CS5560 Knowledge Discovery and Management

Spark MapReduce Programing

Problem Set (PS-2B)
6/12/2017

Class ID: Yalamanchili Sowmya

Name: 30

Spark MapReduce Programming – Calculate everyone's common friends for Facebook

Facebook has a list of friends (note that friends are a bi-directional thing on Facebook. If I'm your friend, you're mine). They also have lots of disk space and they serve hundreds of millions of requests everyday. They've decided to pre-compute calculations when they can to reduce the processing time of requests. One common processing request is the "You and Joe have 230 friends in common" feature. When you visit someone's profile, you see a list of friends that you have in common. We're going to use MapReduce so that we can calculate everyone's common friends once a day and store those results. Later on it's just a quick lookup. We've got lots of disk, it's cheap.

1) Draw a MapReduce diagram similar to the word count diagram below.
2) Sketch a MapReduce algorithm for the common Facebook friends (referring to the word count code below).
3) Sketch Spark Scala implementation (referring to the word count code below).

Example

Assume the friends are stored as Person->[List of Friends], our friends list is then:

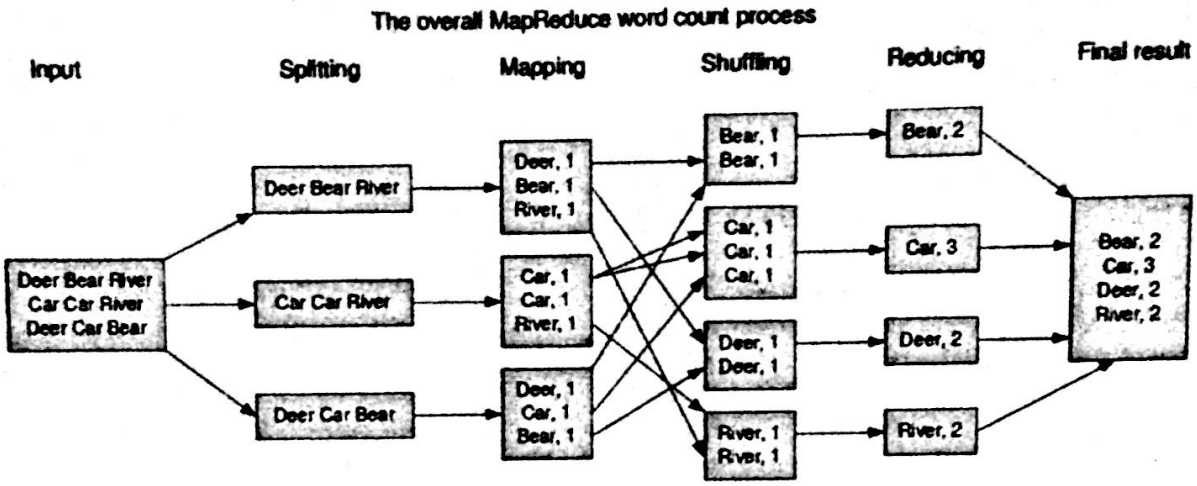
A -> B C D
B -> A C D E
C -> A B D E
D -> A B C E
E -> B C D

The result after reduction is:

(A B) -> (C D)
(A C) -> (B D)
(A D) -> (B C)
(B C) -> (A D E)
(B D) -> (A C E)
(B E) -> (C D)
(C D) -> (A B E)
(C E) -> (B D)
(D E) -> (B C)

Now when D visits B's profile, we can quickly look up (B D) and see that they have three friends in common, (A C E).

WORD COUNT EXAMPLE



Algorithm 2.1 Word count

The mapper emits an intermediate key-value pair for each word in a document. The reducer sums up all counts for each word.

```
1: class MAPPER
2:     method MAP(docid a, doc d)
3:         for all term t ∈ doc d do
4:             EMIT(term t, count 1)

1: class REDUCER
2:     method REDUCE(term t, counts [c1, c2, ...])
3:         sum ← 0
4:         for all count c ∈ counts [c1, c2, ...] do
5:             sum ← sum + c
6:         EMIT(term t, count sum)
```

MapReduce Scala Code for WordCount

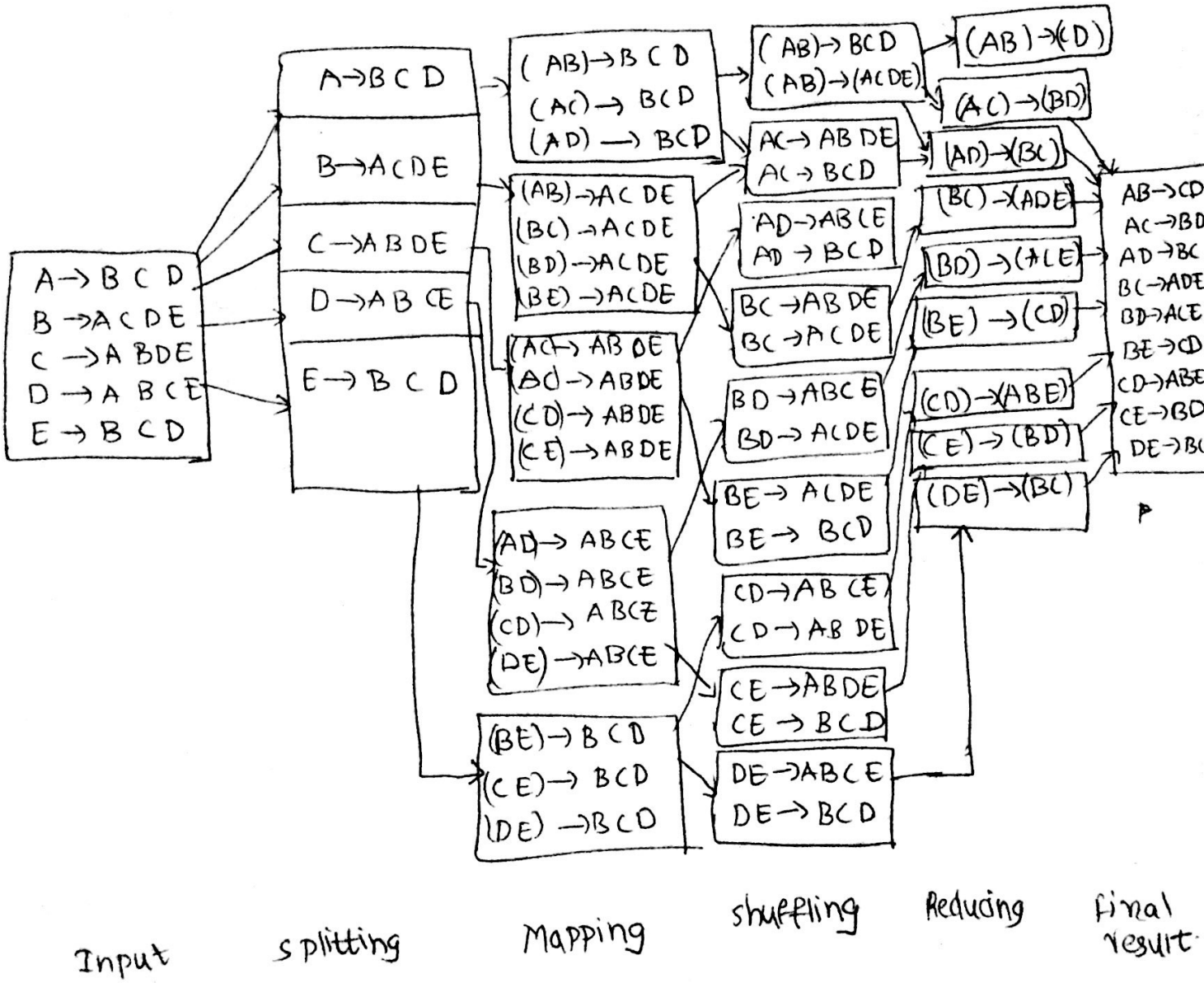
```
// This class performs the map operation, translating raw input into the key-value
// pairs we will feed into our reduce operation.
class TokenizerMapper extends Mapper[Object,Text,Text,IntWritable] {
  val one = new IntWritable(1)
  val word = new Text

  override
  def map(key:Object, value:Text, context:Mapper[Object,Text,Text,IntWritable]#Context) = {
    for (t <-  value.toString().split("\\s")) {
      word.set(t)
      context.write(word, one)
    }
  }
}

// This class performs the reduce operation, iterating over the key-value pairs
// produced by our map operation to produce a result. In this case we just
// calculate a simple total for each word seen
class IntSumReducer extends Reducer[Text,IntWritable,Text,IntWritable] {
  override
  def reduce(key:Text, values:java.lang.Iterable[IntWritable], context:Reducer[Text,IntWritable,Text,IntWritable]#Context) = {
    val sum = values.foldLeft(0) { (t,i) => t + i.get }
    context.write(key, new IntWritable(sum))
  }
}
```

Spark Scala Code for WordCount

```
val textFile = spark.textFile("hdfs://...")
val counts = textFile.flatMap(line => line.split(" "))
                 .map(word => (word, 1))
                 .reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://...")
```

flatMap(func)	Similar to map, but each input item can be mapped to 0 or more output items (so func should return a Seq rather than a single item).
reduceByKey(*func*, [*numTasks*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function *func*, which must be of type (V,V) => V. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument.

The overall MapReduce process



(2) MapReduce algorithm

The MapReduce solution to find "common friends" has a map() and reduce() functions. The mapper accepts a $(key_1, value_1)$ pair, where key_1 is a person and $value_1$ is a list of the associated friends of this person. The mapper emits a set of new $(key_2, value_2)$ pairs where key_2 is a tuple $(key_1, friend_i)$ where $friend_i \in value_1$ and $value_2$ is the same as $value_1$ (list of all friends for key_1). The reducer's key is a pair of two users $(user_j, user_k)$ and value is a list of sets of friends. The reduce() function will intersect all sets of the friends to find common and mutual friends for $(user_j, user_k)$ pair.

Here are the map() and reduce() functions:

map() Function

```
// key is the person
// value is a list of friends for this key=person
// value = (<friend,1><friend,2>... <friend, N>)
map(key, value){
  reducerValue = (<friend_1) <friend-2> ... <friend_N>);
foreach friend in (<friend-1><friend_2> ... <friend_N>) {
    reducerkey= buildSortedkey (person, friend);
    emit (reducerkey, reducerValue);
}}
```

Mapper's output keys are sorted and this property will prevent

Name:- Yalamanchili Soumya
ClassID:- 30

duplicate keys:

common friends buildSortedKey() function:-

```
Tuple buildSortedKey(person1, person2) {
  if (person1 < person2) {
    return Tuple2(person1, person2)
  }
  else {
    return Tuple2(person2, person1)
  }
}
```

The reduce() function finds the Algorithm for common friends for every pair of users by intersecting all associated friends in between

---

(c) spark scala implementation

contents of /tmp/data.txt

A → B C D
B → A C D E
C → A B D E
D → A B C E
E → B C D

```
def pairMapper(line : String) = {
    val words = line.split(" ")
    Val key = words(0)
    val pairs = words.slice(1, word.size).map(friend => {
      if (key < friend) (key, friend) else (friend, key)
    })
  pairs.map(pair => (pair, words.slice(1, words.size).toSet))
}
```

```
def pairReducer(accumulator: Set[String], set: Set[String]) = {
    accumulator intersect set
}
val data = sc.textFile("file:///tmp/data.txt")
val results = data.flatMap(pairMapper)
    .reduceByKey(pairReducer)
    .filter(_._2.isEmpty)
    .sortByKey()
results.collect.foreach(line => {
  println(s"${line._1} ${line._2.mkString(" ")}")
})
```