CSC 369

Introduction to Distributed Computing

Resilient Distributed Dataset (RDD)

- Immutable distributed collection of objects.
- An RDD is just a function, the recipe to compute the RDD is stored, but not the actual result. RDDs are evaluated using a lazy approach.
- Two ways to create an RDD:
  - create them dynamically from an existing list

    ```
    sc.parallelize(range(100))
    ```
  - load them from a file

    ```
    sc.textFile(data_file)
    ```
- Saving text file:

  ```
  rdd.saveAsTextFile("....")
  ```

can save whenever we want

Transformations and Actions

- Transformations:
  - Transforms the RDD into a new RDD. For example:

    ```
    normal_raw_data = raw_data.filter(lambda x: 'normal.' in x)
    ```
  - Usually, result is computed on demand (lazy). For example, using a pipeline when there are several transformations.

- Actions:
  - Compute some result that is not in the form of a RDD. The result is stored on disk or displayed.
  - For example:

    ```
    normal_count = normal_raw_data.count()
    ```

Text

Persisting the Result

- Consider an RDD that is computed by applying several transformations.
- We can call an action on it to get a result.
- However, the result of the RDD is not stored (the reason is that it may be too big).
- Every time we reference the RDD, it will be recomputed (it is resilient, i.e., it can be recomputed if we lost the result data).
- If we are going to call multiple actions on the same RDD result, then we can persist it for efficiency. In this way, we would not do the same computations multiple times.
- By default, `persist` stores the result in main memory, but we can change this behavior.

Persistance Choices

Storage Level	Meaning
MEMORY_ONLY (default)	stores in memory. Whatever doesn't fit will be recomputed on the fly.
MEMORY_AND_DISK	stores what doesn't fit in main memory on the hard disk
MEMORY_ONLY_SER	Serialize the data on a single server
MEMORY_AND_DISK_SER	Serialize whatever doesn't fit in main memory
DISK_ONLY	Store everything on disk
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.	Make 2 copies of everything.

collect() / flatMap()

```
rdd = sc.parallelize(list(range(8)))
rdd.map(lambda x: x ** 2).collect() # Square each element
```

`collect()` Brings all the data to a single node (so data is not partitioned over multiple nodes). You should use `collect()` before printing result. Use it when you don't have an action at the end.

`flatMap()`: converts an RDD of lists into an RDD of elements.

Colab Notebook

Print Distinct Words from a File

- `flatMap` helps us get all the words in the file
- `distinct` gets the distinct elements
- remember to always use `collect` before printing the elements of an RDD.
    - This guarantees that all the print statements will happen on the same machine.

Table 3-2. Basic RDD transformations on an RDD containing {1, 2, 3, 3}

Function name	Purpose	Example	Result
`map()`	Apply a function to each element in the RDD and return an RDD of the result.	`rdd.map(x => x + 1)`	`{2, 3, 4, 4}`
`flatMap()`	Apply a function to each element in the RDD and return an RDD of the contents of the iterators returned. Often used to extract words.	`rdd.flatMap(x => x.to(3))`	`{1, 2, 3, 2, 3, 3, 3}`
`filter()`	Return an RDD consisting of only elements that pass the condition passed to `filter()`.	`rdd.filter(x => x != 1)`	`{2, 3, 3}`
`distinct()`	Remove duplicates.	`rdd.distinct()`	`{1, 2, 3}`
`sample(withReplacement, fraction, [seed])`	Sample an RDD, with or without replacement.	`rdd.sample(false, 0.5)`	Nondeterministic

Actions on RDDs

- RDD: {1,2,3,3}
- `collect`() -gets the data to a single node, use before printing all elements of an RDD
- `count`() - returns number of elements 4
- `countByValue`() - returns number of times each element appears in the RDD {1->1,2=>1,3->2)}
- `take`(2) - returns the first two elements {1,2}
- `top`(2) - returns top two elements {3,3}
- `takeOrdered`(n)(ordering): returns first `n` elements based on ordering.
- `reduce`({(x,y)=> x+y}) - finds the sum of all elements, returns 9
- `fold`(0)({ (x,y)=>x+y}) - same as reduce, but with initial value, returns 9

keyBy

- Convers an RDD of elements to an RDD of (k,v) tuples.
- Applies a function to each element of the RDD.
- The result of the function becomes the key and the original data item becomes the value of the newly created tuple.

```
rdd = sc.parallelize(['John', 'Fred', 'Anna', 'James'])
rdd.keyBy(lambda w: w[0]).collect()
```

result

```
[('J', 'John'),
 ('F', 'Fred'),
 ('A', 'Anna'),
 ('J', 'James')]
```

Using aggregate

- Compute the average of a bunch of numbers.
- We want to compute both the sum of the numbers and their count.
- We will have two accumulators: for sum, and for count, initially, they are both 0.
- We will use aggregate, it is similar to fold, but produces two values.

```
seqOp = (lambda x, y: (x[0] + y, x[1] + 1))    # count
combOp = (lambda x, y: (x[0] + y[0], x[1] + y[1]))    # sum
sc.parallelize([1, 2, 3, 4]).aggregate((0, 0), seqOp, combOp)
```

The aggregate Operation

- This is a more general form of fold and reduce.
- It has similar semantics, but it does not require the result to be the same type as the input type.
- It traverses the elements in different partitions sequentially, using seqop to update the result, and then applies combop to results from different partitions.
- The implementation of this operation may operate on an arbitrary number of collection partitions, so combop may be invoked an arbitrary number of times.

reduceByKey

```
rdd = sc.parallelize([('panda', 0),('pink', 3),('pirate', 3),('panda', 1),('pink', 4)])
rdd.reduceByKey(lambda x, y: x + y).collect()
```

- Gathers all the objects with the same key.
- It merges all the values into a single value.
- The above program merges words together and adds up frequencies.

```
panda 0
pink 3
pirate 3
panda 1
pink 4
```

```
(panda,1)
(pirate,3)
(pink,7)
```

Summary

- RDD = Resilient Distributed Dataset
- Result is computed on demand.
- There are transformations and actions on RDDs.
- When an action is called, all the transformations are performed (usually pipelined).
- Use `persist` to save the result of an RDD if it will be used many times. Otherwise, it will be recomputed every time we need it.