1

3 *Hadoop programming and databases*

Having seen the Hadoop's capabilities to store and process large volumes of data, you might now say, all this is very good, but my data does not reside on hard drives as web access logs. No, my data is stored in relational databases management systems, or RDBMS. How am I to use the wonderful capabilities of Hadoop and HBase in my situation?

You have two ways to go: integrate your RDBMS with Hadoop, or switch to a NoSQL datastore. That is, you may want to import your SQL data into HDFS, HBase, or Cassandra, and do it periodically, or you may re-design your solutions around NoSQL databases from the start. There are advantages in both approaches, and there are a number of tools and complete projects that will help you do it, so that you will not have to re-invent the wheel. What you will need to do, however, is to learn about those approaches, tools, and projects, in order to know how to bring the greatest benefit to your projects.

It is also likely that you are reading this book because you have outgrown your RDBMS solutions, and you would like your datastores to hold more data, to read and write this data very fast, to be scalable, and to be fault-tolerant. You also do not want to spend a fortune on the solution. If so, then read on, because you are not alone. Large companies like Google and Amazon had this problem early on, and they have created a new approach, called NoSQL, or key-value databases.

No program is an island, and in many cases it needs to interact with traditional (SQL) and non-traditional databases (NoSQL). As NoSQL databases get more prominent, the traditional and non-traditional classification fades away, but since this is an introductory chapter, we need somewhere to start. On the one hand, there are prominent companies and web sites where NoSQL databases are paramount, having taken the place of SQL databases. On the other hand, a common use case of

running MapReduce jobs is to exporting the results of the Hadoop processing to a SQL database of choice, for reporting and further analysis.

The major goal of this chapter is thus twofold. We will look at the SQL databases, to see what's wrong with them, if anything. We will show you how and to what extent you can integrate your existing RDBMS solutions into Big Data workflow. The second goal is to introduce you to RDBMS alternative, the NoSQL databases. Since NoSQL databases are new to the majority of readers, we have divided our treatment of them into two parts. In this chapter we will tell you what NoSQL databases our, what they are good for, and how use them to store and retrieve your data. Later, in chapter 5, we will explain the best programming practices, design of data models in NoSQL environment, and address questions such as transactions, so that your systems behave well under the big load that they are designed for.

1

Footnote 1 Given that the RDBMS vs NoSQL is a highly debated subject, nothing I write below is my personal point of view, but is rather based on solid research done by others, usually, the creators of the various products or persons prominent in the world of databases. You are invited to consult an excellent overview of the subject by Christof Strauch, from Stuttgart Media University, http://bit.ly/grXA4S, where you can find the attributions.

3.1 Connecting to SQL databases

Suppose you need to load your data from MySQL to Hadoop. But why might you want that? A typical use case might be when you have an ever-growing MySQL database, which comes from users of a web site, and you require significant amount of computations on this data. You might be running a dating site, and you may want to compare the match of each of (millions) of the candidates to each of the other ones. Or you may be running an auction web site, where you may need to try to match all bids with all offers.

There are a few approaches you can take. You can import the data into Hadoop on a regular basis, then run the computations. This can be done with Sqoop, HIHO, custom file formats on top of Hadoop API, Cascading, and cascading-dbmigrate. You could also dump the data into text format files for Hadoop into HDFS, which will also allow you to parallelize the computations.

Why would you want to load complete data every time? Because your users update it. To optimize, you can select only rows where the information was

updated, and match that, keeping the unchanged results in some other storage. In this section we will discuss the pros and cons of each tool and approach, so that you can select the right one for your situation.

3.1.1 Relational databases and SQL

What is wrong with relational databases? The short answer is: almost nothing is wrong with them. We will expand on a few of their faults below, SQL databases are ubiqutous and serve their purposes very well. The programmers know them, even end users can write ad-hoc queries in SQL.

There is little doubt that the SQL databases are here to stay. That is true of almost any technology, certainly of such wide-spread one as SQL. In fact, most of the big users of NoSQL databases have SQL-based applications also.

If so, then what is the reason for the NoSQL movement? Michael Stonebreaker explains it as follows. "The current RDBMS code lines, while attempting to be a one size fits all solution, in fact, excel at nothing”. They cannot compete with “specialized engines in the data warehouse, stream processing, text, and scientific database markets” which outperform them “by 1–2 orders of magnitude”. Which inherent flaws the critics find in relational database management systems and which suggestions do they provide for the “complete rewrite” they are requiring?

RDBMSs have been architected more than 25 years ago when the hardware characteristics, user requirements and database markets where different from those today. If your application does not need all of the generalities of an RDBMS, then you should instead look at a specific storage engine that excells at a particular task.

In the words of BJ Clark, “If I need reporting, I won’t be using any NoSQL. If I need caching, I’ll probably use Tokyo Tyrant, and so. We have summarized his views in the table below.

Table 3.1 The world of databases according to BJ Clark

Need	Solutions
Reporting	No NoSQL
Caching	Probably Tokyo Tyrant
ACIDity	No NoSQL
Tons of counters	Redis
Transactions	Postgres
Ton of a single type of documents	Probably Mongo
Write 1 billion objects a day	Probably Voldemort
Full text search	Probably Solr
Full text search of volatile data	Probably Sphinx

This opens many new worlds. With this in mind, let us look at how to unlock the data stored away in traditional SQL databases.

Technique 1: Importing data from a relational databases

Relational databases are not a good fit for Hadoop and for Big Data in general. They do not scale well not in terms of data volumes and not in terms of processing load. We can rectify this situation by importing them into the Hadoop environment.

PROBLEM

We need to import an RDBMS into Hadoop environment

SOLUTION

We can use Sqoop. Sqoop (“SQL-to-Hadoop”) is a command-line tool with the following capabilities:

- Imports individual tables or entire databases to files in HDFS
- Generates Java classes to allow you to interact with your imported data
- Provides the ability to import from SQL databases straight into your Hive data warehouse

You can use Sqoop for one-time import, or you can write a script to regularly import your databases for processing, calling on Sqoop to do the work.

Here is what you can do with Sqoop:

```
codegen            Generate code to interact with database records
create-hive-table  Import a table definition into Hive
eval               Evaluate a SQL statement and display the results
export             Export an HDFS directory to a database table
help               List available commands
import             Import a table from a database to HDFS
import-all-tables  Import tables from a database to HDFS
list-databases     List available databases on a server
list-tables        List available tables in a database
version            Display version information
```

DISCUSSION

In evaluating Sqoop, like any open source project, you are well advised to consider the following: who created it, what is the license behind it, and how active is the user community. Even though open source products are free, you are the one who is going to use precious resources in order to learn it, to use it, and perhaps even to improve it. So what about Sqoop. It was created by the people at Cloudera, who offers commercial support for Hadoop and related products. It is open-sourced under the same Apache 2.0 license. Sqoop is part of the Cloudera distribution and is packaged for Fedora and Ubuntu. For example in Ubuntu you can get it by simply typing

```
sudo apt-get install sqoop
```

When would you want to use Sqoop for import? Here is one use case. Consider the task of processing access logs and analyzing user behavior on your web site. Users may present your site with a cookie that identifies who they are. You can log the cookies in conjunction with the pages they visit. This lets you coordinate users

with their actions. But actually matching their behavior against their profiles or their previously recorded history requires that you look up information in a database. If several MapReduce programs needed to do similar joins, the database server would experience very high load, in addition to a large number of concurrent connections, while MapReduce programs were running, possibly causing performance of your interactive web site to suffer.

The solution: periodically dump the contents of the users database and the action history database to HDFS, and let your MapReduce programs join against the data stored there. Going one step further, you could take the in-HDFS copy of the users database and import it into Hive, allowing you to perform ad-hoc SQL queries against the entire database without working on the production database.

Sqoop allows to connect to any database with JDBC, and it also also direct non-jdbc access to MySQL, and PostgreSQL. Still there are more datasources that you may need to connect to, and we will discuss it when talking about HIHO.

Technique 2: Using database connector classes with Hadoop

At times you may want to connect to an RDBMS from your code, rather than using an out-of-process separate command-line utility. Your need may be simple enough, so that you may want to use no third-party code, but connect to an RDBMS directly, using only Hadoop API.

PROBLEM

We need to connect to an RDBMS with Hadoop API

SOLUTION

We will use DBInputFormat and DBOutputFormat. When you want your Hadoop job to read the data directly from a SQL database and write it there, the answer is provided by the DBInputFormat and DBOutputFormat classes.

DBInputFormat uses JDBC to connect to data sources. Therefore, DBInputFormat can work with MySQL, PostgreSQL, and several other databases. The driver appropriate to your database must be made available to DBInputFormat. You may it into the $HADOOP_HOME/lib/ directory on your Hadoop TaskTracker machines, and on the machine where you launch your jobs from. We have described other ways of making third-party jar available to your Hadoop jobs.

DBInputFormat extends the InputFormat. InputFormat is what tell Hadoop how to read your input data. Thus, when you use DBInputFormat, you are telling it to read the data from a database. To use if, you first need to configure your job. Imagine that you have a MySQL database with the following table:

```
CREATE TABLE employees (
  employee_id INTEGER NOT NULL PRIMARY KEY,
  name VARCHAR(32) NOT NULL);
```

You will then have to configure your job in the following way:

Listing 3.1 Configure MapReduce job for MySQL access

```
JobConf conf = new JobConf(getConf(), MyDriver.class);

conf.setInputFormat(DBInputFormat.class);
DBConfiguration.configureDB(conf,
    “com.mysql.jdbc.Driver”,
    “jdbc:mysql://localhost/mydatabase”);

String [] fields = { “employee_id”, "name" };
DBInputFormat.setInput(conf, MyRecord.class, “employees”,
    null,  “employee_id”, fields);
```

1. **Access MySQL with a JDBC driver**
2. **But use Hadoop InputFormat**
3. **Tell it which fields you are doing to read**

After this, you proceed as usual by setting your Mapper and calling JobClient.runJob(conf) to run the job.

The following example provides a DBWritable implementation that holds one record from the employees table, as described above:

Listing 3.2 DBWritable implementation

```
class MyRecord implements Writable, DBWritable {
  long id;
  String name;

  public void readFields(DataInput in) throws IOException {
    this.id = in.readLong();
    this.name = Text.readString(in);
  }
  public void readFields(ResultSet resultSet)
      throws SQLException {
    this.id = resultSet.getLong(1);
    this.name = resultSet.getString(2);
  }
  public void write(DataOutput out) throws IOException {
    out.writeLong(this.id);
    Text.writeString(out, this.name);
  }
  public void write(PreparedStatement stmt) throws SQLException {
    stmt.setLong(1, this.id);
    stmt.setString(2, this.name);
```

1. **Required by your DBInputFormat**
2. **Implement required methods**

```
  }
}
```

A java.sql.ResultSet object represents the data returned from a SQL statement. It contains a cursor representing a single row of the results. This row will contain the fields specified in the setInput() call. In the readFields() method of MyRecord, we read the two fields from the ResultSet. The readFields() and write() methods that operate on java.io.DataInput and DataOutput objects are part of the Writable interface used by Hadoop to marshal data between mappers and reducers, or pack results into SequenceFiles.

Now you can use the data in a mapper. The mapper receives an instance of your DBWritable implementation as its input value. The input key is a row id provided by the database, but you don't need the row it and will probably discard this value. Here how this looks in the code

```
 public class MyMapper extends MapReduceBase
    implements Mapper<LongWritable, MyRecord, LongWritable, Text> {
  public void map(LongWritable key, MyRecord val,
      OutputCollector<LongWritable, Text> output, Reporter reporter) throws IOExcept
    output.collect(new LongWritable(val.id), new Text(val.name));
  }
}
```

If you want to write the results into a SQL database, DBOutputFormat, will allow you to to do that. When setting up the job, call conf.setOutputFormat(DBOutputFormat.class); and then call DBConfiguration.configureDB() as before. The DBOutputFormat.setOutput() method then defines how the results will be written back to the database. Its three arguments are the JobConf object for the job, a string defining the name of the table to write to, and an array of strings defining the fields of the table to populate. e.g., DBOutputFormat.setOutput(job, "employees", "employee_id", "name");. The same DBWritable implementation that you created earlier will work to write records back into the database. The write(PreparedStatement stmt) method will be invoked on each instance of the DBWritable that you pass to the OutputCollector from the reducer. At the end of reducing, those PreparedStatement objects will be turned into INSERT statements to run against the SQL database.

DISCUSSION

There are, of course, many utilities that let you avoid going to this level, which

may be compared to driver-level access. One of them is Sqoop, which we have just described. In fact, Sqoop uses DBInputFormat internally. Still, there is some merit in describing how this works. You may want to learn from the Sqoop's way of doing it, or you may want to know what is available, even if you bypass it by using Sqoop or HIHO. In covering this subject, the author is indebted to Cloudera's Aaron Kimball, and to his presentation at the SF Hadoop User Group.

Natually, there are limitations in using DBInputFormat with Hadoop. JDBC allows applications to generate SQL queries which are executed against the database; the results are then returned to the calling application. Keep in mind that you will be interacting with your database via repeated SQL queries. Therefore: Hadoop may need to execute the same query multiple times. It will need to return the same results each time. So any concurrent updates to your database, etc, should not affect the query being run by your MapReduce job. This can be accomplished by disallowing writes to the table while your MapReduce job runs, restricting your MapReduce’s query via a clause such as “insert_date < yesterday,” or dumping the data to a temporary table in the database before launching your MapReduce process.

In order to parallelize the processing of records from the database, Hadoop will execute SQL queries that use ORDER BY, LIMIT, and OFFSET clauses to select ranges out of tables. Your results, therefore, need to be orderable by one or more keys (either PRIMARY, like the one in the example, or UNIQUE).

In order to set the number of map tasks, the DBInputFormat needs to know how many records it will read. So if you’re writing an arbitrary SQL query against the database, you will need to provide a second query that returns the number of rows that the first query will return (e.g., by using COUNT and GROUP BY). With these restrictions in mind, there’s still a great deal of flexibility available to you. You can bulk load entire tables into HDFS, or select large ranges of data. For example, if you want to read records from a table that is also being populated by another source concurrently, you might set up that table to attach a timestamp field to each record. Before doing the bulk read, pick the current timestamp, then select all records with timestamps earlier than that one. New records being fed in by the other writer will have later timestamps and will not affect the MapReduce job. Finally, be careful to understand the bottlenecks in your data processing pipeline. Launching a MapReduce job with 100 mappers performing queries against a database server may overload the server or its network connection. In this case, you’ll achieve less parallelism than theoretically possible, due to starvation, disk

seeks, and other performance penalties. Mappers mostly do read operations on the data, which can have more parallelism than writes. But reducers usually write data, and will suffer from disk seeks and other possible bottlenecks mentioned above.

The DBOutputFormat has similar limitations. It writes to the database by generating a set of INSERT statements in each reducer. The reducer's close() method then executes them in a bulk transaction. Performing a large number of these from several reduce tasks concurrently can swamp a database. If you want to export a very large volume of data, you may be better off generating the INSERT statements into a text file, and then using a bulk data import tool provided by your database to do the database import.

Technique 3: Connecting to databases with HIHO

PROBLEM

I want to connect to an RDBMS with a third-party library. I want it to offer additional capabilities, such as other data source connectors.

SOLUTION

Use HIHO. HIHO means Hadoop In, Hadoop Out. You can use HIHO as command-line utility, which can be as simple as typing

```
${HIHO_HOME}/scripts/hiho export mysql -conf mysqlExport.xml -inputPath
-url -userName -password -queryPrefix
```

If you use the following command

```
./hiho import -jdbcDriver com.mysql.jdbc.Driver -jdbcUrl jdbc:mysql://localhost:3306
```

this places the delimited output in the output folder on hdfs.

To start using HIHO, downloading the complete project from GitHub. Navigate to the project's directory and type "mvn" to invoke the maven build. It will come back with a list of targets. By typing `mvn site` you will generate the project API, which will tell you how to integrate HIHO into your code.

Here are the various formats HIHO provides which can be used directly in client code:

```
1. NoKeyOnlyValueOutputFormat - used for writing only values and ignoring the keys.
2. AppendTextOutputFormat - write additional files to a pre existing output director
3. AppendSequenceFileOutputFormat - same as above with ability to write Sequence Fil
4. FTPTextOutputFormat - write text output to an FTP location.
```

```
5. FileStreamInputFormat - the entire file is provided as a stream to the mapper. He
6. DBQueryInputFormat - for custom querying a relational database using complex join
```

DISCUSSION

HIHO is on open source project, so the same set of questions come up as we looked at when evaluating the possible use of Sqoop: who created it, what is the licence, and how active is the user/maintainer community. It is licensed under Apache V2.0 license and hosted on GitHub. At the time of writing the project maintainer Sonal Goyal reported a few hundred users, and her company Nube Technologies offers commercial support in using HIHO and in Hadoop consulting. Less active than Sqoop, HIHO nevertheless features a a number of data connector classes. So when and why would you use it?

The short answer: you would use HIHO for the specialized IO capabilities that it allows, such as SalesForce, FTP, performance optimized SQL import/export, and involved SQL operations, such as joins and where conditions. You may also want to consider if you want to integrate the code into your project, rather than using Sqoop, which is an out-of-process utility. Of course, Sqoop also generated classes for you to use, so as often happens you have some overlapping capabilities. You can also use HIHO for incremental update, dedup, append, merge your data on Hadoop. In addition, HIHO offers Ivy based build and dependency management and Junit and mockito based test cases. This is convenient if your project already uses these technologies.

Technique 4: RDBMS import and export with Cascading

Let's say you are an avid user of Cascading. It simplifies your MapReduce programming, and you do not want to go back to dealing with MapReduce in order to import an RDBMS.

PROBLEM

How to import an RDBMS while staying with the Cascading framework

SOLUTION

Use cascading.jbdc or dascading-dbmigrate. cascading.jdbc provides support for reading/writing data to/from an RDBMS via JDBC drivers when bound to a Cascading data processing flow. INSERT and UPDATE are both supported during sinking. Custom SELECTs can also be used during sourcing. Alternatively, you can use cascading-dbmigrate, which is a command-line utility.

DISCUSSION

One of the frameworks building on top of MapReduce and hiding the

complexity of MapReduce is Cascading. We will be talking about Cascading later, but in this section we need to mention the facilities for SQL import and export provided by in Cascading by cascading.jdbc. If your project already uses Cascading, it may make sense for you to re-use your knowledge when it comes to SQL connection.

cascading.jbdc is separate open source project hosted on GitHub. When using cascading.jdbc, you need to know that that UPDATEs are very slow. It is a RDBMS issue, but you should take it into account and to use them sparingly.

Also worthy note is db-migrate. It gives you a command line utility for SQL imports, and thus serves two purposes. It is an alternative to Sqoop written with Cascading and uses a custom version of cascading.jdbc internally. You can use it as is, evaluating if it fits your purposes better than Sqoop. You can also use its code - which is open sourced - as a good example of using the Cascading framework. You can find it on GitHub here, https://github.com/backtype/cascading-dbmigrate.

3.2 Key-value databases

The goal of the second part of this chapter is to introduce you to the key-value, or NoSQL databases. As we have mentioned before, it can mean "no SQL implemented," or "not only SQL," or "take a new, specialized approach to databases storage engines. We will explain why they are needed and how they differ. You will be able to select the right NoSQL database for your application. We will show the basics of working with them.

3.2.1 Non relational databases

The term NoSQL may mean different things to different people. Some take to signify the choice of not implementing the SQL interface. Once the designers are free from the ACID (atomicity, consistency, isolation, durability) requirements that are usually associated with with SQL, they can pursue completely different design decisions. To others it means "not only SQL" - which means that in addition to the traditional SQL databases the designers can and should consider specific storage engine solutions optimized for their application's requirements. Yet others equate the NoSQL to hashmap databases, and to them it is simply a giant distributed hashmap (maybe sparce and multi-dimensional), whether on persisted on disk or in-memory.

The two major influences in the world of NoSQL databases are Google's BigTable and Amazon's Dynamo. The first two major advantages of the use of

specialized NoSQL databases are scalability and performance. Both BigTable and Dynamo run on hundreds and thousands of servers. Cassandra, an open-source project inspired by Dynamo, is reported to write 2,500 faster than MySQL. Therefore, let us first look at the thinking behind these two NoSQL databases. The common reasons to develop and use NoSQL datastores can be summarized as follows.

- Avoidance of Unneeded Complexity. Relational databases provide a variety of features and strict data consistency. But this rich feature set and the ACID properties implemented by RDBMSs might be more than necessary for particular applications and use cases. Instead of implementing all of the RDBMS capabilities, you might need to store only users' sessions, perhaps multiple instances of them. They do not have to be persisted, and an occasional loss of a session does not present a problem. If that is all your require, you may choose to use an in-memory database, with a much-higher performance.
- Horizontal scalability and commodity hardware. RDBMS's were created in the times when thinking concentrated on central expensive machines that could be scaled vertically and could be made to provide all relational capabilities. This thinking is questioned today. In contrast, NoSQL datastores are designed to scale well in the horizontal direction and not to rely on expensive highly available hardware.
- Avoidance of expensive object-relational mapping. Our software today is object-oriented, which RDBMS's are schema-oriented. The translation process is expensive both in terms of performance and in terms of the learning curve of the programmers. RDBMS's appeared when the hardware was expensive while people were cheap. Today the situation is reversed. Hardware is getting more powerful and less expensive, while the software developers are arguably becoming more sophisticated, and are thus being paid more.

Therefore, many designers take a different approach. They analyze the requirements of their applications and uses or build a datastore to fit those requirements.

- Yesterday's vs. Today's needs. James Gosling stated the eight fallacies of distributed computing. “Essentially everyone, when they first build a

distributed application, makes the following eight assumptions. All prove to be false in the long run and all cause big trouble and painful learning experiences. (1) The network is reliable; (2) Latency is zero; (3) Bandwidth is infinite; (4) The network is secure; (5) Topology doesn't change; (6) There is one administrator; (7) Transport cost is zero; (8) The network is homogeneous." The NoSQL architectures address the realities of distributed and cloud-based computing, expressed in the correct understanding of these eight fallacies.

3.2.2 BigTable from Google

BigTable is described as "a distributed storage system for managing structured data that is designed to scale to a very large size: petabytes of data across thousands of commodity servers" It is used by dozens if not hundreds projects at Google, with the most prominent including web indexing, GMail, Google Apps, Google Earth, Google Analytics.

BigTable represents the evolution of data storage and processing architectures at Google. Initially most applications were based on Google File System, GFS, and MapReduce, working in conjunction with GFS. When the requirements of many applications started to differ significantly from the assumption on which the GFS was based, BigTable came to the forefront. BigTable, in turn, is transforming into something else. A very interesting discussion on this evolution can be found here, http://bit.ly/48Cxag. The major principle of this evolution is strikingly similar to eXtreme programming: don't do what is not called for yet. Some of the planned features for BigTable never got implemented because the need for them never arose.

So what is the basic principle behind BigTable? It is familiar to any programmer who has used hash tables. BigTable is simply a giant distributed hash table. To be sure, it has additional levels of complexity, such as families (groups of rows), but it essence the storage unit in BigTable is a key, a column, and a value. You immediately see where BigTable differs from an RDBMS: it does not have a rigid schema. You define the columns as you go. Moreover, if some columns are not present in a row, you can safely forget about these columns and only write the values for those columns that are present. Because of this, BigTable is called a sparce hash table. Another prominent feature of BigTable is the timestamp: a new value in a cell does not erase the previous one, instead, it is recorded with a different timestamp. You thus have value versioning.

BigTable is not directly available to programmers outside of Google, except indirectly in the Google App Engine. Therefore, for all practical purposes we will be discussing HBase, which is the open source implementation of BigTable, much as Hadoop is the open source implementation of MapReduce.

Technique 5: Substituting BigTable with HBase

The capabilities of BigTable are great. However, it is a proprietary Google solution, and unless you want to build your applications with Google App Engine, you cannot use it. However, you would like to try the NoSQL approach and see what it can do for you.

PROBLEM

How to add the capabilities of Google's BigTable to your application

SOLUTION

Staying with the Hadoop environment, use HBase. To illustrate how you work with HBase, you need to first install it on your computer. As with Hadoop, you can download it from hbase.apache.org, or install it using Cloudera's distribution. In the former case you simply unpack it to a directory, and in the latter case you type

```
sudo apt-get install hadoop-hbase
```

For our purposes it will be enough to run HBase server in a standalone single-node mode. You can do this by navigating into the HBase folder (if you use the Apache distribution) and issuing a command

```
bin/start-hbase.sh
```

Once HBase is running, you can type

```
bin/hbase shell
```

and you will see something like the following output

```
HBase Shell; enter 'help <RETURN>' for list of supported commands.
Type "exit <RETURN>" to leave the HBase Shell
Version 0.90.0, r1056514, Fri Jan  7 21:22:53 UTC 2011
```

HBase contains tables. To see the tables you already have, you use the 'list' command, which can give you the following output

```
>list
TABLE
mailbox-status
myTable
```

To get more information on any of the tables, you type `describe 'table-name'`. For example, in our case above, we can see this output

```
> describe 'mailbox-status'
DESCRIPTION                                          ENABLED
 {NAME => 'mailbox-status', FAMILIES => [{NAME => 's true
 tats', BLOOMFILTER => 'NONE', REPLICATION_SCOPE =>
 '0', COMPRESSION => 'NONE', VERSIONS => '3', TTL =>
  '2147483647', BLOCKSIZE => '65536', IN_MEMORY => '
 false', BLOCKCACHE => 'true'}]}
1 row(s) in 0.0230 seconds
```

Here we see for the first time the concept of column families. To you it is just a nice device to separate the areas of concern and to group various columns in your table together. To continue our investigation of HBase, let us set a value in a column. For example, we can create another table with its column family

```
> create 'hadoop_table', 'hadoop_family'
```

then set values for some columns and list the content

```
> put 'hadoop_table', 'row_key1', 'hadoop_family:column1', 'value1'
> put 'hadoop_table', 'row_key2', 'hadoop_family:column1', 'value2'
> put 'hadoop_table', 'row_key2', 'hadoop_family:column2', 'value3'
> put 'hadoop_table', 'row_key3', 'hadoop_family:column2', 'value3'

hbase(main):014:0> scan 'hadoop_table'
ROW                             COLUMN+CELL
 row_key1                       column=hadoop_family:column1, timestamp=13031340250
 row_key2                       column=hadoop_family:column1, timestamp=13031340438
 row_key2                       column=hadoop_family:column2, timestamp=13031340517
 row_key3                       column=hadoop_family:column2, timestamp=13031343687
3 row(s) in 0.0520 seconds
```

That's it in a nutshell, but there is a lot of important stuff that happened. First of all, you saw that to define the column, you just create it at the time you set its value. Here a key-value database differs from an RDBMS: as we said, it has no rigid schema, but we instead define it as we go. Secondly, I have used row_key1, row_key2, etc., for key rows in the table. This is much like the row id, or key, in the RDBMS, only I had to generate it myself, and I have a complete freedom as to how I do this. We will discuss the various strategies for key row generations in the later chapters on the advanced use of HBase. Thirdly, I have defined column1 and column2 for row_key2, but not for row_key1 or row_key3. I am free to define or to omit any column in a row.

DISCUSSION

A word about HBase versions. At the time of writing, version 0.20 and 0.21 represented the stable version run by companies in production, and versions 0.89 and 0.90 symbolized in their name HBase getting ready for official 1.0 release. Don't hesitate and grab the latest. 0.90 has been used by many in production without a problem.

A word of caution on running HBase in pseudo-distributed and truly distributed mode. This may require some configuration effort on your part, and you should follow the instruction even for pseudo-distributed mode. HBase relies on two components out of the Hadoop zoo: HDFS and ZooKeeper. The good side of it is that the HBase code does not have to be concerned with storage space and with coordination services. But the consequence of it for you is that you have to configure it correctly.

The examples above showed you how easy it is to start working with a key-value database. They also showed you the flexibility unusual for the RDBMS schema.Now it is time to do the same from the code.

Technique 6: Loading and querying HBase using Hadoop

PROBLEM

How do I access HBase from my application code?

SOLUTION

You can access HBase from a standalone client, or from a MapReduce job. First, let us look at the code that accesses HBase as a standalone client. We will set some values and read them back.

Listing 3.3 Client accessing HBase

```
package com.shmsoft.hadoopinpractice;
```

```
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.client.Get;
import org.apache.hadoop.hbase.client.HTable;
import org.apache.hadoop.hbase.client.Put;
import org.apache.hadoop.hbase.client.Result;
import org.apache.hadoop.hbase.client.ResultScanner;
import org.apache.hadoop.hbase.client.Scan;
import org.apache.hadoop.hbase.util.Bytes;

public class HBaseClientAccess {

    public static void main(String[] args) throws IOException {

        Configuration config = HBaseConfiguration.create();

        config.set("hbase.zookeeper.quorum", zoookeeper_ip);
        config.set("hbase.zookeeper.property.clientPort", "2181");

        HTable table = new HTable(config, "myTable");

        String myRowKey = "myRow";
        if (args.length > 0) {
            myRowKey = args[0];
        }

        Put p = new Put(Bytes.toBytes(myRowKey));

        p.add(Bytes.toBytes("myFamily"), Bytes.toBytes("someQualifier"),
                Bytes.toBytes("Some Value"));

        table.put(p);

        Get g = new Get(Bytes.toBytes(myRowKey));
        Result r = table.get(g);
        byte[] value = r.getValue(Bytes.toBytes("myFamily"),
                Bytes.toBytes("someQualifier"));

        String valueStr = Bytes.toString(value);
        System.out.println("GET: " + valueStr);

        Scan s = new Scan();
        s.addColumn(Bytes.toBytes("myFamily"), Bytes.toBytes("someQualifier"));
        ResultScanner scanner = table.getScanner(s);
        try {

            for (Result rr = scanner.next(); rr != null; rr = scanner.next()) {
                System.out.println("Found row: " + rr);
            }

        } finally {
            scanner.close();
        }
    }
```

1. This reads the HBase configuration from your path, but local HBase runs it for you
2. Zookeeper must be running
3. Table called "myTable" must be present, create it in the HBase shell
4. Row keys are binary (bytes)
5. Family "myFamily" must be there, the rest will be created
6. This creates both the column name and value

```
}
```

Now let us look at the code accessing HBase from a MapReduce job. Imagine the following scenario...

Listing 3.4 Accessing HBase from a MapReduce job

```
package com.shmsoft.hadoopinpractice;

import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.KeyValue;
import org.apache.hadoop.hbase.client.Put;
import org.apache.hadoop.hbase.client.Result;
import org.apache.hadoop.hbase.client.Scan;
import org.apache.hadoop.hbase.filter.FirstKeyOnlyFilter;
import org.apache.hadoop.hbase.io.ImmutableBytesWritable;
import org.apache.hadoop.hbase.mapreduce.TableMapReduceUtil;
import org.apache.hadoop.hbase.mapreduce.TableMapper;
import org.apache.hadoop.hbase.mapreduce.TableOutputFormat;
import org.apache.hadoop.hbase.mapreduce.TableReducer;
import org.apache.hadoop.hbase.util.Bytes;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.util.GenericOptionsParser;

public class MailboxIdCounter {

    static final String NAME = "mailbox-status";
    private static IntWritable ONE = new IntWritable(1);

    static class RowCounterMapper

            extends TableMapper<ImmutableBytesWritable, IntWritable> {
        private static enum Counters {
            ROWS
        }

        @Override
        public void map(ImmutableBytesWritable row, Result values, Context context)
                throws IOException, InterruptedException {
            for (KeyValue value : values.list()) {
                if (value.getValue().length > 0) {
                    ImmutableBytesWritable key = new ImmutableBytesWritable(value.ge

                    context.write(key, ONE);
                }
            }
        }
    }

     public static class RowCounterReducer extends TableReducer<ImmutableBytesWritab
```

❶ **Extend TableMapper for HBase access**

❷ **Emit the map with the key of mailbox_id and count of 1**

```
        @Override
        public void reduce(ImmutableBytesWritable key, Iterable<IntWritable> values,
                throws IOException, InterruptedException {
            int sum = 0;

            for (IntWritable val : values) {
                sum += val.get();
            }
            Put put = new Put(key.get());
            put.add(Bytes.toBytes("stats"), Bytes.toBytes("mailboxID"), Bytes.toByte
            context.write(key, put);
        }
    }

    public static Job createSubmittableJob(Configuration conf, String[] args)
            throws IOException {
        String tableName = args[0];
        Job job = new Job(conf, NAME + "_" + tableName);
        job.setJarByClass(MailboxIdCounter.class);
        // Columns are space delimited
        StringBuilder sb = new StringBuilder();
        final int columnoffset = 1;
        for (int i = columnoffset; i < args.length; i++) {
            if (i > columnoffset) {
                sb.append(" ");
            }
            sb.append(args[i]);
        }
        Scan scan = new Scan();
        scan.setFilter(new FirstKeyOnlyFilter());
        if (sb.length() > 0) {
            for (String columnName : sb.toString().split(" ")) {
                String[] fields = columnName.split(":");
                if (fields.length == 1) {
                    scan.addFamily(Bytes.toBytes(fields[0]));
                } else {
                    scan.addColumn(Bytes.toBytes(fields[0]), Bytes.toBytes(fields[1]
                }
            }
        }
        job.setOutputFormatClass(TableOutputFormat.class);
        TableMapReduceUtil.initTableMapperJob(tableName, scan,
                RowCounterMapper.class, ImmutableBytesWritable.class, IntWritable.cl

        TableMapReduceUtil.initTableReducerJob("mailbox-status", RowCounterReducer.c
        return job;
    }

    public static void main(String[] args) throws Exception {
        Configuration conf = HBaseConfiguration.create();
        String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs()
        if (otherArgs.length < 1) {
            System.err.println("ERROR: Wrong number of parameters: " + args.length);
            System.err.println("Usage: RowCounter <tablename> [<column1> <column2>..
            System.exit(-1);
        }
```

3 **Count the maps with a given mailbox_id**

4 **Boilerplate to create and run a job accessing HBase**

5 **Write to table mailbox_status**

```
        Job job = createSubmittableJob(conf, otherArgs);
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

DISCUSSION

Using the code above, you can build client applications that will scale because, although you may have any number of clients - say, web applications in a browser - HBase scales well for huge loads under concurrent use. However, there are many other NoSQL databases that scale equally well or arguably even better. When would you want to use HBase?

The baggage that comes with HBase is it use of HDFS and Zookeeper. If you already have an HDFS cluster that you use to run MapReduce jobs, it would make sense to expand its use and add HBase to the mix. That was the case of how things developed at Google.

HBase is an Apache open source project, and it is under active development and use, with a significant user group.However, if you have none of these yet, and you are just beginning to build an application that will need a high-performance scalable database that can take thousands of writes per second, and one that can scale well, you may well be advised to look at Cassandra.

Technique 7: getting a taste of NoSQL database with Cassandra

I want to build a solution with a NoSQL data, but do not need HBase/HDFS and I do not want to administer a Hadoop cluster.

PROBLEM

Get the power of NoSQL without the baggage of Hadoop

SOLUTION

You can use Cassandra, which is a standalone clustered NoSQL database. Cassandra is very easy to set up and use. It does not have a single point of failure, but instead all of its nodes are equal and can store and retrieve data.

DISCUSSION

Cassandra is an open source project inspired by Amazon's Dynamo. It is HBase's competitor and is in production use at a number of companies. There are many beautiful things that Cassandra's design can teach you, and it can provide you with a number of practical benefits. We will show advanced techniques of working with Cassandra in chapter 5, but we had to mention it here, together with HBase. Cassandra has significant publications dedicated to it alone, but we plan to give you enough information to see that it can give you, and how to start with it.

Technique 8: Connecting to Amazon SimpleDB with SimpleJPA and typica

PROBLEM

I do not have and do not want to have a cluster of hardware dedicated to my application. What I do want, however, is to prototype and potentially even to host a NoSQL solution on a rented cluster.

SOLUTION

You can build your solutions on Amazon's Elastic Compute Cloud - EC2. Amazon provides a commercial service called SimpleDB. It is based on the same principles as other key-value storage systems we have described before.

To give you an example of what would be involved in accessing SimpleDB from your code, let us look at the following example

Listing 3.5 Accessing SimpleDB

```
package com.shmsoft.hadoopinpractice;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Properties;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

import com.xerox.amazonws.sdb.Domain;
import com.xerox.amazonws.sdb.ItemAttribute;
import com.xerox.amazonws.sdb.ListDomainsResult;
import com.xerox.amazonws.sdb.QueryWithAttributesResult;
import com.xerox.amazonws.sdb.SimpleDB;
import com.xerox.amazonws.sdb.SDBException;

public class TestSimpleDB {

    private static Log logger = LogFactory.getLog(TestSimpleDB.class);

    public static void main(String[] args) throws Exception {
        try {
            Properties props = new Properties();

            props.load(TestSimpleDB.class.getClassLoader().getResourceAsStream("aws.

            SimpleDB sdb = new SimpleDB(props.getProperty("aws.accessId"), props.get

            logger.info("domains:");
            String nextToken = "";
            while (nextToken != null) {
                ListDomainsResult result = sdb.listDomains(nextToken, 10);
                List<Domain> domains = result.getDomainList();
                for (Domain dom : domains) {
                    logger.info(dom.getName());
```

❶ **Read AWS credentials from your environment, safe practice**

```
    }
    nextToken = result.getNextToken();
}
Domain dom = sdb.createDomain(args[0]);

QueryWithAttributesResult qr = dom.selectItems("select * from " + args[0
Map<String, List<ItemAttribute>> iList = qr.getItems();
for (String id : iList.keySet()) {
    logger.info("item : " + id);
}

Map<String, List<ItemAttribute>> items = new HashMap<String, List<ItemAt
List<ItemAttribute> list = new ArrayList<ItemAttribute>();
list.add(new ItemAttribute("test1", "value1", false));
list.add(new ItemAttribute("t0u000dst1", "value2", false));
list.add(new ItemAttribute("test1", "Jérôme", false));
list.add(new ItemAttribute("test1", "0u000dvalue4&gt;", false));
list.add(new ItemAttribute("test1", "value5", false));
list.add(new ItemAttribute("test1", "value6", false));
items.put("00001", list);

list = new ArrayList<ItemAttribute>();
list.add(new ItemAttribute("test1", "value1", false));
list.add(new ItemAttribute("t0u000dst1", "value2", false));
list.add(new ItemAttribute("test1", "Jérôme", false));
list.add(new ItemAttribute("test1", "0u000dvalue4&gt;", false));
list.add(new ItemAttribute("test1", "value5", false));
list.add(new ItemAttribute("test1", "value6", false));
items.put("00002", list);

list = new ArrayList<ItemAttribute>();
list.add(new ItemAttribute("test1", "value1", false));
list.add(new ItemAttribute("t0u000dst1", "value2", false));
list.add(new ItemAttribute("test1", "Jérôme", false));
list.add(new ItemAttribute("test1", "0u000dvalue4&gt;", false));
list.add(new ItemAttribute("test1", "value5", false));
list.add(new ItemAttribute("test1", "value6", false));
items.put("00003", list);

list = new ArrayList<ItemAttribute>();
list.add(new ItemAttribute("test1", "value1", false));
list.add(new ItemAttribute("t0u000dst1", "value2", false));
list.add(new ItemAttribute("test1", "Jérôme", false));
list.add(new ItemAttribute("test1", "0u000dvalue4&gt;", false));
list.add(new ItemAttribute("test1", "value5", false));
list.add(new ItemAttribute("test1", "value6", false));
items.put("00004", list);

dom.batchPutAttributes(items);

logger.info("items after batch put");
qr = dom.selectItems("select * from " + args[0], null);
iList = qr.getItems();
for (String id : iList.keySet()) {
    logger.info("item : " + id);
}
```

2 Prepare a query for a key with a given start

3 Prepare a list to be inserted in the SimpleDB

4 Store the list with the key

```
            dom.batchDeleteAttributes(items);                    ❺ Test delete operation

            logger.info("items after batch delete");
            qr = dom.selectItems("select * from " + args[0], null);
            iList = qr.getItems();
            for (String id : iList.keySet()) {
                logger.info("item : " + id);
            }

        } catch (SDBException ex) {
            System.err.println("message : " + ex.getMessage());
            System.err.println("requestID : " + ex.getRequestId());
        }
    }
}
```

DISCUSSION

SimpleDB but it is not open source. It is proprietary to Amazon. However, if you are building your system based on Amazon AWS and are using Amazon services like EC2 and S3, it may make sense to use the other Amazon services like SimpleDB and messaging. True, your application will be vendor locked-in and specific, but Amazon is pretty good at what it does, and your principles of architecture will, if the need arises and with some work, be able to transfer to another possible framework.

3.3 Summary

In this chapter we have analyzed the strengths and weaknesses of SQL RDBMS. We saw cases when it would be appropriate to import the RDBMS into the Hadoop world or into a NoSQL datastore, and we saw the tools to do that.

We have also had our first look at NoSQL datastores, saw the pros and cons of using them, and got introduced to accessing them through a client code in your application.

In the next two chapters we will be looking at techniques that make your projects stable and scalable under big load - how to achieve that, and how to verify that this is indeed the case, before you go into production. In chapter 4 we discuss the best programming practices when working with Hadoop and MapReduce, and in chapter 5 we return to NoSQL databases and look deeper into the design and performance considerations that help assure successful projects.