NoSQL Databases

5

What the reader will learn:

- that Web 2.0 and Cloud Computing brought new challenges to the database arena
- that a number of data-centric solutions that are not relational have come to the fore
- how these new data storage mechanisms work by exploring three in detail
- that each new data storage type addresses a particular business need
- some of the strengths and weaknesses of these new approaches, together with an appreciation of why Relational may be with us for many years to come

5.1 Databases and the Web

From the very earliest days of web development programmers have been using databases to provide permanency, and a single source of truth, for their web-based systems. In the beginning this would typically mean connecting to a Relational database (RDBMS) back-end. This is particularly true for the many online trading systems that were developed, as the transactional nature of their *raison d'ētre* demanded the robustness provided by leading RDBMS to ensure reliable financial dealings.

To a degree, of course, the database itself will not care whether its clients are connecting using a two- three- or n-tier architecture. And the functionality they provided also quickly found a place in the far more interactive type of application that came about as a result of Web 2.0, and Cloud computing.

For many designers the choice usually was seen as one of Open Source (such as MySQL) or vendor supplied (such as Oracle). The debate often revolved around cost of purchase, cost of ownership, and trustworthiness. What it seldom revolved around was whether the database should be relational or not. Universities running computing courses would typically have a module called "Database Systems", which examined almost exclusively, the relational model. Naturally, therefore, as the students left and began to develop exciting new applications for the commercial world,

P. Lake, P. Crowther, *Concise Guide to Databases*,
Undergraduate Topics in Computer Science, DOI 10.1007/978-1-4471-5601-7_5,

there was little doubt in their mind that a database problem was to be solved with a relational solution.

However, as web driven systems began to expand, particularly when mass-usage systems such as Facebook and Twitter began to take off, it became clear that the relational model is not good at everything. It is said in some quarters, for example, that Relational does not scale well. And the fact that these new mass-usage systems are global and data is typically spread across many nodes in many countries, is seen by some as something that relational does not cope with well.

As well as pointing up some potential weaknesses in the relational approach, these new applications were often not actually transactional in nature. As we saw in Chap. 4, the ACID test is at the heart of providing the transactional robustness. However, it is a big processing overhead for a RDBMS to maintain the robustness required in an ACID compliant database. And that overhead becomes extremely difficult to manage if the data is spread over many servers all over the world. This usually manifests itself in very poor performance.

In addition to problems with the actual data storage aspect of systems, the retrieval of data from many nodes across the globe was also becoming problematic. Google, for example, have built a world leading brand on their ability to search and retrieve data quickly, and they recognised the weakness in the relational approach.

In the end organisations eventually took the decision to write their own database systems to meet these new demands. Google, famously, designed their own database to suit their needs, called BigTable. This is a proprietary database and you cannot buy a copy to use in your own environment, but the story is well documented, especially by Chang et al. (2006). It is NOT relational. Instead it is a distributed hash mechanism built on their own file handling system, GFS. Although it is Google's own product, and not openly available, other such databases do exist. HBase is an open source database that claims to have a similar data model to that of BigTable, for example. We review what Hbase and similar products allow us to do in the era of "Big Data" in the next chapter.

5.2 The NoSQL Movement

We must not loose sight of the tremendous leaps forward in data management and manipulation techniques that have occurred because of the powerful relational model. This chapter will go on to review some current alternative approaches, but readers should recognise that many corporations have invested many $millions in their RDBMS architecture and will not, especially in a period of comparative recession, rush to spend money on new database technology. Moreover, for many organisations, the current RDBMS does exactly what is required of it.

To quote from the Guide to Cloud Computing (Hill et al. 2013):

> [in the 1980s] … middle managers who needed information to help them to make business decisions would think nothing of having to wait days for the data they needed. The request would need to be coded, probably into Cobol, then perhaps handed to a punch-card operator who would punch the programme, and then on to await an allotted, and very valuable, processing slot. Output from this request would probably be on a continuous run of

> sprocket-holed paper, along with many other outputs. A further wait might ensue whilst awaiting the specialist paper decollator or burster.
> With the advent of personal computing in the 1980s managers were able to collect and manipulate data without waiting for the still very powerful mainframe. The Relational Database, with its English-like SQL language also helped empower decision makers. Critical information could be delivered in a few seconds and organisations could gain competitive advantage by accessing information swiftly. Moreover, new information might be uncovered by using Data Mining techniques.

But Relational databases have been with us for over 30 years. In that period we have seen, for example, the birth and rise to predominance of the Windows Operating System at the cost of command-line interfaces. We are seeing now a move away from traditional client-server applications towards available anywhere, browser or App driven applications. When you consider the technological change underpinning these advances, it is hardly surprising that database technologies, too, should be reviewed and improved.

As we will see in Chaps. 9 to 11, three areas of great import for any database administrator (DBA) are Scalability, Availability and Performance.

For many of the RDBMS vendors availability has always been a key selling point. The traditional measure for assessing how available a database has been is Uptime, often expressed as a percentage of total elapsed time. An uptime of 50 % would mean that the database was available for only half of the elapsed time (probably ensuring the host company went bust if it relied on the database for online trading!). The dream target for a DBA is what is called "Five Nines" availability, which is 99.999 % available.

It is true, however, that we are now in an era when differences between availability statistics for different vendor databases are slim. As a way of differentiating between vendor products, therefore, except for applications which are most sensitive to outages however small—one would hope that the system looking out for incoming nuclear warheads is offline for as short a time as possible, for example—this measure becomes less useful.

Today's sales battlegrounds are therefore more often based around Scalability and Performance. And it is precisely these two aspects that tend to be pushed by NoSQL tool providers as they claim their products outperform and out-scale traditional RDBMS. Naturally the traditional vendors are fighting back and making counter claims. Equally naturally the real position is somewhere between the two, with most answers starting with "It depends . . . ".

When examining the context in which database professionals work these days, we can perhaps get a better understanding of why the new tools might excel; in short, the task often now before them is to quickly find relevant data from terabytes of unstructured web content which may be stored across many widely dispersed nodes. Relational databases, with their large processing overhead in terms of maintaining the ACID attributes of the data they store, and their reliance on potentially processor hungry joins, are not the right tool for these needs.

Relational databases can, and do, store and manipulate very large datasets. In practice these are often vertically scaled systems; that is, they may have access to multiple CPUs to distribute the processing burden, but they share RAM and disks.

NoSQL databases tend to be horizontally scaled; that is the data is stored on one of many individual stand-alone systems which run relatively simple processes, such as key look-ups, or read/write against a few records, against their own data. The individual system onto which the data itself will be stored will be managed according to a key. This is known as "sharding".

Cattell (2010) provides a nice clear definition of horizontal scaling:

> The term "horizontal scalability" means the ability to distribute both the data and the load of [. . .] simple operations over many servers, with no RAM or disk shared among the servers.

5.2.1 What Is Meant by NoSQL?

Unfortunately the answer to this also begins with "It depends. . . ". This is a new term and is therefore going through a common phase in the process of new word definition wherein different people treat the meaning differently.

One frequent usage is merely "any database which doesn't use SQL". The problem with that is that it seems to start from the premise that SQL is the only existing database query tool. Object Query Language (OQL), for example has been with us for many years but is not normally thought of as NoSQL.

To add to the complexity, SQL is a well known, even well loved, query language. So much so that NoSQL databases are beginning to provide SQL-type support to help the poor old relational expert find data in the mysterious world of non-related data stores. Cassandra, for example, after starting with just a command-line, java-like query tool, now (since version 0.8) provides CQL.

Another definition is that NoSQL is N.O.SQL and that the N.O. stands for Not Only. Cassandra after this release might be seen as an example of this.

In this book, whilst not particularly favouring one or other definition, for clarity we will use NoSQL to mean:

> A database which does not store data using the precepts of the relational model and which allows access to the data using query languages which do not follow the ISO 9075 standard definition of SQL.

Perhaps the best way to get an understanding of what NoSQL actually means is to look at examples. Further on in the chapter we will be using a Column-based database and a Document-based database. These are two types of approach to storing data that are generally accepted to be "NoSQL". A good single source for most of the approaches available, and examples thereof, can be found at: http://nosql-database.org/. They claim to be "Your Ultimate Guide to the Non-Relational Universe!" and their definition of NoSQL, at least at the time of writing, is:

> Next Generation Databases mostly addressing some of these points: being non-relational, distributed, open-source and horizontally scalable.

5.3 Differences in Philosophy

Data processing tasks today can involve vast volumes of data which may be duplicated across many nodes in the Web. This can be when RDBMS start to struggle.

Traditionally a DBA would consider indexing as an approach to speeding up the location of particular data items. However, the index process is itself burdensome for the RDBMS, especially if the data is volatile. An index is, after all, just a pointer to the physical location of some data, so there has to be a secondary read process involved in any retrieval based on indexes. In addition, traditional RDBMS databases will have data in a number of tables which often need to be joined to respond to a user query. This too is process intensive for the management system.

When trying to impose ACID rules on data, ensuring that all data items on all nodes are identical before the user can access them is a vital, but time consuming step. The issue is that rows can become locked whilst they wait for network issues to be resolved. (More information on locking and indexes is to be found in the Performance chapter of this book (Chap. 11).)

In recent years, however, some interesting papers have been published which argue that ACID is not the only legitimate set of rules for a database to abide to. CAP theorem, for example (Brewer 2012), makes the point that there is always a balance between competing desires to be found when designing distributed systems in general, and for the Web in particular. The three competing needs are said to be:

- Consistency
- Availability
- Partition Tolerance (coping with breaks in physical network connections)

Before we go any further we need to establish just what is meant by these terms, especially since the "Consistent" used here does not use the same definition as the one in the ACID approach.

The ***consistent*** as used in the ACID test means that the data that is stored in the database has all rules or constraints applied to it. In other words, the data complies with all the business rules designed into the database.

The ***consistent*** as used in the CAP theorem means that all the data accessible by clients, across all nodes, is the same. As we will see below, this is not always the case in NoSQL databases. Often these databases do not have schemas and whatever data validation rules there are have to be implemented at the client end.

The other two terms in CAP Theorem are more straight-forward:

Availability refers to the ability of the database to serve data to clients. In general, the more redundant nodes a database has, the more available it will be since anyone trying to gather data from a node which is "down" can get the data from other nodes. The downside is that performance can suffer.

Partition Tolerance refers to the ability of the database to find alternate routes through the network to get at data from various nodes should there be breaks in communications. As Gilbert and Lynch (2002) suggest tolerance means:

> No set of failures less than total network failure is allowed to cause the system to respond incorrectly.

As these rules can seem a little strange to users of stand-alone RDBMS, it is worth looking at an example scenario and seeing how CAP plays out. Most readers will be used to online shopping applications. Let us take an example of a simple system that maintains data about the number of candles a hardware supplier has in stock. It is a distributed system in that identical data is stored on two geographically separate networks, Network 1 and Network 2.

We should recognise that the importance of the repressions of what CAP Theorem predicts is dependant upon the sort of application we are dealing with. Any transactional system requires Atomicity—that is, a candle is either purchased or it is not, a bank account is either updated or it is not.

However, as we have seen, this sort of atomicity comes at a cost in terms of resource usage and performance. There may be circumstances where the transactional security needed by banks, or shops, is not needed. Take for example, an online Discussion Board application. Just to keep the example simple we look at a two node system.

Julia goes on to the discussion board and asks a question, Question J. She is logged on through Network 1 (though of course she isn't aware of that). Question J gets replicated to Network 2. Users K, L, M write answers to network 1 over the next few minutes. But only K and L get replicated since the connection between the two networks breaks at the time that M is writing a response.

Now hundreds of other users of the Discussion Board log on, some through Network 1 and some through Network 2. Those using the latter will not see Answer M.

But does it really matter? When you first think about this you might well say "*of course it matters!*". But just think about the users who logged on to Network 1 the second before M sent their answer. They were seeing exactly what users of Network 2 are seeing now. And it could well be that Answer K was the most helpful one, so they are not really any worse off because of it.

If you are still not convinced, ask yourself another question—if you want to know every user will be guaranteed access to exactly the same data, are you willing to pay for the service? Naturally such a consistent database is achievable, but it may well mean that you need to purchase an ACID compliant RDBMS, as opposed to using an open source NoSQL solution. And even if that is your preferred solution, you will then need to be prepared to see a fall in performance since no-one will be able to see Answer M until the partition is removed.

As we see in the next section there are various consistency approaches available, depending upon the application's requirements. Many of the mechanisms used in a RDBMS to ensure there is only ever one valid value stored involve preventing users seeing data by locking rows whilst transactions are under way. For the locked-out user this seems like the database is performing poorly, or worse, is not functioning properly. That user might prefer to see stale data, rather than have to wait, potentially hours, for "actual" data.

For the reader interested in proof, Brewer's CAP theorem is proved in the Gilbert and Lynch (2002) paper; *Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services*.

5.4 Basically Available, Soft State, Eventually Consistent (BASE)

BASE is, in effect, the operating premise for most NoSQL databases, in the same way RDBMS databases apply the ACID rules. Hill et al. (2013) note:

> that we move from a world of certainty in terms of data consistency, to a world where all we are promised is that all copies of the data will, at some point, be the same.

However, even this less certain consistency can be managed in different ways. The eventual consistency can be brought about, for example, by:

Read Repair: When a read operation happens and discovers that some data on a node is stale (that is, it has an earlier time-stamp than that on the other nodes), the outdated data is refreshed.

Delayed Repair: The application will return the value it finds, even though it may be stale, but marks it as needing updating. The timing of that update is in the hands of the controlling, or master database. This is known as weak consistency. It is a performance oriented approach.

What we have seen is that we now have more control over whether we want an absolutely consistent collection of nodes, which a traditional ACID-focused DBMS would automatically provide us, or varying degrees of consistency, with the upside to the loss of consistency being in typically better performance.

Many of the NoSQL databases available today use the BASE design philosophy. Many also allow you to manage the degree of consistency required by any application. We will now move on to look at how this different design philosophy is actually implemented by reviewing some NoSQL tools. They all have different approaches to data storage because they aim to meet different needs, but they share the notion that traditional RDBMS rules and practices are not necessarily always the most appropriate.

5.5 Column-Based Approach

Tables of data stored in rows are such a common part of today's ICT systems that even non-database people recognise the concept. This common understanding is probably also helped by the omnipresence of spreadsheets, also made of rows.

Many readers will already be aware of design and implementation techniques, including normalisation and indexing, which can assist in optimising storage and speeding the return of data. Often, this row-oriented tabular storage is taken for granted. If there are specialist requirements to get data out of a database, for example, looking for values from a particular column, the storage approach remains relational, but tools like views, cubes, and star schema are used to speed the search and retrieval process and assist the user in navigating a complex series of joins.

For many Decision Support applications, examining a column, looking for trends, averages and other statistical analysis is a frequent event. And yet the data is stored so that rows can be read as efficiently as possible, not columns. We must

always remember that the seek and read from disk is one of the slowest aspects of any database system. At the heart of the problem is that the DBMS has to read every row, discarding unwanted information (or at least moving the read position so that it skips unwanted data) to get to the data in the desired column.

There are many relational databases which have optimisation techniques that make this problem seem trivial. And yet, the question remains; if we are retrieving data from columns more frequently than looking at several attributes from a row of data, why don't we store the data in columns instead of rows?

And that is exactly what the column-based approach provides us. The data items for each attribute are stored one after another. The read process is thus made very simple and there are only start-toread and end-reading pointers to worry about and no jumping around to different disk segments to jump over unwanted data (see diagram in Fig. 5.2 showing the same data as in Fig. 5.1).

In the type of system we are discussing the column oriented approach improves read efficiency because there is no need for a row identifier, and packing similar data together allows compression algorithms to reduce the volume of disk storage required. Whilst this does allow for more rapid analysis of those column values, it does mean that the reverse problem is true—if there is a need to create a record (or row of data), it would have to be recreated by using positional information to look up the other data items from columns. This would clearly be very inefficient.

A second slight issue is that best performance, in terms of compression, will happen when similar values are adjacent to each other. If the data being recorded is volatile and many inserts and updates occur, this efficiency will eventually be lost. One insert could result in many rewrites as the data is shuffled to ensure similar values are adjacent to each other.

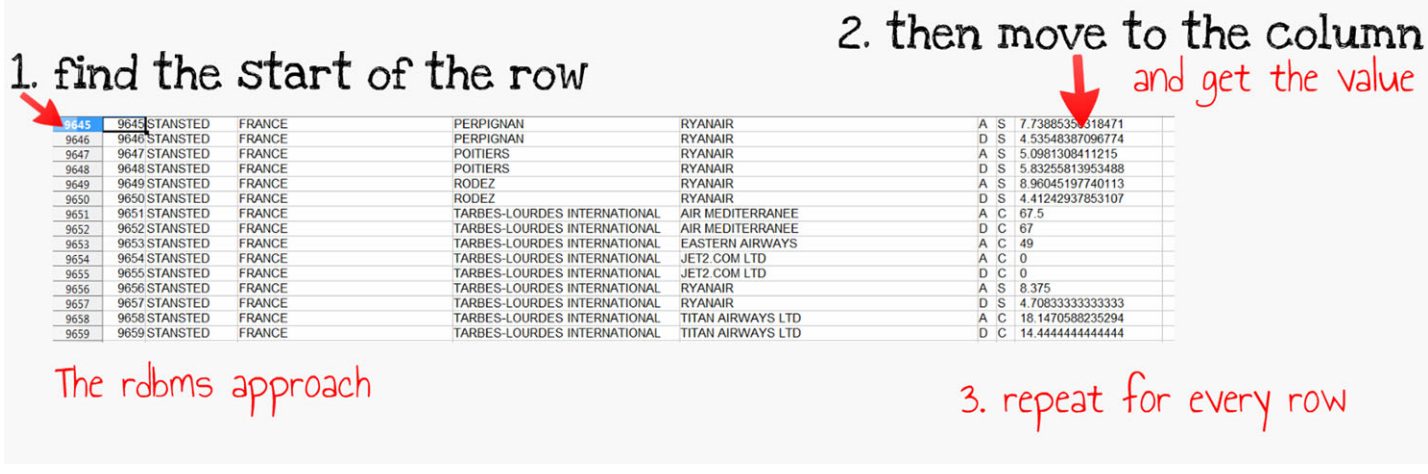
So when would you choose to use a column-based approach? Hill et al. (2013) suggest:

> Column based databases allow for rapid location and return of data from one particular attribute. They are potentially very slow with writing however, since data may need to be shuffled around to allow a new data item to be inserted. As a rough guide then, traditional transactionally orientated databases will probably fair better in a RDBMS. Column based will probably thrive in areas where speed of access to non-volatile data is important, for example in some Decision Support Applications. You only need to review marketing material from commercial vendors to see that business analytics is seen as the key market, and speed of data access the main product differentiator.

To get a feel for the potential of the column-based approach we now go on to use Cassandra. This is only one of the Column-based databases available. You should look at http://nosql-database.org/ to review the others.

5.6 Examples of Column-Based Using Cassandra

The Cassandra data model is very different to RDBMS. Some of the concepts seem alien if you have become very familiar with the relational model. It may help to start by simply thinking of this as a mechanism for storing a key, and a value. This key-value pair storage mechanism is extended by allowing the pairs to be nested. There



9645	9645	STANSTED	FRANCE	PERPIGNAN	RYANAIR	A	S	7.738853318471
9646	9646	STANSTED	FRANCE	PERPIGNAN	RYANAIR	D	S	4.53548387096774
9647	9647	STANSTED	FRANCE	POITIERS	RYANAIR	A	S	5.0981308411215
9648	9648	STANSTED	FRANCE	POITIERS	RYANAIR	D	S	5.83255813953488
9649	9649	STANSTED	FRANCE	RODEZ	RYANAIR	A	S	8.96045197740113
9650	9650	STANSTED	FRANCE	RODEZ	RYANAIR	D	S	4.41242937853107
9651	9651	STANSTED	FRANCE	TARBES-LOURDES INTERNATIONAL	AIR MEDITERRANEE	A	C	67.5
9652	9652	STANSTED	FRANCE	TARBES-LOURDES INTERNATIONAL	AIR MEDITERRANEE	D	C	67
9653	9653	STANSTED	FRANCE	TARBES-LOURDES INTERNATIONAL	EASTERN AIRWAYS	A	C	49
9654	9654	STANSTED	FRANCE	TARBES-LOURDES INTERNATIONAL	JET2.COM LTD	A	C	0
9655	9655	STANSTED	FRANCE	TARBES-LOURDES INTERNATIONAL	JET2.COM LTD	D	C	0
9656	9656	STANSTED	FRANCE	TARBES-LOURDES INTERNATIONAL	RYANAIR	A	S	8.375
9657	9657	STANSTED	FRANCE	TARBES-LOURDES INTERNATIONAL	RYANAIR	D	S	4.70833333333333
9658	9658	STANSTED	FRANCE	TARBES-LOURDES INTERNATIONAL	TITAN AIRWAYS LTD	A	C	18.1470588235294
9659	9659	STANSTED	FRANCE	TARBES-LOURDES INTERNATIONAL	TITAN AIRWAYS LTD	D	C	14.4444444444444

Fig. 5.1 The Row approach

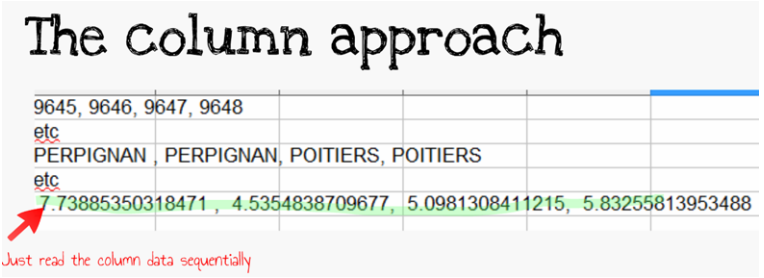


Fig. 5.2 The Column approach

is a section in this chapter about Key-Value stores, so you might want to review that if this is a new concept to you.

5.6.1 Cassandra's Basic Building Blocks

A ***keyspace*** in Cassandra is roughly equivalent to a database schema in a Relational DBMS and it is a group of related column families (see below). Remembering that we are dealing with a database that allows for horizontal scaling, there is an attribute called *placement_strategy* which allows the user to define how to distribute replicas around the nodes they will be using.

A ***Column Family*** is roughly equivalent to a table in a RDBMS. Each such Family is stored in a separate physical file which is sorted into key order. To help reduce disk reads, and therefore improving performance, columns which are regularly accessed together should be kept together, within the same column family.

A ***Column*** is the smallest unit of storage in Cassandra. A standard column is composed of a unique name (key),value and a timestamp. The key identifies a row in a Column Family.

The ***timestamp*** is used to ensure distributed conflict resolution. It is usually defined as the difference between the current time and 00:00:00 UTC on 1 January 1970, also known as Unix epoch. The level of granularity is usually in milliseconds, or microseconds. The timestamp is provided by the client. This can be a problem in cases where the data is volatile and the accuracy of the client timestamp is uncertain. An external time server service can be a solution to this, but it does have performance implications.

A ***SuperColumn*** is a list of columns. Although we have to remember that Cassandra is schema-less, you can think of this as a pot into which to put a variety of columns, rather like a view in a RDBMS. It is a mechanism for containing multiple columns with common look-up values. Again, thinking in relational terms, you might have a transaction with several attributes (Price, date, salesperson, for example). You could have a SuperColumn called TransationID which contains those attributes, each stored as columns.

A SuperColumn does not have a timestamp.

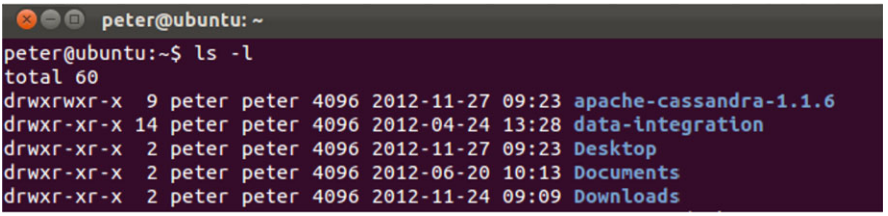


Fig. 5.3 Listing files in the home directory

5.6.1.1 Getting Hands-on with Cassandra

The tutorial material in this chapter demonstrates the use of Cassandra on Ubuntu Linux. Cassandra will also run on other Linux servers, and can be installed on Windows. These notes were created using Ubuntu Linux 11.10 with release **1.1.6** of Cassandra.

We start with the expectation that the reader has basic Unix skills, and has a version of Linux with Cassandra installed on it.

5.6.2 Data Sources

The data in the follow tutorial material is from the UK Civil aviation Authority, made publicly available as Open Data. Many other data sources are available from: http://data.gov.uk/.

5.6.3 Getting Started

Assuming you have followed the default installation of Cassandra, as set out in the appendix, you should open a terminal session and, from you home location, list what is on your file system (Fig. 5.3).

You may see different files and folders, but the one we need to concentrate on here is the apachecassandra-1.1.6 folder, which contains the Cassandra software.

Next, issue the CD command to move to the apache-cassandra-1.1.6\bin folder and see the executable files installed (Fig. 5.4).

In this tutorial we will be using the ***cassandra*** executable, which provides the database functionality and needs to be started, and kept running whilst we work with Cassandra. We will then interact with this programme through the client programme; ***cassandra-cli***. And then we will begin to use the query language; ***cqlsh***.

As we will be regularly using these programmes it may be an idea to create scripts to launch them for us, and place them on the home folder.

Start the text editor and insert the following lines. Once inserted, save to your home folder as something like ***startCass***.

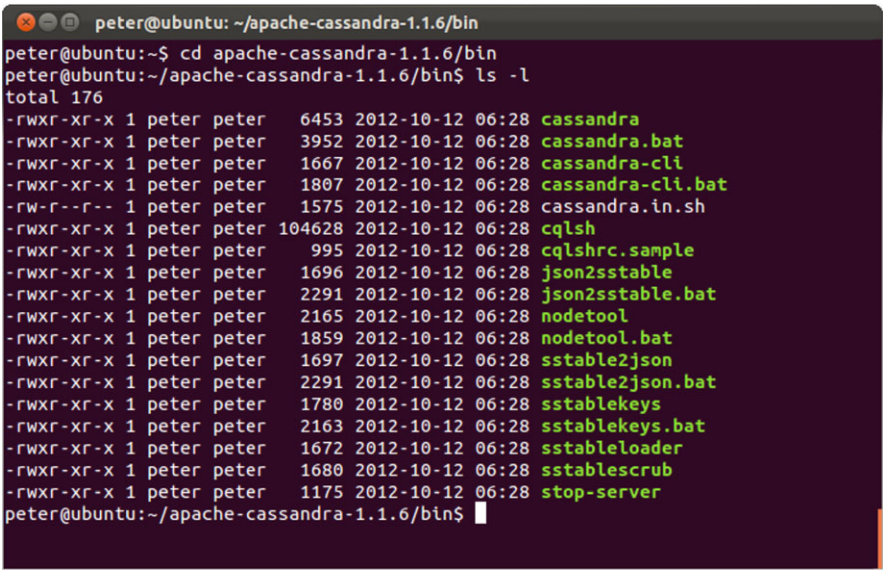


Fig. 5.4 Listing Cassandra related files

```
#!/bin/bash
cd apache-cassandra-1.1.6
cd bin
./cassandra
```

We will also want to start the client often. Using the text editor, create startCassCli with the following content:

```
#!/bin/bash
cd apache-cassandra-1.1.6
cd bin
./cassandra-cli
```

Both of these files need to be turned into executable files. From the $ prompt you will need to type:

```
$chmod 777 startCass
$chmod 777 startCassCli
```

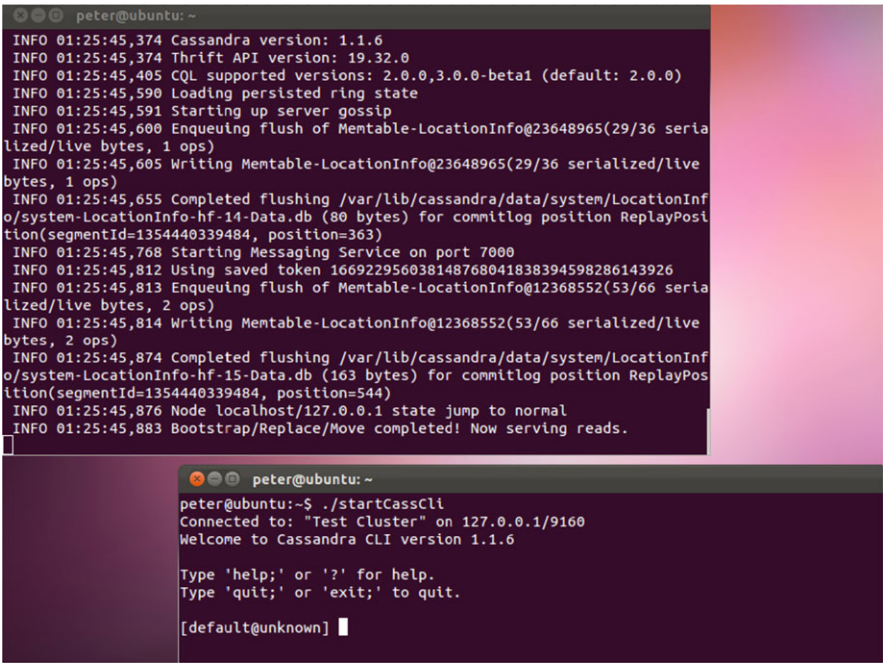


Fig. 5.5 Cassandra running and waiting for input

The first thing we need to do to use Cassandra is run the server. We start it, and then leave it running, probably minimising so the terminal session is out of the way. And then we start the client application.

From the $ prompt in your home folder, run the script we just created above:

```
$./startCass
```

Once the server is started, open a second terminal session. In that one, start the client session:

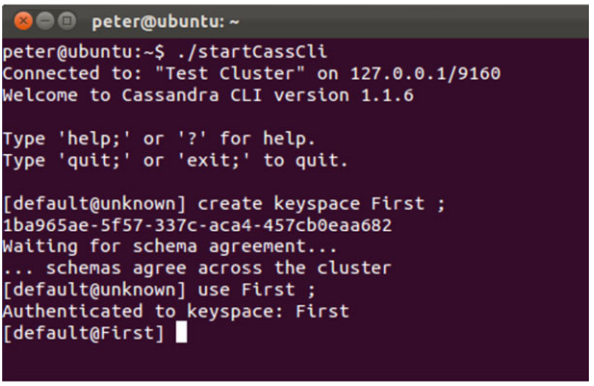
```
$./startCassCli
```

Your desktop should look something similar to the one in Fig. 5.5.

Minimise the terminal window with the server process running. It looks like it has hung as there is no input prompt on it, but do not worry, it is working as it should!

The input prompt in client terminal session is telling us that we have no user (default) or keyspace defined. We can use the default user for now, but we do need to create a keyspace for us to work in (Fig. 5.6).

Fig. 5.6 Creating a keyspace



Note that when creating new objects Cassandra needs to ensure that all instances across the cluster of servers now have this definition in place. As we are currently working in standalone mode, this may seem unhelpful, but we must remember that Cassandra was always designed to work in a distributed environment. After the schema is signalled to be agreed by the cluster we can use the keyspace called First to carry out our work in.

5.6.4 Creating the Column Family

Now we have an area to work in we can begin building structures to store our data in—in much the same way that you can start creating tables in SQL once you have access to a schema. Indeed, a ***Column Family*** is roughly equivalent to a table in a RDBMS and a ***Column*** is the smallest unit of storage in Cassandra. A standard column is composed of a unique name (key),value and a timestamp. The key identifies a row in a Column Family.

The first Column Family we are going to work with is a simple list of UK airlines and their domestic flights. The downloaded.csv is bigger than this, but we have edited it down so that it only contains columns we will be using:

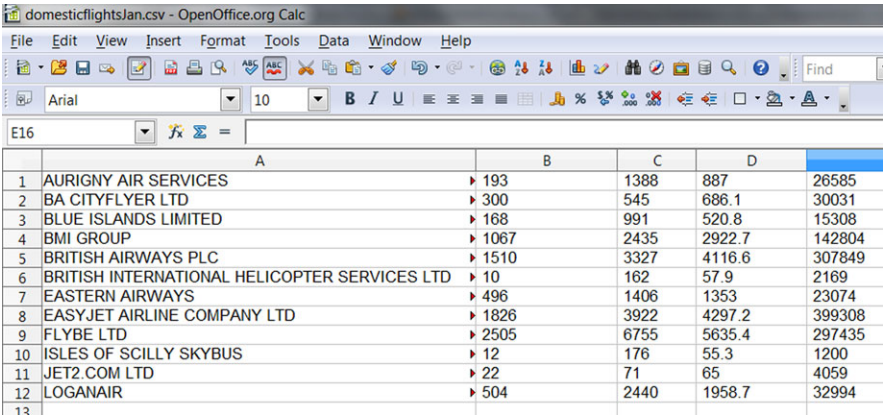
> Airline Name, Km Flown (x1000), Number of Flights, Number of Hours flown, and the Number of Passengers handled.

The data is shown in Fig. 5.7.

There aren't many rows but we are just cutting our teeth at the minute!

We will need a Column Family called DomesticFlights with columns to take this data. We will also need a KEY to uniquely identify each row. Later we will load data straight from a.csv file, but here we are just getting used to the client tool.

With the follow examples you can either type in directly to the $prompt, or cut and paste from this tutorial onto the $prompt.


domesticflightsJan.csv - OpenOffice.org Calc

File Edit View Insert Format Tools Data Window Help

Arial 10

E16

	A	B	C	D	
1	AURIGNY AIR SERVICES	193	1388	887	26585
2	BA CITYFLYER LTD	300	545	686.1	30031
3	BLUE ISLANDS LIMITED	168	991	520.8	15308
4	BMI GROUP	1067	2435	2922.7	142804
5	BRITISH AIRWAYS PLC	1510	3327	4116.6	307849
6	BRITISH INTERNATIONAL HELICOPTER SERVICES LTD	10	162	57.9	2169
7	EASTERN AIRWAYS	496	1406	1353	23074
8	EASYJET AIRLINE COMPANY LTD	1826	3922	4297.2	399308
9	FLYBE LTD	2505	6755	5635.4	297435
10	ISLES OF SCILLY SKYBUS	12	176	55.3	1200
11	JET2.COM LTD	22	71	65	4059
12	LOGANAIR	504	2440	1958.7	32994
13					

Fig. 5.7 Domestic Flights data

```
Create Column Family DomesticFlights
WITH comparator = UTF8Type AND
key_validation_class = UTF8Type AND
column_metadata =
[
      {column_name: airline, validation_class: UTF8Type, index_type: KEYS},
      {column_name: Kms, validation_class: IntegerType},
      {column_name: Flights, validation_class: IntegerType},
      {column_name: Hrs, validation_class: FloatType},
      {column_name: Pass, validation_class: IntegerType}
];
```

Things to note from the above Column Family creation are that:

The **comparator** is required by Cassandra because it is needs to know the order in which to store columns.

Key_validation_class tells Cassandra what datatype the key is going to be. We will be using Airline so we have chosen UTF8, which is a string of characters and may have been described as a varchar in a RDBMS.

The **validation class** is similar to the definition of a datatype in a relational database schema and provides, as its name suggests, the opportunity to validate data on input, only allowing data of the appropriate datatype to be written.

5.6.5 Inserting Data

To insert data we can use the SET command. Here are two examples:

```
set DomesticFlights['Aurigny Air Services']['Kms'] = 193; set
DomesticFlights['Aurigny Air Services']['Flights'] = 1388; set
DomesticFlights['Aurigny Air Services']['Hrs'] = 887; set
DomesticFlights['Aurigny Air Services']['Pass'] = 26585;

set DomesticFlights['BA CityFlyer']['Kms'] = 300;
set DomesticFlights['BA CityFlyer']['Flights'] = 545;
set DomesticFlights['BA CityFlyer']['Hrs'] = 686;
set DomesticFlights['BA CityFlyer']['Pass'] = 30031;
```

After you have carried this out your terminal should look something like Fig. 5.8.

5.6.6 Retrieving Data

To retrieve information we can use LIST or GET depending upon if we know the Airline we want to return data for. Try these two queries:

```
LIST DomesticFlights;
```

```
GET DomesticFlights['BA CityFlyer'];
```

Outputs are shown in Fig. 5.9.

However, unlike data in a RDBMS column, where you would just use the WHERE clause on any column, you can't retrieve data by searching for specific data value in Cassandra unless the data is indexed.

Try this:

```
GET DomesticFlights where Kms = 193;
```

You will get an error message back telling you that there are *No indexed columns present*. We can make this query work by adding a secondary index. This also demonstrates the use of the UPDATE command to change Column Family meta-data.

```
peter@ubuntu: ~
Type 'quit;' or 'exit;' to quit.

[default@unknown] use First ;
Authenticated to keyspace: First
[default@First] Create Column Family DomesticFlights
...     WITH comparator = UTF8Type AND
...     key_validation_class=UTF8Type AND
...     column_metadata =
...       [
...         {column_name: airline, validation_class: UTF8Type, index_type: KEYS}
,
...         {column_name: Kms, validation_class: IntegerType},
...         {column_name: Flights, validation_class: IntegerType},
...         {column_name: Hrs, validation_class: FloatType},
...         {column_name: Pass, validation_class: IntegerType}
...       ];
694d7497-d45e-3032-9b3d-5d2b39107022
Waiting for schema agreement...
... schemas agree across the cluster
[default@First]
[default@First] set DomesticFlights['Aurigny Air Services']['Kms'] = 193 ;
Value inserted.
Elapsed time: 27 msec(s).
[default@First] set DomesticFlights['Aurigny Air Services']['Flights'] = 1388 ;
Value inserted.
Elapsed time: 1.19 msec(s).
[default@First] set DomesticFlights['Aurigny Air Services']['Hrs'] = 887 ;
Value inserted.
Elapsed time: 1.94 msec(s).
[default@First] set DomesticFlights['Aurigny Air Services']['Pass'] = 26585 ;
Value inserted.
Elapsed time: 3.12 msec(s).
[default@First]
[default@First] set DomesticFlights['BA CityFlyer']['Kms'] = 300 ;
Value inserted.
Elapsed time: 1.54 msec(s).
[default@First] set DomesticFlights['BA CityFlyer']['Flights'] = 545 ;
Value inserted.
Elapsed time: 1.03 msec(s).
[default@First] set DomesticFlights['BA CityFlyer']['Hrs'] = 686 ;
Value inserted.
Elapsed time: 1.34 msec(s).
[default@First] set DomesticFlights['BA CityFlyer']['Pass'] = 30031 ;
Value inserted.
Elapsed time: 2.44 msec(s).
```

Fig. 5.8 Inserting data to Cassandra

```
UPDATE COLUMN FAMILY DomesticFlights
WITH comparator = UTF8Type AND
key_validation_class = UTF8Type AND
column_metadata =
  [
    {column_name: airline, validation_class: UTF8Type, index_type: KEYS},
    {column_name: Kms, validation_class: IntegerType, index_type: KEYS},
    {column_name: Flights, validation_class: IntegerType},
```

```
[default@First] LIST DomesticFlights ;
Using default limit of 100
Using default column limit of 100
-------------------
RowKey: BA CityFlyer
=> (column=Flights, value=545, timestamp=1354875194019000)
=> (column=Hrs, value=686.0, timestamp=1354875194033000)
=> (column=Kms, value=300, timestamp=1354875194010000)
=> (column=Pass, value=30031, timestamp=1354875201897000)
-------------------
RowKey: Aurigny Air Services
=> (column=Flights, value=1388, timestamp=1354875193983000)
=> (column=Hrs, value=887.0, timestamp=1354875193991000)
=> (column=Kms, value=193, timestamp=1354875193958000)
=> (column=Pass, value=26585, timestamp=1354875194001000)

2 Rows Returned.
Elapsed time: 26 msec(s).
[default@First] GET DomesticFlights['BA CityFlyer'] ;
=> (column=Flights, value=545, timestamp=1354875194019000)
=> (column=Hrs, value=686.0, timestamp=1354875194033000)
=> (column=Kms, value=300, timestamp=1354875194010000)
=> (column=Pass, value=30031, timestamp=1354875201897000)
Returned 4 results.
Elapsed time: 59 msec(s).
[default@First]
```

Fig. 5.9 Sample LIST output

```
    {column_name: Hrs, validation_class: FloatType},
    {column_name: Pass, validation_class: IntegerType}
];
```

Now try the same GET command and you should get an answer back this time.

5.6.7 Deleting Data and Removing Structures

To remove a particular column from one row we use the **DEL** command. So, to remove the **Hrs** column from the BA CityFlyer row we type:

```
del DomesticFlights['BA CityFlyer']['Hrs'];
```

To remove the entire row we type:

```
del DomesticFlights['BA CityFlyer'];
```

If we want to remove the entire Column Family, or Keyspace, we use the **Drop** command:

```
drop column family DomesticFlights;
```

Try each of the above, followed by **LIST** to see the effects.

5.6.8 Command Line Script

Just as Oracle allows you to runs SQL and PL/SQL scripts from the operating system, so does Cassandra. This allows database administrators to automate, replicate and schedule particular tasks.

Here we will create a script that will drop our DomesticFlights Family and create another one called DomFlights, inserting some rows, and then querying the data.

Save to the Cassandra directory the following, using your editor, naming it egscript.txt:

```
Use First; Drop Column Family DomesticFlights;
Create Column Family DomFlights
WITH comparator = UTF8Type AND
key_validation_class = UTF8Type AND
column_metadata =
   [
      {column_name: airline, validation_class: UTF8Type, index_type: KEYS},
      {column_name: Kms, validation_class: IntegerType},
      {column_name: Flights, validation_class: IntegerType},
      {column_name: Hrs, validation_class: FloatType},
      {column_name: Pass, validation_class: IntegerType}
   ];
set DomFlights['Aurigny Air Services']['Kms'] = 193;
set DomFlights['Aurigny Air Services']['Flights'] = 1388;
set DomFlights['Aurigny Air Services']['Hrs'] = 887;
set DomFlights['Aurigny Air Services']['Pass'] = 26585;

set DomFlights['BA CityFlyer']['Kms'] = 300;
set DomFlights['BA CityFlyer']['Flights'] = 545;
set DomFlights['BA CityFlyer']['Hrs'] = 686;
set DomFlights['BA CityFlyer']['Pass'] = 30031;
LIST DomFlights;
```

Once you have saved that file, from the Linux $prompt type this command to run it:

```
./bin/cassandra-cli -host localhost -port 9160 -f egscript.txt
```

You should now have the same data that we had before, but in a different Column Family.

5.6.9 Shutdown

At the end of our busy day we may need to bring down the Cassandra Client. To do this we simply use the quit; or exit; command from the client terminal session.

If we want to shutdown the Cassandra server we use the CTRL + C keys. BUT do not do this unless you have finished your session since restarting can need a system reboot.

5.7 CQL

In earlier versions of Cassandra the client interface we have just been using was the only way to interact with the database without writing code, for example Java, to call the APIs. Many users wanting to experiment with the new NoSQL databases found this a problem. Most database professionals know SQL well and an SQL-like language was needed to remove the hurdle of forcing potential users to learn a new language.

In Cassandra 0.8 we saw the birth of Cassandra Query Language (CQL). It was deliberately modelled on standard SQL, but naturally the commands are mapped back to the column orientated storage model.

An example of this is that you can actually use the SQL command CREATE TABLE test…, despite the fact that there is no such structure as a table in Cassandra. The above code generates a Column Family, and the two names can be used interchangeably.

Before we use the CQL environment interactively, as promised earlier, let us use the enhanced script facility available using CQL. In the following example we are going to create a Keyspace, Column Family and then insert data from a CSV file called domDataOnly.csv. It contains the data displayed in the spreadsheet at the start of these notes.

Create a script file using your editor and save it as cqlcommands in the Cassandra directory. It should have the following content, which you should make sure you understand before copying it:

```
CREATE KEYSPACE Flights WITH strategy_class = SimpleStrategy
AND strategy_options:replication_factor = 1;

use Flights;

create ColumnFamily FlightDetails
(airline varchar PRIMARY KEY,
Kms int,
Noflights int,
Hrs float,
Pass int);

copy FlightDetails (airline, Kms, Noflights, Hrs, Pass) from 'domDataOnly.csv';

select * from FlightDetails;
```

Note that the syntax is very SQL-like, even down to the commands ending in a semicolon. However, it also maintains its connection with the traditional CassandraCli interface commands. Some details are slightly different. The datatypes for the columns, for example are different. They are also case sensitive in CQL. **Int** will not be allowed, for example, but **int** would.

Another thing to point out is that, unlike the CassandraCli environment, you do have to define the *Replication Strategy*. Cassandra was built from the start as a multiple node database. Copying data to different nodes, known as replication, helps to improve availability and fault tolerance. More detail is available here: http://www.datastax.com/docs/1.1/cluster_architecture/replication#replication-strategy.

The advice in the online guide is that the **replication factor** (the number of times each row is replicated) should be less than or equal to the number of nodes being used to hold the replicated data. As this worked example expects the user to be on a standalone, single node version, we have set the strategy to SimpleStrategy and the replication factor to 1—in other words, only one copy will exist on the single node.

The line that saves us manually inserting the data is the **copy** command. We are telling CQL which Column Family is receiving the data, and then into which Columns the data should go. These columns need to be in the same order that they appear in the CSV file.

To run the script file from the Cassandra directory type this at the $prompt:

```
./bin/cqlsh < 'cqlcommands'
```

You don't get reassurance messages about the creates, but the select shows that the data exists. Your terminal should look like in Fig. 5.10.

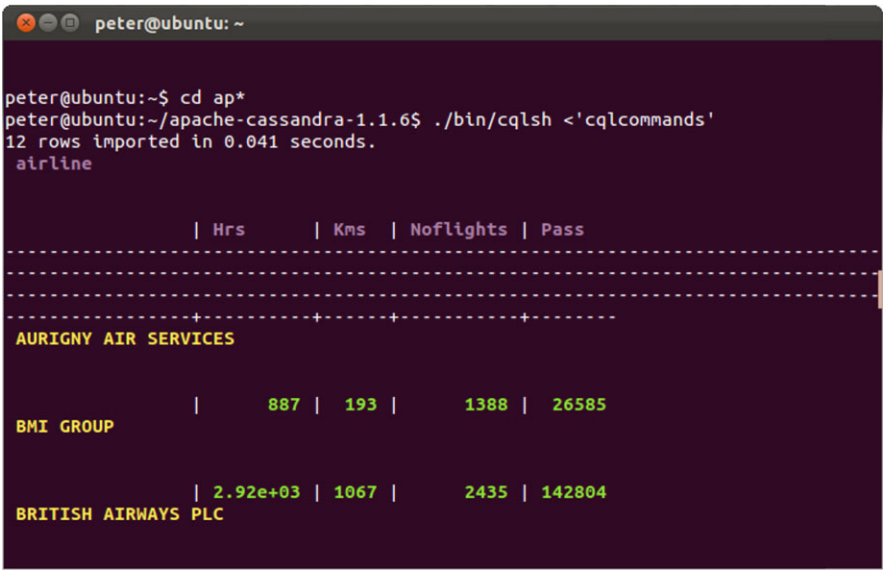


Fig. 5.10 Sample CQL output

5.7.1 Interactive CQL

Just as with the older CassandraCli, you could launch CQL by creating a script like this:

```
#!/bin/bash
cd apache-cassandra-1.1.6
cd bin
./cqlsh
```

(Don't forget to do a Chmod 777.)

Once you have issued the USE command you can begin querying the data we have just input. In this example we see a SELECT using the WHERE clause, and the use of COUNT(∗). These will all be straight forward to anyone who has some SQL experience (Fig. 5.11).

We still need to apply the same rules that we discovered using CassandraCli when retrieving data—there needs to be an index on a column to search on that column. The example above worked because we defined *airline* as the KEY and there is therefore an index created for it. Let's try to access data by the Kms column. You will note that the addition of a secondary index allows this to happen (Fig. 5.12).

```
peter@ubuntu: ~
peter@ubuntu:~$ ./startCQL
Connected to Test Cluster at localhost:9160.
[cqlsh 2.2.0 | Cassandra 1.1.6 | CQL spec 2.0.0 | Thrift protocol 19.32.0]
Use HELP for help.
cqlsh> use Flights ;
cqlsh:Flights> select * from FlightDetails where airline = 'LOGANAIR' ;
 airline  | Hrs      | Kms | Noflights | Pass
----------+----------+-----+-----------+-------
 LOGANAIR | 1.96e+03 | 504 |      2440 | 32994

cqlsh:Flights> select count(*) from FlightDetails ;
 count
-------
    12
```

Fig. 5.11 Interactive CQL output

```
peter@ubuntu: ~
peter@ubuntu:~$ ./startCQL
Connected to Test Cluster at localhost:9160.
[cqlsh 2.2.0 | Cassandra 1.1.6 | CQL spec 2.0.0 | Thrift protocol 19.32.0]
Use HELP for help.
cqlsh> use Flights ;
cqlsh:Flights> select * from FlightDetails where Kms = 504 ;
Bad Request: No indexed columns present in by-columns clause with "equals" operator
cqlsh:Flights> create index K_ind on FlightDetails(Kms) ;
cqlsh:Flights> select * from FlightDetails where Kms = 504 ;
 airline  | Hrs      | Kms | Noflights | Pass
----------+----------+-----+-----------+-------
 LOGANAIR | 1.96e+03 | 504 |      2440 | 32994

cqlsh:Flights>
```

Fig. 5.12 Using indexes in CQL

By accessing the material available on the web, you could experiment further with this data. Here is a good starting place: http://www.datastax.com/docs/1.0/index and CQL specific material is here: http://www.datastax.com/docs/1.0/references/cql/index.

5.7.2 IF You Want to Check How Well You Now Know Cassandra ...

Using this datachimps dataset and Cassandra, answer the question:
How many airports are there in Great Britain north of Heathrow?

You can do this with what we have just learned. You need to know that the country code is GB and that Heathrow's code is LHR.

You should also be aware that CQL does not allow subqueries in the way that SQL does so this process will have to be a two stage one.

If you get the answer without looking for help, award yourself a pat on the back! If you need a pointer or two, have a look at the end of the chapter!

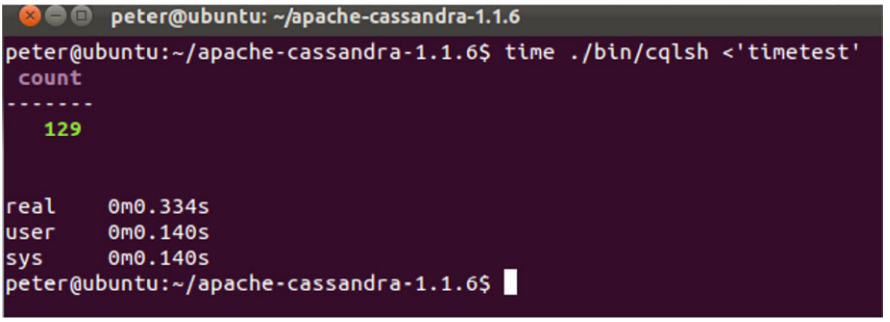


Fig. 5.13 Using the "time" command

5.7.3 Timings

Now we have enough information to be able to begin comparing Cassandra to other databases. You could, for example normalise the AirportLocations data and load it into two related tables (Airport and Country) using MySQL. Using the Linux TIME command you could do some comparisons of load time, and retrieval time.

Figure 5.13 shows an example of TIME being used with a CQL command called timetest which contains this code:

```
use Flights;
select count(*) from Airports where CountryCode = 'GB' and Lat > 51;
```

You could now try to answer the same question in MySQL, or any other environment you choose.

5.8 Document-Based Approach

Many of us are taught that well structured, normalised data is the only form of good database design when we first encounter large scale database systems. Indeed this is true for many systems. However, even in RDBMS design there is sometimes a case for denormalising the data for performance.

The document approach takes this even further. It is schemaless, meaning there can be no "correct" design. The application developer using MongoDB, for example, is therefore responsible for data quality issues, rather than relying on the centralised constraints typical of RDBMS.

Why remove these seemingly sacrosanct rules? Well, performance and flexibility are probably two of the main reasons. A large part of a RDBMS's processing time is spent ensuring that the data entered is correct and so not having to check will, it is suggested, speed up write operations. On top of that, adding fields to MongoDB is a

relatively trivial task—something which often isn't the case in an RDBMS system. When you have ill-structured data to store this flexibility can be a great advantage.

As Hill et al. (2013) put it:

> Many of the document-centric databases don't allow data to be locked in the way that is required for atomic transactions in RDBMS systems. Since locking is a significant performance overhead this enables them to claim performance advantages over traditional databases. The downside, of course, is that some applications absolutely require secure transactional locking mechanisms. Document-oriented is probably not the right vehicle for highly structured data in a transaction dependant environment, as occurs in many OLTP systems.
>
> The performance advantages that accrue from document-centric implementations mean that they are often used when there are large volumes of semi-structured data, such as webpage content, comment storage, or event logging. They may well also support sharding (spreading a table's rows) of data across multiple nodes, again as a performance device.

There is a growing number of Document Databases. Besides MongoDB, for example, there is CouchDB, which, like MongoDB, is open source. Both databases provide APIs for many programming languages, although they have their own inbuilt client environments as well.

They are both written with distributed data handling at their core. They both support the idea of *Sharding*, where data is spread across a number of nodes in what is also sometimes called "horizontal partitioning" to allow for greater scalability. The examples we use here are stand-alone, but when you review the architecture, bear in mind the end product is often to be expected to run in a multi-node environment.

Sharding is a Shared Nothing approach to distributed databases. Each node has its own instance of the database running. When well implemented this allows high levels of parallel processing when searching for data. However, there would be a risk that the whole database (the sum of the shards) would become invalid if one node failed. For that reason these databases will replicate shards to provide redundancy. This in turn needs high powered replication processes to always ensure all shards are always available.

Document databases can be complex. A document can contain arrays and subdocuments. Using JSON-like notation, here are two valid documents which could be used in these databases. Note how the two "documents" do not have the same structure and yet can be stored in the same collection.

```
{
name: { First: "Penelope", Surname: "Pitstop" },
Birthday: new Date("Jun 23, 1912"),
RacesWon: [ "Alaska", "Charlottesville"]
}
{
name: { First: "Peter", Surname: "Pefect" },
Birthday: new Date("May 23, 1940"),
Hometown: "Miami"
Favourite: "Penelope"
}
```

These two pots of information do not look similar enough to become rows in a standard RDBMS. They don't share all fields for a start. And yet a document-based approach allows that sort of flexibility. The name field in both cases is a container for other fields. This is called document embedding. Embedded documents can be complex. RacesWon, however, with its square brackets, is an array.

5.8.1 Examples of Document-Based Using MongoDB

MongoDB chooses to store its data following the JavaScript Object Notation (JSON) rules. JSON is a data-interchange format. You can read about it at the JSON site: http://www.json.org/ For speed this human-readable format is turned into Binary format and stored as BSON.

Architecturally RDBMS users coming to Mongo might find it helpful to think of these comparisons:

- A Mongo Database is a collection of related data, just as in an RDBMS
- A Mongo Collection is a container for documents. It can be thought of as RDBMS table-like
- A Mongo Document is rather like a RDBS Row
- A Mongo Field is rather like a RDBS Column
- A Mongo embedded document is rather like a RDBMS Join
- A Mongo Primary key is the same as a RDBMS Primary Key
- A Mongo Secondary index is the same as a RDBMS Secondary Index

5.8.1.1 Getting Hands-on with MongoDB

The tutorial material in this chapter demonstrates the use of MongoDB on Ubuntu Linux. Mongo will also run on other Linux servers, and can be installed on Windows. These notes were created using Ubuntu Linux 11.10 with release **2.2.2** of MongoDB.

We start with the expectation that the reader has basic Unix skills, and has a version of Linux with MongoDB installed on it.

5.8.2 Data Sources

The data in the follow tutorial material is from the UK Civil aviation Authority, made publicly available as Open Data. Many other data sources are available too, from: http://data.gov.uk/.

5.8.3 Getting Started

Assuming you have followed the default installation of MongoDB, as set out on the MongoDB site: http://docs.mongodb.org/manual/tutorial/install-mongodb-on-ubuntu/ you should open a terminal session and, from your home location, create a script file that we will use to launch MongoDB. It will contain a single line:

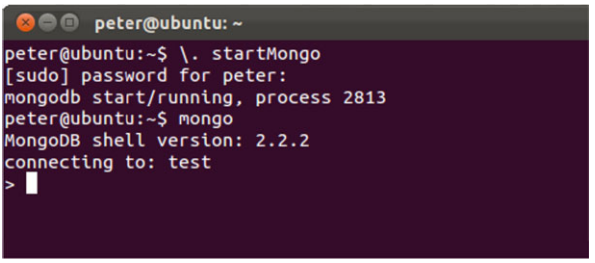


Fig. 5.14 Starting and connecting to Mongo

```
sudo service mongodb start
```

Save this as startMongo. Then when you call it, it will ask for your root password. After entering the password the database server will be running, waiting for connections from a client. Here we will be using the Mongo client that comes with the installation. In the screenshot (Fig. 5.14) you see us starting the server and then starting a client session by issuing the Mongo command.

You will be connected to the default database called "test". As we will see below, the Mongo client is a JavaScript environment and gives the user access to all standard JavaScript functionality.

The service will run until you stop the service:

```
sudo service mongodb stop
```

5.8.4 Navigation

If you need to know what database you are using you can issue the **db** command. In the screenshot (Fig. 5.15) we move to a different db, creating it as we do, by issuing the **use** command—**use** either changes your working database to an existing database, or creates one of the name you provide. We use a call to dropDatabase to remove a database.

If you need further help there is a sizeable manual available online: http://docs.mongodb.org/manual/.

5.8.5 Creating a Collection

The first Collection we are going to work with is a simple list of UK airlines and their domestic flights. The downloaded.csv is bigger than this, but we have edited it down so that it only contains columns we will be using:

Fig. 5.15 Using and dropping a database

domesticflightsJan.csv - OpenOffice.org Calc

File Edit View Insert Format Tools Data Window Help

Arial 10

E16

	A	B	C	D	
1	AURIGNY AIR SERVICES	193	1388	887	26585
2	BA CITYFLYER LTD	300	545	686.1	30031
3	BLUE ISLANDS LIMITED	168	991	520.8	15308
4	BMI GROUP	1067	2435	2922.7	142804
5	BRITISH AIRWAYS PLC	1510	3327	4116.6	307849
6	BRITISH INTERNATIONAL HELICOPTER SERVICES LTD	10	162	57.9	2169
7	EASTERN AIRWAYS	496	1406	1353	23074
8	EASYJET AIRLINE COMPANY LTD	1826	3922	4297.2	399308
9	FLYBE LTD	2505	6755	5635.4	297435
10	ISLES OF SCILLY SKYBUS	12	176	55.3	1200
11	JET2.COM LTD	22	71	65	4059
12	LOGANAIR	504	2440	1958.7	32994
13					

Fig. 5.16 Domestic flights data

Airline Name, Km Flown (x1000), Number of Flights, Number of Hours flown, and the Number of Passengers handled.

The data is shown in Fig. 5.16.

This is the same data we used in the Cassandra tutorial if you have done that already. It is not that appropriate for a real document-based application but will get us started in using the tool, and we can look at data collection types later.

MongoDB is a document oriented storage system. Each document includes one or more key-value pairs. Each document is part of a collection and in a database but more than one collection can exist. A collection can be seen as similar to a table in a RDBMS.

Unlike relational databases, MongoDB is schemaless and we do not need to define the datatypes for our incoming values. This can come as a bit of a shock to people with an RDBMS background!

We are going to store our documents in a collection called Flights. As with the db name, we do not have to create the collection before we call it. As we see below, if we issue the command to insert and the collection does not yet exist, Mongo creates it. It can do this because there is no schema to worry about.

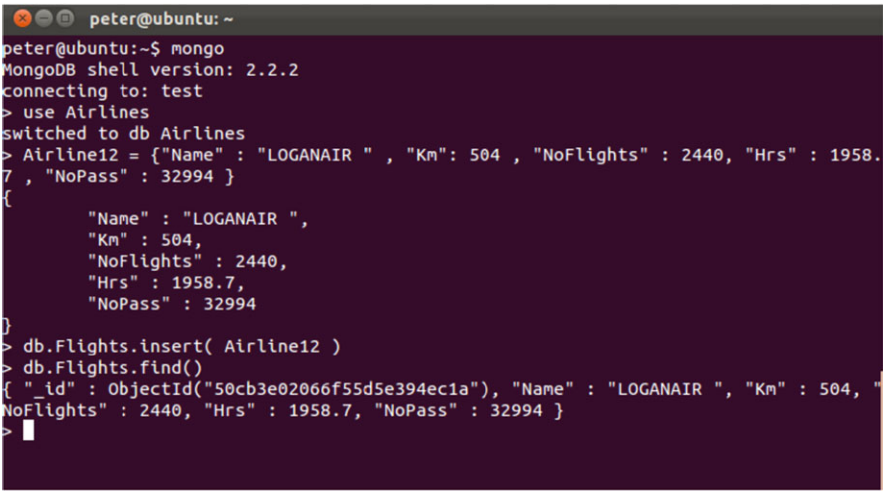


Fig. 5.17 Adding data to Mongo

5.8.6 Simple Inserting and Reading of Data

As we are working in a JavaScript environment we can create variables in memory which we pass to Mongo as parameters when we want to create documents.

In the screenshot (Fig. 5.17) you will see we have:

1. Created a database called Airlines
2. Created a Javascript variable called Airline12 to store information about Loganair.
3. Checked the feedback screen to make sure the variable is correct
4. Used the insert() method of db to add the document to the collection
5. Checked that the data is saved by calling the find() method with no parameters to list all the collection contents

Note the lack of positive feedback from the system after the insert. And the fact that MongoDB has created its own ObjectID to uniquely identify the document.

We can now use find() to look for specific values in the collection. Try this, for example to return the same document which has a Km value of 504:

```
db.Flights.find( {“Km”: 504})
```

Naturally we ought to have a few more rows to make finds more interesting, so cut and paste, or type a few, of the following:

```
Airline1 = { "Name": "AURIGNY AIR SERVICES", "Km": 193, "NoFlights":
1388, "Hrs": 887, "NoPass": 26585 }
db.Flights.insert( Airline1 )

Airline2 = { "Name": "BA CITYFLYER LTD", "Km": 300, "NoFlights":
545, "Hrs": 686.1, "NoPass": 30031 }
db.Flights.insert( Airline2 )

Airline3 = { "Name": "BLUE ISLANDS LIMITED", "Km": 168, "NoFlights":
991, "Hrs": 520.8, "NoPass": 15308 }
db.Flights.insert( Airline3 )

Airline4 = { "Name": "BMI GROUP", "Km": 1067, "NoFlights": 2435,
"Hrs": 2922.7, "NoPass": 142804 }
db.Flights.insert( Airline4 )

Airline5 = { "Name": "BRITISH AIRWAYS PLC", "Km": 1510, "NoFlights":
3327, "Hrs": 4116.6, "NoPass": 307849 }
db.Flights.insert( Airline5 )

Airline6 = { "Name": "BRITISH INTERNATIONAL HEL", "Km": 10,
"NoFlights": 162, "Hrs": 57.9, "NoPass": 2169 }
db.Flights.insert( Airline6 )

Airline7 = { "Name": "EASTERN AIRWAYS", "Km": 496, "NoFlights":
1406, "Hrs": 1353, "NoPass": 23074 }
db.Flights.insert( Airline7 )

Airline8 = { "Name": "EASYJET AIRLINE COMPANY L", "Km": 1826,
"NoFlights": 3922, "Hrs": 4297.2, "NoPass": 399308 }
db.Flights.insert( Airline8 )

Airline9 = { "Name": "FLYBE LTD", "Km": 2505, "NoFlights": 6755,
"Hrs": 5635.4, "NoPass": 297435 }
db.Flights.insert( Airline9 )

Airline10 = { "Name": "ISLES OF SCILLY SKYBUS", "Km": 12, "NoFlights":
176, "Hrs": 55.3, "NoPass": 1200 }
db.Flights.insert( Airline10 )

Airline11 = { "Name": "JET2.COM LTD", "Km": 22, "NoFlights": 71,
"Hrs": 65, "NumPass": 4059 }
db.Flights.insert( Airline11 )
```

Note: all these inserts work, but look closely at the Airline11 row. Instead of NoPass, we have NumPass as the field name. This means, for example, that this would return nothing:

```
db.Flights.find({NoPass:4059})
```

This is one of the issues with working without a schema. The overhead of having to check that this was a valid field name would probably mean a relational database would be slower to insert, but the trade off is that ***you*** have to be responsible for the quality of the data.

We can remove the incorrect document:

```
db.Flights.remove({NumPass:4059})
```

Now let us try to insert something else that may not, at first, feel right to a RDBMS person:

Try creating this variable and then inserting it:

```
another = { "Name": "Metal Bird", "Km": 112, "NoFlights": 72, "Hrs": 165,
"Wings": 2, "Animals": "Elephants/Hippos" }
```

As we might expect after the previous example, the fact that the 5th Field is called "Wings" rather than "NoPass" does not worry Mongo. But then we add an extra field called "Animals" which does not appear in any other document. Again, this just emphasises the schemaless nature of MongoDB.

5.8.7 More on Retrieving Data

To find out what collections there are in a database, you use the following method:

```
db.getCollectionNames();
```

We can perform some aggregation on the data we now have in the database using some of the methods provided.

The first thing we might like to do is count the number of documents we have stored, which we call the count() method to do:

```
db.Flights.count()
```

Then we might want to count particular items. For example, how many of the Airlines in the list fly more than 1000k Kms?

```
db.Flights.count({ Km: { $gt: 1000 } })
```

So we pass a parameter to the count method which Mongo has to evaluate. Note the syntax, with the use of "{}" pairs to separate out the distinct elements of the expression. $gt means *greater than*.

We saw earlier that.find() can be used to find specific values. These searches can be ANDed, again with the use of "{}". Have a look at the example below and see if you can work out what will be returned. Note the use of "[]" with the $and to pass an array of several expressions.

```
db.Flights.find({ $and: [ { Km: {$gt: 2000} }, { NoPass: {$gt:140000} } ] } )
```

There is also an $or operator that works in a similar way. Will the following generate more or less rows?

```
db.Flights.find({ $or: [ { Km: {$gt: 2000} }, { NoPass: {$gt:140000} } ] } )
```

Looking for values from a list is also possible, using the $in operator. In this example we look for two values of Km.

```
db.Flights.find( { Km: { $in: [ 300, 496 ] } } )
```

If you want your output sorted you can use the $sort operator. It needs a sort type to be passed of either -1 (descending) or 1 (ascending).

```
db.Flights.find().sort({Km: -1})
```

A little confusingly you would need to use this sort to discover the Min and Max values in a field since Mongo uses Min and Max in a very different way elsewhere. Here we answer the question *Which Airline flew the most Kms?*

```
db.Flights.find().sort({Km: -1}).limit(1)
```

So we sort descending but limit printed output to one row, answering the question in doing so.

Remembering that we can have a variety of field names, we will occasionally need to check if a field actually exists. Using the final insert from earlier for "Metal Bird", we could ask for all documents which contain a field called "Animals":

```
db.Flights.find( { Animals: { $exists: true } } )
```

5.8.8 Indexing

MongoDB allows us to add indexes to fields. As with any database we need to be aware that whilst indexes can speed the retrieval of data, they are a heavy processing overhead during Inserts and Updates, so we should use them with caution.

Indexes in Mongo are stored in B-Trees (see Chap. 11, Performance, for a discussion of index types) and can either be placed on one field, or on multiple fields to form a compound index. They are held on collections. The MongoDB manual suggests that:

> In general, you should create indexes that support your primary, common, and user-facing queries. Doing so requires MongoDB to scan the fewest number of documents possible.

Lets assume we decide we will query the collection often using the Km in our queries. To have secondary index on that field we would issue this command (the 1 indicating Ascending)

```
db.Flights.ensureIndex( { Km: 1 } )
```

Queries such as our earlier one to count documents where Km is above a level would automatically use this index. You can also force the optimiser to use the index, as with the example below where we hint to use the index that has been created on the Km field. Note how the data output is sorted in Km order when you use the index.

```
db.Flights.find().hint( { Km: 1 } )
```

You can review the indexes available with the getIndexes method on the collection:

```
db.Flights.getIndexes()
```

5.8.9 Updating Data

We can either update the values of a particular field, or alter the fields themselves using the Update() method.

After trying each of these examples, use the find method to check your entry has worked.

Assuming we made a mistake with the entry in the Animals field, we could locate the document using the unique identifier, or as a result of a query, and then pass the change as a parameter. Below, we find the first document for which the Km = 11, and then change the value in the Animals field "Elephants and Badgers":

```
db.Flights.update( { Km: 11 }, { $set: { Animals: "Elephants and
Badgers" }})
```

If we wanted to change the name of fields we could do this by replacing $set with $rename, as below;

```
db.Flights.update( { Km: 112 }, { $rename: { Animals: "Creatures" }})
```

Finally, we may want to add a field, with a value in it:

```
db.Flights.update ( { Km: 112 },{ $set: { Rivers: "Don and Ouse" }})
```

In this case, because there is no field called "Rivers", Mongo creates the field and then adds the value "Don and Ouse".

5.8.10 Moving Bulk Data into Mongo

CSVs can be input directly using the MongoImport utility This utility can also cope with JSON data. The utility is called from the $prompt, not within the Mongo client.

Here is an example of how to import a CSV file called AirportLocations.csv, creating a Database called Airports and a Collection called AllAirports. Note that we have told MongoImport that the first row contains field names in the header. Parameters are identified with a leading double minus sign (Fig. 5.18).

The CSV file being input is shown in Fig. 5.19.

5.9 IF You Want to Check How Well You Now Know MongoDB ...

Using this datachimps dataset and Mongodb, answer the question:

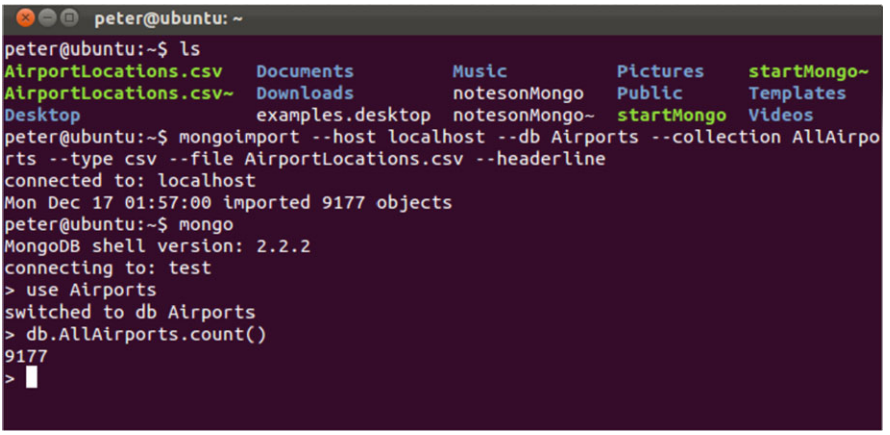


Fig. 5.18 Bulk imports

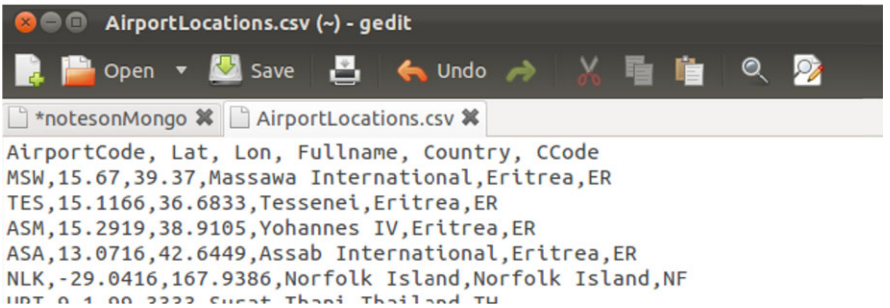


Fig. 5.19 The dataset used for the bulk import

How many airports are there in Great Britain north of Heathrow?

You can do this with what we have just learned. You need to know that the country code is GB and that Heathrow's code is LHR.

If you get the answer without looking for help, award yourself a pat on the back! If you need a pointer or two, have a look at the end of this chapter.

5.9.1 Timings

Now we have enough information to be able to begin comparing MongoDB to other databases. You could, for example normalise the AirportLocations data and load it into two related tables (Airport and Country) using MySQL. Using the Linux TIME command you could do some comparisons of load time, and retrieval time.

Of course MongoDb was designed from the outset as a distributed database and real performance benefits are more likely to come from large volumes of data spread across more than one instance in a distributed document database. Mongo refers

to this as Sharding. For now, however, let us just be happy with some elementary reconnaissance!

5.10 Summary

In this chapter we have seen that there are many different data storage methods available for a database. The decision as to which is the right one should be driven by the requirements of the system being supported.

We have examined column-based and a document-based examples of NoSQL databases and seen that they support different types of applications from those transactionally based relational systems we might be used to. There can be no doubt that the advent of Web and Cloud computing generated opportunities and challenges for data professionals and that NoSQL is a potentially useful set of tools to deal with this new era.

5.11 Review Questions

The answers to these questions can be found in the text of this chapter.

- What do ACID and BASE stand for, and what is the most significant difference between them?
- What type of data is best stored using Cassandra?
- What type of data is best stored using MongoDB?
- What is meant by Sharding?
- What do the letters "CAP" stand for, and describe what is meant by each letter.

5.12 Group Work Research Activities

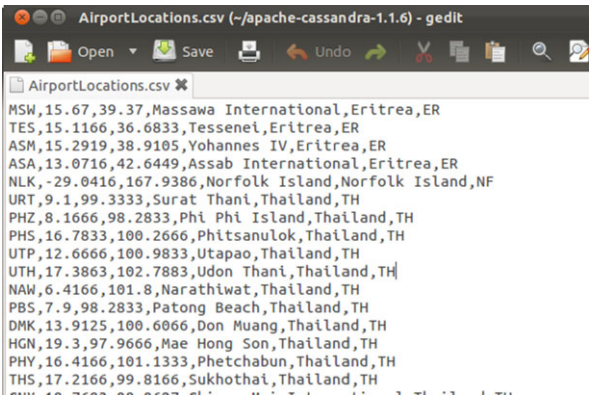
These activities require you to research beyond the contents of the book and can be tackled individually or as a discussion group.

Discussion Topic 1 Once you have become familiar with the NoSQL databases in the tutorials above, you should draw up a SWOT analysis to see what strengths and weaknesses, threats and opportunities may be derived from adopting the database in any organisation.

Discussion Topic 2 Try to think of criteria you might use to compare different types of database, including RDBMS and NoSQL examples, to help you decide which might be the most appropriate for a given application. For example you might think Performance is an important criterion. Having established your criteria, consider how you would measure them. What sort of tests might you need to carry out in order to compare the different databases?

5.12.1 Sample Solutions

Open the spreadsheet you have downloaded from Infochimps. Remember it is Tab separated. Remove the unwanted columns and then save as a CSV file and name it AirportLocations.csv. It should now look like this:



Then, using your editor, create a CQL Script and call it NorthofHeathrow. It should contain:

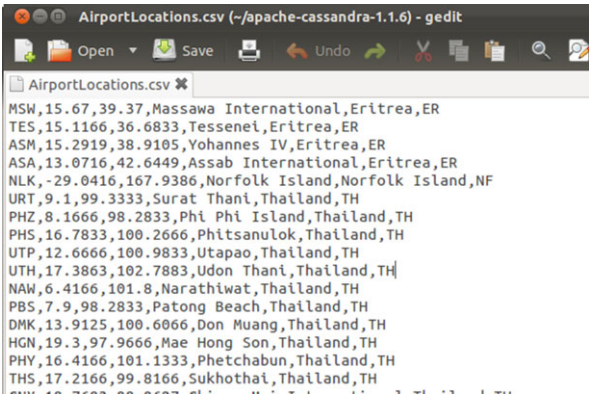
```
use Flights;
create ColumnFamily Airports
(KEY varchar PRIMARY KEY,
 Lat float,
 Lon float,
 Fullname varchar,
 Country varchar,
 CountryCode varchar
) ;
Create index on Airports (Lat) ;
Create index on Airports (CountryCode) ;
copy Airports (KEY, Lat, Lon, Fullname, Country, CountryCode) from 'AirportLocations.csv' ;
select Lat from Airports where KEY = 'LHR';
select count(*) from Airports where CountryCode = 'GB' and Lat > 51.5;
```

Run the script:

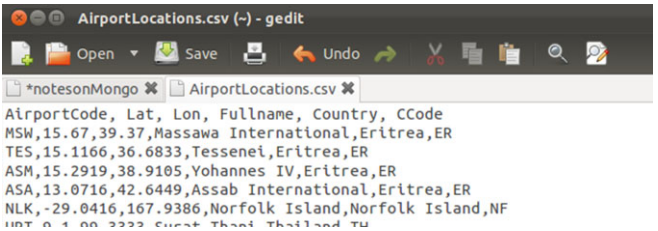
```
./bin/cqlsh < 'NorthofHeathrow'
```

5.12.2 MongoDB Crib

Open the spreadsheet you have downloaded from Infochimps. Remember it is Tab separated. Remove the unwanted columns and then save as a CSV file and name it AirportLocations.csv. It should now look like this:



Add a row at the beginning to give the field names;



Use MongoImport, as described earlier to import the data. Then issue the following queries:

db.AllAirports.find({AirportCode: “LHR”})
db.AllAirports.find({Lat: {$gt: 51}, CCode: “GB”})
db.AllAirports.count({Lat: {$gt: 51.47}, CCode: “GB”})

References

Brewer E (2012) CAP twelve years later: how the “rules” have changed. Computer 45(2):23–29. doi:10.1109/MC.2012.37

Cattell R (2010) Scalable SQL and NoSQL data stores. SIGMOD Rec 39(4)

Chang F, Dean J, Ghemawat S, Hsieh WC, Wallach DA, Burrows M, Chandra T, Fikes A, Gruber RE (2006) Bigtable: a distributed storage system for structured data. In: Proceedings of the 7th symposium on operating systems design and implementation (OSDI ’06). USENIX Association, Berkeley, pp 205–218

Gilbert S, Lynch N (2002) Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services. SIGACT News 33(2)

Hill R, Hirsch L, Lake P, Moshiri S (2013) Guide to cloud computing: principles and practice. Springer, London