Classic MapReduce:

A job run in classic MapReduce is illustrated in Figure 6-1. At the highest level, **there are four independent entities**:

• The client, which submits the MapReduce job.

• The jobtracker, which coordinates the job run. The jobtracker is a Java application whose main class is JobTracker.

• The tasktrackers, which run the tasks that the job has been split into. Tasktrackers are Java applications whose main class is TaskTracker.

• The distributed filesystem (normally HDFS, covered in Chapter 3), which is used for sharing job files between the other entities.

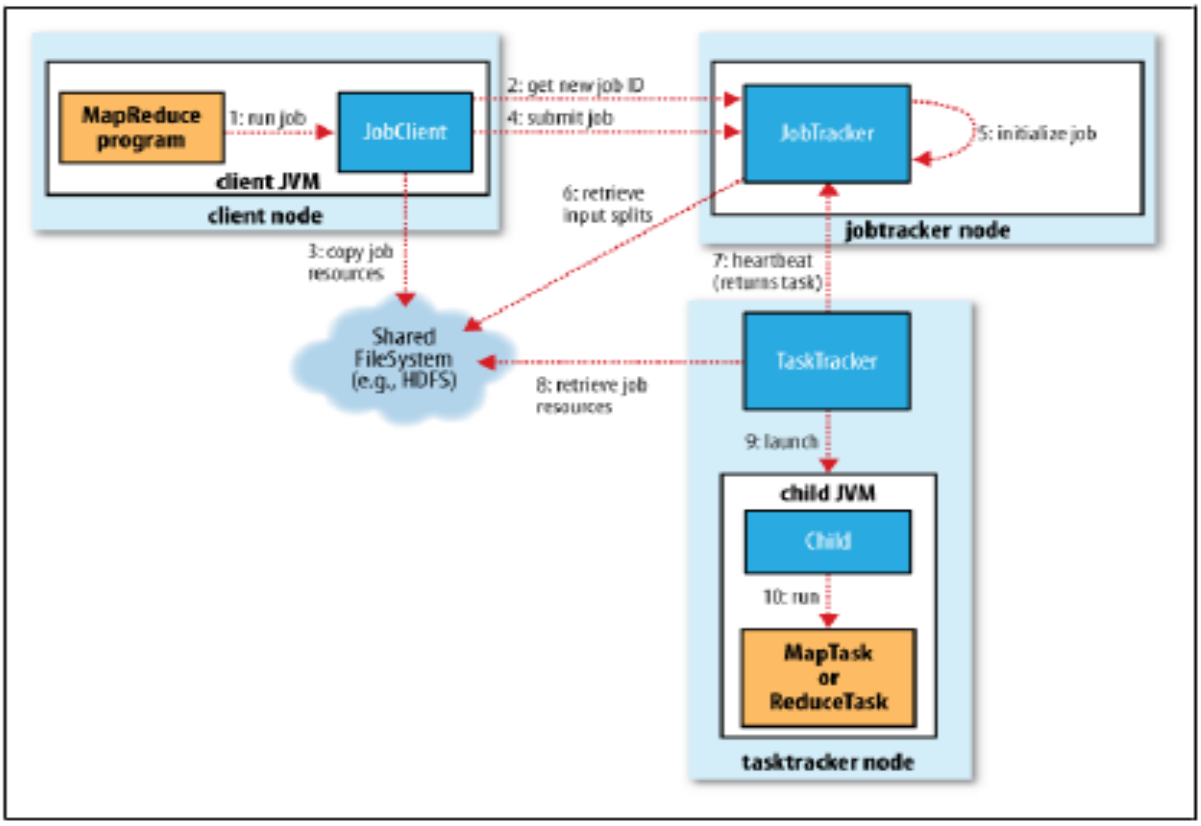


Figure 6-1. How Hadoop runs a MapReduce job using the classic framework

Job Submission:

The submit() method on Job creates an internal JobSummitter instance and calls submitJobInternal() on it (step 1 in Figure 6-1).

The job submission process implemented by JobSummitter does the following:

- Asks the jobtracker for a new job ID (by calling getNewJobId() on JobTracker) (step 2).
- Computes the input splits for the job. Copies the resources needed to run the job, including the job JAR file, the configuration file, and the computed input splits, to the jobtracker's filesystem in a directory named after the job ID. (step 3).

- Tells the jobtracker that the job is ready for execution (by calling submitJob() on JobTracker) (step 4).

Job Initialization:

When the JobTracker receives a call to its submitJob() method, it puts it into an internal queue from where the job scheduler will pick it up and initialize it. Initialization involves creating an object to represent the job being run (step 5).

To create the list of tasks to run, the job scheduler first retrieves the input splits computed by the client from the shared filesystem (step 6). It then creates one map task for each split.

Task Assignment:

Tasktrackers run a simple loop that periodically sends heartbeat method calls to the jobtracker. Heartbeats tell the jobtracker that a tasktracker is alive As a part of the heartbeat, a tasktracker will indicate whether it is ready to run a new task, and if it is, the jobtracker will allocate it a task, which it
communicates to the tasktracker using the heartbeat return value (step 7).

Task Execution:

Now that the tasktracker has been assigned a task, the next step is for it to run the task. First, it localizes the job JAR by copying it from the shared filesystem to the tasktracker's filesystem. It also copies any files needed from the distributed cache by the application to the local disk; see "Distributed Cache" on page 288 (step 8).

TaskRunner launches a new Java Virtual Machine (step 9) to run each task in (step 10).

Progress and Status Updates:

MapReduce jobs are long-running batch jobs, taking anything from minutes to hours to run. Because this is a significant length of time, it's important for the user to get feedback on how the job is progressing. A job and each of its tasks have a *status*.

When a task is running, it keeps track of its *progress*, that is, the proportion of the task completed.

Job Completion:

When the jobtracker receives a notification that the last task for a job is complete (this will be the special job cleanup task), it changes the status for the job to "successful."

Shuffle and Sort:

The Map Side :

MapReduce makes the guarantee that the input to every reducer is sorted by key. The process by which the system performs the sort—and transfers the map outputs to the reducers as inputs—is known as the shuffle.

Each map task has a circular memory buffer that it writes the output to. The buffer is 100 MB by default, a size which can be tuned by changing the io.sort.mb property.When the contents of the buffer reaches a certain threshold size (io.sort.spill.per cent, default 0.80, or 80%), a background thread will start to spill the contents to disk. Each time the memory buffer reaches the spill threshold, a new spill file is created. Spills are written in round-robin fashion to the directories. Before it writes to disk, the thread first divides the data into partitions corresponding to the reducers that they will ultimately be sent to. Within each partition, the background thread performs an in-memory sort by key, and if there is a combiner function, it is run on the output of the sort. Running the combiner function makes for a more compact map output, so there is less data to write to local disk and to transfer to the reducer.

so after the map task has written its last output record there could be several spill files.Before the task is finished, the spill files are merged into a single partitioned and sorted output file.

If there are at least three spill files (set by the min.num.spills.for.combine property) then the combiner is run again before the output file is written. If there are only one or two spills, then the potential reduction in map output size is not worth the overhead in invoking the combiner.

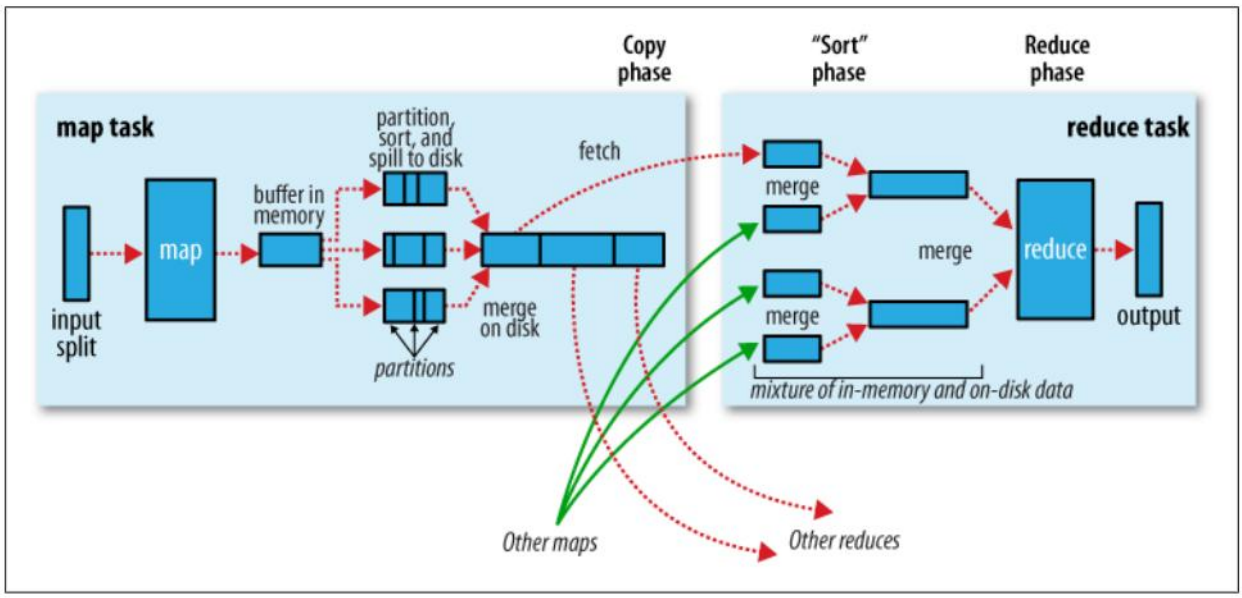


Figure 6-6. Shuffle and sort in MapReduce

The Reduce Side :

Let's turn now to the reduce part of the process.

The map output file is sitting on the local disk of the machine that ran the map task.But now it is needed by the machine that is about to run the reduce task for the partition. The reduce task needs the map output for its particular partition from several map tasks across the cluster. The copy phase of the reduce task. The reduce task has a small number of copier threads so that it can fetch map outputs in parallel. The default is five threads, but this number can be changed by setting the mapred.reduce.parallel.copies property.

The map outputs are copied to the reduce task JVM's memory if they are small enough,otherwise they are copied to disk.When the in-memory buffer reaches a threshold size (controlled by

mapred.job.shuffle.merge.percent), or reaches a threshold number of map outputs (mapred.inmem.merge.threshold), it is merged and spilled to disk.
When all the map outputs have been copied, the reduce task moves into the merge phase.which merges the map outputs, maintaining their sort ordering. This is done in rounds. For example, if there were 50 map outputs, and the merge factor was 10 (the default, controlled by the io.sort.factor property, just like in the map's merge),then there would be 5 rounds. Each round would merge 10 files into one, so at the end there would be five intermediate files.
During the reduce phase, the reduce function is invoked for each key in the sorted output. The output of this phase is written directly to the output filesystem, typically HDFS.

Configuration Tuning :

Table 6-1. Map-side tuning properties

Property name	Type	Default value	Description
io.sort.mb	int	100	The size, in megabytes, of the memory buffer to use while sorting map output.
io.sort.record.percent	float	0.05	The proportion of io.sort.mb reserved for storing record boundaries of the map outputs. The remaining space is used for the map output records themselves. This property was removed in release 0.21.0 as the shuffle code was improved to do a better job of using all the available memory for map output and accounting information.
io.sort.spill.percent	float	0.80	The threshold usage proportion for both the map output memory buffer and the record boundaries index to start the process of spilling to disk.
io.sort.factor	int	10	The maximum number of streams to merge at once when sorting files. This property is also used in the reduce. It's fairly common to increase this to 100.

min.num.spills.for.combine	int	3	The minimum number of spill files needed for the combiner to run (if a combiner is specified).
mapred.compress.map.output	boolean	false	Compress map outputs.
mapred.map.output.compression.codec	Class name	org.apache.hadoop.io.compress.DefaultCodec	The compression codec to use for map outputs.
task tracker.http.threads	int	40	The number of worker threads per tasktracker for serving the map outputs to reducers. This is a cluster-wide setting and cannot be set by individual jobs. Not applicable in MapReduce 2.

Table 6-2. Reduce-side tuning properties

Property name	Type	Default value	Description
mapred.reduce.parallel.copies	int	5	The number of threads used to copy map outputs to the reducer.
mapred.reduce.copy.backoff	int	300	The maximum amount of time, in seconds, to spend retrieving one map output for a reducer before declaring it as failed. The reducer may repeatedly reattempt a transfer within this time if it fails (using exponential backoff).
io.sort.factor	int	10	The maximum number of streams to merge at once when sorting files. This property is also used in the map.
mapred.job.shuffle.input.buffer.percent	float	0.70	The proportion of total heap size to be allocated to the map outputs buffer during the copy phase of the shuffle.
mapred.job.shuffle.merge.percent	float	0.66	The threshold usage proportion for the map outputs buffer (defined by mapred.job.shuffle.input.buffer.percent) for starting the process of merging the outputs and spilling to disk.
mapred.inmem.merge.threshold	int	1000	The threshold number of map outputs for starting the process of merging the outputs and spilling to disk. A value of 0 or less means there is no threshold, and the spill behavior is governed solely by mapred.job.shuffle.merge.percent.
mapred.job.reduce.input.buffer.percent	float	0.0	The proportion of total heap size to be used for retaining map outputs in memory during the reduce. For the reduce phase to begin, the size of map outputs in memory must be no more than this size. By

Hadoop Streaming

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.

Hadoop Pipes Hadoop Pipes is the name of the C++ interface to Hadoop MapReduce. Unlike Streaming, which uses standard input and output to communicate with the map and reduce code, Pipes uses sockets as the channel over which the tasktracker communicates with the process running the C++ map or reduce function. JNI is not used.

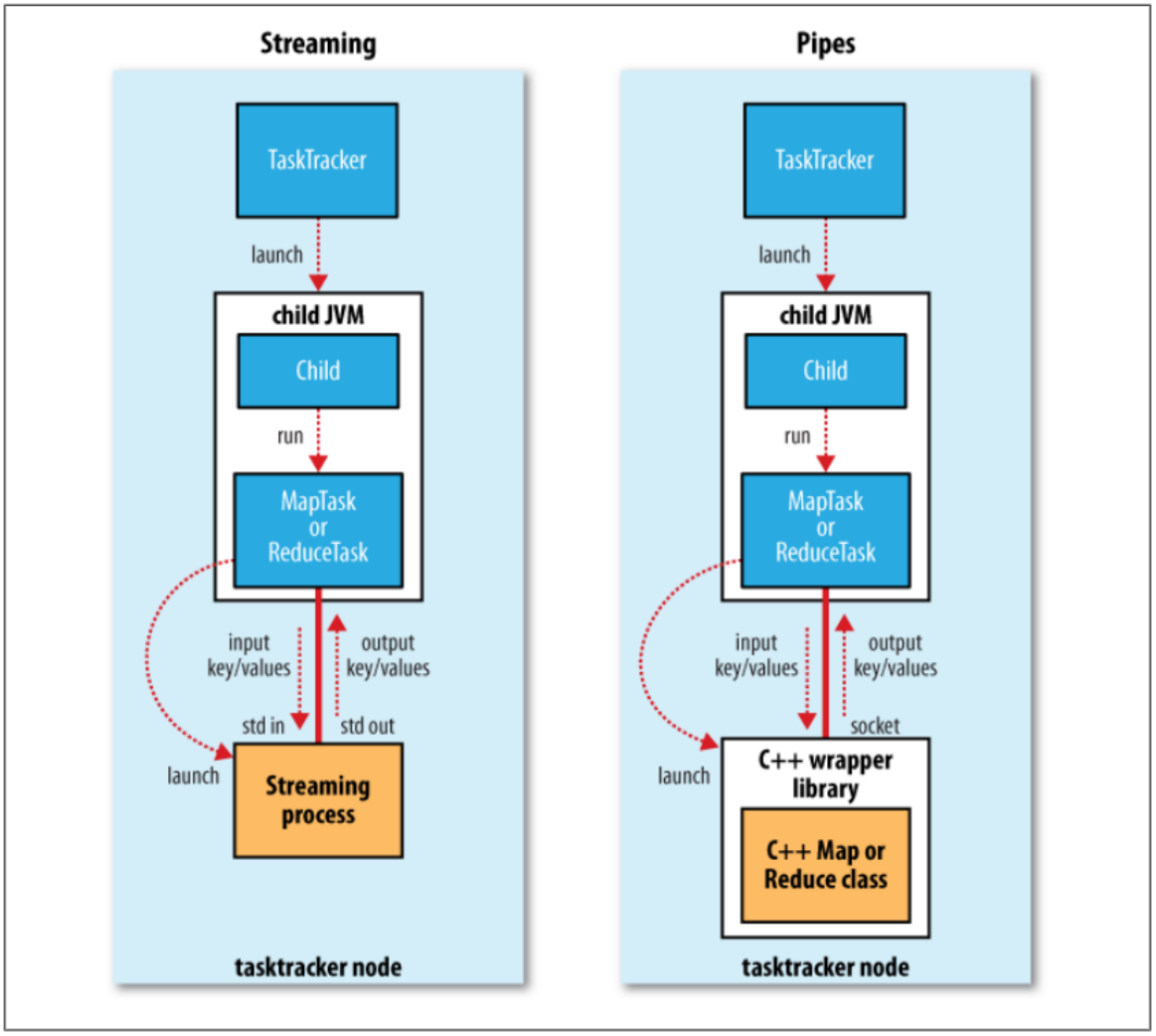


Figure 6-2. The relationship of the Streaming and Pipes executable to the tasktracker and its child

Map Reduce Types:

The map and reduce functions in Hadoop MapReduce have the following general form:

map: (K1, V1) → list(K2, V2)

reduce: (K2, list(V2)) → list(K3, V3)

In general, the map input key and value types (K1 and V1) are different from the map output types (K2 and V2). However, the reduce input must have the same types as the map output, although the reduce output types may be different again (K3 and V3).

```
public void map(LongWritable key, Text value, Context context)
{
        .....
        .....
        context.write(new Text(year), new IntWritable(airTemperature));
}

public void reduce(Text key, Iterable  <IntWritable> values, Context context)
{
        ..................
        context.write(key, new IntWritable(maxValue));
}
public void combiner(Text key, Iterable  <IntWritable> values, Context context)
{
        ..................
        context.write(key, new IntWritable(maxValue));
}
```

If a combine function is used, then it is the same form as the reduce function (and is an implementation of Reducer), except its output types are the intermediate key and value types (K2 and V2), so they can feed the reduce function:

map: (K1, V1) → list(K2, V2)
combine: (K2, list(V2)) → list(K2, V2)
reduce: (K2, list(V2)) → list(K3, V3)

Often the combine and reduce functions are the same, in which case, K3 is the same as K2, and V3 is the same as V2.

Input types are set by the input format. So, for instance, a TextInputFormat generates keys of type LongWritable and values of type Text. The other types are set explicitly by calling the methods on the Job as follows.

```
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);
```

So if K2 and K3 are the same, you don't need to call setMapOutputKeyClass(), since it falls back to the type set by calling setOutputKeyClass(). Similarly, if V2 and V3 are the same, you only need to use setOutputValueClass().

Table 7-1. Configuration of MapReduce types in the new API

Property	Job setter method	Input types		Intermediate types		Output types	
		K1	V1	K2	V2	K3	V3
Properties for configuring types:							
mapreduce.job.inputformat.class	setInputFormatClass()	•	•				
mapreduce.map.output.key.class	setMapOutputKeyClass()			•			
mapreduce.map.output.value.class	setMapOutputValueClass()				•		
mapreduce.job.output.key.class	setOutputKeyClass()					•	
mapreduce.job.output.value.class	setOutputValueClass()						•
Properties that must be consistent with the types:							
mapreduce.job.map.class	setMapperClass()	•	•	•	•		
mapreduce.job.combine.class	setCombinerClass()			•	•		
mapreduce.job.partitioner.class	setPartitionerClass()			•	•		
mapreduce.job.output.key.comparator.class	setSortComparatorClass()			•			
mapreduce.job.output.group.comparator.class	setGroupingComparatorClass()			•			
mapreduce.job.reduce.class	setReducerClass()			•	•	•	•
mapreduce.job.outputformat.class	setOutputFormatClass()					•	•

Input Formats:

Hadoop can process many different types of data formats, from flat text files to databases.

The Relationship Between Input Splits and HDFS Blocks Figure 7-3 shows an example. A single file is broken into lines, and the line boundaries do not correspond with the HDFS lock boundaries. Splits honor logical record boundaries,in this case lines, so we see that the first split contains line 5, even though it spans the first and second block. The second split starts at line 6.

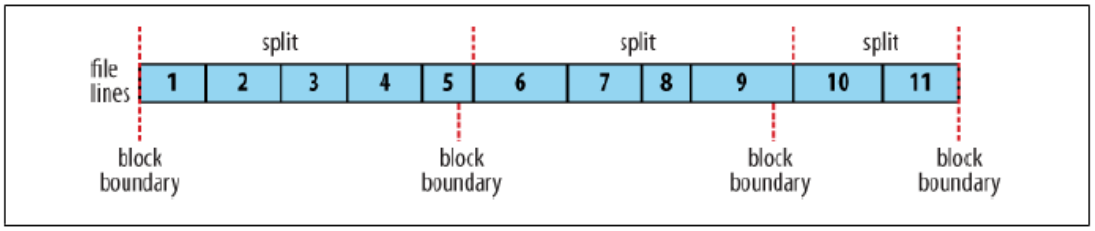


Figure 7-3. Logical records and HDFS blocks for TextInputFormat

Text Input:

TextInputFormat :

TextInputFormat is the default InputFormat. Each record is a line of input. The key, a LongWritable, is the byte offset within the file of the beginning of the line. The value is the contents of the line.

So a file containing the following text:
On the top of the Crumpetty Tree
The Quangle Wangle sat,
But his face you could not see,
On account of his Beaver Hat.

The records are interpreted as the following
key-value pairs:
(0, On the top of the Crumpetty Tree)
(33, The Quangle Wangle sat,)
(57, But his face you could not see,)
(89, On account of his Beaver Hat.)

KeyValueTextInputFormat

TextInputFormat's keys, being simply the offset within the file, are not normally very useful. It is common for each line in a file to be a key-value pair, separated by a delimiter such as a tab character.

You can specify the separator via the **mapreduce.input.keyvaluelinerecordreader.key.value.separator property** (or key.value.separator.in.input.line in the old API). It is a tab character by default. Consider the following input file, where → represents a (horizontal) tab character:

line1→On the top of the Crumpetty Tree
line2→The Quangle Wangle sat,
line3→But his face you could not see,
line4→On account of his Beaver Hat.

Like in the TextInputFormat case, the input is in a single split comprising four records, although this times the keys are the Text sequences before the tab in each line:

(line1, On the top of the Crumpetty Tree)
(line2, The Quangle Wangle sat,)
(line3, But his face you could not see,)
(line4, On account of his Beaver Hat.)

NLineInputFormat

With TextInputFormat and KeyValueTextInputFormat, each mapper receives a variable number of lines of input. The number depends on the size of the split and the length of the lines. If you want your mappers to receive a fixed number of lines of input, then NLineInputFormat is the InputFormat to use. Like TextInputFormat, the keys are the byte offsets within the file and the values are the lines themselves.

N refers to the number of lines of input that each mapper receives. With *N* set to one (the default), each mapper receives exactly one line of input.

On the top of the Crumpetty Tree
The Quangle Wangle sat,
But his face you could not see,
On account of his Beaver Hat.

If, for example, *N* is two, then each split contains two lines. One mapper will receive the first two key-value pairs:
(0, On the top of the Crumpetty Tree)
(33, The Quangle Wangle sat,)

And another mapper will receive the second two key-value pairs:
(57, But his face you could not see,)
(89, On account of his Beaver Hat.)

XML

Most XML parsers operate on whole XML documents, so if a large XML document is made up of multiple input splits, then it is a challenge to parse these individually.

Large XML documents that are composed of a series of "records" (XML document fragments) can be broken into these records using simple string or regular-expression matching to find start and end tags of records.

set your input format to **StreamInputFormat** and set the **stream.recordreader.class property to org.apache.hadoop.streaming.StreamXmlRecordReader** to use xml as an input format.

To take an example, Wikipedia provides dumps of its content in XML form, which are appropriate for processing in parallel using MapReduce using this approach.

Binary Input :

SequenceFileInputFormat

Hadoop's sequence file format stores sequences of binary key-value pairs. Sequence files are well suited as a format for MapReduce data since they are splittable they support compression as a part of the format

SequenceFileAsTextInputFormat

SequenceFileAsTextInputFormat is a variant of SequenceFileInputFormat that converts the sequence file's keys and values to Text objects.

SequenceFileAsBinaryInputFormat

SequenceFileAsBinaryInputFormat is a variant of SequenceFileInputFormat that retrieves the sequence file's keys and values as opaque binary objects.

Multiple Inputs

The input to a MapReduce job may consist of multiple input files. This case is handled elegantly by using the **MultipleInputs** class. For example, if we had weather data from the **UK Met Office6** that we wanted to combine with the **NCDC** data for our maximum temperature analysis, then we might set up the input as follows:

MultipleInputs.addInputPath(job, ncdcInputPath, TextInputFormat.class, MaxTemperatureMapper.class);

MultipleInputs.addInputPath(job, metOfficeInputPath, TextInputFormat.class, MetOfficeMaxTemperatureMapper.class);

Database Input

DBInputFormat is an input format for reading data from a relational database, using JDBC. It is best used for loading relatively small datasets, perhaps for joining with larger datasets from HDFS, using MultipleInputs.

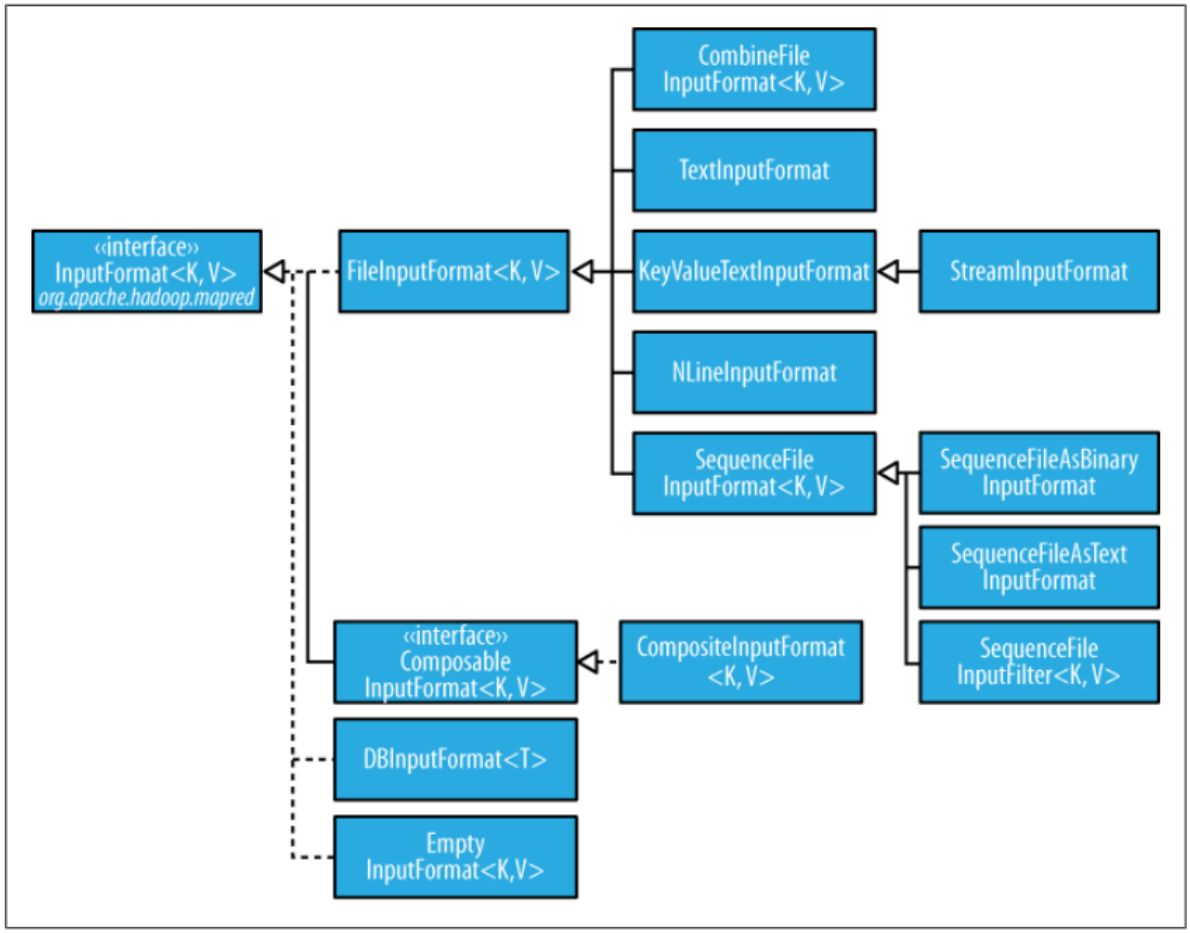


Figure 7-2. InputFormat class hierarchy

Output Formats

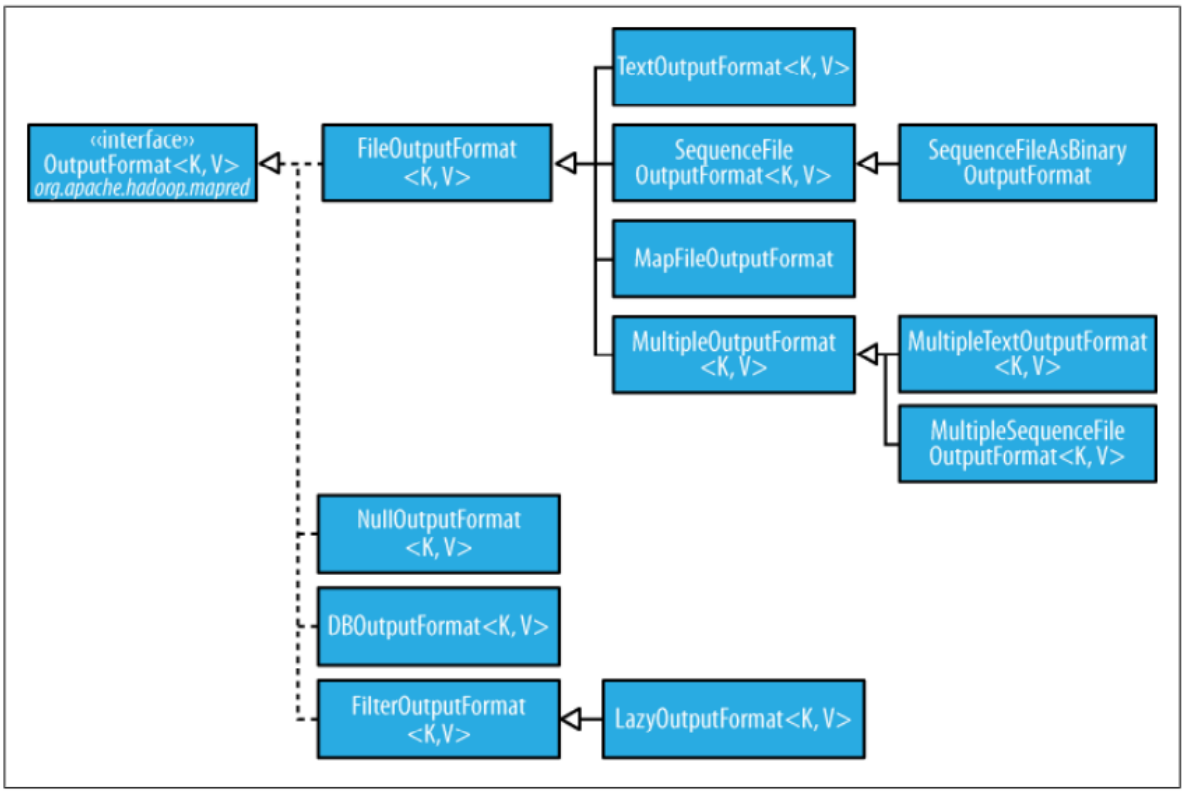


Figure 7-4. OutputFormat class hierarchy

Text Output

The default output format, **TextOutputFormat**, writes records as lines of text. Its keys and values may be of any type, since TextOutputFormat turns them to strings by calling toString() on them. Each key-value pair is separated by a tab character. The counterpart to TextOutput Format for reading in this case is KeyValueTextInputFormat.

Binary Output

SequenceFileOutputFormat As the name indicates, SequenceFileOutputFormat writes sequence files for its output.

SequenceFileAsBinaryOutputFormat

SequenceFileAsBinaryOutputFormat is the counterpart to SequenceFileAsBinaryInput Format, and it writes keys and values in raw binary format into a SequenceFile container.

MapFileOutputFormat

MapFileOutputFormat writes MapFiles as output. The keys in a MapFile must be added in order, so you need to ensure that your reducers emit keys in sorted order.

Multiple Outputs

FileOutputFormat and its subclasses generate a set of files in the output directory. There is one file per reducer, and files are named by the partition number: part-r-00000, partr-00001, etc. There is sometimes a need to have more control over the naming of the files or to produce multiple files per reducer. MapReduce comes with the **MultipleOutputs** class to help you do this.

An example: Partitioning data Consider the problem of partitioning the weather dataset by weather station. We would like to run a job whose output is a file per station, with each file containing all the records for that station.

One way of doing this is to have a reducer for each weather station. To arrange this, we need to do two things. First, write a partitioner that puts records from the same weather station into the same partition. Second, set the number of reducers on the job to be the number of weather stations. The partitioner would look like this:

```
public class StationPartitioner extends Partitioner<LongWritable, Text> {

 private NcdcRecordParser parser = new NcdcRecordParser();

 @Override
 public int getPartition(LongWritable key, Text value, int numPartitions) {
 parser.parse(value);
 return getPartition(parser.getStationId());
 }
 private int getPartition(String stationId) {
 ...
 }
}
```

Lazy Output

FileOutputFormat subclasses will create output (part-r-nnnnn) files, even if they are empty. Some applications prefer that empty files not be created, which is where **LazyOutputFormat** helps.

Database Output

The output formats for writing to relational databases and to HBase. DBOutputFormat, which is useful for dumping job outputs (of modest size) into a database.

A partitioner works like a condition in processing an input dataset. The partition phase takes place after the Map phase and before the Reduce phase.

The number of partitioners is equal to the number of reducers. That means a partitioner will divide the data according to the number of reducers. Therefore, the data passed from a single partitioner is processed by a single Reducer.

Partitioner

A partitioner partitions the key-value pairs of intermediate Map-outputs. It partitions the data using a user-defined condition, which works like a hash function. The total number of partitions is same as the number of Reducer tasks for the job. Let us take an example to understand how the partitioner works.

The default partitioner is the hash partitioner.

Custom partitioner

name<tab>age<tab>gender<tab>score

Input

Alice<tab>23<tab>female<tab>45

Bob<tab>34<tab>male<tab>89

Chris<tab>67<tab>male<tab>97

Kristine<tab>38<tab>female<tab>53

Connor<tab>25<tab>male<tab>27

Daniel<tab>78<tab>male<tab>95

James<tab>34<tab>male<tab>79

Alex<tab>52<tab>male<tab>69

Nancy<tab>7<tab>female<tab>98

Adam<tab>9<tab>male<tab>37

Jacob<tab>7<tab>male<tab>23

Mary<tab>6<tab>female<tab>93

Clara<tab>87<tab>female<tab>72

Monica<tab>56<tab>female<tab>92

PartitionMapper

PartitionMapper prepares the data for the partitioner and the reducer. It parses the input records and emits key-value pairs where the key is the gender and the value is t
information associated with a person.

```
1. //mapper output format : gender is the key, the value is formed by concatenating the name, age and t
   score
2.
3.     // the type parameters are the input keys type, the input values type, the
4.     // output keys type, the output values type
5.        @Override
6.     public static class PartitionMapper extends
7.             Mapper<Object, Text, Text, Text> {
8.
9.
10.        public void map(Object key, Text value, Context context)
11.                throws IOException, InterruptedException {
12.
13.            String[] tokens = value.toString().split("\t");
14.
15.            String gender = tokens[2].toString();
16.            String nameAgeScore = tokens[0]+"\t"+tokens[1]+"\t"+tokens[3];
17.
18.            //the mapper emits key, value pair where the key is the gender and the value is the othe
   information which includes name, age and score
19.            context.write(new Text(gender), new Text(nameAgeScore));
20.        }
21.    }
```

```java
//AgePartitioner is a custom Partitioner to partition the data according to age.
    //The age is a part of the value from the input file.
    //The data is partitioned based on the range of the age.
    //In this example, there are 3 partitions, the first partition contains the information where the age is less than 20
    //The second partition contains data with age ranging between 20 and 50 and the third partition contains data where the age is >50.
    public static class AgePartitioner extends Partitioner<Text, Text> {

        @Override
        public int getPartition(Text key, Text value, int numReduceTasks) {

            String [] nameAgeScore = value.toString().split("\t");
            String age = nameAgeScore[1];
            int ageInt = Integer.parseInt(age);

            //this is done to avoid performing mod with 0
            if(numReduceTasks == 0)
                return 0;

            //if the age is <20, assign partition 0
            if(ageInt <=20){
                return 0;
            }
            //else if the age is between 20 and 50, assign partition 1
            if(ageInt >20 && ageInt <=50){

                return 1 % numReduceTasks;
            }
            //otherwise assign partition 2
            else
                return 2 % numReduceTasks;

        }
    }
```

Partial sorting:

When a reducer receives those pairs they are sorted by key, so generally the output of a reducer is also sorted by key. However, the outputs of different reducers are not ordered between each other, so they cannot be concatenated or read sequentially in the correct order.

For example with 2 reducers, sorting on simple Text keys, you can have :
Reducer 1 output : (**a**,5), (**d**,6), (**w**,5)
Reducer 2 output : (**b**,2), (**c**,5), (**e**,7)
The keys are only sorted if you look at each output individually, but if you read one after the other, the ordering is broken.

Total Sort:

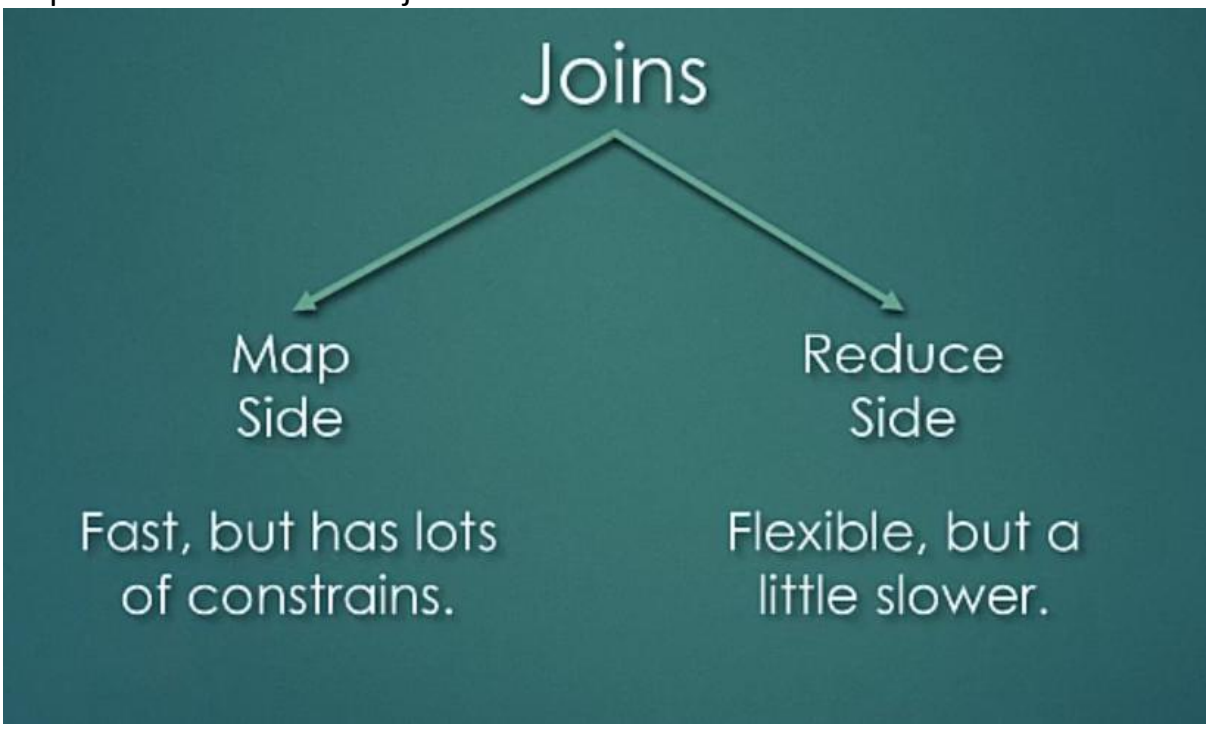
The objective of *Total Order Sorting* is to have all outputs sorted **across all reducers** :
Reducer 1 output : (**a**,5), (**b**,2), (**c**,5)
Reducer 2 output : (**d**,6), (**e**,7), (**w**,5)
This way the outputs can be read/searched/concatenated sequentially as a single ordered output.

Note: Use **TotalOrderPartitioner** to get global sorting.

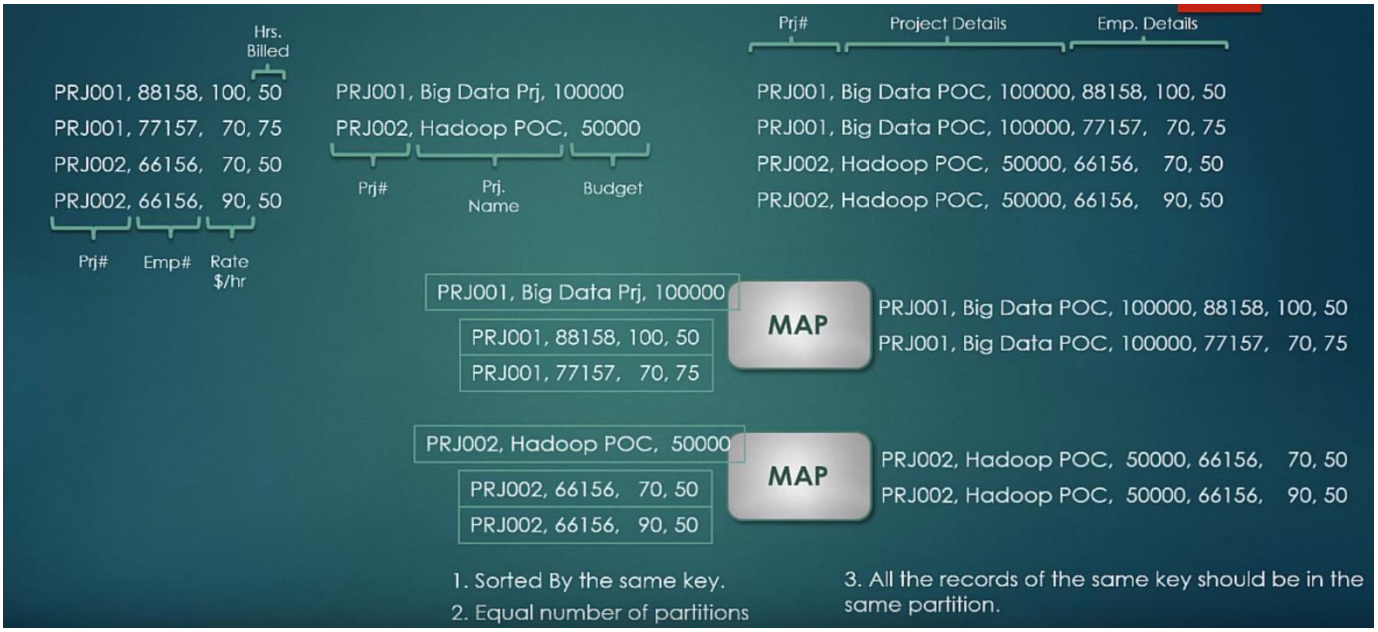
Example:

```
1  public class TotalOrderPartitionerExample {
2
3      public static void main(String[] args) throws Exception {
4
5          // Create job and parse CLI parameters
6          Job job = Job.getInstance(new Configuration(), "Total Order Sorting exa
7          job.setJarByClass(TotalOrderPartitionerExample.class);
8
9          Path inputPath = new Path(args[0]);
10         Path partitionOutputPath = new Path(args[1]);
11         Path outputPath = new Path(args[2]);
12
13         // The following instructions should be executed before writing the par
14         job.setNumReduceTasks(3);
15         FileInputFormat.setInputPaths(job, inputPath);
16         TotalOrderPartitioner.setPartitionFile(job.getConfiguration(), partitio
17         job.setInputFormatClass(KeyValueTextInputFormat.class);
18         job.setMapOutputKeyClass(Text.class);
19
20         // Write partition file with random sampler
21         InputSampler.Sampler<Text, Text> sampler = new InputSampler.RandomSampl
22         InputSampler.writePartitionFile(job, sampler);
23
24         // Use TotalOrderPartitioner and default identity mapper and reducer
25         job.setPartitionerClass(TotalOrderPartitioner.class);
26         job.setMapperClass(Mapper.class);
27         job.setReducerClass(Reducer.class);
28
29         FileOutputFormat.setOutputPath(job, outputPath);
30         System.exit(job.waitForCompletion(true) ? 0 : 1);
31     }
32 }
```

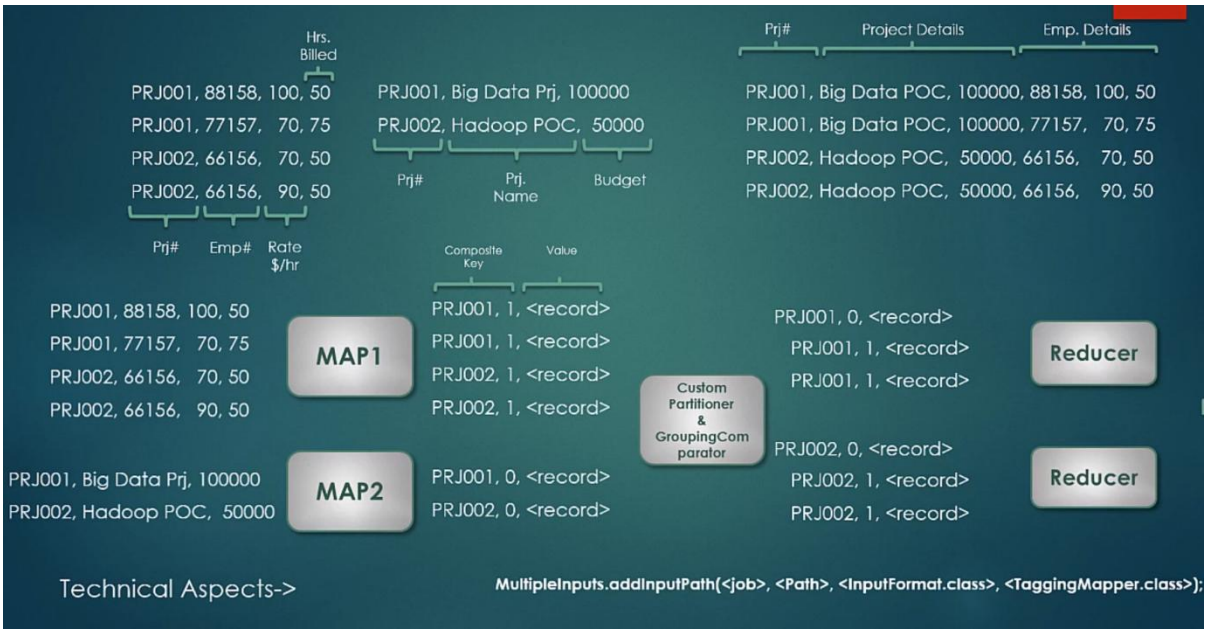
Map side and Reduce side joins :



Map Side Join:



Reduce Side Join:



Secondary Sort :

A *secondary sort problem* relates to sorting values associated with a key in the reduce phase. Sometimes, it is called *value-to-key conversion*. The secondary sorting technique will enable us to sort the values (in ascending or descending order) passed to each reducer.

A dump of the temperature data might look something like the following (columns are year, month, day, and daily temperature, respectively):

2012, 01, 01, 5
2012, 01, 02, 45

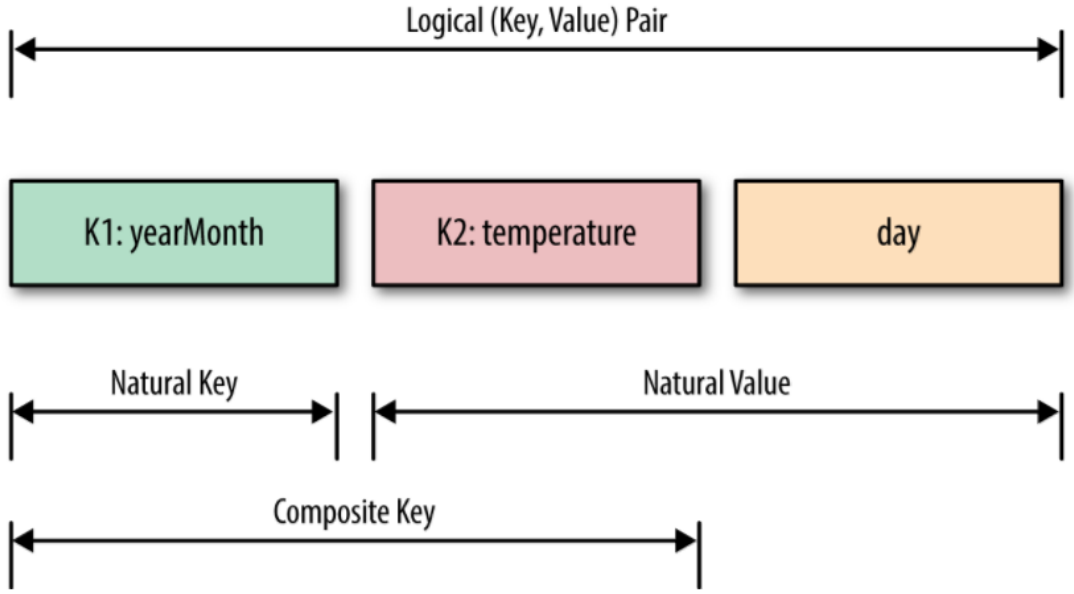
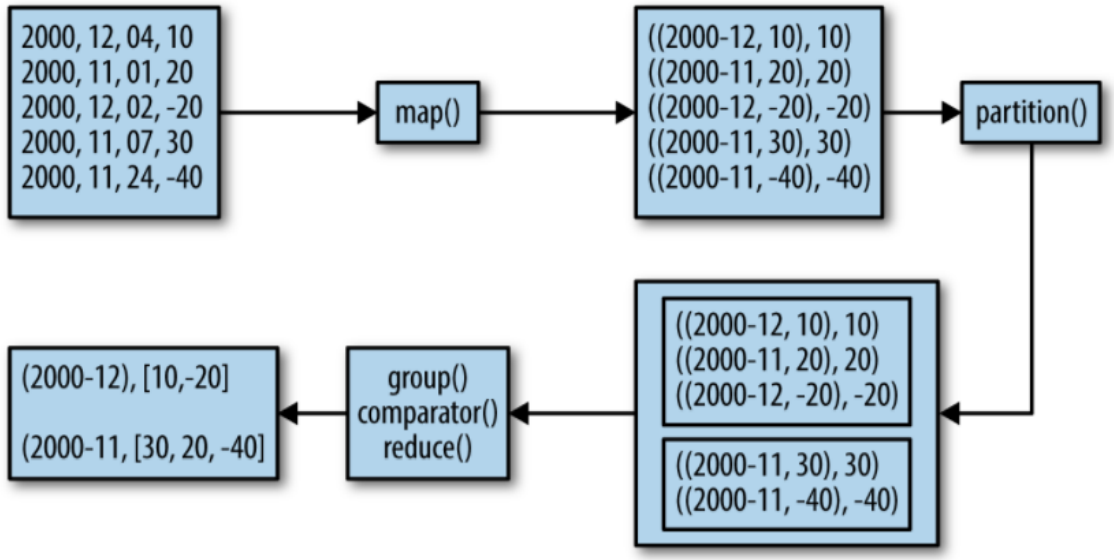


Figure 1-1. Secondary sorting keys



Secondary sorting data flow

The mappers create (*K*,*V*) pairs, where *K* is a composite key of (year,month,temperature) and *V* is temperature. The (year,month) part of the composite key is the natural key. The partitioner plug-in class enables us to send all natural keys to the same reducer

and the grouping comparator plug-in class enables temperatures to arrive sorted at reducers. The Secondary Sort design pattern uses MapReduce's framework for sorting the reducers' values rather than collecting them all and then sorting them in memory. The Secondary Sort design pattern enables us to "scale out" no matter how many reducer values we want to sort.