CHAPTER 9

Solving Problems with Hadoop

On the Hadoop Core mailing list, a user was wondering about the way to handle a specific style of range query with MapReduce. The application had a search space and incoming search requests. In this chapter, we'll look at a similar setup, as follows:

- The search space dataset has the key *range begin, range end* and the value *search space data*. For simplicity's sake, let's assume that ranges in the search space do not overlap.
- The search request dataset has the key *value* and the value *search request data*.
- The result set for a value that is between *range begin and range end* has the key *value* and the value *search request data, search space data*.

How do you solve this problem with a traditional MapReduce application? That's the focus of this chapter.

There are a couple of overall design goals, and the weights of the different factors will vary by installation and by job. In today's environment, there is an intense pressure to get processes up quickly and evolve them. Given agile business practices and tight budgets, rapid evolution becomes the norm. This practice means that there will be little design time, and the application will be modified, possibly by multiple teams, over a medium to long period of time.

Design Goals

Our overall goal is to have a job that runs reliably and fast. To achieve reliability, we aim for simple code, and implement monitoring to be informed when the algorithms being used are no longer suitable for the scale or patterns of data.

Given that this application is going to evolve rapidly, and eventually be modified, perhaps by different people, each piece of code needs to be simple and clear. This is in direct opposition to the requirement that the map and reduce methods be treated as the deeply nested inner loops that they are and carefully optimized.

The data is expected to be real-world, dirty, and to change over time. Wherever possible, the application must handle malformed records in a graceful manner and report on the malformed rate.

To achieve good performance, the job must minimize underuse of the hardware, by managing how the data is split, partitioned, and compressed and by tuning the number of tasks run per node. To avoid having the network speed become the limiting factor, the transform

design must attempt to minimize the number of times the data is written to HDFS and the volume of data passed to reduce tasks.

This example will have as input two datasets. One dataset—the search requests—is composed of Apache log file data in common log format. The other dataset—the search space—is composed of IP address ranges and a network name. The output of the job, shown in Table 9-1, will be a modified common log format with the IP address, the network range, and the network name, in place of the original IP address, for those search requests for which a network was found.

■ **Note** The last two octets of all IP addresses in the log files have been randomized. The command used was `perl -ne 'chomp; if ( $_ !~ /^(\d+\.\d+)\.(\d+\.\d+)\s(.*)$/) { print STDERR "Failure on $_\n"; next; } print $1, ".", int(rand(256)), ".", int(rand(256)), " ", $3, "\n";' < access_log.txt > access_log_randomized.txt---; mv –f access_log_randomized.txt access_log.txt`.

Table 9-1. *Sample Job Output*

Log IP	Net Range Begin	Net Range End	Net Name	Log Record
12.229.91.253	12.0.0.0	12.255.255.255	ATT	- - [18/Nov/2008:14:08:59 -0800] "GET / HTTP/1.1" 404 293 "-" "Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET CLR 1.0.3705; .NET CLR 1.1.4322; Media Center PC 4.0; .NET CLR 2.0.50727; .NET CLR 3.0.4506.2152; .NET CLR 3.5.30729)"
58.68.24.75	58.68.0.0	58.68.127.255	DWL NET	- - [20/Nov/2008:00:47:53 -0800] "GET / HTTP/1.1" 302 315 "-" "Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1; Trident/4.0; .NET CLR 2.0.50727; .NET CLR 3.0.04506.30; .NET CLR 3.0.04506.648; .NET CLR 3.5.21022)"
59.92.27.113	59.88.0.0	59.99.255.255	BSNLNET	- - [19/Nov/2008:12:38:42 -0800] "GET / HTTP/1.1" 302 309 "-" "Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.9.0.4) Gecko/2008102920 Firefox/3.0.4"
60.50.230.247	60.48.0.0	60.54.255.255	XDSLSTREAMYX	- - [16/Nov/2008:21:06:18 -0800] "GET / HTTP/1.1" 404 293 "-" "Mozilla/4.0 (compatible; MSIE 6.0; Windows 98)"
61.135.0.92	61.135.0.0	61.135.255.255	CNCGROUP BJ	- - [22/Nov/2008:07:07:16 -0800] "GET / HTTP/1.1" 302 309 "-" "Baiduspider+(+http://www.baidu.com/search/spider.htm)"

■ **Note** Thanks to Apress for the log file samples.

Design 1: Brute-Force MapReduce

The brute-force MapReduce pattern is generally the quickest to get going and the simplest to manage. The downside is that these jobs quickly become bound by the network speed and the sorting speed for the cluster.

In a brute-force MapReduce, the only time you have ordered data is in the reduce step. This forces all of the data to flow through to the reduce task. There is also the additional complexity that you have multiple record types, which need to be distinguished at reduce time.

The overriding constraint here is ensuring that any given search request record finds all records that it is in range of in the search space.

A Single Reduce Task

If a single reduce task is used, all search request records are guaranteed to be in the same partition as their respective search space records. Table 9-2 defines the comparator behavior for the three cases the comparator will encounter.

Table 9-2. *Comparator Cases*

Type of Item 1	Comparison Region of Item 1	Type of Item 2	Comparison Region of Item 2	Equality Condition
Search request	Entire key	Search request	Entire key	Key_1 equal to key_2
Search request	Entire key	Search space	Begin range	Search request key equal to begin range
Search space	Begin range, end range	Search space	Begin range, end range	Begin $range_1$ equal to begin $range_2$ and end $range_1$ equal to end $range_2$

The input plan for the reduce method is to receive individual records and to manage the join behavior by maintaining memory about previous records. This adds complexity to the reduce method and increases the risk of out-of-memory conditions. To enable the framework to do the aggregation would require having redundant data in the records; the end range would need to be in the value of the search space records. This requirement is driven by the fact that the `OutputCompartor` object receives only the key. A simplification that results from this decision is that, in the first pass, using `Text` is acceptable for the key and value, as the records may be distinguished lexically. In a future step, as a performance optimization, we will implement a key class that provides a `WritableComparator` that handles our keys at the byte level rather than at the object level. Using the byte-level comparator for a complex key opens the door to the key format and the comparator getting out of sync, introducing the possibility of errors.

■ **Note** Having `Text` objects for the key and value greatly simplifies the initial debugging of the jobs, as the data can be readily examined by eye.

Key Contents and Comparators

For simplicity in this pass, we are going to use the same object, `Text`, for the keys for both datasets, and `Text` for the values. To do this, a simple encoding must be defined that allows the origination dataset to be determined easily from the text of the key. If there is a way to do this without needing to write a custom comparator, the job can be up and running very quickly. For the stock comparator to work, the keys must lexically compare an order that the reduce method understands and can process with minimal complexity.

In this application, a key is an IPv4 address for a search request record, and a pair of IPv4 addresses for the search space records. If all IP addresses are encoded as a zero-padded, fixed-length hexadecimal string, the primary lexical ordering issue is addressed. This leaves a single issue: lexically, keys for the search requests will sort before a search space key that has a begin range value equal to the key of the search request. In the best of all possible worlds, search request keys would appear in the sorted output, after the search space key that opens the range for the request.

The search space key may simply be the begin range and end range values, with a separator character. There are many simple tools for splitting strings based on a separator character. This has the advantage that if a lexically larger character is used as a suffix for the search request keys, the search request keys will sort after the search space key that defines the relevant range. An example is shown in Table 9-3.

Table 9-3. *Expected Sorting Order for Search Space and Search Request Keys Using a Separator Character for the Space Range and a Suffix Character for the Request Keys*

Address	Key Type	Encoded Key
`220.255.7.213:220.255.7.217`	Space	`dcff07d5:dcff07d9`
`220.255.7.217`	Search	`dcff07d9;`

This can be quickly tested by running a small sample dataset through a streaming job to verify that the data compares the way we expect. A test dataset will be prepared from an Apache log file, with the Perl command in Listing 9-1. The code in this section takes the first field of the access log, commonly an IP address, and converts it to an unsigned integer, which is then printed as an eight-character-wide hexadecimal number, with a semicolon (`;`), as a suffix. A fake range is generated by printing that original value, without the semicolon, with a number ten higher, with a colon (`:`) separating them. A few lines of the output are included. Notice that the output ordering is exactly the reverse of what our application needs.

■ **Note** Listing 9-1 is structured to run from within the Cygwin environment, in the `examples` directory, on a Windows installation. Adjust the paths and file names as needed for your local installation.

Listing 9-1. *Generating a Sample Set of IP Addresses and Ranges from an Apache Log File*

```
perl -MSocket -ne 'chomp; my @parts = split(/\s/, $_); my $ip = $parts[0]; ➥
my $val = inet_aton($ip); my $num = unpack( "N", $val); printf "%08x;\n", ➥
 $num; printf ➥"%08x:%08x\n", $num, $num+10;' < access_log.txt | ➥
 sort -u > 'C:\tmp\dataset'
head /cygdrive/c/tmp/dataset
```

```
0c065a60:0c065a6a
0c065a60;
0c067f4d:0c067f57
0c067f4d;
0c06e9c7:0c06e9d1
0c06e9c7;
0c06ef2d:0c06ef37
0c06ef2d;
0c1e1694:0c1e169e
0c1e1694;
```

In the command shown in Listing 9-1, a dataset was prepared with converted IP addresses from an Apache log file. Listing 9-2 runs a streaming job to see how the records will actually be sorted by the default comparator. As you can see from the Listing 9-2 output, the search space records (`0c065a60:0c065a6a`) sort before a search request record that starts with the same address (`0c065a60;`). Success—this is the pattern we were hoping to achieve.

■ **Note** Cygwin users are likely to always have an error message that starts with `cygpath: cannot create short name of c:\Documents and Settings\Jason\My Documents\Hadoop Source\hadoop-0.19\logs`. This error may be ignored. Listing 9-2 is structured to run from the Hadoop installation directory.

Listing 9-2. *Running a Streaming Job to Verify Comparator Ordering*

```
bin/hadoop jar contrib/streaming/hadoop-0.19-streaming.jar –D ➥
mapred.job.tracker=local -D fs.default.name=file:/// -input 'C:\tmp\dataset' ➥
-output 'C:\tmp\sorted' -mapper 'C:\cygwin\bin\cat' -reducer 'C:\cygwin\bin\cat' ➥
 -numReduceTasks 1;
```

```
jvm.JvmMetrics: Initializing JVM Metrics with processName=JobTracker, sessionId=
mapred.JobClient: No job jar file set.  User classes may not be found. ➥
See JobConf(Class) or JobConf#setJar(String).
mapred.FileInputFormat: Total input paths to process : 1
streaming.StreamJob: getLocalDirs(): [/tmp/hadoop-Jason/mapred/local]
streaming.StreamJob: Running job: job_local_0001
streaming.StreamJob: Job running in-process (local Hadoop)
mapred.FileInputFormat: Total input paths to process : 1
mapred.MapTask: numReduceTasks: 1
mapred.MapTask: io.sort.mb = 1
mapred.MapTask: data buffer = 796928/996160
mapred.MapTask: record buffer = 2620/3276
streaming.PipeMapRed: PipeMapRed exec [C:\cygwin\bin\cat]
streaming.PipeMapRed: R/W/S=1/0/0 in:NA [rec/s] out:NA [rec/s]
streaming.PipeMapRed: R/W/S=10/0/0 in:NA [rec/s] out:NA [rec/s]
streaming.PipeMapRed: R/W/S=100/0/0 in:NA [rec/s] out:NA [rec/s]
streaming.PipeMapRed: mapRedFinished
streaming.PipeMapRed: Records R/W=616/1
streaming.PipeMapRed: MRErrorThread done
streaming.PipeMapRed: MROutputThread done
mapred.MapTask: Starting flush of map output
mapred.MapTask: Finished spill 0
mapred.TaskRunner: Task:attempt_local_0001_m_000000_0 is done. ➥
 And is in the process of commiting
mapred.LocalJobRunner: Records R/W=616/1
mapred.TaskRunner: Task 'attempt_local_0001_m_000000_0' done.
streaming.PipeMapRed: PipeMapRed exec [C:\cygwin\bin\cat]
mapred.Merger: Merging 1 sorted segments
mapred.Merger: Down to the last merge-pass, with 1 segments ➥
 left of total size: 10474 bytes
streaming.PipeMapRed: R/W/S=1/0/0 in:NA [rec/s] out:NA [rec/s]
streaming.PipeMapRed: R/W/S=10/0/0 in:NA [rec/s] out:NA [rec/s]
streaming.PipeMapRed: R/W/S=100/0/0 in:NA [rec/s] out:NA [rec/s]
streaming.PipeMapRed: mapRedFinished
streaming.PipeMapRed: MRErrorThread done
streaming.PipeMapRed: Records R/W=616/1
streaming.PipeMapRed: MROutputThread done
mapred.TaskRunner: Task:attempt_local_0001_r_000000_0 is done. ➥
 And is in the process of commiting
mapred.LocalJobRunner:
mapred.TaskRunner: Task attempt_local_0001_r_000000_0 is allowed to commit now
mapred.FileOutputCommitter: Saved output of task ➥
 'attempt_local_0001_r_000000_0' to file:/C:/tmp/sorted
mapred.LocalJobRunner: Records R/W=616/1 > reduce
mapred.TaskRunner: Task 'attempt_local_0001_r_000000_0' done.
```

```
streaming.StreamJob:  map 100%  reduce 100%
streaming.StreamJob: Job complete: job_local_0001
streaming.StreamJob: Output: C:\tmp\sorted
```

```
head /cygdrive/c/tmp/sorted/part-00000
```

```
0c065a60:0c065a6a
0c065a60;
0c067f4d:0c067f57
0c067f4d;
0c06e9c7:0c06e9d1
0c06e9c7;
0c06ef2d:0c06ef37
0c06ef2d;
0c1e1694:0c1e169e
0c1e1694;
```

A Helper Class for Keys

Key management is critical for this job, and to help avoid introducing errors later in the application life cycle, a helper class for keys will be provided. The initial version needs to be able to validate, pack, and unpack keys to and from the `Text` objects.

TASK-SPECIFIC CONFIGURATION PARAMETERS

The Hadoop framework creates a runtime environment for the tasks of the job. In the TaskTracker's local working area, the path set defined by the configuration key, `mapred.local.dir`, a directory tree is built for the job, which contains the unpacked `DistributedCache` items, a file `job.xml` that contains the job configuration, a shared directory for all tasks of the job, and a working directory for the task to be run. An instance of the configuration date is created, and the per-task information modified by adding per-task parameters and adjusting the paths of configuration parameters that have been unpacked into the job or task working areas. The bulk of this localization process is handled by `TaskTracker.localizeJob`. The following parameters are added or modified for a task as of Hadoop 0.19.0:

- `job.local.dir`: The directory that will be used as root of the local file system space allocated for this job. `JobConf.getJobLocalDir()` returns this directory. All tasks of this job running on a TaskTracker node will share this directory. A Java system property of the same name is also set.
- `mapred.local.dir`: The root of the local file system space for this TaskTracker node.
- `map.input.file`: For the map task, the input file name, if the input split has a file name.
- `map.input.start`: For the map task, the starting offset in the `map.input.file`.

- `map.input.length`: The amount of data to read from `map.input.file`, starting from `map.input.start`.
- `mapred.tip.id`: The task ID for this task. All task attempts for this task will have the same value for this key.
- `mapred.task.id`: The task ID for this attempt of this task. The framework will make multiple attempts to complete a task. This value for this key holds the ID of the current attempt instance. In Hadoop 0.19, the value stored under this key is very similar to `mapred.tip.id`, except that it will have a prefix of `attempt_`. This is unique per task run.
- `mapred.task.is.map`: Set to `true` if this is a map task.
- `mapred.task.partition`: The partition number for this task, if known. For a map task, this is the ordinal number of the task. For a reduce task, it is both the ordinal number of the reduce task and the result of `Partitioner.getPartition( K,V, numPartitions)`, which will be identical for all key/value pairs passed to this reduce task.
- `mapred.job.id`: The ID of the job that this task is being run on behalf of.
- `mapred.work.output.dir`: The task-specific directory that output files will be created in by default. `FileOutput.getWorkOutputPath(JobConf conf)` provides this value.
- `mapred.map.tasks`: In the reduce task, the actual number of map tasks that succeeded.
- `hadoop.net.static.resolutions`: Any hostname/IP address mappings that will override the normal lookup results.
- `task.memory.mgmt.enabled`: Set to `true` if the TaskTracker is enforcing memory utilization limits.

In our example, four classes are associated with key handling:

- An interface, `KeyHelper<K>`
- An abstract class, `AbstractKeyHelper<K>`
- An implementation, `TextKeyHelperWithSeparators`, for `Text`-based keys
- A unit test, `TestTextKeyHelperWithSeparators`, to verify the expected behavior

These classes provide a way to extract the IP address from a key, shown in Listing 9-3, and to pack IP addresses into a key, shown in Listing 9-4. Two configuration parameters are available: `examples.ch9.search.suffix.char`, which defines the character to be used as a suffix when encoding a search request IP address, and `examples.ch9.range.separator.char`, which defines the character to be used to separate a pair of IP addresses in a search space key. These parameters have default values of semicolon (`;`) and colon (`:`), respectively. They may be any pair of characters, as long as the range separator character sorts first.

Listing 9-3. *boolean TextKeyHelperWithSeparators.getFromRaw(Text raw)*

```
public boolean getFromRaw(Text raw) {
isValid = false;
hasEndRange = false;
String rawText = raw.toString();
if (rawText.length()==(addressLen+1)
    && rawText.charAt(addressLen)==searchRequestSuffix) {
    String searchRequest = rawText.substring(0, addressLen);
    beginRangeOrKey = Long.valueOf(searchRequest,16);
} else if (rawText.length()==(addressLen*2+1)
            && rawText.charAt(addressLen)==rangeSeparator) {

    String beginRange = rawText.substring( 0, addressLen);
    beginRangeOrKey = Long.valueOf(beginRange,16);
    endRange = Long.valueOf(rawText.substring(addressLen+1,addressLen*2+1),16);

    /** Verify that the begin range is less or equal to the end */
    if (beginRangeOrKey>endRange) {
        if (LOG.isDebugEnabled()) {
 LOG.debug("key [" + rawText + "] length " + rawText.length() + " begin > end "
  + beginRangeOrKey + " " + endRange);
       }
        return false;
    }
    hasEndRange = true;
} else {
    if (LOG.isDebugEnabled()) {
 LOG.debug("key [" + rawText + "] length " + rawText.length() + " invalid");
    }
    /** length is wrong, or the separator or suffix is wrong. */
    return false;
}
isValid = true;
return true;
}
```

In Listing 9-3, the key is converted to a `String` and examined to see if it is one of the two patterns that are accepted. All IP addresses will be encoded as eight hexadecimal digits. If the key is a search request, there will be one IP address and a trailing `searchRequestSuffix` character only, forcing the string to be only nine characters in length. If the key is a search space item, there will be two IP addresses, with a `rangeSeparator` character between them only, forcing the string to be seventeen characters in length. The IP addresses are converted into long values via `Long.valueOf(address,16)`. The `String.substring` method is used for extracting the actual IP address data from the raw string.

If a valid search request or search space definition is found, the helper object is marked valid, `isValid = true`, and `beginRangeOrKey` is set to the first IP address found. If the key contained a search space request, `hasEndRange` is set to `true` and `endRange` is set to the second IP address.

The `setToRaw` method, in Listing 9-4, is used to create and store a value in a key object that correctly encodes either a search request or a search space. If the helper object is not valid, nothing is done, and no indication of this is made. This will open the door to missing errors. Changing this behavior requires rearchitecting the application to provide a visible trace of this error; logging it is not likely to be sufficient. A `StringBuilder` and `Formatter` are `ThreadLocal` instance variables, making this class thread-safe. This is done as a small efficiency and a protection against the day when the helper is used in a multithreaded map task.

Listing 9-4. *void TextKeyHelperWithSeparators.setToRaw(Text raw)*

```
public void setToRaw(Text raw) {
if (!isValid) {
    return;
}
Formatter fmt = keyFormatter.get();
fmt.flush();
StringBuilder sb = keyBuilder.get();
sb.setLength(0);

if(hasEndRange) {
    fmt.format( "%08x%c%08x", beginRangeOrKey, rangeSeparator, endRange );
} else {
    fmt.format( "%08x%c", beginRangeOrKey, searchRequestSuffix);
}
fmt.flush();
raw.set(sb.toString());
}
```

■ **Note** It is reasonable to assume that anything written to the log by a task will never have been seen by a human being unless something is visibly wrong with the job. The volume of data is just too large.

The Mapper

With the plan for the comparator handled, it is time to design the mapper. This mapper must handle two tasks:

- For the search requests, the mapper must accept Apache log files and extract a key from the line in the key format, passing the rest of the line as the value.
- The search space items will be stored as straight text, with a tab (`\t`) separating the range from the data. The mapper may distinguish between the two records either from the input file name or by the length of the key.

As a demonstration of using chain mapping, our mapper is going to run a chain to process the incoming values. The first element in the chain will take action only if the incoming record does not look like a search request or search space key, but instead looks like an Apache log file record. This mapper will transform the record into a search request. The next map in the chain will perform validity checking on the keys.

■ **Note** In the next version, the example will use `org.apache.hadoop.mapred.lib.MultipleInputs`, and have the search space dataset be in a `SequenceFile`. For simplicity of debugging, this version uses text records only.

This example has two mapper classes: `ApacheLogTransformMapper` and `KeyValidatingMapper`. Listing 9-5 shows the mapper preamble in `ApacheLogTransformMapper`. This demonstrates our standard practice of having a counter, named `TOTAL INPUT`. This provides a clear indication of how the job is going. The `helper` object parses a string that is either a search request or a search space, returning `true` if the key was recognized. In this preamble, if the `helper` can parse the key, it is just passed forward. As a general rule, we log per-key data only at level debug, as the logging volume will be very large.

Listing 9-5. *The Mapper Preamble, ApacheLogTransformMapper.java*

```
reporter.incrCounter("ApacheLogTransformMapper", "TOTAL INPUT", 1);

if (helper.getFromRaw(key)) {
    reporter.incrCounter("ApacheLogTransformMapper", "ALREADY PREPARED KEYS", 1);
    if (LOG.isDebugEnabled())
        LOG.debug("complete key passed forward untouched [" + key + "]");
    }
    output.collect( key, value );
    return;
}
```

In Listing 9-6, the key was not recognized as a prepared key and is assumed to be an Apache log line. If the input separator for the `TextInputFormat` happens to be a single space:

```
conf.get("key.value.separator.in.input.line", "\t");
```

then the key is assumed to be the IP address. The test `keyValueSeparator.length()==1 && keyValueSeparator.charAt(0)==' '` verifies this.

■ **Note** If the input format happens to not be `KeyValueTextInputFormat`, the configuration key changes in `KeyValueTextInputFormat`, or the default value changes, this code will fail silently.

The method `parseAddressIntoKey()` will take the IP address and convert it into our established format and pass the new key and the value to the output.

Lisitng 9-6. *The Mapper Log Line Processing Part 1, ApacheLogTransformMapper.java*

```
if (LOG.isDebugEnabled()) { LOG.debug("Working on [" + key + "]"); }
reporter.incrCounter("ApacheLogTransformMapper", "LOG LINES", 1);
String logLine = key.toString();
String keyValueSeparator = conf.get("key.value.separator.in.input.line", "\t");
String ipAddress;
/** The IP address in the standard log file entry is the first field,
  * with a trailing space to separate it from the next field.
 */
if (keyValueSeparator.length()==1 && keyValueSeparator.charAt(0)==' ') {
    /** The key and value are already parsed out. */
    ipAddress = logLine;
    if (parseAddressIntoKey(ipAddress, outputKey, reporter)) {
        reporter.incrCounter("ApacheLogTransformMapper", "VALID LOG LINES", 1);
        if (LOG.isDebugEnabled()) {
          LOG.debug( "Key transforms from [" + key + "] to [" + outputKey + "]");
        }
        output.collect(outputKey, value);
        return;
    }
```

In Listing 9-7, the default case of a raw log line is handled. This code does make the assumption that the `keyValueSeparator` computed in Listing 9-6, is correct. A complete line is assembled in `sb`, and then parsed. The IP address is assumed to be the first text in the line and to be terminated by an ASCII space character. This code accepts only IPv4 addresses in the format of four dot-separated octets. Once the correct key and new value are produced, they are output. The use of chain mapping actually reduces the efficiency of the task, but it is nice to have a demonstration.

Listing 9-7. *The Mapper Log Line Processing Part 2, ApacheLogTransformMapper.java*

```
/** For paranoia sake, reassemble the log line and split it ourselves
  * on the first space. */
sb.setLength(0);
sb.append(logLine);
sb.append( keyValueSeparator);
sb.append( value.toString());
logLine = sb.toString();

int indexOfSpace = logLine.indexOf(' ');
if (indexOfSpace< 7 || indexOfSpace > 15) {
                        /** xxx.xxx.xxx.xxx = 15 chars, 1.1.1.1 = 7 chars */
```

```
    if (LOG.isDebugEnabled()) {
 LOG.debug("Log line does not start with an ip address [" + logLine + "]" );
    }
    reporter.incrCounter("ApacheLogTransformMapper", "BAD LOG LINES", 1);
    return;
}

ipAddress = logLine.substring(0,indexOfSpace);
logLine = logLine.substring(indexOfSpace+1);

if (parseAddressIntoKey(ipAddress, outputKey, reporter)) {
    outputValue.set( logLine );
    reporter.incrCounter("ApacheLogTransformMapper", "VALID LOG LINES", 1);
    if (LOG.isDebugEnabled()) {
       LOG.debug( "Key transforms from [" + key + "] to [" + outputKey + "]");
    }
    output.collect( outputKey, outputValue );
    return;
}
```

The KeyValidatingMapper, shown in Listing 9-8, just checks the keys for the proper shape—that they are valid IPv4 addresses—and swaps the search space begin and end range values if begin is greater than end. At this point, all keys are assumed to be valid, and this map verifies that. Several counters are kept to help with sort and long-term monitoring of the job.

Listing 9-8. *KeyValidatingMapper.java*

```
if (!helper.getFromRaw(key)) {
    reporter.incrCounter("KeyvalidatingMapper", "INVALID KEYS", 1);
    return;
}
if (helper.isSearchRequest()) {
    reporter.incrCounter("KeyValidatingMapper", "TOTAL SEARCH", 1);

    if (helper.getSearchRequest()<0 || helper.getSearchRequest()>4294967296L) {
        if (LOG.isDebugEnabled()) {
           LOG.debug("Search Key out of range [" + key + "]");
        }
        reporter.incrCounter("KeyValidatingMapper", "SEARCH OUT OF RANGE", 1);
        return;
    }
    output.collect( key, value);
    return;

} else {
    reporter.incrCounter("KeyValidatingMapper", "TOTAL SPACE", 1);
```

```
    if (helper.getBeginRange()<0||helper.getBeginRange()>4294967296L) {
        reporter.incrCounter("KeyValidatingMapper", "SPACE BEGIN OUT OF RANGE", 1);
        return;
    }
    if (helper.getEndRange()<0||helper.getEndRange()>4294967296L) {
        reporter.incrCounter("KeyValidatingMapper", "SPACE END OUT OF RANGE", 1);
        return;
    }

    /** Verify the ordering of the search space item. */
    if (helper.getBeginRange()<=helper.getEndRange()) {
        output.collect(key, value);
        return;

    } else {
        reporter.incrCounter("KeyValidatingMapper", "SPACE OUT OF ORDER", 1);
        long tmp = helper.getBeginRange();
        helper.setBeginRange(helper.getEndRange());
        helper.setEndRange(tmp);
        helper.setToRaw(outputKey);
        output.collect( outputKey, value);
        return;
    }
}
```

The Combiner

The combiner is often one of the more complex pieces of a MapReduce job, and it's usually given the least thought. What is the correct behavior for encountering duplicate keys in the map output? For simple aggregation jobs, this is straightforward. In our case, we have two different types of keys, and what to do for a duplicate in either case is unclear.

The first proposal would be to use a `TextArrayWritable`, and just keep all of values. This doesn't provide much of a space saving, compared to just not running a combiner. The second proposal would be to discard duplicates. Neither choice is appealing. A combiner should provide either a significant reduction in I/O volume or a significant reduction in resource use for the reduce phase. Neither of the preceding proposals can provide those. If a custom comparator were written, a combiner might make sense.

In the type of MapReduce application we are working on here, a combiner that suppresses duplicate key/value pairs could be helpful. In our constructed example, we know there are no exact duplicates.

The Reducer

Each reducer task will need to receive a stream of key values, where the range statements will be first in the sorting order. This forces the reducer class to maintain state information about which ranges have been seen, and the value of those ranges. This prior range information is bounded, and ranges may be flushed when the end range value is less than the current input

key. As an added bonus, the reduce task is also run as a chain, with a postprocessing map that converts the encoded key formats back into dot-separated octet format. The actual reduce task is performed by `ReducerForStandardComparator.java`, shown in Listing 9-9.

Lisitng 9-9. *ReducerForStandardComparator.java*

```
reporter.incrCounter("ReducerForStandardComparator", "TOTAL KEYS", 1);
if (!helper.getFromRaw(key)) {
    reporter.incrCounter("ReducerForStandardComparator", "BAD KEYS", 1);
    return;
}

if (helper.isSearchSpace()) {
    reporter.incrCounter("ReducerForStandardComparator", "SPACE KEYS", 1);

    /** For simplicity, put all of the values in. */
    while (values.hasNext()) {
        final Text value = values.next();
        reporter.incrCounter("ReducerForStandardComparator", "SPACE VALUES", 1);
        activeRanges.activate( reporter, "ReducerForStandardComparator", ➥
           helper.getBeginRange(), helper.getEndRange(), value.toString());
    }
    return;
}

if (helper.isSearchRequest()) {
    /** First, lets prune the activeRanges. */
    final long searchRequest = helper.getSearchRequest();
    activeRanges.deactivate(searchRequest);

    /** Because the ranges are removed when their end is less than end,
     * and because keys are always sorted after the beginning of a range
     * all active ranges are now 'hits' for this search request.
     */
    int max = activeRanges.size();
    while (values.hasNext()) {
        final Text value = values.next();
        for (int i = 0; i < max; i++) {
            ActiveRanges.Range<String> hit = activeRanges.get(i);
            handleHit( key, output, reporter, value, hit);
        }
    }
}
```

In Listing 9-9, our standard counters are in use. At this point, any invalid key is an indication that something has gone very wrong—data corruption at some level, given the level of verification performed on the keys in earlier steps.

Our algorithm is very simple. We keep a queue of networks, ordered by the network end-of-address range. If the current key is a search request and the current key is larger than the end of a network's address range, the network is removed from the active queue. The call `activeRanges.deactivate(searchRequest)` clears any networks from the `activeRanges` queue that can no longer be matched. If the current key is a search space key, it is added to the set of active ranges, via the following:

```
activeRanges.activate( reporter, "ReducerForStandardComparator",
                       helper.getBeginRange(), helper.getEndRange(),
                       value.toString());
```

At this point, each network in `activeRanges` is a match. A network's end range is guaranteed to be larger than the search request key, and due to our comparator's ordering of the keys, the network begin range must be less than or equal to our search request.

For each log line, `while (values.hasNext())`, an output record is generated for each network, `for (int i = 0; i < max; i++) {`, via the call to `handleHit( key, output, reporter, value, hit)`, which is shown in Listing 9-10.

Listing 9-10. *ReducerForStandardComparator.handleHit*

```
StringBuilder sb = new StringBuilder();
Formatter fmt = new Formatter(sb);

protected void handleHit(Text key,
        OutputCollector<Text, Text> output, Reporter reporter,
              Text value, Range<String> hit) throws IOException {

    /** For this version we leave the end alone. */
    sb.setLength(0);
    fmt.format( "%s\t%s", hit.getValue(), value.toString());
    fmt.flush();
    outputValue.set(sb.toString());
    sb.setLength(0);
    /** Lose the suffix */
    fmt.format("%8.8s\t%08x\t%08x", key.toString(),
       hit.getBegin(), hit.getEnd()); fmt.flush();
    outputKey.set(sb.toString());
    output.collect( outputKey, outputValue );

}
```

In Listing 9-10, a `StringBuilder` and `Formatter` are built. These are used to construct the actual output key and output value. The key will be the original log record IP address, followed by the network begin and end addresses. For ease of parsing, these will be separated by an ASCII tab character. The value is simply the network name, ASCII tab, and the rest of the original log line.

The Driver

The driver, shown in Listing 9-11, builds on our base class, `utils/MainProgramShell.java`, and defines only a small number of methods. This example relies on there being only a single reduce task, as the default partitioner will cause this job to fail. In our next design iteration, we will write a custom partitioner.

All of the examples in this chapter are structured to run on small machines, so the reduce sort space has been reduced from 100MB to 10MB, using the following line:

```
conf.setInt("io.sort.mb", 10);
```

The values for `input` and `output` are set by the use of the command-line flags `--input` and `--output`, respectively. The setup follows the general rule for using the chain, and allocates `dummyConf` to use as the private configuration object for the chained map and reduce tasks. The framework serializes the contents in each call to the `ChainMapper` methods, making it safe to clear `dummyConf` and reuse it.

Listing 9-11. *The Job Setup, BruteForceMapReduceDriver.java*

```
super.customSetup(conf);
conf.setJobName("BruteForceRangeMapReduce");
conf.setNumReduceTasks(1);
conf.setInt("io.sort.mb", 10);
conf.setInputFormat( KeyValueTextInputFormat.class);
for( String input : inputs) {
    if (verbose) {
        LOG.info("Adding input path " + input);
    }
    FileInputFormat.addInputPaths(conf, input);
}
if (verbose) {
    LOG.info( "Setting output path " + output);
}
FileOutputFormat.setOutputPath(conf, new Path(output));
conf.setOutputFormat(TextOutputFormat.class);

JobConf dummyConf = new JobConf(false);
ChainMapper.addMapper(conf, ApacheLogTransformMapper.class,
         Text.class, Text.class, Text.class, Text.class, false, dummyConf);
dummyConf.clear();
ChainMapper.addMapper(conf, KeyValidatingMapper.class, Text.class, Text.class,
         Text.class, Text.class, false, dummyConf);

dummyConf.clear();
ChainReducer.setReducer(conf, ReducerForStandardComparator.class,
         Text.class, Text.class, Text.class, Text.class, false, dummyConf);
dummyConf.clear();
ChainReducer.addMapper(conf, TranslateBackToIPMapper.class,
         Text.class, Text.class, Text.class, Text.class, false, dummyConf);
```

The map and reduce methods used do not modify the passed-in key or value objects; therefore, the chaining framework is being formed to pass keys and values by reference. The second-to-last argument, `false`, in the `ChainMapper.addMapper` and `ChainMapper.setReducer` methods forces this behavior.

All of the mappers and reducers expect `Text` objects for the input key and value, and output `Text`. In an updated version of chaining, in which the key and value objects implement `WritableComparable` and `Writable`, passing `TextKeyHelperWithSeparators` objects for the key would probably be significantly more efficient.

The Pluses and Minuses of the Brute-Force Design

The biggest plus of this design is that it is simple and took about a day to put together. The biggest disadvantages are that all of the data must pass through the mapper and be sorted, and that only a single reduce task may be used. Given that the total number of networks is relatively bounded, if the incoming log records are batched in smaller sizes, this job will run reasonably well and reasonably fast. Without a custom partitioner, this job cannot be made to run with multiple reduce tasks.

Design 2: Custom Partitioner for Segmenting the Address Space

The biggest boost for the brute-force method would be to find a simple way to allow multiple reduce tasks. The standard partitioner uses the hash value of the key, modulus the number of partitions as the partition number. A simple strategy for this application might be to simply segment the IP address range. There is no guarantee that the network ranges will fall cleanly on these segments. There will need to be a mechanism to split search space keys into segment-appropriate boundaries during the job, while putting the full range in the output record. Perhaps simply modifying the format for the search space records to allow for an original range to be part of the record will address this.

■ **Note** This partitioning method is still subject to uneven distributions of the key space resulting in a subset of reduce tasks running much longer. To ameliorate this, the key space may be sampled and the partitioning table built using the sample data, in a manner similar to that done by the Hadoop terasort example.

The Simple IP Range Partitioner

The partitioner class for this example is `SimpleIPRangePartitioner`. The `getPartition()` method, shown in Listing 9-12, simply takes the IP address of a search request key or the begin range address of a search space key and returns the partition for that record.

A SCOPE REDUCTION IN THE PARTITIONER

The original design supported a configurable table to ensure that the records were partitioned approximately evenly. This required a tool to scan the records to generate a distribution map and code to load that map into the partitioner. During the process of actually writing the code, the decision was made that if that feature is needed, it may be implemented later. Instead, each partition gets an approximately even number or span of addresses out of the IPv4 space.

For a job with one reduce task, the span for partition 0 is from `0.0.0.0` to `255.255.255.255`. For a job with two reduce tasks, partition 0 would span from `0.0.0.0` to `127.255.255.255`, and partition 1 would span from `128.0.0.0` to `255.255.255`.

This left a few artifacts in the `SimpleIPRangePartitioner`. A `TreeSet` is used instead of simply maintaining an array of long values. The array of long values would be faster and would greatly reduce object churn.

Listing 9-12. *SimpleIPRangePartitioner.getPartition*

```
@Override
public int getPartition(final Text key, final Text value, final int numPartitions) {
    if (!(helper.getFromRaw(key) && helper.isValid())) {
        throw new IllegalArgumentException("key " + key +
                " cannot be parsed as a network range set");
    }
    /** The IP address that effectively defines this range. */
    final long begin;

    if (helper.isSearchRequest()) {
        begin = helper.getSearchRequest();
    } else {
        begin = helper.getBeginRange();
    }

    /** Find the bucket in ranges that is the lowest bucket
      * that is valued higher than begin.
     * That bucket's partition is the partition for this value.
     */e
    final Entry<Long, Integer> partition = ranges.higherEntry(Long.valueOf(begin));
    /** Stored as a variable for debugging ease */
    final int realPartition = partition.getValue();

assert (helper.isSearchSpace() ? partition.getKey() >= helper.getEndRange() : true)
 : String.format( "search space range end %08x exceeds partition limit %0x8",
    helper.getEndRange(), partition.getKey());

    return realPartition;
}
```

The first step is to initialize the key `helper` and to determine if the key is actually a valid search space or search request key:

```
if (!(helper.getFromRaw(key) && helper.isValid())) {
```

If the key is valid, the IP address of the search request record or the range begin address of the search space record is stored in `begin`. Once `begin` is known, it may be looked up in the table, `ranges`, that maps addresses to reduce partitions. The table is actually a `TreeMap`, and entry keys are the ending IP address of the partition. The partition number is the entry value. This data structure allows the following line to provide the entry of the partition that the key/value pair must go to:

```
partition = ranges.ceilingEntry(Long.valueOf(begin));
```

The `TreeMap` method `higherEntry` returns the element in `ranges` where the entry key is closest to `begin`, while not being less than `begin`. range end is larger than `begin`. The value of that entry is the partition number for this key/value pair.

For debugging purposes, the entry is assigned to a local variable, `partition`. The entry value could simply be returned at this point, but a little checking is done to verify that this key/value pair is a search space record, where the end of the search space is also an address that will be in this partition. No checking is made for the case where `ranges.higherValue` returns `null`, as it is assumed that the `ranges` table spans the full IPv4 address space range.

The `ranges` table is constructed in the `configure()` method, shown in Listing 9-13, as this is the first time the number of reduce tasks is known.

Listing 9-13. *SimpleIPRangePartitioner.configure*

```
public void configure(JobConf job) {

conf = job;
/** Now that we have a conf object we can initialize the
  * helper and build ranges, using the number of reduces. */
helper = new PartitionedTextKeyHelperWithSeparators(conf);

final int numPartitions = conf.getNumReduceTasks();

ranges = new TreeMap<Long,Integer>();

long rangeSpan = 4294967296L / numPartitions;

/** The partition that ends at <code>spanned</code> */
int partition = 0;
/** The end of the address space already in ranges. */
long spanned;
/** The value stored is the end of the range, the range
  * starts at the previous value + 0, or for the first value
 * at 0.
```

```
 * Note that the test is less than and not less than or equals,
 * the ranges have to end at 2^32-1 as we only have 32 bits.
 */
for (spanned= rangeSpan; spanned < 4294967296L; spanned += rangeSpan, partition++) {
    ranges.put( spanned, partition );
}
/** First address is 0, last address is 2^32 - 1, make sure we cover
  * all the way to the end of the range if
 * the 2^32/numPartitions is not an integer. The last partition may be a small */
if (spanned>4294967296L-1){
    ranges.put( 4294967296L -1, partition); /** The end range */
}

}
```

The first step is to save a copy of the `JobConf` object into `conf`, our standard practice. The key helper for this example is `PartitionedTextKeyHelperWithSeparators`. This class delegates to the `TextKeyHelperWithSeparators` class for any unrecognized input keys, and handles an extended form for search space keys that provides a way of splitting a search space key across multiple partitions and then assembling the resulting records later.

IPv4 addresses are simply unsigned 32-bit integer values, and the entire space runs from `0` through `4294967295` inclusive. Each partition will span approximately `rangeSpan` addresses, defined as `4294967295L / numPartitions`. The application uses long values to avoid issues with sign extension, as Java does not provide an unsigned integer type.

The variable `spanned` contains the ending IPv4 address of the previous partition. Each pass through the `for` loop adds `rangeSpan` to `spanned` defining the ending address of the next partition and increments the partition number:

```
for (spanned= rangeSpan; spanned < 4294967296L; spanned += rangeSpan, partition++) {
    ranges.put( spanned, partition );
}
```

`ranges.put( 4294967296L -1, partition)` stores the partition end address and partition number in `ranges`. These are currently added in order, which is not optimal for a `TreeMap`, as `TreeMaps` are stored as red-black trees and ordered insertion will result in an unbalanced tree. Casting our gaze into the future, it seems unlikely that there may be more than small hundreds of reduce tasks and a rewrite might be planned to eliminate the use of `TreeMap` and simply use an array.

Search Space Keys for Each Reduce Task That May Contain Matching Keys

The `SimpleIPPartitioner` also provides a method `spanSpaceKeys`, shown in Listing 9-14, which is not part of the partitioner interface. Here, I took a design expedience step that perhaps was not optimal given my later experience. I decided to use the `BruteForceMapReduceDriver` (Listing 9-11), and allow more than one reduce task. To achieve this, each search space record must be replicated so that any partition that could have matching requests each gets a copy of

the search space record. The concept is that an addition map will, for each incoming search space record, output a set of search space records such that each reduce partition that could receive a matching search request will receive one of the output search space records. This addition map, `RangePartitionTransformingMapper` (shown later in Listing 9-16), will be added to the mapper chain.

Listing 9-14. *SimpleIPRangePartitioner.spanSpaceKeys Preamble*

```
public int spanSpaceKeys( PartitionedTextKeyHelperWithSeparators outsideHelper,
        Text forConstructedKeys, final Text value,
        final OutputCollector<Text, Text> output, Reporter reporter)
              throws IOException {

    /** If the key isn't valid bail. */
    if (!outsideHelper.isValid()) {
        reporter.incrCounter("KeySpanning", "Invalid Keys", 1);
        throw new IllegalArgumentException("Cannot span invalid keys");
    }

    /** This could just pass the key forward quietly. */
    if (!outsideHelper.isSearchSpace()) {
        reporter.incrCounter("KeySpanning", "Not Search Space", 1);
        throw new IllegalArgumentException("Cannot span search request keys");
    }

    /** If the passed in key is a regular search space key,
      * set the extended attributes for a spanning search space key. */
    if (!outsideHelper.isHasRealRange()) {
        outsideHelper.setRealRangeBegin(outsideHelper.getBeginRange());
        outsideHelper.setRealRangeEnd(outsideHelper.getEndRange());
    }
```

The first portion of Listing 9-14 handles the setup and validation. The calling convention requires that the caller pass in an initialized key helper (`outsideHelper`) and the `value` to `output` (`OutputCollector`). The `Reporter` object (`reporter`) is used to log metrics and failures. The key helper is checked for validity (`outsideHelper.isValid()`) and that it contains a search space request (`outsideHelper.isSearchSpace()`). If either constraint check fails, an exception is thrown. The key helper class for these spanned keys has two additional fields: the actual begin and end of the search space request. The begin and end fields will now be fields for the address span of the partition for which the record is output. `outsideHelper.setRealRangeBegin(outsideHelper.getBeginRange())` and `outsideHelper.setRealRangeEnd(outsideHelper.getEndRange())` initialize the helper correctly if it is not already set up.

As a quick recap, the search space key contains an IPv4 address range, represented as a beginning and ending address. To enable multiple reduce tasks, the search space records must be available in each reduce task that could receive search requests that would match the search space record. This allows the search space requests to be mixed into the job input with the search requests. Each search space key is split into a set of search space keys, such

that each individual key contains that portion of the original range that fits within the range of addresses that will be routed to a specific reduce task. Implicit is that each partition starts with the address after the prior partition and there is no overlap in address space between partitions. Partition 0 is assumed to start at address `0`, (`0.0.0.0`), and the last partition is assumed to end at `4294967295` (`255.255.255.255`).

The block of code in Listing 9-15 is the part of the `spanSpaceKeys` method that produces the per-partition keys.

Listing 9-15. *Producing Search Space Keys for the Required Reduce Partitions*

```
NavigableMap<Long, Integer> spannedRanges =
   ranges.tailMap(outsideHelper.getRealRangeBegin(), true);

/** The loop below uses the the begin range of
  * <code>outsideHelper</code> as the start point for the next
 * output record.
 * The end range value is used as a convenience and should not be used in test.
 * The real end and real beginning are always the actual
 * begin and end of the search space request.
 */
helper.setBeginRange( outsideHelper.getRealRangeBegin());
helper.setRealRangeBegin( outsideHelper.getRealRangeBegin());
helper.setEndRange( outsideHelper.getRealRangeEnd());
helper.setRealRangeEnd( outsideHelper.getRealRangeEnd());

int count = 0;
/** The real ranges are untouched, and the begin range is moved up
  * and the end range is just set in the loop.
 * When end range <= the spanEnd no more ranges are spanned.
 * the value of getEndRange() is never valid for use in tests.
 */
if (LOG.isDebugEnabled()) {
    LOG.debug(String.format("Spanning key %x:%x %s", helper.getRealRangeBegin(),
      helper.getRealRangeEnd(),value));
}
for( Map.Entry<Long, Integer> span : spannedRanges.entrySet()) {
    final Long spanEnd = span.getKey();

    /** If the newly adjusted begin range is past the end of our key's range,
      * there will be no more keys output. so finish up */
    if (helper.getBeginRange()>helper.getRealRangeEnd()) {
        helper.isValid = false;
        break; /** Done, no more ranges spanned. We could just
                 * return count from here, but this way there is only one
                 * valid exit point */
    }
```

```
/** This should never happen. */
if (spanEnd.longValue() < helper.getBeginRange()) {
    /** at least a partial span. */
    throw new IOException( String.format(
 "Constraint failure, the partition end %d %x is less than the key begin %d %x",
     spanEnd, spanEnd, helper.getBeginRange(), helper.getBeginRange()) );
}

/** The begin value for the current portion of <code>outsideHelper</code>
  * is inside the span of this partition. We have to assume at this point
  * that it is not before the start of the partition.
 *
 * If the spanEnd >= the getRealRangeEnd, this output key is contained entirely
 * within this partition
 *
 */


/** This case indicates that the end of this partition span is past the end of
  * the real search space request.
 * This is the last key that will be output, the output key end to be the real
 * end and finish
 */
if (spanEnd.longValue()>=helper.getRealRangeEnd()) {
    /** The range of the key only extends to this partition. */
    helper.setEndRange(helper.getRealRangeEnd());
    if (LOG.isDebugEnabled()) {
     LOG.debug(String.format(">= spanEnd %x %x of %x:%x %s",
         spanEnd, helper.getRealRangeEnd(), helper.getRealRangeBegin(),
         helper.getRealRangeEnd(), value ) );
    }

} else {    // There will be at least one more output key after this one

    /** In this case, the search space real end is past the end
      * of this partition, output a record from the
     * begin that was setup on the previous run through here or the initial
     * condition and an end == to the span end
     * and continue our loop
     */
```

```
            // Has to be less than the real end range
            helper.setEndRange(spanEnd.longValue());
            if (LOG.isDebugEnabled()) {
               LOG.debug(String.format(" < spanEnd  %x %x:%x  %x:%x %s", spanEnd,
                  helper.getBeginRange(), helper.getEndRange(),
                  helper.getRealRangeBegin(), helper.getRealRangeEnd(), value ) );
            }


        }

        count++;
        helper.setToRaw(forConstructedKeys);
        output.collect(forConstructedKeys,value);
        helper.setBeginRange(helper.getEndRange()+1); // One past the last record output
        reporter.incrCounter("KeySpanning", "Partition " + span.getValue(), 1);

    }
    reporter.incrCounter("KeySpanning", "OUTPUT KEYS", count);
    return count;
```

The passed-in, parsed-input key is in `outsideHelper`, the working object is `helper`, and the actual begin and end addresses for the network are stored in the real begin (`helper.getRealRangeBegin()`) and real end (`helper.getRealRangeEnd()`) fields of `helper`.

The `helper`, a `PartitionedTextKeyHelperWithSeparators` object, holds both the actual original search space key, using the `realRangeBegin` and `realRangeEnd` fields, and the begin and end address of the range within a partition, in the `begin` and `end` fields. For each partition, the begin `helper.setBeginRange()` and end `helper.setEndRange()` will be set to the address range within that partition that this search space record will match, and the `realRangeBegin` and `realRangeEnd` fields will be untouched.

The variable `spannedRanges` is a subset of ranges that contains only partitions that have an end address larger or equal to the real begin range of the key, and equal to or less than the real end range of the key. Put simply, `spannedRanges` contains the partitions that may contain addresses that would match the passed-in search space record.

The following loop examines each of the candidate partitions in ascending order of the partition end address:

```
for( Map.Entry<Long, Integer> span : spannedRanges.entrySet()) {
```

The variable `spanEnd` contains the ending address for the current partition. It is implicit in the data structures used that `spanEnd` will be greater than or equal to `helper.getBeginRange()` (the beginning address of the portion of the key that has not yet been output to a partition is always available as `helper.getBeginRange()`).

When a per-partition key is to be output, the `helper` is set up with the correct end address for that partition. The end address will either be the last address of the partition, `spanEnd`, or

the last address of the actual range, `getRealRangeEnd()`, whichever address is least. If the end address of the output key is less than or equal to the end address of the current partition, no more keys need to be output. The begin field of helper is set to the address after the end of the previous output key, `helper.setBeginRange(spanEnd.longValue()+1`.

The core loop is run once for each potential partition that this key may need to have a record placed. The variable `count` keeps track of the number of records output, and `span` contains the information about the current partition, in particular the end address and the partition number. There are a couple checks: one to see if the partition end addresses are not in ascending order (`spanEnd.longValue() < helper.getBeginRange()`) and another to see if the key has been fully spanned across the partitions (`helper.getBeginRange()>helper.getRealRangeEnd()`).

There are two possible cases:

- The remaining portion of the key fits entirely in the current partition, `span`, `spanEnd.longValue()>=helper.getRealRangeEnd()`. The range end of the helper is set to the applicable end value in this case, `helper.setEndRange(helper.getRealRangeEnd())`.
- The key has address space that extends past the end of `span`. In this case, `helper.setEndRange(spanEnd.longValue())` is called.

The end of the loop actually builds the `Text` object with the appropriate data, `helper.setToRaw(forConstructedKeys)`, and resets begin to the address after the just output key, `helper.setBeginRange(helper.getEndRange()+1)`. Each input search space request now has a record that will be placed by the partitioner into each partition that could have search requests that match.

In `RangePartitionTransformingMapper`, shown in Listing 9-16, is a very simple `map()` method. It initializes the key helper from the passed-in key, `helper.getFromRaw(key)`, and for a valid search space key, calls the `spanSpaceKeys` method of `SimpleIPPartitioner` (search requests are just passed through as output).

Listing 9-16. *RangePartitionTransformingMapper*

```
public void map(Text key, Text value,
        OutputCollector<Text, Text> output, Reporter reporter)
        throws IOException {
    try {
        reporter.incrCounter("RangePartitionTransformingMapper", "INPUT KEYS", 1);
        if (!helper.getFromRaw(key)) {
            reporter.incrCounter("RangePartitionTransformingMapper", "", 1);
            return;
        }
        if (helper.isSearchRequest()) {
            output.collect(key, value);
            reporter.incrCounter(
                "RangePartitionTransformingMapper", "Request Keys", 1);
            return;
```

```
        }
        partitioner.spanSpaceKeys(helper, outputKey, value, output, reporter);
    } catch( Throwable e) {
        throwsIOExcepction( reporter, "RangePartitionTransformingMapper", e);
    }


}
```

The original concept was to take the search requests, feed them through the `RangePartitioningTransformingMapper` using `RangePartitionTransformingMapper` as a driver class, convert the search space records into a sorted and partitioned dataset, run another MapReduce job over the incoming search requests, and then perform a map-side join on the resulting datasets. After working with the data for a short time, I realized that the search space was so small that it wasn't worth the extra complexity or time to have an additional step for presorting the search space records. I decided to simply add this mapper as part of the mapper chain, and read the search space records as input with the search request records. The configuration changes to `BruteForceMapReduceDriver` are shown in the next section.

Helper Class for Keys Modifications

The class `PartitionedTextKeyHelperWithSeparators` will be the new `KeyHelper` and will support carrying the original key data, so that the output records can be provided with the actual network range instead of that portion of the network range that fits in this partition. A new record format needs to be designed that can carry the additional data. The key format for the search space keys has been *begin*`:`*end*, where *begin* and *end* are the first and last addresses of the network, each an eight-digit hexadecimal number. For example, `0.0.0.0` would be `00000000`, `255.255.255.255` would be `ffffffff`, and the search space key representing the entire IPv4 address space would be `00000000:ffffffff`. To allow partitioning, the search case keys must match keys in a particular partition. My first idea on how to address this was to just have four values instead of two, with the same separator between each. The full code for that version is in `com.apress.hadoopbook.examples.ch9.PartitionedTextKeyHelperWithSeparators.java`, available with the rest of the downloadable code for this book.

The code for the first design must be modified to examine a configuration parameter, `range.key.helper`, and instantiate the value as a class, defaulting to the `TextKeyHelperWithSeparators` class. Listing 9-17 provides an example of this from `ApacheLogTransformMapper`.

Listing 9-17. *Modifications to Load a Key Helper Based on the Value of range.key.helper*

```
public void configure(JobConf conf) {
    super.configure(conf);
    helper = ReflectionUtils.newInstance(conf.getClass("range.key.helper",
      TextKeyHelperWithSeparators.class,
            TextKeyHelperWithSeparators.class),conf);


}
```

The existing mapper and reducer classes are modified to instantiate their `KeyHelper` class based on a configuration property, `range.key.helper`, defaulting to `TextKeyHelperWithSeparators`. `BruteForceMapReduceDriver` is modified to set the `range.key.helper` configuration parameter value to `PartitionedTextKeyHelperWithSeparators` when the number of reduce tasks is more than one. This leaves the old behavior intact, while allowing multiple reduce tasks.

In Listing 9-18, the configuration key `range.key.helper` is set to be our partitioning class by `conf.setClass("range.key.helper", PartitionedTextKeyHelperWithSeparators.class, KeyHelper.class)`, and an additional map is placed in the chain, to span the search space keys:

```
ChainMapper.addMapper(conf, RangePartitionTransformingMapper.class, Text.class,
                      Text.class, Text.class, Text.class, false, dummyConf)
```

Listing 9-18. *Modifications to the Setup Method in BruteForceMapReduceDriver.java*

```
if (conf.getNumReduceTasks()!=1) {
    /** If more that one reduce is to be run, the spanning partitioner must be used.
      */
    conf.setClass("range.key.helper", PartitionedTextKeyHelperWithSeparators.class,
                  KeyHelper.class);

    /** Add in the map that takes incoming search space records and spans them
      * across the partitions */
    ChainMapper.addMapper(conf, RangePartitionTransformingMapper.class,
            Text.class, Text.class, Text.class, Text.class, false, dummyConf);
    dummyConf.clear();
}
```

The reducer, `ReducerForStandardComparator.java`, does not need any changes, but the `ActiveRanges` class, which provides the `hit` method, does. In Listing 9-19, we simplify it to make it aware of the `PartitionedTextKeyHelperWithSeparators` class, and in that case, to use the real begin and end ranges for a search space request, rather than the per-partition begin and end ranges. If many types of keys are used, this method will quickly become excessively complex. In this case, there is only one type of key, so we can defer that code cleanup to a future that may not come.

Listing 9-19. *Modifications to ActiveRanges.activate to Support the Partition Spanned Search Space Keys*

```
if (helper instanceof PartitionedTextKeyHelperWithSeparators) {
    begin = ((PartitionedTextKeyHelperWithSeparators)helper).getRealRangeBegin();
    end = ((PartitionedTextKeyHelperWithSeparators)helper).getRealRangeEnd();
} else {
    begin = helper.getBeginRange();
    end = helper.getEndRange();
}
```

To provide a secondary sort of the final output, we have the classes `DataJoinReduceOutput`, `DataJoinMergeMapper`, and `IPv4TextComparator`. This set of classes performs a map-side join on all of the reduce output partitions of `BruteForceMapReduceDriver`, producing a single sorted file as output. The output uses the network begin, end, and name values as secondary sort keys. These also provide an example of how to perform a merge-sort of any reduce task output efficiently using map-side joins.

Listing 9-20 shows the `DataJoinReduceOutput` method.

Listing 9-20. *DataJoinReduceOutput.java, CustomSetup*

```
ArrayList<String> tables = new ArrayList<String>();
for( String input : inputs ) {
    String []parts = input.split(":");
    if (parts.length==2) {
        Class<? extends InputFormat> candidateInputFormat =
                    conf.getClass(parts[0],null,InputFormat.class);
        if (candidateInputFormat!=null) {
            addFiles(conf,candidateInputFormat, parts[1], tables);
            continue;
        }
    }
    addFiles(conf, KeyValueTextInputFormat.class,input, tables);
}

FileOutputFormat.setOutputPath(conf, new Path(output));
conf.set("mapred.join.expr", "outer(" + StringUtils.join(tables, ",") + ")");
conf.setNumReduceTasks(0);
conf.setMapperClass(DataJoinMergeMapper.class);
conf.setOutputKeyClass(Text.class);
conf.setOutputValueClass(Text.class);
conf.setInputFormat(CompositeInputFormat.class);
//conf.setOutputKeyComparatorClass(IPv4TextComparator.class);
conf.setClass("mapred.join.keycomparator", IPv4TextComparator.class,
      WritableComparator.class);
conf.setJarByClass(DataJoinMergeMapper.class);
```

`DataJoinReduceOutput` accepts the standard command-line arguments, including the `-i path [, path, [path...]] -o output`, to set the input datasets and the output path. Unlike a traditional map-side join, where each path item in the input is a table and the matching `part-XXXXX` files of each input path are joined, each individual `part-XXXXX` file is taken as a table, and all of the `part-XXXXX` files are joined together. This causes the map-side join to perform a streaming merge-sort on all of the input data files.

The `customSetup()` method examines each input in turn. If the input string has a colon (`:`), it is split and the parts examined:

```
String[] parts = input.split(":");
```

If there are exactly two parts and the first part is a class name that implements `InputFormat`, that input format is used for loading the directory name in `parts[1]`. If there is not exactly two parts, the original input is used with `KeyValueTextInputFormat`. Basically, the input directory can be preceded by a class name and a colon, and the class will be used as the input format for loading files from that input directory.

The `addFiles` method in shown in Listing 9-21.

Listing 9-21. *DataJoinReducerOutput.addFiles*

```
Path inputPath = new Path(path);
FileSystem fs = inputPath.getFileSystem(conf);
if (!fs.exists(inputPath)) {
    System.err.println(String.format(
               "Input item %s does not exist, ignoring", path));
    return;
}
FileStatus status = fs.getFileStatus(inputPath);
if (!status.isDir()) {
    String composed = CompositeInputFormat.compose(inputFormat, path);
    if (verbose) { System.err.println( "Adding input " + composed); }
    tables.add(composed);
    return;
}
FileStatus[] statai = fs.listStatus(inputPath, new PathFilter() {
    @Override
    public boolean accept(Path path) {
        if (path.getName().matches("^part-[0-9]+$")) {
            return true;
        }
        return false;
    }
}
);
if (statai==null) {
    System.err.println(
         String.format("Input item %s does not contain any parts, ignoring", path));
    return;
}
for( FileStatus status1 : statai) {
    String composed = CompositeInputFormat.compose(inputFormat,
                           status1.getPath().toString());
    if (verbose) { System.err.println( "Adding input " + composed); }
    tables.add(composed);
}
```

This method examines `inputPath`, constructed from that passed-in `path` element. If it exists (`fs.exists(inputPath)`) and is a directory (`status.isDir()`), the method collects the `FileStatus` information for each child:

```
FileStatus[] statai = fs.listStatus(inputPath,...
```

The `PathFilter` restricts the `FileStatus` entries returned to those that satisfy the `accept()` method. In this case, the only items accepted have file names that match the regular expression `^part-[0-9]+$`, our standard reduce output file format. Rather than try to manage the map-side join table format, the following call builds the table format for the input file:

```
String composed =
  CompositeInputFormat.compose(inputFormat, status1.getPath().toString());
```

All of the individual table entries are aggregated in the `ArrayList` tables.

The actual join command is built (`"outer(" + StringUtils.join(tables, ",") + ")"`) and stored in the configuration under the key `mapred.join.expr`. This by itself will merge-sort all of the input data into a single output file. The new piece, the specialty sorting of the input records before the map method, is triggered by the following line:

```
conf.setClass("mapred.join.keycomparator", IPv4TextComparator.class,
              WritableComparator.class);
```

This tells the map-side join framework to use `IPv4TextComparator` (Listing 9-23) as the key comparator when performing the merges.

The mapper, shown in Listing 9-22, provides a secondary sort by network for the matched requests.

Listing 9-22. *DataJoinMergeMapper.java*

```
reporter.incrCounter("DataJoinReduceOutput", "Input Keys", 1);
/** The number of tables in the join. */
final int size = value.size();
/** Allocate the values array if needed. a null indicates end,
  * so one extra allocated */
if (values==null) {
    values = new Text[size+1];
    outputText = new Text[size];
    /** Make some {@link Text} items, just in case. These probably aren't needed
      * but are made only once. */
    for (int i = 0; i < size; i++ ) {
        outputText[i] = new Text();
    }
}
/** For each table, check to see if it has a value for the key.
 * If it does, store it in values, possibly converting it to a text object by
 * calling {@link Text#set(String)} with the
 * with the string conversion.
```

```
 */
/** The current index to store into values. */
int valuesIndex = 0;
for (int i = 0; i < size; i++) {
    if (value.has(i)) {
        Writable outputValue = value.get(i);
        if (outputValue instanceof Text) {
            values[valuesIndex] = (Text) outputValue;
        } else {
            /** Force a text conversion to simplfy life later. */
            outputText[valuesIndex].set(outputValue.toString());
            values[valuesIndex] = outputText[valuesIndex];
        }
        valuesIndex++;
        reporter.incrCounter("DataJoinReduceOutput", "Output Keys", 1);
    }
}
values[valuesIndex] = null;
if (valuesIndex>1) {
    /** If only one, no reason to bother sorting. */
    Arrays.sort( values, 0, valuesIndex, comparator );
}
for ( int i = 0; i < valuesIndex; i++ ) {
    if( LOG.isDebugEnabled()) {
       LOG.debug( String.format( "Output of %d of %d, %s %s",
                   i, size, key, values[i]));
    }
    output.collect( key, values[i] );
}
```

Each table is checked for a value (`value.has(i)`) and each table value (`Writable outputValue = value.get(i)`) accumulated in the `values` array. Just as a safety check, the values are converted to `Text` objects when needed (`outputText[valuesIndex].set(outputValue.toString())`), and the converted value stored (`values[valuesIndex] = outputText[valuesIndex]`).

If more than one table has a value for this key, the accumulated table values are sorted via `Arrays.sort( values, 0, valuesIndex, comparator )`, using the comparator `TabbedNetRangeComparator` (shown later in Listing 9-24). Once any required sorting is completed, the records are output (`output.collect( key, values[i] )`).

The actual input and output will be detailed in `HADOOP_CLASSPATH=/tmp/commons-lang-2.4.jar hadoop jar /tmp/hadoopprobook.jar com.apress.hadoopbook.examples.ch9.DataJoinReduceOutput -libjars /tmp/hadoopprobook.jar,/tmp/commons-lang-2.4.jar -jt cloud9:8021 -fs hdfs://cloud9:8020  -v -del -i range_join -o merged_range_join`.

The `IPv4TextComparator`, shown in Listing 9-23, provides a binary comparator that handles keys that are IPv4 addresses in the standard dotted-octet format, such as 192.168.0.1. It attempts to operate at the byte level and to minimize object allocation. This class is used in the

map-side join to force the correct ordering of the input keys, as the lexical ordering is not what is expected.

Listing 9-23. *IPv4TextComparator.java*

```
public IPv4TextComparator()
{
    super(Text.class);
}
/** Compare the serialized form of two text objects containing IPv4 addresses
 * of the form 0.0.0.0 through 255.255.255.255.
 * @see org.apache.hadoop.io.RawComparator#compare(byte[], int, int, byte[],➥
 int, int)
 */
@Override
public int compare(byte[] b1, int s1, int l1, byte[] b2, int s2, int l2) {
     long a1 = unpack( b1, s1, l1 );
     long a2 = unpack( b2, s2, l2 );
     if (a1<a2) {
         return -1;
     }
     if (a1>a2) {
         return 1;
     }
     return 0;
}

/** Given a byte buffer that contains a standard decimal dotted octet IPv4 address
 * (ie: 0.0.0.0 through 255.255.255.255), as a byte stream, return the long value
 * of the ip address
 *
 * @param buf The byte buffer containing the bytes.
 * @param s The start address in <code>buf</code>.
 * @param l The length of data in <code>buf</code> to use.
 * @return the numeric value of the address 0 -> 2^32, or -1 for parse errors.
 */
public static long unpack( final byte []buf, int s, int l) {
    long result = 0;
    long part = 0;
    l += s;
    for( ; s < l; s++ ) {
        byte b = buf[s];
        switch(b) {
        case '.':
            result <<= 8;
            result += part;
```

```
            part = 0;
            continue;
        case '0': case '1': case '2': case '3': case '4':
        case '5': case '6': case '7': case '8': case '9':
            part *= 10;
            part += Character.getNumericValue((int)b);
            continue;
        default:
            return -1;
        }
    }
    result <<= 8;
    result += part;
    return result;
}

/* (non-Javadoc)
 * @see org.apache.hadoop.io.WritableComparator#compare ➥
(org.apache.hadoop.io.WritableComparable, org.apache.hadoop.io.WritableComparable)
 */
@Override
public int compare(WritableComparable a, WritableComparable b) {
    // TODO Auto-generated method stub
    if (a instanceof Text && b instanceof Text) {
        return compare((Text)a, (Text)b);
    }
    return super.compare(a, b);
}

/** Compare to text objects that are IPv4 addresses in dotted octet notation.
 * @see org.apache.hadoop.io.RawComparator#compare(Object, Object)
 */
@Override
public int compare(final Object a, final Object b) {
    if (a instanceof Text && b instanceof Text) {
        return compare((Text)a, (Text)b);
    }
    return super.compare(a, b);
}
```

The comparator in Listing 9-24 expects input lines of the form:

```
IP tab IP tab Network Name tab other data
```

It will do a primary sort using the first IP address, secondary on the second IP address, and tertiary on the network name. If at any point there is a parse failure, the element that the parse failed on is considered greater. The parsing is deferred as long as possible in the hopes that it

won't be needed. This code tries very hard to work at the byte level and not convert items back into strings.

Listing 9-24. *DataJoinMergeMapper.TabbedNetRangeComparator*

```
public static class TabbedNetRangeComparator implements Comparator<Text> {

    /** The comparator from the {@link Text} class, used for comparing
      * the network names. */
    Text.Comparator comparator = new Text.Comparator();


    /** This expects and requires the value to be IPv4TABIPv4TaBnetworkTABline.
     * the the comparison order is addr1, add2, network
     *
     * @param a Text value 1
     * @param b Text value 2
     * @return -1 1 or 0 less, greater or equal, the first item with
     * a parse failure is considered greater.
     */
    @Override
    public int compare( Text a, Text b ) {
        if( LOG.isDebugEnabled()) {
           LOG.debug( String.format("Comparing %s and %s", a, b));
        }

        /** Do the basic check on <code>a</code>, see if we find the first bit. */
        final byte[] ab = a.getBytes();
        final int al = a.getLength();
        final int at1 = findTab( ab, 0, al );
        if (at1==-1) {
            if( LOG.isDebugEnabled()) {
               LOG.debug(String.format("a %s failed to find first tab", a));
            }
            return 1;
        }

        /** Do the basic check on <code>b</code>, see if we find the first bit. */
        final byte[] bb = b.getBytes();
        final int bl = b.getLength();
        final int bt1 = findTab( bb, 0, bl );
        if (bt1==-1) {
            if( LOG.isDebugEnabled()) {
               LOG.debug(String.format("b %s failed to find first tab", b));
            }
            return -1;
        }
```

```
/** Get the first ip address from <code>a</code>. */
final long aip1 = IPv4TextComparator.unpack( ab, 0, at1 );
if (aip1==-1) {
    if( LOG.isDebugEnabled()) {
       LOG.debug(String.format("a %s failed to unpack %s",
         a, new String( ab, 0, at1)));
    }
    return 1;
}

/** Get the first ip address from <code>b</code>. */
final long bip1 = IPv4TextComparator.unpack( bb, 0, bt1 );
if (bip1==-1) {
    if( LOG.isDebugEnabled()) {
       LOG.debug(String.format("b %s failed to unpack %s", b,
         new String( bb, 0, bt1)));
    }
    return -1;
}
if( LOG.isDebugEnabled()) {
   LOG.debug(String.format("a %x b%x", aip1, bip1));
}

/** Do the ip address comparison on the first IP,
  * if they are different, this routine is done.
 * Since we have longs and the result is int, a simple
 * subtraction may not work as the result may not be an int.
 */
if (aip1<bip1) {
    return -1;
}
if (aip1>bip1) {
    return 1;
}


/** Check the second IP address in <code>a</code> and <code>b</code> */

final int at2 = findTab( ab, at1+1, al);
if (at2==-1) {
    if( LOG.isDebugEnabled()) {
       LOG.debug(String.format("a %s failed to find second tab", a));
    }
    return 1;
}
```

```
final long aip2 = IPv4TextComparator.unpack( ab, at1+1, at2 );
if (aip2==-1) {
    if( LOG.isDebugEnabled()) {
       LOG.debug(String.format("a %s failed to unpack %s", a,
         new String( ab, at1+1, at2)));
    }
    return 1;
}

final int bt2 = findTab( bb, bt1+1, bl);
if (bt2==-1) {
    if( LOG.isDebugEnabled()) {
       LOG.debug(String.format("b %s failed to find second tab", b));
    }
    return -1;
}
final long bip2 = IPv4TextComparator.unpack( bb, bt1+1, bt2 );
if (bip2==-1) {
    if( LOG.isDebugEnabled()) {
       LOG.debug(String.format("b %s failed to unpack %s", b,
         new String( bb, bt1+1, bt2)));
    }
    return -1;
}
if (aip2<bip2) {
    return -1;
}
if (aip2>bip2) {
    return 1;
}

/** At this point both pairs of IP addresses are the same.
  * Pass the network names off to Text, which knows how to compare
  * utf-8 bytes. */
final int at3 = findTab( ab, at2+1, al);
if (at3==-1) {
    if( LOG.isDebugEnabled()) {
       LOG.debug(String.format("a %s failed to find third tab", a));
    }
    return 1;
}

final int bt3 = findTab( bb, bt2+1, al);
if (bt3==-1) {
    if( LOG.isDebugEnabled()) {
       LOG.debug(String.format("b %s failed to find second tab", b));
    }
```

```
            return -1;
        }
        if( LOG.isDebugEnabled()) {
            LOG.debug(String.format("a %s b %s",
              new String( ab, at2+1, at3), new String( bb, bt2+1, bt3)));
        }
        return comparator.compare( ab, at2+1, at3, bb, bt2+1, bt3 );

    }

    @Override
    public boolean equals(Object o) {
        if (o==null) {
            return false;
        }
        if (o==this) {
            return true;
        }
        if (o instanceof TabbedNetRangeComparator) {
            return true;
        }
        return false;
    }

}
```

Listing 9-25 shows the commands used to generate the output. These commands use the machine `cloud9` on port 8021 for JobTracker services and `cloud9` port 8020 for HDFS services. Your local installation will be different.

Listing 9-25. *The Commands Used to Generate the Output*

```
hadoop jar /tmp/hadoopprobook.jar ➥
com.apress.hadoopbook.examples.ch9.BruteForceMapReduceDriver -jt cloud9:8021 ➥
 -fs hdfs://cloud9:8020 -libjars /tmp/hadoopprobook.jar ➥
 -D mapred.reduce.tasks=10 -v --deleteOutput --input searchspace.txt ➥
 access_log.txt -o range_join
HADOOP_CLASSPATH=/tmp/commons-lang-2.4.jar hadoop jar /tmp/hadoopprobook.jar ➥
 com.apress.hadoopbook.examples.ch9.DataJoinReduceOutput -libjars ➥
 /tmp/hadoopprobook.jar,/tmp/commons-lang-2.4.jar -jt cloud9:8021 ➥
 -fs hdfs://cloud9:8020  -v -del -i range_join -o merged_range_join
```

The first command runs the `BruteForceMapReduceDriver`, passing in the JAR file included with the book examples, and specifies that ten reduce tasks are to be run:

```
-D mapred.reduce.tasks=10
```

Most of our later examples accept the arguments `-v -deleteOutput`, enabling verbose logging and causing the job output directory to be deleted if the directory exists. The two input

files are a file of network ranges with names, `searchspace.txt`, shown in Listing 9-26, and some Apress.com access log data, `access_log.txt`, shown in Listing 9-27. The first output directory is `range_join`, which will be the input directory of the next command. The second line runs the command `DataJoinReduceOutput` to take the ten partition files and produce a single file that is sorted in IP address order, with secondary sorts on the network begin and end addresses and the network name. The actual output is listed in Table 9-4.

Listing 9-26. *searchspace.txt, Search Space Network Ranges*

```
72810800:72810fff   InTech Online
747d8000:747d8fff   HANANET INFRA
74480000:744bffff   HATHWAY NET
76000000:760fffff   OCN
77ea0000:77eaffff   SINGTELMOBILE
0c000000:0cffffff   ATT
79f00000:79f7ffff   TATACOMM IN
79fec000:79fec1ff   KIDC INFRA SERVERROOM DAUM
79080000:790fffff   CHINANET GD
796150c0:796150cf   BAYAN_REDMAP AP
7aa42000:7aa43fff   ABTS TN DSL 9111 chn
7aa70000:7aa77fff   ABTS KK DSL 9102 blr
7aa98000:7aa9bfff   ABTS AP DSL 9112 hyd
7aea0000:7aeaffff   CHINANET ZJ HZ
7b644000:7b647fff   MAXNET NZ
7c720000:7c73ffff   CHINANET SN
7c512a00:7c512aff   CMTSBDG IM2 HFC ID
7d11ab00:7d11abff   BTNL CHN DSL
80d20000:80d2ffff   PURDUE CCNET
836b0000:836bffff   MICROSOFT
```

Listing 9-27. *First 20 access_log.txt Lines, with the lines truncated for clarity*

```
116.125.47.43 - - [15/Nov/2008:22:07:47 -0800]...
116.125.162.223 - - [15/Nov/2008:22:07:47 -0800]...
193.238.120.192 - - [15/Nov/2008:22:23:13 -0800]...
193.238.192.10 - - [15/Nov/2008:22:23:13 -0800]...
193.238.186.77 - - [15/Nov/2008:22:23:14 -0800]...
193.238.83.101 - - [15/Nov/2008:22:23:14 -0800]...
193.47.137.43 - - [15/Nov/2008:22:25:34 -0800]...
193.47.58.78 - - [15/Nov/2008:22:25:34 -0800]...
193.252.4.14 - - [15/Nov/2008:22:56:05 -0800]...
193.252.144.172 - - [15/Nov/2008:22:56:05 -0800]...
208.80.221.10 - - [15/Nov/2008:23:07:05 -0800]...
208.80.179.99 - - [15/Nov/2008:23:07:05 -0800]...
208.80.168.223 - - [15/Nov/2008:23:14:49 -0800]...
208.80.57.128 - - [15/Nov/2008:23:14:49 -0800]...
66.233.41.9 - - [15/Nov/2008:23:29:13 -0800]...
```

```
66.233.254.37 - - [15/Nov/2008:23:29:13 -0800]...
66.233.78.158 - - [15/Nov/2008:23:29:14 -0800]...
66.233.212.119 - - [15/Nov/2008:23:29:14 -0800]...
66.233.242.121 - - [15/Nov/2008:23:29:17 -0800]...
66.233.220.139 - - [15/Nov/2008:23:29:17 -0800]...
```

Table 9-4. *The First 20 Job Output Lines*

Log IP	Network Start	Network End	Network Name	Log Line
12.6.90.96	12.0.0.0	12.255.255.255	ATT	- - [19/Nov/2008:17:01:18 -0800] "GET / HTTP/1.1" 404 293 "-" "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; InfoPath.1; .NET CLR 1.1.4322; .NET CLR 2.0.50727; .NET CLR 3.0.04506.30)"
12.6.127.77	12.0.0.0	12.255.255.255	ATT	- - [19/Nov/2008:17:03:00 -0800] "GET / HTTP/1.1" 404 293 "http://www.dtsearch.com/CS_Apress_SuperIndex.html" "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; InfoPath.1; .NET CLR 1.1.4322; .NET CLR 2.0.50727; .NET CLR 3.0.04506.30)"
12.6.233.199	12.0.0.0	12.255.255.255	ATT	- - [19/Nov/2008:17:03:00 -0800] "GET / HTTP/1.1" 404 293 "http://www.dtsearch.com/CS_Apress_SuperIndex.html" "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; InfoPath.1; .NET CLR 1.1.4322; .NET CLR 2.0.50727; .NET CLR 3.0.04506.30)"
12.6.239.45	12.0.0.0	12.255.255.255	ATT	- - [19/Nov/2008:17:01:18 -0800] "GET / HTTP/1.1" 404 293 "-" "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; InfoPath.1; .NET CLR 1.1.4322; .NET CLR 2.0.50727; .NET CLR 3.0.04506.30)"
12.30.22.148	12.0.0.0	12.255.255.255	ATT	- - [19/Nov/2008:10:28:55 -0800] "GET /favicon.ico HTTP/1.0" 404 304 "-" "Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.9.0.4) Gecko/2008102920 Firefox/3.0.4"
12.30.31.111	12.0.0.0	12.255.255.255	ATT	- - [19/Nov/2008:10:28:55 -0800] "GET /favicon.ico HTTP/1.0" 404 304 "-" "Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.9.0.4) Gecko/2008102920 Firefox/3.0.4"
12.30.136.50	12.0.0.0	12.255.255.255	ATT	- - [19/Nov/2008:10:28:55 -0800] "GET / HTTP/1.0" 404 293 "-" "Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.9.0.4) Gecko/2008102920 Firefox/3.0.4"
12.30.180.46	12.0.0.0	12.255.255.255	ATT	- - [19/Nov/2008:10:28:55 -0800] "GET / HTTP/1.0" 404 293 "-" "Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.9.0.4) Gecko/2008102920 Firefox/3.0.4"

Log IP	Network Start	Network End	Network Name	Log Line
12.69.76.145	12.0.0.0	12.255.255.255	ATT	- - [18/Nov/2008:13:07:55 -0800] "GET / HTTP/1.1" 404 293 "-" "Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET CLR 1.1.4322; .NET CLR 2.0.50727; InfoPath.1; .NET CLR 3.0.04506.30; .NET CLR 3.0.04506.648)"
12.69.85.24	12.0.0.0	12.255.255.255	ATT	- - [18/Nov/2008:13:07:30 -0800] "GET / HTTP/1.1" 404 293 "-" "Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET CLR 1.1.4322; .NET CLR 2.0.50727; InfoPath.1; .NET CLR 3.0.04506.30; .NET CLR 3.0.04506.648)"
12.69.130.223	12.0.0.0	12.255.255.255	ATT	- - [18/Nov/2008:13:07:30 -0800] "GET / HTTP/1.1" 404 293 "-" "Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET CLR 1.1.4322; .NET CLR 2.0.50727; InfoPath.1; .NET CLR 3.0.04506.30; .NET CLR 3.0.04506.648)"
12.69.229.167	12.0.0.0	12.255.255.255	ATT	- - [18/Nov/2008:13:07:55 -0800] "GET / HTTP/1.1" 404 293 "-" "Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET CLR 1.1.4322; .NET CLR 2.0.50727; InfoPath.1; .NET CLR 3.0.04506.30; .NET CLR 3.0.04506.648)"
12.167.105.82	12.0.0.0	12.255.255.255	ATT	- - [18/Nov/2008:08:21:28 -0800] "GET / HTTP/1.1" 404 293 "-" "Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 6.0; WOW64; SLCC1; .NET CLR 2.0.50727; .NET CLR 3.0.04506; .NET CLR 3.5.21022)"
12.167.179.22	12.0.0.0	12.255.255.255	ATT	- - [18/Nov/2008:08:21:28 -0800] "GET / HTTP/1.1" 404 293 "-" "Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 6.0; WOW64; SLCC1; .NET CLR 2.0.50727; .NET CLR 3.0.04506; .NET CLR 3.5.21022)"
12.216.40.118	12.0.0.0	12.255.255.255	ATT	- - [17/Nov/2008:11:37:59 -0800] "GET /book/errataSubmit.html?bID=10187 HTTP/1.1" 404 311 "-" "Opera/7.23 (Windows 98; U) [en]"
12.216.48.187	12.0.0.0	12.255.255.255	ATT	- - [17/Nov/2008:11:37:59 -0800] "GET /book/errataSubmit.html?bID=10187 HTTP/1.1" 404 311 "-" "Opera/7.23 (Windows 98; U) [en]"
12.229.67.237	12.0.0.0	12.255.255.255	ATT	- - [18/Nov/2008:14:08:59 -0800] "GET / HTTP/1.1" 404 293 "-" "Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET CLR 1.0.3705; .NET CLR 1.1.4322; Media Center PC 4.0; .NET CLR 2.0.50727; .NET CLR 3.0.4506.2152; .NET CLR 3.5.30729)"

Continued

Table 9-4. *Continued*

Log IP	Network Start	Network End	Network Name	Log Line
12.229.91.253	12.0.0.0	12.255.255.255	ATT	- - [18/Nov/2008:14:08:59 -0800] "GET / HTTP/1.1" 404 293 "-" "Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET CLR 1.0.3705; .NET CLR 1.1.4322; Media Center PC 4.0; .NET CLR 2.0.50727; .NET CLR 3.0.4506.2152; .NET CLR 3.5.30729)"
58.68.24.75	58.68.0.0	58.68.127.255	DWL NET	- - [20/Nov/2008:00:47:53 -0800] "GET / HTTP/1.1" 302 315 "-" "Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1; Trident/4.0; .NET CLR 2.0.50727; .NET CLR 3.0.04506.30; .NET CLR 3.0.04506.648; .NET CLR 3.5.21022)"
59.92.23.105	59.88.0.0	59.99.255.255	BSNLNET	- - [19/Nov/2008:12:38:42 -0800] "GET / HTTP/1.1" 302 309 "-" "Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.9.0.4) Gecko/2008102920 Firefox/3.0.4"

Design 3: Future Possibilities

Two possibilities come to mind for this sample MapReduce job:

An indexed map file of search requests in the reduce task: For each search request key, the `configure()` method will open the relevant search space map file—either the full map file for the entire search space or a partitioned file—where the partition contains the networks that keys in this reduce task partition could match. The `MapFile.getClosest()` method would be used to find search space records that could match.

Map-side join of the presorted search requests and a presorted search space: This method requires presorting the search request records and the search space records, and then using the map-side join techniques discussed in Chapter 8 and the classes for working with the IP address described in this chapter.

Both require that the search space records be presorted. Also, in both cases, the search space records can either be partitioned as the search request records are partitioned, or the entire search space be present in each task, in Google Bigtable style (see `http://labs.google.com/papers/bigtable.html`).

There are trade-offs between prepartitioning versus full replicas. The partitioned case reduces the data volume that must be scanned. Even with indexes, the amount of data that needs to be fetched from disk will be smaller in the partitioned case. The downsides are that search space needs to be repartitioned if the number of reduce tasks for the search requests is changed, and there is additional (though small) code complexity to ensure that the correct search space map file is opened in each search request reduce task.

Both techniques lose the data being local for at least the search space records, and neither seem worth the bother at present, as it is not clear that there would be any performance gain.

They also require the search request records to be sorted, and the search space is expected to be relatively small.

Summary

This chapter has walked you through the design and implementation of a nontrivial real-world Hadoop application. In the process, you have seen a number of design decisions made that become invalid as understanding arrives. The design and development process was deliberately oriented to provide initial functionality quickly so that this understanding could arrive sooner, rather than after a large and costly development cycle.

A number of the advanced features, such as chaining and map-side joins, were used in the application, and a partitioner and several comparators were written.

The tight coupling between the custom partitioner and the comparator allowed the application to perform range-based matching very efficiently using MapReduce techniques.

The techniques that you have learned will allow you to efficiently and effectively tackle very complex problems that do not appear to fit the MapReduce framework, but in fact are ideally suited for MapReduce.

Particularly in the rapidly evolving environment of today, you will never have time to build the perfect application—just an application that works for yesterday's goals. Someone else will come along later and modify the application until it meets the new goals. Be kind to that person by leaving comments, testing, and keeping it simple. The person doing those future modifications may be you!