A common way of avoiding data loss is through replication: redundant copies of the data are kept by the system so that in the event of failure, there is another copy available.

MapReduce is a *batch* query processor, and the ability to run an ad hoc query against your whole dataset and get the results in a reasonable time is transformative.

MapReduce is fundamentally a batch processing system, and is not

suitable for interactive analysis. You can’t run a query and get results back in a few

seconds or less.

The first component to provide online access was HBase, a key-value store that uses HDFS for its underlying storage. HBase provides both online read/write access of in-dividual rows and batch operations for reading and writing data in bulk, making it a good solution for building applications on.

YARN (which stands for *Yet Another Resource Negotiator*) in Hadoop 2

YARN is a cluster resource management system.

*Table 1-1. RDBMS compared to MapReduce*

**Traditional RDBMS MapReduce**

**Data size** Gigabytes Petabytes

**Access** Interactive and batch Batch

**Updates** Read and write many times Write once, read many times

**Transactions** ACID None

**Structure** Schema-on-write Schema-on-read

**Integrity** High Low

**Scaling** Nonlinear Linear

MapReduce is a good fit for problems that need to analyze the whole dataset

in a batch fashion, particularly for ad hoc analysis. An RDBMS is good for point queries or updates, where the dataset has been indexed to deliver low-latency retrieval and update times of a relatively small amount of data. MapReduce suits applications where the data is written once and read many times, whereas a relational database is good for datasets that are continually updated

the implementation detects failed tasks and reschedules replacements on machines that are healthy. MapReduce is able to do this because it is a

*shared-nothing* architecture, meaning that tasks have no dependence on one other.

a text input format that gives us each line in the dataset as a text value. The key is the offset of the beginning of the line from the beginning of the file,

Hadoop provides its own set of basic types that are optimized for network serialization.

“Counters,” shows the statistics that Hadoop generates for each job it runs. These are very useful for checking whether the amount of data processed is what you expected.

Hadoop divides the input to a MapReduce job into fixed-size pieces called *input splits*, or just *splits*. Hadoop creates one map task for each split, which runs the user-defined map function for each *record* in the split.

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

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.

The output of the reduce is normally stored in HDFS for reliability

HDFS is built around the idea that the most efficient data processing pattern is a

write-once, read-many-times pattern.

Applications that require low-latency access to data, in the tens of milliseconds

range, will not work well with HDFS

HBase (see

Chapter 20) is currently a better choice for low-latency access.

An HDFS cluster has two types of nodes operating in a master-worker pattern: a

*namenode* (the master) and a number of *datanodes* (workers

This information is stored persistently on the local disk in the form of two files: the namespace image and the edit log

Without the namenode, the filesystem cannot be used

*single point of failure* (SPOF)

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

*Scalability*

*Availability*

*Utilization*

*Multitenancy*

hree schedulers are available in YARN: the FIFO, Capacity, and Fair Schedulers. The

FIFO Scheduler places applications in a queue and runs them in the order of submission

(first in, first out). Requests for the first application in the queue are allocated first; once

its requests have been satisfied, the next application in the queue is served, and so on.

The FIFO Scheduler has the merit of being simple to understand and not needing any

configuration, but it’s not suitable for shared clusters. Large applications will use all the

resources in a cluster, so each application has to wait its turn. On a shared cluster it is

better to use the Capacity Scheduler or the Fair Scheduler. Both of these allow longrunning

jobs to complete

in a timely manner,

while still allowing users who are running

concurrent

smaller ad hoc queries to get results back in a reasonable time.

File compression brings two major benefits: it reduces the space needed to store files,and it speeds up data transfer across the network or to or from disk.

Compression formts:--

DEFLATE

Gzip

Bzip--- splitable

Lzo

Snappy

A *codec* is the implementation of a compression-decompression algorithm. In Hadoop,a codec is represented by an implementation of the CompressionCodec interface.

In order to compress the output of a MapReduce job, in the job configuration, set themapreduce.output.fileoutputformat.compress property to true and set the mapre

duce.output.fileoutputformat.compress.codec property to the classname of the

compression codec you want to use

**FileOutputFormat.setCompressOutput(job, true);**

**FileOutputFormat.setOutputCompressorClass(job, GzipCodec.class);**

Here are the lines to add to enable gzip map output compression in your job (using the

new API):

Configuration conf = **new** Configuration();

conf.setBoolean(Job.MAP\_OUTPUT\_COMPRESS, **true**);

conf.setClass(Job.MAP\_OUTPUT\_COMPRESS\_CODEC, GzipCodec.class,

CompressionCodec.class);

Job job = **new** Job(conf);

In the old API (see Appendix D), there are convenience methods on the JobConf object

for doing the same thing:

conf.setCompressMapOutput(**true**);

conf.setMapOutputCompressorClass(GzipCodec.class);

*Serialization* is the process of turning structured objects into a byte stream for trans-

mission 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: for

interprocess communication and for persistent storage.

Avro (a serialization system that was designed to overcome some of the limitations of Writables

Although most MapReduce programs use Writable key and value types, this isn’t man-

dated 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 seriali-zation 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)

Java comes with its own serialization mechanism, called Java Object Serialization (often

referred to simply as “Java Serialization”), that is tightly integrated with the language, so

it’s natural to ask why this wasn’t used in Hadoop

SequenceFiles also work well as containers for smaller files. HDFS and MapReduce areoptimized for large files, so packing files into a SequenceFile makes storing

and processing the smaller files more efficient

Fileformats:--

sequence files

map files

Avro datafiles (covered in “Avro Datafiles” on page 352) are like sequence files in that they are designed for large-scale data processing—they are compact and splittable—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

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.

MRUnit is a testing library that makes it easy to pass known inputs to a mapper or a reducer and check that the outputs are as expected.

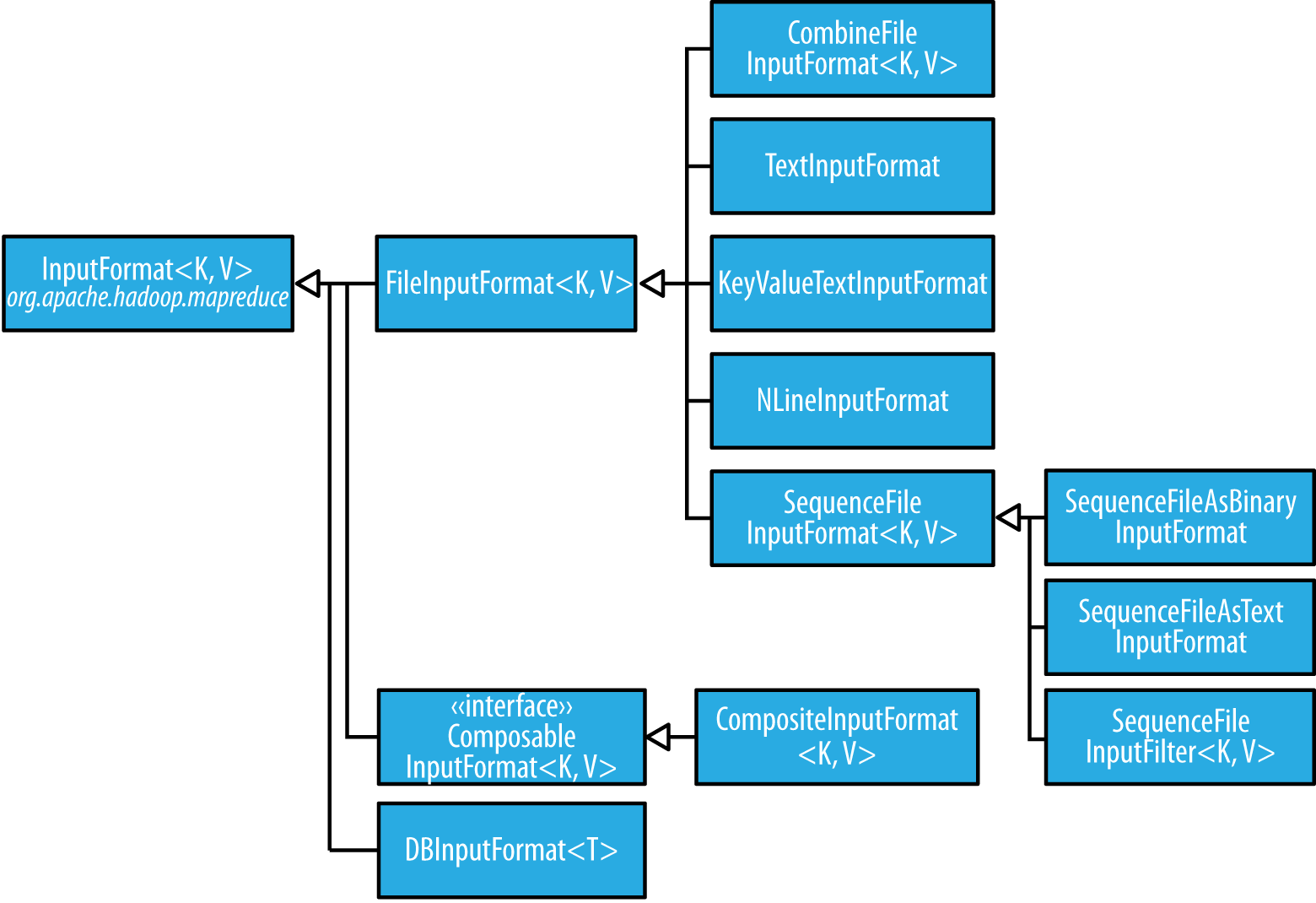
**FileInputFormat**

FileInputFormat is the base class for all implementations of InputFormat that use files

as their data source (see Figure 8-2). It provides two things: a place to define which files

are included as the input to a job, and an implementation for generating splits for the

input files. The job of dividing splits into records is performed by subclasses.



Whole file as inpu record

First, the format is careful to specify that input files

should never be split, by overriding isSplitable() to return false. Second, we

**228 | Chapter 8: MapReduce Types and Formats**

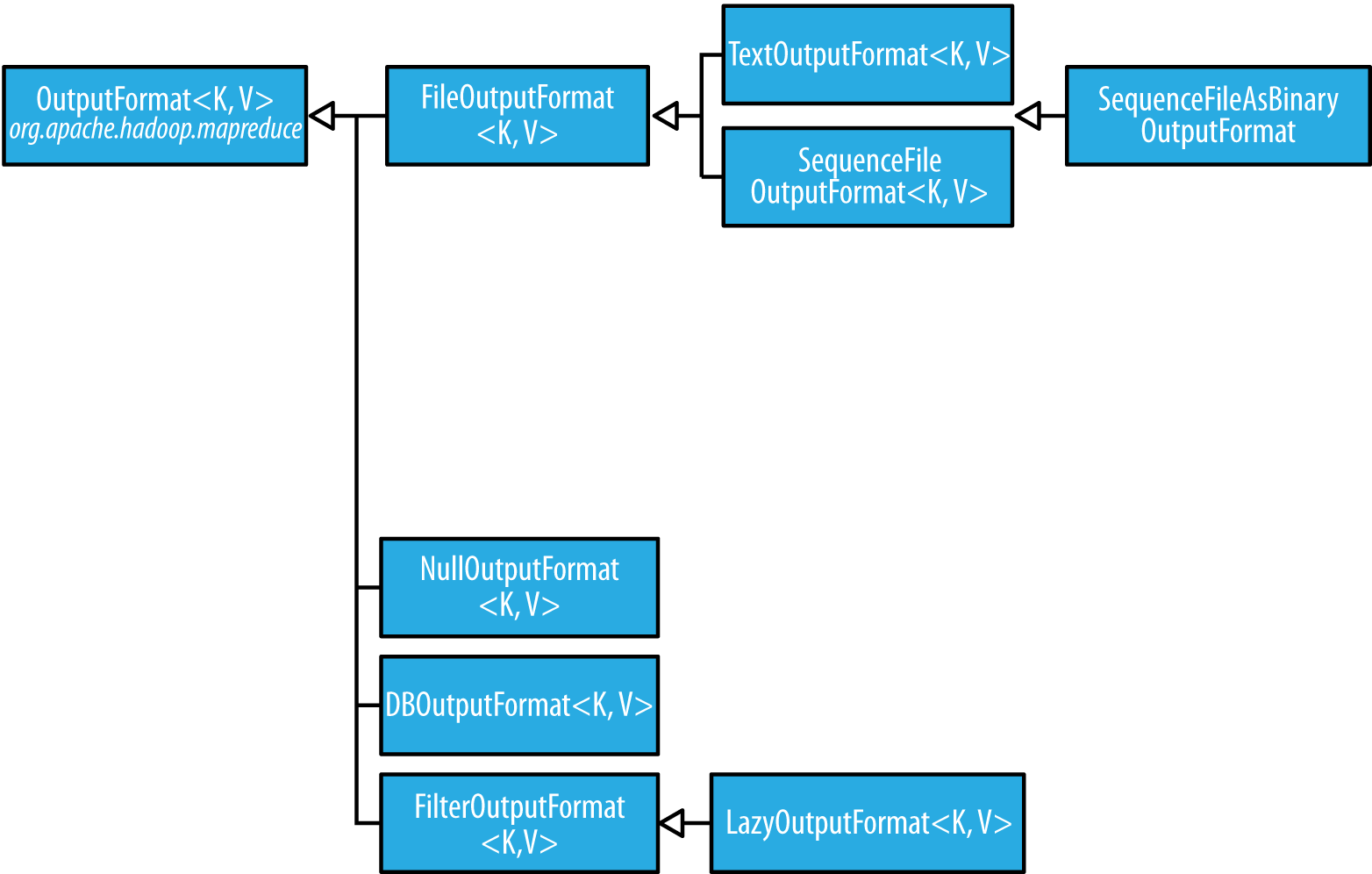
implement createRecordReader() to return a custom implementation of

RecordReader,

**Output Formats**

Hadoop has output data formats that correspond to the input formats covered in the

previous section. The OutputFormat class hierarchy appears in Figure 8-4.



, by default, MapReduce will sort input records by their keys.

The MapReduce framework sorts the records by key before they reach the reducers. For

any particular key, however, the values are *not* sorted. The order in which the values

appear is not even stable from one run to the next, because they come from different

map tasks, which may finish at different times from run to run.