1

4 *Everyday best practices*

Imagine you are asked to do design a compliance solution for the enterprise. You know that this involved reading millions of files found on corporate computers. With this task, many questions come to mind. Like how to deal with millions of files, how do you make sure that your solutions scales to tens and hundreds of millions, how do you debug the application and prove that it works correctly, and so on. Most of these questions already have answers and in this chapter we're going to look at how to solve the scenarios that you may face daily while designing systems for Big Data processing.

This chapter is roughly divided into two parts: design and development, and performance improvements. In the first part, we discuss how to get it right the first time - which considerations should go into the design. Then, once you have it working and it is gaining popularity, we teach you how to refactor and improve the performance further.

4.1 Design and development

When you are designing for Hadoop and other big data applications, it's best to think about performance and the volume of data from the beginning. If we don't think about performance and size when working with big data, we're likely to run into scalability problems pretty fast, and may have to re-design it. Worse yet, this may happen when the site or application are already under load and you have to maintain it while improving it.

In this section, we are going to focus on the issues we might face when designing and developing for big data applications, like Hadoop, and learn how to solve them. Let's start with the problem of dealing with many small files.

4.1.1 Dealing with many small files

Imagine that you are asked to create a compliance solutions. You need to open every file on a hard drive and analyze its content. Or, it may be an eDiscovery or a search application - they are all similar in this respect.

Imagine that in your application you have to process millions, or even billions, of small files. There are quite a few situations when this can occur.

Take a break, look at your hard drive, and see how many files you have there. Chances are that your computer has between a few hundred thousand to a few million files. Processing them all in a meaningful way will take hours. Add to this that you may need to do this processing for a few computers, or for hundreds of computers within your company. If your goal is be compliance, security, or eDiscovery, the millions of files occur natively, and you cannot change this.

You may try to design an "appliance" type of solution, but you will soon outgrow your hardware. Then you may try to coordinate the working of many computers for this task, but you will have to deal with task organization, computer and network failures, and so on. Then, in the words of the Bard, "Haply I think on thee," that is, Hadoop, and you are right - but things are not as obvious as you think, so read on.

Alternatively, you may be the creator and the original of your own predicament. You may be saving user sessions as small files, and now you want to process those millions of accumulated valuable piees of information. Or, your software, such as voice messaging software, may be creating those files in great numbers. Here the files are created as artifacts, but either because they are already there, or because you don't want to change the other software pieces that you are using, you are stuck dealing with them. Ideally, you should have changed your architecture to foresee that, but that's a task for another life. Again you think of Hadoop, and again you are right, but things are not as simple as they seem to be. So what's the problem? To use a borrowed terminology, you outgrow your file allocation table. This problem and the general approaches to its solution is illustrated in the figure below. [1]

Footnote 1 This figure is a tribute to Ikai Lan (http://ikaisays.com/) who with his doodles explained a BigTable performance improvement and started a tradition of illustrating computer books with cartoon art, see http://ikaisays.com/2011/01/25/app-engine-datastore-tip-monotonically-increasing-values-are-bad/. He was consequently declared to be the most famous Google doodler.

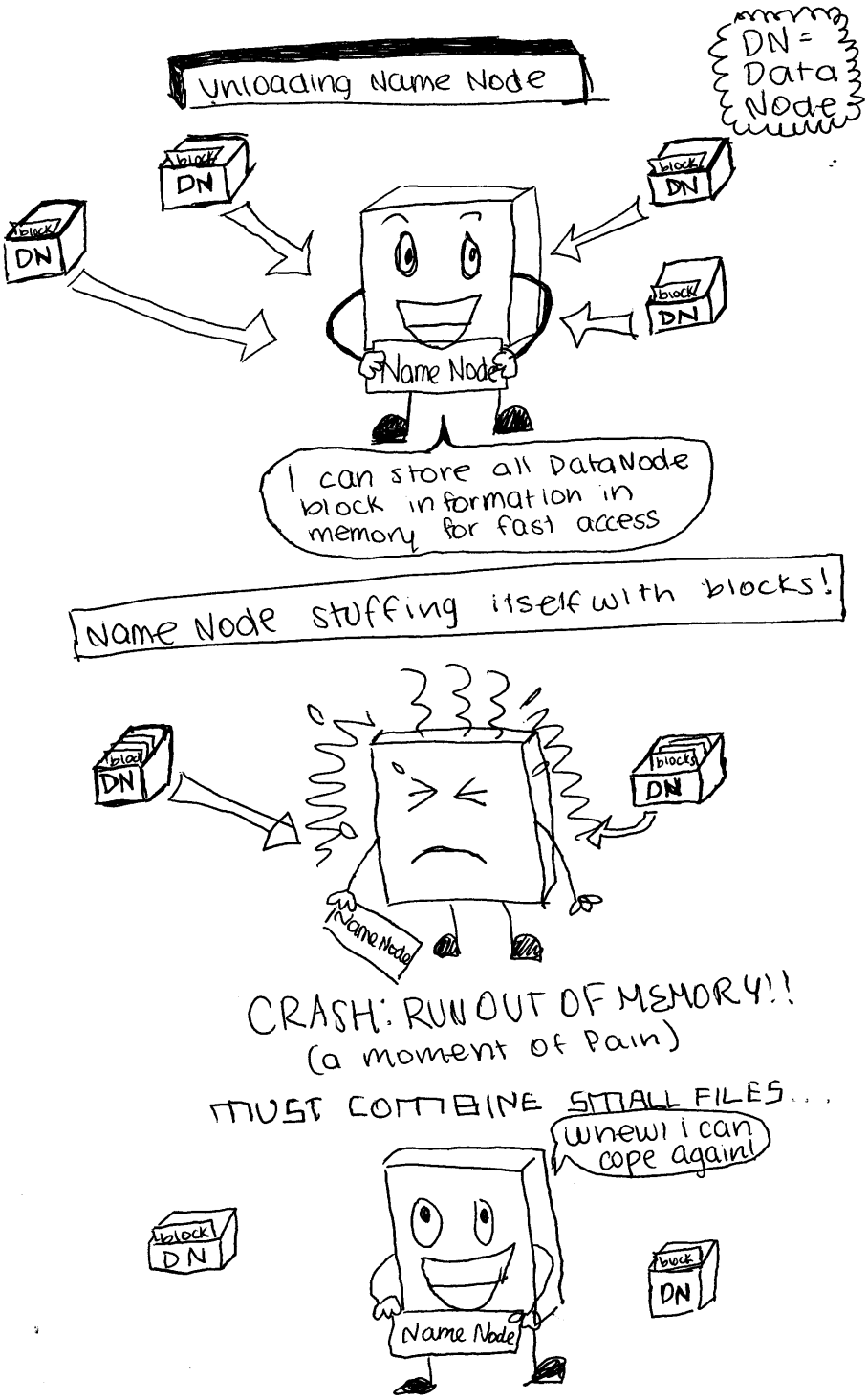

Figure 4.1 Helping the NameNode to copy with millions of small files

Technique 1 - packaging many small files in a zip archive

Information about files is stored by Hadoop in two types of nodes, NameNode

and DataNode. There are many DataNodes, but only one NameNode. Moreover, the NameNode stores the information about files in memory. Therefore, with too many files, you run the risk of overwhelming the NameNode and make it run out of memory.

PROBLEM

How to process millions of small input files

SOLUTION

To help the NameNode cope with the large number of files, we can package them in an archive, such as tar or zip, and make Hadoop read the files from the archives.

Hadoop Distributed File System, or HDFS, has two types of nodes. The first is the NameNode, which stores the information about the blocks found on DataNodes, and it stores them in RAM. The DataNodes store the blocks on disk. Your application files are stored in the blocks, and the blocks then are registered on the NameNode and are stored on DataNodes. This organization of things follows the Google File System, or GFS, and it provides two major benefits. The HDFS is hugely scalable, because you can add DataNodes at will. It is also redundant without the use of RAID, because each block is replicated on multiple DataNodes. Furthermore, you control the replication factor.

If you need to process a large number of small files, you are running into a problem. A small files is one significantly smaller than the default block size, which is 64MB. If your files are much smaller, your use of blocks becomes very inefficient. Even worse, you put a huge load on the NameNode. Since it stores the block information in memory, then a million small files may already lead to a problem, and a billion is not feasible on today's hardware. There are a number of approaches for dealing with small files, discussed in this section. This first one that we will look at occurs as a practical question in eDiscovery, but it can be useful in a number of other situations.

I happen to love law and to love computers. Somewhere on the intersection of these two loves is the problem of eDiscovery. In eDiscovery, one of the feuding sides tells the other to produce all electronic documents relevant to the case. This means dealings with millions of small files. The defendant's computer is taken, and the files on his hard drive are processed. This problem begs for a Hadoop solution, and here is why.

Ever since the shift from paper discovery to eDiscovery, the latter has been getting prominence and wide coverage. What used to be truckloads of boxes of

paper became hard drives and millions of files. Also, millions of dollars. At its peak, processing of a GB of data could cost a $1,000. The price has gone down significantly by now, but the problem has been further exacerbated by the ever-growing volumes of data. Hence the need for an efficient solution, and arguably for an open source one. I have written two proprietary eDiscovery grid systems, and my third one is an open source project, called FreeEed, for Free Electronic Evidence Discovery, found on GitHub. Often eDiscovery is used as a weapon to force the other side to settle based on price of eDiscovery - which is huge - rather than on the merits of the case, and an open source project may help the smaller entity and restore justice. You are free to use the approach and the code of FreeEed of dealing with small files.

In additional to taking the load away from the NameNode, packaging your native files into an archive has a number of other advantages.

- ZipFile structure has a provision for comments. The comments are limited to 64K (this limitation is found deep inside the Java documentation), but that is sufficient for what you want to say about your files when you package them. In particular, for eDiscovery, you can record the source of the data, the custodian, and even forensics information.
- Put a limit on the file size. Some systems may have limitations on the file size. In particular, a likely candidate you would use for your processing is EC2, and its associated S3 system limits your files to 5 Gigs. You may in any case want to limit the file size, to avoid repeating the lengthy upload on failure. Packaging your files into zip archives helps to observe this limitation.
- Provides a natural division for MapReduce. We are going to assign one zip archive to one Hadoop node. By varying the size of your zip archive, you can optimize the overall throughput of your MapReduce job.

The code to package your your files in a zip archive recursively is fairly straightforward. The points of interest in it are putting in the comments and limiting the size of individual archives. The function below is the workhorse of the packaging operation, and you can get the rest of the code on GitHub.

Listing 4.1 Packaging small files into zip archives

```
void packageArchiveRecursively(File file) throws IOException {
  if (file.isFile()) {

    if (++filesCount > filesPerArchive) {
      resetZipStreams();
    }
    FileInputStream fileInputStream = new FileInputStream(file);
    BufferedInputStream bufferedInputStream = new BufferedInputStream(fileInputStrea
    ZipEntry zipEntry = new ZipEntry(file.getPath());
```

1 Here we limit the zip size by file count; you can use size instead

```
    zipEntry.setComment("Custodian, path, forensics info...");
    zipOutputStream.putNextEntry(zipEntry);
    int count;
    while ((count = bufferedInputStream.read(data, 0,
      BUFFER)) != -1) {
      zipOutputStream.write(data, 0, count);
    }
    bufferedInputStream.close();
    fileInputStream.close();
  } else if (file.isDirectory()) {

    for (File f : file.listFiles()) {
      packageArchiveRecursively(f);
    }
  }
}
```

2 Set comment to go with this file

3 Recursive call for a directory

You can notice that we have put some sample comment with the zip file. In real life, you will package more meaningful information here. For eDiscovery, there is a wealth of information available with the file, called file metadata. This includes such items as data source, date creation, file owner, potentially file version, and then you can go into data blocks, slack space, if you are so inclined.

We have limited the zip file size by the number of small files that go into it. That is an organizational decisions, and depending on your situation you may use the file size. If you want to be even more fancy, you can use not the size of the file that you are adding to the archive, but an estimate of what it will be after compression.

DISCUSSION

There is one caveat which is important not only here but for other cases of reading a text file as well. In order to tell Hadoop to give one zip file entry to one node, we write an "inventory" of files, which looks like this:

```
file000001.zip
file000002.zip
file000003.zip
and so on...
```

This inventory is placed into HDFS and is given as input file to Hadoop, telling the MapReduce job to read the file as text, with the following setting:

```
job.setInputFormatClass(TextInputFormat.class);
```

However - and here is the caveat - by default Hadoop will read 64M of text from our input file at one scoop, since it is optimized to process text files, and to do it in blocks. Therefore, we have to limit the size of a read to a small number, to make it read just one file name at a time. In Hadoop 0.20 this is accomplished with the following call,

```
configuration.setInt("mapred.linerecordreader.maxlength", 20);
```

where 20 is selected make it read exactly one file name and not two or more. In Hadoop 0.21 it is accomplished with a slightly different call,

```
configuration.setInt("mapreduce.input.linerecordreader.line.maxlength", 20);
```

To dig up things like that you need to read the Hadoop code, following the famous open source advice, "Read the code, stupid!" Otherwise, the call will work but fail to produce the desired effect.

So how do we process the zip files in a Hadoop? In the mapper, the "value" passed to us is the path to the zip file:

```
    public void map(LongWritable key, Text value, Context context)
            throws IOException, InterruptedException {
        String oneFile = value.toString();
        System.out.println("Ready to process file: " + oneFile);
        ZipFileProcessor processor = new ZipFileProcessor(oneFile, context);
        processor.process();
    }
```

We then pass this value to the ZipFileProcesor, the most important task of which is to open and process the zip file, like this

```
FileInputStream fileInputStream = new FileInputStream(zipFileName);
ZipInputStream zipInputStream = new ZipInputStream(new BufferedInputStream(fileInput
ZipEntry zipEntry;
while ((zipEntry = zipInputStream.getNextEntry()) != null) {
  try {
    processZipEntry(zipInputStream, zipEntry);
  } catch (InterruptedException e) {
    processException(
  }
}
```

```
zipInputStream.close();
```

Using the zip file archives in the way we have described above is highly customizable, but sometimes you may not want that. Rather, you might be looking for a solution with using more standard blocks, where all you want is to load a number of small files into HDFS and process them there.

Technique 2 - convert tar files into SequenceFiles

The zip-file based solution we described above is customizable, and can meet any specific need. The problem though is that you roll your own. It is non-standard, and you are guessing that you may spend the time on custom solution only to discover that it has some other limitation which may be important in your situation.

PROBLEM

How to load small files into HDFS in a standard way

SOLUTION

If you have a large number of small files, and you just need to load the into HDFS in the fastest possible way, you may use the solution described by Stuart Sierra here, http://stuartsierra.com/2008/04/24/a-million-little-files. In it, you would combine your files into tar files, and the load these tar files into HDFS's SequenceFiles. The advantage of this approach lies in the fact that you are using standard means, in fact, a utility designed and maintained by someone else, saving on design and maintenance. To load your files, you simply type

```
java -jar tar-to-seq.jar tar-file sequence-file
```

DISCUSSION

The loading of tar files into HDFS SequenceFiles can be slow. However, it can be parallelized, thus increasing the effective throughput. Still, knowing what you know now, if you are in control of creating your files, you could design your application to write the data SequenceFile, if possible, rather than writing to small files as an intermediate step.

If your small files are already in HDFS, but there are too many of them, and they are creating an undue load on your name node, then Hadoop itself provides a utility to collapse these files into Hadoop Archives. These files have the extension

of *.har, and you need a Hadoop cluster to create them, because the utility runs as a Hadoop job. Once you have that, however, running the utility can be as easy as typing

```
hadoop archive -archiveName zoo.har -p /foo/bar /outputdir
```

HDFS understands its archives, so looking up files in the Hadoop Archive can be done with the regular `ls` command, like this

```
hadoop dfs -ls har:///user/zoo/foo.har
```

Note that you have to use the `har:` protocol, but the packages files appear as if they reside on the HDFS system itself. You MapReduce job will see them the same way.

All of the above may be insufficient for your needs, and in fact it was not sufficient for Google. Imagine that you need an unlimited file storage, such as voice messaging or email/text/universal messaging box mail require. You may decide to use HDFS, because it fits your application just fine. Furthermore, you may mount it as a local file system, using one of the MountableHDFS, such as FuseDFS. This makes the overall application design real clear and simple. However, when you ask your open source message-generating software, such as OpenVoice, to save your voice messages, you discover that something does not work. It saves the files all right, but when it tries to read them, they are not present! This is because HDFS is eventually consistent, but does not guarantee the read immediately after the write. Remember that the write has to propagate through the cluster! HDFS was not designed for your app.

Technique 3 - Using HBase for many small files

Sometimes, you need to process your files at real-time or close to real-time speed. HDFS gives you a great scalable and redundant storage system, but it is not designed for real-time access. So if the essential part of your application is reading or writing those files, you may have a problem. However, you don't have to give Hadoop just yet - for there is HBase that is running on top of Hadoop and that does not suffer from the same read/write lag.

PROBLEM

The need for low-latency read-write on HDFS

SOLUTION

Use the HBase instead. That is in fact the route that Google took. As the demands placed on HDFS by various applications increased, they invented BigTable, and HBase is an open source NoSQL database inspired by BigTable. You don't have to take my word for it, but you can read the discussion of the evolution of GFS by the Google designer, Sean Quinlan, found here, http://queue.acm.org/detail.cfm?id=1594206. Since people do use HBase for such applications, its designers have paid attention to its latency and have improved it in the last two years.

DISCUSSION

Going all the way to a NoSQL database is not the only solution, and it does not have to be HBase. You can make good use of the local hard drive storage of your nodes and make them store intermediate files on local hard drives, where you have the guarantee of write and subsequent read with low latency. You can also use a different NoSQL database. There is a good case study of implementing a mail store with Cassandra, found here

```
http://ewh.ieee.org/r6/scv/computer/nfic/2009/IBM-Jun-Rao.pdf
```

A very interesting and new approach is Brisk, or, as its authors describe it, "Apache Hadoop powered by Cassandra. Recall that Cassandra is modeled after the features of Amazon's Dynamo. It has no single point of failure in the design, and all of its nodes are equal. Its goal is to provide low-latency read and write operations. Sound like a solution to our problem? But wait, it gets better. Part of the Brisk solution is CassandraFS, or distributed file system backed by the Cassandra database. All of these are open source projects, and the commercial support for them is provided by DataStax. At the time of writing, this offering was only a few months old, so at the risk of being trivial, I want to remind you of always checking the latest news. Besides, Brisk provides a way for MapReduce jobs to run with Cassandra, which is another advantage. Yet other NoSQL implementations may have a potential to provide you with the needs specific to you application.

What have we done so far? We have looked at a design pattern that essentially tells us to combine small files into larger ones. Ah, you will say, the design patterns are a familiar device in software development. They tell us how to approach some well-understood problems in general, and specific implementation

in each situation may be different. Then there are anti-patterns, and the whole discussion of the validity of the pattern approach.

To make our discussion more specific and a little different from the regular design pattern discussion, we will be calling the patterns that have to deal with Big Data and with computing grids the "grid design patterns." The term "grid patterns" was suggested by Arun C. Murthy of Yahoo, and we just embellished it a little. Incidentally, as Arun explains [2] , when we combined the small files, we got an additional benefit: we limited the number of maps in MapReduce processing. Each map process takes some time to start up. Therefore, the map tasks given to the individual nodes should be reasonably big to justify this startup time. So on short, our first grid design pattern was "combine multiple small input files". Now we can continue onto our next grid design pattern, dealing with inter-node communications.

Footnote 2 See Arun's blog entry on Hadoop Grid Patterns here, http://developer.yahoo.com/blogs/hadoop/posts/2010/08/apache_hadoop_best_practices_a/

4.1.2 Making Hadoop nodes work as a whole

Imagine that you need to impose some overarching rules on the work of your independent Hadoop nodes. For example, you may need to assign a count number to each produced result, and these count numbers must be consecutive. This does happen in eDiscovery, where each next output page must have the next "Bates number," so that after producing 00023 the system must produce 00024. You might think of allowing each reduce task to do its own counting, and then run another Hadoop job to combine and renumber the results. But that won't work if you need to name our output files by using the Bates number, such as ABC00023.tif, ABC00024.tif, and so on - because there will be too much to renumber. In short, you need a global counter. Another situation where a similar requirement might occur is if you need to count some totals between the nodes, like the total number of errors in your data, and stop the calculation after a certain overall threshold is reached.

These and similar situations call for some internode communication. Earlier we said that Hadoop runs by breaking the computation in an "embarrassingly parallel" manner, and the communications between nodes are a no-no. Well, this is not exactly true. There are a few ways how to make the nodes play together, and in this section we will be looking at them. The most important thing though is to know the trade-off of each.

Technique 4 - Implementing a global counter

Earlier we described a situation where we wanted to number output documents (pages) consecutively, regardless that they were created in different MapReduce jobs.

PROBLEM

How do you maintain a global counter for all the nodes?

SOLUTION

There are a few possible approaches to this: you can have a single reducer, or you can run an outside service to provide the count. If your requirement is a little relaxed and the precise counter is not required but an overall count is sufficient, even if somewhat delayed, you can use a Hadoop Counter. Let us look at the each of these approaches.

A simple, straightforward, but limited to approach is to use just one Reducer. It is accomplished with one code of line

```
job.setNumReduceTasks(1);
```

in your main() job setup, which then becomes something like this

```
@Override
public int run(String[] args) throws Exception {
  Configuration configuration = getConf();
  Job job = new Job(configuration);
  ...
  job.setNumReduceTasks(1);
  boolean success = job.waitForCompletion(true);
  return success ? 0 : 1;
}
```

This makes sure that there is only one reducer and that all processing goes through one instance of this class. Then you declare the counter as a class variable and use it, like this

Listing 4.2 Using global counter in the Reducer

```
public class Reduce extends Reducer<MD5Hash, MapWritable, Text, Text> {

  private int outputFileCount;                                          (1) Global counter which will be formatted as String
  private DecimalFormat UPIFormat = new DecimalFormat("00000");
  @Override
```

```
  public void reduce(MD5Hash key, Iterable<MapWritable> values, Context context)
    throws IOException, InterruptedException {
    String outputKey = key.toString();
    for (MapWritable value : values) {
      ++outputFileCount;
      String documentText = allMetadata.get(DocumentMetadataKeys.DOCUMENT_TEXT);

      String entryName = "text/" + UPIFormat.format(outputFileCount) + ".txt";
      zipFileWriter.addTextFile(entryName, documentText);
      context.write(new Text(outputKey), new Text(columnMetadata.tabSeparatedValues(
     }
  }
}
```

2 We use it to form the file name in the output zip

The code above illustrates the most important points. Some variables are used but not defined; we have sacrificed formal correctness for brevity. You can find the complete code in the chapter code examples.

What are the specific tradoffs of the single reducer approach? It can become a bottleneck. You need to put all of your time-consuming processing in the mapper code. However, this may be acceptable in your situation. If your reducer fails, you may have a problem cleaning up - at least because in our solution we are writing one output zip file. Usually, however, the probability of a single machine failing is low, and this eventuality is considered acceptable by the designers. You would have to rerun your job, but the same is true should your master node fail.

We have described the first and most simplistic approach to global number of the output results – using a single reduce. It is a good time to mention that in most situation using a single reducer is considered an anti-pattern. So is it good or is it bad? Let us look at the pros and cons, and to give justice to the anti-pattern idea, let us start with cons:

- Puts a significant load on the single node that is executing the reducer;
- A potential bottleneck;
- Bad failure recovery.

However, let us also look at the pros of the single reducer:

- Very simple. Note that one of principles of grid programming is that programmers' time is the most valuable resource. We have already met with this principle when we were discussing databases. If it works for you and none of the cons apply, why go for the more complex one?
- Avoids introducing additional components. If you decide to use an outside service, such as a database or a JavaSpaces container, your application will be significantly more complex. If you used one of the approaches we discussed above to combine processing in relatively few maps, you may be fine with one reducer.

So which approach is right? Obviously, there is no solution that is right for every situation. Knowledge of grid patterns and anti-patterns, and the reasons behind them, will lead you to correctly architected solutions.

Another approach to a global counter would be to get it from an outside service. For example, you could use a JavaSpaces container. A free license from GigaSpaces would be sufficient for your purpose. In fact, it would be an overkill, but we will assume that once you have it running, you may find other uses for it as well.

But what is a JavaSpaces container? It is another device for concurrent programming. You can look at it as a synchronized hash map running on a separate server. You put your counter in, based on a key that your nodes know about. When you need to increment it, you take it out of JavaSpace, and it becomes invisible to other machines. If they need it, they will have to wait until you return it. You increment it and put it back, and the other nodes can proceed. The operations in JavaSpaces are guaranteed to be atomic, so that one node will never see your counter in an incomplete or transitional state.

Alternatively, you could use some database, SQL or NoSQL, to supply you with the counter. You would need to make sure that the update is synchronized. If you are using a ZooKeeper - which is a Hadoop centralized service providing distributed synchronization - you could use it for your counter. Regardless of which of the approaches you choose, you would need to make sure that the counter update is synchronized and that it does not become a bottleneck.

If you requirement is somewhat relaxed and delayed propagation is acceptable, Hadoop provides for you a mechanism called Counters. For example, you may want to count the total number of words processed, or the number of relevant documents found. You create a counter with the following constructor

```
Counter(String name, String displayName, long value)
```

Another possible reason for the use of counters is to ensure that the number of output pairs produced equals the number of input pairs processed, or that the fraction of relevant documents found is within some reasonable limits.

You can define arbitrary Counters and update them in the map and/or reduce methods. These counters are then globally aggregated by the framework. However, you must keep in mind that Counters are very expensive, since the JobTracker has

to maintain every counter of every map/reduce task for the entire duration of the application. Therefore, counters are appropriate for tracking a few most important, global bits of information. The rule of thumb is that you can use somewhere between 10 to 25 global counters.

DISCUSSION

The main idea of the global counters grid design pattern is that they should be used judiciously. If we are talking about Hadoop counters, then they take resources. If we are using some outside means of synchronizing the nodes or communicating between them, we are going against the parallel processing of Hadoop. Sometimes the programmer has to do what the programmer has to do and introduce global dependencies, but he should also analyze them for the potential problems that they can create.

People at Google also saw the need for a more cohesive Hadoop operation, and they created another computing framework called Pregel. While MapReduce is used by Google for running 80% of their computations, the other 20% are handled by Pregel, which is optimized to mine relationships from graphs. In Pregels, internode communication is built into the framework. The open source implementation of Pregel is called Hama, and it is an active project in somewhat early stages. Therefore, this is not something available to you, and not only for internal use at Google. In this section we only discussed the simple approaches approaches, and we will show you Pregel in chapter 8.

4.1.3 Better Hadoop application performance with compression

Imagine you have squeezed every ounce of performance from your Hadoop cluster, or so you think. Still, regardless of however extra power the hardware has when you start out, there soon comes a time when you would wish for more. But there is no more, so you have to optimize. Compression comes to mind.

Technique 5 - improving application performance with compression

Most likely you have used compression as a computer user, but now you are going to be the designer of it. The question that we need to answer is when and where to use compression, and what kind of compression?

PROBLEM

How to design your application with compression

SOLUTION

Hadoop is ready to help, because it has compression built in.

Hadoop supports a number of compression options, Zlib, Gzip, and LZO to name a few. Additionally, you can apply compression to the intermediate results or

to the final output. Your final output may be used by itself or it may be used by the next application in line. What are the main principles of using compression?

The first question you should ask yourself, if my application CPU-bound or I/O-bound. The majority of Hadoop applications are I/O bound. If it is CPU-bound, it is usually so special that you know it. By way of example, the eDiscovery/compliance group of applications is CPU-bound. That is because in the architecture we described above each mapper is given the task of opening the file with appropriate software that understands its specific format. Then it has to create a Lucene search index, in many cases on-the-fly and in memory, to run it against search queries that express what we are looking for. In some instances they also do optical character recognition, or OCR, and in extreme cases they generate TIF or PDF images of the document pages.

But a much larger percentage of applications processes what amounts to text information, and in such situation the compression is very appropriate. In fact – and we will see this later – Twitter already stores their information compressed, so that Hadoop has to uncompress the input files.

So let's say you decide to give a try to intermediate compression. As a general guideline, text files can be compressed by the factor of four. Therefore, if you tell Hadoop to compress the intermediate maps that are passed between mappers and reducers, you may be able to compress your network traffic by the factor of four. This might result in a speed improvement by the same amount. Luckily, Hadoop understands this soft of compression without you having to change your code. To tell Hadoop to compress the intermediate results, all you do is...

TODO – specific commands to tell Hadoop to do intermediate compression

Discussion

An alternative approach to intermediate compression is to use use SequenceFiles. This does require the change in the code, because you need to tell Hadoop what your files are, in the job setup, and then writing your mapping code acorrdingly. However, the SequenceFiles are built-in into Hadoop and are already compressed.

Which compression should you use? Zlib, Gzip, and LZO are all good choices, in that they provide a reasonable compression rate with high CPU efficiency. However, here it is time to remember the rules of optimization from Scott Meyer's classic “Effective C++.” Number one, it is “don't optimize!” This rule is not a

general call against optimization, but it is a warning not to change your code in the anticipation of the required optimization. Rather, make your code work correctly first, then optimize.

In our case, this hurdle is easily overcome, because you do NOT have to change your code in order for Hadoop to compress intermediate maps. The second rule of optimization is “profile!” Again, since it is so easy to try different compression algorithms, you can just try every one of them under a realistic load. If you don't have the time to try, use LZO.

Now you might say that I go against my own rules in selecting on compression method in a crunch, but LZO has an additional advantage which you may use in other situation, and it pays to familiarize oneself with it – that is, LZO is splittable.

Technique – using splittable LZO for performance gain in MapReduce

Let's say you want to store your data compressed, and then you want to process it with MapReduce. The problem is, you will need to be able to split your input file or files into separate independent pieces, to be given over to your mappers. It is easy to do with text files, where new lines are a natural demarcation sign. It is not so easy to do with a compressed file. One of the possible answers is provided with LZO compression.

PROBLEM

How to split a compressed files into independent pieces for MapReduce processing.

SOLUTION.

This can be done with LZO compression, using a separate pre-processing pass.

Consider again how we compressed our eDiscovery/Compliance input data into zip archives, and provided a “catalog,” which listed the zip archives. This catalog served as an input file for the MapReduce job, with one line being processed by one mapper, which then resulted in it processing one archive.

Splittable LZO compression works in a similar way, only the order of actions is reversed. First, we build an LZO-compressed file, and it can be as big as you like. Then, using the fact that LZO is designed to be splittable by containing blocks, you do the first pass and find the block boundaries. The output of the first pass is an index file, which tells your mappers where to start. Then each mapper gets a reasonable-sized split that it can process.

So how do you practically do it? First, we must look more closely at what LZO is. LZO stands for Lev-Zimpel-Oberhumer, and it is a compression methods that Lev, Zimpel, and Oberhumer described in 1996 In around 2009-2010 there was

some confusion about the licensing. The original LZO code was licenced under the GPL license, while Hadoop uses the Apache 2.0 licence, and these two are not compatible. H owever, by now this has beem largely resolved by re-implementing the LZO code under a Hadoop-compatible license. As a good choice, you can use the LZO contributed by the people at Twitter, who also provided a clear and convincing example of the LZO use. Therefore, you first install the lzop libraries with this command

```
sudo apt-get install liblzo2-dev
```

Clone the Twitter github repo and build it according to the instructions in the README.

Place the hadoop-lzo-*.jar somewhere on your cluster nodes; Twitter recommends to use /usr/local/hadoop/lib Place the native hadoop-lzo binaries (which are JNI-based and used to interface with the lzo library directly) on your cluster as well; Twitter recommends to use /usr/local/hadoop/lib/native/

Add the following to your core-site.xml:

```
                <property>
                    <name>io.compression.codecs</name>
                    <value>org.apache.hadoop.io.compress.GzipCodec,
                    org.apache.hadoop.io.compress.DefaultCodec,
                    org.apache.hadoop.io.compress.BZip2Codec,
                    com.hadoop.compression.lzo.LzoCodec,com.hadoop.compression.l
        </value<
                </property>
                <property>
                    <nameio.compression.codec.lzo.class</name>
                    <value>com.hadoop.compression.lzo.LzoCodec</value>
    </property>
```

Add the following property to mapred-site.xml. We should note that the Cloudera guys backported the patch implementing this into their distribution f or us to make configuration easier (thanks!).

```
    <property>
        <name>mapred.child.env</name>
        <value>JAVA_LIBRARY_PATH=/path/to/your/native/hadoop-lzo/libs</value>
    </property>
```

To index your LZO-compressed files, run this command

```
hadoop jar /path/to/hadoop-lzo.jar com.hadoop.compression.lzo.LzoIndexer /lzo_logs
```

Take any job you used to have running over those files, say a WordCount, and run it just like before but using the hadoop-lzo LzoTextInputFormat in place of the TextInputFormat. You are done!

DISCUSSION

LZO is a tried and proven approach, but it is not the only one. Arguably, you can take almost any archiving format, such as a zip archives, and run an indexer on it. The result will be someone non-standard and home-grown, but it may be worth a try.

Using LZO or another splittable archive format has yet another advantage. So far we discussed storing the Hadoop input data in a compressed format. But you might as well output it in the compressed format, and pass it on to the next stages of processing. You just need to create an index each time. However, once you are on this road, you are not limited by the number of jobs that you can run one after another compressed – you can chain them as much as you want.

Does it really work? Big geeky internet companies are run by geeks, and they are known to unpredictabely abandon projects and quickly start other ones. However, LZO has been tried by many. TODO – reference on “is it really used.”