An Introduction to group/ join Type-safe API in Scalding

Sean Chang[1]

Introduction of scalding & goals

In this tutorial, I will demonstrate how to use group & join tools in the Typed-safe API in Scalding- a powerful library combining Scala and Cascading which simplifies coding in MapReduce and prevents runtime types errors. Many interesting Scalding examples in big data can be seen online such as movie recommendation, semantics analysis and portfolio selection. In particular, I will focus on the following e-commerce example: imagine a considerable number of merchandise items is presented on ebay and the price and the category of each item are saved in two separate files. In addition, every transaction has been recorded - a document containing item ID and buyerID. After merging these three documents, our main goal is to find out who are the top three customers in each category. This sorting problem is simplified by MapReduce techniques. Essentially, map function rearranges entries so that those entries representing the same user in the same category are listed adjacently; reduce function computes the total expense of each buyer by summing up these adjacent entries and reduces the complicated transactions to those quantities that we are concern.

Prerequisites

The prerequisites for running Scalding APIs are Jave JDK, Scala and sbt (simple build tools). Jave JDK can be downloaded from the office website. For Mac folks, an easy way to install the latest Scala and sbt is using Homebrew: run the following script in the terminal:

```
ruby -e "$(curl -fsSL https://raw.github.com/Homebrew/homebrew/go/install)"
```

and

```
brew install scala
brew install sbt
```

Now we are ready to go! All examples in this tutorial are reproducible. The setting environment is based on the github scalding-tutorial (which runs scalding on Hadoop without *scald.rb*). For detailed implementations of sample codes and input/ output format,

[1] PhD candidate in Statistical Science, Duke University. Email sean.chang@duke.edu

see the *Demonstration* chapter.

Typed Safe APIs

There Scalding Typed-safe api is composed of two fundamental elements: *TypedPipe[T]* , *KeyedList[K,T]* . In addition, a very common *Grouped[K,V]* is a subclass *of KeyedList[K,T].*

a. TypedPipe[T]

A typedpipe can be think as a particular transformation of its input data, or a data frame. For example, we can create a TypedPipe to store all drivers' names and their car makers and made years in Durham, N.C.:

```
case class Car(owner: String, Maker: String, Made: Int)
val durham_drivers : TypedPipe[Car]=TypedPipe.from(TypedTsv[(String,String, Int)]("someinput.tsv"))
```

We can also create a TypedPipe directly without define a case class in advance:

```
val patient_info: TypedPipe[(String,String ,Double)]  //(patientID, treatment, dose)
                  =TypedPipe.from(TypedTsv[(String,String, Double)]("somename.tsv"))
```

Same as other powerful Scalding apis, TypedPipes can store texts as well:

```
val bible: TypedPipe[String]= TypedPipe.from(TextLines(bible.txt))
```

To summarize, TypedPipes can be used either repeatedly by defining a case class beforehand or directly with the declaration of the data types. Also, a TypedPipe can be exported conveniently by .write(TypedTsv[type]("output")):

```
case class CompanyDirectory(name: String, address: String, phone: Int)
val GOOGLE: TypedPipe[CompanyDirectory]=TypedPipe.from(TypedTsv[(String,String,Int)]("input"))
val Tweeter //another TypedPipe[CompanyDirectory]
    .write(TypedTsv[(String,String,Double)](sampleoutput.tsv))
```

b. *Grouped[K,V] : GroupBy & GroupAll/*

Grouped[K,V] represents a categorized lists of items based on the key information in a TypedPipe V. To be more specific, suppose we have a TypedPipe which is composed of itemID, price, category and buyerID. We might be interested in answering: which is the largest category in terms of volume of trade? Who are the top buyers among all categories? Therefore, grouping each entry by some shared quantity would be useful. We can group by category, our key, then those entries have the same key will be in the same cluster. This data structure is Grouped[K,V] and compute the volume of trade in each group becomes pretty straightforward- sum of all spending in each category. Or group by users and sort by total expenses, return a list of top buyers. The following codes demonstrate how to use groupBy in Scalding:

```
val transcations: TypedPipe[(String, Double, String, String)]=get_pipes  //(itemID,price,category,buyerID)
transcations.groupBy{_._3}
//group by category, resulting a Grouped[String,  (String, Double, String,String)]
//creates                        Grouped[Key=category, V=  original TypedPipe    ]
```

And we can compute interesting quantities within each group:

```
transcations.mapValues{_._3}.sum //compute volume of trade in each category
```

Similarly, we can group two objects simultaneously:

```
val book: TypedPipe[(String, String, String)]=getbook_pipe //(bookName, subject , level)
    book.groupBy{y:(String, String, String)=>(y._2,y_3)}
/* only books in the same level and subject will be in the same category
  ((stat, graduate), (Machine learning, stat, graduate))
  ((stat, graduate), Statistical Inference, stat,  graduate)
  ((econ, undergraduate), Intro to Economics,econ, undergraduate)
  ((phys, undergraduate), Classical Mechanics,phys,undergraduate)*/
```

GroupAll is useful when we want to calculate the overall mean, average, max, min or the number of entries:

```
val mathpipe: TypedPipe[(String, Double)]=get_mathpipe // (name, math_score)
    mathpipe.groupAll                                  //Grouped[Unit, (String, Double)]
            .mapValues{_._2}.max
```

GroupAll =Grouped[unit, V] representing all entries are in the same group and the unit can be removed by group.values.

Here are some other useful functions for transferring Grouped to TypedPipe:

```
val group: Grouped[K, V]
group.keys      //get TypedPipe[K]
group.values    //get TypedPipe[V]
group.mapValues{values:V => mapingFuncion(values)} //TypedPipe[K,V']
```

c. ***KeyedList[K,T] : Join***

Same as Grouped[K,T], KeyedList[K,T] is also a sharded lists of items but usually its members are TypedPipes and key is the sharing informaiton among these Typepipes

For example, if the first typedpipe has students' names, & math scores; and the second typedpipe has student' names & verbal scores, we can merge these two based on names:

```
val verbalpipe: TypedPipe[(String, Double)]=get_verbal_pipe // (name, verbal_score)
val merge = dat1.groupBy(_._1).join(tsv2.groupBy(_._1))  //merge dat1 & dat2 by names
 //  KeyedList[String, ((String, Double),(String, Double))]
 //           [name , ((name, math score),(name, verbal score))]
```

Merge Grouped[K,V], Grouped[K,W] resulting a KeyedList[K, (V,W)]. In this example,

```
dat1.groupBy(_._1) : a Grouped[String, (String, Double)]
dat2.groupBy(_._1) : a Grouped[String, (String, Double)]
merge : a KeyedList[String, ((String, Double),(String, Double))]
```

Most importantly, a KeyedList can be transferred to a TypedPipe easily by toTypedPipe (in the TDsl._ package) :

```
merge.toTypedPipe                               //KeyedInList->TypedPipe
     .map{case (x,((_,y),(_,z)))=>(x,y,z)}  //(name, math score, verbal score)
                                                //TypedPipe(String, Double, Double)
```

Be aware that *join* is an inner join, that means it only keeps all rows or entries that appear in both pipes, if we have a students' math but not verbal score, then this student' name will not appear after join two TypedPipes. Scalding offers other join tools: left / rightJoin which keeps all rows/ entries from the left/ right pipe, and outerJoin, which keeps all entries that originate either from left or from right pipe. Furthermore, an null will be assigned if there's no match fields.

```
leftJoin: leftJoin[W](smaller : TypedPipe[(K, W)]): KeyedList[K, (V, Option[W])]
rightJoin: rightJoin[W](smaller : TypedPipe[(K, W)]): KeyedList[K, (Option[V], W)]
outerJoin: outerJoin[W](smaller : TypedPipe[(K, W)]): KeyedList[K, (Option[V], Option[W])]
```

The output formats of left/ right and outerJoin can been seen above. And in Scalding, always put the smaller group on the right for computational reasons.

Demonstration

In this section, I demonstrate how to implement the sample codes (ebayapi.scala) step by step, some steps are optional and may vary with different machines.

First, clone the github scalding-tutorial and run:

```
sbt update
sbt test
sbt assembly
```

Second, save *ebayapi.scala* with other tutorials (~/src/main/scala/tutorial) and three input files must be in *~/data* and .tsv (tab separated values files, entries are separated by "tab"). In addition, each file contains (item_id & price) , (item_id category) and (item_id & buyer_id) respectively. (see snapshots below).

TSV1.tsv

itemID_1	100
itemID_2	350
itemID_3	20
itemID_4	899.05
itemID_5	28.88
itemID_6	15
itemID_7	7
itemID_8	299
itemID_9	38
itemID_10	383

TSV2.tsv

itemID_1	motors
itemID_2	fashion
itemID_3	motors
itemID_5	electronics
itemID_6	motors
itemID_7	fashion
itemID_9	motors
itemID_11	electronics

TSV3.tsv

itemID_2	buyerID_2
itemID_4	buyerID_4
itemID_5	buyerID_3
itemID_6	buyerID_2
itemID_7	buyerID_1
itemID_8	buyerID_1
itemID_9	buyerID_3
itemID_10	buyerID_3

Sample inputs: TSV1.tsv,TSV2.tsv,TSV3.tsv

Remark: Scalding also supports inputs in .csv or .txt by using Csv or TextLine.
Now execute the following codes in the terminal:

```
sbt assembly
hadoop jar target/scalding-tutorial-0.8.11.jar ebayapi_merge
hadoop jar target/scalding-tutorial-0.8.11.jar ebayapi_sort
```

"ebayapi_merge" reates the merged data frame (*TSV4.tsv)* having itemID, category, price and buyerID. Since we only curious about who are the top buyers, those items not appear in the list of transaction (*TSV3.tsv*) will be discarded. "ebayapi_sort" computes the total expense of each buyer in each category (including no category), sorting and generates *TopBuyers.tsv* in *~/myoutput/* which contains top 3 buyers in each category and the amount of money these buyers have spend.

TopBuyers.tsv

```
no category buyerID_4   902.49
no category buyerID_3   617.2
no category buyerID_1   345.0
motors  buyerID_4   78.4
motors  buyerID_3   76.0
motors  buyerID_2   30.3
fashion buyerID_2   641.79
fashion buyerID_1   171.0
electronics buyerID_3   75.32
electronics buyerID_5   23.79
electronics buyerID_7   0.99
```

Sample output: TopBuyers.tsv

Summary

In this tutorial we have seen many typed api examples, readers might be familiar with group & join functions in Scalding now. Typed safe api in Scalding is relatively new and is thriving due to its modern, concise and easy to implement features. With the advancement of distributed computing, I believe many robust and scalable algorithms will be developed with cutting-edge Bayesian and machine learning methods, and Undoubtedly, data scientists play a decisive role in technology innovation by unveiling hidden patterns behind big data.

References

1. *https://github.com/Cascading/scalding-tutorial*
2. *http://twitter.github.io/scala_school/*
3. *https://github.com/twitter/scalding*
4. *http://www.tutorialspoint.com/scala/*
5. *https://github.com/ThinkBigAnalytics/scalding-workshop*
6. *https://groups.google.com/forum/#!forum/cascading-user*
7. *White, T. (2012). Hadoop: the definitive guide. O'Reilly.*

Appendix

Codes: ebayapi.scala

```scala
import com.twitter.scalding._
import TDsl._

class ebayapi_join(args: Args) extends Job(args) {

val tsv1: TypedPipe[(String, Double)] = TypedTsv[(String, Double)]("data/TSV1.tsv")  //(itemID, price)
val tsv2: TypedPipe[(String, String)] = TypedTsv[(String, String)]("data/TSV2.tsv")  //(itemID, category)
val tsv3: TypedPipe[(String, String)] = TypedTsv[(String, String)]("data/TSV3.tsv")  //(itemID, buyerID)

val joined= tsv1.groupBy(_._1).outerJoin(tsv2.groupBy(_._1))               // merge TSV1 & TSV2 by its itemID : KeyedList[String, (
           .toTypedPipe                                                    //Keyedlist-> TypedPipe  Keyedlist(itemID, (Option[(item
           .map { kvw : (String,((Option[(String, AnyVal)],Option[(String, String)])))
                       => (kvw._1, kvw._2._1.getOrElse(("no itemID",0))._2, kvw._2._2.getOrElse(("no itemID","no category"))._2)}
           .write(TypedTsv[(String,AnyVal,String)]("myoutput/merged.tsv")) //AnyVal appears since "getOrElse", however, it is alway
           .groupBy(_._1).join(tsv3.groupBy(_._1))                            //merge with TSV3 by itemID: KeyedList[String, ((Stri
           .toTypedPipe
           .map{case (x,((_,y2,y3), (_,z2))) => (x,y2,y3,z2)}                 //TypedPipe[(String, Double, String, String)]
           .write(TypedTsv[(String, AnyVal, String, String)]("data/TSV4.tsv"))
 }

class ebayapi_sort(args: Args) extends Job(args) {

val temp: TypedPipe[(String, Double, String, String )] = TypedTsv[(String, Double, String, String)]("data/TSV4.tsv")

  temp.groupBy{y: (String, Double, String, String)=> (y._3,y._4)}
      .mapValues{_._2}.sum                      //  calculate total expense of each buyer in each categoryTypedPipe((String,Strin
      .map{case ((x, y), z)  => (x,y,z)}
      .groupBy(_._1)                            //group by category; Grouped
      .sortBy(_._3).reverse.take(3)             //sort by total expense
      .toTypedPipe                              // ((String, String), ((String, String), Double))
      .map{case (x,(_,y,z)) => (x,y,z)}
      .write(TypedTsv[(String, String, Double)]("myoutput/TopBuyers.tsv"))
}
```