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

Serialization

Serialization is the process of turning structured objects into a byte stream for transmission over a network or for writing to persistent storage. *Deserialization* is the reverse process of turning a byte stream back into a series of structured objects. Serialization appears in two quite distinct areas of distributed data processing: for interprocess communication and for persistent storage. In Hadoop, interprocess communication between nodes in the system is implemented using *remote procedure calls* (RPCs). The RPC protocol uses serialization to render the message into a binary stream to be sent to the remote node, which then deserializes the binary stream into the original message. In general, an RPC serialization format is:

Compact

A compact format makes the best use of network bandwidth, which is the most scarce resource in a data center.

Fast

Interprocess communication forms the backbone for a distributed system, so it is essential that there is as little performance overhead as possible for the serialization and deserialization process.

Extensible

Protocols change over time to meet new requirements, so it should be straightforward to evolve the protocol in a controlled manner for clients and servers.

Interoperable

For some systems, it is desirable to be able to support clients that are written in different languages to the server, so the format needs to be designed to make this possible.

Hadoop uses its own serialization format, Writables, which is certainly compact and fast, but not so easy to extend or use from languages other than Java.

The Writable Interface

The Writable interface defines two methods: one for writing its state to a DataOutput binary stream by using write(), and one for reading its state from a DataInput binary stream by using readFields()as below:

```
package org.apache.hadoop.io;
import java.io.DataOutput;
import java.io.DataInput;
import java.io.IOException;
public interface Writable
{
void write(DataOutput out) throws IOException;
void readFields(DataInput in) throws IOException;
}
```

We will use IntWritable, a wrapper for a Java int. We can create one and set its value using the set() method:

```
IntWritable writable = new IntWritable();
writable.set(163);
```

Equivalently, we can use the constructor that takes the integer value:

```
IntWritable writable = new IntWritable(163);
```

To examine the serialized form of the IntWritable, we write a small helper method that wraps a java.io.ByteArrayOutputStream in a java.io.DataOutputStream to capture the bytes in the serialized stream:

```
public static byte[] serialize(Writable writable) throws IOException
{
ByteArrayOutputStream out = new ByteArrayOutputStream();
DataOutputStream dataOut = new DataOutputStream(out);
writable.write(dataOut);
dataOut.close();
return out.toByteArray();
}
```

An integer is written using four bytes

```
byte[] bytes = serialize(writable);
assertThat(bytes.length, is(4));
```

The bytes are written in big-endian order and we can see their hexadecimal representation by using a method on Hadoop's StringUtils:

```
assertThat(StringUtils.byteToHexString(bytes), is("000000a3"));
```

Let's try deserialization. Again, we create a helper method to read a Writable object from a byte array:

```
public static byte[] deserialize(Writable writable, byte[] bytes)
throws IOException
{
ByteArrayInputStream in = new ByteArrayInputStream(bytes);
DataInputStream dataIn = new DataInputStream(in);
writable.readFields(dataIn);
dataIn.close();
return bytes;
}
```

We construct a new, value-less, IntWritable, then call deserialize() to read from the output data that we just wrote. Then we check that its value, retrieved using the get() method, is the original value, 163:

```
IntWritable newWritable = new IntWritable();
deserialize(newWritable, bytes);
assertThat(newWritable.get(), is(163));
```

WritableComparable and comparators

IntWritable implements the WritableComparable interface, which is just a subinterface of the Writable and java.lang.Comparable interfaces:

```
package org.apache.hadoop.io;
public interface WritableComparable<T> extends Writable, Comparable<T>
{
}
```

Comparison of types is crucial for MapReduce, where there is a sorting phase during which keys are compared with one another. One optimization that Hadoop provides is the RawComparator extension of Java's Comparator:

```
package org.apache.hadoop.io;
import java.util.Comparator;
public interface RawComparator<T> extends Comparator<T>
{
public int compare(byte[] b1, int s1, int l1, byte[] b2, int s2, int l2);
}
```

In the above example, the comparator for IntWritables implements the raw compare() method by reading an integer from each of the byte arrays b1 and b2 and comparing them directly, from the given start positions (s1 and s2) and lengths (l1 and l2). This interface permits implementors to compare records read from a stream without deserializing them into objects, thereby avoiding any overhead of object creation.
WritableComparator is a general-purpose implementation of RawComparator for WritableComparable classes. It provides two main functions. First, it provides a default implementation of the raw compare() method that deserializes the objects to be compared from the stream and invokes the object compare() method. Second, it acts as a factory for RawComparator instances.
For example, to obtain a comparator for IntWritable, we just use:

```
RawComparator<IntWritable> comparator = WritableComparator.get(IntWritable.class);
```

The comparator can be used to compare two IntWritable objects:

```
IntWritable w1 = new IntWritable(163);
IntWritable w2 = new IntWritable(67);
assertThat(comparator.compare(w1, w2), greaterThan(0));
```

or their serialized representations:

```
byte[] b1 = serialize(w1);
byte[] b2 = serialize(w2);
assertThat(comparator.compare(b1, 0, b1.length, b2, 0, b2.length),
greaterThan(0));
```

Writable Classes

Hadoop comes with a large selection of Writable classes in the org.apache.hadoop.io package. They form the class hierarchy shown in Figure 1.

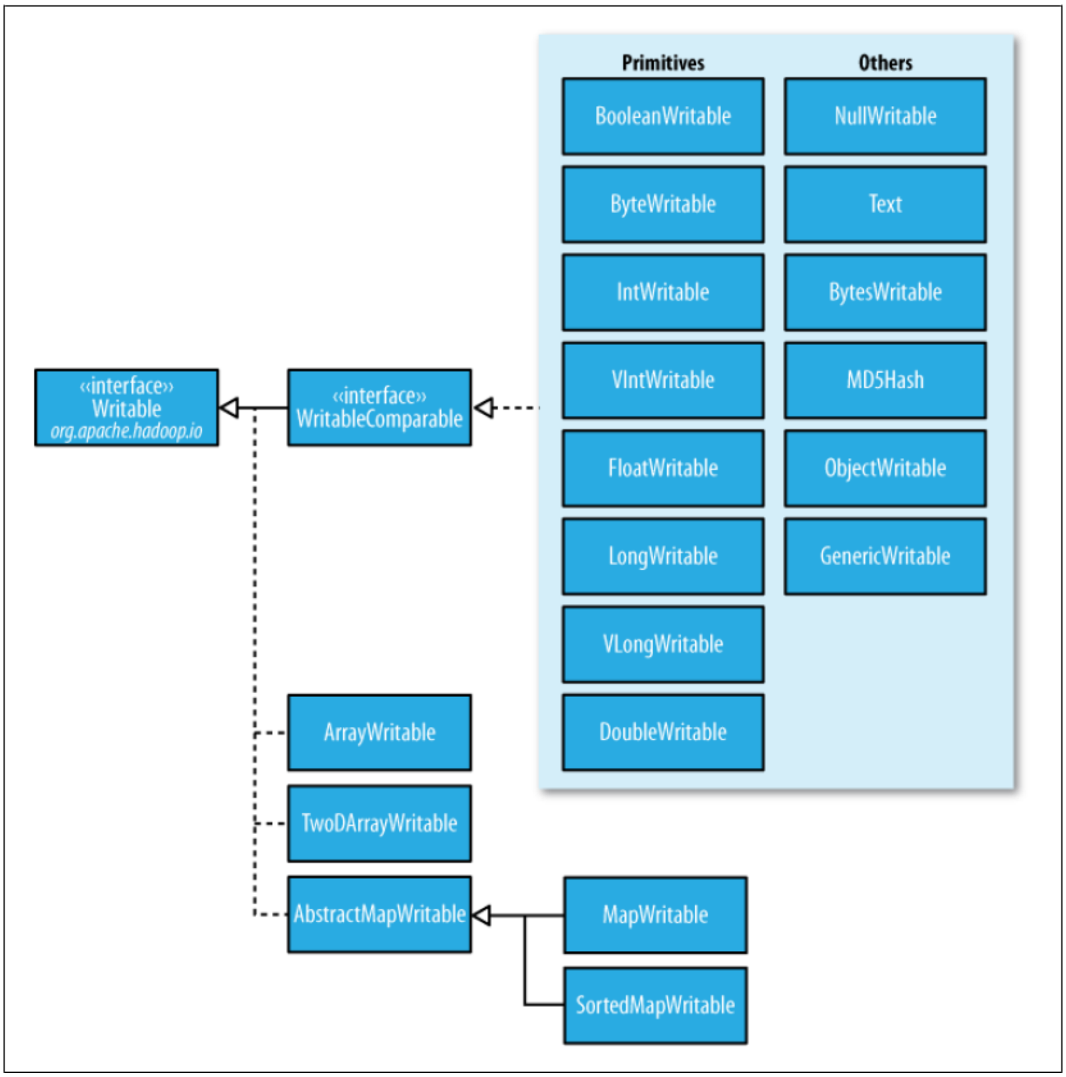

Figure 1. Writable class hierarchy

Writable wrappers for Java primitives

There are Writable wrappers for all the Java primitive types except char (which can be stored in an IntWritable) as shown in Table 1. All have a get() and a set() method for retrieving and storing the wrapped value.

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

Table 1. Writable wrapper classes for java primitives

When encoding integers, there is a choice between the fixed-length formats (IntWritable and LongWritable) and the variable-length formats (VIntWritable and VLongWritable). The variable-length formats use only a single byte to encode the value if it is small enough (between –112 and 127, inclusive); otherwise, they use the first byte to indicate whether the value is positive or negative, and how many bytes follow.
For example, 163 requires two bytes:

```
byte[] data = serialize(new VIntWritable(163));
assertThat(StringUtils.byteToHexString(data), is("8fa3"));
```

Fixedlength encodings are good when the distribution of values is fairly uniform across the whole value space, such as a (well-designed) hash function. Most numeric variables tend to have non uniform distributions, and on average the variable-length encoding will save space. Another advantage of variable-length encodings is that you can switch from VIntWritable to VLongWritable

Text

Text is a Writable for UTF-8 sequences. It can be thought of as the Writable equivalent of java.lang.String. The Text class uses an int (with a variable-length encoding) to store the number of bytes in the string encoding, so the maximum value is 2 GB.

Indexing

Because of its emphasis on using standard UTF-8, there are some differences between Text and the Java String class.
Here is an example to demonstrate the use of the charAt() method:

```
Text t = new Text("hadoop");
assertThat(t.getLength(), is(6));
assertThat(t.getBytes().length, is(6));
assertThat(t.charAt(2), is((int) 'd'));
assertThat("Out of bounds", t.charAt(100), is(-1));
```

Notice that charAt() returns an int representing a Unicode code point, unlike the String variant that returns a char. Text also has a find() method, which is analogous to String's indexOf():

```
Text t = new Text("hadoop");
assertThat("Find a substring", t.find("do"), is(2));
assertThat("Finds first 'o'", t.find("o"), is(3));
assertThat("Finds 'o' from position 4 or later", t.find("o", 4), is(4));
assertThat("No match", t.find("pig"), is(-1));
```

Unicode:

When we start using characters that are encoded with more than a single byte, the differences between Text and String become clear. Consider the Unicode characters shown in Table2.

Unicode code point	U+0041	U+00DF	U+6771	U+10400
Name	LATIN CAPITAL LETTER A	LATIN SMALL LETTER SHARP S	N/A (a unified Han ideograph)	DESERET CAPITAL LETTER LONG I
UTF-8 code units	41	c3 9f	e6 9d b1	f0 90 90 80
Java representation	\u0041	\u00DF	\u6771	\uuD801\uDC00

Table 2. Unicode characters

All but the last character in the table, U+10400, can be expressed using a single Java char. U+10400 is a supplementary character and is represented by two Java chars, known as a surrogate pair. The following example show the differences between String and Text when processing a string of the four characters from Table 2.

```
public class StringTextComparisonTest {
@Test
public void string() throws UnsupportedEncodingException {
String s = "\u0041\u00DF\u6771\uD801\uDC00";
assertThat(s.length(), is(5));
assertThat(s.getBytes("UTF-8").length, is(10));
assertThat(s.indexOf("\u0041"), is(0));
assertThat(s.indexOf("\u00DF"), is(1));
assertThat(s.indexOf("\u6771"), is(2));
assertThat(s.indexOf("\uD801\uDC00"), is(3));
assertThat(s.charAt(0), is('\u0041'));
assertThat(s.charAt(1), is('\u00DF'));
assertThat(s.charAt(2), is('\u6771'));
assertThat(s.charAt(3), is('\uD801'));
assertThat(s.charAt(4), is('\uDC00'));
assertThat(s.codePointAt(0), is(0x0041));
assertThat(s.codePointAt(1), is(0x00DF));
assertThat(s.codePointAt(2), is(0x6771));
assertThat(s.codePointAt(3), is(0x10400));
}
@Test
public void text() {
Text t = new Text("\u0041\u00DF\u6771\uD801\uDC00");
assertThat(t.getLength(), is(10));
assertThat(t.find("\u0041"), is(0));
assertThat(t.find("\u00DF"), is(1));
assertThat(t.find("\u6771"), is(3));
assertThat(t.find("\uD801\uDC00"), is(6));
assertThat(t.charAt(0), is(0x0041));
assertThat(t.charAt(1), is(0x00DF));
assertThat(t.charAt(3), is(0x6771));
assertThat(t.charAt(6), is(0x10400));
}
}
```

Example . Tests showing the differences between the String and Text classes

The test confirms that the length of a String is the number of char code units it contains (5, one from each of the first three characters in the string, and a surrogate pair from the last), whereas the length of a Text object is the number of bytes in its UTF-8 encoding (10 = 1+2+3+4). Similarly, the indexOf() method in String returns an index in char code units, and find() for Text is a byte offset.
The charAt() method in String returns the char code unit for the given index, which in the case of a surrogate pair will not represent a whole Unicode character. The codePointAt() method, indexed by char code unit, is needed to retrieve a single Unicode character represented as an int. In fact, the charAt() method in Text is more like the codePointAt() method than its namesake in String. The only difference is that it is indexed by byte offset.

Iteration

Iterating over the Unicode characters in Text is complicated by the use of byte offsets for indexing, since you can't just increment the index. Turn the Text object into a java.nio.ByteBuffer, then repeatedly call the bytesToCodePoint() static method on Text with the buffer. This method extracts the next code point as an int and updates the position in the buffer. The end of the string is detected when bytesToCodePoint() returns –1. See the following example.

```
public class TextIterator
{
public static void main(String[] args)
{
Text t = new Text("\u0041\u00DF\u6771\uD801\uDC00");
ByteBuffer buf = ByteBuffer.wrap(t.getBytes(), 0, t.getLength());
int cp;
while (buf.hasRemaining() && (cp = Text.bytesToCodePoint(buf)) != -1)
{
System.out.println(Integer.toHexString(cp));
}
}
}
```

Example . Iterating over the characters in a Text object

Running the program prints the code points for the four characters in the string:

```
% hadoop TextIterator
41
df
6771
10400
```

Another difference with String is that Text is mutable. We can reuse a Text instance by calling one of the set() methods on it. For example:

```
Text t = new Text("hadoop");
t.set("pig");
assertThat(t.getLength(), is(3));
assertThat(t.getBytes().length, is(3));
```

Resorting to String Text doesn't have as rich an API for manipulating strings as java.lang.String, so in many cases, you need to convert the Text object to a String. This is done in the usual way, using the toString() method:

```
assertThat(new Text("hadoop").toString(), is("hadoop"));
```

BytesWritable

BytesWritable is a wrapper for an array of binary data. Its serialized format is an integer field (4 bytes) that specifies the number of bytes to follow, followed by the bytes themselves. For example, the byte array of length two with values 3 and 5 is serialized as a 4-byte integer (00000002) followed by the two bytes from the array (03 and 05):

```
BytesWritable b = new BytesWritable(new byte[] { 3, 5 });
byte[] bytes = serialize(b);
assertThat(StringUtils.byteToHexString(bytes), is("000000020305"));
```

BytesWritable is mutable, and its value may be changed by calling its set() method.

NullWritable

NullWritable is a special type of Writable, as it has a zero-length serialization. No bytes are written to, or read from, the stream. It is used as a placeholder; for example, in MapReduce, a key or a value can be declared as a NullWritable when you don't need to use that position—it effectively stores a constant empty value. NullWritable can also be useful as a key in SequenceFile when you want to store a list of values, as opposed to key-value pairs.

ObjectWritable and GenericWritable

ObjectWritable is a general-purpose wrapper for the following: Java primitives, String, enum, Writable, null, or arrays of any of these types.
GenericWritable is useful when a field can be of more than one type: for example, if the values in a SequenceFile have multiple types, then you can declare the value type as an GenericWritable and wrap each type in an GenericWritable.

Writable collections

There are six Writable collection types in the org.apache.hadoop.io package: Array Writable, ArrayPrimitiveWritable, TwoDArrayWritable, MapWritable, SortedMapWritable, and EnumSetWritable.
ArrayWritable and TwoDArrayWritable are Writable implementations for arrays and two-dimensional arrays (array of arrays) of Writable instances. All the elements of an ArrayWritable or a TwoDArrayWritable must be instances of the same class, which is specified at construction, as follows:

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

In contexts where the Writable is defined by type, such as in SequenceFile keys or values, or as input to MapReduce in general, you need to subclass ArrayWritable (or TwoDArrayWritable, as appropriate) to set the type statically. For example:

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

ArrayWritable and TwoDArrayWritable both have get() and set() methods, as well as a toArray() method, which creates a shallow copy of the array.

ArrayPrimitiveWritable is a wrapper for arrays of Java primitives. The component type is detected when you call set(), so there is no need to subclass to set the type.
MapWritable and SortedMapWritable are implementations of java.util.Map<Writable,Writable> and java.util.SortedMap<WritableComparable, Writable>, respectively. Here's a demonstration of using a MapWritable with different types for keys and values:

```
MapWritable src = new MapWritable();
src.put(new IntWritable(1), new Text("cat"));
src.put(new VIntWritable(2), new LongWritable(163));
MapWritable dest = new MapWritable();
WritableUtils.cloneInto(dest, src);
assertThat((Text) dest.get(new IntWritable(1)), is(new Text("cat")));
assertThat((LongWritable) dest.get(new VIntWritable(2)), is(new
LongWritable(163)));
```

Conspicuous by their absence are Writable collection implementations for sets and lists. A general set can be emulated by using a MapWritable (or a SortedMapWritable for a sorted set), with NullWritable values. There is also EnumSetWritable for sets of enum types. For lists of a single type of Writable, ArrayWritable is adequate, but to store different types of Writable in a single list, you can use GenericWritable to wrap the elements in an ArrayWritable.

Implementing a Custom Writable

Hadoop comes with a useful set of Writable implementations that serve most purposes; however, on occasion, you may need to write your own custom implementation. nWith a custom Writable, you have full control over the binary representation and the sort order. Because Writables are at the heart of the MapReduce data path, tuning the binary representation can have a significant effect on performance. To demonstrate how to create a custom Writable, we shall write an implementation that represents a pair of strings, called TextPair. The basic implementation is shown in following Example.

```
import java.io.*;
import org.apache.hadoop.io.*;
public class TextPair implements WritableComparable<TextPair>
{
private Text first;
private Text second;
public TextPair() {
set(new Text(), new Text());
}
public TextPair(String first, String second)
{
set(new Text(first), new Text(second));
}
public TextPair(Text first, Text second)
{
set(first, second);
}
public void set(Text first, Text second)
{
```

```
this.first = first;
this.second = second;
}
public Text getFirst()
{
return first;
}
public Text getSecond()
{
return second;
}
@Override
public void write(DataOutput out) throws IOException
{
first.write(out);
second.write(out);
}
@Override
public void readFields(DataInput in) throws IOException
{
first.readFields(in);
second.readFields(in);
}
@Override
public int hashCode()
{
return first.hashCode() * 163 + second.hashCode();
}
@Override
public boolean equals(Object o)
{
if (o instanceof TextPair) {
TextPair tp = (TextPair) o;
return first.equals(tp.first) && second.equals(tp.second);
}
return false;
}
@Override
public String toString()
{
return first + "\t" + second;
}
@Override
public int compareTo(TextPair tp)
{
int cmp = first.compareTo(tp.first);
if (cmp != 0) {
return cmp;
}
```

```
return second.compareTo(tp.second);
}
}
```

Example . A Writable implementation that stores a pair of Text objects

The first part of the implementation is straightforward: there are two Text instance variables, first and second, and associated constructors, getters, and setters. All Writable implementations must have a default constructor so that the MapReduce framework can instantiate them, then populate their fields by calling readFields().

TextPair's write() method serializes each Text object in turn to the output stream, by delegating to the Text objects themselves. Similarly, readFields() deserializes the bytes from the input stream by delegating to each Text object. The DataOutput and DataInput interfaces have a rich set of methods for serializing and deserializing Java Primitives.

Just as you would for any value object you write in Java, you should override the hashCode(), equals(), and toString() methods from java.lang.Object. The hash Code() method is used by the HashPartitioner (the default partitioner in MapReduce) to choose a reduce partition, so you should make sure that you write a good hash function that mixes well to ensure reduce partitions are of a similar size.

If you ever plan to use your custom Writable with TextOutputFormat, then you must implement its toString() method. TextOutputFormat calls toString() on keys and values for their output representation. For TextPair, we write the underlying Text objects as strings separated by a tab character.

TextPair is an implementation of WritableComparable, so it provides an implementation of the compareTo() method that imposes the ordering you would expect: it sorts by the first string followed by the second.

Implementing a RawComparator for speed

In the above example, when TextPair is being used as a key in MapReduce, it will have to be deserialized into an object for the compareTo() method to be invoked. since TextPair is the concatenation of two Text objects, and the binary representation of a Text object is a variable-length integer containing the number of bytes in the UTF-8 representation of the string, followed by the UTF-8 bytes themselves. The trick is to read the initial length, so we know how long the first Text object's byte representation is; then we can delegate to Text's RawComparator, and invoke it with the appropriate offsets for the first or second string. Consider the following example for more details.

```
public static class Comparator extends WritableComparator
{
private static final Text.Comparator TEXT_COMPARATOR = new Text.Comparator();
public Comparator()
{
super(TextPair.class);
}
@Override
public int compare(byte[] b1, int s1, int l1,
byte[] b2, int s2, int l2)
{
try
{
int firstL1 = WritableUtils.decodeVIntSize(b1[s1]) + readVInt(b1, s1);
int firstL2 = WritableUtils.decodeVIntSize(b2[s2]) + readVInt(b2, s2);
int cmp = TEXT_COMPARATOR.compare(b1, s1, firstL1, b2, s2, firstL2);
```

```
if (cmp != 0)
{
return cmp;
}
return TEXT_COMPARATOR.compare(b1, s1 + firstL1, l1 - firstL1,
b2, s2 + firstL2, l2 - firstL2);
}
 catch (IOException e)
{
throw new IllegalArgumentException(e);
}
}
}
Static
 {
WritableComparator.define(TextPair.class, new Comparator());
}
```

Example 4-8. A RawComparator for comparing TextPair byte representations

We actually subclass WritableComparator rather than implement RawComparator directly, since it provides some convenience methods and default implementations. The subtle part of this code is calculating firstL1 and firstL2, the lengths of the first Text field in each byte stream. Each is made up of the length of the variable-length integer (returned by decodeVIntSize() on WritableUtils) and the value it is encoding (returned by readVInt()).
The static block registers the raw comparator so that whenever MapReduce sees the TextPair class, it knows to use the raw comparator as its default comparator.

Custom comparators

As we can see with TextPair, writing raw comparators takes some care, since you have to deal with details at the byte level. Custom comparators should also be written to be RawComparators, if possible. These are comparators that implement a different sort order to the natural sort order defined by the default comparator. The following Example a comparator for TextPair, called FirstComparator, that considers only the first string of the pair. Note that we override the compare() method that takes objects so both compare() methods have the same semantics.

```
public static class FirstComparator extends WritableComparator
{
private static final Text.Comparator TEXT_COMPARATOR = new Text.Comparator();
public FirstComparator() {
super(TextPair.class);
}
@Override
public int compare(byte[] b1, int s1, int l1,
byte[] b2, int s2, int l2)
{
Try
 {
int firstL1 = WritableUtils.decodeVIntSize(b1[s1]) + readVInt(b1, s1);
```

```
int firstL2 = WritableUtils.decodeVIntSize(b2[s2]) + readVInt(b2, s2);
return TEXT_COMPARATOR.compare(b1, s1, firstL1, b2, s2, firstL2);
}
 catch (IOException e)
{
throw new IllegalArgumentException(e);
}
}
@Override
public int compare(WritableComparable a, WritableComparable b)
{
if (a instanceof TextPair && b instanceof TextPair)
{
return ((TextPair) a).first.compareTo(((TextPair) b).first);
}
return super.compare(a, b);
}
}
```

Example . A custom RawComparator for comparing the first field of TextPair byte representations