Hadoop and Big Data

Unit 4: Hadoop I/O: The Writable Interface, WritableComparable and comparators, Writable Classes: Writable wrappers for Java primitives, Text, BytesWritable, NullWritable, ObjectWritable and GenericWritable, Writable collections, Implementing a Custom Writable: Implementing a RawComparator for speed, Custom comparators

Reference: Hadoop: The Definitive Guide by Tom White, 3rd Edition, O'reilly

Unit 4

1. Hadoop I/O:
    - The Writable Interface,
    - WritableComparable and comparators,
2. Writable Classes:
    - Writable wrappers for Java primitives,
    - Text,
    - BytesWritable,
    - NullWritable,
    - ObjectWritable
    - GenericWritable,
    - Writable collections,
3. Implementing a Custom Writable:
    - Implementing a Raw Comparator for speed,
    - Custom comparators

1. **Hadoop I/O**

The Writable Interface

Writable in an interface in Hadoop and types in Hadoop must implement this interface. Hadoop provides these writable wrappers for almost all Java primitive types and some other types,but sometimes we need to pass custom objects and these custom objects should implement Hadoop's Writable interface. Hadoop MapReduce uses implementations of Writables for interacting with user-provided Mappers and Reducers.

To implement the Writable interface we require two methods:

```
public interface Writable {

void readFields(DataInput in);

void write(DataOutput out);
}
```

Why use Hadoop Writable(s)?

As we already know, data needs to be transmitted between different nodes in a distributed computing environment. This requires serialization and deserialization of data to convert the data that is in structured format to byte stream and vice-versa. Hadoop therefore uses simple and efficient serialization protocol to serialize data between map and reduce phase and these are called Writable(s). Some of the examples of writables as already mentioned before are IntWritable, LongWritable, BooleanWritable and FloatWritable.

WritableComparable interface is just a sub interface of the Writable and java.lang.Comparable interfaces. For implementing a WritableComparable we must have compareTo method apart from readFields and write methods, as shown below:

```
public interface WritableComparable extends Writable, Comparable
{
   void readFields(DataInput in);
   void write(DataOutput out);
   int compareTo(WritableComparable o)
}
```

Comparison of types is crucial for MapReduce, where there is a sorting phase during which keys are compared with one another.

Implementing a comparator for WritableComparables like the org.apache.hadoop.io.RawComparator interface will definitely help speed up your Map/Reduce

(MR) Jobs. As you may recall, a MR Job is composed of receiving and sending key-value pairs. The process looks like the following.

```
(K1,V1) –> Map –> (K2,V2)
(K2,List[V2]) –> Reduce –> (K3,V3)
```

The key-value pairs (K2,V2) are called the intermediary key-value pairs. They are passed from the mapper to the reducer. Before these intermediary key-value pairs reach the reducer, a shuffle and sort step is performed.

The shuffle is the assignment of the intermediary keys (K2) to reducers and the sort is the sorting of these keys. In this blog, by implementing the RawComparator to compare the intermediary keys, this extra effort will greatly improve sorting. Sorting is improved because the RawComparator will compare the keys by byte. If we did not use RawComparator, the intermediary keys would have to be completely deserialized to perform a comparison.

Note (In Short):

1)WritableComparables can be compared to each other, typically via Comparators. Any type which is to be used as a key in the Hadoop Map-Reduce framework should implement this interface.

2) Any type which is to be used as a value in the Hadoop Map-Reduce framework should implement the Writable interface.

Writables and its Importance in Hadoop

Writable is an interface in Hadoop. Writable in Hadoop acts as a wrapper class to almost all the primitive data type of Java. That is how int of java has become IntWritable in Hadoop and String of Java has become Text in Hadoop.

Writables are used for creating serialized data types in Hadoop. So, let us start by understanding what data type, interface and serilization is.

Data Type

A data type is a set of data with values having predefined characteristics. There are several kinds of data types in Java. For example- int, short, byte, long, char etc. These are called as primitive data types. All these primitive data types are bound to classes called as wrapper class. For example int, short, byte, long are grouped under INTEGER which is a wrapper class. These wrapper classes are predefined in the Java.

Interface in Java

An interface in Java is a complete abstract class. The methods within an interface are abstract methods which do not accept body and the fields within the interface are public, static and final, which means that the fields cannot be modified.

The structure of an interface is most likely to be a class. We cannot create an object for an interface and the only way to use the interface is to implement it in other class by using **'implements' keyword.**

Serialization

Serialization is nothing but converting the raw data into a stream of bytes which can travel along different networks and can reside in different systems. Serialization is not the only concern of Writable interface; it also has to perform compare and sorting operation in Hadoop.

Why are Writables Introduced in Hadoop?

Now the question is whether Writables are necessary for Hadoop. Hadoop frame work definitely needs Writable type of interface in order to perform the following tasks:

Implement serialization ,Transfer data between clusters and networks

Store the deserialized data in the local disk of the system

Implementation of writable is similar to implementation of interface in Java. It can be done by **simply writing the keyword 'implements' and overriding the default writable method.**

Writable is a strong interface in Hadoop which while serializing the data, reduces the data size enormously, so that data can be exchanged easily within the networks. It has separate read and write fields to read data from network and write data into local disk respectively. Every data inside Hadoop should accept writable and comparable interface properties.

We have seen how Writables reduces the data size overhead and make the data transfer easier in the network.

What if Writable were not there in Hadoop?

Let us now understand what happens if Writable is not present in Hadoop.

Serialization is important in Hadoop because it enables easy transfer of data. If Writable is not present in Hadoop, then it uses the serialization of Java which increases the data over-head in the network.

smallInt serialized value using Java serializer

aced0005737200116a6176612e6c616e672e496e74656765

7212e2a0a4f781873802000149000576616c7565787200106a6176612e

6c616e672e4e756d62657286ac951d0b94e08b020000787000000064

smallInt serialized value using IntWritable

00000064

This shows the clear difference between serialization in Java and Hadoop and also the difference between ObjectInputStream and Writable interface. If the size of serialized data in Hadoop is like that of Java, then it will definitely become an overhead in the network.

Also the core part of Hadoop framework i.e., shuffle and sort phase won't be executed without using Writable.

How can Writables be Implemneted in Hadoop?

Writable variables in Hadoop have the default properties of Comparable. For example:

When we write a key as IntWritable in the Mapper class and send it to the reducer class, there is an intermediate phase between the Mapper and Reducer class i.e., shuffle and sort, where each key has to be compared with many other keys. If the keys are not comparable, then shuffle and **sort phase won't be executed or may be executed with high amount of overhead.**

If a key is taken asIntWritable by default, then it has comparable feature because of RawComparator acting on that variable. It will compare the key taken with the other keys in the network. This cannot take place in the absence of Writable.

Can we make custom Writables? The answer is definitely 'yes'. We can make our own custom Writable type.

Let us now see how to make a custom type in Java.

The steps to make a custom type in Java is as follows:

```
public class add {
   int a;
   int b;
   public add() {
      this.a = a;
      this.b = b;
   }
}
```

Similarly we can make a custom type in Hadoop using Writables.

For implementing Writables, we need few more methods in Hadoop:

```
public interface Writable {

void readFields(DataInput in);

void write(DataOutput out);

}
```

Here, readFields, reads the data from network and write will write the data into local disk. Both are necessary for transferring data through clusters. DataInput and DataOutput classes (part of java.io) contain methods to serialize the most basic types of data.

Suppose we want to make a composite key in Hadoop by combining two Writables then follow the steps below:

```
public class add implements Writable{
public int a;

public int b;

public add(){

this.a=a;

this.b=b;

}

public void write(DataOutput out) throws IOException {

    out.writeInt(a);

    out.writeInt(b);

  }

public void readFields(DataInput in) throws IOException {

    a = in.readInt();
```

```
    b = in.readInt();
  }
  public String toString() {
    return Integer.toString(a) + ", " + Integer.toString(b)
  }
}
```

Thus we can create our custom Writables in a way similar to custom types in Java but with two additional methods, write and readFields. The custom writable can travel through networks and can reside in other systems.

This custom type cannot be compared with each other by default, so again we need to make them comparable with each other.

Let us now discuss what is WritableComparable and the solution to the above problem.

As explained above, if a key is taken as IntWritable, by default it has comparable feature because of RawComparator acting on that variable and it will compare the key taken with the other keys **in network and If Writable is not there it won't be executed.**

By default, IntWritable, LongWritable and Text have a RawComparator which can execute this comparable phase for them. Then, will RawComparator help the custom Writable? The answer is no. So, we need to have WritableComparable.

WritableComparable can be defined as a sub interface of Writable, which has the feature of Comparable too. If we have created our custom type writable, then

why do we need WritableComparable?

We need to make our custom type, comparable if we want to compare this type with the other.

we want to make our custom type as a key, then we should definitely make our key type as WritableComparable rather than simply Writable. This enables the custom type to be compared **with other types and it is also sorted accordingly. Otherwise, the keys won't be compared with** each other and they are just passed through the network.

What happens if WritableComparable is not present?

If we have made our custom type **Writable rather than WritableComparable our data won't be** compared with other data types. There is no compulsion that our custom types need to be **WritableComparable until unless if it is a key. Because values don't need to be compared with** each other as keys.

If our custom type is a key then we should have WritableComparable or else the data won't be sorted.

How can WritableComparable be implemented in Hadoop?

The implementation of WritableComparable is similar to Writable but with an additional **'CompareTo' method inside it.**

```
public interface WritableComparable extends Writable, Comparable

{

   void readFields(DataInput in);

    void write(DataOutput out);

    int compareTo(WritableComparable o)

}
```

How to make our custom type, WritableComparable?

We can make custom type a WritableComparable by following the method below:

```
public class add implements WritableComparable{

public int a;

public int b;

public add(){

this.a=a;

this.b=b;

}

public void write(DataOutput out) throws IOException {
```

```
    out.writeint(a);

    out.writeint(b);

  }

public void readFields(DataInput in) throws IOException {

    a = in.readint();

    b = in.readint();

  }

public int CompareTo(add c){
 int presentValue=this.value;
 int CompareValue=c.value;
 return (presentValue < CompareValue ? -1 : (presentValue==CompareValue ? 0 : 1));
 }

public int hashCode() {
     return Integer.IntToIntBits(a)^ Integer.IntToIntBits(b);
   }

}
```

These read fields and write make the comparison of data faster in the network.

With the use of these Writable and WritableComparables in Hadoop, we can make our serialized custom type with less difficulty. This gives the ease for developers to make their custom types based on their requirement.

2. Writable Classes – Hadoop Data Types

Hadoop provides classes that wrap the Java primitive types and implement the *WritableComparable* and *Writable* Interfaces. They are provided in the org.apache.hadoop.io package.

All the Writable wrapper classes have a get() and a set() method for retrieving and storing the wrapped value.

Primitive Writable Classes

These are Writable Wrappers for Java primitive data types and they hold a single primitive value that can be set either at construction or via a setter method.

All these primitive writable wrappers have get() and set() methods to read or write the wrapped value. Below is the list of primitive writable data types available in Hadoop.

- BooleanWritable
- ByteWritable
- IntWritable
- VIntWritable
- FloatWritable
- LongWritable
- VLongWritable
- DoubleWritable

In the above list VIntWritable and VLongWritable are used for variable length Integer types and variable length long types respectively.

Serialized sizes of the above primitive writable data types are same as the size of actual java data type. So, the size of IntWritable is 4 bytes and LongWritable is 8 bytes.

Array Writable Classes

Hadoop provided two types of array writable classes, one for *single-dimensional* and another for *two-dimensional* arrays. But theelements of these arrays must be other writable objects like IntWritable or LongWritable only but not the java native data types like int or float.

- ArrayWritable
- TwoDArrayWritable

Map Writable Classes

Hadoop provided below MapWritable data types which implement java.util.Map interface

- AbstractMapWritable – This is abstract or base class for other MapWritable classes.

- MapWritable – This is a general purpose map mapping Writable keys to Writable values.
- SortedMapWritable – This is a specialization of the MapWritable class that also implements the SortedMap interface.

Other Writable Classes

- NullWritable

NullWritable is a special type of Writable representing a null value. No bytes are read or written when a data type is specified as NullWritable. So, in Mapreduce, a key or a value can be declared as a NullWritable when we **don't need to use that field.**

- ObjectWritable

This is a general-purpose generic object wrapper which can store any objects like Java primitives, String, Enum, Writable, null, or arrays.

- Text

Text can be used as the Writable equivalent of java.lang.String **and It's max size is 2 GB. Unlike java's String data type, Text is mutable in Hadoop.**

- BytesWritable

BytesWritable is a wrapper for an array of binary data.

- GenericWritable

It is similar to ObjectWritable but supports only a few types. User need to subclass this GenericWritable class and need to specify the types to support.

Example Program to Test Writables

Lets write a WritablesTest.java program to test most of the data types mentioned above in this post with get(), set(), getBytes(), getLength(), put(), containsKey(), keySet() methods.

WritablesTest.java

```
import org.apache.hadoop.io.* ;
import java.util.* ;
 public class WritablesTest
{
```

```
public static class TextArrayWritable extends
ArrayWritable {
   public TextArrayWritable()
  {
    super(Text.class) ;
  }
}

public static class IntArrayWritable extends
ArrayWritable {
  public IntArrayWritable()
  {
    super(IntWritable.class) ;
  }
}

public static void main(String[] args)
{

 IntWritable i1 = new IntWritable(2) ;
 IntWritable i2 = new IntWritable() ;
 i2.set(5);

 IntWritable i3 = new IntWritable();
 i3.set(i2.get());

 System.out.printf("Int Writables Test I1:%d , I2:%d , I3:%d", i1.get(), i2.get(), i3.get()) ;

 BooleanWritable bool1 = new BooleanWritable()
 ; bool1.set(true);

 ByteWritable byte1 = new ByteWritable( (byte)7) ;

 System.out.printf("\n Boolean Value:%s Byte Value:%d", bool1.get(), byte1.get()) ;

 Text t = new Text("hadoop");
 Text t2 = new Text();
 t2.set("pig");

 System.out.printf("\n t: %s, t.legth: %d, t2: %s, t2.length: %d \n", t.toString(), t.getLength(),
t2.getBytes(), t2.getBytes().length);

 ArrayWritable a = new ArrayWritable(IntWritable.class) ;
 a.set( new IntWritable[]{ new IntWritable(10), new IntWritable(20), new IntWritable(30)}) ;

 ArrayWritable b = new ArrayWritable(Text.class) ;
```

```
   b.set( new Text[]{ new Text("Hello"), new Text("Writables"), new Text("World !!!")}) ;

   for (IntWritable i: (IntWritable[])a.get())
   System.out.println(i) ;

   for (Text i: (Text[])b.get())
   System.out.println(i) ;

   IntArrayWritable ia = new IntArrayWritable() ;
   ia.set( new IntWritable[]{ new IntWritable(100), new IntWritable(300), new
IntWritable(500)}) ;

   IntWritable[] ivalues = (IntWritable[])ia.get() ;

   for (IntWritable i : ivalues)
   System.out.println(i);

   MapWritable m = new MapWritable() ;
   IntWritable key1 = new IntWritable(1) ;
   NullWritable value1 = NullWritable.get() ;

   m.put(key1, value1) ;
   m.put(new  VIntWritable(2),  new  LongWritable(163));
   m.put(new VIntWritable(3), new Text("Mapreduce"));

   System.out.println(m.containsKey(key1)) ;
   System.out.println(m.get(new VIntWritable(3))) ;

   m.put(new LongWritable(1000000000), key1) ;
   Set<Writable> keys = m.keySet() ;

   for(Writable w: keys)
    System.out.println(m.get(w)) ;
  }
}
```

```
hadoop1@ubuntu-1:~/project$ javac -classpath $CLASSPATH WritablesTest.java
hadoop1@ubuntu-1:~/project$ java WritablesTest
Int Writables Test I1:2 , I2:5 , I3:5
Boolean Value:true Byte Value:7
t: hadoop, t.legth: 6, t2: [B@1a93a7ca, t2.length: 3
-------Array variables-------
10
20
30
Hello
Writables
World !!!
100
300
500
-------Map Variables-------
true
Mapreduce
(null)
163
Mapreduce
1
hadoop1@ubuntu-1:~/project$
```

3. **Implementing a Custom Writable:**

IMPLEMENTING RAWCOMPARATOR WILL SPEED UP YOUR HADOOP MAP/REDUCE (MR) JOBS

Implementing the org.apache.hadoop.io.RawComparator interface will definitely help speed up your Map/Reduce (MR) Jobs. As you may recall, a MR Job is composed of receiving and sending key-value pairs. The process looks like the following.

(K1,V1) –> Map –> (K2,V2)

(K2,List[V2]) –> Reduce –> (K3,V3)

The key-value pairs (K2,V2) are called the intermediary key-value pairs. They are passed from the mapper to the reducer. Before these intermediary key-value pairs reach the reducer, a shuffle and sort step is performed. The shuffle is the assignment of the intermediary keys (K2) to reducers and the sort is the sorting of these keys. In this blog, by implementing the RawComparator to compare the intermediary keys, this extra effort will greatly improve sorting. Sorting is improved because the RawComparator will compare the keys by byte. If we did not use RawComparator, the intermediary keys would have to be completely deserialized to perform a comparison.

BACKGROUND

Two ways you may compare your keys is by implementing the org.apache.hadoop.io.WritableComparable interface or by implementing the RawComparator interface. In the former approach, you will compare (deserialized) objects, but in the latter approach, you will compare the keys using their corresponding raw bytes.

The empirical test to demonstrate the advantage of RawComparator over WritableComparable. **Let's say we are processing a file that has a list of pairs of indexes {i,j}. Thes**e pairs of indexes could refer to the i-th and j-th matrix element. The input data (file) will look something like the following.

1, 2

3, 4

5, 6

...

...

...

0, 0

What we want to do is simply count the occurrences of the {i,j} pair of indexes. Our MR Job will look like the following.

(LongWritable,Text) –> Map –> ({i,j},IntWritable)

({i,j},List[IntWritable]) –> Reduce –> ({i,j},IntWritable)

METHOD

The first thing we have to do is model our intermediary key K2={i,j}. Below is a snippet of the IndexPair. As you can see, it implements WritableComparable. Also, we are sorting the keys ascendingly by the i-th and then j-th indexes.

```
public class IndexPair implements WritableComparable<IndexPair> {

    private IntWritable i;

    private IntWritable j;
```

```
    public IndexPair(int i, int j) {
        this.i = new IntWritable(i);
        this.j = new IntWritable(j);
    }
    public int compareTo(IndexPair o) {
        int cmp = i.compareTo(o.i);
        if(0 != cmp)
            return cmp;
        return j.compareTo(o.j);
    }
    //....
}
```

Below is a snippet of the RawComparator. As you notice, it does not directly implement RawComparator. Rather, it extends WritableComparator (which implements RawComparator). We could have directly implemented RawComparator, but by extending WritableComparator, depending on the complexity of our intermediary key, we may use some of the utility methods of WritableComparator.

```
public class IndexPairComparator extends WritableComparator {
    protected IndexPairComparator() {
        super(IndexPair.class);
    }
    @Override
    public int compare(byte[] b1, int s1, int l1, byte[] b2, int s2, int l2) {
        int i1 = readInt(b1, s1);
        int i2 = readInt(b2, s2);

        int comp = (i1 < i2) ? -1 : (i1 == i2) ? 0 : 1;
```

```
        if(0 != comp)
            return comp;
        int j1 = readInt(b1, s1+4);
        int j2 = readInt(b2, s2+4);
        comp = (j1 < j2) ? -1 : (j1 == j2) ? 0 : 1;

        return comp;
    }
}
```

As you can see the above code, for the two objects we are comparing, there are two corresponding byte arrays (b1 and b2), the starting positions of the objects in the byte arrays, and the length of the bytes they occupy. Please note that the byte arrays themselves represent other things and not only the objects we are comparing. That is why the starting position and length are also passed in as arguments. Since we want to sort ascendingly by i then j, we first compare the bytes representing the i-th indexes and if they are equal, then we compare the j-th indexes. You can also see that we use the util method, readInt(byte[], start), inherited from WritableComparator. This method simply converts the 4 consecutive bytes beginning at start into a primitive int (the primitive int in Java is 4 bytes). If the i-th indexes are equal, then we shift the starting point by 4, read in the j-th indexes and then compare them.

A snippet of the mapper is shown below.

```
public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
    String[] tokens = value.toString().split(",");
    int i = Integer.parseInt(tokens[0].trim());
    int j = Integer.parseInt(tokens[1].trim());

    IndexPair indexPair = new IndexPair(i, j);
    context.write(indexPair, ONE);
}
```

A snippet of the reducer is shown below.

```
public void reduce(IndexPair key, Iterable<IntWritable> values, Context context) throws
IOException, InterruptedException {

   int sum = 0;

   for(IntWritable value : values) {

      sum += value.get();

   }



   context.write(key, new IntWritable(sum));

}
```

The snippet of code below shows how I wired up the MR Job that does NOT use raw byte comparison.

```
public int run(String[] args) throws Exception {

   Configuration conf = getConf();

   Job job = new Job(conf, "raw comparator

     example"); job.setJarByClass(RcJob1.class);



   job.setMapOutputKeyClass(IndexPair.class);

   job.setMapOutputValueClass(IntWritable.class);



   job.setOutputKeyClass(IndexPair.class);

   job.setOutputValueClass(IntWritable.class);



   job.setMapperClass(RcMapper.class);

   job.setReducerClass(RcReducer.class);
```

```
    job.waitForCompletion(true);

    return 0;
}
```

The snippet of code below shows how I wired up the MR Job using the raw byte comparator.

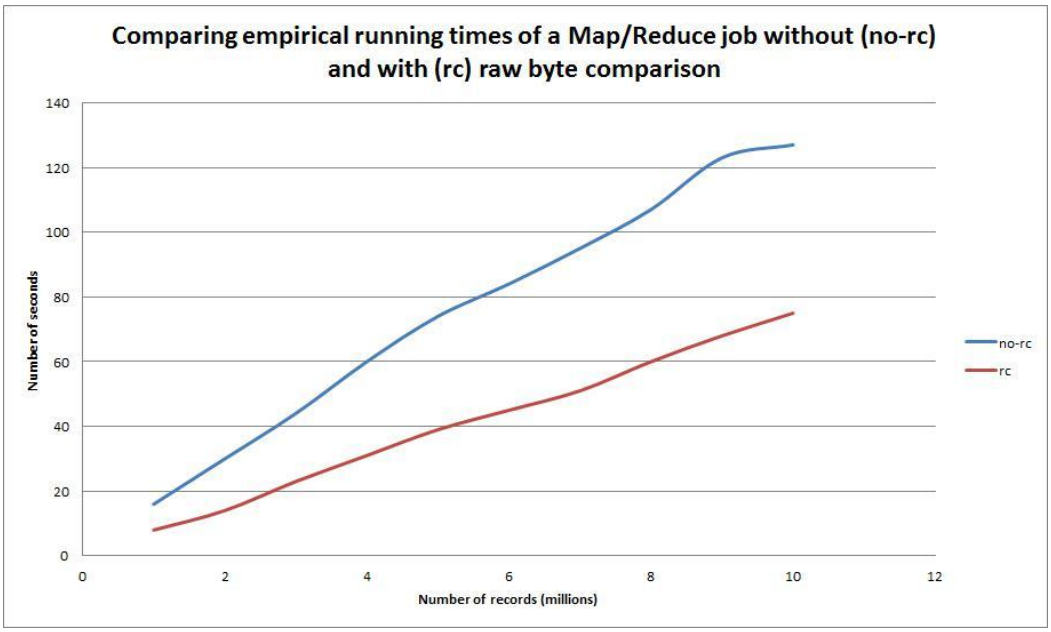
```
public int run(String[] args) throws Exception {
    Configuration conf = getConf();
    Job job = new Job(conf, "raw comparator example");

    job.setJarByClass(RcJob1.class);
    job.setSortComparatorClass(IndexPairComparator.class);

    job.setMapOutputKeyClass(IndexPair.class);
    job.setMapOutputValueClass(IntWritable.class);

    job.setOutputKeyClass(IndexPair.class);
    job.setOutputValueClass(IntWritable.class);

    job.setMapperClass(RcMapper.class);
    job.setReducerClass(RcReducer.class);

    job.waitForCompletion(true);

    return 0;
}
```

As you can see, the only difference is that in the MR Job using the raw comparator, we explicitly set its sort comparator class.

RESULTS

I ran the MR Jobs (without and with raw byte comparisons) 10 times on a dataset of 4 million rows of {i,j} pairs. The runs were against Hadoop v0.20 in standalone mode on Cygwin. The average running time for the MR Job without raw byte comparison is 60.6 seconds, and the average running time for the job with raw byte comparison is 31.1 seconds. A two-tail paired t-test showed $p < 0.001$, meaning, there is a statistically significant difference between the two implementations in terms of empirical running time.

I then ran each implementation on datasets of increasing record s**izes from 1, 2, …, and 10** million records. At 10 million records, without using raw byte comparison took 127 seconds (over 2 minutes) to complete, while using raw byte comparison took 75 seconds (1 minute and 15 seconds) to complete. Below is a line graph.



Custom comparators.

Frequently, objects in one `Tuple` are compared to objects in a second `Tuple`. This is especially true during the sort phase of `GroupBy` and `CoGroup` in Cascading Hadoop mode.

By default, Hadoop and Cascading use the native `Object` methods `equals()` and `hashCode()` to compare two values and get a consistent hash code for a given value, respectively.

To override this default behavior, you can create a custom `java.util.Comparator` class to perform comparisons on a given field in a Tuple. For instance, to secondary-sort a collection of custom `Person` objects in a `GroupBy`, use the `Fields.setComparator()` method to designate the custom `Comparator` to the `Fields` instance that specifies the sort fields.

Alternatively, you can set a default `Comparator` to be used by a `Flow`, or used locally on a given `Pipe` instance. There are two ways to do this. Call `FlowProps.setDefaultTupleElementComparator()` on a `Properties` instance, or use the property key `cascading.flow.tuple.element.comparator`.

If the hash code must also be customized, the custom Comparator can implement the interface `cascading.tuple.Hasher`

```
public class CustomTextComparator extends

{

private Collator collator;

public CustomTextComparator()

 super(Text.class);


 final Locale locale = new Locale("pl");

 collator = Collator.getInstance(locale);

 }
```

```
public int compare(WritableComparable a, WritableComparable b) {
 synchronized (collator) {
 return collator.compare(
 ((Text) a).toString(), ((Text) b).toString());
}
 }
 }
```