Здравейте колеги,

Извиняваме се за просроченото време на лекцията от предния път - 11 Април. Като последица от това, упражнението беше по-кракто. Освен това, то беше и несъвсем ефективно, може би защото липсваше стъпка-по-стъпка описание на задачите. Препоръчваме ви да преглеждате серията от PDF документи, които описват заниманията ни по време на упражненията. Например, първият такъв документ, от 4 Април, предлага относително ясни детайли по работата с нашия импровизиран VMware image environment, сред които повечето стъпки са общи за няколко упражнения, и е добре да сте наясно с тях.

Благодарим на колегите, които откриха причините предишният WordCount пример да се изпълнява неправилно - т.е. да връща за всяка дума стойност от 1 :)

Благодарим и на онези от вас, които предложиха оптимизации по процеса на провеждане на упражненията, предимно насочени към това да се повиши partition tolerance-а на групата от присъстващите :).

Ето как протекоха заниманията ни от 11 Април.

Заредихме Fedora image-ите си и стартирахме Hadoop клъстера.

Заредихме в Eclipse, и изпълнихме 'истински' работещ WordCount пример за MapReduce job. Кодът WordCount.java взехме от примерите, които вървят с Hadoop инсталацията - те се намират в директория

Desktop\Installation\hadoop-0.20.2\src\examples\org\apache\hadoop\examples.

За да го пригодим за работа върху HDFS, без да се налага да подаваме аргументи от командния ред на main() метода, променихме следните два реда:

```
FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
```

```
FileOutputFormat.setOutputPath(job, new Path(otherArgs[1])); на
```

```
FileInputFormat.addInputPath(job, new Path("In"));
```

```
FileOutputFormat.setOutputPath(job, new Path("Out"));
```

За да имаме успешна компилация пък, трябваше да добавим този клас в MapReduce проект в Eclipse: или в съществуващ вече, или в нов такъв.

Напомняме: изпълнението *context menu -> Run As -> Run on Hadoop* и селектирането след това на конфигурирания вече Hadoop клъстер на localhost, изпълнява MapReduce job-а **спрямо HDFS**, а **не спрямо файловата система на Fedora**! Следователно, In и Out в случая трябва да са имена на директории в HDFS. Ако вече имате директория с име Out, ще трябва да я изтриете, иначе job-ът ви ще fail-не. Ако пък искате да тествате изпълнение върху локалната файлова система на Fedora, пробвайте с *context menu -> Run As -> Run as Java application*.

Поглеждайки в сорс кода на WordCount.java, коментирахме ползваните типове от Java API-то на Hadoop (което е доста нестабилно и се мени от версия на версия). Ето какво споменахме.

Както стана дума вече, една MapReduce програма обработва данните, структурирани като (ключ, стойност) двойки, и има следния общ вид:
map: (K1,V1) →list(K2,V2)
reduce: (K2,list(V2)) →list(K3,V3)

Ето как изглеждат отделните части на един MapReduce job в случая на Hadoop, заедно с това кой е отговорен за тях:

Part	Handled By
Configuration of the job	User
Input splitting and distribution	Hadoop framework
Start of the individual map tasks with their input split	Hadoop framework
Map function, called once for each input key/value pair	User
Shuffle, which partitions and sorts the per-map output	Hadoop framework
Sort, which merge sorts the shuffle output for each partition of all map outputs	Hadoop framework
Start of the individual reduce tasks, with their input partition	Hadoop framework
Reduce function, which is called once for each unique input key, with all of the input values that share that key	User
Collection of the output and storage in the configured job output directory, in *N* parts, where *N* is the number of reduce tasks	Hadoop framework

Ключове и стойности

Въпреки сериозното ползване на понятията 'ключ' и 'стойност', когато говорим за MapReduce, досега не сме споменавали нищо за техните типове. MapReduce имплементацията на Hadoop не позволява произволни Java типове да се ползват за ключове и стойности. Например, въпреки че често даваме пример за ключове и стойности като integer или string, съответните стандартни Java класове не биха ни свършили работа в Hadoop. Това е защото MapReduce framework-ът има специфичен начин за сериализация на ключ/стойност двойки, позволяващ оптималното им придвижване между елементите на Hadoop клъстера, и само класове, които поддържат такъв тип сериализация, могат да служат за ключове и стойности в този framework.

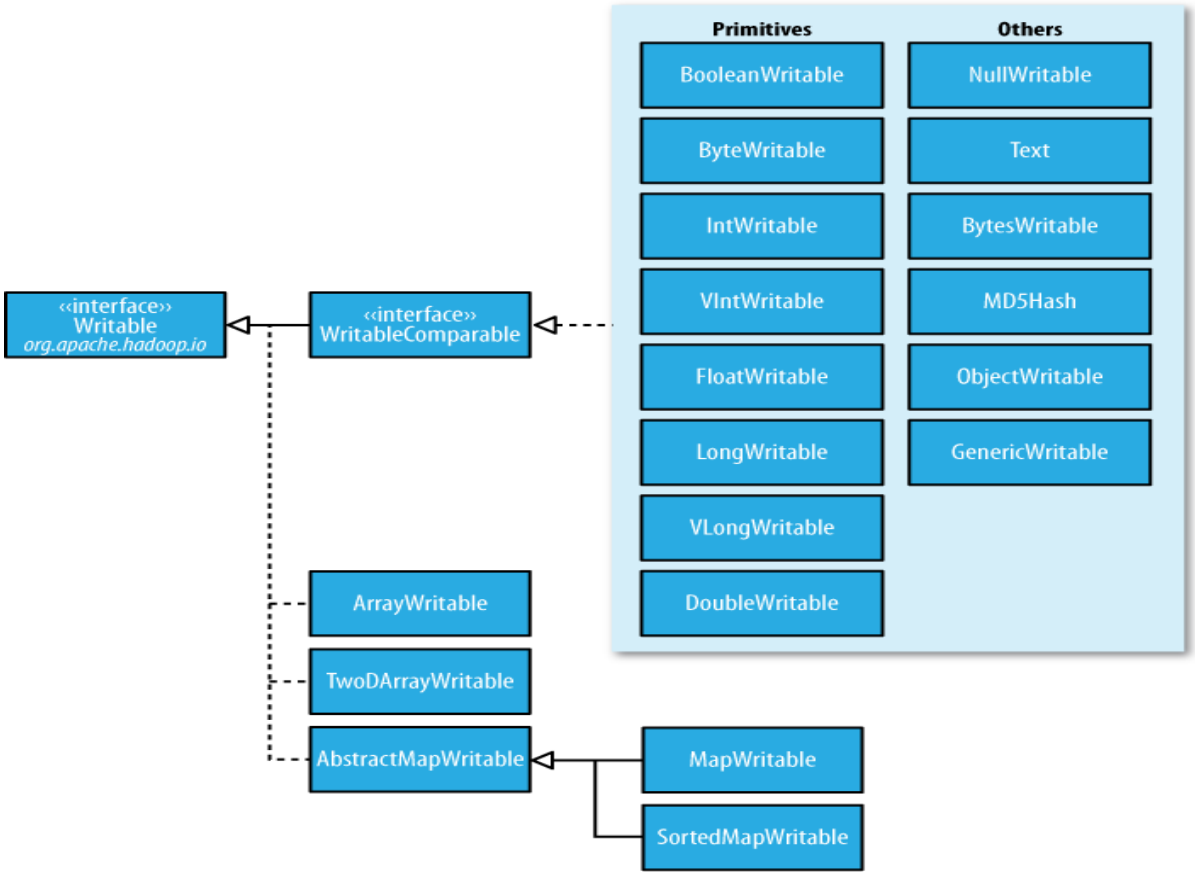
По-конкретно: класове, които имплементират `Writable` интерфейса, могат да са стойности, и класове, които имплементират `WritableComparable<T>` интерфейса, могат да са ключове или стойности. Интерфейсът `WritableComparable<T>` е комбинация от

`java.lang.Comparable` и `Writable` интерфейсите. Имаме нужда от сравнимост, защото ключовете подлежат на сортиране по време на reduce фазата, докато стойностите просто се записват и четат.

Hadoop ни предоставя известно количество класове, които имплементират `WritableComparable`, в това число и wrapper-и на базовите Java типове (такива се ползват и от WordCount примера):

Class	Description
`BooleanWritable`	Wrapper for a standard Boolean variable
`ByteWritable`	Wrapper for a single byte
`DoubleWritable`	Wrapper for a Double
`FloatWritable`	Wrapper for a Float
`IntWritable`	Wrapper for a Integer
`LongWritable`	Wrapper for a Long
`Text`	Wrapper to store text using the UTF8 format
`NullWritable`	Placeholder when the key or value is not needed

Ето малко по-детайлно описание на йерархията на вградените `Writable` типове:



Естествено, ключове и стойности могат да бъдат и ваши custom типове. Ето един пример, който дефинира калс, описващ полет между два града:

```
public class Edge implements WritableComparable<Edge>{
      private String departureNode;
      private String arrivalNode;
      public String getDepartureNode() {
      return departureNode;
      }

      @Override
      public void readFields(DataInput in) throws IOException {
            departureNode = in.readUTF();
            arrivalNode = in.readUTF();
      }

      @Override
      public void write(DataOutput out) throws IOException {
            out.writeUTF(departureNode);
            out.writeUTF(arrivalNode);
      }

      @Override
      public int compareTo(Edge o) {
            return (departureNode.compareTo(o.departureNode) != 0)
                  ? departureNode.compareTo(o.departureNode)
                  : arrivalNode.compareTo(o.arrivalNode);
      }
}
```

Този клас имплементира `readFields()` и `write()` методите на `Writable` интерфейса. Те работят с `DataInput` и `DataOutput` класовете от HDFS API-то на Hadoop, за да сериализират състоянието на обектите от този клас.

Mapper-и

За да може да служи за mapper в Hadoop, един Java клас трябва да имплементира `Mapper` интерфейса, и да наследи `MapReduceBase` класа. `MapReduceBase` класът служи за базов клас както за mapper-и, така и за reducer-и. Той съдържа два метода, които по същество се ползват като конструктор и деструктор:

`void configure( JobConf job)` — Тук може да извлечете параметри, които могат да се задават или в някакъв конфигурационен XML файл, или в главната програма, която конфигурира вашия MapReduce job (aka вашият MapReduce driver). Този метод се извиква, преди обработката на данни да почне.

`void close ()` — Непосредствено преди да приключи map фазата, и нейният резултат да се подаде към reducer-ите, framework-ът извиква този метод, в който може да изчистите ползваните от вас ресурси - DB конекции, I/O stream-ове, и т.н.

`Mapper` интерфейсът контролира обработката на данни. Той ползва Java generics и така пълният му вид е `Mapper<K1,V1,K2,V2>`, където сте посочили ключ/стойност типовете, които са нужни за вашата map фаза (тези от входа, и тези от изхода). Единственият метод тук, е свързан с обработката на една ключ/стойност двойка (как точно Hadoop взема входа на MapReduce job-а, и от него генерира такива двойки, с които извиква вашия mapper, ще видим по-нататък) :

```
void map(K1 key, V1 value, OutputCollector<K2,V2> output, Reporter
reporter) throws IOException
```

В съглашение с общите MapReduce разбирания, този метод генерира (може и празен) списък от (K2, V2) двойки, за дадена (K1, V1) входна двойка. `OutputCollector` параметърът получава резултата от mapping процеса, а `Reporter`-ът ви дава възможност да записвате допълнителна информация, докато map задачата се изпълнява.
Hadoop върви с едно базово количество mapper-и, които може да ползвате за да 'стартирате' по-бързо с framework-а:

Class	Description
`IdentityMapper<K,V>`	Implements Mapper<K,V,K,V> and maps inputs directly to outputs
`InverseMapper<K,V>`	Implements Mapper<K,V,V,K> and reverses the key/value pair
`RegexMapper<K>`	Implements Mapper<K,Text,Text,LongWritable> and generates a (match, 1) pair for every regular expression match
`TokenCountMapper<K>`	Implements Mapper<K,Text,Text,LongWritable> and generates a (token, 1) pair when the input value is tokenized

Reducer-и

Тук API-то е аналогично. Както и всяка mapper имплементация, един reducer трябва първо да наследи `MapReduceBase` класа за своите конфигурационни и cleanup нужди. Освен това, той трябва да имплементира `Reducer` интерфейса, който съдържа следния единствен метод:

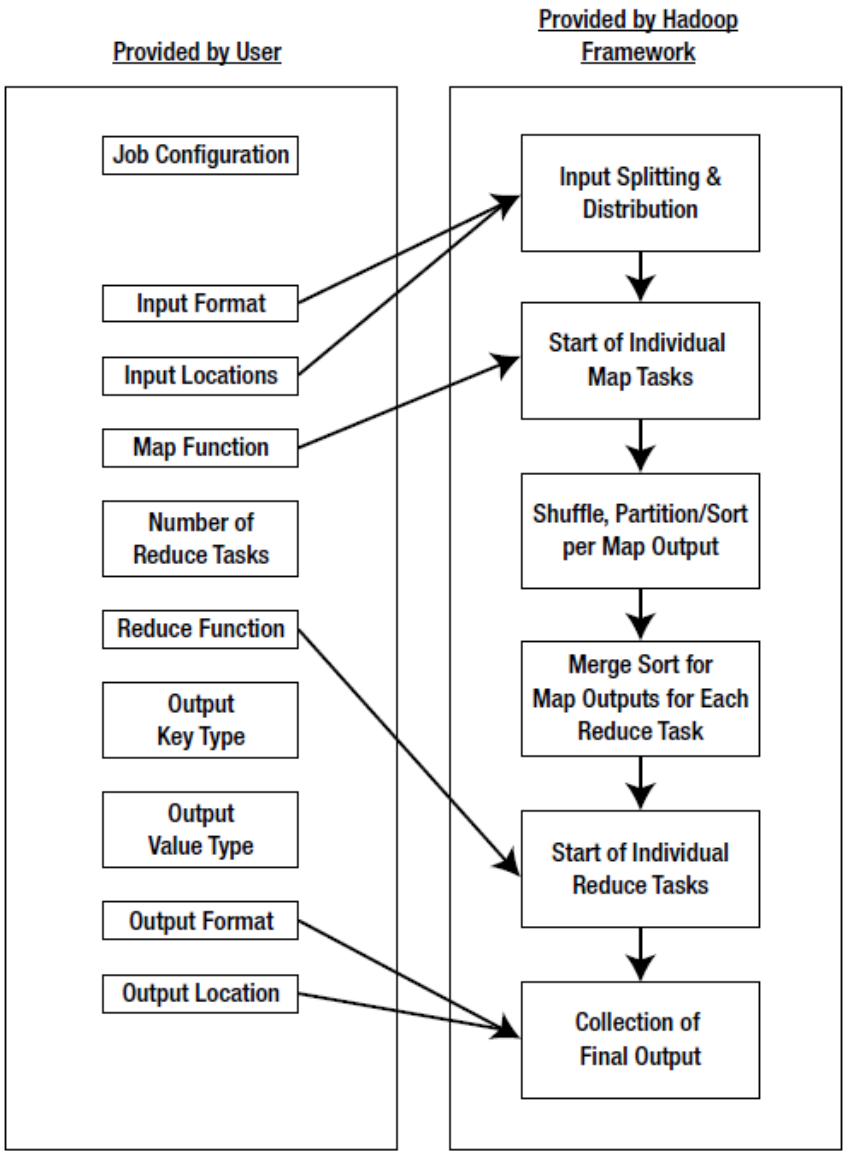
```
void reduce(K2 key, Iterator<V2> values, OutputCollector<K3,V3>
output, Reporter reporter) throws IOException
```

Когато reduce задачата получава изхода от различни mapper-и, тя сортира идващите данни по ключ, и групира всички стойности за даден ключ. След това framework-ът извиква reduce метода, който итерира стойностите за даден ключ, и генерира своят изход. `OutputCollector` обектът получава резултата от reduce процеса, и го записва в изходен файл. `Reporter`-ът ви дава възможност да записвате допълнителна информация, по време на работата на reduce процеса.
Ето някои базови reducer имплементации, идващи с Hadoop:

Class	Description
IdentityReducer<K,V>	Implements Reducer<K,V,K,V> and maps inputs directly to outputs
LongSumReducer<K>	Implements Reducer<K,LongWritable,K,LongWritable> and determines the sum of all values corresponding to the given key

Срещнахме по време на упражненията и понятието **driver**, което се ползва от Hadoop plug-in-а за Eclipse, и означава обикновен Java клас, в main() метода на който се конфигурира и изпълнява вашия MapReduce job.

Ето още веднъж MapReduce workflow-а на Hadoop:



HDFS

По отношение на HDFS файловата система, споменахме общия начин да се обръщате към нея от командния ред:

```
$>  hadoop fs -cmd <args>
```

където cmd е специфичната файлова команда, а <args> е списък от аргументи. Командата cmd обикновено носи името (и смисъла) на позната Unix команда. Например, командата за показване на списъка от файловете в дадена директория е:

```
$> hadoop fs -ls
```

На вашите Fedora image-и има създадена HDFS. Директорията по подразбиране на HDFS е с име /user/$USER, където $USER е името на потребителя, с който сте влезнали - в нашия случай, отново е user.

WordCount MapReduce

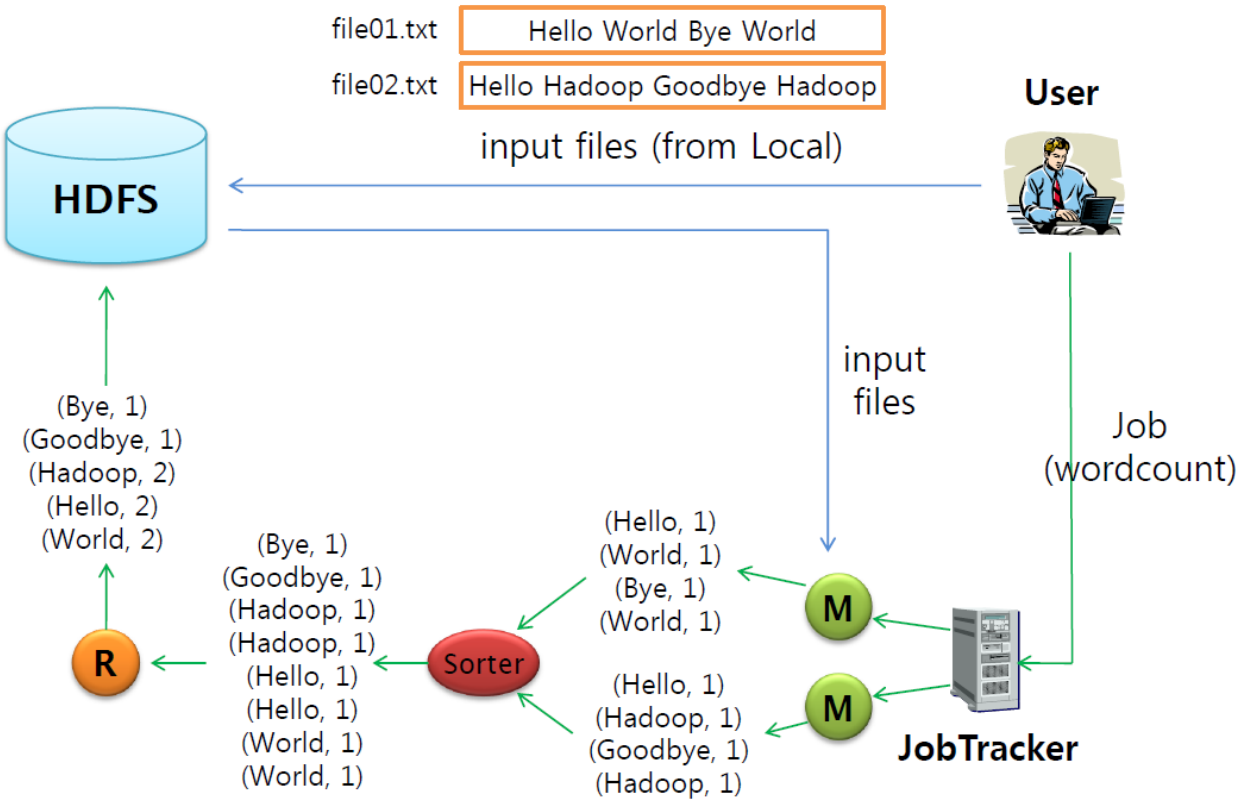
Map

```
public class WCMap extends MapReduceBase implements Mapper {

  private static final IntWritable ONE = new IntWritable(1);

  public void map(WritableComparable key, Writable value,
                  OutputCollector output,
                  Reporter reporter) throws IOException {
    StringTokenizer itr = new StringTokenizer(value.toString());
    while (itr.hasMoreTokens()) {
      output.collect(new Text(itr.next()), ONE);
    }
  }
}
```

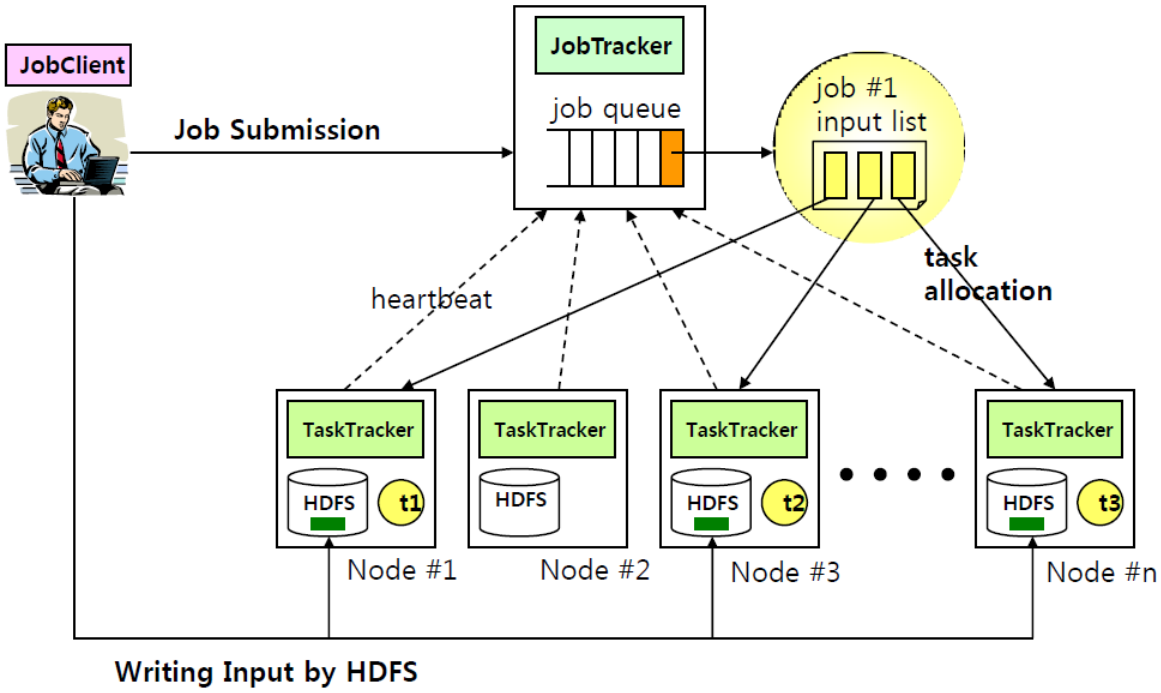
Reduce

```
public class WCReduce extends MapReduceBase implements Reducer {

  public void reduce(WritableComparable key, Iterator values,
                     OutputCollector output,
                     Reporter reporter) throws IOException {
    int sum = 0;
    while (values.hasNext()) {
      sum += ((IntWritable) values.next()).get();
    }
    output.collect(key, new IntWritable(sum));
  }
}
```

WordCount MapReduce



Hadoop MapReduce Architecture



Забележка: Може да преглеждате онлайн Hadoop сорс кода чрез GrepCode приложението.

Поздрави,

Крум.