W2

8. Reduction Operations

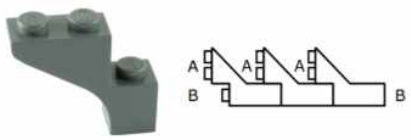
- 여기서도 latency의 의미를 아는 것이 중요하며 스칼라 컬렉션과 비슷해보이지만 내부적으로 다르다.

* Reduction Operation
- ex) fold, reduce, aggregate, foldLeft, reduceRight
- 컬렉션을 walk through 하면서 combine neighboring elemnents하여 single combined result를 낸다.

- 스칼라 컬렉션에서의 R.O.

Example:

```
case class Taco(kind: String, price: Double)

val tacoOrder =
  List(
    Taco("Carnitas", 2.25),
    Taco("Corn", 1.75),
    Taco("Barbacoa", 2.50),
    Taco("Chicken", 2.00))

val cost = tacoOrder.foldLeft(0.0)((sum, taco) => sum + taco.price)
```



* foldLeft는 병렬처리가 불가능하다.
- def foldLeft[B](z: B)(f: (B, A) => B): B

(z: B) 시작타입

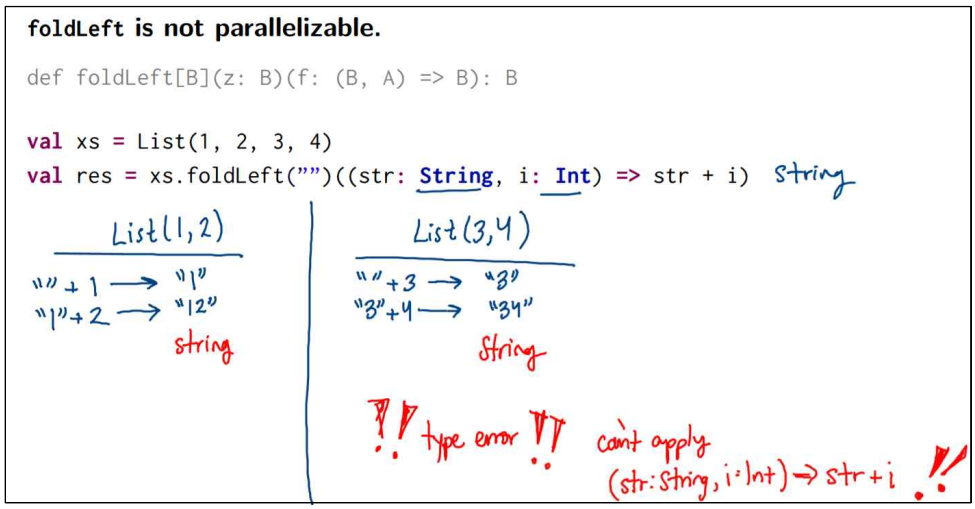
(f: (B, A) => B): B 시작 타입과 다른 것을 합쳐서 돌려주는 것인데 A와 B의 carry type이 달라서 병렬처리가 안된다.

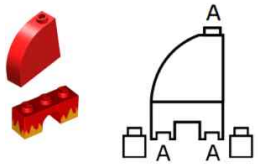
- Scala API 문서: foldLeft는 binary operator를 시작 값에서 컬렉션 또는 이터레이터의 모든 원소들을 왼쪽에서 오른쪽으로 sequentially 적용한다.

- "1234"로 string이 나오도록 병렬 처리가 "12", "34"를 합칠 때 에러가난다. str+i 에서

ex) val xs = List(1, 2, 3, 4)

val res = xs.foldleft("")((str: String, i: Int) => str + i)





* 반면에 fold는 병렬처리가 가능하지만 하나의 타입으로 제한한다.

def fold(z: A)(f: (A, A)=> A): A

* Aggregate는 seqop이나 combop를 활용해 좀 더 일반화 되어 병렬처리와 return typ들 변경하는 것 모두 가능
- 2 파라미터리스트에 3개의 파라미터가 있다. sequential operator는 foldleft처럼 2개의 type을 operate 한다. combination operator는 fold와 같이 하나의 type에서 operate한다.

- aggregate[B](z: => B)(seqop: (B, A)=> B, combop: (B, B) => B): B
seqop: (B, A)=> B 리스트의 element type은 A 였는데, 중간 결과값과 값이 B가 되도록(**:B**)

* 스칼라는 병렬처리가 안되도 되는 foldLeft, foldRight를 지원하지만 Spark에서는 뺐다.
- Scala Collections: fold, foldLeft/foldRight, reduce, aggregate
- Spark: fold, reduce, aggregate

- 스파크는 구현상으로 serially(sequantially) across a cluster하는 것이 synchronization이 많아지고, 많이 어려워 뺐다.

- 프로젝트를 할 때의 목적은 project down(줄이는) from larger/more complex data types.하는 것이 된다.
예를들어
case class WikipediaPage(title: String, redirectTitle: String, timestamp: String, lastContributorUsername: String, text: String) 일 때
- title과 timstamp만 신경쓰는 것이 text전체를 carry하는 것보다 시간과 메모리를 더 좋다.
그래서 aggregate를 쓰자?

9. Distributed Key-Value Pairs (Pair RDDs)
- Pair RDDs: a popular way of representing and organizing large quantities of data in practice 이다.

- 스칼라 컬렉션의 map에서는 겹치는 키가 없지만, key-value pair 에서는 key가 겹쳐도 된다.
- map, dict는 많은 언어에 있지만, 리스트와 같이 하나의 프로그램에서 주로 사용하는 컬렉션은 아니다.
- 하지만 Big Data processing에서는 key-value pair는 정말 많이 쓰인다.

- 맵리듀스 페이퍼에서 “We realized that most of our computations involved applying a map operation to each logical “record” in our input in order to compute a set of intermediate key/value pairs, and then applying a reduce operation to all the values that shared the same key, in order to combine the derived data appropriately.”
이말은 computations that they were already doing at Google always ended up producing key-value pairs. So, they focused the design of MapReduce around this common pattern for manipulating large amounts of data.

- 특정 column을 세우는 것을 key로 잡고, reduce operation을 할 수 있게 해주는 것을 groupby 결과값으로 본다? 26분 20초 쯤

- 대규모 데이터는 종종 헤아릴 수 없게 복잡하고 중첩된 데이터 레코드로 구성된다. 이러한 데이터에서는 key value pairs를 사용해 project down 는 것이 바람직하다.
- 오른쪽과 같은 구조를 갖는 json 파일에서는 아래와 같이 할 수 있다.

```
RDD[(String, Property)] // where 'String' is a key representing a city,
                        // and 'Property' is its corresponding value.

case class Property(street: String, city: String, state: String)
```

- 스파크에서는 key-value pairs는 "Pair RDDs"이며 실무에서도 많이 사용한다.

```scala
val rdd: RDD[WikipediaPage] = ...

// Has type: org.apache.spark.rdd.RDD[(String, String)]
val pairRdd = rdd.map(page => (page.title, page.text))
```

- 이런 메소드는 high-level api에도 key에대한 oper는 없어 특히 rbk, gbk는 많이 쓴다.

```scala
def groupByKey(): RDD[(K, Iterable[V])]
def reduceByKey(func: (V, V) => V): RDD[(K, V)]
def join[W](other: RDD[(K, W)]): RDD[(K, (V, W))]
```

```
{
   "definitions":{
      "firstname":"string",
      "lastname":"string",
      "address":{
         "type":"object",
         "properties":{
            "street_address":{
               "type":"string"
            },
            "city":{
               "type":"string"
            },
            "state":{
               "type":"string"
            }
         },
         "required":[
            "street_address",
            "city",
            "state"
         ]
      }
   }
}
```

10. Transformations and Actions on Pair RDDs

* Important operations defined on Pair RDDs: 자동으로 추가된다.
- Transformation: groupByKey, reduceByKey, mapValues, keys, join, left0uterJoin/right0uterJoin
- Action: countByKey

* groupByKey
- Recall groupBy from Scala collections.

def groupBy[KJ(f: A=> K): Map[K, Traversable[A]]

```scala
val ages = List(2, 52, 44, 23, 17, 14, 12, 82, 51, 64)
val grouped = ages.groupBy {age =>
  if (age >= 18 && age < 65) "adult"
  else if (age < 18) "child"
  else "senior"
}
```

adult, child 등을 key로 갖는 map을 만든다.

- def groupByKey(): RDD[(K, Iterable[V])]

```scala
case class Event(organizer: String, name: String, budget: Int)
val eventsRdd = sc.parallelize(...)
                  .map(event => (event.organizer, event.budget))

val groupedRdd = eventsRdd.groupByKey()
```

클래스 만들고 pairRDD만들고 group by key 때리면, transformation이라 아무것도 안함
action을 다음과 같이 때리면, **groupedRdd.collect().foreach(println)** 아래와 같은 결과가 나올 것임.

// (Prime Sound, CompactBuffer(42000))
// (Sportorg, CompactBuffer(23000, 12000, 1400))
// ...

- collect: forces that computation to take place.

* reduceByKey: groupbyKey와 reduce로 2단계의 operation으로 나누어 볼 수 있다.
- 하지만 나누어 하는 것보다 reduceByKey를 한번에 하는 것이 훨씬 효율적이다. (중요)

- **def reduceByKey(func: (V, V) => V): RDD[(K, V)]**
(func: (V, V) => V): 이부분을 보면, key는 안쓰고 value만 신경쓴다는 것을 알 수 있다. 이는 values들이 어떻게 이미 key로 groupby됐다고 생각하고 이제 컬렉션의 값들을 reduce하는 function을 적용한다고 생각하는 것이다.

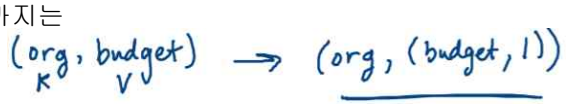
```
val budgetsRdd = eventsRdd.reduceByKey(_+_)

reducedRdd.collect().foreach(println)
// (Prime Sound,42000)
// (Sportorg,36400)
// (Innotech,320000)
// (Association Balélec,50000)
```

RBK도 transformation이다. 과제에도 써라

* mapValues: 나중에 말하겠지만, mapvalues는 pairRDD에 values에만 function을 적용한다.
- oftenly, pairRDD에 map을 하고 싶을 때 하고 싶을 것이며 key를 handle하는 function 구절에 넣어 map에 전달해야한다. 그리고 더 좋은 장점은 뒤에 말하겠다.

- **rdd.map { case (x, y): (x, func(y))}**
- def mapValues[U](f: (V) ⇒ U): RDD[(K, U)]

-ex) (budget, #events)를 val로 갖는 RDD 만들기
val intermediate = eventsRdd.mapValues (b => (b, 1)) // 여기까지는
오른쪽과 같이 만들어주는 것이고



val intermediate = eventsRdd.mapValues (b => (b, 1))
.reduceByKey ((v1, v2) => (v1._1 + v2._1, v1._2 + v2._2))
v1, v2모두 (budget, 1)인 값들이고
여기까지 하면, **RDD[(String, (Int, Int))]** value의 왼쪽은 예산총합, 오른쪽은 # of budget이 된다.

그리고 평균을 구하려면, value에만 focus하는 mapvalues를 활용해 아래처럼 한다.

```
val avgBudgets = intermediate.mapValues {
        case (budget, numberOfEvents) => budget / numberOfEvents
}
avgBudgets.collect().foreach(println)
```

물론 countByKey를 활용해 기릿할 수도 있다.

* countByKey: aciton 이며 간단하게 pairRDD의 key에 있는 element들의 수를 센다.

* keys: Transformation이며, Return an RDD with the keys of each tuple 한다.
- keys (def keys: RDD[K]), key가 많을 수도 있어 transf. 이다.

* 다른 pairRDD function은 아래를 참고하자.
http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.rdd.PairRDDFunctions

11. JOIN

- PairRDD에서의 join을 볼 것이다.

- inner, left outer, rigt outer

* join: Inner Join


```
val abos = ... // RDD[(Int, (String, Abonnement))J
val locations= ... // RDD[(Int, String)]
val trackedCustomers = abos.join(locations)
// trackedCustomers: RDD[(Int, ((String, Abonnement), String))] , 어느 쪽으로 하던 상관없다.
```

- groupBy, groupByKey

분산데이터에서는 shuffle을 사용하며 shuffle 은 latency 비용이 많이 크니 잘 써야한다.

<summary>

* foldLeft 와 aggregate 둘다 inputType과 outputType이 다른데 왜 aggregate 만 병렬 처리가 가능한가.
- foldLeft에서 데이터를 나누고 처리한 후 합칠 대 type 에러가 나는 부분을 aggregate는 seqop(sequential operator)로 foldleft의 처리를 수행한 후, fold가 수행하는(병렬처리가 가능한) combop(combination operator)을 통해 합쳐주는 연산을 수행하기 때문에 aggregate는 병렬처리가 가능하다.
- foldLeft의 경우 병렬 처리한 결과를 합칠 수 없는데. aggregate의 경우 병렬 처리의 결과를 합칠 수 있는 function을 combo op을 따로 주어서 병렬처리를 가능하게 한다.

* pairRDD는 어떤 데이터 구조에 적합하고 pairRDD는 어떻게 만드는가.
- 데이터 레코드들이 헤아릴 수 없게 중첩되어 있는 큰 데이터에 적합하다. pairRDD를 활해 project down하는 것이 좋으며, RDD를 읽고 map함수를 활용해 다음과 같이 만든다.
val pirRdd = rdd.map(page => (page.title, page.text))
- Pair RDD는 같은 key로 묶어지는 데이터를 효율적으로 처리 할 수 있게해줍니다. pairRDD는 RDD의 map을 통해 각 row를 Tuple로 만들어주면 자동으로 RDD 타입이 PairRDD로 전환된다.

* groupByKey()와 mapValues()를 통해 나온 결과를 reduce()를 사용해서도 똑같이 만들 수 있지만 reduce를 쓰는 것이 더 효율적인 이유는 무엇인가
- groupByKey는 데이터를 단순화하기 전에 노드간에 데이터 통신이 (shuffle이) 일어난다 .
(같은 key를 가진 데이터들은 한 node에 몰아줘야하기 때문에).
reduce를 사용하면 데이터 구조를 단순화 하여 shuffle하기 때문에 더 효율적이다.
실제로 reduce로 진행되어 project down 된 데이터를 워커노드가 교환할 경우 네트워크 비용이 그렇지 않을 경우에 비해 현저히 감소한다.

* join 과 leftOuterJoin, rightOuterJoin이 어떻게 다른가
- join은 inner join을 의미하며, 두 pair RDD에 key가 없는 행들은 빼고 합친 pair RDD를 만든다. outer join은 두 RDD중 한 곳에 Key가 없어도 추가하며, 오른쪽의 key가 없어도 왼쪽이 있다면 추가하는 것이 leftOuterJoin, 반대는 rightOuterJoin이라 하며, 둘 다 포함하는 것은 full outer join이라 한다.

* Shuffling은 무엇이며 이것은 어떤 distributed data paraellism의 성능에 어떤 영향을 가져오는가.
- 데이터가 노드에서 노드로 key별로 그룹되어 움직이는 것을 셔플이라하며, 이는 Latency에 영향을 미친다.
- 스파크가 클러스터에서, 클러스터의 물리적 머신에 존재하는 로우의 집합인 파티션을 교환하는 것을 셔플이라 한다. 넓은 트랜스포메이션을 의미하며, 셔플링을 최적화해야 distributed data paraellism의 성능을 높일 수 있다.
- RDD가 어떠한 연산을 통해 재구성될 때, 네트워크를 통해 다른 서버로 이동하는 것. Shuffling 연산이 자주 일어나게 되면 분산 처리 작업의 전체적인 속도가 느려질 수 있음.