Serialization

- The Writable Interface
- Writable Classes
- Implementing a Custom Writable
- Serialization Frameworks

What is 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.

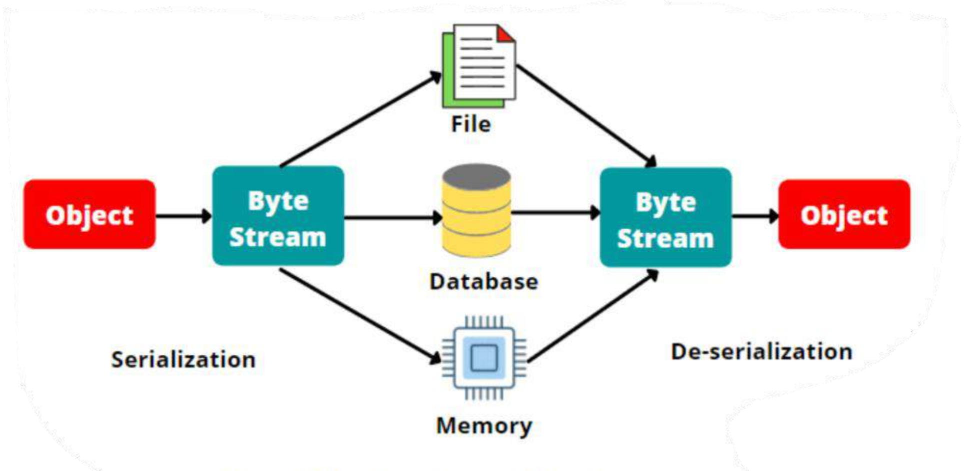


Fig: Serialization vs De-serialization process

Why do we Serialize the data?

Serialization is used in two quite distinct areas of distributed data processing:

1. For interprocess communication and
2. 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, it is desirable that an RPC serialization format is:

Compact: A compact format makes the best use of network bandwidth, which is the scarcest resource in a data centre.

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. For example, it should be possible to add a new argument to a method call and have the new servers accept messages in the old format (without the new argument) from old clients.

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.

The four desirable properties of an RPC's serialization format are also crucial for a persistent storage format. We want the storage format to be compact (to make efficient use of storage space), fast (so the overhead in reading or writing terabytes of data is minimal), extensible (so we can transparently read data written in an older format), and interoperable (so we can read or write persistent data using different languages).

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

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

The Writable interface defines two methods:

- One for writing its state to a DataOutput binary stream and
- One for reading its state from a DataInput binary stream

```
public interface Writable {
void write(DataOutput out) throws IOException;
void readFields(DataInput in) throws IOException;
}
```

```
public class Serialization {

   public static byte[] serialize(Writable writable) throws IOException  {

         /* 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 (an
         implementation of java.io.DataOutput) to capture the bytes in the serialized stream: */

         ByteArrayOutputStream out = new ByteArrayOutputStream();
         DataOutputStream dataOut = new DataOutputStream(out);
         writable.write(dataOut);
         dataOut.close();
         return out.toByteArray();
      }

   public static byte[] deserialize(Writable writable, byte[] bytes) throws IOException {

         /* we create a helper method to read a Writable object from a byte array: */
         ByteArrayInputStream in = new ByteArrayInputStream(bytes);
         DataInputStream dataIn = new DataInputStream(in);
         writable.readFields(dataIn);
         dataIn.close();
         return bytes;
         }


public static void main(String args[]) throws IOException{

                  /* 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);
                           byte[] bytes = serialize(writable);

                  /* The bytes are written in big-endian order */

                           System.out.println("The Converted Bytes for 163 :: "+bytes);


                    /* We construct a new, value-less IntWritable, and 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);
                           System.out.println("The deserialize data is :: "+newWritable.get());
         }
}
```

Writable Comparable and comparators

IntWritable implements the WritableComparable interface, which is just a sub interface of the Writable and java.lang.Comparable interfaces. 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)
}
```

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

```
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);
}
```

- Where b1,b2 are byte arrays, s1, s2 are starting positions, l1,l2 are lengths
- This interface permits implementors to compare records read from a stream without deserializing them into objects, there by avoiding any overhead of object creation.
- Writable comparator is a general purpose implementation of RawComparator for Writable Comparable classes
- It provides two main functions, first it provides default implementation of raw compare() method, second, it acts as a factory for RawComparator instances
- For example, to obtain a comparator for IntWritable, we just use
  RawComparator <IntWritable> c = WritableComparator.get(IntWritable.class);
- The Comparator can be used to compare two IntWritable objects

```
IntWritable w1 =new IntWritable(164);
IntWritable w2 =new IntWritable(67);
assertThat(c.compare(w1,w2), greaterThan(0));
```

Writable Classes

Hadoop comes with a large collection of Writable classes in the org. apache.hadoop.io package as shown in figure 1.

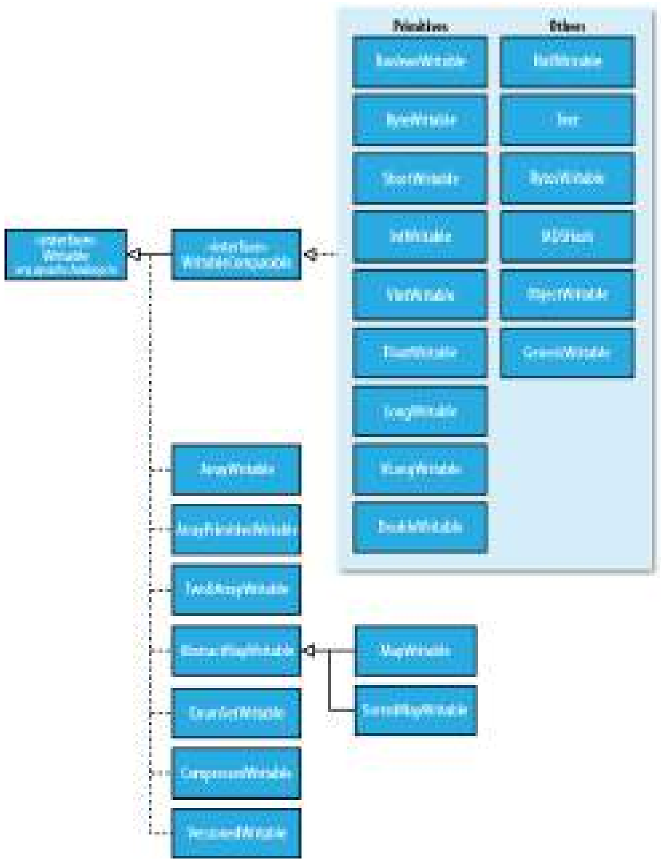
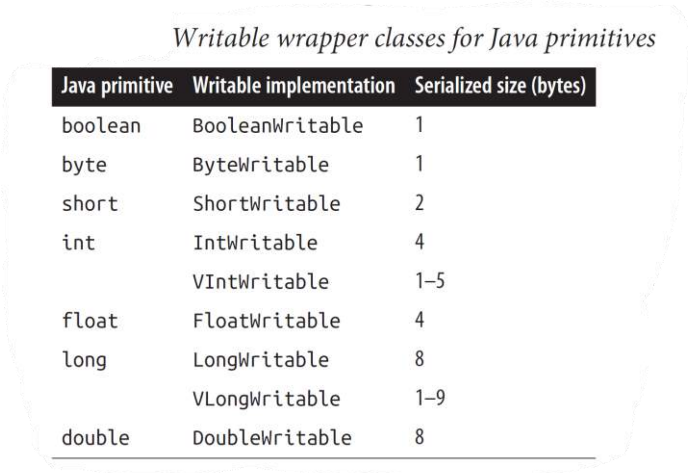


Figure 1. Writable Classes hierarchy

Writable wrappers for java primitives

- There are Writable wrappers for all the java primitive data types except char (which can be stored in IntWritable)
- All have a get() and set() method for retrieving and storing the wrapped value.



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

- IntWritable is a wrapper for java int.
- We can create IntWritable and set its value using the set() method

```
IntWritable iw= new IntWritable();
iw.set(154);
```

- Equivalently, we can use the constructor that takes the integer value.

```
IntWritable iw = new IntWritable(154);
```

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.

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 when using a (well-designed) hash function.
- Most numeric variables tend to have non uniform distributions, and on average variable length encoding will save space.
- Another advantage of variable length encoding is that you can switch from VIntWritable to VLongWritable, because their encodings are actually the same.

Text

- Text is a Writable for UTF-8 sequences (UTF - Unicode Transformation Format, 8-eight bit block)
- It is 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.
- Text uses standard UTF-8, which makes it potentially easier to interoperate with other tools that understand UTF-8.

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

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

  t.toString();

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 two values 3and 5 is serialized as 000000020305.

  BytesWritable b = new BytesWritable (new byte[] {3,5});
- BytesWritable is mutable , and its value may be changed by calling its set() method.

Null Writable

- 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
- In MapReduce, a key or a value can be declared as a NullWritable when you dont need to use that position, effectively storing a constant empty value.
- It is immutable singleton, and the instance can be retrieved by calling NullWritable.get()

Object Writable and Generic Writable

- ObjectWritable is a general-purpose wrapper for java primitives, String, enum, Writable, null or arrays of any of these types.
- ObjectWritable is useful when a field can be of more than one type.
- Being a general-purpose mechanism, it wasted a fair amount of space because it writes class name of the wrapped type every time it is serialized.
- If 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 aapoach can be taken in Generic writable.

Writable collections

There are six Writable collection types in the org.apache.hadoop.io package

- ArrayWritable
- ArrayPrimitiveWritable
- TwoDArrayWritable
- MapWritable
- SortedMapWritable
- EnumSetWritable

ArrayWritable and TwoDArrayWritable are Writable implementations for arrays and two dimensional arrays (array of arrays) of Writable instances.

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

if the Writable is defined by the type, you need to subclass to set the type statically.

```
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 createa 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.

  ```
  MapWritable src = new MapWritable();
  src.put(new IntWritable(1), new Text("cat"));
  src.put(new VIntWritable(2), new LongWritable(163));
  ```
- EnumSetWritable is used for sets of enum types.

Implementing a custom Writable

- If none of the built-in Hadoop Writable data types matches our requirements, then we can create custom Hadoop data type by implementing Writable interface or WritableComparable interface.
- With a custom Writable, you have full control over the binary representation and the sort order.
- Because Writables are at the heart of 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 the implementation that represents a pair of strings, called TextPair.

Prog: The Writable implementation that stores a pair of Text objects

```
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);
                    }
}
```

- TextPair is an implementation of WritableComparable, so it provides implementation of the compareTo() method that imposes the ordering i.e, it sorts by the first string followed by second string.

Implementing a Raw Comparator for speed

- We can further optimize the above code using RawComparator/WritableComparator
- Two TextPair objects can be compared without deserialization.

Prog: A Raw Comparator for comparing Text pair binary representations

```
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());
        }
```

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

- We can write our own comparators using some of the implementations of Writable in the org.apache.hadoop.io package
- The utility methods on WritableUtils are very handy.

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

```
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);
        }
}
```

Serialization Frameworks

- Although most MapReduce programs use Writable key and value types, this isn't mandated by the MapReduce API.
- In fact, any type can be used; the only requirement is a mechanism that translates to and from a binary representation of each type.
- To support this, Hadoop has an API for pluggable serialization frameworks.
- A serialization framework is represented by an implementation of Serialization.
- WritableSerialization, for example, is the implementation of Serialization for Writable types.
- A Serialization defines a mapping from types to Serializer instances (for turning an object into a byte stream) and Deserializer instances (for turning a byte stream into an object).
- Set the io.serializations property to a comma-separated list of classnames in order to register Serialization implementations.
- Its default value includes org.apache.hadoop.io.serializer.WritableSerialization and the Avro Specific and Reflect serializations, which means that only Writable or Avro objects can be serialized or deserialized out of the box.
- Hadoop includes a class called JavaSerialization that uses Java Object Serialization.
- Although it makes it convenient to be able to use standard Java types such as Integer or String in MapReduce programs, Java Object Serialization is not as efficient as Writables, so it's not worth making this trade-off.

File-Based Data Structures

- Sequence File
- Map File
- Other File Formats and Column-Oriented Formats

SequenceFile and MapFile are two file formats used in Hadoop to store binary key-value pairs.

Sequence File

What is a Hadoop Sequence File?

A Hadoop Sequence File is a binary file format that is designed for storing key-value pairs. It is widely used in Hadoop applications and is optimized for high-speed data processing. The Sequence File format is highly flexible and can be used for a variety of data types, including text, images, and audio files.

A Sequence File consists of two parts - a header and a data section. The header contains metadata about the file, such as the compression format and the key and value classes. The data section contains the actual key-value pairs, which are stored in a serialized format.

Writing a Sequence Files

- To create a SequenceFile, use one of its createWriter() static methods, which return a SequenceFile.Writer instance.
- There are several overloaded versions, but they all require you to specify a stream to write, a Configuration object, and the key and value types.
- The keys and values stored in a SequenceFile do not necessarily need to be Writables.
- Any types that can be serialized and deserialized by a Serialization may be used.
- Once you have a SequenceFile.Writer, you then write key-value pairs using the append() method.
- When you've finished, you call the close() method.

```
public class SequenceFileExample {
  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Path filePath = new Path("hdfs://your-hdfs-path/your-sequence-file.seq");

    // Define options for the SequenceFile writer
    Option fileOption = Writer.file(filePath);
    Option keyOption = Writer.keyClass(Text.class);
```

```
        Option valueOption = Writer.valueClass(IntWritable.class);

        // Create a SequenceFile writer
        Writer writer = SequenceFile.createWriter(conf, fileOption, keyOption, valueOption);

        // Write key-value pairs to the SequenceFile
        Text key = new Text("example_key");
        IntWritable value = new IntWritable(42);
        writer.append(key, value);

        // Close the writer
        writer.close();
    }
}
```

Reading a SequenceFile

Reading sequence files from beginning to end is a matter of creating an instance of SequenceFile.Reader and iterating over records by repeatedly invoking one of the next() methods.

```
public class SequenceFileReader {
    public static void main(String[] args) throws Exception {
        // Create a Hadoop Configuration object
        Configuration conf = new Configuration();

        // Specify the path to your SequenceFile on HDFS
        Path filePath = new Path("hdfs://your-hdfs-path/your-sequence-file.seq");

        // Open a SequenceFile for reading
        FileSystem fs = FileSystem.get(conf);
        SequenceFile.Reader reader = new SequenceFile.Reader(fs, filePath, conf);

        // Create variables to store key and value
        Text key = new Text();
        IntWritable value = new IntWritable();

        try {
            // Read key-value pairs from the SequenceFile
            while (reader.next(key, value)) {
                // Process the key and value (you can replace this with your logic)
                System.out.println("Key: " + key.toString() + ", Value: " + value.get());
            }
        } finally {
            // Close the reader
            reader.close();
        } } }
```

In this program:

- We start by creating a Hadoop Configuration object to specify Hadoop-related settings.
- We define the path to the SequenceFile you want to read from using the Path class.
- We open the SequenceFile for reading using SequenceFile.Reader. Make sure to provide the correct FileSystem, Path, and Configuration.
- We define variables of the appropriate types (Text for keys and IntWritable for values) to store the key-value pairs read from the SequenceFile.
- Inside the while loop, we use reader.next(key, value) to read the next key-value pair from the SequenceFile. You should replace the System.out.println line with your own logic to process the data.
- Finally, we close the SequenceFile.Reader in a finally block to ensure it's properly closed, even if an exception occurs during processing.

The SequenceFile format

These files contain data as binary key-value pairs. There are further 3 formats –

Uncompressed – No compression

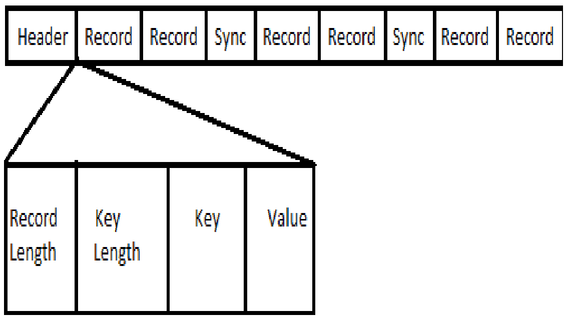
Record compressed – Records are compressed when they are added to file.

Block compressed – This format waits until data reaches to block size.

- A sequence file consists of a header followed by one or more records (Figure 1 & 2).
- The first three bytes of a sequence file are the bytes SEQ, which act as a magic number.
- These are followed by a single byte representing the version number.
- The header contains other fields, including the names of the key and value classes, compression details, user defined metadata, and the sync marker.
- The sync marker is used to allow a reader to synchronize to a record boundary from any position in the file.
- Each file has a randomly generated sync marker, whose value is stored in the header.
- Sync markers appear between records in the sequence file.
- The internal format of the records depends on whether compression is enabled, and if it is, whether it is record compression or block compression.
- If no compression is enabled, each record is made up of the record length (in bytes), the key length, the key, and then the value.

- The length fields are written as 4-byte integers adhering to the contract of the writeInt() method of java.io.DataOutput. Keys and values are serialized using the Serialization defined for the class being written to the sequence file.

In the below figures, two types of Sequence file format, first one is without compression and the second one is with compression. If sequence file is compressed, then only the value in the sequence file will be compressed but not the other metadata. The compression shown in the second figure is record level compression. Sequence file can also have block level compression.



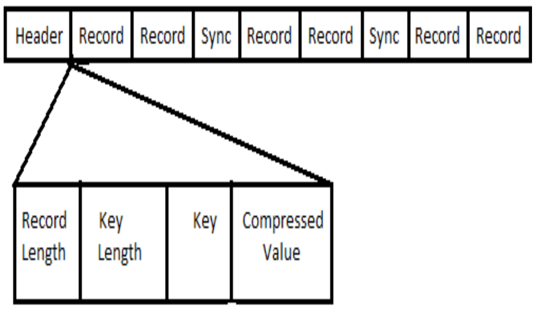


Figure 1. Structure of a sequence file with no compression **Figure 2. Structure of a sequence file with compression**

Block compression (Figure 3) compresses multiple records at once; it is therefore more compact than and should generally be preferred over record compression because it has the opportunity to take advantage of similarities between records. Records are added to a block until it reaches a minimum size in bytes, defined by the io.seqfile.compress.blocksize property; the default is one million bytes. A sync marker is written before the start of every block. The format of a block is a field indicating the number of records in the block, followed by four compressed fields: the key lengths, the keys, the value lengths, and the values.

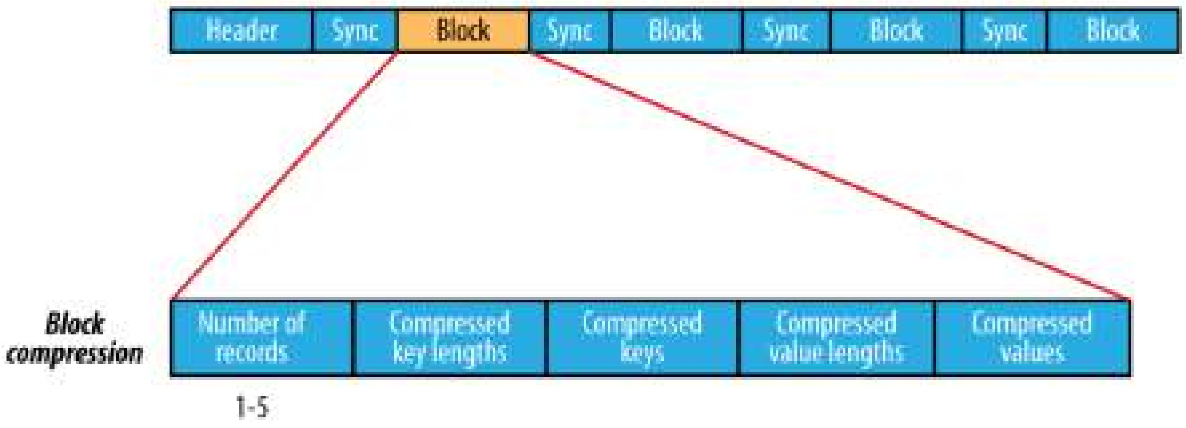


Figure 3. The internal structure of a sequence file with block compression

Map File

A MapFile is a sorted SequenceFile with an index to permit lookups by key. The index is itself a SequenceFile that contains a fraction of the keys in the map (every 128th key, by default). The idea is that the index can be loaded into memory to provide fast lookups from the main data file, which is another SequenceFile containing all the map entries in sorted key order.

MapFile offers a very similar interface to SequenceFile for reading and writing, the main thing to be aware of is that when writing using MapFile.Writer, map entries must be added in order, otherwise an IOException will be thrown.

MapFile variants

Hadoop comes with a few variants on the general key-value MapFile interface:

• SetFile is a specialization of MapFile for storing a set of Writable keys. The keys must be added in sorted order.

• ArrayFile is a MapFile where the key is an integer representing the index of the element in the array and the value is a Writable value.

• BloomMapFile is a MapFile that offers a fast version of the get() method, especially for sparsely populated files. The implementation uses a dynamic Bloom filter for testing whether a given key is in the map. The test is very fast because it is in memory, and it has a nonzero probability of false positives. Only if the test passes (the key is present) is the regular get() method called.

Other File Formats and Column-Oriented Formats

While sequence files and map files are the oldest binary file formats in Hadoop, they are not the only ones, and in fact there are better alternatives that should be considered for new projects.

Avro datafiles are like sequence files, they are designed for large-scale data processing, they are compact and splitable, but they are portable across different programming languages. Objects stored in Avro datafiles are described by a schema, rather than in the Java code of the implementation of a Writable object (as is the case for sequence files), making them very Java-centric. Avro datafiles are widely supported across components in the Hadoop ecosystem, so they are a good default choice for a binary format.

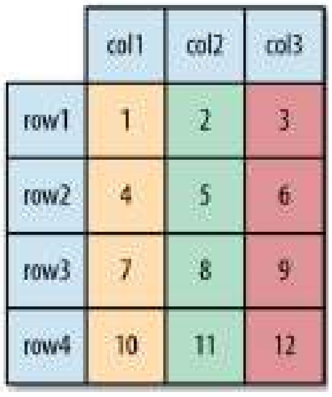


Fig. Logical Table

Sequence files, map files, and Avro datafiles are all row-oriented file formats, which means that the values for each row are stored contiguously in the file.

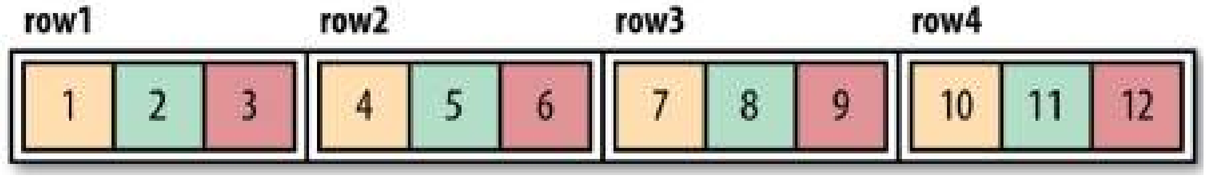
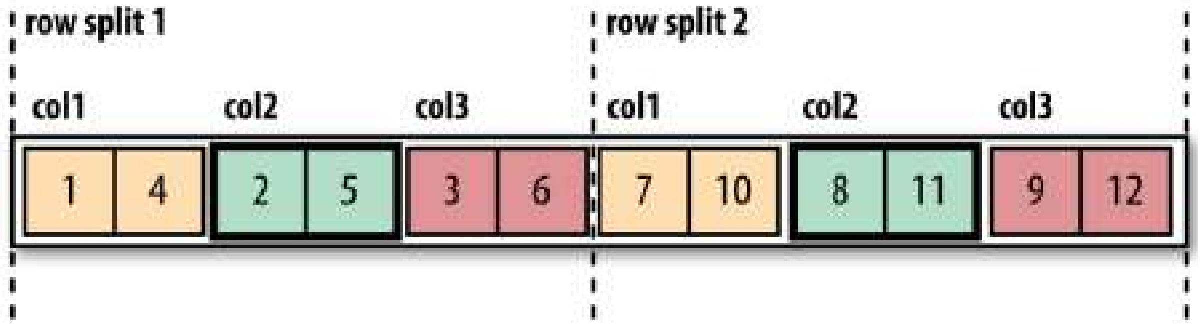


Fig. Row Oriented Layout (Sequence Files, Map File, and Avro Files)

In a column-oriented format, the rows in a file (or, equivalently, a table in Hive) are broken up into row splits, then each split is stored in column-oriented fashion: the values for each row in the first column are stored first, followed by the values for each row in the second column, and so on.



With row-oriented storage, like a sequence file, the whole row (stored in a sequence file record) is loaded into memory, even though only the second column is actually read. Lazy deserialization saves some processing cycles by deserializing only the column fields that are accessed, but it can't avoid the cost of reading each row's bytes from disk.

With column-oriented storage, only the column 2 parts of the file (green color highlighted in the figure) need to be read into memory.

In general, column-oriented formats work well when queries access only a small number of columns in the table. Conversely, row-oriented formats are appropriate when a large number of columns of a single row are needed for processing at the same time.

- Column-oriented formats need more memory for reading and writing, since they have to buffer a row split in memory, rather than just a single row.
- It's not usually possible to control when writes occur, so column-oriented formats are not suited to streaming writes.
- On the other hand, row-oriented formats like sequence files and Avro datafiles can be read up to the last sync point after a writer failure.
- It is for this reason that Flume uses row-oriented formats.
- The column-oriented file format in Hadoop was Hive's RCFile and Parquet.
- Parquet is a general-purpose column-oriented file format based on Google's Dremel, and has wide support across Hadoop components.
- Avro also has a column-oriented format called Trevni.

Map Reduce

- A Weather Dataset - Data Format
- Analyzing the Data with Unix Tools
- Analyzing the Data with Hadoop - Map and Reduce - Java Map Reduce
- Scaling Out - Data Flow - Combiner Functions
- Hadoop Streaming

A Weather Dataset

For our example, we will write a program that mines weather data. Weather sensors collect data every hour at many locations across the globe and gather a large volume of log data, which is a good candidate for analysis with MapReduce because we want to process all the data, and the data is semi-structured and record-oriented.

Data Format

- The data is extracted from the **National Climatic Data Center, (NCDC)**.
- The data is stored using a line-oriented ASCII format, in which each line is a record.
- The format supports a rich set of meteorological elements, many of which are optional or with variable data lengths.
- For simplicity, we focus on the basic elements, such as temperature, which are always present and are of fixed width.

Dataset Description

If we consider the details mentioned in the file, each file has entries that look like this:

0029**029070**99999**19050101**06004+64333+023450FM-
12+000599999V0202301N008219999999N0000001N9-
01391+99999102641ADDGF102991999999999999999999

When we consider the highlighted fields, the first one (029070) is the USAF (**U**nited **S**tates **A**ir **F**orce) weather station identifier. The next one (19050101) represents the observation date. The third highlighted one (0139) represents the air temperature in Celsius times ten. So, the reading of 0139 equates to 13.9 degrees Celsius. The next highlighted and italic item indicates a reading quality code.

Analysing the Data with Unix Tools

A program for finding the maximum recorded temperature by year from NCDC weather records

```
#!/usr/bin/env bash
for year in all/*
do
echo -ne `basename $year .gz`"\t"
gunzip -c $year | \
awk '{ temp = substr($0, 88, 5) + 0;
q = substr($0, 93, 1);
if (temp !=9999 && q ~ /[01459]/ && temp > max) max = temp }
END { print max }'
done
```

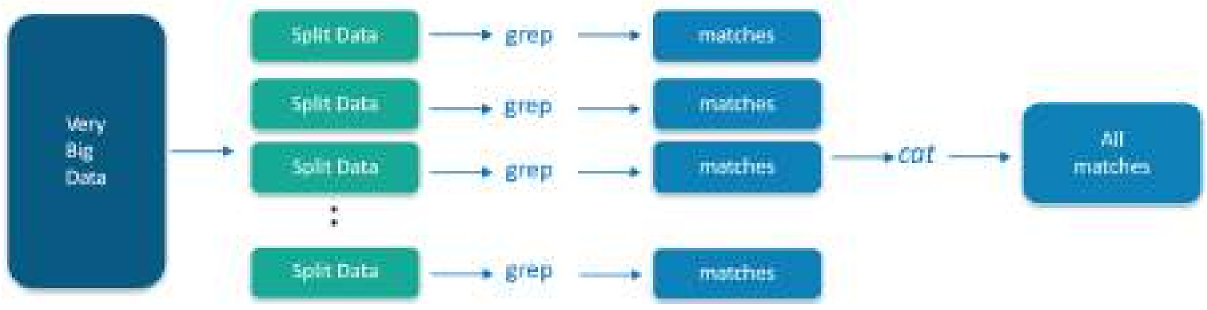
- The script loops through the compressed year files, first printing the year, and then processing each file using awk.
- The awk script extracts two fields from the data: the air temperature and the quality code.
- The air temperature value is turned into an integer by adding 0.
- Next, a test is applied to see whether the temperature is valid (the value 9999 signifies a missing value in the NCDC dataset) and whether the quality code indicates that the reading is not suspect or erroneous.
- If the reading is OK, the value is compared with the maximum value seen so far, which is updated if a new maximum is found.
- The END block is executed after all the lines in the file have been processed, and it prints the maximum value.

Here is the beginning of a run:

```
% ./max_temperature.sh
1901 317
1902 244
1903 289
1904 256
1905 283
```

The temperature values in the source file are scaled by a factor of 10, so this works out as a maximum temperature of 31.7°C for 1901

Traditional Way of Processing Data in a Distributed Environment



Let us understand, when the MapReduce framework was not there, how parallel and distributed processing used to happen in a traditional way.

So, let us take an example where I have a weather log containing the daily average temperature of the years from 1901 to 1905. Here, I want to calculate the day having the highest temperature each year.

So, in the traditional way, I will split the data into smaller parts or blocks and store them in different machines. Then, I will find the highest temperature in each part stored in the corresponding machine. At last, I will combine the results received from each of the machines to have the final output. Let us look at the challenges associated with this traditional approach:

Reliability problem: What if, any of the machines which are working with a part of data fails? The management of this failover becomes a challenge.

Equal split issue: How will I divide the data into smaller chunks so that each machine gets even part of data to work with. In other words, how to equally divide the data such that no individual machine is overloaded or underutilized.

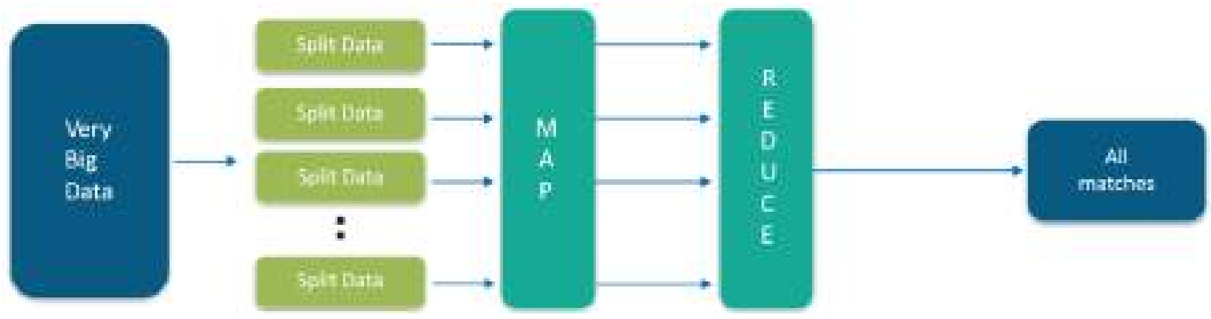
Single split may fail: If any of the machines fail to provide the output, I will not be able to calculate the result. So, there should be a mechanism to ensure this fault tolerance capability of the system.

Aggregation of the result: There should be a mechanism to aggregate the result generated by each of the machines to produce the final output.

These are the issues which I will have to take care individually while performing parallel processing of huge datasets when using traditional approaches.

To overcome these issues, we have the MapReduce framework which allows us to perform such parallel computations without bothering about the issues like reliability, fault tolerance etc.

What is MapReduce?



MapReduce is a programming framework that allows us to perform distributed and parallel processing on large data sets in a distributed environment.

- MapReduce consists of two distinct tasks — Map and Reduce.
- As the name MapReduce suggests, reducer phase takes place after the mapper phase has been completed.
- So, the first is the map job, where a block of data is read and processed to produce key-value pairs as intermediate outputs.
- The output of a Mapper or map job (key-value pairs) is input to the Reducer.
- The reducer receives the key-value pair from multiple map jobs.
- Then, the reducer aggregates those intermediate data tuples (intermediate key-value pair) into a smaller set of tuples or key-value pairs which is the final output.

Analyzing the Data with Hadoop

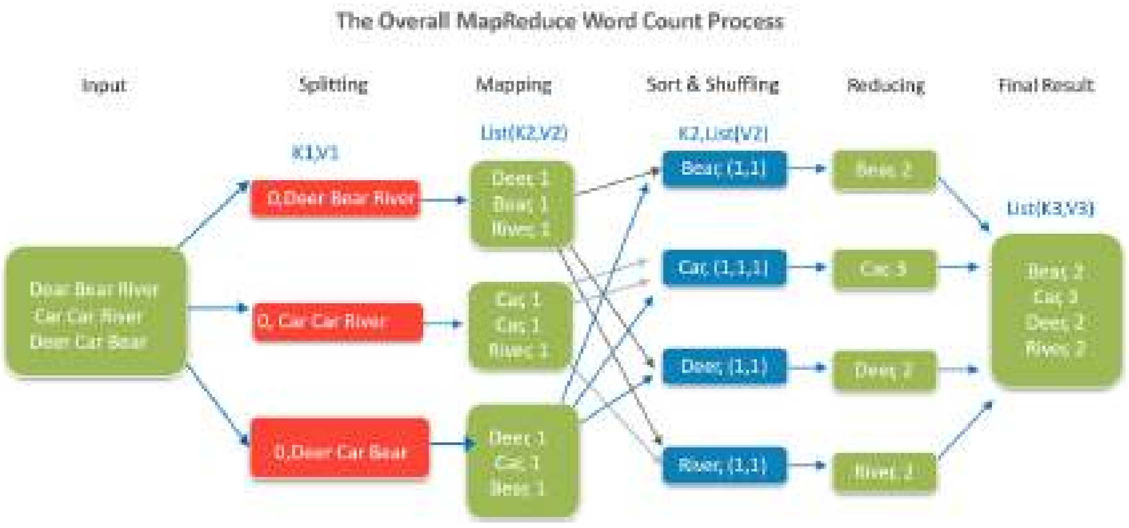
Map and Reduce

- MapReduce works by breaking the processing into two phases:
  - Map phase and
  - Reduce phase
- Each phase has key-value pairs as input and output, the types of which may be chosen by the programmer.
- The programmer also specifies two functions: the map function and the reduce function.

A Word Count Example of MapReduce

Let us understand, how a MapReduce works by taking an example where I have a text file called sample.txt whose contents are : **Dear, Bear, River, Car, Car, River, Deer, Car and Bear**

Now, suppose, we have to perform a word count on the sample.txt using MapReduce. So, we will be finding unique words and the number of occurrences of those unique words.



- First, we divide the input into three splits as shown in the figure. This will distribute the work among all the map nodes.
- Then, we tokenize the words in each of the mappers and give a hardcoded value (1) to each of the tokens or words. The rationale behind giving a hardcoded value equal to 1 is that every word, in itself, will occur once.
- Now, a list of key-value pair will be created where the key is nothing but the individual words and value is one. So, for the first line (Dear Bear River) we have 3 key-value pairs — Dear, 1; Bear, 1; River, 1. The mapping process remains the same on all the nodes.
- After the mapper phase, a partition process takes place where sorting and shuffling happen so that all the tuples with the same key are sent to the corresponding reducer.
- So, after the sorting and shuffling phase, each reducer will have a unique key and a list of values corresponding to that very key. For example, Bear, [1,1]; Car, [1,1,1].., etc.
- Now, each Reducer counts the values which are present in that list of values. As shown in the figure, reducer gets a list of values which is [1,1] for the key Bear. Then, it counts the number of ones in the very list and gives the final output as — Bear, 2.
- Finally, all the output key/value pairs are then collected and written in the output file.

Java MapReduce

Having run through how the MapReduce program works, the next step is to express it in code. The same word count example where, find out the number of occurrences of each word.

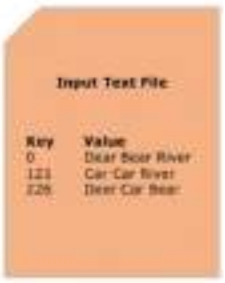
The entire MapReduce program can be fundamentally divided into three parts:

- Mapper Phase Code
- Reducer Phase Code
- Driver Code

Mapper code:

```
public static class Map extends Mapper<LongWritable,Text,Text,IntWritable> {
public void map(LongWritable key, Text value, Context context) throws IOException,InterruptedException {
                String line = value.toString();
                StringTokenizer tokenizer = new StringTokenizer(line);
                while (tokenizer.hasMoreTokens()) {
                value.set(tokenizer.nextToken());
                context.write(value, new IntWritable(1));
}
```

- We have created a class Map that extends the class Mapper which is already defined in the MapReduce Framework.
- We define the data types of input and output key/value pair after the class declaration using angle brackets.
- Both the input and output of the Mapper is a key/value pair.



Input Text File

Key	Value
0	Dear Bear River
121	Car Car River
226	Deer Car Bear

Input:

1. The key is nothing but the offset of each line in the text file: LongWritable
2. The value is each individual line (as shown in the figure at the right): Text

Output:

1. The key is the tokenized words: Text

2. We have the hardcoded value in our case which is 1: IntWritable
3. Example — Dear 1, Bear 1, etc.

- We have written a java code where we have tokenized each word and assigned them a hardcoded value equal to 1.

Reducer Code:

```
public static class Reduce extends Reducer<Text,IntWritable,Text,IntWritable> {
public void reduce(Text key, Iterable<IntWritable> values,Context context)
throws IOException,InterruptedException {
            int sum=0;
           for(IntWritable x: values)
           {
           sum+=x.get();
           }
           context.write(key, new IntWritable(sum));
           }
}
```

- We have created a class Reduce which extends class Reducer like that of Mapper.
- We define the data types of input and output key/value pair after the class declaration using angle brackets as done for Mapper.
- Both the input and the output of the Reducer is a key-value pair.

Input:

1. The key nothing but those unique words which have been generated after the sorting and shuffling phase: Text
2. The value is a list of integers corresponding to each key: IntWritable
3. Example — Bear, [1, 1], etc.

Output:

1. The key is all the unique words present in the input text file: Text
2. The value is the number of occurrences of each of the unique words: IntWritable
3. Example — Bear, 2; Car, 3, etc.

- We have aggregated the values present in each of the list corresponding to each key and produced the final answer.
- In general, a single reducer is created for each of the unique words, but, we can specify the number of reducer in mapred-site.xml.

Driver Code:

```
Configuration conf= new Configuration();
Job job = new Job(conf,"My Word Count Program");
job.setJarByClass(WordCount.class);
job.setMapperClass(Map.class);
job.setReducerClass(Reduce.class);
job.setOutputKeyClass(Text.class);

job.setOutputValueClass(IntWritable.class);
job.setInputFormatClass(TextInputFormat.class);
job.setOutputFormatClass(TextOutputFormat.class);
Path outputPath = new Path(args[1]);

//Configuring the input/output path from the filesystem into the job
FileInputFormat.addInputPath(job, new Path(args[0]));
FileOutputFormat.setOutputPath(job, new Path(args[1]));
```

- In the driver class, we set the configuration of our MapReduce job to run in Hadoop.
- We specify the name of the job, the data type of input/output of the mapper and reducer.
- We also specify the names of the mapper and reducer classes.
- The path of the input and output folder is also specified.
- The method setInputFormatClass () is used for specifying that how a Mapper will read the input data or what will be the unit of work. Here, we have chosen TextInputFormat so that single line is read by the mapper at a time from the input text file.
- The main () method is the entry point for the driver. In this method, we instantiate a new Configuration object for the job.

Run the MapReduce code:

The command for running a MapReduce code is:

```
hadoop jar hadoop-mapreduce-example.jar WordCount /sample/input /sample/output
```

Scaling Out

To scale out, we need to store the data in a distributed filesystem (typically HDFS). This allows Hadoop to move the MapReduce computation to each machine hosting a part of the data, using Hadoop’s resource management system, called YARN.

Data Flow

A MapReduce job is a unit of work that the client wants to be performed: it consists of

- The input data
- The MapReduce programs
- The configuration information

Hadoop runs the job by dividing it into tasks, of which there are two types:

- Map tasks
- Reduce tasks

The tasks are scheduled using YARN and run on nodes in the cluster. If a task fails, it will be automatically rescheduled to run on a different node.

Hadoop divides the input to a MapReduce job into fixed-size pieces called input splits.

Hadoop creates one map task for each split, which runs the user-defined map function for each record in the split.

Having many splits means the time taken to process each split is small compared to the time to process the whole input.

On the other hand, if splits are too small, the overhead of managing the splits and map task creation begins to dominate the total job execution time.

For most jobs, a good split size tends to be the size of an HDFS block, which is 128 MB by default.

(a) Hadoop does its best to run the map task on a node where the input data resides in HDFS, because it doesn't use valuable cluster bandwidth. This is called the data locality optimization.

(b) Sometimes, however, all the nodes hosting the HDFS block replicas for a map task's input split are running other map tasks, so the job scheduler will look for a free map slot on a node in the same rack as one of the blocks.

(c) Very occasionally even this is not possible, so an off-rack node is used, which results in an inter-rack network transfer. The three possibilities are illustrated in Figure 2-2.

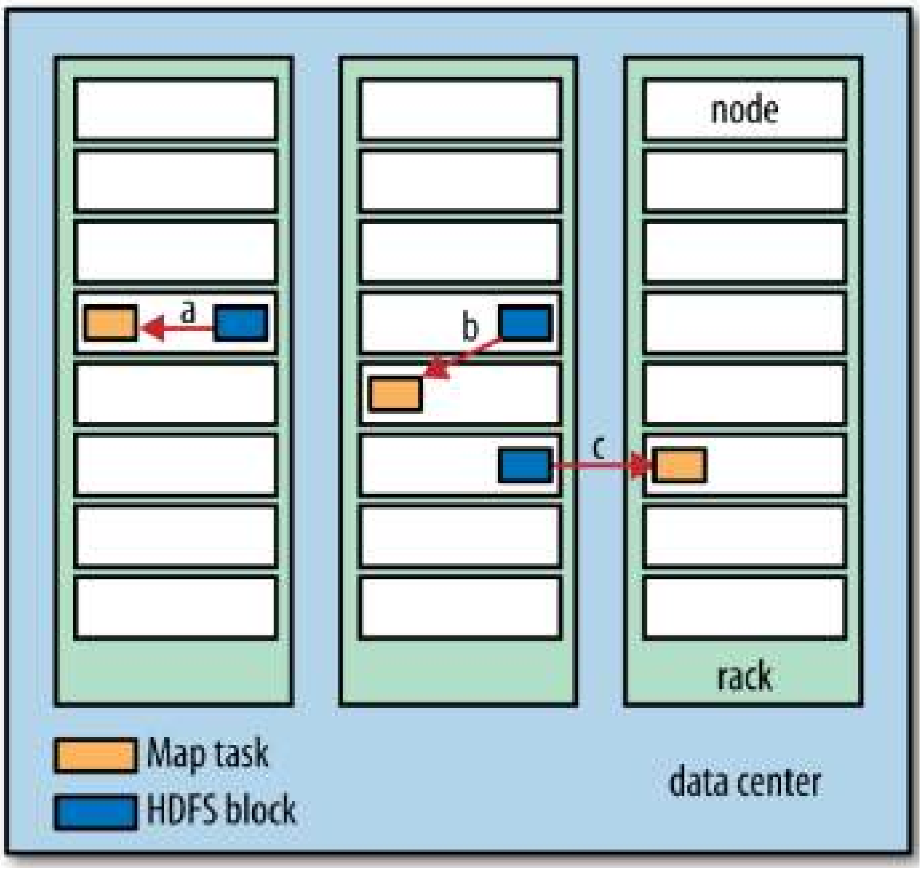


Figure 2-2. Data-local (a), rack-local (b), and off-rack (c) map tasks

Why Map tasks write their output to the local disk, not to HDFS?

Map output is intermediate output: it's processed by reduce tasks to produce the final output, and once the job is complete, the map output can be thrown away. So, storing it in HDFS with replication would be overkill. If the node running the map task fails before the map output has been consumed by the reduce task, then Hadoop will automatically rerun the map task on another node to re-create the map output.

Reduce tasks don't have the advantage of data locality; the input to a single reduce task is normally the output from all mappers. In the present example, we have a single reduce task that is fed by all of the map tasks. Therefore, the sorted map outputs have to be transferred across the network to the node where the reduce task is running, where they are merged and then passed to the user-defined reduce function. The output of the reduce is normally stored in HDFS for reliability. For each HDFS block of the reduce output, the first replica is stored on the local node, with other replicas being stored on off-rack nodes for reliability.

The whole data flow with a single reduce task is illustrated in Figure 2-3. The dotted boxes indicate nodes, the dotted arrows show data transfers on a node, and the solid arrows show data transfers between nodes.

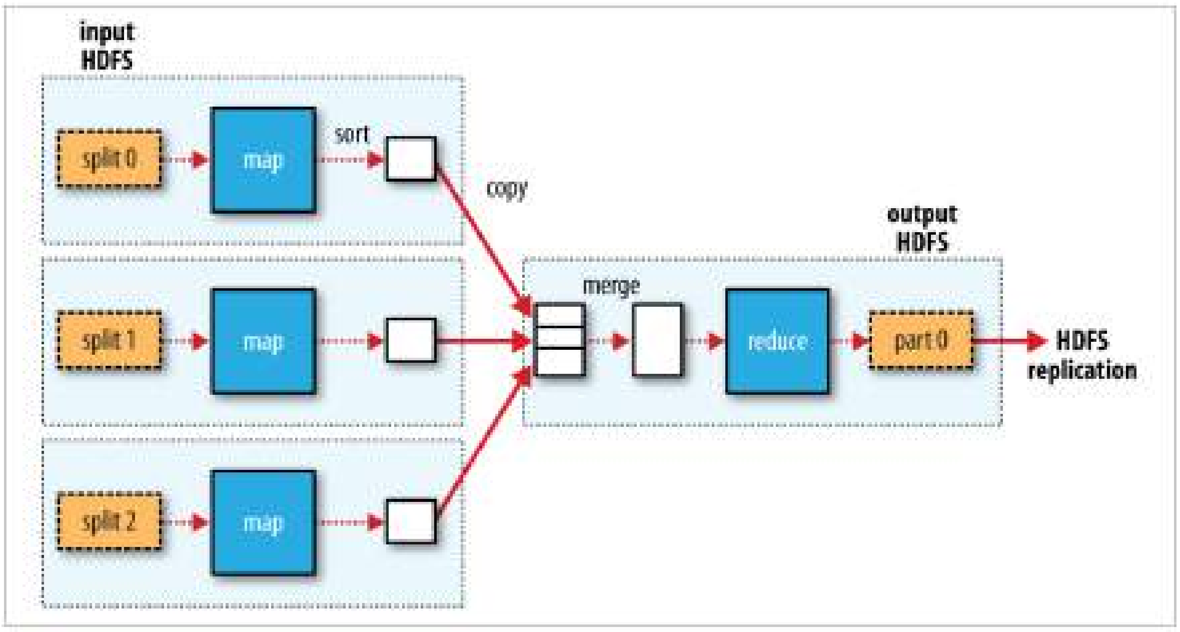


Figure 2-3. MapReduce data flow with a single reduce task

The number of reduce tasks is not governed by the size of the input, but instead is specified independently.

When there are multiple reducers, the map tasks partition their output, each creating one partition for each reduce task. There can be many keys in each partition, but the records for any given key are all in a single partition. The partitioning can be controlled by a user-defined partitioning function, but normally the default partitioner—which buckets keys using a hash function—works very well.

The data flow for the general case of multiple reduce tasks is illustrated in Figure 2-4. This diagram makes it clear why the data flow between map and reduce tasks is colloquially known as "the shuffle," as each reduce task is fed by many map tasks.

Finally, it's also possible to have zero reduce tasks. This can be appropriate when you don't need the shuffle because the processing can be carried out entirely in parallel.

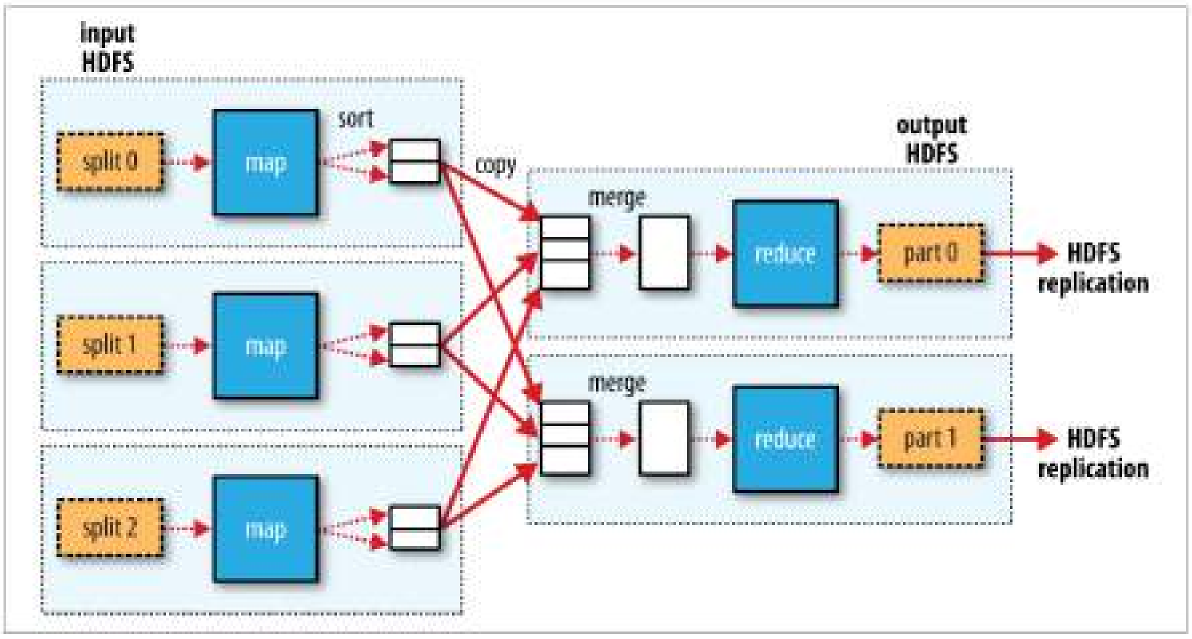


Figure 2-4. MapReduce data flow with multiple reduce tasks

Combiner Functions

Many MapReduce jobs are limited by the bandwidth available on the cluster, so it pays to minimize the data transferred between map and reduce tasks. Hadoop allows the user to specify a combiner function to be run on the map output, and the combiner function's output forms the input to the reduce function. Because the combiner function is an optimization, Hadoop does not provide a guarantee of how many times it will call it for a particular map output record, if at all. In other words, calling the combiner function zero, one, or many times should produce the same output from the reducer.

The contract for the combiner function constrains the type of function that may be used. This is best illustrated with an example. Suppose that for the maximum temperature example, readings for the year 1950 were processed by two maps. Imagine the first map produced the output:

(1950, 0)

(1950, 20)

(1950, 10)

and the second produced:

(1950, 25)

(1950, 15)

The reduce function would be called with a list of all the values:

(1950, [0, 20, 10, 25, 15])

with output:

(1950, 25)

since 25 is the maximum value in the list. We could use a combiner function that, just like the reduce function, finds the maximum temperature for each map output. The reduce function would then be called with:

(1950, [20, 25])

and would produce the same output as before. More succinctly, we may express the function calls on the temperature values in this case as follows:

max(0, 20, 10, 25, 15) = max(max(0, 20, 10), max(25, 15)) = max(20, 25) = 25

The combiner function doesn't replace the reduce function. But, it can help cut down the amount of data shuffled between the mappers and the reducers, and for this reason alone it is always worth considering whether you can use a combiner function in your MapReduce job.

Hadoop Streaming

What is Hadoop Streaming?

Hadoop Streaming is a utility that comes with the Hadoop distribution. It can be used to execute programs for big data analysis. Hadoop streaming can be performed using languages like Python, Java, PHP, Scala, Perl, UNIX, and many more. The utility allows us to create and run Map/Reduce jobs with any executable or script as the mapper and/or the reducer. For example:

Hadoop provides an API to MapReduce that allows you to write your map and reduce functions in languages other than Java. Hadoop Streaming uses Unix standard streams as the interface between Hadoop and your program, so you can use any language that can read standard input and write to standard output to write your MapReduce program.

Streaming is naturally suited for text processing. Map input data is passed over standard input to your map function, which processes it line by line and writes lines to standard output. A map output key-value pair is written as a single tab-delimited line. Input to the reduce function is in the same format—a tab-separated key-value pair—passed over standard input. The reduce function reads lines from standard input, which the framework guarantees are sorted by key, and writes its results to standard output.

```
$HADOOP_HOME/bin/Hadoop jar $HADOOP_HOME/hadoop-streaming.jar

-input myInputDirs

-output myOutputDir

-mapper /bin/cat

-reducer /bin/wc
```

Let’s illustrate this by rewriting our MapReduce program for finding frequencies of words in a text file in Streaming.

Python MapReduce Code for WordCount Example:

mapper.py

```
#!/usr/bin/python

import sys

#Word Count Example

# input comes from standard input STDIN

for line in sys.stdin:

line = line.strip() #remove leading and trailing whitespaces

words = line.split() #split the line into words and returns as a list

for word in words:

#write the results to standard output STDOUT

print'%s    %s' % (word,1) #Emit the word
```

reducer.py

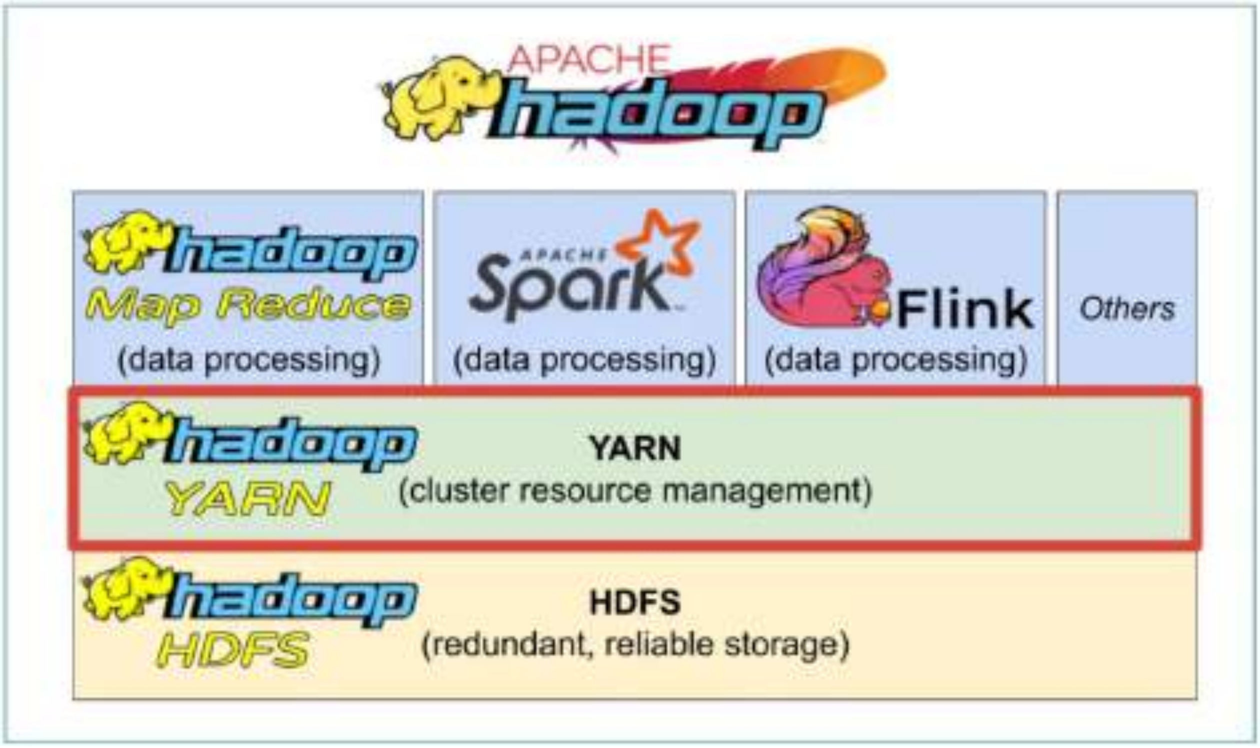
```
#!/usr/bin/python

import sys

from operator import itemgetter # using a dictionary to map words to their counts

current_word = None

current_count = 0

word = None

# input comes from STDIN

for line in sys.stdin:

line = line.strip()

word,count = line.split('	',1)

try:

count = int(count)

except ValueError:

continue

if current_word == word:

current_count += count

else:

if current_word:

print '%s	%s' % (current_word, current_count)

current_count = count

current_word = word

if current_word == word:

print '%s	%s' % (current_word,current_count)
```

YARN

- Anatomy of a YARN Application Run
  - Resource Requests
  - Application Lifespan
  - Building YARN Applications
- YARN Compared to Map Reduce 1
- Scheduling in YARN

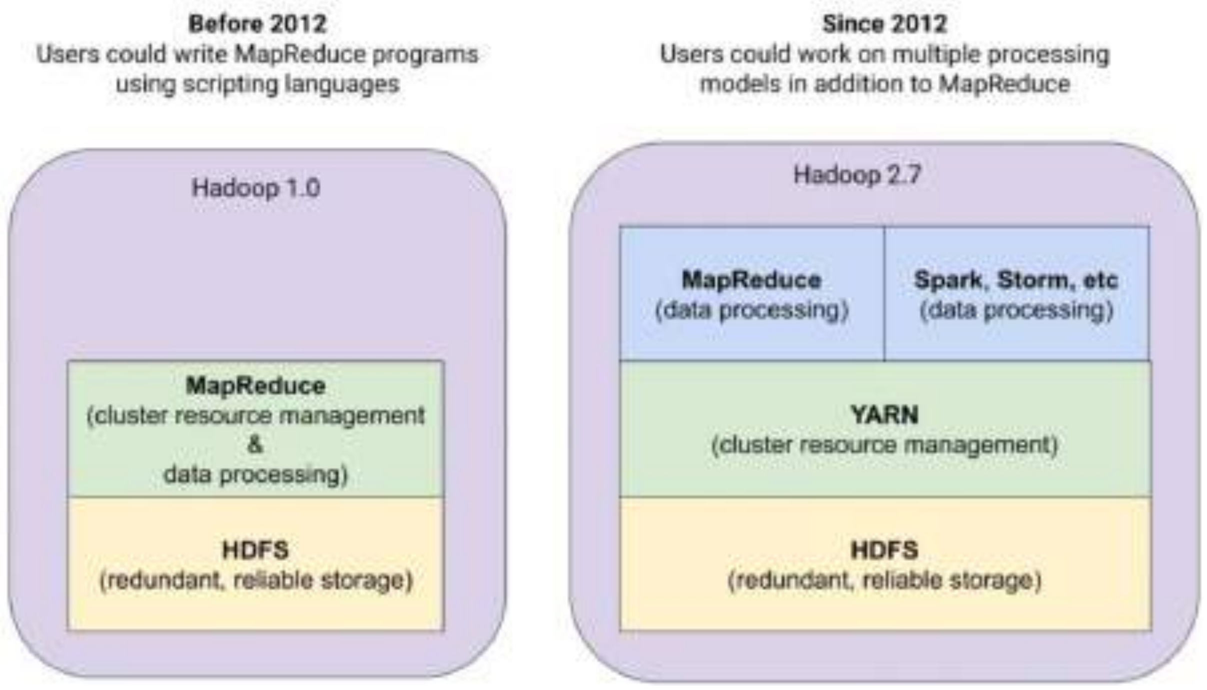
What is YARN?

YARN (**Y**et **A**nother **R**esource **N**egotiator) is a critical component of the Hadoop ecosystem. It functions as the cluster resource management layer and job scheduling technology in the Hadoop ecosystem. It is responsible for managing and allocating resources such as CPU, memory, and storage for distributed applications running on a Hadoop cluster.



Why do we need YARN?

The primary goal of YARN is to decouple the functionalities of resource management and data processing into separate components. This allows a global resource manager that can support diverse data processing applications, such as MapReduce, Spark, Storm, Tez, and others, to run on a single cluster.


Before 2012
Users could write MapReduce programs using scripting languages
Since 2012
Users could work on multiple processing models in addition to MapReduce
Hadoop 1.0
MapReduce (cluster resource management & data processing)
HDFS (redundant, reliable storage)
Hadoop 2.7
MapReduce (data processing)
Spark, Storm, etc (data processing)
YARN (cluster resource management)
HDFS (redundant, reliable storage)

YARN was introduced in Hadoop 2 to improve the MapReduce implementation, but it is general enough to support other distributed computing paradigms as well. YARN was introduced in Hadoop version 2.0 to address some limitations of the earlier MapReduce framework's resource management.

Before YARN was introduced, Apache Hadoop used a resource manager called MapReduce, which served as both a resource manager and a processing engine. This system was tightly coupled with the Hadoop Distributed File System (HDFS) and limited to running MapReduce jobs only. This made it difficult for users to run other types of applications on the Hadoop cluster.

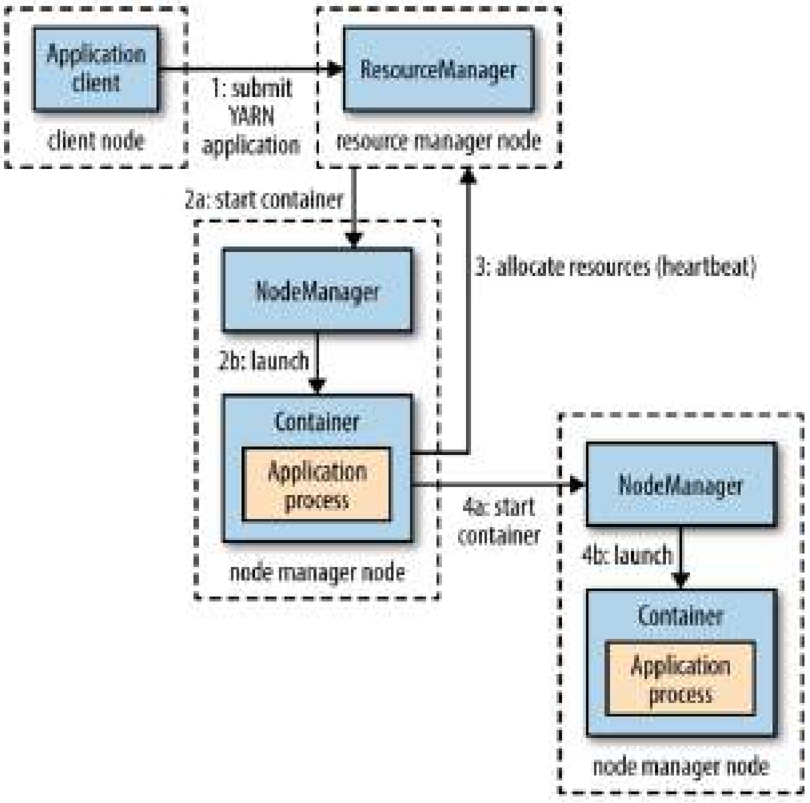
To overcome these limitations, YARN was developed. YARN separates the processing engine and management functions of MapReduce, allowing it to support multiple processing engines and applications.

Anatomy of a YARN Application Run

YARN provides its core services via two types of long-running daemons:

1) **Resource Manager** (one per cluster) to manage the use of resources across the cluster.
2) **Node Managers** running on all the nodes in the cluster to launch and monitor containers.

A container executes an application-specific process with a constrained set of resources (memory, CPU, and so on). Below figure illustrates how YARN runs an application.



Step 1: To run an application on YARN, a client contacts the resource manager and asks it to run an application master process.

Step 2 (a & b): The resource manager then finds a node manager that can launch the application master in a container. Precisely what the application master does once it is running depends on the application.

Step 3: It could simply run a computation in the container it is running in and return the result to the client. Or it could request more containers from the resource managers.

Step 4 (a & b): By using these resources resource manager can also run a distributed computation.

Resource Requests

YARN has a flexible model for making resource requests. A request for a set of containers can express:

1) The amount of computer resources required for each container (memory and CPU).
2) As well as locality constraints for the containers in that request.

Locality is critical in ensuring that distributed data processing algorithms use the cluster bandwidth efficiently, so YARN allows an application to specify locality constraints for the containers it is requesting. Locality constraints can be used to request a container on a specific node or rack, or anywhere on the cluster (off-rack).

Sometimes the locality constraint cannot be met, in which case either no allocation is made or, optionally, the constraint can be loosened. For example, if a specific node was requested but it is not possible to start a container on it (because other containers are running on it), then YARN will try to start a container on a node in the same rack, or, if that's not possible, on any node in the cluster.

In the common case of launching a container to process an HDFS block, the application will request a container on one of the nodes hosting the block's three replicas, or on a node in one of the racks hosting the replicas, or, failing that, on any node in the cluster.

A YARN application can make resource requests at any time while it is running. For example, an application can make all of its requests up front, or it can take a more dynamic approach whereby it requests more resources dynamically to meet the changing needs of the application.

Application Lifespan

The lifespan of a YARN application can vary dramatically: from a short-lived application of a few seconds to a long-running application that runs for days or even months. Rather than look at how long the application runs for, it's useful to categorize applications in terms of how they map to the jobs that users run.

- The simplest case is one application per user job, which is the approach that MapReduce takes.
- The second model is to run one application per workflow or user session jobs. This approach can be more efficient than the first, since containers can be reused between

jobs, and there is also the potential to cache intermediate data between jobs. Spark is an example that uses this model.

- The third model is a long-running application that is shared by different users.
  For example, Apache Slider has a long-running application master for launching other applications on the cluster.
  This approach is also used by Impala to provide a proxy application that the Impala daemons communicate with to request cluster resources.

Building YARN Applications

Writing a YARN application from scratch is fairly involved, but in many cases is not necessary, as it is often possible to use an existing application that fits the bill.

For example, if you are interested in running a directed acyclic graph (DAG) of jobs, then Spark or Tez is appropriate; or for stream processing, Spark, Samza, or Storm works.

There are a couple of projects that simplify the process of building a YARN application.

Apache Slider, makes it possible to run existing distributed applications on YARN. Users can run their own instances of an application (such as HBase) on a cluster, independently of other users, which means that different users can run different versions of the same application. Slider provides controls to change the number of nodes an application is running on, and to suspend then resume a running application.

Apache Twill is similar to Slider, but in addition provides a simple programming model for developing distributed applications on YARN. Twill allows you to define cluster processes as an extension of a Java Runnable, then runs them in YARN containers on the cluster. Twill also provides support for, among other things, real-time logging and command messages.

In cases where none of these options are sufficient such as an application that has complex scheduling requirements then the distributed shell application that is a part of the YARN project itself serves as an example of how to write a YARN application. It demonstrates how to use YARN's client APIs to handle communication between the client or application master and the YARN daemons.

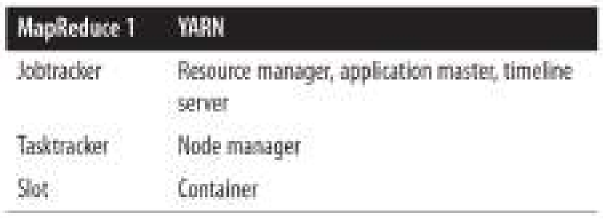
YARN Compared to MapReduce 1

- In MapReduce 1, there are two types of daemons that control the job execution process: a jobtracker and one or more tasktrackers. The jobtracker coordinates all the jobs run on the system by scheduling tasks to run on tasktrackers. Tasktrackers run tasks and send progress reports to the jobtracker, which keeps a record of the overall progress of each job. If a task fails, the jobtracker can reschedule it on a different tasktracker. In MapReduce 1, the jobtracker takes care of both job scheduling and task progress monitoring (keeping track of tasks, restarting failed or slow tasks, and doing task bookkeeping, such as maintaining counter totals).

  By contrast, in YARN these responsibilities are handled by separate entities: the resource manager and an application master (one for each MapReduce job).
- The jobtracker is also responsible for storing job history for completed jobs, although it is possible to run a job history server as a separate daemon to take the load off the jobtracker.

  In YARN, the equivalent role is the timeline server, which stores application history.

Table 4-1. A comparison of MapReduce 1 and YARN components



MapReduce 1	YARN
Jobtracker	Resource manager, application master, timeline server
Tasktracker	Node manager
Slot	Container

YARN was designed to address many of the limitations in MapReduce 1. The benefits to using YARN include the following:

Scalability

YARN can run on larger clusters than MapReduce 1. MapReduce 1 hits scalability bottlenecks in the region of 4,000 nodes and 40,000 tasks, stemming from the fact that the jobtracker has to manage both jobs and tasks.

YARN overcomes these limitations by virtue of its split resource manager/application master architecture: it is designed to scale up to 10,000 nodes and 100,000 tasks.

Availability

High availability (HA) is usually achieved by replicating the state needed for another daemon to take over the work needed to provide the service, in the event of the service daemon failing.

However, the large amount of rapidly changing complex state in the jobtracker's memory makes it very difficult to retrofit HA into the jobtracker service.

With the jobtracker's responsibilities split between the resource manager and application master in YARN, making the service highly available. Hadoop 2 supports HA both for the resource manager and for the application master for MapReduce jobs.

Utilization

In MapReduce 1, each tasktracker is configured with a static allocation of fixed-size "slots," which are divided into map slots and reduce slots at configuration time. A map slot can only be used to run a map task, and a reduce slot can only be used for a reduce task.

In YARN, a node manager manages a pool of resources, rather than a fixed number of designated slots. MapReduce running on YARN will not hit the situation where a reduce task has to wait because only map slots are available on the cluster, that happens in MapReduce 1.

Multitenancy

In some ways, the biggest benefit of YARN is that it opens up Hadoop to other types of distributed application beyond MapReduce. MapReduce is just one YARN application among many. It is even possible for users to run different versions of MapReduce on the same YARN cluster, which makes the process of upgrading MapReduce more manageable.

YARN Schedulers

The YARN scheduler is a critical component of Hadoop that facilitates the scheduling and allocation of available resources within a cluster among the submitted applications. The scheduler offers different policies for users to choose from to meet unique requirements, such as minimum capacity, resource fairness, and job priority.

YARN supports 3 different scheduler policies:

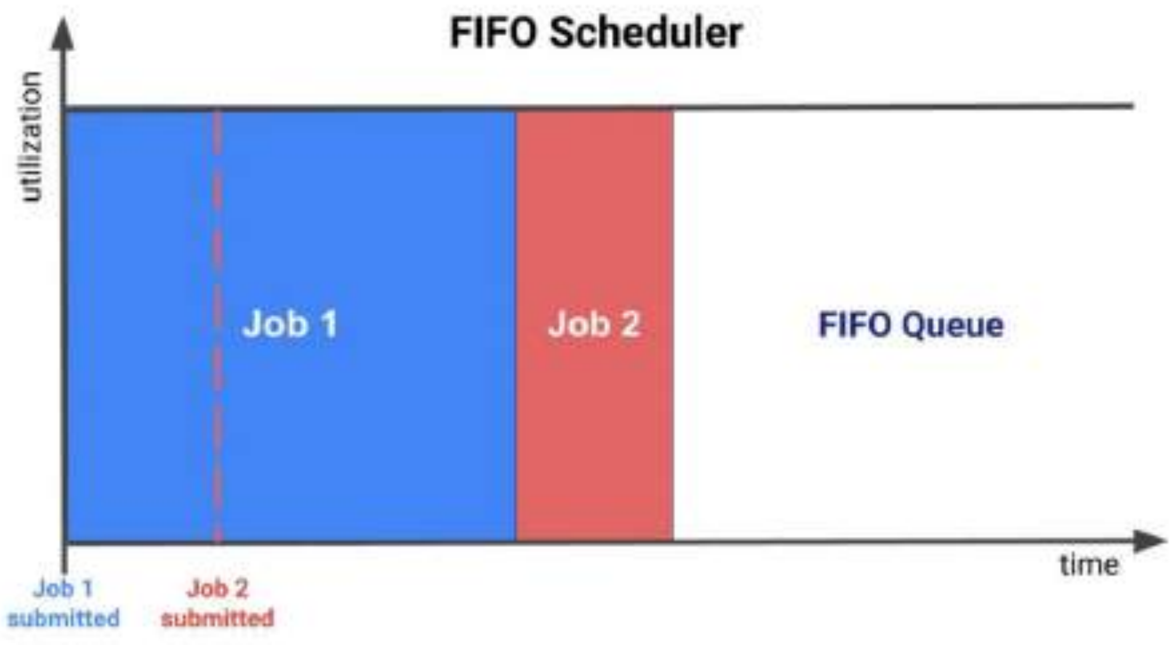
- FIFO (First-In, First-Out) Scheduler

- Capacity Scheduler
- Fair Scheduler

Each scheduler has unique features, advantages, and trade-offs. Administrators can select the most suitable scheduler based on their specific requirements and use cases.

FIFO Scheduler

The FIFO Scheduler is the simplest of all the schedulers. As the name suggests, it schedules applications based on the order in which they are submitted to the queue, following on first-come-first-serve principle. Once an application starts running, it consumes all the resources until it completes, and then the next application in the queue gains access to the resources.



Pros:

- Simple and easy-to-understand scheduling mechanism.
- Minimal configuration is required.

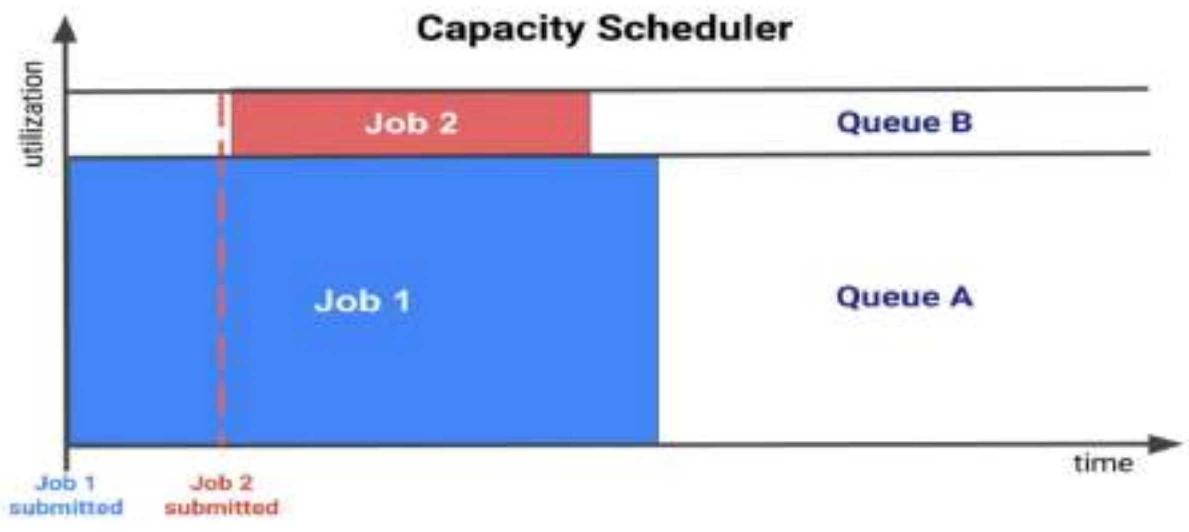
Cons:

Inefficient resource allocation: The FIFO Scheduler doesn’t take into account the resource requirements of the applications, which can lead to suboptimal resource utilization.

Lack of advanced features: Unlike other YARN schedulers (such as the Capacity Scheduler and Fair Scheduler), the FIFO Scheduler doesn't support advanced features like resource-based scheduling, queues, or priority-based allocation.

Capacity Scheduler

The Capacity Scheduler is designed to support multi-tenant environments where multiple users share the same cluster. It divides the cluster resources into multiple queues, each with its own reserved resources (CPUs, RAM) and guarantees (minimum capacity), while allowing for elasticity (maximum capacity) to dynamically utilize unused resources from other queues.



Pros:

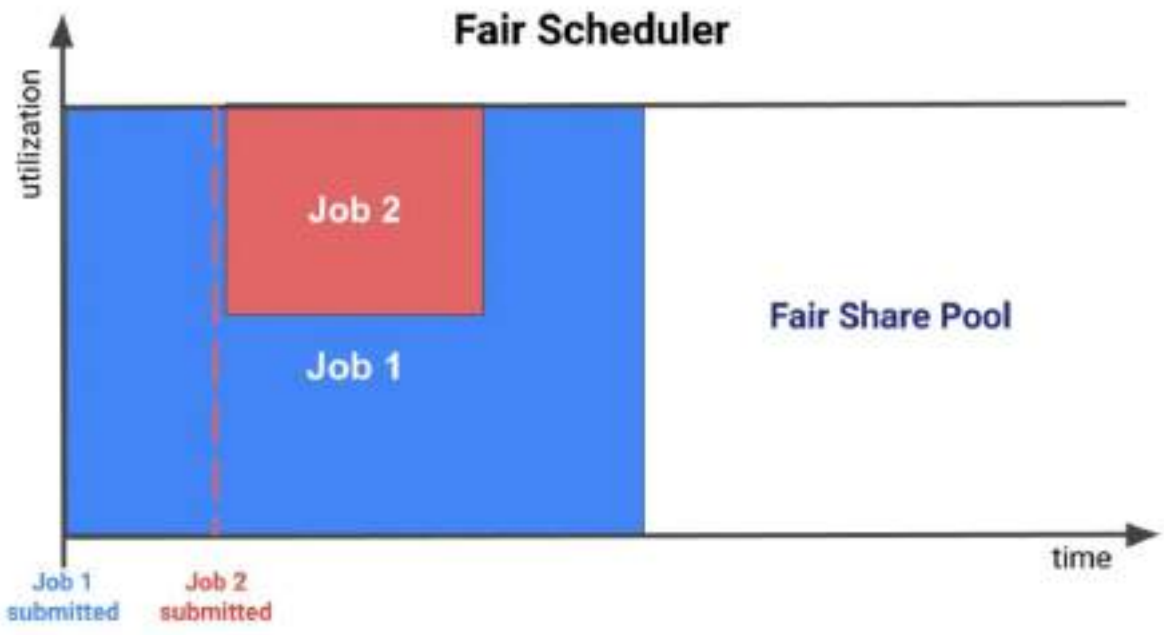
- Designed for multi-tenant clusters, allowing multiple organizations to share the same cluster.
- Allows administrators to define hierarchical queues and assign capacities to them, ensuring minimum guaranteed resource allocation.
- Supports features like access control, resource limits, and elastic resource allocation.

Cons:

- Might require more complex configuration, management, and tuning compared to the FIFO scheduler.
- Resource allocation is based on the pre-defined capacities of the queues, which might not be as dynamic as the Fair Scheduler.

Fair Scheduler

The Fair Scheduler is designed to provide fair resource allocation across all applications running in a Hadoop cluster, ensuring that no single application dominates the cluster resources, without requiring a set amount of reserved capacity. When a job starts — if it is the only job running — it gets all the resources of the cluster. When a second job starts, it gets resources as soon as some containers are freed (not utilized in some stages) from the first running job. After the smaller job finishes, the scheduler reassigns resources to the larger one. The Fair Scheduler enables short apps to finish in a reasonable time without stopping long-lived apps.



Pros:

- Provides fair resource allocation across all applications in the cluster by dynamically adjusting resources based on demand.
- Prevents any single application from dominating the cluster resources.

Cons:

- More complex configuration and management compared to the FIFO scheduler.
- In some cases, might lead to sub-optimal resource allocation if the fairness policies are not properly configured.