IMPORTANT QUESTIONS

PART-A

Q.1 ***What is writable?***

Ans. 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.

Q.2 ***What is serialization?***

Ans. Serialization is the process of converting object data into byte stream data for transmission over a network across different nodes in a cluster or for persistent data storage.

Q.3 ***Define deserialization?***

Ans. Deserialization is the reverse process of serialization and converts byte stream data into object data for reading data from HDFS. Hadoop provides Writables for serialization and deserialization purpose.

Q.4 ***Write the specification of writable interface.***

Ans.

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

Q.5 ***Write the specification of Writable Comparable.***

Ans. WritableComparable interface is sub-interface of Hadoop's Writable and Java's Comparable interfaces and its specification is shown below:

```
public interface writableComparable extends writable, Comparable
{
}
```

The standard java.lang.Comparable Interface contains single method compareTo() method for comparing the operators passed to it.

```
public interface Comparable
{
public int compareTo(Object obj);
}
```

The compareTo( ) method returns –1, 0, or 1 depending on whether the compared object is less than, equal to, or greater than the current object.

Q.6 ***How is Hadoop related to Big Data?***

Ans. Hadoop is an open-source framework for storing processing, and analyzing complex unstructured data sets for deriving insights and intelligence.

Q.7 ***Why do we need Hadoop for Big Data Analytics?***

Ans. In most cases, Hadoop helps in exploring and analyzing large and unstructured data sets. Hadoop offers storage, processing and data collection capabilities that help in analytics.

Q.8 ***Explain the different features of Hadoop.***

Ans. Open-source : Hadoop is an open-sourced platform. It allows the code to be rewritten or modified according to user and analytics requirements.

Scalability : Hadoop supports the addition of hardware resources to the new nodes.

Data Recovery : Hadoop follows replication which allows the recovery of data in the case of any failure.

Data Locality : This means that Hadoop moves the computation to the data and not the other way round. This way, the whole process speeds up.

Q.9 ***How WritableComparable can be implemented in Hadoop?***

Ans. 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)
}
```

PART-B

Q.10 ***Write short note on hadoop I/O.***

Ans. Hadoop comes with a set of primitives for data I/O. Some of these are techniques that are more general than Hadoop, such as data integrity and compression, but deserve special consideration when dealing with multiterabyte datasets. Others are Hadoop tools or APIs that form the building blocks for developing distributed systems, such as serialization frameworks and on-disk data structures.

Data Integrity

Users of Hadoop rightly expect that no data will be lost or corrupted during storage or processing. However, since every I/O operation on the disk or network carries with it a small chance of introducing errors into the data that it is reading or writing, when the volumes of data flowing through the system are as large as the ones Hadoop is capable of handling, the chance of data corruption occurring is high.

The usual way of detecting corrupted data is by computing a checksum for the data when it first enters the system, and again whenever it is transmitted across a channel that is unreliable and hence capable of corrupting the data. The data is deemed to be corrupt if the newly generated checksum doesn't exactly match the original. This technique doesn't offer any way to fix the data—merely error detection. (And this is a reason for not using low-end hardware; in particular, be sure to use ECC memory.) Note that it is possible that it's the checksum that is corrupt, not the data, but this is very unlikely, since the checksum is much smaller than the data.

A commonly used error-detecting code is CRC-32 (cyclic redundancy check), which computes a 32-bit integer checksum for input of any size.

Data Integrity in HDFS

HDFS transparently checksums all data written to it and by default verifies checksums when reading data. A separate checksum is created for every 10 bytes per checksum bytes of data. The default is 512 bytes, and since a CRC-32 checksum is 4 bytes long, the storage overhead is less than 1%.

Datanodes are responsible for verifying the data they receive before storing the data and its checksum. This applies to data that they receive from clients and from other datanodes during replication. A client writing data sends it to a pipeline of datanodes, and the last datanode in the pipeline verifies the checksum. If it detects an error, the client receives a ChecksumException, a subclass of IOException, which it should handle in an application-specific manner, by retrying the operation, for example.

When clients read data from datanodes, they verify checksums as well, comparing them with the ones stored at the datanode. Each datanode keeps a persistent log of checksum verifications, so it knows the last time each of its

blocks was verified. When a client successfully verifies a block, it tells the datanode, which updates its log. Keeping statistics such as these is valuable in detecting bad disks.

Aside from block verification on client reads, each datanode runs a DataBlockScanner in a background thread that periodically verifies all the blocks stored on the datanode. This is to guard against corruption due to "bit roc" in the physical storage media.

Since HDFS stores replicas of blocks, it can "heal" corrupted blocks by copying one of the good replicas to produce a new, uncorrupt replica. The way this works is that if a client detects an error when reading a block, it reports the bad block and the datanode it was trying to read from to the namenode before throwing a ChecksumException. The namenode marks the block replica as corrupt, so it doesn't direct clients to it, or try to copy this replica to another datanode. It then schedules a copy of the block to be replicated on another datanode, so its replication factor is back at the expected level. Once this has happened, the corrupt replica is deleted.

Q.11 ***Explain the following:***

(a) localfilesystem

(b) checksumfilesystem

Ans.(a) LocalFileSystem : The Hadoop LocalFileSystem performs client-side checksumming. This means that when you write a file called filename, the filesystem client transparently creates a hidden file,. filename.crc, in the same directory containing the checksums for each chunk of the file. Like HDFS, the chunk size is controlled by the 10 bytes per checksumproperty, which defaults to 512 bytes. The chunk size is stored as metadata in the .crc file, so the file can be read back correctly even if the setting for the chunk size has changed. Checksums are verified when the file is read, and if an error is detected, LocalFileSystem throws a ChecksumException.

Checksums are fairly cheap to compute (in Java, they are implemented in native code), typically adding a few percent overhead to the time to read or write a file. For most applications, this is an acceptable price to pay for data integrity. It is, however, possible to disable checksums: typically when the underlying filesystem supports checksums natively. This is accomplished by using RawLocalFileSystem in place of LocalFileSystem. To do this globally in an application, it suffices to remap the implementation for file URIs by setting the property fs.file.impl to the value org.apache.hadoop.fs .RawLocalFileSystem, Alternatively, you can directly create a Raw LocalFileSystem instance, which may be useful if you want to disable checksum verification for only some reads; for example:

```
Configuration conf = ...
FileSystem fs = new RawLocalFileSystem();
fs, initialize(null, conf);
Configuration conf = ...
FileSystem fs = new RawLocalFileSystem();
fs, initialize(null, conf);
```

Ans.(b) ChecksumFileSystem

LocalFileSystem uses ChecksumFileSystem to do its work, and this class makes it easy to add checksumming to other (nonchecksummed) filesystems, as ChecksumFileSystem is just a wrapper around FileSystem. The general idiom is as follows:

```
FileSystem rawFs = ...
FileSystem checksummedFs = new
ChecksumFileSystem(rawFs);
```

The underlying filesystem is called the raw filesystem, and may be retrieved using the getRawFileSystem() method on ChecksumFileSystem. ChecksumFileSystem has a few more useful methods for working with checksums, such as getChecksumFile() for getting the path of a checksum file for any file. Check the documentation for the others. If an error is detected by ChecksumFileSystem when reading a file, it will call its reportChecksumFailure() method. The default implementation does nothing, but LocalFileSystem moves the offending file and its checksum to a side directory on the same device called bad_files. Administrators should periodically check for these bad files and take action on them.

Q.12 ***Explain writable collections.***

Ans. Writable collections

There are six Writable collection types in the org.apache.hadoop.io package: Array Writable, Array PrimitiveWritable, TwoDArrayWritable, MapWritable, Sorted MapWritable, 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 (or 2D 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. The type of each key and value field is a part of the serialization format for that field. The type is stored as a single byte that acts as an index into an array of types. The array is populated with the standard types in the org.apache.hadoop.io package, but custom Writable types are accommodated, too, by writing a header that encodes the type array for nonstandard types. As they are implemented, MapWritable and SortedMapWritable use positive byte values for custom types, so a maximum of 127 distinct nonstandard Writable classes can be used in any particular MapWritable or SortedMapWritable instance. Here's a demonstration of using a MapWritable with different types for keys and values:

```
Mapiiritable src = new MapWritable();
src.put(new IntWritable(1), new Text("cat"));
src.put(new VIntWritable(2), new LongWritable(163));
Mapiritable 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. Alternatively, you could write a general ListWritable using the ideas from MapWritable.

Q.13 ***Explain custom comparator.***

Ans. Custom Comparators : Writing rawcomparators takes some care, since you have to deal with details at the byte level. It is worth looking at some of the implementations of Writable in the org.apache .hadoop.io package for further ideas, if you need to write your own. The utility methods on WritableUtils are very handy, too.

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. Example shows a comparator for TextPatr, called First Comparator, that considers only the first string of the pair. Note that we override the compare() method that cakes objects so both compare() methods have the same semantics.

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

Solution :

```
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);
}
}
```

Q.14 ***Explain why we don't use java object serialization?***

Ans. Java comes with its own serialization mechanism, called Java Object Serialization (often referred to simply as "Java Serialization'), that is tightly integrated with the language, so it's natural to ask why this wasn't used in Hadoop. Here's what Doug Cutting said in response to that question: Why didn't I use Serialization when we first started Hadoop? Because it looked big and hairy and I thought we needed something lean and mean, where we had precise control over exactly how objects are written and read, since that is central to Hadoop. With Serialization you can get some control, but you have to fight for it.

The logic for not using RMI was similar. Effective, high-performance inter-process communications are critical to Hadoop. I felt like we'd need to precisely control how things like connections, timeouts and buffers are handled, and RMI gives you little control over those. The problem is that Java Serialization doesn't meet the criteria for a serialization format listed earlier: compact, fast, extensible, and interoperable.

Java Serialization is not compact: it writes the classname of each object being written to the stream this is true of classes that implement java.io.Serializable or java.io.Externalizable. Subsequent instances of the same class write a reference handle to the first occurrence, which occupies only 5 bytes. However, reference handles don't work well with random access, since the referent class may occur at any point in the preceding stream that is, there is state stored in the stream. Even worse, reference handles play havoc with sorting records in a serialized stream, since the first record of a particular class is distinguished and must be treated as a special case.

All these problems are avoided by not writing the classname to the stream at all, which is the approach that Writable takes. This makes the assumption that the client knows the expected type. The result is that the format is considerably more compact than Java Serialization, and random access and sorting work as expected since each record is independent of the others (so there is no stream state).

Java Serialization is a general-purpose mechanism for serializing graphs of objects, so it necessarily has some overhead for serialization and deserialization operations. What's more, the deserialization procedure creates a new instance for each object deserialized from the stream. Writable objects, on the other hand, can be (and often are) reused. For example, for a MapReduce job, which at its core serializes and deserializes billions of records of just a handful of different types, the savings gained by not having to allocate new objects are significant.

In terms of extensibility, Java Serialization has some support for evolving a type, but it is brittle and hard to use effectively (Writables have no support: the programmer has to manage them himself).In principle, other languages could interpret the Java Serialization stream protocol (defined by the Java Object Serialization Specification), but in practice there are no widely used implementations in other languages, so it is a Java- only solution. The situation is the same for Writables.

Q.15 ***Why do we need WritableComparable?***

OR

What happens if WritableComparable is not present?

Ans. 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.

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.

Q.16 ***How to make our custom type, WritableComparable?***

Ans. 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.

PART-C

Q.17 ***Explain the significance of Writable interface along with Writable Comparable and comparators w.r.to implementing the serialization.***

Ans. 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 tender 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.Byte ArrayOutputStream in a java.io.Data OutputStream 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();
tetum 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(new Writable.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(Int Writable.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));
```

Q.18 ***Explain the Writable class hierarchy with a neat sketch.***

Ans. Java is an object-oriented language and can view everything as an object. A simple file can be treated as an object (with java.io.File), an image can be treated as an object (with java.awt.Image) and a simple data type can be converted into an object (with wrapper classes).

Wrapper classes are used to convert any data type into an object.

The primitive data types are not objects; they do not belong to any class; they are defined in the language itself. Sometimes, it is required to convert data types into objects in Java language. For example, upto JDK1.4, the data structures accept only objects to store. A data type is to be converted into an object and then added to a Stack or Vector etc. For this conversion, the designers introduced wrapper classes.

- Each of Java's eight primitive data types has a class dedicated to it. These are known as wrapper classes, because they "wrap" the primitive data type into an object of that class.
- The wrapper classes are part of the java.lang package, which is imported by default into all Java programs.

The following two statements illustrate the difference between a primitive data type and an object of a wrapper class:

```
int x = 25:
Integer y = new Integer(33);
```

The following two statements illustrate the difference between a primitive data type and an object of a wrapper class:

```
int x = 25:
Integer y = new Integer(33);
```

The first statement declares an int variable named x and initializes it with the value 25. The second statement instantiates an Integer object. The object is initialized with the value 33 and a reference to the object is assigned to the object variable y.

Below table lists wrapper classes in Java API with constructor details.

Table 1

Primitive	Wrapper class	Constructor argument
boolean	Boolean	boolean or String
byte	Byte	byte or String
char	Character	char
int	Integer	int or String
float	Float	float, double or String
double	Double	double or String
long	Long	long or String
short	Short	short or String

Below is wrapper class hierarchy as per Java API.

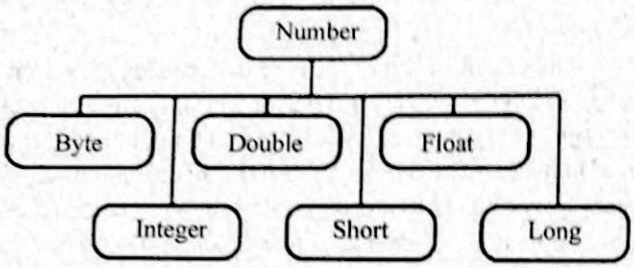

Fig. 1

As explain in above table all wrapper classes (except Character) take String as argument constructor. Please note we might get NumberFormatException if we try to assign invalid argument in constructor. For example to create Integer object we can have following syntax.

Integer intObj = new Integer (25);

Integer intObj2 = new Integer ("25");

Here, we can provide any number as string argument but not the words etc. Below statement will throw run time exception (NumberFormatException)

Integer intObj3 = new Integer ("Two");

The following discussion focuses on the Integer wrapper class, but applies in a general sense to all eight wrapper classes.

The most common methods of the Integer wrapper class are summarized in below table. Similar methods for the other wrapper classes are found in the Java API documentation.

Table 2

Method	Purpose
parseInt(s)	returns a signed decimal integer value equivalent to string s
toString(i)	returns a new String object representing the integer i
byteValue()	returns the value of this Integer as a byte
doubleValue()	returns the value of this Integer as a double
floatValue()	returns the value of this Integer as a float
intValue()	returns the value of this Integer as an int
shortValue()	returns the value of this Integer as a short
longValue()	returns the value of this Integer as a long
int compareTo (int i)	Compares the numerical value of the invoking object with that of i Returns 0 if the values are equal. Returns a negative value if the invoking object has a lower value. Returns a positive value if the invoking object has a greater value.
static int compare(int num1, int num2)	Compares the values of num1 and num2. Returns 0 if the values are equal. Returns a negative value if num1 is less than num2. Returns a positive value if num1 is greater than num2.
booleanequals (Object intObj)	Returns true if the invoking Integer object is equivalent to intObj. Otherwise, it returns false.

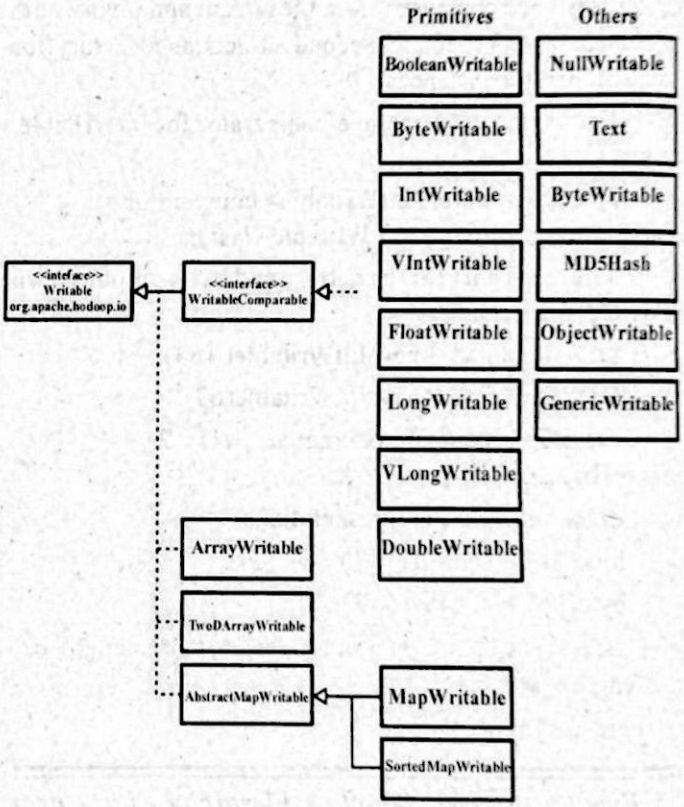

Fig. 2 : Writable class hierarchy

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

Table 3 : Writable Wrapper Classes 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

When it comes to 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"));

How do you choose between a fixed-length and a variable-length encoding? Fixed-length 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 nonuniform 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, since their encodings are actually the same. So by choosing a variable-length representation, you have room to grow without committing to an 8-byte long representation from the beginning.

Q.19 *Explain compression and codec in hadoop in detail.*

Ans. Compression : File compression brings two major benefits: it reduces the space needed to store files, and it speeds up data transfer across the network, or to or from disk. When dealing with large volumes of data, both of these savings can be significant, so it pays to carefully consider how to use compression in Hadoop.

There are many different compression formats, tools and algorithms, each with different characteristics. Table 1 lists some of the more common ones that can be used with Hadoop.

Table 1 : A Summary of Compression Formats

Compression format	Tool	Algorithm	Filename extension	Split table
DEFLATE	N/A	DEFLATE	.deflate	No
gzip	gzip	DEFLATE	.gz	No
bzip2	bzip2	bzip2	.bz2	Yes
LZ0	lzop	LZ0	.lzo	No
Snappy	N/A	Snappy	.snappy	No

All compression algorithms exhibit a space/time trade-off: faster compression and de- compression speeds usually come at the expense of smaller space savings. The tools listed in Table 1 typically give some control over this trade-off at compression time by offering nine different options: -1 means optimize for speed and -9 means optimize for space. For example, the following command creates a compressed file file.gz using the fastest compression method:

gzip -1 file

The different tools have very different compression characteristics. Gzip is a general- purpose compressor, and sits in the middle of the space/time trade-off. Bzip2 compresses more effectively than gzip, but is slower. Bzip2's decompression speed is faster than its compression speed, but it is still slower than the other formats. LZO and Snappy, on the other hand, both optimize for speed and are around an order of magnitude faster than gzip, but compress less effectively. Snappy is also significantly faster than LZO for decompression.1

Codecs : A codec is the implementation of a compression-decompression algorithm. In Hadoop, a codec is represented by an implementation of the CompressionCodec interface. So, for example, GzipCodec encapsulates the compression and decompression algorithm for gzip. Table 2 lists the codecs that are available for Hadoop.

Table 2 : Hadoop compression codecs

Compression format	Hadoop CompressionCodec
DEFLATE	org.apache.hadoop. io. compress.DefaultCodec
gzip	org.apache.hadoop.io. compress.GzipCodec
bzip2	org.apache hadoop.io. compress.BZipzCodec
LZ0	com.hadoop.compression. *lz*0.LzopCodec
Snappy	org.apache.hadoop.io. compress.SnappyCodec

The LZO libraries are GPL-licensed and may not be included in Apache distributions, so for this reason the Hadoop codecs must be downloaded separately from hitp://code .google.com/p/hadoop-gp*l*-compression/ {or http://git hub.com/kevinweil/hadoop-*l*zo, which includes bugfixes and more tools}. The LzopCodec is compatible with the *l*zop tool, which is essentially the LZO format with extra headers, and is the one you normally want. There is also a LzoCodec for the pure LZO format, which uses the .lzo_de-flate filename extension (by analogy with DEFLATE, which is gzip without the headers).

Compressing and Decompressing Streams with CompressionCodec

CompressionCodec has two methods that allow you to easily compress or decompress data. To compress data being written to an output stream, use the createOutput Stream(OutputStream out) method to create a CompressionOutput Stream to which you write your uncompressed data to have it written in compressed form to the underlying stream. Conversely, to decompress data being read from an input stream, call createInputStream (InputStream in) to obtain a CompressionInputStream, which allows you to read uncompressed data from the underlying stream.

CompressionOutputStream and CompressionInput Stream are similar to Java.uti1.zip.DeflaterOutputStream and java.util.zip.Deflater InputStream, except that both of th former provide the ability ro reset their underlying compress or de-compressor, which is important for applications th compress sections of the data stream as separate blocks.

Example : A program to compress data read fro standard input and write it to standard output

```
public class StreamCompressor {
public static void main(String[] args) throws Exception {
String codecClassname = args[0];
Class<?> codecClass = Class. forName(codecClassname
Configuration conf = new Configuration();
CompressionCodec codec = (CompressionCodec)
ReflectionUtils .newInstance(codecClass, conf);
CompressionOutputStream out = codec.createoutputS
ream(System.out);
IOUtils.copyBytes(System.in, out, 4096, false);
out. finish();
}
}
```

The application expects the fully qualified name of th CompressionCodec implementation as the first command-lin argument. We use ReflectionUtils to construct a new instanc of the codec, then obtain a compression wrapper aroun System. out. Then we call the utility method copyBytes() o IOUtils to copy the input to the output, which is compresse by the CompressionOutputStream. Finally, we call finish(on CompressionOutput Stream, which tells the compresso to finish writing to the compressed stream, but doesn't clos the stream. We can try it out with the following comman line, which compresses the string "Text" using th StreamCompressor program with the GzipCodec, the decompresses it from standard input using gunzip:

```
% echo "Text" | hadoop StreamComprossor org
apache.hadoop.io.compress.GzipCodec \
| gunzip -
Text
```

Q.20 Write detailed note on writable wrapper class.

Ans. Text : Text is a Writable for UTF-8 sequences. It can be thought of as the writable equivalent of java.lang.String. Text is a replacement for the UTF8 class, which was deprecated because it didn't support strings whose encoding was over 32,767 bytes, and because it used Java's modified UTF-8.

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. Furthermore, Text uses standard UTF-8, which makes it potentially easier to interoperate with other tools that understand UTF-8.

Indexing : Because of its emphasis on using standard UTF-8, there are some differences between Text and the Java String class. Indexing for the Text class is in terms of position in the encoded byte sequence, not the Unicode character in the string, or the Java char code unit (as it is for String). For ASCII strings, these three concepts of index position coincide. 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 Table.

Table : Unicode characters

Unicode code point	U+0041	U+00DF	U+6771	U+10400
Nane	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	\uD801\uDC00

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 tests in Example 1 show the differences between String and Text when processing a string of the four characters from Table.

Example 1 : Tests showing the differences between the String and Text classes

```
public class StringTextComparisonTest {
@Test
public vold string() throws UnsupportedEncodingException {
String s = "\u0041\u00DF\u6771\uD801\uDC00";
assertThat(s.length(), is(5));
assertThat (s.getBytes("UTF-8"), length, is(10))};
assert That(s. indexOf("\u0041"), is(0));
assertThat(s.indexOf("\ua0DF"), is(1));
assertThat(s. indexOF("\u6771"}, is(2));
assertThat (s. indexOF("\uD8o1\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));
    }
}
```

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 code PointAt() 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. The idiom for iteration is a little obscure (see Example 2): turn the Text object int Java.nio.ByteBuffer, then repeatedly call t bytesToCodePoint() static method on Text with the buff This method extracts the next code point as an int and upda the position in the buffer. The end of the string is detec when bytesToCodePoint() returns –1.

Example 2 : Iterating over the characters in a Te object

```
public class TextIterator {

public static void main(String[] args) {
Text t = new Text ("\u0041\u00DF\u6771\uD801\ UDC00"

ByteBuffer buf = ByteBuffer.wrap(t.getBytes(), (
t.getLength());
int cp;
while (buf.hasRemaining() && (cp =
Text.bytesToCodePoint(buf)) != -1) {
System. out .println(Integer. toHexString(cp));
    }
    }
}
```

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

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

Mutability : Another difference with String is that Text is mutable (like all writable implementations in Hadoop, excep NullWritable, which is a singleton). You can reuse a Tex 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));
```

In some situations, the byte array returned by the getBytes() method may be longer than the length returned by getLength( ):

```
Text t = new Text("hadoop");
t.set(new Text("pig"));
assert That(t.getlength(), is(3));
assertThat("Byte length not shortened", t.getBytes(). length,
is(6));
```

This shows why it is imperative that you always call getLength() when calling getBytes(), so you know how much of the byte array is valid data.

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. As with Text, the size of the byte array returned from the getBytes() method for BytesWritable—the capacity—may not reflect the actual size of the data stored in the BytesWritable. You can determine the size of the BytesWritable by calling get Length(). To demonstrate:

```
b.setCapacity(11);
assertThat(b.getLength(), is(2));
assertThat (b.getBytes().length, is(11));
```

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. It is an immutable singleton: the instance can be retrieved by calling NullWritable.get().

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. It is used in Hadoop RPC to marshal and unmarshal method arguments and return types.

ObjectWritable 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 Objectwritable and wrap each type in an ObjectWritable. Being a general-purpose mechanism, it's fairly wasteful of space since it writes the classname of the wrapped type every time it is serialized. In cases where the number of types is small and known ahead of time, this can be improved by having a static array of types, and using the index into the array as the serialized reference to the type. This is the approach that GenericWritable takes, and you have to subclass it to specify the types to support.

Q.21 ***Explain how to implement a custom variable?***

Ans. 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;
ptivate 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()
{
retum second:
}
@Override
public void write(DataOutput out) throws IOExceptio
{
first.write(out);
second. write(out);
}
@Override
public void readFields(DataInput in) throw
IOException
{
first.readFields(in);
second.readFields(in);
}
@Override
public int hashCode()
{
return first.hashCode() * 163 + second.hashCode();
}
@Override
public boolean equals(Object o)
{
@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.

Q.22 ***Explain serializing and deserializing the data in hadoop. Also give advantages and disadvantages of hadoop serialization.***

Ans. Serializing the Data in Hadoop : The procedure to serialize the integer type of data is discussed below :

- Instantiate IntWritable class by wrapping an integer value in it.
- Instantiate ByteArrayOutput Stream class.
- Instantiate DataOutputStream class and pass the object of ByteArrayOutputStream class to it.
- Serialize the integer value in IntWritable object using write() method. This method needs an object of DataOutputStream class.
- The serialized data will be stored in the byte array object which is passed as parameter to the DataOutputStream class at the time of instantiation. Convert the data in the object to byte array.

Example : The following example shows how to serialize data of integer type in Hadoop -

```
import java.io.ByteArrayOutputStream;
import java.io.DataOutputStream;
import java.io. IOException;
import org.apache.hadoop.io.IntWritable;
public class Serialization {
public byte[] serialize() throws IOException{
//Instantiating the IntWritable object
IntWritable intwritable = new IntWritable(12);
//Instantiating ByteArrayOutputStream object
```

```
ByteArrayOutputStream byteoutputStream = new
ByteArrayOutputStream();
//Instantiating DataOutputStream object
DataOutputStream dataOutputStream = new
DataOutputStream(byteoutputStream) ;
//Serializing the data
intwritable.write(dataOutputStream) ;
//storing the serialized object in bytearray
byte[] byteArray = byteoutputStream. toByteArray() ;
//Closing the OutputStream
dataOutputStream. close();
return(byteArray) ;
}
public static void main(String args[]) throws
IOException{
Serialization serialization= new Serialization();
serialization.serialize();
System.out.println();
}
```

Deserializing the Data in Hadoop

The procedure to deserialize the integer type of data is discussed below -

- Instantiate IntWritable class by wrapping an integer value in it.
- Instantiate ByteArrayOutputStream class.
- Instantiate DataOutputStream class and pass the object of ByteArrayOutputStream class to it.
- Deserialize the data in the object of DataInputStream using readFields() method of IntWritable class.
- The deserialized data will be stored in the object of IntWritable class. You can retrieve this data using get() method of this class.

Example

The following example shows how to deserialize the data of integer type in Hadoop -

```
import java.io.ByteArrayInputStream;
import java.io.DataInputStream;
import org.apache.hadoop.io.IntWritable;
public class Deserialization {
public void deserialize(byte[]byteArray) throw
Exception{
//Instantiating the IntWritable class
IntWritable intwritable =new IntWritable();
//Instantiating ByteArrayInputStream object
ByteArrayInputStream InputStream = new
ByteArrayInputStream(byteArray) ;
//Instantiating DataInputStream object
DataInputStream datainputstream=new
DataInputStream(InputStream) ;
//deserializing the data in DataInputStream
intwritable.readFields(datainputstream) ;
//printing the serialized data
System. out. println((intwritable).get());
}
public static void main(String args[]) throws Exception {
Deserialization dese = new Deserialization();
dese. deserialize(new Serialization().serialize());
}
}
```

Advantage of Hadoop over Java Serialization

Hadoop's Writable-based serialization is capable of reducing the object-creation overhead by reusing the Writable objects, which is not possible with the Java's native serialization framework.

Disadvantages of Hadoop Serialization

To serialize Hadoop data, there are two ways —

- You can use the Writable classes, provided by Hadoop's native library.
- You can also use Sequence Files which store the data in binary format.

The main drawback of these two mechanisms is that Writables and SequenceFiles have only a Java API and they cannot be written or read in any other language.

Therefore any of the files created in Hadoop with above two mechanisms cannot be read by any other third language, which makes Hadoop as a limited box. To address this drawback, Doug Cutting created Avro, which is a language independent data structure.

Q.23 *How can Writables be Implemented in Hadoop?*

Ans. Writable Variables in Hadoop have the default Properties of Comparable.

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 as IntWritable 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.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.

❑❑❑