In a typical MapReduce job, key-value pairs are emitted from the mappers, shuffled, and sorted, and then finally passed to the reducers. There is no attempt by the MapReduce framework to sort the values passed to the reducers for processing. However, there are cases when we need the values passed to the reducers to be sorted, such as in the case of counting page views.

To calculate page views, we need to calculate distinct IPs by page. One way to calculate this is to have the mappers emit the key-value pairs: page and IP. Then, in the reducer, we can store all of the IPs associated with a page in a set. However, this approach is not scalable. What happens if the weblogs contain a large number of distinct IPs visiting a single page? We might not be able to fit the entire set of distinct IPs in memory.

The MapReduce framework provides a way to work around this complication. In this recipe, we will write a MapReduce application that allows us to sort the values going to a reducer using an approach known as the **secondary sort**. Instead of holding all of the distinct IPs in memory, we can keep track of the last IP we saw while processing the values in the reducer, and we can maintain a counter to calculate distinct IPs.

How it works...

We first created a class named `CompositeKey`. This class extends the Hadoop `WritableComparable` interface so that we can use the `CompositeKey` class just like any normal Hadoop `WritableComparable` interface (for example, `Text` and `IntWritable`). The `CompositeKey` class holds two `Text` objects. The first `Text` object is used to partition and group the key-value pairs emitted from the mapper. The second `Text` object is used to perform the secondary sort.

Next, we wrote a mapper class to emit the key-value pair `CompositeKey` (which consists of page and IP) as the key, and IP as the value. In addition, we wrote a reducer class that receives a `CompositeKey` object and a sorted list of IPs. The distinct IP count is calculated by incrementing a counter whenever we see an IP that does not equal a previously seen IP.

After writing the mapper and reducer classes, we created three classes to partition, group, and sort the data. The `CompositeKeyPartitioner` class is responsible for partitioning the data emitted from the mapper. In this recipe, we want all of the same pages to go to the same partition. Therefore, we calculate the partition location based only on the first field of the `CompositeKey` class.

Next, we created a `GroupComparator` class that uses the same logic as `CompositeKeyPartitioner`. We want all of the same page keys grouped together for processing by a reducer. Therefore, the group comparator only inspects the first member of the `CompositeKey` class for comparison.

Finally, we created the `SortComparator` class. This class is responsible for sorting all of the values that are sent to the reducer. As you can see from the method signature, `compare(WritableComparable a, WritableComparable b)`, we only receive the keys that are sent to each reducer, which is why we needed to include the IP with each and every key the mapper emitted. The `SortComparator` class compares both the first and second members of the `CompositeKey` class to ensure that the values a reducer receives are sorted.