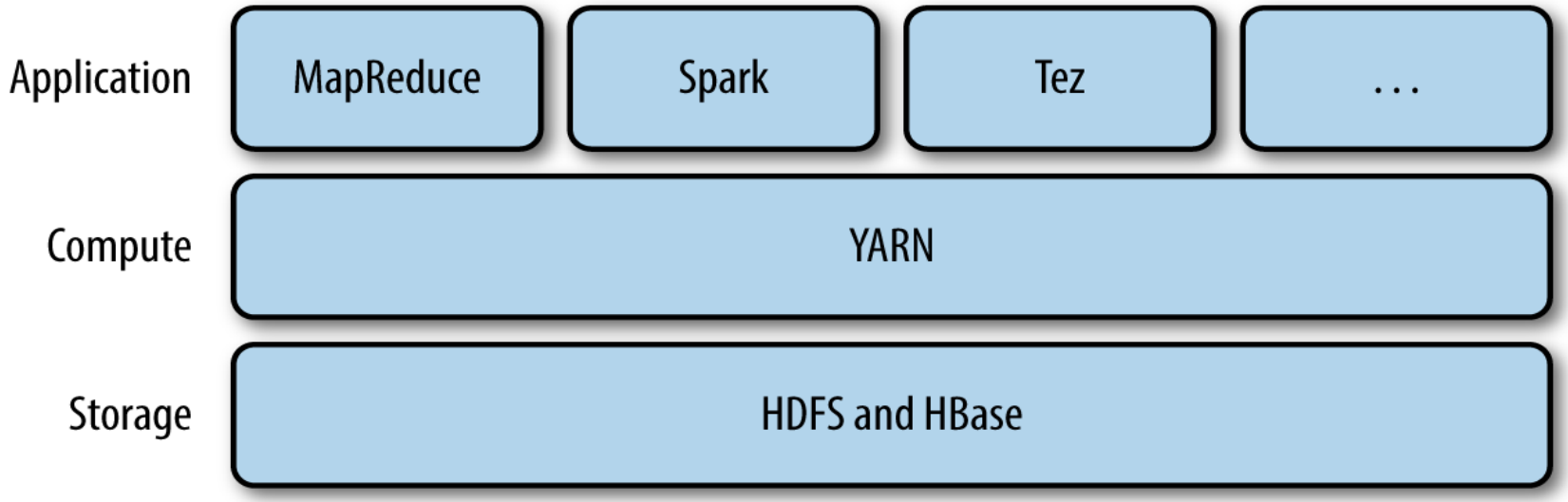
Unit-III

YARN and Hadoop I/O

Yet Another Resource Negotiator (YARN)

- Apache YARN (Yet Another Resource Negotiator) is Hadoop's cluster resource management system.
- 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 provides APIs for requesting and working with cluster resources – **Not used directly by user code**
- Users write higher-level APIs provided by distributed computing frameworks, which themselves are built on YARN
- Hides resource management details from the user

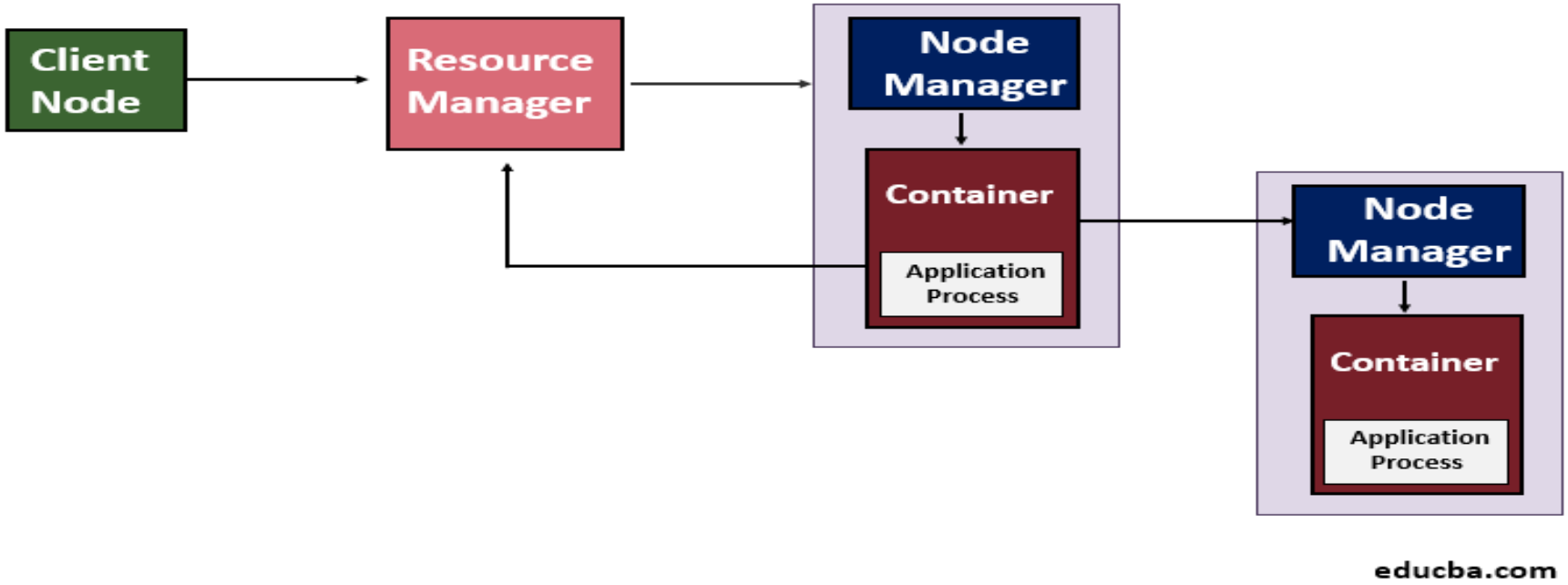


YARN Applications

Yet Another Resource Negotiator (YARN)

- Cluster management component of Hadoop 2.0.
- It includes
  - Resource Manager,
  - Node Manager,
  - Containers, and
  - Application Master.
- The Resource Manager is the major component that manages application management and job scheduling for the batch process.
- Node manager is the component that manages task distribution for each data node in the cluster.
- Containers are the hardware components such as CPU, RAM for the Node that is managed through YARN.
- Application Master is for monitoring and managing the application lifecycle in the Hadoop cluster.

Hadoop YARN Architecture



- Resource Manager sees the usage of the resources across the Hadoop cluster
- Life cycle of the applications that are running on a particular cluster is supervised by the Application Master.
- For cluster resources, Application Master negotiates with the Resource Manager

Components of YARN

Resource Manager

- YARN works through a Resource Manager which is one per node and Node Manager which runs on all the nodes.
- The Resource Manager manages the resources used across the cluster and the Node Manager lunches and monitors the containers.
- Scheduler and Application Manager are two components of the Resource Manager.

- **Scheduler:**
  - Scheduling is performed based on the requirement of resources by the applications.
  - YARN provides few schedulers to choose from and they are Fair and Capacity Scheduler.
  - Scheduler allocates resources to the running applications based on the capacity and queue.

- **Application Manager:**
  - It manages the running of Application Master in a cluster
  - On the failure of Application Master Container, it helps in restarting it.
  - It bears the responsibility of accepting the submission of the jobs.

Components of YARN

Node Manager

- Node Manager is responsible for execution of the task in each data node.
- Sends a heartbeat to the Resource Manager
- It is responsible for seeing to the nodes on the cluster individually and manages the workflow and user jobs on a specific node.
- It manages the application containers which are assigned by the Resource Manager.
- The Node Manager starts the containers by creating the container processes which are requested.
- It also kills the containers as asked by the Resource Manager.

Components of YARN

Containers

- The Containers are set of resources like RAM, CPU, and Memory etc. on a single node.
- They are scheduled by Resource Manager and monitored by Node Manager.
- Container Life Cycle manages the YARN containers by using container launch context and provides access to the application for the specific usage of resources in a particular host.

Components of YARN

Application Master

- It monitors the execution of tasks and also manages the lifecycle of applications running on the cluster.
- An individual Application Master gets associated with a job when it is submitted to the framework.
- Its chief responsibility is to negotiate the resources from the Resource Manager.
- It works with the Node Manager to monitor and execute the tasks.

YARN

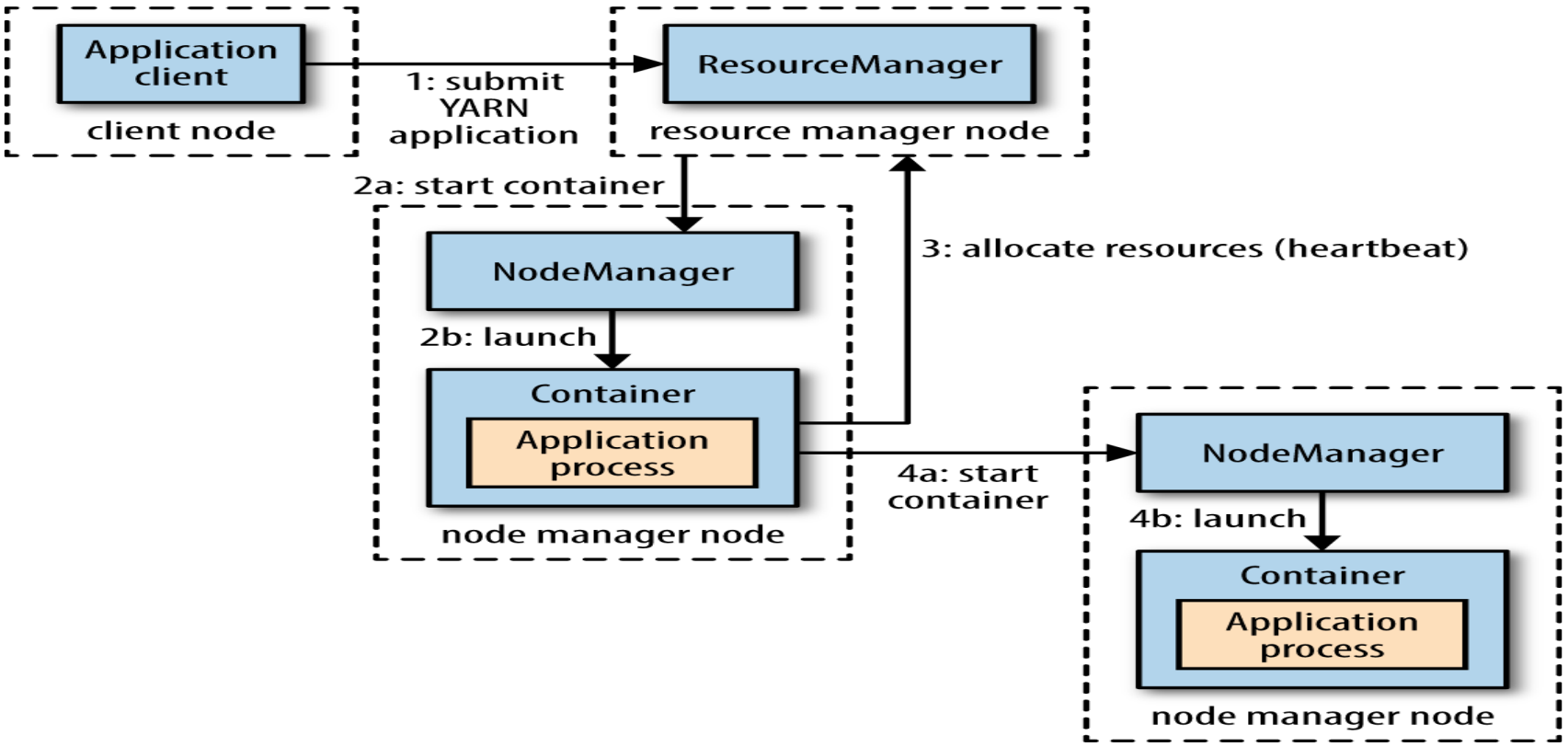
In order to run an application through YARN, below steps are performed.

- The client contacts the Resource Manager which requests to run the application process i.e. it submits the YARN application.
- The next step is that the Resource Manager searches for a Node Manager which will, in turn, launch the Application Master in a container.
- The Application Master can either run the execution in the container in which it is running currently and provide the result to the client or it can request more containers from resource manager which can be called distributed computing.
- The client then contacts the Resource Manager to monitor the status of the application.

Anatomy of a YARN Application Run

- YARN provides its core services via two types of long-running daemon:
  - a *resource manager* (one per cluster) to manage the use of resources across the cluster, and
  - *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).
- Depending on how YARN is configured, a container may be a Unix process or a Linux cgroup.

Anatomy of a YARN Application Run



- To run an application on YARN, a client contacts resource manager and asks it to run an *application master* process.

Resource Requests

- YARN has a flexible model for making resource requests.
- Locality is critical in ensuring that distributed data processing algorithms use cluster bandwidth efficiently.
- Sometimes the locality constraint cannot be met, constraint can be loosened.
- Will request a container on one of the nodes hosting the block's three replicas
- A YARN application can make resource requests at any time while it is running
- An application can make all of its requests up front, or it can take a more dynamic approach whereby it requests more resources dynamically
- Spark takes first approach, starting a fixed number of executors on cluster
- MapReduce : map task containers are requested up front, but reduce task containers are not started until later.
- If any tasks fail, additional containers will be requested so failed tasks can be rerun.

Application Lifespan

- Lifespan of a YARN application can vary dramatically: from few seconds to days or even months.
- 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 of (possibly unrelated) jobs.
- More efficient than the first, since containers can be reused between jobs
- Spark is an example that uses this model.

Application Lifespan

- 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.
- The "always on" application master means that users have very low latency responses to their queries since the overhead of starting a new application master is avoided

Building YARN Applications

- Writing a YARN application from scratch is fairly involved
- Interested in running a directed acyclic graph (DAG) of jobs, then Spark or Tez is appropriate.
- There are a couple of projects that simplify the process of building a YARN application.
- Apache Slider, mentioned earlier, 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
- Slider provides controls to change 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.
- Complex scheduling requirements—then the *distributed shell* application

A comparison of MapReduce 1 and YARN components

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

Benefits of using YARN

The benefits to using YARN include following:

- ❑ Scalability
- ❑ Availability
- ❑ Utilization
- ❑ Multitenancy

Benefits of using YARN - *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.
- In contrast to the jobtracker, each instance of an application—here, a MapReduce job—has a dedicated application master, which runs for the duration of the application.
- This model is actually closer to the original Google MapReduce paper, which describes how a master process is started to coordinate map and reduce tasks running on a set of workers.

Benefits of using YARN - *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, large amount of rapidly changing complex state in jobtracker's memory (each task status is updated every few seconds, for example) makes it very difficult to retrofit HA into the jobtracker service.
- With the jobtracker's responsibilities split between resource manager and application master in YARN, making service highly available became a divide and- conquer problem: provide HA for the resource manager, then for YARN applications (on a per-application basis).
- And indeed, Hadoop 2 supports HA both for the resource manager and for the application master for MapReduce jobs.

Benefits of using YARN - *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, which can happen in MapReduce 1.
- Furthermore, resources in YARN are fine grained

Benefits of using YARN - *Multitenancy*

- 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 process of upgrading MapReduce more manageable.
- However, that some parts of MapReduce, such as the job history server and shuffle handler, as well as YARN itself, still need to be upgraded across cluster.

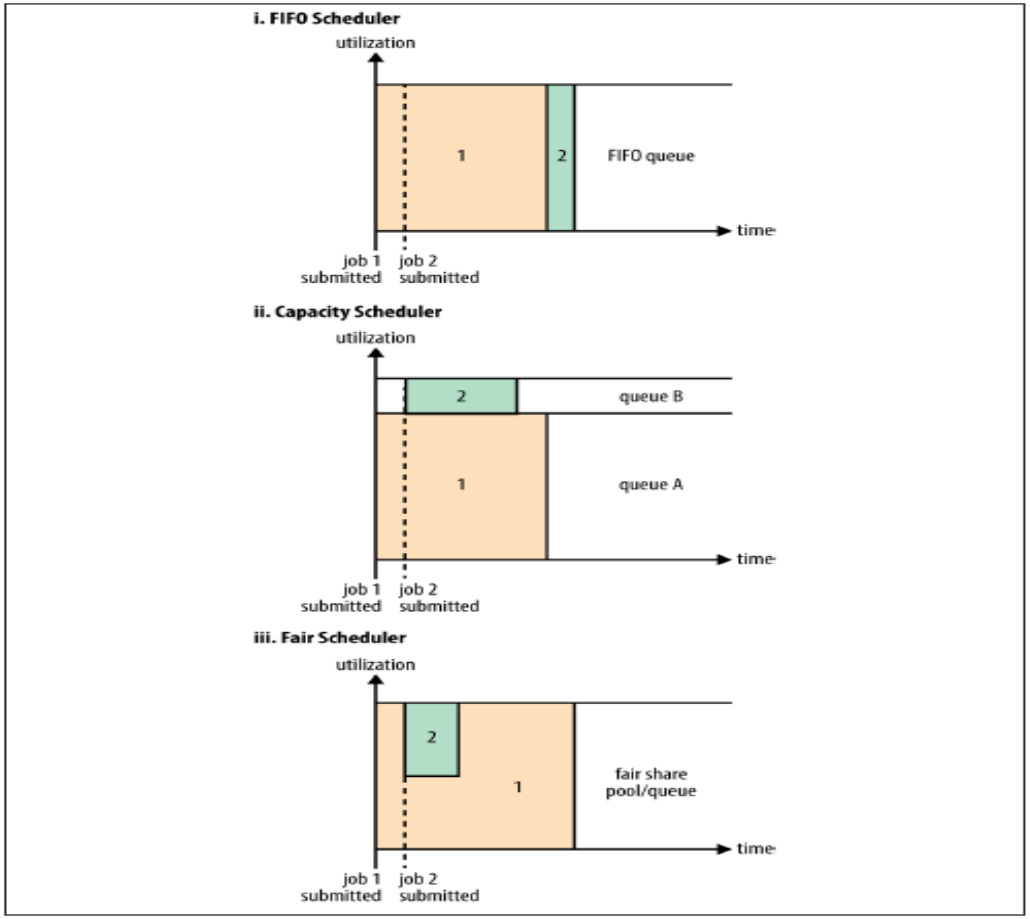
Scheduling in YARN

- In an ideal world, requests that a YARN application makes would be granted immediately.
- In real world, however, resources are limited, and on a busy cluster, an application will often need to wait to have some of its requests fulfilled.
- It is the job of the YARN scheduler to allocate resources to applications according to some defined policy.
- Scheduling in general is a difficult problem and there is no one “best” policy, which is why YARN provides a choice of schedulers and configurable policies.

Scheduling in YARN

- Three schedulers are available in YARN:
  1. the FIFO,
  2. Capacity, and
  3. Fair Schedulers
- FIFO Scheduler places applications in a queue and runs them in the order of submission
- Simple to understand
- Large applications will use all the resources in a cluster, so each application has to wait its turn
- On shared cluster it is better to use the Capacity Scheduler or the Fair Scheduler

Scheduling in YARN



Cluster utilization over time when running a large job and a small job under the FIFO Scheduler (i), Capacity Scheduler (ii), and Fair Scheduler (iii)

Scheduling in YARN

Capacity Scheduler

- With the Capacity Scheduler, a separate dedicated queue allows the small job to start as soon as it is submitted,
- This means that the large job finishes later than when using the FIFO Scheduler.

Scheduling in YARN

Fair Scheduler

- With the Fair Scheduler, there is no need to reserve a set amount of capacity
- It will dynamically balance resources between all running jobs.
- Just after the first (large) job starts, it is the only job running, so it gets all the resources in the cluster.
- When the second (small) job starts, it is allocated half of the cluster resources
- So that each job is using its fair share of resources

Capacity Scheduler Configuration

- The Capacity Scheduler allows sharing of a Hadoop cluster along organizational lines, whereby each organization is allocated a certain capacity of the overall cluster.
- Each organization is set up with a dedicated queue that is configured to use a given fraction of the cluster capacity.
- Queues may be further divided in hierarchical fashion, allowing each organization to share its cluster allowance between different groups of users within the organization.
- Within a queue, applications are scheduled using FIFO scheduling.

Imagine a queue hierarchy that looks like this:

```
root
├── prod
└── dev
    ├── eng
    └── science
```

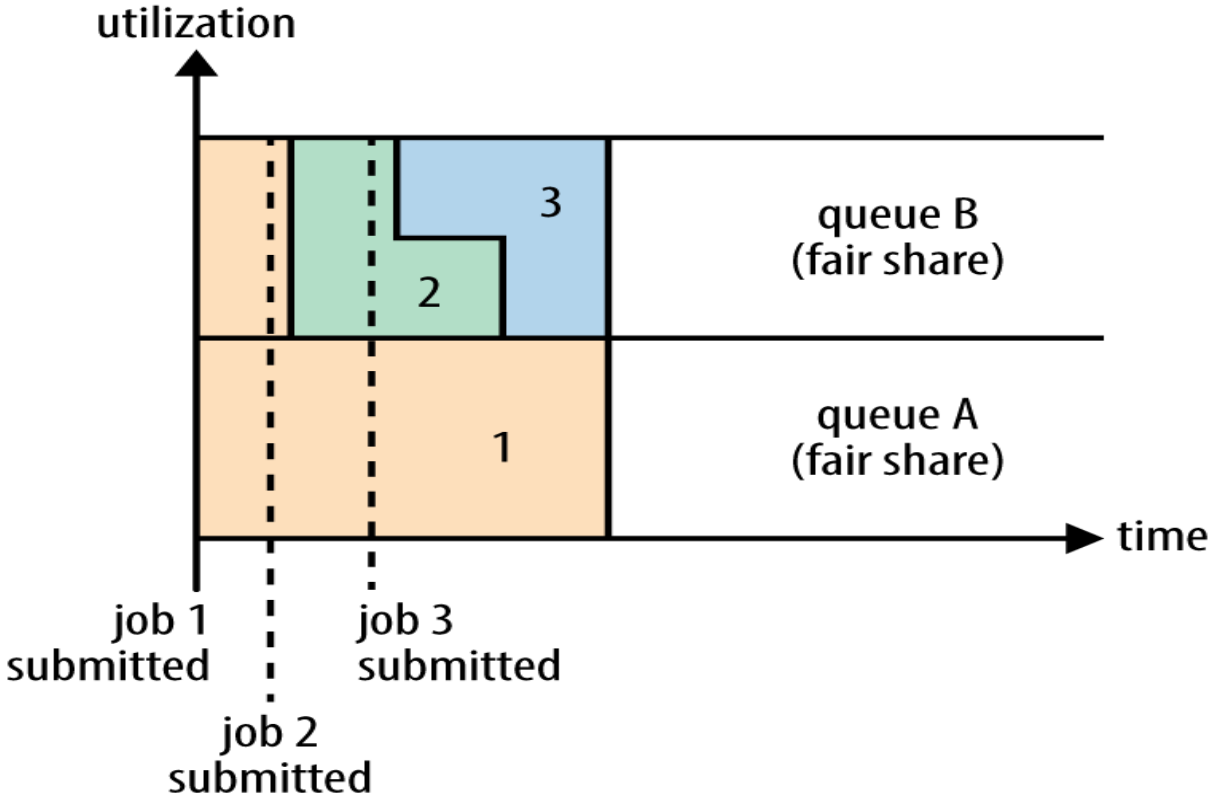
Capacity Scheduler Configuration

```
<?xml version="1.0"?>
<configuration>
  <property>
    <name>yarn.scheduler.capacity.root.queues</name>
    <value>prod,dev</value>
  </property>
  <property>
    <name>yarn.scheduler.capacity.root.dev.queues</name>
    <value>eng,science</value>
  </property>
  <property>
    <name>yarn.scheduler.capacity.root.prod.capacity</name>
    <value>40</value>
  </property>
  <property>
    <name>yarn.scheduler.capacity.root.dev.capacity</name>
    <value>60</value>
  </property>
  <property>
    <name>yarn.scheduler.capacity.root.dev.maximum-capacity</name>
    <value>75</value>
  </property>
  <property>
    <name>yarn.scheduler.capacity.root.dev.eng.capacity</name>
    <value>50</value>
  </property>
  <property>
    <name>yarn.scheduler.capacity.root.dev.science.capacity</name>
    <value>50</value>
  </property>
</configuration>
```

capacity-scheduler.xml

Fair Scheduler Configuration

- The Fair Scheduler attempts to allocate resources so that all running applications get the same share of resources.
- To understand how resources are shared between queues, imagine two users *A* and *B*, each with their own queue



Fair Scheduler Configuration

Enabling the Fair Scheduler

- Scheduler in use is determined by the setting of yarn.resourcemanager.scheduler.class.
- The Capacity Scheduler is used by default
- This can be changed by setting yarn.resourcemanager.scheduler.class in *yarn-site.xml* to the fully qualified classname of the scheduler, org.apache.hadoop.yarn.server.resourcemanager.scheduler.fair.FairScheduler.

Fair Scheduler Configuration

Queue configuration

Fair Scheduler is configured using an allocation file named *fair-scheduler.xml*

```
<?xml version="1.0"?>
<allocations>
<defaultQueueSchedulingPolicy>fair</defaultQueueSchedulingPolicy>
    <queue name="prod">
        <weight>40</weight>
        <schedulingPolicy>fifo</schedulingPolicy>
        </queue>
<queue name="dev">
        <weight>60</weight>
        <queue name="eng" />
<queue name="science" />
</queue>
<queuePlacementPolicy>
        <rule name="specified" create="false" />
        <rule name="primaryGroup" create="false" />
        <rule name="default" queue="dev.eng" />
</queuePlacementPolicy>
</allocations>
```

Hadoop I/O

Hadoop I/O

- 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.
- Small chance of introducing errors into the data that it is reading or writing
- Can detect corrupted data is by computing a *checksum* for the data
- Data is deemed to be corrupt if the newly generated checksum doesn't exactly match the original
- Doesn't offer any way to fix the data—it is merely error detection
- Possible that it's the checksum that is corrupt, not the data
- This is very unlikely, because checksum is much smaller than the data.

Data Integrity

- A commonly used error-detecting code is CRC-32 (32-bit cyclic redundancy check), which computes a 32-bit integer checksum for input of any size.
- CRC-32 is used for checksumming in Hadoop's ChecksumFileSystem, while HDFS uses a more efficient variant called CRC-32C.

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 dfs.bytes-perchecksum bytes of data. (default is 512 bytes).
- CRC-32C checksum is 4 bytes long, storage overhead is less than 1%.
- Datanodes are responsible for verifying the data they receive before storing the data and its checksum.
- Last datanode in the pipeline verifies the checksum.
- If the datanode detects an error, the client receives a subclass of IOException.
- When clients read data from datanodes, they verify checksums as well, comparing them with the ones stored at the datanodes.
- Data node keeps a persistent log of checksum verifications.
- Each datanode runs a DataBlockScan ner in a background thread

Data Integrity in HDFS

- Because 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 any more 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.

Data Integrity in HDFS

- It is possible to disable verification of checksums by passing false to the setVerify Checksum() method on FileSystem before using the open() method to read a file.
- The same effect is possible from the shell by using the -ignoreCrc option with the -get or the equivalent -copyToLocal command.
- This feature is useful if you have a corrupt file that you want to inspect so you can decide what to do with it.
- For example, you might want to see whether it can be salvaged before you delete it.

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.
- The chunk size is controlled by the file.bytes-per-checksum property
- The chunk size is stored as metadata in the *.crc* file,
- Checksums are verified when the file is read, and if an error is detected, LocalFileSystem throws a ChecksumException.

LocalFileSystem

- Checksums are fairly cheap to compute
- For most applications, this is an acceptable price to pay for data integrity.
- It is, however, possible to disable checksums, which is typically done when the underlying filesystem supports checksums natively.
- This is accomplished by using RawLocalFileSystem in place of LocalFileSystem.
- To do this globally in an application, set the property fs.file.impl
- Alternatively, you can directly create a RawLocalFileSystem instance
- 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);
```

ChecksumFileSystem

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

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

- 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.

Compression

- File compression brings two major benefits:
  1. it reduces the space needed to store files, and
  2. 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.
- There are many different compression formats, tools, and algorithms, each with different characteristics.
- Some of more common ones that can be used with Hadoop.

A summary of compression formats

Compression format	Tool	Algorithm	Filename extension	Splittable?
DEFLATE[a]	N/A	DEFLATE	*.deflate*	No
gzip	*gzip*	DEFLATE	*.gz*	No
bzip2	*bzip2*	bzip2	*.bz2*	Yes
LZO	*lzop*	LZO	*.lzo*	No[b]
LZ4	N/A	LZ4	*.lz4*	No
Snappy	N/A	Snappy	*.snappy*	No

Compression

- All compression algorithms exhibit a space/time trade-off: faster compression and decompression speeds usually come at the expense of smaller space savings.
- Tools 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, 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 generalpurpose compressor and sits in middle of the space/time trade-off.
- bzip2 compresses more effectively than gzip, but is slower
- LZO, LZ4, and Snappy optimize for speed.
- “Splittable” indicates whether compression format supports splitting

Codecs

- A *codec* is the implementation of a compression-decompression algorithm.
- In Hadoop, a codec is represented by an implementation of CompressionCodec interface.
- For example, GzipCodec encapsulates the compression and decompression algorithm for gzip.
- codecs that are available for Hadoop

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.BZip2Codec`
LZO	`com.hadoop.compression.lzo.LzopCodec`
LZ4	`org.apache.hadoop.io.compress.Lz4Codec`
Snappy	`org.apache.hadoop.io.compress.SnappyCodec`

Codecs

Compressing and decompressing streams with CompressionCodec

Example illustrates how to use the API to compress data read from standard input and write it to standard output.

A program to compress data read from 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.createOutputStream(System.out);
                IOUtils.copyBytes(System.in, out, 4096, false);
                out.finish();
        }
}
```

Use following command line

% **echo "Text" | hadoop StreamCompressor org.apache.hadoop.io.compress.GzipCodec \gunzip -**Text

Codecs

Native libraries

- For performance, it is preferable to use a **native library** for compression and decompression.
- For example, in one test, using the native gzip libraries reduced decompression times by up to 50% and compression times by around 10% (compared to the built-in Java implementation).
- Table shows the availability of Java and native implementations for each compression format.
- All formats have native implementations, but not all have a Java implementation (LZO, for example).

Compression library implementations

Compression format	Java implementation?	Native implementation?
DEFLATE	Yes	Yes
gzip	Yes	Yes
bzip2	Yes	Yes
LZO	No	Yes
LZ4	No	Yes
Snappy	No	Yes

Compression and Input Splits

- When considering how to compress data that will be processed by MapReduce, it is important to understand whether the compression format supports splitting.
- Consider an uncompressed file stored in HDFS whose size is 1 GB.
- With an HDFS block size of 128 MB, the file will be stored as eight blocks, and a MapReduce job using this file as input will create eight input splits, each processed independently as input to a separate map task.
- Imagine now that file is a gzip-compressed file & size is 1 GB.
- HDFS will store the file as eight blocks
- Impossible for a map task to read its split independently
- The gzip format uses DEFLATE, and DEFLATE stores data as a series of compressed blocks.
- In this case, MapReduce will do right thing and not try to split gzipped file, since it knows that input is gzip-compressed and that gzip does not support splitting.
- This will work, but at the expense of locality.

Using Compression in MapReduce

- If input files are compressed, they will be decompressed automatically as they are read by MapReduce

```
public class MaxTemperatureWithCompression {
public static void main(String[] args) throws Exception {
if (args.length != 2) {
System.err.println("Usage: MaxTemperatureWithCompression <input path> " +
"<output path>");
System.exit(-1);
}
Job job = new Job();
job.setJarByClass(MaxTemperature.class);
FileInputFormat.addInputPath(job, new Path(args[0]));
FileOutputFormat.setOutputPath(job, new Path(args[1]));
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);
FileOutputFormat.setCompressOutput(job, true);
FileOutputFormat.setOutputCompressorClass(job, GzipCodec.class);
job.setMapperClass(MaxTemperatureMapper.class);
job.setCombinerClass(MaxTemperatureReducer.class);
job.setReducerClass(MaxTemperatureReducer.class);
System.exit(job.waitForCompletion(true) ? 0 : 1);
}
}
```

Using Compression in MapReduce

- Run the program over compressed input

```
% hadoop MaxTemperatureWithCompression input/ncdc/sample.txt.gz
  output
```

- Each part of the final output is compressed; in this case, there is a single part:

```
% gunzip -c output/part-r-00000.gz
1949 111
1950 22
```

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 is used in two quite distinct areas of distributed data processing:
  1. for interprocess communication and
  2. for persistent storage.
- Hadoop, interprocess communication between nodes in the system is implemented using remote procedure calls (RPCs)
- The RPC protocol uses serialization to render message into a binary stream to be sent to the remote node, which then deserializes the binary stream into the original message.

Serialization

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 most scarce resource in a data center.
- ***Fast***
  - Interprocess communication - backbone for a distributed system,
  - Essential that little performance overhead as possible for serialization and deserialization process.
- ***Extensible***
  - Protocols change over time to meet new requirements
  - 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.

Serialization

- Data format chosen for persistent storage would have different requirements from a serialization framework.
- Lifespan of an RPC is less than a second, whereas persistent data may be read years after it was written.
- 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.

Serialization

The Writable Interface

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:

```
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;
}
```

Let's look at a particular Writable to see what we can do with it.

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);
```

Serialization

To examine the serialized form of the IntWritable, we write a small helper method

```
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));
```

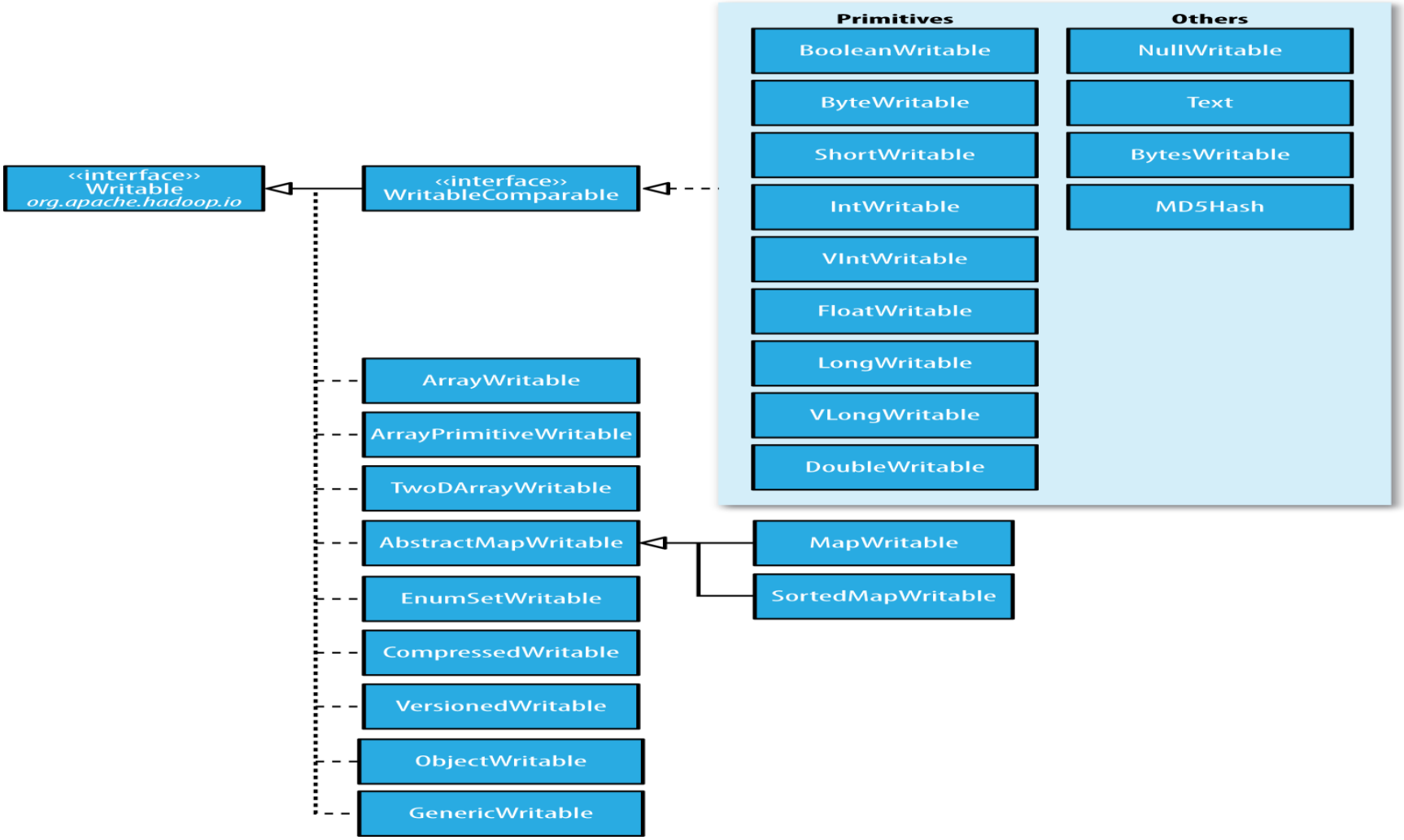
The bytes are written in big-endian order

Let's try deserialization. Again, we create a helper method.

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

Writable Classes

Hadoop comes with a large selection of Writable classes, which are available in the org.apache.hadoop.io package



Writable wrappers for Java primitives

- There are Writable wrappers for all the Java primitive types except char (which can be stored in an 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

Serialization Frameworks

- MapReduce programs use Writable key and value types, this isn't mandated by 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 (in the org.apache.hadoop.io.serializer package).
- 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).
- 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.

Serialization *interface description language (*IDL)

- Number of serialization frameworks that approach the problem in a different way: rather than defining types through code, you define them in a language neutral, declarative fashion, using an *interface description language* (IDL).
- The system can generate types for different languages, which is good for interoperability.
- They also typically define versioning schemes that make type evolution straightforward.
- Apache Thrift and Google Protocol Buffers are both popular serialization frameworks, and both are commonly used as a format for persistent binary data.
- There is limited support for these as MapReduce formats; however, they are used internally in parts of Hadoop for RPC and data exchange.
- Avro is an IDL-based serialization framework designed to work well with large-scale data processing in Hadoop.

File-Based Data Structures

- For some applications, you need a specialized data structure to hold your data.
- For doing MapReduce-based processing, putting each blob of binary data into its own file doesn't scale
- So Hadoop developed a number of higher-level containers for these situations.
- File Based containers are
  1. SequenceFile
  2. MapFile

SequenceFile

- Imagine a logfile where each log record is a new line of text.
- If you want to log binary types, plain text isn't a suitable format.
- Hadoop's SequenceFile class fits the bill in this situation, providing a persistent data structure for binary key-value pairs.
- To use it as a logfile format, you would choose a key, such as timestamp represented by a LongWritable, and value would be a Writable that represents the quantity being logged.
- SequenceFiles also work well as containers for smaller files.
- HDFS and MapReduce are optimized for large files, so packing files into a SequenceFile makes storing and processing the smaller files more efficient

Writing a SequenceFile

To create a SequenceFile, use one of its createWriter() static methods, which return a SequenceFile.Writer instance.

Example 5-10. Writing a SequenceFile

```
public class SequenceFileWriteDemo
{
        private static final String[] DATA = {
                "One, two, buckle my shoe",
                "Three, four, shut the door",
                "Five, six, pick up sticks",
                "Seven, eight, lay them straight",
                "Nine, ten, a big fat hen"
        };
public static void main(String[] args) throws IOException
{
        String uri = args[0];
        Configuration conf = new Configuration();
        FileSystem fs = FileSystem.get(URI.create(uri), conf);
        Path path = new Path(uri);
        IntWritable key = new IntWritable();
        Text value = new Text();
        SequenceFile.Writer writer = null;
```

Writing a SequenceFile

```
        try
        {
                writer = SequenceFile.createWriter(fs, conf, path,
                key.getClass(), value.getClass());
                for (int i = 0; i < 100; i++)
                {
                        key.set(100 - i);
                        value.set(DATA[i % DATA.length]);
                        System.out.printf("[%s]\t%s\t%s\n", writer.getLength(), key, value);
                        writer.append(key, value);
                }
        }
        finally
        {
                IOUtils.closeStream(writer);
        }
    }
}
```

Writing a SequenceFile

% **hadoop SequenceFileWriteDemo numbers.seq**
[128] 100 One, two, buckle my shoe
[173] 99 Three, four, shut the door
[220] 98 Five, six, pick up sticks
[264] 97 Seven, eight, lay them straight
[314] 96 Nine, ten, a big fat hen
[359] 95 One, two, buckle my shoe
[404] 94 Three, four, shut the door
[451] 93 Five, six, pick up sticks
[495] 92 Seven, eight, lay them straight
[545] 91 Nine, ten, a big fat hen
...
[1976] 60 One, two, buckle my shoe
[2021] 59 Three, four, shut the door
[2088] 58 Five, six, pick up sticks
[2132] 57 Seven, eight, lay them straight
[2182] 56 Nine, ten, a big fat hen
...
[4557] 5 One, two, buckle my shoe
[4602] 4 Three, four, shut the door
[4649] 3 Five, six, pick up sticks
[4693] 2 Seven, eight, lay them straight
[4743] 1 Nine, ten, a big fat hen

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.
- Which one to use depends on the serialization framework used.
- If you are using Writable types, then use

  public boolean next(Writable key, Writable val

- For other, non-Writable serialization frameworks (such as Apache Thrift), use

  public Object next(Object key) **throws** IOException

  public Object getCurrentValue(Object val) **throws** IOException

- next() method returns a non-null object, a key-value pair was read from stream, and value can be retrieved using the getCurrentValue() method.
- Otherwise, if next() returns null, the end of the file has been reached.

Reading a SequenceFile

Example - Reading a SequenceFile

```
public class SequenceFileReadDemo {
        public static void main(String[] args) throws IOException {
        String uri = args[0];
        Configuration conf = new Configuration();
        FileSystem fs = FileSystem.get(URI.create(uri), conf);
        Path path = new Path(uri);
        SequenceFile.Reader reader = null;
        try {
                reader = new SequenceFile.Reader(fs, path, conf);
                Writable key = (Writable)
                ReflectionUtils.newInstance(reader.getKeyClass(), conf);
                Writable value = (Writable)
                ReflectionUtils.newInstance(reader.getValueClass(), conf);
                long position = reader.getPosition();
                while (reader.next(key, value)) {
                        String syncSeen = reader.syncSeen() ? "*" : "";
                        System.out.printf("[%s%s]\t%s\t%s\n", position, syncSeen, key, value);
                        position = reader.getPosition(); // beginning of next record
                }
        } finally {
                IOUtils.closeStream(reader);
        }
    }
}
```

Reading a SequenceFile

- Another feature of the program is that it displays the positions of the *sync points* in the sequence file.
- **A sync point** is a point in the stream that can be used to resynchronize with a record boundary if the reader is "lost" - for example, after seeking to an arbitrary position in the stream.
- Sync points are recorded by SequenceFile.Writer, which inserts a special entry to mark the sync point every few records as a sequence file is being written.

Reading a SequenceFile

% hadoop SequenceFileReadDemo numbers.seq

[128] 100 One, two, buckle my shoe
[173] 99 Three, four, shut the door
[220] 98 Five, six, pick up sticks
[264] 97 Seven, eight, lay them straight
[314] 96 Nine, ten, a big fat hen
[359] 95 One, two, buckle my shoe
[404] 94 Three, four, shut the door
[451] 93 Five, six, pick up sticks
[495] 92 Seven, eight, lay them straight
[545] 91 Nine, ten, a big fat hen
[590] 90 One, two, buckle my shoe
...
[1976] 60 One, two, buckle my shoe
[2021*] 59 Three, four, shut the door
[2088] 58 Five, six, pick up sticks
[2132] 57 Seven, eight, lay them straight
[2182] 56 Nine, ten, a big fat hen
...
[4557] 5 One, two, buckle my shoe
[4602] 4 Three, four, shut the door
[4649] 3 Five, six, pick up sticks
[4693] 2 Seven, eight, lay them straight
[4743] 1 Nine, ten, a big fat hen

Displaying a SequenceFile with the command-line interface

- Hadoop fs command has a -text option to display sequence files in textual form.
- It looks at a file's magic number so that it can attempt to detect the type of the file and appropriately convert it to text.
- It can recognize gzipped files, sequence files, and Avro datafiles; otherwise, it assumes the input is plain text.
- Running it on the sequence file gives the following output:

```
% hadoop fs -text numbers.seq | head

100 One, two, buckle my shoe
99 Three, four, shut the door
98 Five, six, pick up sticks
97 Seven, eight, lay them straight
96 Nine, ten, a big fat hen
95 One, two, buckle my shoe
94 Three, four, shut the door
93 Five, six, pick up sticks
92 Seven, eight, lay them straight
91 Nine, ten, a big fat hen
```

Sorting and merging SequenceFiles

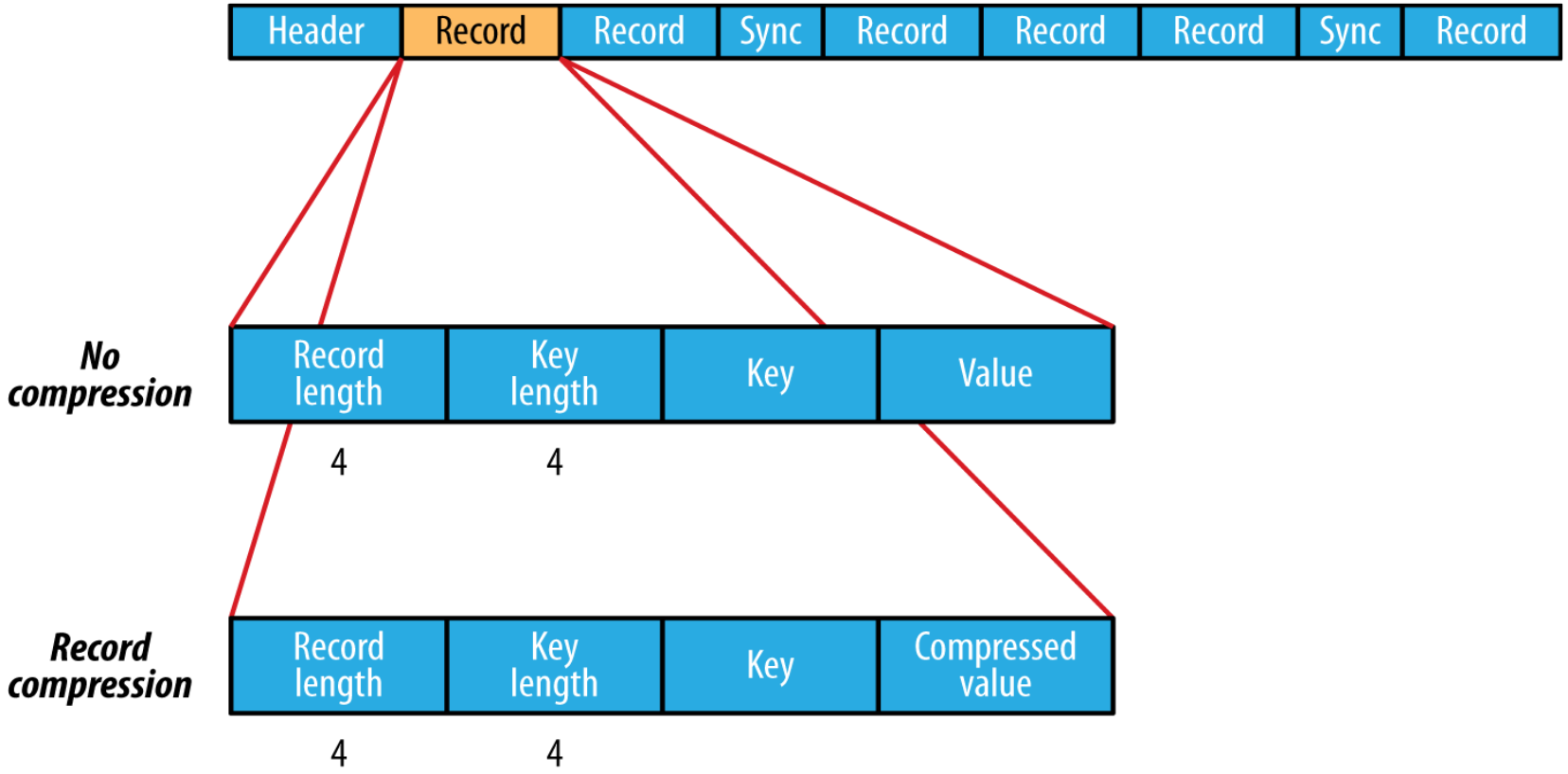
- The most powerful way of sorting (and merging) one or more sequence files is to use MapReduce.
- MapReduce is inherently parallel and will let you specify the number of reducers to use, which determines the number of output partitions.
- For example, by specifying one reducer, you get a single output file.
- Can use the sort example that comes with Hadoop by specifying that the input and output are sequence files and by setting the key and value types:

```
% hadoop jar \
$HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-examples-*.jar \
sort -r 1 \
-inFormat org.apache.hadoop.mapreduce.lib.input.SequenceFileInputFormat \
-outFormat org.apache.hadoop.mapreduce.lib.output.SequenceFileOutputFormat \
-outKey org.apache.hadoop.io.IntWritable \
-outValue org.apache.hadoop.io.Text \
numbers.seq sorted
```

The SequenceFile format

- A sequence file consists of a header followed by one or more records.
- First three bytes of a sequence file are bytes SEQ
- Followed by a single byte representing the version number.
- Header contains other fields, including names of the key and value classes, compression details, user defined metadata, and sync marker.
- Recall that 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.
- They are designed to incur less than a 1% storage overhead.

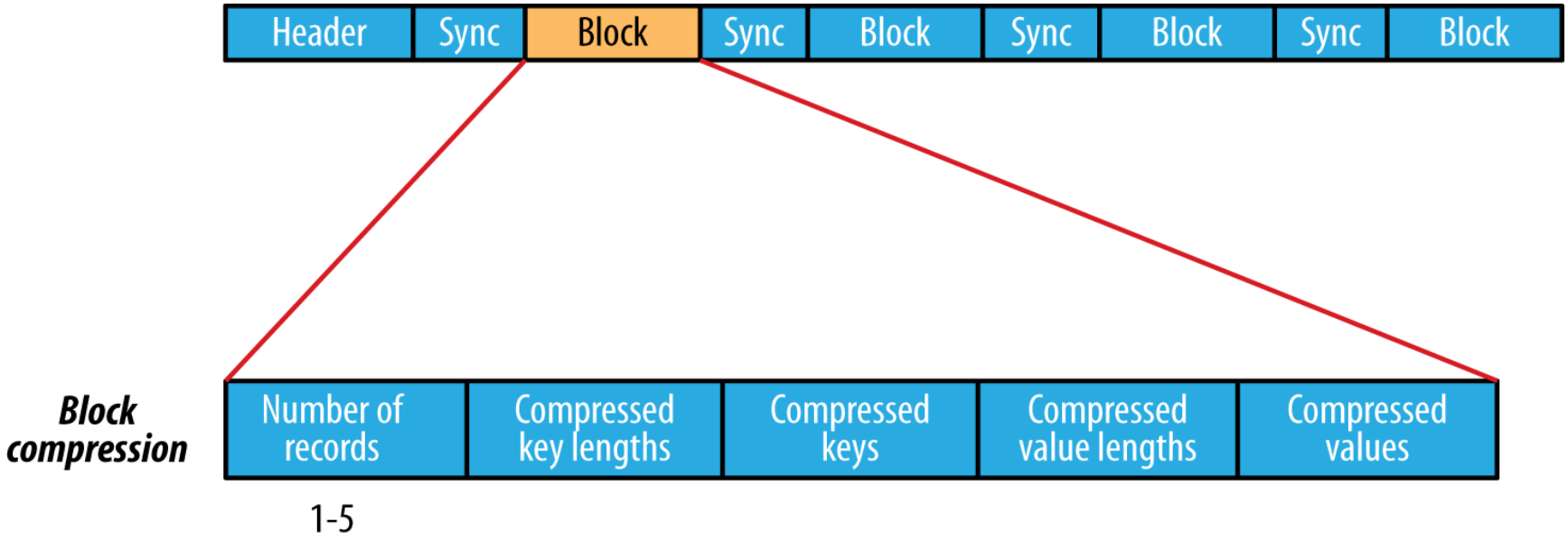
The SequenceFile format



The internal structure of a sequence file with no compression and with record compression

The SequenceFile format

- Block compression compresses multiple records at once
- 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 number of records in block, followed by four compressed fields: the key lengths, the keys, the value lengths, and values.



1-5

MapFile

- 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.
- 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

- Sequence and map files are the oldest binary file formats in Hadoop, there are better alternatives that should be considered for new projects.
- Avro datafiles are like sequence files in that they are designed for
  - large-scale data processing
  - they are compact and splittable
  - 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.

Other File Formats and Column-Oriented Formats

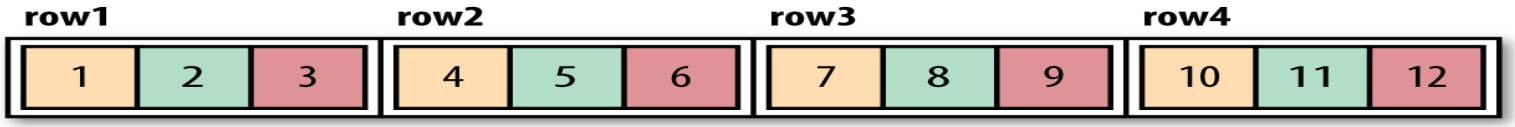
- Sequence files, map files, and Avro datafiles are all row-oriented file formats, which means that the values for each row are stored contiguously in the file.
- In a column oriented format,
  - the rows in a file (or, equivalently, a table in Hive) are broken up into row splits,
  - then each split 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.

Other File Formats and Column-Oriented Formats

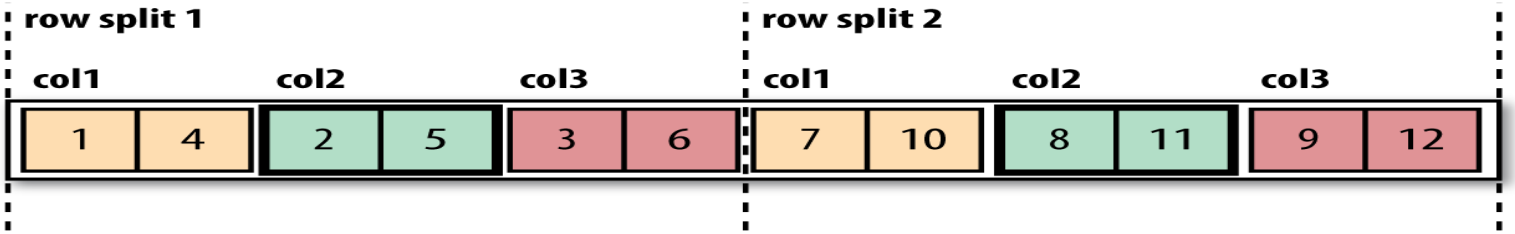
Logical table

	col1	col2	col3
row1	1	2	3
row2	4	5	6
row3	7	8	9
row4	10	11	12

Row-oriented layout (SequenceFile)



Column-oriented layout (RCFile)



Row-oriented versus column-oriented storage

Other File Formats and Column-Oriented Formats

- 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
- Column-oriented formats are not suited to streaming writes.
- On other hand, row-oriented formats like sequence files and Avro data files can be read up to last sync point after a writer failure.

Other File Formats and Column-Oriented Formats

- The first column-oriented file format in Hadoop was Hive's *RCFile*, short for Record Columnar File.
- It has since been superseded by Hive's ORCFile (*Optimized Record Columnar File*), 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.

Thank You !!!