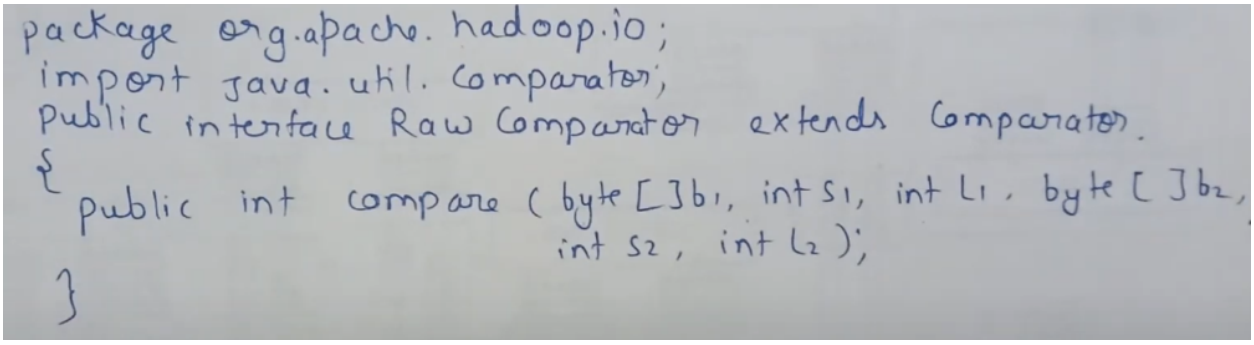
Unit -1 Hadoop I/O

Writable Comparator

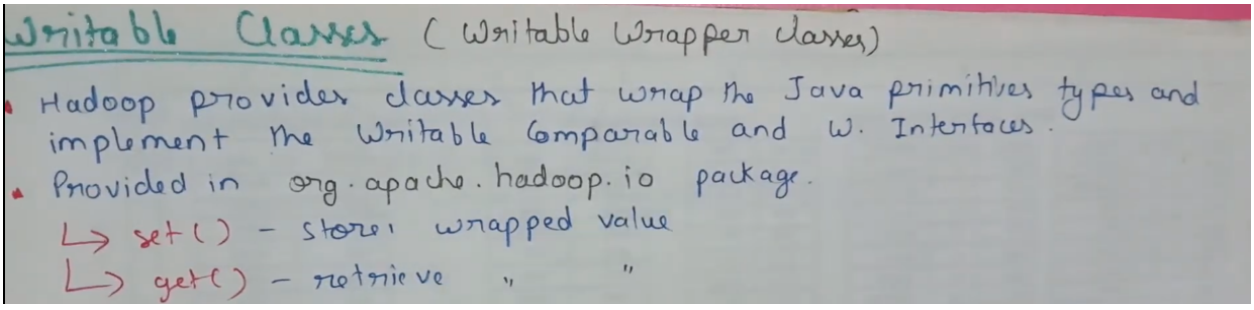
Writable Comparator in Hadoop is mainly used for efficient comparison of Hadoop Writable objects. It works directly on binary (raw) data without fully deserializing objects, making sorting faster during the shuffle and sort phase in MapReduce.

Hadoop stores key-value pairs in serialized binary format.
Writable Comparator reads bytes directly from the stream, making it efficient.



Writable Classes: Writable classes in Hadoop are specialized data structures that enable efficient storage, transmission, and processing of data in a distributed environment. Since Hadoop deals with massive datasets across multiple machines, it uses Writable classes to serialize (convert data into bytes) and deserialize (reconstruct data from bytes) quickly and efficiently.

Hadoop provides several built-in Writable classes, such as IntWritable for integers, Text for strings, and LongWritable for long numbers, which replace standard Java types. However, for handling more complex data structures, developers can create custom Writable classes by implementing the Writable interface.



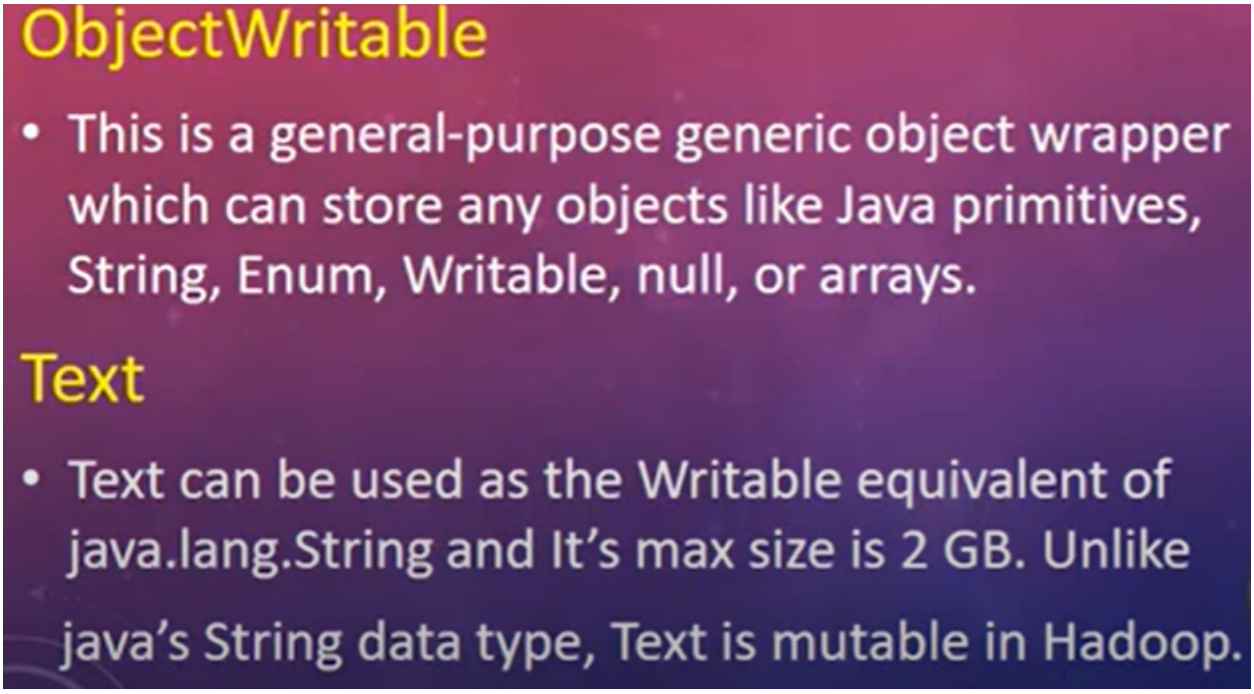
Writable wrappers for Java Primitives

Java primitive	Writable Implementation	Serialized size (bytes)
boolean	BooleanWritable	1
byte	ByteWritable	1
short	ShortWritable	2
int	IntWritable	4
	VIntWritable	1-5
float	FloatWritable	4
long	LongWritable	8
	VLongWritable	1-9
double	DoubleWritable	8

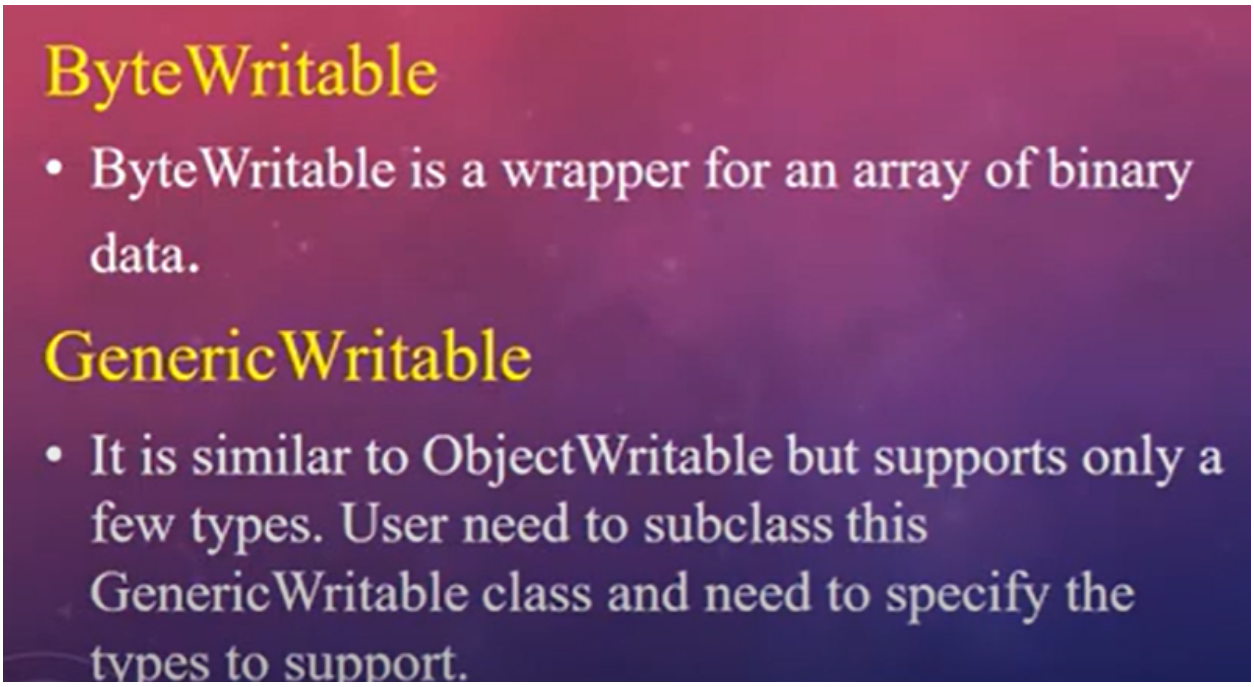
- VIntWritable and VLongWritable are used for variable length integer types and variable length long types respectively.
- Serialized sizes of the primitives ~~wrapper~~ writable data types are same as the size of actual java datatype.
- So, the size of IntWritable is 4 bytes and LongWritable is 8 bytes

NULLWRITABLE CLASS

- 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.



Text is writable for UTF -8 Sequences means all strings, characters, Words, or UTF-8 Sequences itself will be used under TEXT WRITABLES.



WRITABLE COLLECTION

Hadoop's **Writable Collection** classes are designed to handle complex data types efficiently in a distributed computing environment. These classes extend the basic **Writable** interface and allow structured data to be processed in MapReduce applications.

Types of Writable Collections

1. **ArrayWritable**
    - It is used to store an array of Writable objects of the same type.
    - Suitable for scenarios where multiple values of the same data type need to be grouped together.
    - Example Use Case: Storing multiple temperature readings for a particular day in weather data processing.
2. **Array Primitive Writable**
    - **ArrayPrimitiveWritable** is designed to handle arrays of Java's primitive data types (int, double, float, etc.) in a memory-efficient manner.
    - Unlike **ArrayWritable**, which works with **Writable objects**, **ArrayPrimitiveWritable** is optimized for raw primitive types, reducing serialization overhead.
    - This class is particularly useful for numerical computations where primitive arrays are frequently used.
3. **TwoDArrayWritable**
    - A specialized version of **ArrayWritable** that stores a two-dimensional array of Writable objects.
    - Useful in scenarios involving matrices or grid-based computations.
    - Example Use Case: Representing image pixel data in Hadoop-based image processing tasks.
4. **MapWritable**
    - It functions like a **HashMap**, storing key-value pairs where both the key and value implement the **Writable** interface.
    - Unlike Java's standard HashMap, it is optimized for Hadoop's serialization framework.
    - Example Use Case: Storing word counts in a word frequency analysis task.
5. **SortedMapWritable**
    - It extends **MapWritable** but maintains a sorted order of keys.
    - The sorting mechanism ensures efficient retrieval and processing of key-value pairs.
    - Example Use Case: Storing aggregated stock market data sorted by time or stock symbol.

6. **EnumSetWritable**
    - It is a Hadoop-specific writable implementation that allows **EnumSet** (a specialized Java Set for Enums) to be serialized and deserialized efficiently.
    - It is useful when dealing with categorical data that is best represented as a set of enumerated values.
    - Unlike Java's regular **Set**, which can store any object, **EnumSet** is optimized for enums, making it more memory-efficient and faster in operations.

Custom Comparator

In Hadoop, a **custom comparator** is used to control the sorting order of keys during the **shuffle and sort phase** of MapReduce. By default, Hadoop sorts keys in **ascending order**, but sometimes you need a different order — like **descending order** or sorting based on some **custom logic**. That's where a custom comparator comes in.

Let's break down how it works:

1. **Create a Custom Comparator**:
   You extend the **WritableComparator** class and override the **compare()** method. Here, you define your custom logic for comparing two keys.
2. **Register the Comparator**:
   In your **Job configuration**, you use `job.setSortComparatorClass(YourComparator.class)` to tell Hadoop to use your custom comparator instead of the default one.

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}. These** 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.

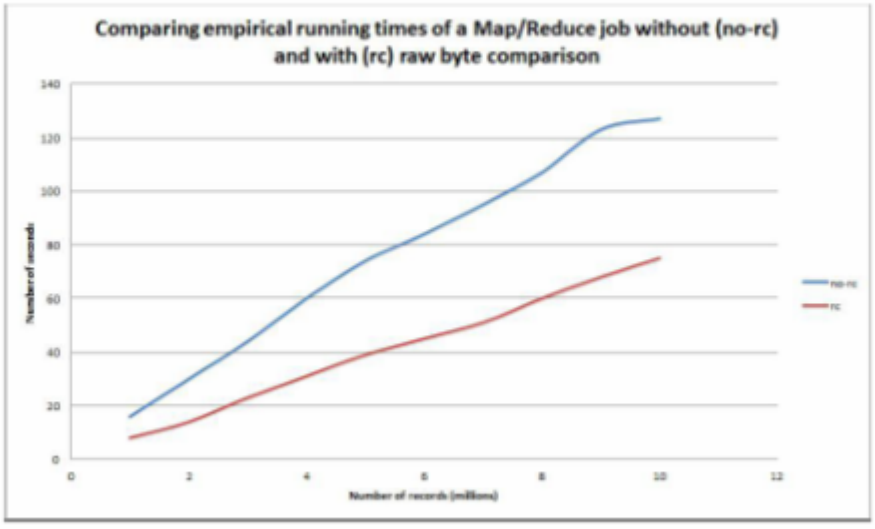
```
public int run(String[] args) throws Exception {
    Configuration conf = getConf();
    Job job = new Job(conf, "raw comparator example");

    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 **sizes 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.



Implementing RawComparator will speed up your Hadoop Map/Reduce (MR) Jobs

Introduction

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.

I conducted an 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}. These 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.

```
1  public class IndexPair implements WritableComparable<IndexPair> {
2     private IntWritable i;
3     private IntWritable j;
4     //....
5     /**
6      * Constructor.
7      * @param i i.
8      * @param j j.
9      */
10    public IndexPair(int i, int j) {
11       this.i = new IntWritable(i);
12       this.j = new IntWritable(j);
13    }
14    //....
15    @Override
16    public int compareTo(IndexPair o) {
17       int cmp = i.compareTo(o.i);
18       if(0 != cmp)
19          return cmp;
20       return j.compareTo(o.j);
21    }
22    //....
23}
```

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.

```
1  public class IndexPairComparator extends WritableComparator {
2     protected IndexPairComparator() {
3        super(IndexPair.class);
```

```
4     }
5
6     @Override
7     public int compare(byte[] b1, int s1, int l1, byte[] b2, int s2, int l2) {
8        int i1 = readInt(b1, s1);
9        int i2 = readInt(b2, s2);
10
11       int comp = (i1 < i2) ? -1 : (i1 == i2) ? 0 : 1;
12       if(0 != comp)
13          return comp;
14
15       int j1 = readInt(b1, s1+4);
16       int j2 = readInt(b2, s2+4);
17       comp = (j1 < j2) ? -1 : (j1 == j2) ? 0 : 1;
18
19       return comp;
20    }
21}
```

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.

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

```
4    int j = Integer.parseInt(tokens[1].trim());
5
6    IndexPair indexPair = new IndexPair(i, j);
7    context.write(indexPair, ONE);
8}
```

A snippet of the reducer is shown below.

```
1public void reduce(IndexPair key, Iterable<IntWritable> values, Context context) throws IOException, InterruptedExcep
2    int sum = 0;
3    for(IntWritable value : values) {
4        sum += value.get();
5    }
6
7    context.write(key, new IntWritable(sum));
8}
```

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

```
1  public int run(String[] args) throws Exception {
2      Configuration conf = getConf();
3      Job job = new Job(conf, "raw comparator example");
4
5      job.setJarByClass(RcJob1.class);
6
7      job.setMapOutputKeyClass(IndexPair.class);
8      job.setMapOutputValueClass(IntWritable.class);
9
10     job.setOutputKeyClass(IndexPair.class);
11     job.setOutputValueClass(IntWritable.class);
12
13     job.setMapperClass(RcMapper.class);
14     job.setReducerClass(RcReducer.class);
```

```
15
16    job.waitForCompletion(true);
17
18    return 0;
19}
```

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

```
1  public int run(String[] args) throws Exception {
2     Configuration conf = getConf();
3     Job job = new Job(conf, "raw comparator example");
4
5     job.setJarByClass(RcJob1.class);
6     job.setSortComparatorClass(IndexPairComparator.class);
7
8     job.setMapOutputKeyClass(IndexPair.class);
9     job.setMapOutputValueClass(IntWritable.class);
10
11    job.setOutputKeyClass(IndexPair.class);
12    job.setOutputValueClass(IntWritable.class);
13
14    job.setMapperClass(RcMapper.class);
15    job.setReducerClass(RcReducer.class);
16
17    job.waitForCompletion(true);
18
19    return 0;
20}
```

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 sizes 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.