Mastering Apache Cassandra
Second Edition

With ever increasing rates of data creation comes the demand to store data as fast and reliably as possible, a demand met by modern databases such as Cassandra. Apache Cassandra is the perfect choice for building fault tolerant and scalable databases. Through this practical guide, you will program pragmatically and understand completely the power of Cassandra. Starting with a brief recap of the basics to get everyone up and running, you will move on to deploy and monitor a production setup, dive under the hood, and optimize and integrate it with other software.

You will explore the integration and interaction of Cassandra components, and explore great new features such as CQL3, vnodes, lightweight transactions, and triggers. Finally, by learning Hadoop and Pig, you will be able to analyze your big data.

Who this book is written for
The book is aimed at intermediate developers with an understanding of core database concepts who want to become a master at implementing Cassandra for their application.

What you will learn from this book
- Write programs using Cassandra's features more efficiently
- Get the most out of a given infrastructure, improve performance, and tweak JVM
- Use CQL3 in your application, which makes working with Cassandra more simple
- Configure Cassandra and fine-tune its parameters depending on your needs
- Set up a cluster and learn how to scale it
- Monitor Cassandra cluster in different ways
- Use Hadoop and other big data processing tools with Cassandra


Nishant Neeraj

Prices do not include local sales tax or VAT where applicable

$ 49.99 US
£ 32.99 UK
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Quick Start'
- A synopsis of the book’s content
- More information on Mastering Apache Cassandra Second Edition

About the Author

Nishant Neeraj is an independent software developer with experience in developing and planning out architectures for massively scalable data storage and data processing systems. Over the years, he has helped design and implement a wide variety of products and systems for companies ranging from small start-ups to large multinational companies. Currently, he helps drive WealthEngine's core product to the next level by leveraging a variety of big data technologies.
Mastering Apache Cassandra
Second Edition

Back in 2007, Twitter users would experience "fail whale" captioned with "Too many tweets..." occasionally. On August 03, 2013, Twitter posted a new high-tweet rate record: 143,199 per second, and we rarely saw the fail whale. Many things changed since 2007. People and things connected to the Internet have increased exponentially. Cloud computing and hardware on demand have become cheap and easily available. Distributed computing and the NoSQL paradigm have taken off with a plethora of freely available, robust, proven, and open source projects to store large datasets, process it, and visualize it. "Big Data" has become a cliché. With massive amounts of data that get generated at a very high speed via people or machines, our capability to store and analyze data has increased. Cassandra is one of the most successful data stores that scales linearly, is easy to deploy and manage, and is blazing fast.

This book is about Cassandra and its ecosystem. The aim of this book is to take you from the basics of Apache Cassandra to understand what goes on under the hood. The book has three broad goals. First, to help you take right design decisions and understand the patterns and antipatterns. Second, to enable you to manage infrastructure on a rainy day. Third, to introduce you to some of the tools that work with Cassandra to monitor and manage Cassandra and to analyze the big data that you have inside it.

This book does not take a purist approach, rather a practical one. You will come to know proprietary tools, GitHub projects, shell scripts, third-party monitoring tools, and enough references to go beyond and dive deeper if you want.

What This Book Covers

Chapter 1, Quick Start, is about getting excited and having the instant gratification of Cassandra. If you have no prior experience with Cassandra, you leave this chapter with enough information to get yourself started on the next big project.

Chapter 2, Cassandra Architecture, covers design decisions and Cassandra's internal plumbing. If you have never worked with a distributed system, this chapter has some gems of distributed design concepts. It will be helpful for the rest of the book when we look at patterns and infrastructure management. This chapter will also help you understand the discussion of the Cassandra mailing list and JIRA. It is a theoretical chapter; you can skip it and come back to it later if you wish.
Chapter 3, Effective CQL, covers CQL, which is the de facto language to communicate with Cassandra. This chapter goes into the details of CQL and various things that you can do using it.

Chapter 4, Deploying a Cluster, is about deploying a cluster right. Once you go through the chapter, you will realize it is not really hard to deploy a cluster. It is probably one of the simplest distributed systems.

Chapter 5, Performance Tuning, deals with getting the maximum out of the hardware the cluster is deployed on. Usually, you will not need to rotate lots of knobs, and the default is just fine.

Chapter 6, Managing a Cluster – Scaling, Node Repair, and Backup, is about the daily DevOps drills. Scaling up a cluster, shrinking it down, replacing a dead node, and balancing the data load across the cluster is covered in this chapter.

Chapter 7, Monitoring, talks about the various tools that can be used to monitor Cassandra. If you already have a monitoring system, you would probably want to plug Cassandra health monitoring to it, or you can choose the dedicated and thorough Cassandra monitoring tools.

Chapter 8, Integration with Hadoop, covers Cassandra, which is about large datasets, fast writes and reads, and terabytes of data. What is the use of data if you can't analyze it? This chapter gives an introduction to get you started with the Cassandra and Hadoop setups.
Welcome to Cassandra and congratulations on choosing a database that beats most of the NoSQL databases in performance. Cassandra is a powerful database based on solid fundamentals of distributed computing and fail-safe design, and it is well-tested by companies such as Facebook, Twitter, and Netflix. Unlike conventional databases and some of the modern databases that use the master-slave pattern, Cassandra uses the all-nodes-the-same pattern; this makes the system free from a single point of failure. This chapter is an introduction to Cassandra. The aim is to get you through with a proof-of-concept project to set the right state of mind for the rest of the book.

With version 2, Cassandra has evolved into a mature database system. It is now easier to manage, and more developer-friendly compared to the previous versions. With CQL 3 and removal of super columns, it is less likely that a developer can go wrong with Cassandra. In the upcoming sections, we will model, program, and execute a simple blogging application to see Cassandra in action. If you have a beginner-level experience with Cassandra, you may opt to skip this chapter.

**Introduction to Cassandra**

Quoting from Wikipedia:

"Apache Cassandra is an open source distributed database management system designed to handle large amounts of data across many commodity servers, providing high availability with no single point of failure. Cassandra offers robust support for clusters spanning multiple datacenters, with asynchronous masterless replication allowing low latency operations for all clients."

Let’s try to understand in detail what it means.
A distributed database

In computing, distributed means splitting data or tasks across multiple machines. In the context of Cassandra, it means that the data is distributed across multiple machines. It means that no single node (a machine in a cluster is usually called a node) holds all the data, but just a chunk of it. It means that you are not limited by the storage and processing capabilities of a single machine. If the data gets larger, add more machines. If you need more parallelism (ability to access data in parallel/concurrently), add more machines. This means that a node going down does not mean that all the data is lost (we will cover this issue soon).

If a distributed mechanism is well designed, it will scale with a number of nodes. Cassandra is one of the best examples of such a system. It scales almost linearly, with regard to performance, when we add new nodes. This means that Cassandra can handle the behemoth of data without wincing.


High availability

We will discuss availability in the next chapter. For now, assume availability is the probability that we query and the system just works. A high-availability system is one that is ready to serve any request at any time. High availability is usually achieved by adding redundancies. So, if one part fails, the other part of the system can serve the request. To a client, it seems as if everything works fine.

Cassandra is a robust software. Nodes joining and leaving are automatically taken care of. With proper settings, Cassandra can be made failure-resistant. This means that if some of the servers fail, the data loss will be zero. So, you can just deploy Cassandra over cheap commodity hardware or a cloud environment, where hardware or infrastructure failures may occur.
Replication

Continuing from the previous two points, Cassandra has a pretty powerful replication mechanism (we will see more details in the next chapter). Cassandra treats every node in the same manner. Data need not be written on a specific server (master), and you need not wait until the data is written to all the nodes that replicate this data (slaves). So, there is no master or slave in Cassandra, and replication happens asynchronously. This means that the client can be returned with success as a response as soon as the data is written on at least one server. We will see how we can tweak these settings to ensure the number of servers we want to have data written on before the client returns.

From this, we can derive that when there is no master or slave, we can write to any node for any operation. Since we have the ability to choose how many nodes to read from or write to, we can tweak it to achieve very low latency (read or write from one server).

Multiple data centers

Expanding from a single machine to a single data center cluster or multiple data centers is very simple compared to traditional databases where you need to make a plethora of configuration changes and watch replication. If you are planning to shard, it becomes a developer's nightmare. We will see later in this book that we can use this data center setting to make a real-time replicating system across data centers. We can use each data center to perform different tasks without overloading the other data centers. This is a powerful support when you do not have to worry whether users in Japan with a data center in Tokyo and users in the US with a data center in Virginia, are in sync or not.

These are just broad strokes of Cassandra's capabilities. We will explore more in the upcoming chapters. This chapter is about getting excited learning about Cassandra.
A brief introduction to a data model

Cassandra has three containers, one within another. The outermost container is **keyspace**. You can think of keyspace as a database in the RDBMS land. Tables reside under keyspace. A table can be assumed as a relational database table, except it is more flexible. A table is basically a sorted map of sorted maps (refer to the following figure). Each table must have a primary key. This primary key is called **row key** or **partition key**. (We will later see that in a CQL table, the row key is the same as the primary key. If the primary key is made up of more than one column, the first component of this composite key is equivalent to the row key). Each partition is associated with a set of cells. Each cell has a name and a value. These cells may be thought of as columns in the traditional database system. The CQL engine interprets a group of cells with the same cell name prefix as a row. The following figure shows the Cassandra data model:

![Cassandra Data Model Diagram]

Note that if you come with Cassandra Thrift experience, it might be hard to view how Cassandra 1.2 and newer versions have changed terminology. Before CQL, the tables were called **column families**. A column family holds a group of rows, and rows are a sorted set of columns.
One obvious benefit of having such a flexible data storage mechanism is that you can have arbitrary number of cells with customized names and have a partition key store data as a list of tuples (a tuple is an ordered set; in this case, the tuple is a key-value pair). This comes handy when you have to store things such as time series, for example, if you want to use Cassandra to store your Facebook timeline or your Twitter feed or you want the partition key to be a sensor ID and each cell to represent a tuple with name as the timestamp when the data was created and value as the data sent by the sensor. Also, in a partition, cells are by default naturally ordered by the cell's name. So, in our sensor case, you will get data sorted for free.

The other difference is, unlike RDBMS, Cassandra does not have relations. This means relational logic will be needed to be handled at the application level. This also means that we may want to denormalize the database because there is no join and to avoid looking up multiple tables by running multiple queries. Denormalization is a process of adding redundancy in data to achieve high read performance. For more information, visit http://en.wikipedia.org/wiki/Denormalization.

Partitions are distributed across the cluster, creating effective auto-sharding. Each server holds a range(s) of keys. So, if balanced, a cluster with more nodes will have less rows per node. All these concepts will be repeated in detail in the later chapters.

Types of keys

In the context of Cassandra, you may find the concept of keys a bit confusing. There are five terms that you may encounter. Here is what they generally mean:

• **Primary key**: This is the column or a group of columns that uniquely defines a row of the CQL table.

• **Composite key**: This is a type of primary key that is made up of more than one column. Sometimes, the composite key is also referred to as the compound key.

• **Partition key**: Cassandra's internal data representation is large rows with a unique key called row key. It uses these row key values to distribute data across cluster nodes. Since these row keys are used to partition data, they as called partition keys. When you define a table with a simple key, that key is the partition key. If you define a table with a composite key, the first term of that composite key works as the partition key. This means all the CQL rows with the same partition key lives on one machine.
- **Clustering key**: This is the column that tells Cassandra how the data within a partition is ordered (or clustered). This essentially provides presorted retrieval if you know what order you want your data to be retrieve in.

- **Composite partition key**: Optionally, CQL lets you define a composite partition key (the first part of a composite key). This key helps you distribute data across nodes if any part of the composite partition key differs. Let's take a look at the following example:

```sql
CREATE TABLE customers (
    id uuid,
    email text,
    PRIMARY KEY (id)
)
```

In the preceding example, `id` is the primary key and also the partition key. There is no clustering. It is a simple key. Let's add a twist to the primary key:

```sql
CREATE TABLE country_states (
    country text,
    state text,
    population int,
    PRIMARY KEY (country, state)
)
```

In the preceding example, we have a composite key that uses `country` and `state` to uniquely define a CQL row. The `country` column is the partition key, so all the rows with the same country node will belong to the same node/machine. The rows within a partition will be sorted by the `state` names. So, when you query for states in the US, you will encounter the row with California before the one with New York. What if I want to partition by composition? Let's take a look at the following example:

```sql
CREATE TABLE country_chiefs (
    country text,
    prez_name text,
    num_states int,
    capital text,
    ruling_year int,
    PRIMARY KEY ((country, prez_name), num_states, capital)
)
```

The preceding example has a composite key involving four columns: `country`, `prez_name`, `num_states`, and `capital`, with `country` and `prez_name` constituting composite partition key. This means the rows with the same country but different president will be in a different partition. Rows will be ordered by the number of states followed by the capital name.
Installing Cassandra locally

Installing Cassandra on your local machine for experimental or development purposes is as easy as downloading and unzipping the tarball (the .tar compressed file). For development purposes, Cassandra does not have any extreme requirements. Any modern computer with 1 GB of RAM and a dual-core processor is good to test the water. All the examples in this chapter are performed on a laptop with 4 GB of RAM, a dual-core processor, and the Ubuntu 14.04 operating system. Cassandra is supported on all major platforms; after all, it's Java. Here are the steps to install Cassandra locally:

1. Install Oracle Java 1.6 (Java 6) or higher. Installing the JVM is sufficient, but you may need the Java Development Kit (JDK) if you are planning to code in Java:

   # Check whether you have Java installed in your system
   $ java -version
   java version "1.7.0_21"
   Java(TM) SE Runtime Environment (build 1.7.0_21-b11)
   Java HotSpot(TM) 64-Bit Server VM (build 23.21-b01, mixed mode)

   If you do not have Java, you may want to follow the installation details for your machine from the Oracle Java website (http://www.oracle.com/technetwork/java/javase/downloads/index.html).

2. Download Cassandra 2.0.0 or a newer version from the Cassandra website (http://archive.apache.org/dist/cassandra/ or http://cassandra.apache.org/download/). This book uses Cassandra 2.1.2, which was the latest version at the time of writing this book. Decompress this file to a suitable directory:

   # Download Cassandra
   wget http://archive.apache.org/dist/cassandra/2.1.2/apache-cassandra-2.1.2-bin.tar.gz

   # Untar to your home directory
   tar xzf apache-cassandra-2.1.2-bin.tar.gz -C $HOME

   The unzipped file location is $HOME/apache-cassandra-2.1.2. Let's call this location CASSANDRA_HOME. Wherever we refer to CASSANDRA_HOME in this book, always assume it to be the location where Cassandra is installed.

3. You may want to edit $CASSANDRA_HOME/conf/cassandra.yaml to configure Cassandra. It is advisable to change the data directory to a writable location when you start Cassandra as a nonroot user.
4. To change the data directory, change the `data_file_directories` attribute in `cassandra.yaml`, as follows (here, the data directory is chosen as `/mnt/Cassandra/data`; you may want to set the directory where you want to put the data):

   ```yaml
   data_file_directories:
     - /mnt/cassandra/data
   ```

5. Set the commit log directory:

   ```yaml
   commitlog_directory: /mnt/cassandra/commitlog
   ```

6. Set the saved caches directory:

   ```yaml
   saved_caches_directory: /mnt/cassandra/saved_caches
   ```

7. Set the logging location. Edit `$CASSANDRA_HOME/conf/log4j-server.properties` as follows:

   ```properties
   log4j.appender.R.File=/tmp/cassandra.log
   ```

With this, you are ready to start Cassandra. Fire up your shell and type in `$CASSANDRA_HOME/bin/cassandra -f`. In this command, `-f` stands for foreground. You can keep viewing the logs and press `Ctrl + C` to shut the server down. If you want to run it in the background, do not use the `-f` option. If your data or log directories are not set with the appropriate user permission, you may want to start Cassandra as superuser using the `sudo` command. The server is ready when you see statistics in the startup log:

```
INFO 14:55:37 system.schema_triggers    0,0
INFO 14:55:37 system.local              0,0
INFO 14:55:37 system.peers             1,152
INFO 14:55:37 system.batchlog          0,0
INFO 14:55:37 system.compactions_in_progress 0,0
INFO 14:55:37 system.schema_usertypes 0,0
INFO 14:55:37 system.schema_keyspaces 4,471
INFO 14:55:37 system.compaction_history 19,2573
INFO 14:55:37 system.paxos             0,0
INFO 14:55:37 system.schema_columns   792,215617
INFO 14:55:37 system.schema_columnfamilies 433,97089
INFO 14:55:37 system.IndexInfo         0,0
INFO 14:55:37 system.range_xfers       0,0
INFO 14:55:37 system.peer_events      0,0
INFO 14:55:37 system.hints            0,0
INFO 14:55:37 system.sstable_activity 0,0
INFO 14:55:37 hadoop_test.output      0,0
INFO 14:55:37 hadoop_test.lines       0,0
INFO 14:55:37 system_traces.sessions  0,0
INFO 14:55:37 system_traces.events    0,0
```
Cassandra in action

There is no better way to learn a technology than by performing a proof of concept of the technology. In this section, we will work on a very simple application to get you familiarized with Cassandra. We will build the backend of a simple blogging application, where a user can perform the following tasks:

- Create a blogging account
- Publish posts
- Tag the posts, and posts can be searched using those tags
- Have people comment on those posts
- Have people upvote or downvote a post or a comment

Modeling data

In the RDBMS world, you would glance over the entities and think about relations while modeling the application. Then, you will join tables to get the required data. There is no join option in Cassandra, so we will have to denormalize things. Looking at the previously mentioned specifications, we can say that:

- We need a blogs table to store the blog name and other global information, such as the blogger's username and password
- We will have to pull posts for the blog, ideally, sorted in reverse chronological order
- We will also have to pull all the comments for each post, when we see the post page
- We will have to maintain tags in such a way that tags can be used to pull all the posts with the same tag
- We will also have to have counters for the upvotes and downvotes for posts and comments

With the preceding details, let's see the tables we need:

- **blogs**: This table will hold global blog metadata and user information, such as blog name, username, password, and other metadata.
• **posts:** This table will hold individual posts. At first glance, `posts` seems to be an ordinary table with primary keys as post ID and a reference to the blog that it belongs to. The problem arises when we add the requirement of being able to be sorted by timestamp. Unlike RDBMS, you cannot just perform an `ORDER BY` operation across partitions. The work-around for this is to use a composite key. A composite key consists of a partition key and one or more column(s) that determines where the other columns are going to be stored. Also, the other columns in the composite key determine relative ordering for the set of columns that are being inserted as a row with the key.

Remember that a partition is completely stored on a node. The benefit of this is that the fetches are faster, but at the same time a partition is limited by the total number of cells that it can hold, which is 2 billion cells. The other downside of having everything on one partition may cause lots of requests to go to only a couple of nodes (replicas), making them a hotspot in the cluster, which is not good. You can avoid this by using some sort of bucketing such as involving months and years in the partition key. This will make sure that the partition changes every month and each partition has only one month worth of records. This will solve both the problems: the cap on the number of records and the hotspot issue. However, we will still need a way to order buckets. For this example, we will have all the posts in one partition just to keep things simple. We will tackle the bucketing issues in Chapter 3, *Effective CQL*. The following figure shows how to write time series grouped data using composite columns:

![posts table](image)

• **comments:** They have a similar property as post, except it is linked to a post instead of being linked to a blog.

• **tags:** They are a part of post. We use the `set` data type to represent tags on the posts. One of the features that we mentioned earlier is to be able to search posts by tags. The best way to do it is to create an index on the `tags` column and make it searchable. Unfortunately, index on collections data types has not been supported until Cassandra Version 2.1 ([https://issues.apache.org/jira/browse/CASSANDRA-4511](https://issues.apache.org/jira/browse/CASSANDRA-4511)). In our case, we will have to create and manage this sort of indexing manually. So, we will create a `tags` table that will have a compound primary key with tag and blog ID as its components.
• **counters:** Ideally, you would think that you want to put upvote and downvote counters as a part of the `posts` and `comments` tables' column definition, but Cassandra does not support a table that has a counter type column(s) and some other type column unless the counter is a part of the primary key definition. So, in our case, we will create two new tables just to keep track of votes.

With this, we are done with data modeling. The next step is inserting and getting data back.
Quick Start

Writing code
Time to start something tangible! In this section, we will create the schema, insert the data, and make interesting queries to retrieve the data. In a real application, you will have a GUI with button and links to be able to log in, post, comment, upvote and downvote, and navigate. Here, we will stick to what happens in the backend when you perform those actions. This will keep the discussion from any clutter introduced by other software components. Also, this section contains Cassandra Query Language (CQL), a SQL-like query language for Cassandra. So, you can just copy these statements and paste them into your CQL shell ($CASSANDRA_HOME/bin/cqlsh) to see it working. If you want to build an application using these statements, you should be able to just use these statements in your favorite language via the CQL driver library that you can find at http://www.datastax.com/download#dl-datastax-drivers. You can also download a simple Java application that is built using these statements from my GitHub account (https://github.com/naishe/mastering-cassandra-v2).

Setting up
Setting up a project involves creating a keyspace and tables. This can be done via the CQL shell or from your favorite programming language.

Here are the statements to create the schema:

```cql
CREATE KEYSPACE weblog WITH REPLICATION = {'class': 'SimpleStrategy', 'replication_factor': 1};

USE weblog;

CREATE TABLE blogs (id uuid PRIMARY KEY, blog_name varchar, author varchar, email varchar, password varchar);

CREATE TABLE posts (id timeuuid, blog_id uuid, posted_on timestamp, title text, content text, tags set<varchar>, PRIMARY KEY(blog_id, id));

CREATE TABLE categories (cat_name varchar, blog_id uuid, post_id timeuuid, post_title text, PRIMARY KEY(cat_name, blog_id, post_id));

CREATE TABLE comments (id timeuuid, post_id timeuuid, title text, content text, posted_on timestamp, commenter varchar, PRIMARY KEY(post_id, id));
```
Universally unique identifiers: uuid and timeuuid

In the preceding CQL statements, there are two interesting data types—uuid and timeuuid. uuid stands for universally unique identifier. There are five types of them. One of these uuid types is timeuuid, which is essentially uuid type 1 that takes timestamp as its first component. This means it can be used to sort things by time. This is what we wanted to do in this example: sort posts by the time they were published.

On the other hand, uuid accepts any of these five types of uuid as long as the format follows the standard uuid format.

In Cassandra, if you have chosen the uuid type for a column, you will need to pass uuid while inserting the data. With timeuuid, just passing timestamp is enough.

The first statement requests Cassandra to create a keyspace named weblog with replication factor 1 because we are running a single node Cassandra on a local machine. Here are a couple of things to notice:

- The column tags in the posts table is a set of strings.
- The primary key for posts, categories, and comments has more than one component. The first of these components is a partition key. Data with the same primary key in a table resides on the same machine. This means, all the posts' records that belong to one blog stays on one machine (not really; if the replication factor is more than one, the records get replicated to as many machines). This is true for all the tables with composite keys.
- Categories have three components in its primary key. One is the category name, which is the partition key, another is the blog ID, and then the post ID. One can argue that inclusion of the post ID in the primary key was unnecessary. You could just use the category name and blog ID. The reason to include the post ID in the primary key was to enable sorting by the post ID.
- Note that some of the IDs in the table definition are timeuuid. The timeuuid data type is an interesting ID generation mechanism. It generates a timestamp-based (provided by you) uuid, which is unique and you can use it in applications where you want things to be ordered by chronology.
Inserting records

This section demonstrates inserting the records in the schema. Unlike RDBMS, you will find that there are some redundancies in the system. You may notice that you cannot have a lot of rules enforced by Cassandra. It is up to the developer to make sure the records are inserted, updated, and deleted from appropriate places.

Note that the CQL code is just for instruction purposes and is just a snippet. Your output may vary.

We will see a simple INSERT example now:

```cql
INSERT INTO blogs (id, blog_name, author, email, password) VALUES (blobAsUuid(timeuuidAsBlob(now())), 'Random Ramblings', 'JRR Rowling', 'rowling@yahoo.com', 'someHashed#password');
```

```cql
SELECT * FROM blogs;
```

```
id           | author      | blog_name        | email             | password
------------+-------------+------------------+-------------------|---
83ccc... | JRR Rowling | Random Ramblings | rowling@yahoo.com | someHashed#password
```

(1 rows)

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.
The application would generate uuid or you will get uuid from an existing record in the blogs table based on a user's e-mail address or some other criteria. Here, just to be concise, the uuid generation is left to Cassandra, and it is retrieved by running the SELECT statement. Let’s insert some posts to this blog:

# First post

```sql
CQLSH:weblog> INSERT INTO posts (id, blog_id, title, content, tags, posted_on) VALUES (now(), 83cec740-22b1-11e4-a4f0-7f1a8b30f852, 'first post', 'hey howdy!', {'random','welcome'}, 1407822921000);
CQLSH:weblog> SELECT * FROM posts;
```

<table>
<thead>
<tr>
<th>blog_id</th>
<th>id</th>
<th>content</th>
<th>posted_on</th>
<th>tags</th>
</tr>
</thead>
</table>
| 83cec... | 04722... | hey howdy! | 2014-08-12 11:25:21+0530 | {'random', 'welcome'} | first post
| (1 rows) |

```sql
CQLSH:weblog> INSERT INTO categories (cat_name, blog_id, post_id, post_title) VALUES ( 'random', 83cec740-22b1-11e4-a4f0-7f1a8b30f852, 047224f0-22b2-11e4-a4f0-7f1a8b30f852, 'first post');
CQLSH:weblog> INSERT INTO categories (cat_name, blog_id, post_id, post_title) VALUES ( 'welcome', 83cec740-22b1-11e4-a4f0-7f1a8b30f852, 047224f0-22b2-11e4-a4f0-7f1a8b30f852, 'first post');
```

# Second post

```sql
CQLSH:weblog> INSERT INTO posts (id, blog_id, title, content, tags, posted_on) VALUES (now(), 83cec740-22b1-11e4-a4f0-7f1a8b30f852, 'Fooled by randomness...', 'posterior=(prior*likelihood)/evidence', {'random','maths'}, 1407823189000);
CQLSH:weblog> select * from posts;
```
Quick Start

<table>
<thead>
<tr>
<th>blog_id</th>
<th>id</th>
<th>content</th>
<th>title</th>
</tr>
</thead>
<tbody>
<tr>
<td>posted_on</td>
<td>tags</td>
<td>title</td>
<td></td>
</tr>
<tr>
<td>--------------------</td>
<td>-----------------------</td>
<td>----------------------</td>
<td></td>
</tr>
<tr>
<td>---</td>
<td></td>
<td>---</td>
<td></td>
</tr>
</tbody>
</table>

83cec.... | 04722... | hey howdy! | 2014-08-12 11:25:21+0530 | {'random', 'welcome'} | first post

83cec... | c06a4... | posterior=(prior*likelihood)/evidence | 2014-08-12 11:29:49+0530 | {'maths', 'random'} | Fooled by randomness...

(2 rows)

cqlsh:weblog> INSERT INTO categories (cat_name, blog_id, post_id, post_title) VALUES ( 'random', 83cec740-22b1-11e4-a4f0-7f1a8b30f852, c06a42f0-22b2-11e4-a4f0-7f1a8b30f852, 'Fooled by randomness...');
cqlsh:weblog> INSERT INTO categories (cat_name, blog_id, post_id, post_title) VALUES ( 'maths', 83cec740-22b1-11e4-a4f0-7f1a8b30f852, c06a42f0-22b2-11e4-a4f0-7f1a8b30f852, 'Fooled by randomness...');

You may want to insert more rows so that we can experiment with pagination in the upcoming sections.

You may notice that the primary key, which is of type timeuuid, is created using Cassandra's built-in \texttt{now()} function, and we repeated the title in the \texttt{categories} table. The rationale behind repetition is that we want to display the title of all the posts that match a tag that a user clicked. These titles will have URLs to redirect us to the posts (a post can be retrieved by the blog ID and post ID). Alternatively, Cassandra does not support a relational connect between two tables, so you cannot join categories and posts to display the title. The other option is to use the blog ID and post ID to retrieve the post's title. However, that's more work, and somewhat inefficient.

Let's insert some comments and upvote and downvote some posts and comments:

\texttt{# Insert some comments}

cqlsh:weblog> INSERT INTO comments (id, post_id, commenter, title, content, posted_on) VALUES (now(), c06a42f0-22b2-11e4-a4f0-7f1a8b30f852, 'liz@gmail.com', 'Thoughtful article but...', 'It is too short to describe the complexity.', 1407868973000);
Chapter 1

cqlsh:weblog> INSERT INTO comments (id, post_id, commenter, title, content, posted_on) VALUES (now(), c06a42f0-22b2-11e4-a4f0-7f1a8b30f852, 'tom@gmail.com', 'Nice!', 'Thanks, this is good stuff.', 1407868975000);

cqlsh:weblog> INSERT INTO comments (id, post_id, commenter, title, content, posted_on) VALUES (now(), c06a42f0-22b2-11e4-a4f0-7f1a8b30f852, 'g@ouv.com', 'Follow my blog', 'Please follow my blog.', 1407868979000);

cqlsh:weblog> INSERT INTO comments (id, post_id, commenter, title, content, posted_on) VALUES (now(), 047224f0-22b2-11e4-a4f0-7f1a8b30f852, 'liz@gmail.com', 'New blogger?', 'Welcome to weblog application.', 1407868981000);

# Insert some votes

cqlsh:weblog> UPDATE comment_votes SET upvotes = upvotes + 1 WHERE comment_id = be127d00-22c2-11e4-a4f0-7f1a8b30f852;

cqlsh:weblog> UPDATE comment_votes SET upvotes = upvotes + 1 WHERE comment_id = be127d00-22c2-11e4-a4f0-7f1a8b30f852;

cqlsh:weblog> UPDATE comment_votes SET downvotes = downvotes + 1 WHERE comment_id = be127d00-22c2-11e4-a4f0-7f1a8b30f852;

cqlsh:weblog> UPDATE post_votes SET downvotes = downvotes + 1 WHERE post_id = d44e0440-22c2-11e4-a4f0-7f1a8b30f852;

cqlsh:weblog> UPDATE post_votes SET upvotes = upvotes + 1 WHERE post_id = d44e0440-22c2-11e4-a4f0-7f1a8b30f852;

Counters are always inserted or updated using the UPDATE statement.

**Retrieving data**

Now that we have data inserted for our application, we need to retrieve it. To blog applications, usually the blog name serves as the primary key in their database. So, when you request cold-caffein.blogspot.com, a blog metadata table with the blog ID as cold-caffein exists. We, on the other hand, can use the blog uuid to request to serve the contents. So, we assume that having the blog ID is handy.
Quick Start

Let's display posts. We should not load all the posts for the user upfront. It is not a good idea from the usability point of view. It demands more bandwidth, and it is probably a lot of reads for Cassandra. So first, let's pull two posts at a time from ones posted earlier:

```
cqlsh:weblog> select * from posts where blog_id = 83cec740-22b1-11e4-a4f0-7f1a8b30f852 order by id desc limit 2;
```

<table>
<thead>
<tr>
<th>blog_id</th>
<th>id</th>
<th>content</th>
<th>posted_on</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>posterior=(prior*likelihood)/evidence</td>
<td>2014-08-12 11:29:49+0530</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{'maths', 'random'}</td>
<td>Fooled by randomness...</td>
</tr>
<tr>
<td></td>
<td></td>
<td>hey howdy!</td>
<td>2014-08-12 11:25:21+0530</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{'random', 'welcome'}</td>
<td>first post</td>
</tr>
</tbody>
</table>

(2 rows)

This was the first page. For the next page, we can use an anchor. We can use the last post's ID as an anchor, as its timeuuid increases monotonically with time. So, posts older than that will have the post ID with smaller values, and this will work as our anchor:

```
cqlsh:weblog> select * from posts where blog_id = 83cec740-22b1-11e4-a4f0-7f1a8b30f852 and id < 8eab0c10-2314-11e4-bac7-3f5f68a133d8 order by id desc limit 2;
```

<table>
<thead>
<tr>
<th>blog_id</th>
<th>id</th>
<th>content</th>
<th>posted_on</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>random content</td>
<td>2014-08-13 23:33:00+0530</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{'garbage', 'random'}</td>
<td>random post</td>
</tr>
<tr>
<td></td>
<td></td>
<td>random content</td>
<td>2014-08-13 23:32:58+0530</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{'garbage', 'random'}</td>
<td>random post</td>
</tr>
</tbody>
</table>

(2 rows)
You can retrieve the posts on the next page as follows:

```sql
Cqlsh:weblog> select * from posts where blog_id = 83cec740-22b1-11e4-a4f0-7f1a8b30f852 and id < 76738dc0-2314-11e4-bac7-3f5f68a133d8 order by id desc limit 2;
```

<table>
<thead>
<tr>
<th>blog_id</th>
<th>id</th>
<th>content</th>
<th>posted_on</th>
</tr>
</thead>
<tbody>
<tr>
<td>tags</td>
<td>title</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>--------------</td>
<td>-----------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>83cec...</td>
<td>6f85d...</td>
<td>random content6</td>
<td>2014-08-13 23:32:56+0530</td>
</tr>
<tr>
<td>{'garbage', 'random'}</td>
<td>random post6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| 83cec... | 684c5... | random content5 | 2014-08-13 23:32:54+0530 |
| {'garbage', 'random'} | random post5 |

(2 rows)

Now for each post, we need to perform the following tasks:

- Pull a list of comments
- Up and downvotes
- Load comments as follows:

```sql
Cqlsh:weblog> select * from comments where post_id = c06a42f0-22b2-11e4-a4f0-7f1a8b30f852 order by id desc;
```

<table>
<thead>
<tr>
<th>post_id</th>
<th>id</th>
<th>commenter</th>
<th>content</th>
<th>posted_on</th>
</tr>
</thead>
<tbody>
<tr>
<td>title</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>--------------</td>
<td>---------------</td>
<td>-----------------------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>c06a4...</td>
<td>cd5a8...</td>
<td><a href="mailto:g@ouv.com">g@ouv.com</a></td>
<td>Please follow my blog.</td>
<td>2014-08-13 00:12:59+0530</td>
</tr>
<tr>
<td>Follow my blog</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

[19]
• Individually fetch counters for each post and comment as follows:

```cqlsh:weblog> select * from comment_votes where comment_id = be127d00-22c2-11e4-a4f0-7f1a8b30f852;

<table>
<thead>
<tr>
<th>comment_id</th>
<th>downvotes</th>
<th>upvotes</th>
</tr>
</thead>
<tbody>
<tr>
<td>be127...</td>
<td>1</td>
<td>6</td>
</tr>
</tbody>
</table>

(1 rows)
```

```cqlsh:weblog> select * from post_votes where post_id = c06a42f0-22b2-11e4-a4f0-7f1a8b30f852;

<table>
<thead>
<tr>
<th>post_id</th>
<th>downvotes</th>
<th>upvotes</th>
</tr>
</thead>
<tbody>
<tr>
<td>c06a4...</td>
<td>2</td>
<td>7</td>
</tr>
</tbody>
</table>

(1 rows)
```

Now, we want to facilitate the users of our blogging website with the ability to click on a tag and see a list of all the posts with that tag. Here is what we do:

```cqlsh:weblog> select * from categories where cat_name = 'maths' and blog_id = 83cec740-22b1-11e4-a4f0-7f1a8b30f852 order by blog_id desc;

<table>
<thead>
<tr>
<th>cat_name</th>
<th>blog_id</th>
<th>post_id</th>
<th>post_title</th>
</tr>
</thead>
<tbody>
<tr>
<td>maths</td>
<td>83cec...</td>
<td>a865c...</td>
<td>YARA</td>
</tr>
<tr>
<td>maths</td>
<td>83cec...</td>
<td>c06a4...</td>
<td>Fooled by randomness...</td>
</tr>
</tbody>
</table>

(2 rows)
We can obviously use the pagination and sorting here. I think you have got the idea.

Sometimes, it is nice to see what people generally comment. It would be great if we could find all the comments by a user. To make a nonprimary key field searchable in Cassandra, you need to create an index on that column. So, let's do that:

```cqlsh:weblog> CREATE INDEX commenter_idx ON comments (commenter);
```

```cqlsh:weblog> select * from comments where commenter = 'liz@gmail.com';
```

<table>
<thead>
<tr>
<th>post_id</th>
<th>id</th>
<th>commenter</th>
<th>content</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td><a href="mailto:liz@gmail.com">liz@gmail.com</a></td>
<td>Welcome to weblog application.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>New blogger?</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c06a4...</td>
<td>be127...</td>
<td><a href="mailto:liz@gmail.com">liz@gmail.com</a></td>
<td>It is too short to describe the complexity.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Thoughful article but...</td>
</tr>
</tbody>
</table>

(2 rows)

This completes all the requirements we stated. We did not cover the update and delete operations. They follow the same pattern as the insertion of records. The developer needs to make sure that the data is updated or deleted from all the places. So, if you want to update a post's title, it needs to be done in the posts and category tables.

**Writing your application**

Cassandra provides the API for almost all the main stream programming languages. Developing applications for Cassandra is nothing more than actually executing CQL through an API and collection result set or iterator for the query. This section will give you a glimpse of the Java code for the example we discussed earlier. It uses the DataStax Java driver for Cassandra. The full code is available at https://github.com/naishe/mastering-cassandra-v2.
Getting the connection

An application creates a single instance of the Cluster object and keeps it for its life cycle. Every time you want to execute a query or a bunch of queries, you ask for a session object from the Cluster object. In a way, it is like a connection pool. Let's take a look at the following example:

```java
public class CassandraConnection {
    private static Cluster cluster = getCluster();
    public static final Session getSession(){
        if ( cluster == null ){
            cluster = getCluster();
        }
        return cluster.connect();
    }

    private static Cluster getCluster(){
        Cluster clust = Cluster
                .builder()
                .addContactPoint(Constants.HOST)
                .build();
        return clust;
    }
}[-- snip --]
```

Executing queries

Query execution is barely different from what we did in the command prompt earlier:

```java
private static final String BLOGS_TABLE_DEF =
"CREATE TABLE IF NOT EXISTS "+
+ Constants.KEYSPACE + ".blogs "
+ "("
+ "id uuid PRIMARY KEY, "
+ "blog_name varchar, "
+ "author varchar, "
+ "email varchar, "
+ "password varchar"
+ ")";
[-- snip --]
    Session conn = CassandraConnection.getSession();
[--snip--]
    conn.execute(BLOGS_TABLE_DEF);
[-- snip --]
    conn.close();
```
Object mapping

The DataStax Java driver provides an easy-to-use, annotation-based object mapper, which can help you avoid a lot of code bloat and marshalling effort. Here is an example of the Blog object that maps to the blogs table:

```java
@Table(keyspace = Constants.KEYSPACE, name = "blogs")
public class Blog extends AbstractVO<Blog> {
    @PartitionKey
    private UUID id;
    @Column(name = "blog_name")
    private String blogName;
    private String author;
    private String email;
    private String password;

    public UUID getId() {
        return id;
    }
    public void setId(UUID id) {
        this.id = id;
    }
    public String getBlogName() {
        return blogName;
    }
    public void setBlogName(String blogName) {
        this.blogName = blogName;
    }
    public String getAuthor() {
        return author;
    }
    public void setAuthor(String author) {
        this.author = author;
    }
    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        /* Ideally, you’d use a unique salt with this hashing */
    }
```
this.password = Hashing
  .sha256()
  .hashString(password, Charsets.UTF_8)
  .toString();
}

@Override
public boolean equals(Object that) {
  return this.getId().equals(((Blog)that).getId());
}

@Override
public int hashCode() {
  return Objects.hashCode(getId(), getEmail(), getAuthor(),
      getBlogName());
}

@Override
protected Blog get Instance() {
  return this;
}

@Override
protected Class<Blog> getType() {
  return Blog.class;
}

// ----- ACCESS VIA QUERIES -----
public static Blog getBlogByName(String blogName,
      SessionWrapper sessionWrapper)
  throws BlogNotFoundException {
  AllQueries queries = sessionWrapper.getAllQueries();
  Result<Blog> rs = queries.getBlogByName(blogName);
  if (rs.isExhausted()){
    throw new BlogNotFoundException();
  }
  return rs.one();
}
For now, forget about the AbstractVO super class. That is just some abstraction, where common things are thrown into AbstractVO. You can see the annotations that basically show which keyspace and table this class is mapped to. Each instance variable is mapped with a column in the table. For any column that has a different name than the attribute name in the class, you will have to explicitly state that. Getters and setters do not have to be dumb. You can get creative in there. For example, setPassword setter takes a plain text password and hashes it before storing. Note that you must mention which field acts as the partition key. You do not have to specify all the fields that consist of the primary key, just the first component. Now you can use DataStax's mapper to create, retrieve, update, and delete an object without having to marshal the results into the object. Here is an example:

```java
Blog blog =
    new MappingManager(session)
    .mapper(Blog.class)
    .get(blogUUID);
```

You can execute any arbitrary queries and map it to an object. To do that, you will have to write an interface that contains a method signature of what the query consumes as its argument and what it returns as the method return type, as follows:

```java
@Accessor
public interface AllQueries {
    [--snip--]

    @Query("SELECT * FROM " + Constants.KEYSPACE + ".blogs WHERE blog_name = :blogName")
    public Result<Blog> getBlogByName(@Param("blogName") String blogName);
    [-- snip --]

    This interface is annotated with Accessor, and it has methods that basically satisfy the Query annotation that it carries. The snippet of the Blog class uses this method to retrieve the names blog by blog.
```
Summary

We have started learning about Cassandra. You can set up your local machine, play with CQL3 in cqlsh, and write a simple program that uses Cassandra on the backend. It seems like we are all done. But, it's not so. Cassandra is not all about ease in modeling or simple to code around with (unlike RDBMS). It is all about speed, availability, and reliability. The only thing that matters in a production setup is how quickly and reliably your application can serve a fickle-minded user. It does not matter if you have an elegant database architecture with the third normal form or if you use a functional programming language and follow the Don't Repeat Yourself (DRY) principle religiously. Cassandra and many other modern databases, especially in the NoSQL space, are there to provide you with speed. Cassandra's performance increases almost linearly with the addition of new nodes, which makes it suitable for high throughput applications without committing a lot of expensive infrastructure to begin with. For more information, visit http://vldb.org/pvldb/vol5/p1724_tilmannrabl_vldb2012.pdf. The rest of the book is aimed at giving you a solid understanding of the following aspects of Cassandra—one chapter at a time:

- You will learn the internals of Cassandra and the general programming pattern for Cassandra
- Setting up a cluster and tweaking Cassandra and Java settings to get the maximum out of Cassandra for your use
- Infrastructure maintenance—nodes going down, scaling up and down, backing the data up, keeping vigil monitoring, and getting notified about an interesting event on your Cassandra setup will be covered
- Cassandra is easy to use with the Apache Hadoop and Apache Pig tools and we will see simple examples of this

The best thing about these chapters is that there is no prerequisite. Most of these chapters start from the basics to get you familiar with the concept and then take you to an advanced level. So, if you have never used Hadoop, do not worry. You can still have a simple setup up and running with Cassandra.

In the next chapter, we will see Cassandra internals and what makes it so fast.
Where to buy this book

You can buy Mastering Apache Cassandra Second Edition from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.