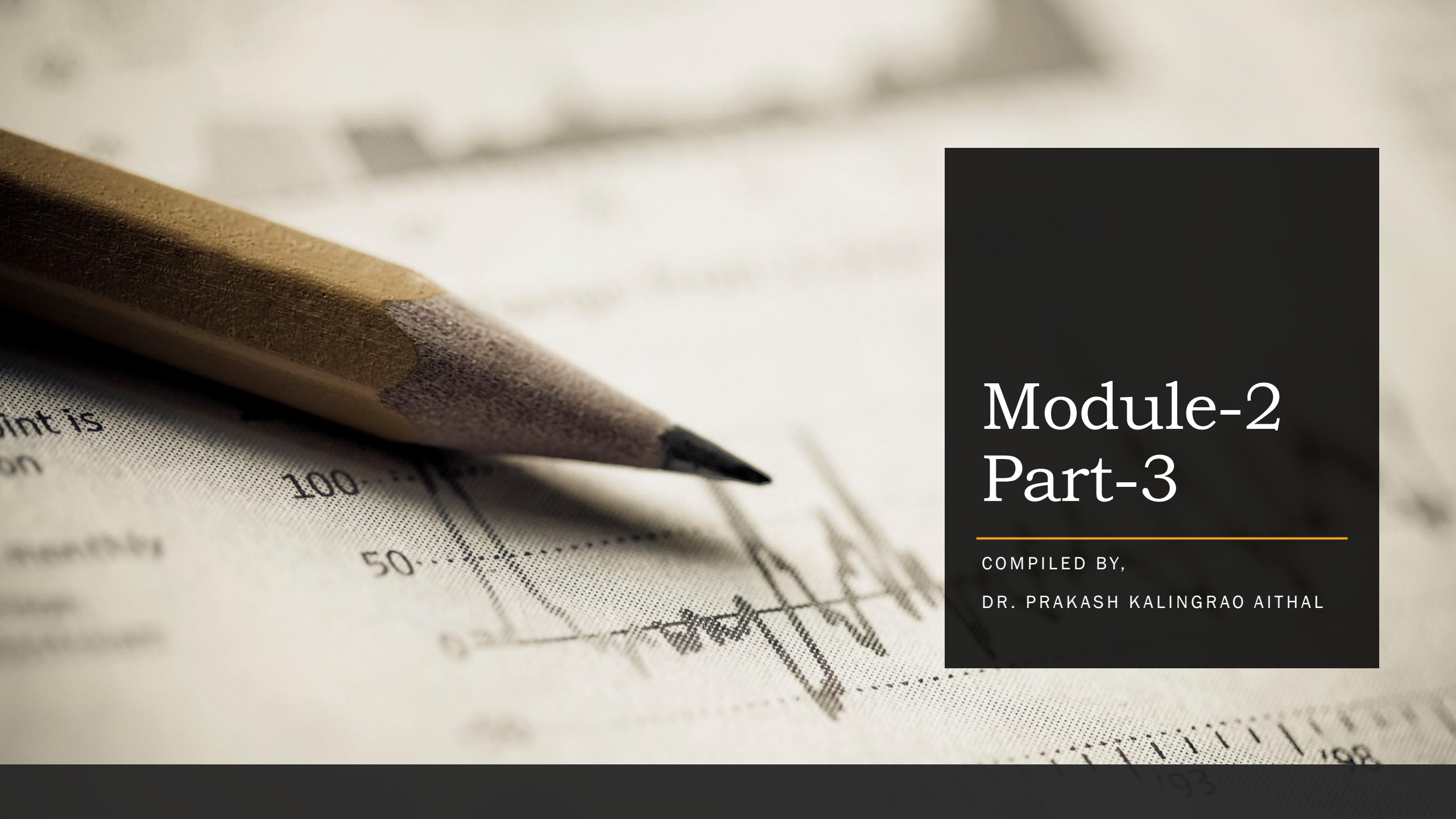

Module-2 Part-3

COMPILED BY,

DR. PRAKASH KALINGRAO AITHAL

Basic RDD Transformations

RDD.map(<function>, preservesPartitioning=False)

The map() transformation is the most basic of all transformations. It evaluates a named or anonymous function for each element within a dataset partition. One or many map() functions can run asynchronously because they shouldn't produce any side effects, maintain state, or attempt to communicate or synchronize with other map() operations. That is, they are *shared nothing* operations.

Basic RDD Transformations

RDD.flatMap(<function>, preservesPartitioning=False)

The flatMap() transformation is similar to the map() transformation in that it runs a function against each record in the input dataset. However, flatMap() "flattens" the output, meaning it removes a level of nesting. For example, given a list containing lists of strings, flattening would result in a single list of strings—"flattening" all of the nested lists.

Basic RDD Transformations

RDD.filter(<function>)

The filter transformation evaluates a Boolean expression, usually expressed as an anonymous function, against each element in the dataset. The Boolean value returned determines whether the record is included in the resultant output RDD. This is another common transformation used to remove from RDD records that are not required for intermediate processing and that are not included in the final output.

Basic RDD Transformations

```
licenses = sc.textFile('file:///opt/spark/licenses')
words = licenses.flatMap(lambda x: x.split(' '))
words.take(5)
# returns [u'The', u'MIT', u'License', u'(MIT)', u'']
lowercase = words.map(lambda x: x.lower())
lowercase.take(5)
# returns [u'the', u'mit', u'license', u'(mit)', u'']
longwords = lowercase.filter(lambda x: len(x) > 12)
longwords.take(2)
# returns [u'documentation', u'merchantability,']
```

Basic RDD Transformations

RDD.distinct(numPartitions=None)

The distinct() transformation returns a new RDD containing distinct elements from the input RDD. It is used to remove duplicates, where *duplicates* are defined as having all elements or fields within a record that are the same as other records in the dataset. The numPartitions argument can redistribute data to a target number of partitions

Basic RDD Transformations


```
licenses = sc.textFile('file:///opt/spark/licenses')
words = licenses.flatMap(lambda x: x.split(' '))
lowercase = words.map(lambda x: x.lower())
allwords = lowercase.count()
distinctwords = lowercase.distinct().count()
print "Total words: %s, Distinct Words: %s" % (allwords, distinctwords)
# returns "Total words: 11484, Distinct Words: 892"
```

2 ways to groupBy (0 -

- by a key (x : x[0])
- an expression to evaluate all ele (x: x%2==0)

Basic RDD Transformations

RDD.groupBy(<function>, numPartitions=None)

The groupBy() transformation returns an RDD of items grouped by a specified function. The <function> argument is an anonymous or named function used to nominate a key by which to group all elements or to specify an expression to evaluate against elements to determine a group, such as when grouping elements by odd or even numbers of a numeric field in the data.

Basic RDD Transformations

```
licenses = sc.textFile('file:///opt/spark/licenses')
words = licenses.flatMap(lambda x: x.split(' ')) \
                .filter(lambda x: len(x) > 0)
groupedbyfirstletter = words.groupBy(lambda x: x[0].lower())
```

Basic RDD Transformations

If your ultimate intention in using `groupBy()` is to aggregate values, such as when performing a `sum()` or `count()` operation, you should opt for more efficient operators for this purpose in Spark, including `aggregateByKey()` and `reduceByKey()`, which we will discuss shortly. The `groupBy()` transformation does not perform any aggregation prior to shuffling data, resulting in more data being shuffled. Furthermore, `groupBy()` requires that all values for a given key fit into memory. The `groupBy()` transformation is useful in some cases, but you should consider these factors before deciding to use this function.

Basic RDD Transformations

RDD.sortBy(<keyfunc>, ascending=True, numPartitions=None)

The sortBy() transformation sorts an RDD by the <keyfunc> argument (a named or anonymous function) that nominates the key for a given dataset. It sorts according to the sort order of the key object type. For instance, int and double data types are sorted numerically, whereas String types are sorted in lexicographical order. The Boolean ascending function determines the sort order.

Basic RDD Actions

RDD.count()

The count() action takes no arguments and returns a long value, which represents the count of the elements in the RDD.

Basic RDD Actions

RDD.collect()

The collect() action returns a list that contains all the elements in an RDD to the Spark Driver. Because collect() does not restrict the output, which can be quite large and can potentially cause out-of-memory errors on the Driver, it is typically useful for only small RDDs or development.

Basic RDD Actions

RDD.take(n)

The take() action returns the first n elements of an RDD. The elements taken are not in any particular order; in fact, the elements returned from a take() action are non-deterministic, meaning they can differ if the same action is run again, particularly in a fully distributed environment. There is a similar Spark function, takeOrdered(), which takes the first n elements ordered based on a key supplied by a key function.

Basic RDD Actions

RDD.top(n, key=None)

The top() action returns the top n elements from an RDD, but unlike with take(), with top() the elements are ordered and returned in descending order. Order is determined by the object type, such as numeric order for integers or dictionary order for strings.

The key argument specifies the key by which to order the results to return the top n elements. This is an optional argument; if it is not supplied, the key will be inferred from the elements in the RDD.

Basic RDD Actions

RDD.first()

The first() action returns the first element in this RDD. Similar to the take() and collect() actions and unlike the top() action, first() does not consider the order of elements and is a non-deterministic operation, especially in fully distributed environments.

Basic RDD Actions

the primary difference between first() and take(1) is that first() returns an atomic data element, and take() (even if n = 1) returns a list of data elements. The first() action is useful for inspecting the output of an RDD as part of development or data exploration.

Basic RDD Actions

RDD.reduce(<function>)

The reduce() action reduces the elements of an RDD using a specified commutative and/or associative operator. The <function> argument specifies two inputs (lambda x, y: ...) that represent values in a sequence from the specified RDD.

Basic RDD Actions

RDD.fold(zeroValue, <function>)

The fold() action aggregates the elements of each partition of an RDD and then performs the aggregate operation against the results for all, using a given function and a zeroValue. Although reduce() and fold() are similar in function, they differ in that fold() is not commutative, and thus an initial and final value (zeroValue) is required.

Basic RDD Actions

```
numbers = sc.parallelize([1,2,3,4,5,6,7,8,9])
numbers.fold(0, lambda x, y: x + y)
# returns 45
```

Basic RDD Actions

```
empty = sc.parallelize([])
empty.reduce(lambda x, y: x + y)
# returns:
# ValueError: Cannot reduce() empty RDD
empty.fold(0, lambda x, y: x + y)
# returns 0
```

Basic RDD Actions

RDD.foreach(<function>)

The foreach() action applies a function specified in the <function> argument, anonymous or named, to all elements of an RDD. Because foreach() is an action rather than a transformation, you can perform functions otherwise not possible or intended in transformations

Basic RDD Actions

```
def printfunc(x):
    print(x)
licenses = sc.textFile('file:///opt/spark/licenses')
longwords = licenses.flatMap(lambda x: x.split(' ')) \
           .filter(lambda x: len(x) > 12)
longwords.foreach(lambda x: printfunc(x))
# returns:
# ...
# Redistributions
# documentation
# distribution.
# MERCHANTABILITY
# ...
```

Transformations on PairRDDs

Key/value pair RDDs, or simply PairRDDs, contain records consisting of keys and values. The keys can be simple objects such as integer or string objects or complex objects such as tuples. The values can range from scalar values to data structures such as lists, tuples, dictionaries, or sets. This is a common data representation in multi-structured data analysis on schema-on-read and NoSQL systems. PairRDDs and their constituent functions are integral to functional Spark programming. These functions are broadly classified into four categories:

- Dictionary functions
- Functional transformations
- Grouping, aggregation, and sorting operations
- Join functions

Transformations on PairRDDs

RDD.keys()

The keys() function returns an RDD with the keys from a key/value pair RDD or the first element from each tuple in a key/value pair RDD.

Transformations on PairRDDs

```
kvpairs = sc.parallelize([('city','Hayward')
                          ,('state','CA')
                          ,('zip',94541)
                          ,('country','USA')])
kvpairs.keys().collect()
# returns ['city', 'state', 'zip', 'country']
```

Transformations on PairRDDs

RDD.values()

The values() function returns an RDD with values from a key/value pair RDD or the second element from each tuple in a key/value pair RDD.

Transformations on PairRDDs

RDD.keyBy(<function>)

The keyBy() transformation creates a tuple consisting of a key and a value from the elements in the RDD by applying a function specified by the <function> argument. The value is the complete tuple from which the key was derived.

Transformations on PairRDDs

```
locations = sc.parallelize([('Hayward', 'USA', 1)
                           ,('Baumholder','Germany', 2)
                           ,('Alexandria','USA', 3)
                           ,('Melbourne','Australia', 4)])
bylocno = locations.keyBy(lambda x: x[2])
bylocno.collect()
# returns:
#[(1, ('Hayward', 'USA', 1)), (2, ('Baumholder', 'Germany', 2)),
# (3, ('Alexandria', 'USA', 3)), (4, ('Melbourne', 'Australia', 4))]
```

Transformations on PairRDDs

RDD.mapValues(<function>)

The mapValues() transformation passes each value in a key/value pair RDD through a function (a named or anonymous function specified by the <function> argument) without changing the keys. Like its generalized equivalent map(), mapValues() outputs one element for each input element.

Transformations on PairRDDs

RDD.flatMapValues(<function>)

The flatMapValues() transformation passes each value in a key/value pair RDD through a function without changing the keys and produces a flattened list.

Transformations on PairRDDs

RDD.groupByKey(numPartitions=None, partitionFunc=<hash_fn>)

The groupByKey() transformation groups the values for each key in a key/value pair RDD into a single sequence. The numPartitions argument specifies how many partitions—how many groups, that is—to create. The partitions are created using the partitionFunc argument, which defaults to Spark's built-in hash partitioner. If numPartitions is None, which is the default, then the configured system default number of partitions is used (spark.default.parallelism).

Transformations on PairRDDs

RDD.reduceByKey(<function>, numPartitions=None, partitionFunc=<hash_fn>)

The reduceByKey() transformation merges the values for the keys by using an associative function. The reduceByKey() method is called on a dataset of key/value pairs and returns a dataset of key/value pairs, aggregating values for each key.

Transformations on PairRDDs

RDD.foldByKey(zeroValue, <function>, numPartitions=None, partitionFunc=<hash_fn>)

The foldByKey() transformation is functionally similar to the fold() action discussed in the previous section. However, foldByKey() is a transformation that works with predefined key/value pair elements

Transformations on PairRDDs

RDD.sortByKey(ascending=True, numPartitions=None, keyfunc=<function>)

The sortByKey() transformation sorts a key/value pair RDD by the predefined key. The sort order is dependent on the underlying key object type, where numeric types are sorted numerically and so on. The difference between sort(), discussed earlier, and sortByKey() is that sort() requires you to identify the key by which to sort, whereas sortByKey() is aware of the key already.

Transformations on Sets

RDD.union(<otherRDD>)

The union() transformation takes one RDD and appends another RDD to it, resulting in a combined output RDD. The RDDs are not required to have the same schema or structure. For instance, the first RDD can have five fields, whereas the second can have more or fewer than five fields. Union operation neither sorts nor removes duplicates

Transformations on Sets

RDD.intersection(<otherRDD>)

The intersection() transformation returns elements that are present in both RDDs. In other words, it returns the overlap between two sets. The elements or records must be identical in both sets, with each respective record's data structure and all of its fields matching in both RDDs.

Transformations on Sets

RDD.subtract(<otherRDD>, numPartitions=None)

The subtract() transformation returns all elements from the first RDD that are not present in the second RDD. This is an implementation of a mathematical set subtraction.

Transformations on Sets

RDD.subtractByKey(<otherRDD>, numPartitions=None)

The subtractByKey() transformation is a set operation similar to the subtract transformation. The subtractByKey() transformation returns key/value pair elements from an RDD with keys that are not present in key/value pair elements from otherRDD.