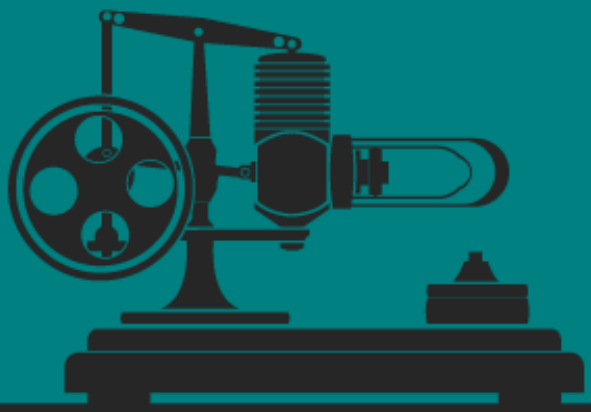

Cassandra

Succinctly

by Marko Švaljek

Syncfusion | Technology Resource Portal

Cassandra Succinctly

By
Marko Švaljek

Foreword by Daniel Jebaraj

2501 Aerial Center Parkway
Suite 200
Morrisville, NC 27560
USA

Technical Reviewer: Buddy James
Copy Editor: Suzanne Kattau
Acquisitions Coordinator: Hillary Bowling, marketing coordinator, Syncfusion, Inc.
Proofreader: Graham High, content producer, Syncfusion, Inc.

Table of Contents

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to "enable AJAX support with one click," or "turn the moon to cheese!"

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and "Like" us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

Marko Švaljek is a software developer and blogger with almost 10 years of professional experience in front-end and back-end development for leading financial and telecommunications companies in the Central European region, with emphasis on machine-to-machine, mobile banking, and e-commerce solutions. Most of the professional experience he has gathered is associated with the Java ecosystem. His professional experience with Cassandra started in 2013 when he began working at the machine networks department of the Kapsch Group. There he came into contact with Apache Cassandra and has enjoyed working with it ever since. Marko is also the organizer of the Croatian Cassandra Users Meetup Group.

Chapter 1 Introduction

Apache Cassandra is a very scalable, NoSQL open-source database designed as a peer-to-peer distributed system where all nodes are the same and the data is distributed among the nodes in the cluster. Since there is no single point of failure, having nodes out of order in a cluster is not a big deal for Cassandra. It's a common thing that nodes are added and taken out of the cluster during regular working hours, without having to wait for the system load to drop somewhere during the night. Also, one of the key features of Cassandra is that it works on commodity hardware and is easily deployed on a cloud-based infrastructure.

Note: In Cassandra terminology, a node is one Cassandra instance. A cluster is a group of two or more nodes working together and communicating with the gossip protocol.

The Name Cassandra

Let's start with the name. Cassandra was the very beautiful daughter of King Priam and Queen Hecuba of Troy. Apollo, Greek god of music, poetry, art, oracles, archery, plague, medicine, sun, light, and knowledge, fell in love with Cassandra the first time he saw her. He offered her the gift of prophecy in exchange for a kiss. Cassandra agreed and she received the gift of prophecy, but she then refused Apollo. The gift could not be taken back, so Apollo cursed her so that nobody would believe her prophecies.

The story behind the name is not as simple as it might seem at first. Some say that the chosen name is actually related to the name of a popular storage technology and believe that the engineers at Facebook chose the name Cassandra because Cassandra was a cursed oracle.

History of Apache Cassandra

Many concepts Cassandra relies on come from Google's BigTable developed in 2005, and Amazon's Dynamo from 2007. Avinash Lakshman, one of the authors of the Dynamo paper, and Prashant Malik developed the first version of Cassandra at Facebook to support the Inbox Search feature.

Facebook then released it on Google Code in 2008. In 2009, it became an Apache Incubator project; it was promoted to a top-level project in 2010. Since then, Cassandra has gone through more than eight releases and is used by more than 400 companies in 39 countries. The impact that Cassandra has had on the NoSQL scene is pretty significant, especially considering that 25 percent of the Fortune 100 companies use Cassandra even though there are literally hundreds of other NoSQL solutions available today.

Let's take a look at some companies and organizations that use Cassandra. Netflix uses Cassandra for storing customer viewing data, eBay uses it for fraud detection and the social component of their platform, Expedia uses it for caching hotel room prices, SoundCloud uses it to store user's dashboards, and CERN uses it for monitoring the data acquisition system of the ATLAS experiment (one of two systems involved into the discovery of the Higgs boson).

As mentioned previously, Cassandra is an open-source project and there are many committers working on it. Cassandra committers are employed at companies such as Facebook, Apple, Twitter, and nScaled—with the majority of the committers coming from DataStax. DataStax's business model is based on providing distribution and support for an enterprise-grade version of Cassandra. DataStax also supports and hosts many Cassandra community-related activities.

Basic Theory behind Cassandra

To understand the principles behind (and the motivation that led to the development of) Cassandra and distributed systems in general, we have to take a step back and look at how it was all done with systems that didn't cope with the scale of data that today's systems often do.

Vertical Scaling

Back in the day, big centralized servers managed all the storage. If the system needed to scale, one bought a server with better performance characteristics and moved the data to the new hardware. This type of scaling is called vertical or upscaling.

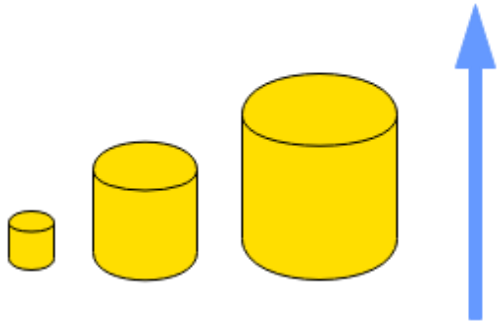

Figure 1: Vertical Scaling

This approach to scaling is limited by currently available hardware, not to mention that the bigger the hardware, the bigger the price. One other issue is that, for instance, if one needed just 10 percent more capacity (or something in that range), new hardware still had to be acquired. Now, there is nothing wrong with this approach; it functioned just fine for decades and is still used a lot today on a day-to-day basis. However, with the development of various systems and an increase in the scale of these systems, the vertical scaling approach hit a wall, not to mention that it became very difficult to move gigabytes and gigabytes of data around for even the smallest changes on the system. Also, losing a server often meant losing data or involved a complicated process to restore it.

Relational Databases

For many decades, relational databases were the main technology when handling data. In a relational database, a unit of work performed against the database is called a transaction. Relational databases are often referred to as transactional systems, and the relational database basically has to provide an environment for executing transactions.

Transaction properties are:

- Atomicity: If one part of the transaction fails, the whole transaction fails.
- Consistency: Every change has to be in accordance to constraints, triggers, etc.
- Isolation: Ongoing transactions don't interfere with one another.
- Durability: After the transaction is finished, the data remains stored in case of a failure.

The transaction properties (often referred to as ACID) are harder to provide when the data is distributed across multiple physical machines. For instance, ACID would require distributed locking of resources. This locking becomes a problem when the number of transactions increases and the system starts to spend more and more time on coordinating while nodes wait for other nodes to finish and acknowledge operations.

Oftentimes, some sort of master/slave architecture is used. The master decides where the data is stored and waits for the slaves to respond. The scaling problem is twofold: throughput is dependent on the master's capacity and, the more nodes that are included in the system, the more communication that is required to maintain the ACID properties of the transactions.

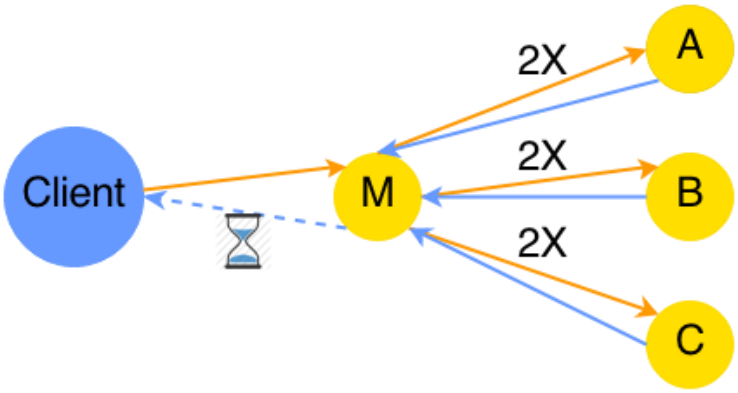

Figure 2: Client waiting for operation to finish in an ACID-distributed system with master node

Take into account that in the previous figure, the number of requests from master to slaves is actually twice as many because, in the first phase, the master notifies the slaves of what the change is going to be; every slave then acknowledges if it's all right. The master then has to notify the nodes that the other nodes agreed and that the transaction went through, and it has to wait for their response once again.

CAP Theorem

In 2000, Eric Brewer published the CAP theorem. Seth Gilbert and Nancy Lynch of MIT formally proved the theorem in 2002. In short, the theorem states that a distributed system can simultaneously provide just two of the three following guarantees:

- Consistency: Every read gets the data from the most recent write.
- Availability: Every running node always responds to a query.
- Partition tolerance: The system continues to operate despite node outage.

This choice is often depicted with the following triangle.

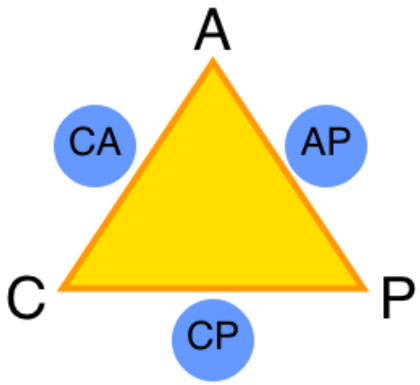

Figure 3: CAP Theorem Choices

Although there's a fine amount of theory behind this "two out of three" rule, Brewer stated in 2012 that it's a bit misunderstood. First, most systems aren't partitioned, so choosing between A and C is oftentimes not really necessary because P is irrelevant within a one node system. Second, the choice between A and C can happen at many levels within the system itself. Finally, all three properties are more continuous than binary. Brewer also pointed out that CAP consistency is not the same as ACID consistency, and that CAP only refers to single-copy consistency which is a pure subset of ACID consistency.

Note: In distributed systems, it comes down to availability or consistency.

Previous reasoning around misunderstood concepts in CAP might seem a bit complicated but "two out of three", in practice, comes down to "one out of two", so it's more like picking between "A" and "C". To have availability, data is replicated to different machines. Consistency is achieved by updating other nodes before allowing further reads. So, systems allowing reads before other nodes are updated get a high "A". Systems that lock other nodes before allowing reads have a high "C". With Cassandra, this choice is more toward "A".

Consistent Hashing

Cassandra is a peer-to-peer system. We have already mentioned a couple of times that having some node or other central place to distribute data within the cluster is, by itself, very inefficient when we pass a certain size cluster. So, the basic idea is to provide some way for the nodes to know on which node within the cluster to place the data. For now, let's assume that the data key is going to be a really large number—large enough for us to pick it randomly without any fear of this number appearing again. Let's call it "RowID".

The simplest algorithm to distribute this data to *N* nodes would be to divide RowID by *N* and then look at the remainder. The remainder will always range from 0 to *N*–1. If we assign all of the keys with the remainder of 0 to Node A, 1 to Node B, 2 to Node C, and so on, we would have a nice way of knowing which node a row belongs to. Now, as great as this might sound, this way of distributing data has a very serious weakness in that when the number of nodes changes, all of the data stored in the cluster actually starts belonging to some other node. In practice, this would mean that taking any node down while the system is running would cause the cluster to become unresponsive and probably become useless until all of the data was shifted to the appropriate server.

One way of avoiding shifting data on such a scale is to partition the whole space of possible RowID values into equal sizes.

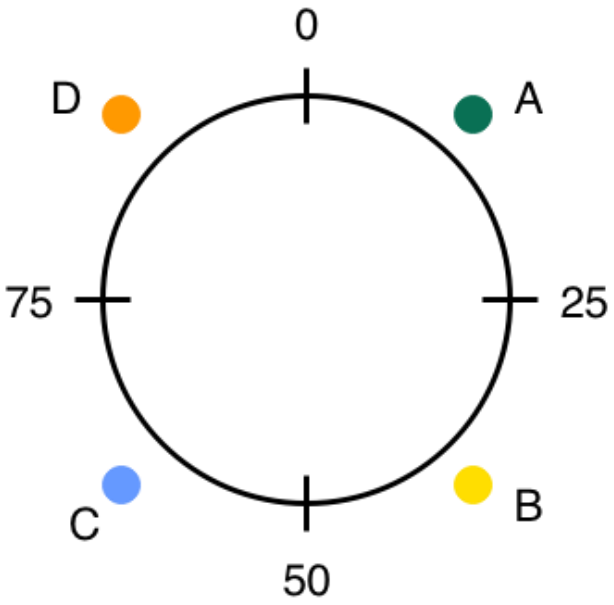

Figure 4: Partitioning with a range of values

Figure 4 shows RowID values from 0 to 99. Remember, RowID is a very large number representing a row, but for simplicity we're working with a range from 0 to 99. The first 25 RowID values belong to A, the next 25 to B, and so on. If, for some reason, another node joins the cluster, only a part of the data would be shifted between the nodes, not all of it. Cassandra functioned like this for a long time. One didn't specify the numbers from 0 to 99, but used special tools to generate these boundary values for nodes, and the boundary values were held in configuration files. The weakness of this approach was that it had to be done manually, and thus was prone to errors.

Before going into a final round of explaining what consistent hashing is and why it is used, let's take a step back. Previously, we had this large number, RowID, identifying the data. We assumed it was just some random number, but that's not quite the truth of it. Every piece of data in Cassandra has a row key, some deterministic value that identifies the row we want to store and manipulate.

This value is determined on an application level and is used to retrieve data when necessary. Usually it is referred to as a key. This key value is unique, but since it probably isn't a number (i.e. it is a username or e-mail address), it's not possible to map it directly to a unique number that we could use to determine which node this data belongs to. Being able to deterministically get a number for storing the data from the application key is very important. The function that enables us to turn any size key into a fixed-sized number that we need to manipulate the data is called a hash function. In Cassandra's case, the Murmur3 hash function is used, and it has a range from -2^{63} to $2^{63}-1$.

To avoid manual assignment of hash function ranges to nodes, a simple algorithm could be used. We could, for instance, take a node's name and calculate its hash value, which would produce a number. If we did this for every node in the cluster, we could partition the possible hash ranges with boundary marker values defined by the numbers produced as hash values of the node names.

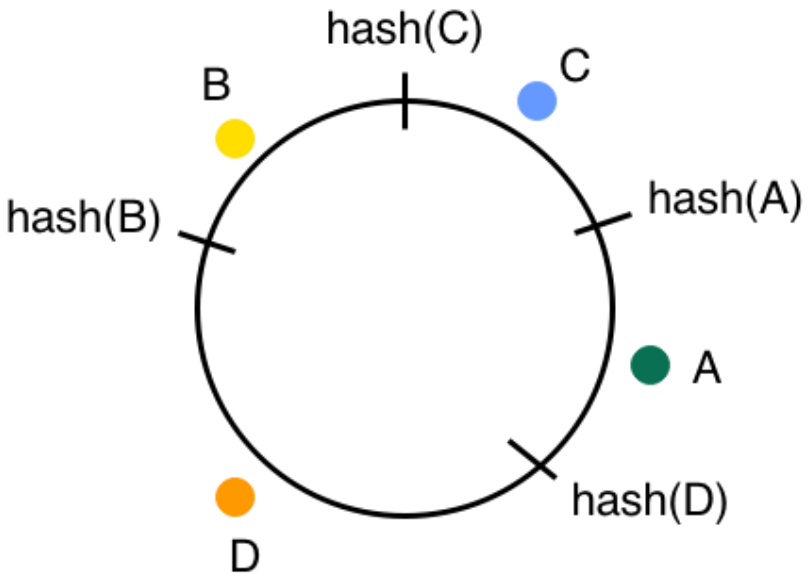

Figure 5: Dynamic partitioning with hash values of node names

The technique shown in the previous figure is called consistent hashing. Note that there are no guarantees of any kind for the partition sizes or their ordering. The algorithm is pretty simple and any node participating in the cluster can reconstruct it and know where a row with some hash value belongs. Whatever the result of a hash function on a row key is, the node is found by moving clockwise or counterclockwise until a marker belonging to the node is found. The calculated RowID then belongs to the found node's marker value.

The output of the hash function is not guaranteed to fall into any kind of symmetrical range. As depicted in the previous figure, some nodes (Node D in our case) may hold much more data than others. This could lead to one node being underutilized and another working under heavy loads. To distribute this data evenly among nodes, one could create a lot of smaller partitions. Theoretically, there is no practical limit on how many markers we set into this range. If we simply combine some predefined numerical or alphabetical prefix or suffix with the node's name and put the resulting hashes in the previous range as markers, we would probably get a much better distribution of ranges. The algorithm would remain the same: move clockwise or counterclockwise until a marker is found. By default, Cassandra puts 256 markers for every node in the range.

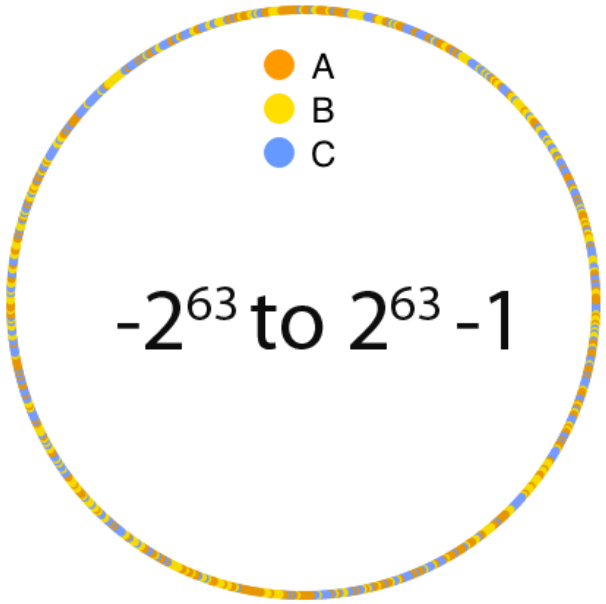

Figure 6: Sample of Murmur3 hash distribution on Cassandra cluster with three nodes

If we want some nodes to hold smaller amounts of data, we assign them fewer markers (or more markers for larger amounts of data). Having more markers in hash output distribution means a greater probability of getting more data and vice versa. This technique provides many advantages in that, if we add a node to the cluster, all nodes are going to distribute a bit of their data to the new node. If a node leaves the cluster, the remaining nodes will split its data almost equally. Later we will cover Cassandra's virtual nodes technology. As complex as it might seem, virtual nodes are just a lot of markers on the output range of the Murmur3 hash function as depicted earlier.

Architecture Overview

Cassandra is a row-oriented database. Every row is identified by its key. One of Cassandra's limitations is that a single row must fit on a single node. The placement of the rows on the nodes happens automatically because there is no central node determining where a row is stored. All of the nodes use the same algorithm which determines the data distribution, basically calculating a hash value for the row key and looking up the nearest marker.

This hash function has ranges and, depending on the available nodes in the cluster, the node is responsible for a certain number of rows. If some node leaves the cluster, the key hashes remain the same but some other node or nodes become responsible for the hash ranges the failed node was responsible for. The story is similar when a new node is added to the cluster.

It's important to note that a single row is usually not stored on just one node because of the fault tolerance of the system. One row is usually stored on one node only in development and test environments. Having rows stored on multiple nodes is important because any node can be taken out or have some sort of failure at any time.

Replication Factor

When talking about data replication, one of the first concerns is how many copies of a single row are actually needed? This question is not easy to answer; it depends on a lot of circumstances. When discussing data replication, the most important term to remember is replication factor, which is the number of nodes that store the same row.

For instance, replication factor two guarantees that there will be two copies of data on the nodes in the cluster. Choosing two is fine for covering single-node failure and is a minimum for production-level deployments (although not advisable because if anything happens to one node, the remaining node has to handle all of the requests, which is always a bad idea).

When nodes fail more often, a higher replication factor is desirable. Also, a higher replication factor will, in some cases, increase the read speed because multiple nodes will respond faster with data parts than a single node transmitting all of the required data. On the other hand, the data replication from node to node will take longer. As a rule of thumb, going below or above 3 must be justified by a design or environment circumstance.

Keyspace and Column Family

Usually, not all data is equally important. Some historical logging information might be less valuable than measurement data. Cassandra allows grouping of data into so-called keyspaces. Generally speaking, a keyspace is a container for the application data. Typically, a cluster has one keyspace per application. But from Cassandra's point of view, the main concern with the keyspaces is to control replication.

Note: Replication is defined on the keyspace level.

The next subunit of storing data is called column family. Column family is a container for an ordered collection of rows. Every row is an ordered collection of columns. Most literature on Cassandra describes column family as something similar to the table in the relational world.

Replication Strategy

Earlier we covered nodes and determining which node the data is going to be stored on. To achieve a specific replication factor, the simplest strategy would be to copy the data to the next node on the hash distribution ring until the desired replication factor is achieved. This strategy is called SimpleStrategy in Cassandra. This strategy is just fine for many smaller-scale systems, but sometimes just copying the data around the data center is not enough. Some systems might have different topologies, and constantly replicating data from coast to coast is not the best solution if every millisecond counts.

The second replication strategy in Cassandra is called NetworkTopologyStrategy. This strategy is the best choice if you plan to distribute your data among multiple data centers. The main difference between the NetworkTopologyStrategy and the SimpleStrategy is that determining the next node to hold the replicated data continues clockwise until a node outside of the same rack is found. Nodes in a rack often fail together because of power failures or facility conditions such as broken air conditioners or faulty network equipment.

The most important problem here is determining the number of replicas within the data center needed to satisfy reads without going to other data centers for the data. With two replicas in each data center, one node can fail at any time and rows will still be readable. Three replicas enable two nodes to fall out and so on. It's also possible to have different replication factors in other data centers; for instance, having three replicas in a data center to serve real-time requests and a single replica in another data center for batch processing or analytics.

Gossip Protocol and the Snitch

The protocol used for sharing node locations and state information about the nodes participating in a Cassandra cluster is called gossip. Cassandra nodes are constantly gossiping; every node initiates a gossip exchange approximately once per second with up to three nodes in the cluster.

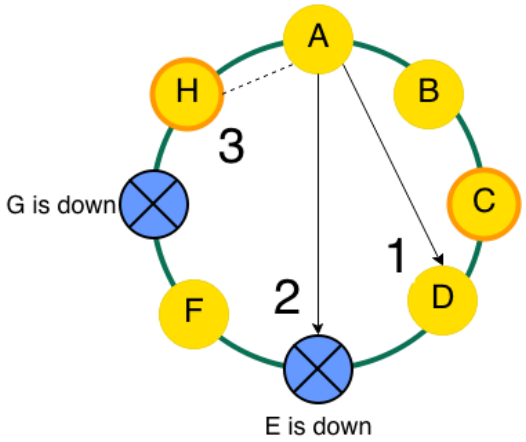

Figure 7: Gossip Interaction

In order to prevent a chicken and egg story, an initial host list is stored in the configuration file. The initial hosts are usually picked so that they are relatively stable nodes in the cluster. In the previous figure, the initial nodes in the cluster are H and C; they are illustrated with a slight border around them.

The nodes stored in this initial list are referred to as seeds. It's important to note that the seed nodes have no special treatment within the cluster. A seed node is more of a gossip protocol, faster initialization, and recovery type of thing. Also, the gossip information is persisted locally by every node so that the node recovers faster in case of a restart. As with any protocol, there are rules by which the nodes send requests to other nodes. The gossip protocol is straightforward; it consists of three steps (with the third step being optional). Every live node always repeats these three steps:

1. Gossip to random live node.
2. Initiate gossip toward a random down node.
3. If the node selected in step one was not a seed node, gossip to random seed node.

The data moved around and communicated between the nodes is then used by the Cassandra component called snitch. Basically, this component picks other nodes for querying and replication based on some heuristics. Snitch configuration is the same for all nodes within the cluster. A snitch can, depending on the configuration, be data center-aware and rack-aware.

Snitch configurations:

- SimpleSnitch: Used in single data center deployments; a snitch configured this way uses no information about the rack or data center. It is a plain "find the next in the ring" behavior.
- Dynamic snitching: Monitors the performance of the replicas by history and chooses the best one based on a simple metric that penalizes long response times and avoids querying nodes that are compacting their data.
- RackInferringSnitch: Uses IP address octets to determine node location. The last part of the IP address identifies the node. The second to last identifies racks, and the third to last identifies the data center.
- PropertyFileSnitch: Allows definition of a cluster topology in a property file. Mostly used if RackInferringSnitch is not applicable because of random and unstructured IP addresses.
- GossipingPropertyFileSnitch: Uses a property file for initial configuration and continues on with gossip to pass that information to other nodes.
- EC2Snitch: Used for relatively simple deployments on Amazon EC2, with all nodes in a single region.
- EC2MultiRegionSnitch: Same as the previous, but treats Amazon's regions as data centers and availability zones as racks.

Coordinator Node

In Cassandra's architecture, all nodes are the same; there are no specialized nodes of any kind. Any node in the cluster can handle a client's read and write requests. The usual way of representing the Cassandra cluster is a ring, mostly because of the inner workings of Cassandra and the consistent hashing function mentioned earlier.

A client can request the information from, or send write requests to, any node in the cluster. It's not important which node the clients choose; all of the clients usually have multiple nodes in their list. If a client has just one node in the list, it gets the list of other nodes when contacting the node from the list for the first time.

In Cassandra terminology, the node performing the operations and requests necessary for the response to the current request is called the coordinator node. The coordinator node acts as a proxy between a client request and the nodes in the cluster that actually have the requested data stored on them. Depending on the network topology, the coordinator node will contact one or more nodes from other data centers, making the contacted nodes coordinators for the contacted data center. On a single data center level, this might look something like the following figure.

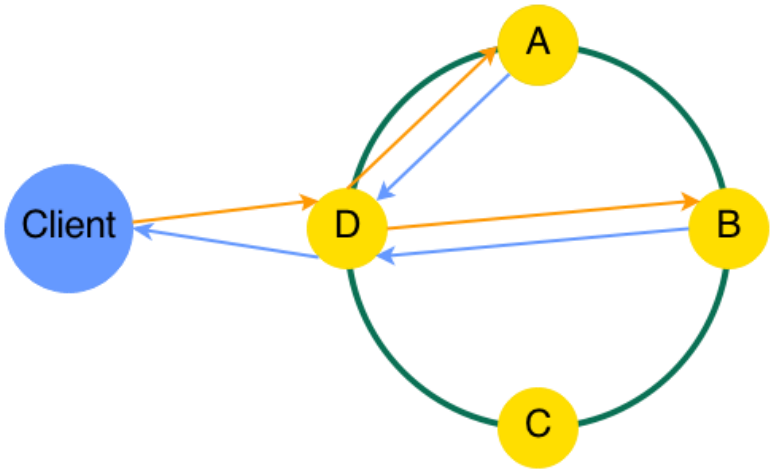

Figure 8: Coordinator node (D) handling client request

The number of requests that go out to the other nodes depends on the replication factor. The bigger the replication factor, the more nodes the coordinator node will query. But, in order to respond to the client request, the coordinator node doesn't always have to wait for all of them. The client might be satisfied with just the first and fastest response from any of the nodes. This depends on the consistency level the client wants for the reads or writes.

Note: The consistency level is specified by the application on the request level.

When writing, the coordinator will sometimes not be able to reach the node responsible for the record; for instance, when the node is down or there is some kind of networking issue. Sometimes not even the replica nodes will be available. If specified by the request consistency level, the coordinator node can then keep this record stored and wait for the nodes to come back up again. This technique is called hinted handoff. Keep in mind that this record will not be visible to any queries as long as it is not written to the nodes responsible for this record. The node holding the hinted handoff data never responds to queries with that data. The hint has all the information needed to transfer the data to unavailable nodes.

A hint includes:

- The location of the unavailable node.
- The row requiring the replay.
- The actual data being written.

By default, hints are stored for three hours. If the replica is unavailable for a longer time, then it's probably severely damaged. In this case, the data has to be re-replicated with the help of the Cassandra repair tool. The hint data is stored in the system.hints table.

Note: In hinted handoffs, the coordinator node saves writes for the unavailable node.

Also, the coordinator node examines the data received from replicas. The data with the newest time stamp is sent to the client. Sometimes it might happen that some replicas have older data. If the coordinator node notices that a replica has out-of-date data, it will send an update for that row. This process is known as read repair.

Consistency

Consistency level is basically the number of nodes that need to respond to a write or read request from a client. Since Cassandra 2.0, there are two types of consistency:

- Tunable consistency: The desired consistency is specified by the client making the request.
- Linearizable consistency: Similar to a 2-phase commit in master/slave systems. Allows operations to be performed in sequence without other operations interfering.

When talking about tunable consistency, there are differences between the read and the write request types.

Before discussing consistency further, it's important to understand what a quorum is. In essence, a quorum is just a number determining the minimum number of replicas needed to respond to a request in order for it to be considered successful.

Note: Quorum is calculated by rounding down (replication_factor / 2) + 1.

With a replication factor of 3, a quorum is 2. So, in a replication factor of 3, the quorum consistency level tolerates one node being down. For a replication factor of 5, a quorum is 3, tolerating 2 nodes down.

Let's start with the read consistency levels and then look at the write consistency levels.

The Cassandra read consistency levels are:

- **ALL**: Returns the record with the newest time stamp. Fails if one replica doesn't respond.
- **EACH_QUORUM**: Returns the newest row after a quorum of nodes from each data center responds.
- **QUORUM**: Returns a record after a quorum of replicas has responded.
- **LOCAL_QUORUM**: Returns the newest time stamp with a quorum in the current data center.
- **SERIAL**: Used with linearizable consistency level.
- **LOCAL_SERIAL**: Used to achieve linearizable consistency level within the data center.
- **ONE**: Returns the closest replica as determined by the snitch.
- **LOCAL_ONE**: Returns the closest replica in the local data center determined by the snitch.
- **TWO**: Returns the most recent data from the two closest replicas.
- **THREE**: Returns the most recent data from the three closest replicas.

The Cassandra write consistency levels are:

- **ANY**: At least one replica must be written to or a hinted handoff must be written.
- **ALL**: A write must be written to the commit log and memtable on all replica nodes.
- **EACH_QUORUM**: A write must be written to the commit log and memtable on a quorum of nodes in all centers.
- **QUORUM**: A write must be acknowledged by a quorum of replicas.
- **LOCAL_QUORUM**: A write must be acknowledged by a quorum of replicas in the local data center.
- **SERIAL**: Used to achieve linearizable consistency.
- **LOCAL_SERIAL**: Used to achieve linearizable consistency in the local data center.
- **ONE**: A write must be acknowledged by one replica.
- **LOCAL_ONE**: A write must be acknowledged by at least one node in the local data center.
- **TWO**: A write must be acknowledged by two replicas.
- **THREE**: A write must be acknowledged by three replicas.

Keep in mind that the data will still be sent to all replicas depending on the replication factor when read or written. The consistency level is set to determine how many replicas must respond for the coordinator to send the response back to the client.

With linearizable consistency, the story is a bit different. This is related to strong consistency in distributed systems. In other words, the reader always sees the most recent written value. But sometimes there are operations that have to be performed in a sequence without other operations interfering. This is called linearizable consistency. The simplest case for linearizable consistency is the problem of having a unique username or e-mail in the system. Before writing a username to the cluster, we have to make sure there are no other records with that username. Furthermore, no data may be written with this username from the time we checked to the time we issue a write request. This problem in Cassandra is addressed by using the Paxos consensus protocol.

In the Paxos consensus protocol, any node can propose a value; this node is called the leader. Multiple nodes can act as leaders at the same time, but before becoming a leader, every replica node must process the leader's request. The protocol runs in four phases and is request-intensive, but there are few occasions where it has to be used. As mentioned earlier, it's about ensuring the uniqueness of a globally unique username, e-mail, or other piece of data.

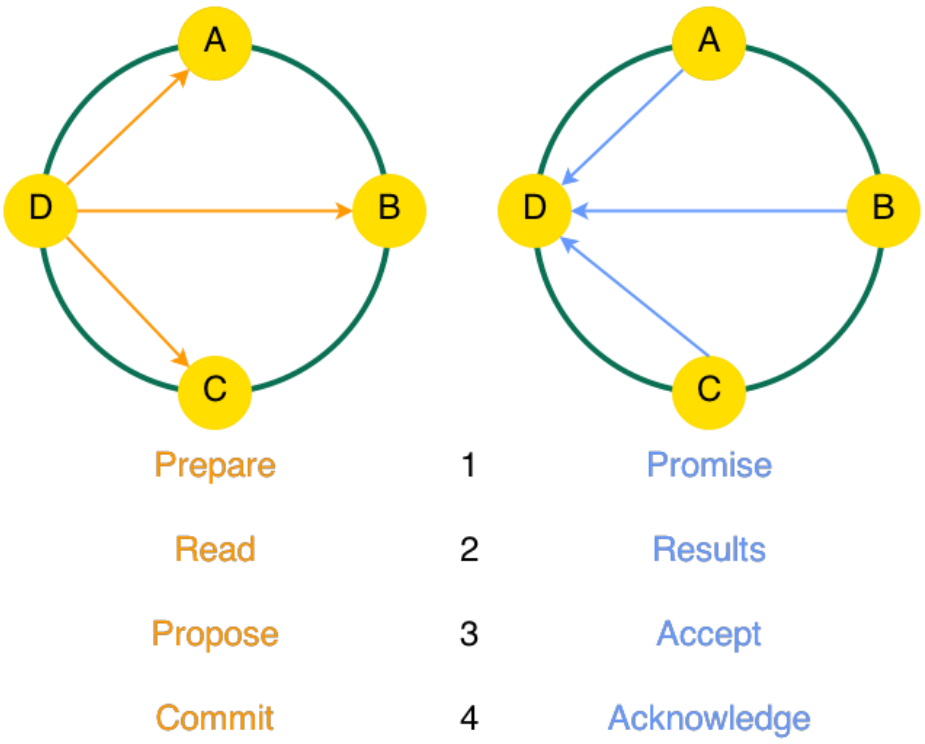

Figure 9: Four phases of linearizable consistency

Cassandra Internals

Traditional relational database storage systems spend a lot of resources on structuring data and maintaining relational integrity. In Cassandra, maintaining the relational integrity is not an issue because Cassandra's tables are not related, and the relation resolution is left up to the application. On most occasions, Cassandra is much faster with writes than traditional database systems. This is all related to the internal workings of Cassandra.

Write Path

The processes related to writing data in Cassandra are:

- Writing the data to the commit log.
- Writing data to the memtable.
- Flushing data from the memtable.
- Storing the flushed data into SSTables.
- Compacting the data.

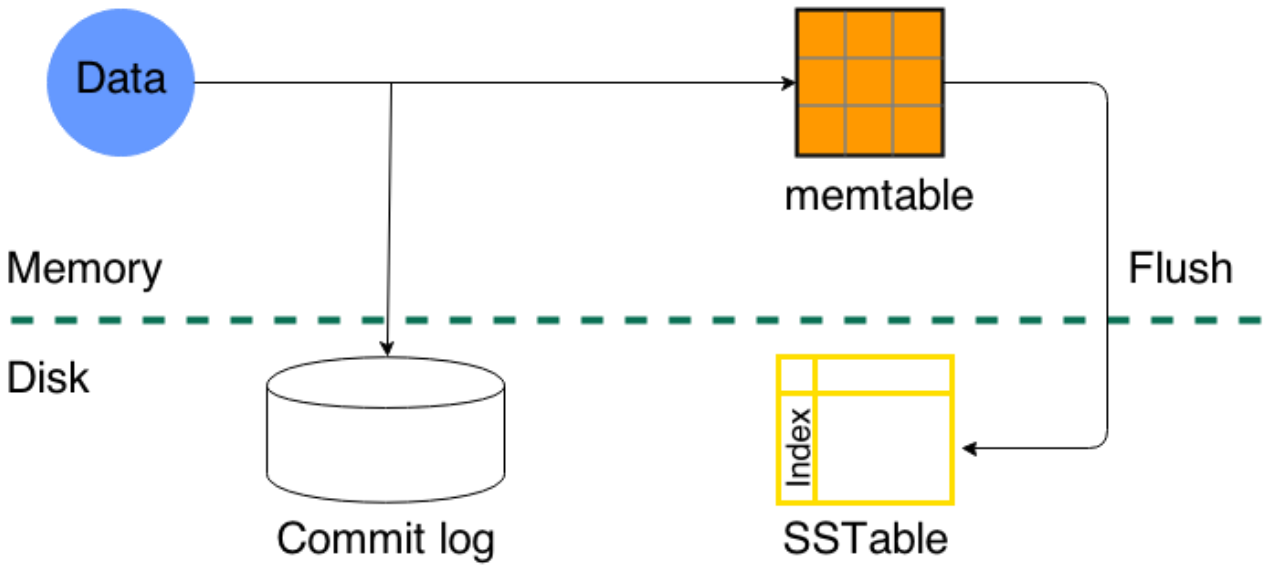

Figure 10: Cassandra write path

The data is acknowledged as written when Cassandra stores the data into the memtable and the commit log. The commit log stores every write made to the node in append-only fashion, without any random seeks to the disk. The memtable holds the written data up to a certain configurable limit. After reaching the limit, memtable data is flushed to the SSTables on the disk. When flushing the memtable to SSTable, the SSTable remains immutable; the data is usually stored across multiple SSTables. After flushing the data to the SSTable, Cassandra creates two in-memory data structures: a partition index with primary keys and the starting position of a row in the SSTable file. The partition summary, a subset of the primary key set, is usually every 128th key.

Data Update, Removal, and Compaction

Cassandra does not do any insert or update operations on written data. This would require random I/O operations, which are not very efficient. If data is updated or inserted, Cassandra simply writes new data with a new timestamp into the memtable and then into the SSTable. As time progresses, the redundant data accumulates; this redundant data is then removed in a process called compaction.

The delete operation is also not done in place, but the story is a bit different. Cassandra simply marks the data for deletion. This marker is called a tombstone. Data marked with a tombstone exists for a preconfigured period of time, usually 10 days. The data is not removed immediately because of other nodes that might still have this data not marked with a tombstone. As soon as the node that received the delete request gets a read request, it would respond that it doesn't have the record. But, since other nodes may still have the record stored, the coordinator node would conclude that the data needs to be replicated to the node that actually deleted the data. This is where the tombstone comes into play. When the coordinator receives data with a tombstone mark, it notifies other nodes that this data is deleted, and the other nodes then also mark it with a tombstone.

Note: A deleted row is marked with a tombstone and not removed completely because of the distributed nature of Cassandra. Without tombstones, other nodes might think a node missed a write and re-create the deleted row.

Inserting new data and marking data with tombstones alone would eventually fill up the disk space. From time to time, Cassandra runs compaction on the data, which removes deleted and redundant data from the SSTables.

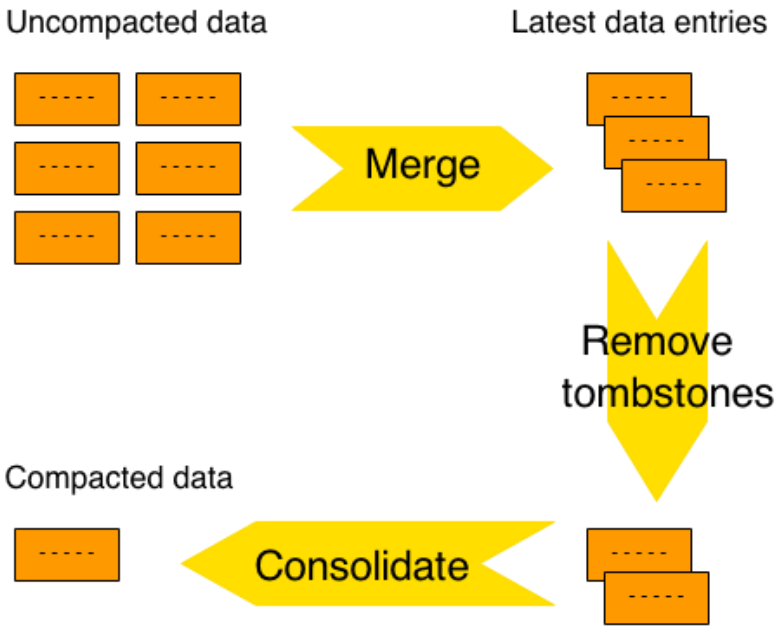

Figure 11: The compaction process

The downside of compaction is that the Cassandra instance is usually configured to use only up to half of the available storage to make the compaction process possible. During compaction, the system doubles the existing data and then optimizes it.

Read Path

When reading data, the data can be found in multiple locations, such as:

- In-memory memtable.
- Memtable being flushed.
- One or multiple SSTables.

All of these locations need to be checked and then compared with a time stamp to determine what the relevant data is. Finding data in the memtable is relatively simple and doesn't cost a lot from the performance side. Very often, the data will actually be located in one or more SSTables.

SSTables have been mentioned a couple of times now, but we haven't gone into detail about them—except that they are memtables flushed to the disk. We also mentioned the SSTable file. In reality, there are three separate files created for every column family:

- Bloom filter
- Index
- Data

A Bloom filter is a relatively old and well-known technique for determining whether or not an element is in a set. Burton Howard Bloom introduced the technique in 1970. There is a lot of theory and statistics behind the Bloom filter, but basically it is a long bit array initially filled with zeroes. To add an element to the Bloom filter, its hash value is calculated. Since the Bloom filter requires multiple bits per element, the structure can be relatively large but is still only a fraction of the original data set size and is suitable for keeping in memory. The data is hashed and the resulting bit ones are put into this long bit array. Most of the hash functions have an output of 128 bits or something in that range, so when calculating the value for the Bloom filter, there are often multiple hash functions invoked upon the key to spread the bits evenly in the filter and to have the hash output equal to the Bloom filter size.

Also, the Bloom filter is more about probability. If the hash value is found in the filter, then that means that the searched value might be there. If the hash value is not found in the filter, that means the element is definitely not in the set.

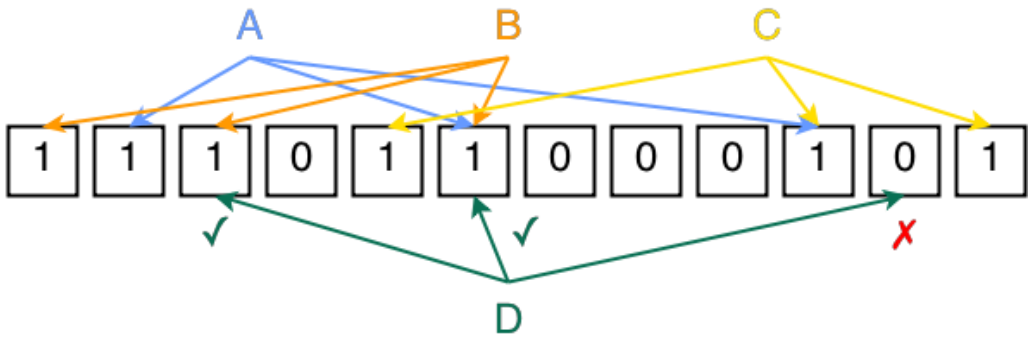

Figure 12: Example of a Bloom filter with values A, B, and C. Value D is not found in the filter.

The previous figure shows how element D isn't found in the filter. Still, sometimes it can happen that the searched value would be found in the filter, but actually going into SSTable and looking for the value would return no data. This case is called a false positive.

Note: The Bloom filter is not 100 percent certain. False positives are possible.

Determining whether the searched row is in the Bloom filter of an SSTable is just the first step to reading data. Before going further, it's important to mention two Cassandra data caches:

- Partition key cache: Index with partition (row) keys.
- Row cache: For every read row, an entire row is put into memory (multiple SSTables).

Now, caching might always seem like a good idea but there are guidelines as to when and how to use caching in Cassandra:

- Rarely-read data and data with very long rows should not be cached.
- Try to keep the load per node as low as possible.
- Separate heavily-read data into multiple column families.

Now let's get back to what happens after the Bloom filter returns the results of searching for the data.

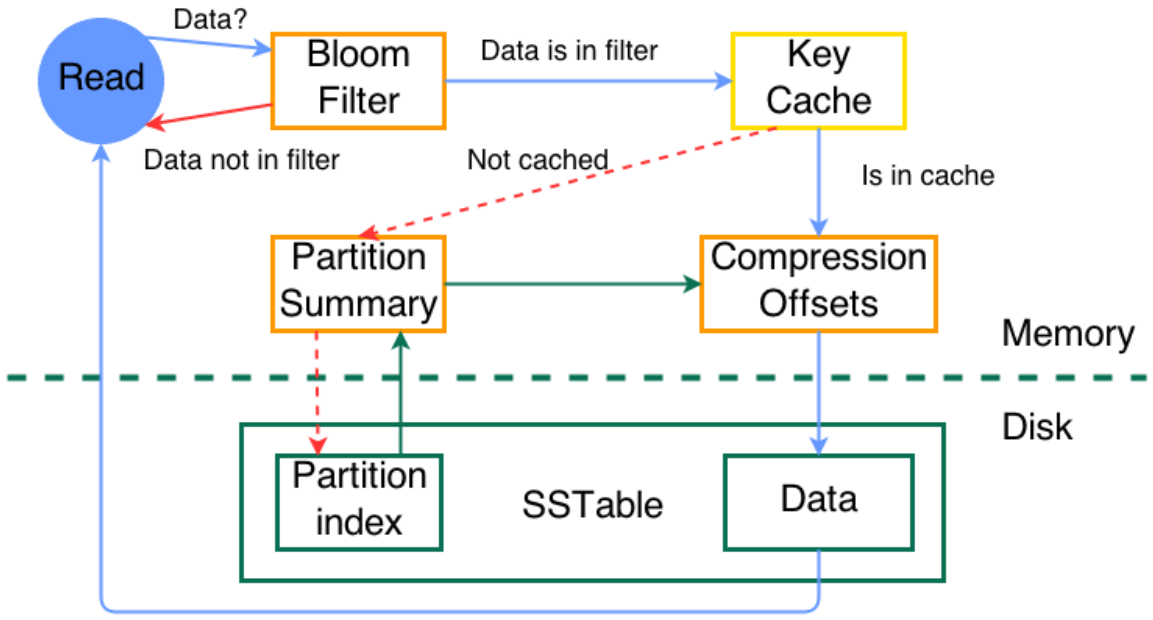

Figure 13: Data reading in Cassandra

If the Bloom filter returns that the requested data is not available, the response is returned to the client as saying there is no data that matches the request. If the Bloom filter says that there is data, it's looked up in the cache. If the cache contains it, the location of the data is calculated from compression offsets and then it's retrieved from the disk. If the data is not in the cache, Cassandra reads the partition summary and tries to approximate the position of the searched data based on the partition summary. After that, a disk seek is made to read the index and determine the exact location of the data. Before actually loading the data from the disk, the process needs to check compression offsets.

Summary

Some parts of this introductory chapter compared Cassandra with relational databases. It's important to note that there is nothing wrong with classical relational databases, but they do have a problem when it comes to horizontal scaling. Most of the classical systems simply come to a point where adding new nodes causes little or no change in the system performance. In short, there are simply areas where storage systems such as Cassandra are a better fit.

The strengths of Cassandra are:

- Fast reads and writes.
- Adding new machines causes linear increase in performance.
- Easy and reliable setup of cross-datacenter replication processes.

Cassandra uses and relies on a lot of principles not usually found in relational databases. This chapter was more about the concepts and ideas behind Cassandra and not so much about actually using or interacting with Cassandra. Getting started with Cassandra is probably possible without getting to know every approach Cassandra uses for the challenges it encounters, but many readers have been in the relational world for a longer period of time, and some might even want to do something with Cassandra right away. Getting to know the background of Cassandra provides a solid foundation for better absorption of the information to come.

Chapter 2 Getting Started with Cassandra

Cassandra is open-source software. Getting started with Cassandra is completely free. The most basic version sufficient for getting into Cassandra is the DataStax Community Edition. The download link changes from time to time, but at the time of writing it is currently available here.

The DataStax Community Edition of Cassandra supports the following systems:

- CentOS 5.X and 6.X
- Red Hat Enterprise Linux 5.X, 6.X
- Debian 6.X
- Ubuntu 10.X, 11.X, 12.X
- Mac OSX 10.X
- Windows OS family

The range of systems able to install Cassandra is pretty wide and covers all of the most popular operating systems out there today. To get started, it's even possible to just download the DataStax Sandbox virtual machine containing all of the necessary software to try Cassandra out preinstalled. However, registration is required and there are licensing terms users must accept. Enterprise-grade distributions are also available but they will not be covered here. Most of the download links for DataStax Community Edition have a tarball download option. In the next section, we'll cover how to install Cassandra on CentOS 6.X. The other systems are pretty similar. The Windows platform has its own installer and we'll describe it, too.

Installing Cassandra on Linux with a Tarball

This section will show you how to install Cassandra on CentOS 6.X from scratch. Cassandra actually runs on top of Java, so we will first have to install Java. Most Linux distributions actually come with a preinstalled Java, but we will cover installing Java just in case.

Installing Java

Skip this step if the system has the appropriate version of Java installed. The easiest way to check this is with the **`java -version`** command. The latest version of Oracle Java 7 should be installed. To download Java, the **`wget`** command is used. If the **`wget`** command is not available on the system, install it. For CentOS, it's as simple as **`yum install wget`**. If the following **`wget`** command fails for any reason, download the archive manually and continue from there.

```
# cd /opt/
# wget --no-check-certificate -c --header "Cookie: oraclelicense=accept-
securebackup-cookie" http://download.oracle.com/otn-pub/java/jdk/7u60-b19/jdk-
7u60-linux-i586.tar.gz
# tar xzf jdk-7u60-linux-i586.tar.gz
# cd /opt/jdk1.7.0_60/
```

```
# alternatives --install /usr/bin/java java /opt/jdk1.7.0_60/jre/bin/java 2
# alternatives --install /usr/bin/javaws javaws /opt/jdk1.7.0_60/jre/bin/javaws 2
# alternatives --install /usr/bin/javac javac /opt/jdk1.7.0_60/bin/javac 2
# alternatives --config java
There are 2 programs which provide 'java'.
  Selection    Command
-----------------------------------------------
*  1           /opt/jdk1.7.0_60/bin/java
 + 2           /opt/jdk1.7.0_60/jre/bin/java
Enter to keep the current selection[+], or type selection number: 2 [ENTER]
# java –version
java version "1.7.0_60"
Java(TM) SE Runtime Environment (build 1.7.0_60-b19)
Java HotSpot(TM) Client VM (build 24.60-b09, mixed mode)
# export JAVA_HOME=/opt/jdk1.7.0_60
# export JRE_HOME=/opt/jdk1.7.0_60/jre
# export PATH=$PATH:/opt/jdk1.7.0_60/bin:/opt/jdk1.7.0_60/jre/bin
[Add these to ~/.bashrc or the exports will not be set on the next boot]
```

Code Listing 1

Installing DataStax Community Edition of Cassandra

Once Oracle Java 7 is installed, we need to download Cassandra. After downloading Cassandra, there are couple of things that we need to check, including the location of directories where Cassandra stores data.

```
# cd /opt/
# wget http://downloads.datastax.com/community/dsc.tar.gz
# tar xzf dsc.tar.gz
# sudo mkdir /var/lib/cassandra
# sudo mkdir /var/log/cassandra
# sudo chown -R  $USER: $GROUP /var/lib/cassandra
# sudo chown -R  $USER: $GROUP /var/log/cassandra
# cd /opt/dsc-cassandra-2.0.9/bin

[Starting Cassandra in foreground mode]
# ./cassandra –f
[If you don’t see error or fatal stack traces you are running Cassandra, but there
will be a lot of logging. To stop Cassandra, simply press Ctrl + C]
[Or start Cassandra in daemon mode. Log will jump out, but pressing Ctrl + C gets
out of it and Cassandra will continue to work]
# ./cassandra
[To shut down daemon Cassandra use]
# pkill -f CassandraDaemon
```

Code Listing 2

As you can see from the previous code example, starting Cassandra usually takes just a couple of minutes and nothing more. Take into account that this is a more complicated version of installing Cassandra and that there are easier solutions that include system package managers. The installation from a tarball was shown here because it will work on most of the available Linux environments out there today.

Installing Cassandra on Windows

The most popular version of Microsoft Windows at the time of writing is Windows 7. The rest of the Windows systems are either not officially supported or have a much smaller market share than Windows 7.

The installation procedures will be explained for Windows 7. If you're installing Cassandra on a 32-bit system, download and install Microsoft Visual C++ 2008 Redistributable Package (x86) from Microsoft's website. The web utilities that accompany Cassandra do not support Internet Explorer; install Chrome or Firefox if you want to use these tools. You will not need these tools to follow along with this book, however.

There are also some hardware prerequisites. DataStax Community Edition of Cassandra for Windows is preconfigured to use 1 GB of RAM. An operating system such as Windows 7 usually takes at least 1 GB to run smoothly, so it's recommended to have an environment with at least 2 GB of RAM available. Be very careful not to allocate too little memory to the system because the installation might complete without warning you about the RAM requirement, and Cassandra will simply not start when you try to run it.

Tip: DataStax Community Edition of Cassandra for Windows requires 1 GB of RAM or more.

Installing Java

Java for Windows is available here. Agree to the terms and conditions on that page to download the Java installer, and then run the downloaded software.

Figure 14: Initial screen of Java installer for Windows

Figure 15: Successful Java install on Windows

The previous figures are fairly standard screens when installing something on Windows. Note that during the installation, a security message may be displayed to notify you that you are about to make changes to the system. To install Java successfully, allow the application to make changes to the system.

After installing Java, it's important to set the JAVA_HOME environment variable. The Cassandra installation will run without giving any warning about the missing environment variable, but Cassandra will not actually start if this variable is not set. To check or set the JAVA_HOME variable:

1. Right-click **My Computer**.
2. Select **Properties** from the context menu.
3. Open **Advanced System Settings**.
4. In the System Properties window, click the **Advanced** tab.
5. Click **Environment Variables**.
6. Add or update the **JAVA_HOME** variable as shown in the following figure.

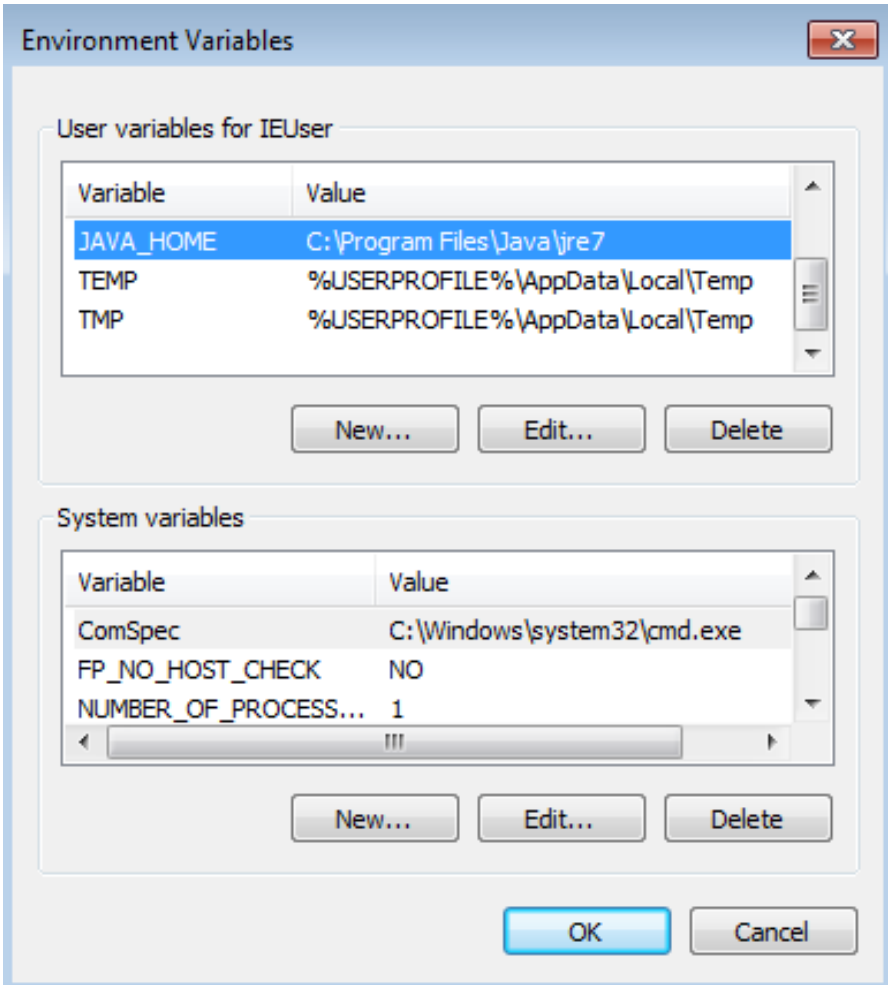

Figure 16: Adding Java environment variable

The value for the JAVA_HOME variable might be different from system to system because the system disk might have different name than "C:", or you may have selected a different destination when running the Java installer. Make sure the path value in the variable matches the path where Java was installed. Cassandra will not start if this variable is not set properly.

Installing DataStax Community Edition of Cassandra

DataStax Community Edition is available here. Download the version for your machine depending on whether you're running 32-bit or 64-bit Windows. The only other option is MSI Installer 2.x. After downloading the file, run the installation as an administrator. The following figures show the steps of the installer on the Windows 7 operating system.

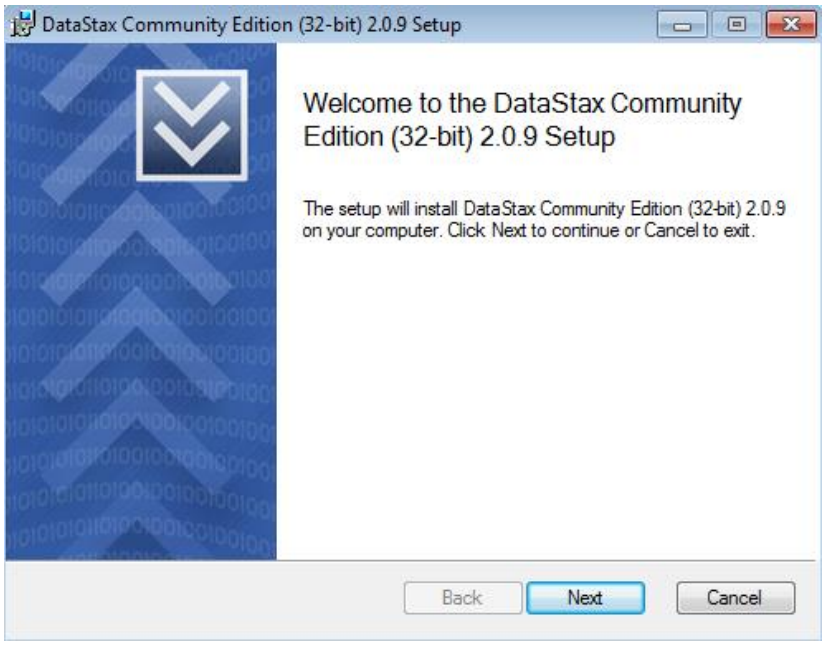

Figure 17: Initial screen of the DataStax Community Edition installer

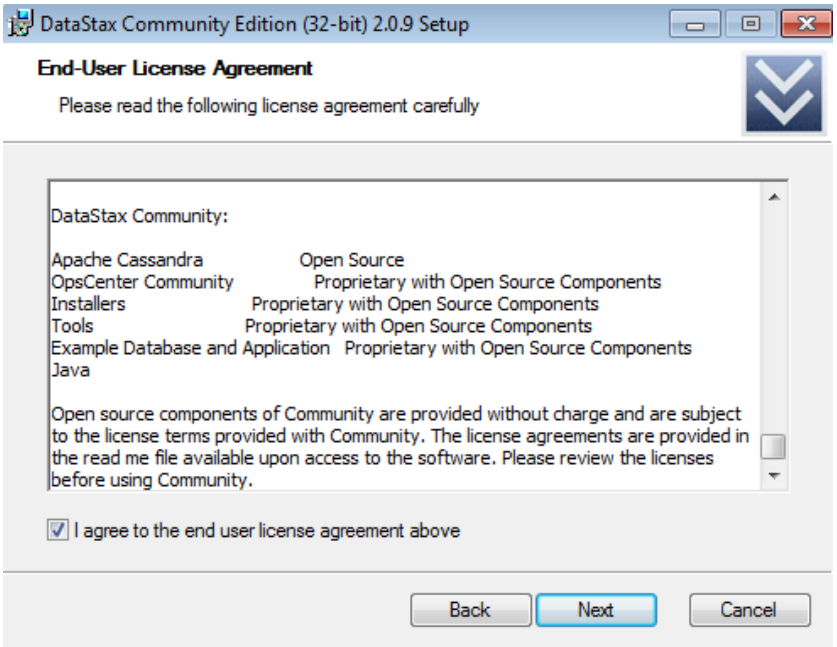

Figure 18: License agreement for the DataStax Community Edition installer

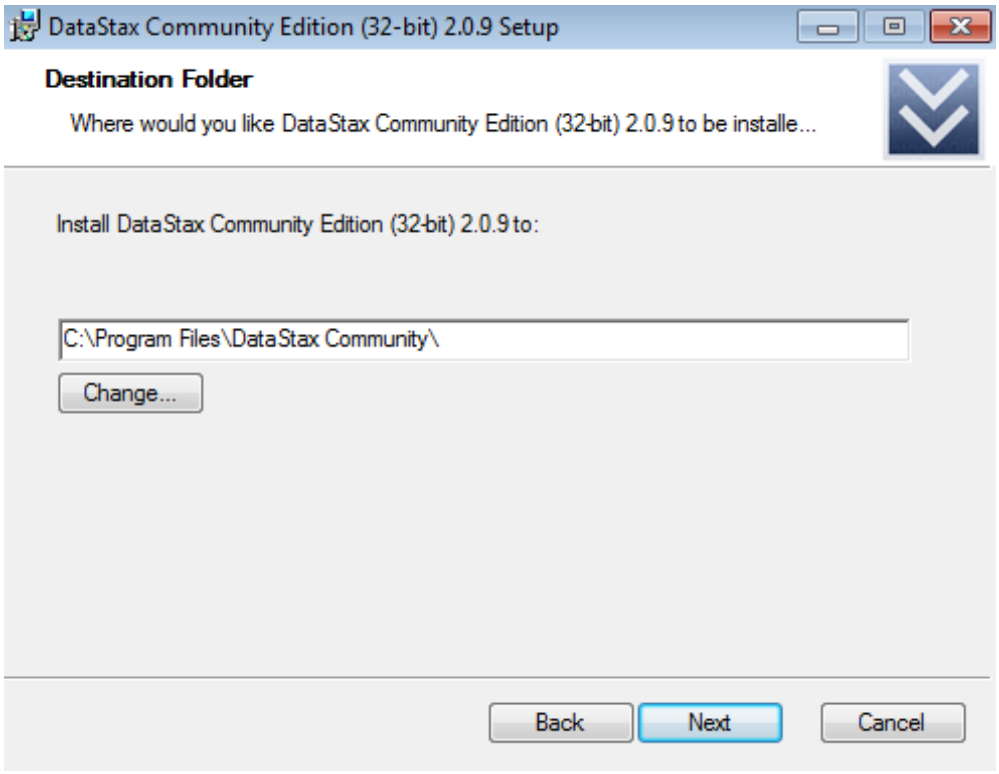

Figure 19: Destination folder selection

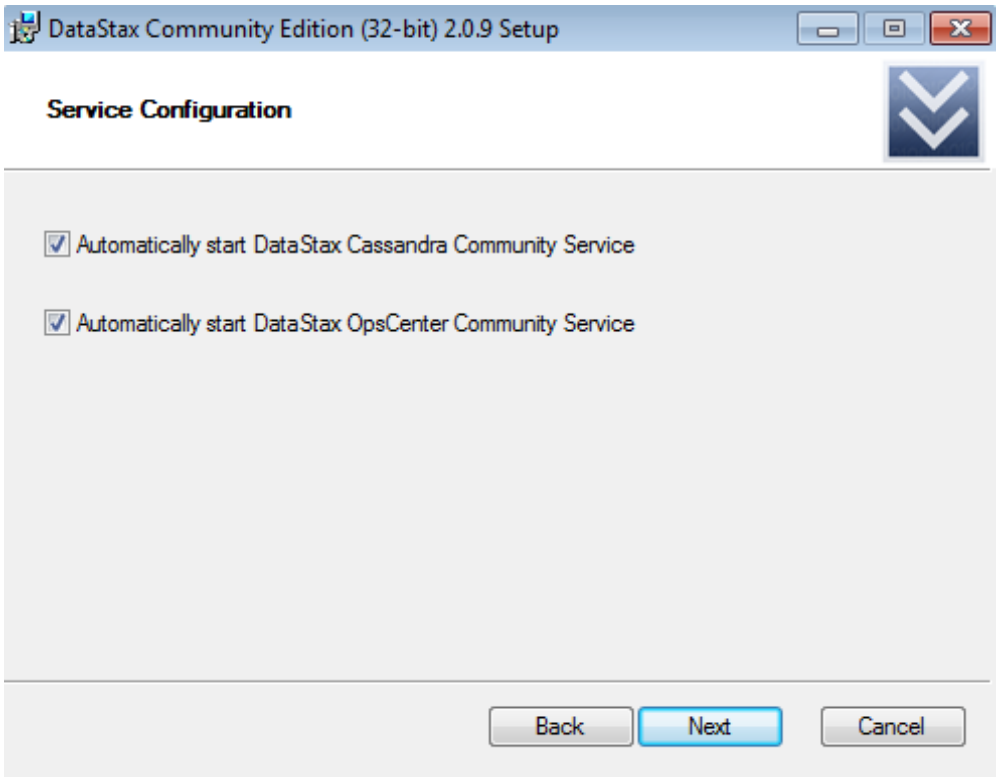

Figure 20: Setting Cassandra to start automatically

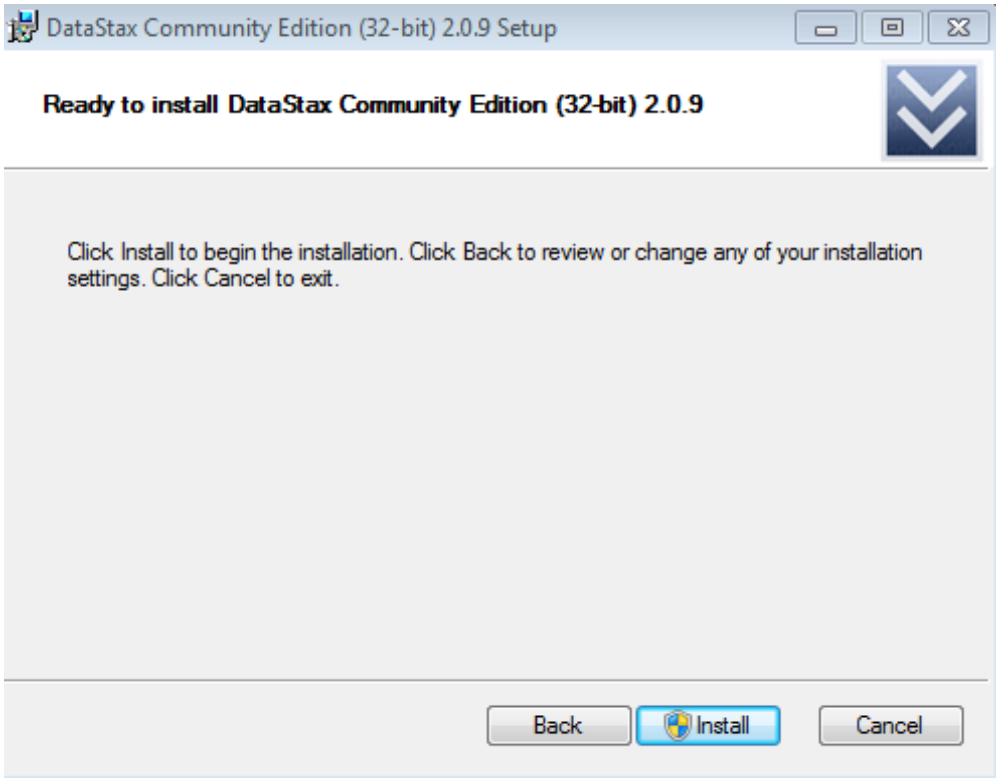

Figure 21: Authorizing changes to the system

Figure 22: Final screen of the DataStax Community Edition installer

The installation process is similar to any Windows application installation. The installation itself shouldn't take more than a couple of minutes. If all of the installation steps were successful, your system should now run Apache Cassandra. Also, some new items should appear in the Start menu as shown in the following figure.

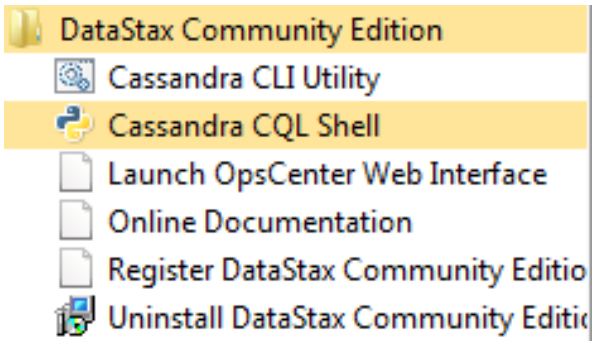

Figure 23: New Start menu items for DataStax Community Edition on Windows

The final check to ensure Cassandra installed would be to click **Cassandra CQL Shell** inside the **DataStax Community Edition** folder in the Start menu. If something like the following figure appears on the screen, then you are ready to work with Cassandra on Windows.

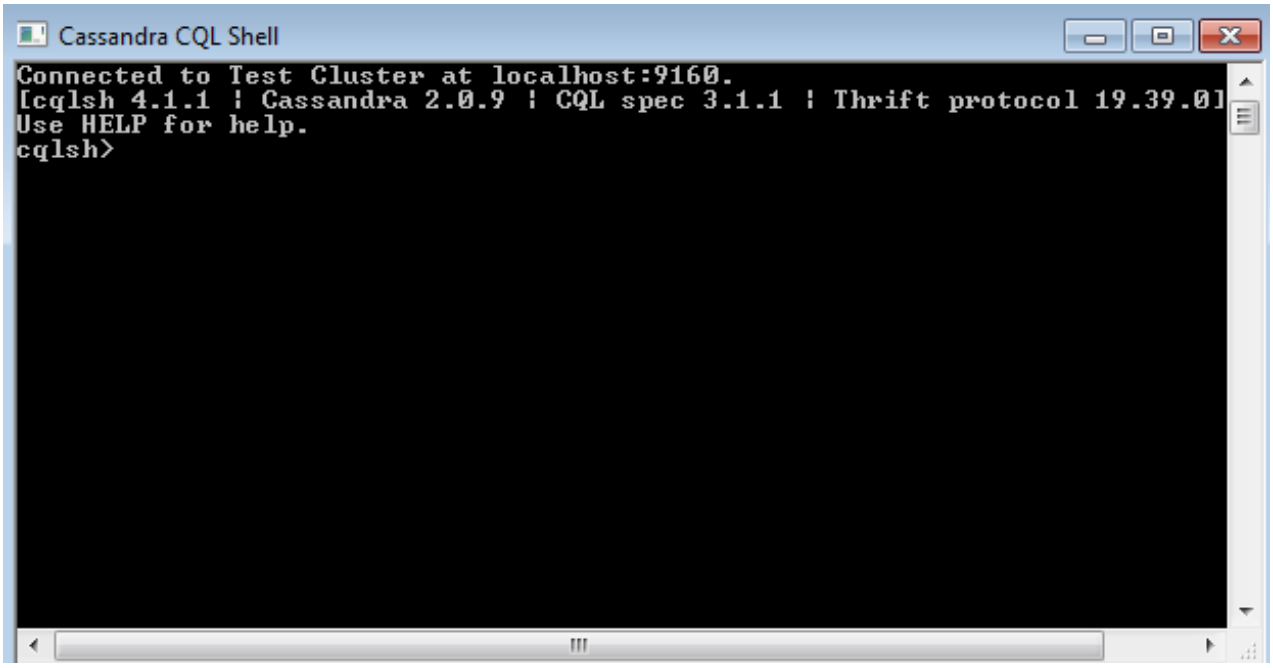

Figure 24: Cassandra CQL Shell on Windows

The CQL shell is a command-line utility for interacting with the Cassandra database. The instructions for the database to run are given in a language called CQL, which stands for Cassandra Query Language. Using CQL will be covered in a later chapter. For now, we won't cover how to use CQL when interacting with the CQL shell. The CQL shell is just fine for most tasks. A more graphical tool for developing with CQL is the DataStax DevCenter.

Installing and Using DataStax DevCenter

The basic tooling shipped with Cassandra has a sufficient set of features for most daily operations that administrators and developers will need to perform. A free visual tool with more advanced features and a friendlier user interface for creating CQL statements is a stand-alone app called DataStax DevCenter.

To install DataStax DevCenter:

1. Go to the download section of the DataStax website at http://www.datastax.com/download.
2. Click the DataStax DevCenter link.
3. Select the operating system you are using and be careful to choose the correct 32-bit or 64-bit option. The download will start automatically.
4. The downloaded application is compressed. Extract it to the desired location.
5. Go to the extracted location and run the **DevCenter.exe** executable.

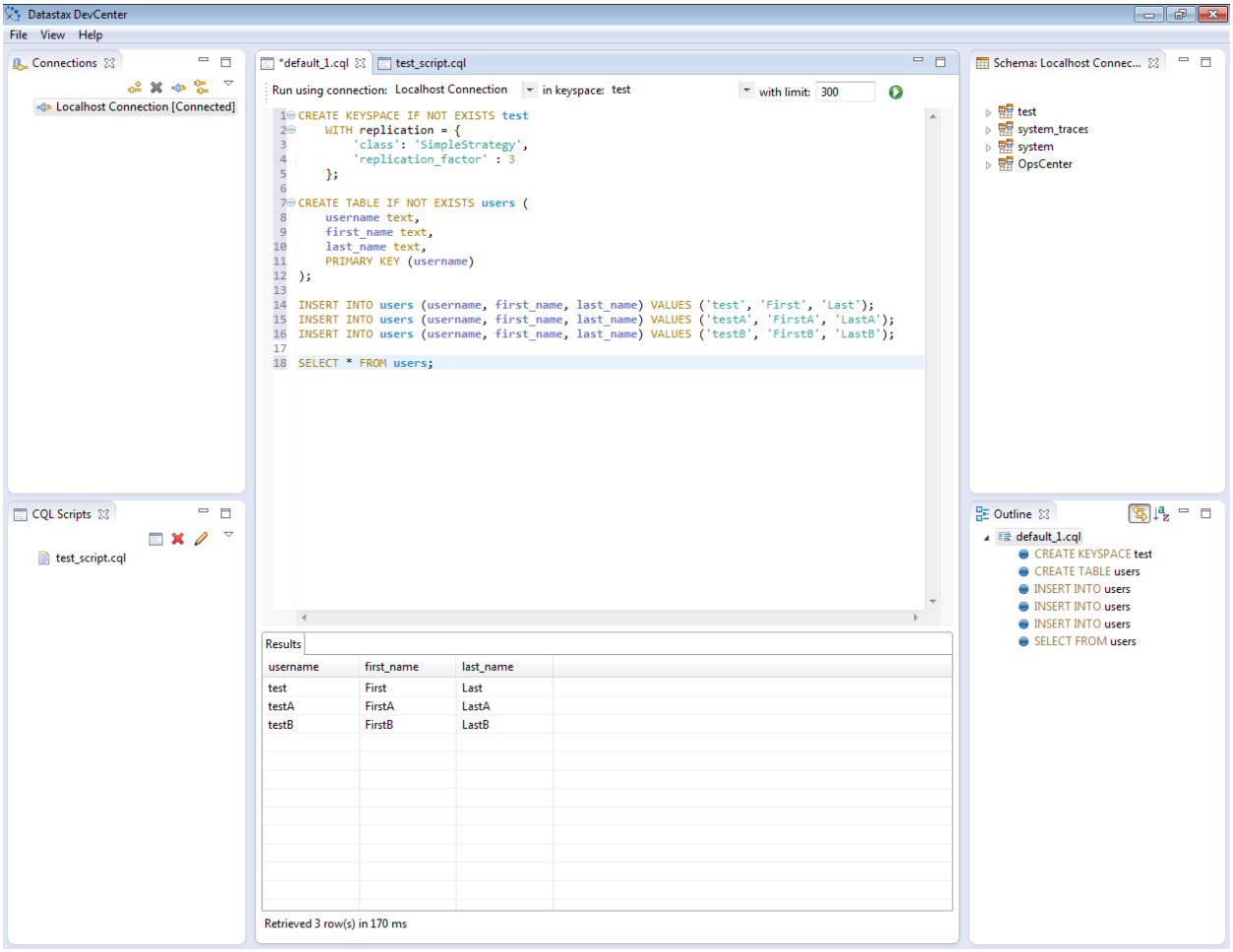

Figure 25: DataStax DevCenter

DataStax DevCenter is based on the Eclipse Rich Client Platform. The tool has several panes whose organization can be customized by the user. The layout shown in Figure 25 is the default one. Initially, the interface will not show much information.

To connect to a Cassandra instance or a cluster, we need to add a new connection by clicking the New Connection button in the **Connections** pane, known as the Connection Manager.

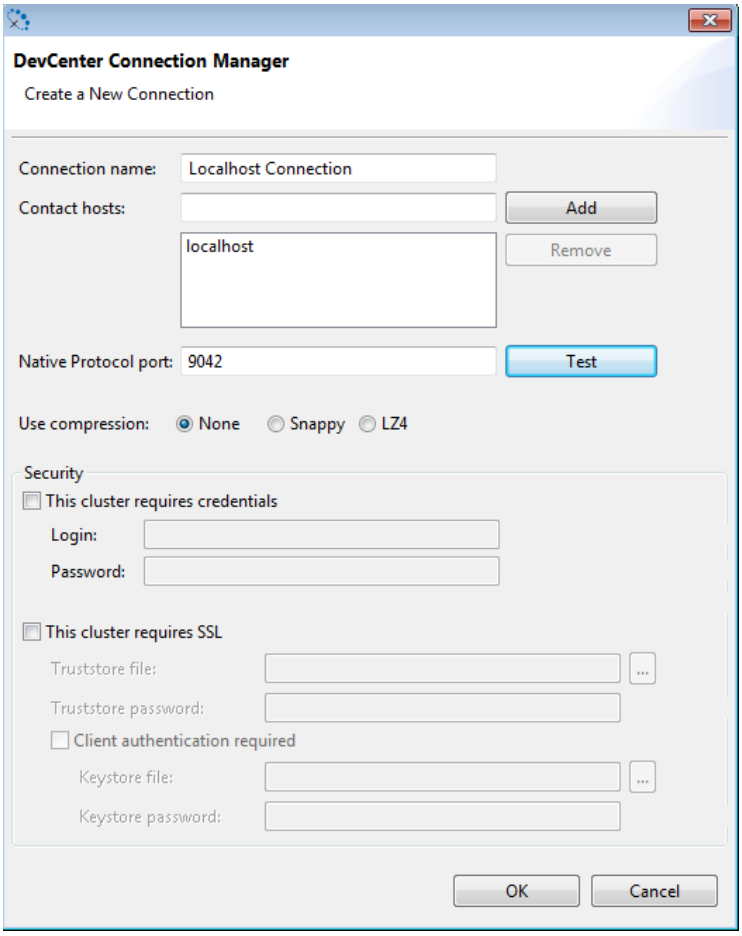

Figure 26: Adding a local connection

The initial port parameters loaded in the new connection window will be just fine for a standard Cassandra installation. To make sure the connection parameters are valid, click the **Test** button. If the connection is successful, a confirmation dialog will be displayed.

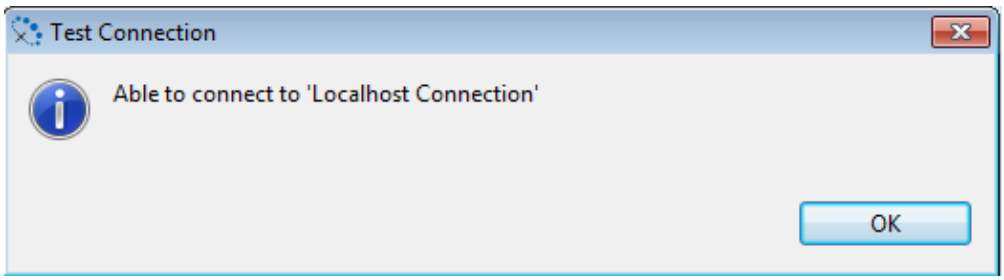

Figure 27: Successful Cassandra Connection Test

After adding the connection in DevCenter, not much will happen. In order to connect to Cassandra, we have to use the Connection Manager, which is used to create, edit, delete, open, and close cluster connections.

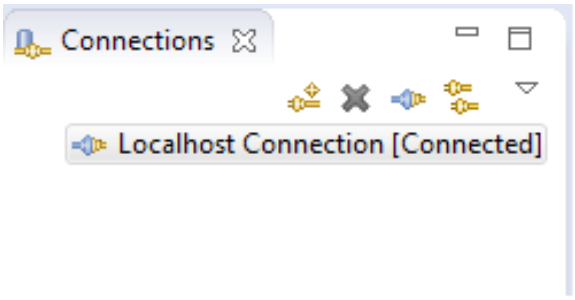

Figure 28: DevCenter Connections Manager

After clicking the Open Connection icon, DevCenter connects to the selected Cassandra cluster. Earlier we mentioned keyspaces, the top-level containers for data in Cassandra where replication is defined. DevCenter provides a nice overview of the keyspaces available to the user in the Schema Navigator, located in the top right corner.

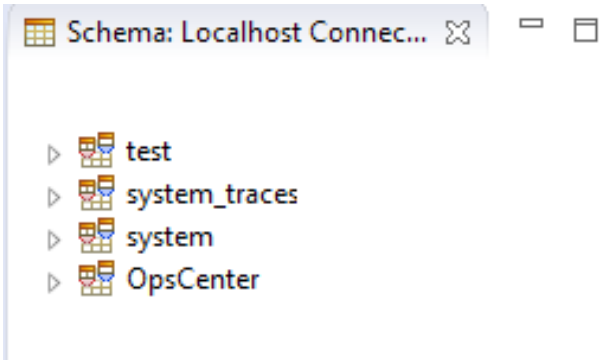

Figure 29: DevCenter Schema Navigator

The Schema Navigator can be used to browse Cassandra's objects structure down to the column level. It is useful for checking what a column is called or what the data type of a certain column is.

With time, users will accumulate a number of scripts to perform various operations on the Cassandra database. DevCenter also has a pane for managing CQL scripts.

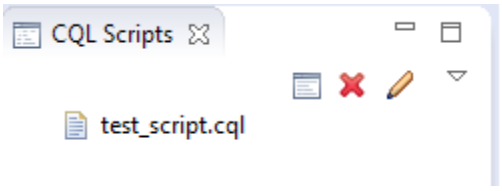

Figure 30: DevCenter CQL Scripts

The pane you will probably spend the most time in is the Query Editor, the upper middle pane in DevCenter. It has features such as syntax highlighting and autocomplete, which is invoked by pressing Ctrl+Spacebar. If multiple connections are open at the same time, you can select which connection to run the commands on in the Run using connection drop-down at the top of the Query Editor pane. No connection is selected by default, so you'll have to select a connection before running queries. After selecting a connection, you can also select the keyspace in which the commands will be executed from the **In keyspace** drop-down.

I have mentioned previously that Cassandra usually deals with a large amount of data. To prevent users from trying to fetch too much data from the system and possibly causing performance issues in the cluster, the Query Editor runs all read statements with a limit. The initial limit is set to 300. The final limit adjustment is left up to the user.

Running queries is done by selecting a script and then clicking the Execute CQL Script button to the right of the **With limit** text box, or pressing Alt+F11. Multiple open scripts will be shown as tabs in the editor, as shown in the following figure.

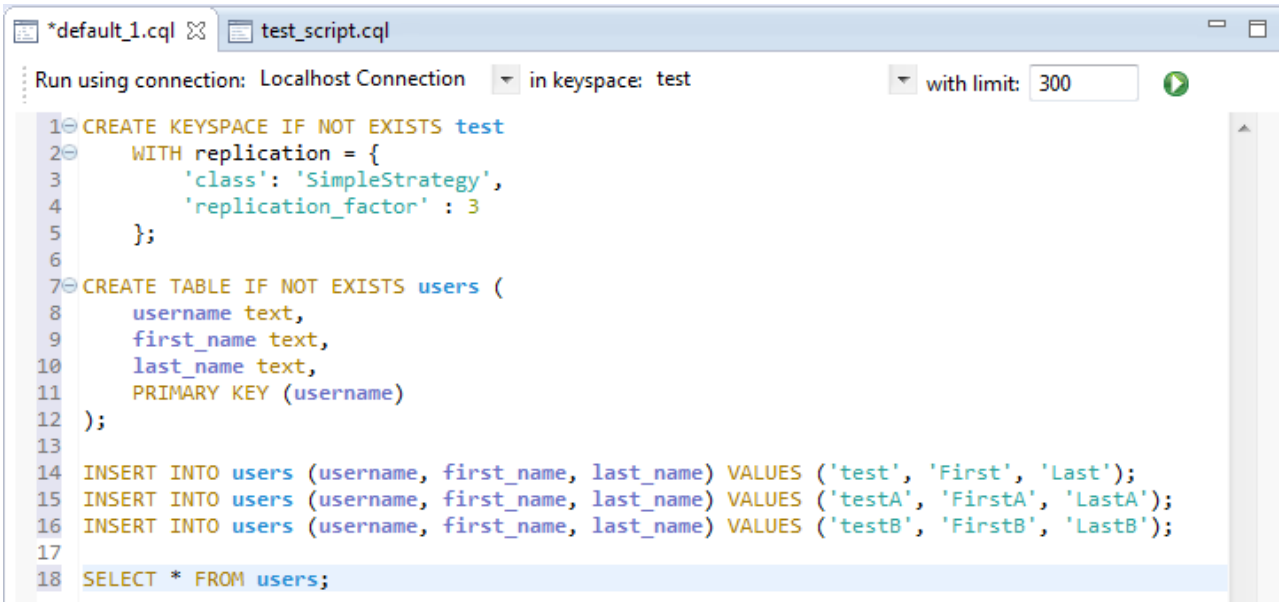

Figure 31: DevCenter Query Editor with multiple scripts

The Query Editor will also detect errors as you type and suggest possible solutions.

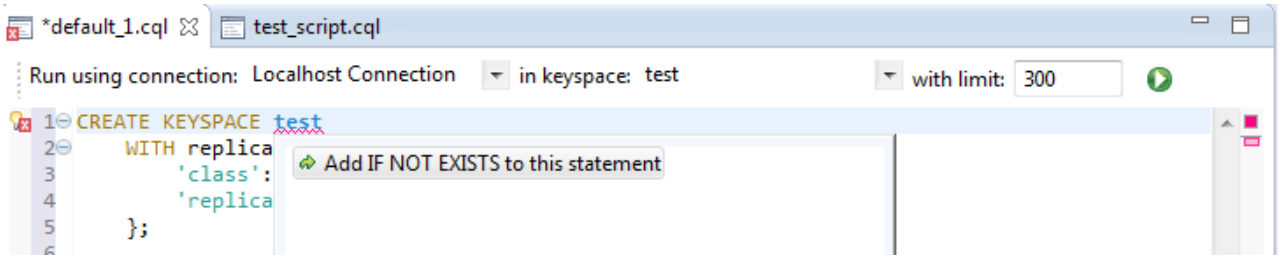

Figure 32: DevCenter Query Editor error detection and correction

Some commands will produce results in a tabular form. These results are displayed in the Results pane, where you see the results from the latest query. The Results pane is pretty simple; it shows the selected columns and their values. The binary values will be displayed with a mark, and going into single bytes is not yet possible with the DevCenter Query Editor.

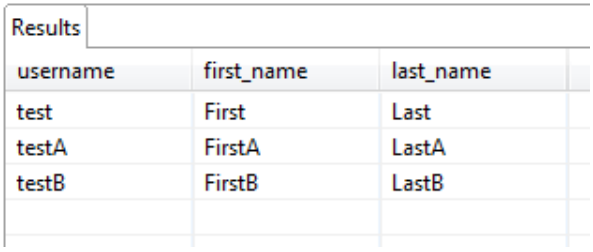

Figure 33: DevCenter Results Pane

When navigating large scripts, one can very easily lose the overview of the whole script. In the bottom right corner of DevCenter, the Outline pane is available to remedy such situations. Double-clicking a statement in it takes the user to the corresponding statements in the currently open script.

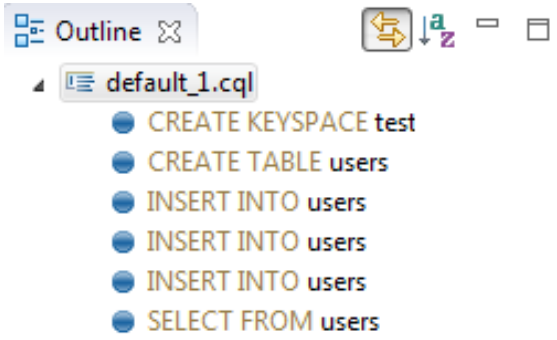

Figure 34: DevCenter Outline Pane

Summary

In this chapter, we discussed how to install Cassandra on two of the most popular operating systems today: Linux and Microsoft Windows. We saw that the basic installation of Cassandra is simple and usually takes only a couple of minutes to set everything up.

Cassandra is, at its core, a Java application, so we covered how to set up the basic Java environment required for Cassandra to run. The installation of Java needs a bit of tweaking, especially on the Windows platform, but all in all Java is a pretty stable, scalable, and battle-tested platform used in countless production systems by many companies.

Cassandra includes a couple of useful and powerful tools such as **`cqlsh`** and **`nodetool`**. We didn't cover these in this chapter because to use them, it is not necessary to set up the same environment we need to start interacting with Cassandra.

Also in this chapter we covered how to install and use a tool called DataStax DevCenter. DevCenter is suitable for users with no command line experience, and it has many ergonomic features for newcomers to Cassandra. It enables users to make changes to queries much easier, and it has decent context-aware autocomplete abilities, such as quick fixes for most common mistakes when using CQL.

Chapter 3 Data Modeling with Cassandra and CQL

If you are coming from the relational world, it will take some time for you to get used to the many ways problems are solved in Cassandra. The hardest thing to let go of is data normalization. In the relational world, data is split into multiple tables so that the data has very little redundancy. The data is logically grouped and stored, and if we need a certain view on this data, we make a query and present the desired data to the user.

With Cassandra the story is a bit different. We have to think about how our data is going to be queried from the start. This causes a lot of write commands to the database because data is often inserted into multiple tables every time we store something in Cassandra. One of the first thoughts that may come to mind here is that this is not going to be responsive or fast, but remember, Cassandra is really fast at writing.

Another concern you may have is that this will probably store a lot more data than we actually need, and you are right about that; Cassandra will probably store a lot more data on the disk than a relational database will. This heavy writing pays off when running queries because the queries are already prepared and there is no complex, time-consuming data joining. The entire Cassandra philosophy could be summarized with the premise that the disk is cheap.

Comparing Relational and Cassandra Data Storage

To compare classical relational storage and Cassandra, we need to start with the model. As an example, we'll use an online used car market. Let's start with the relational data model.

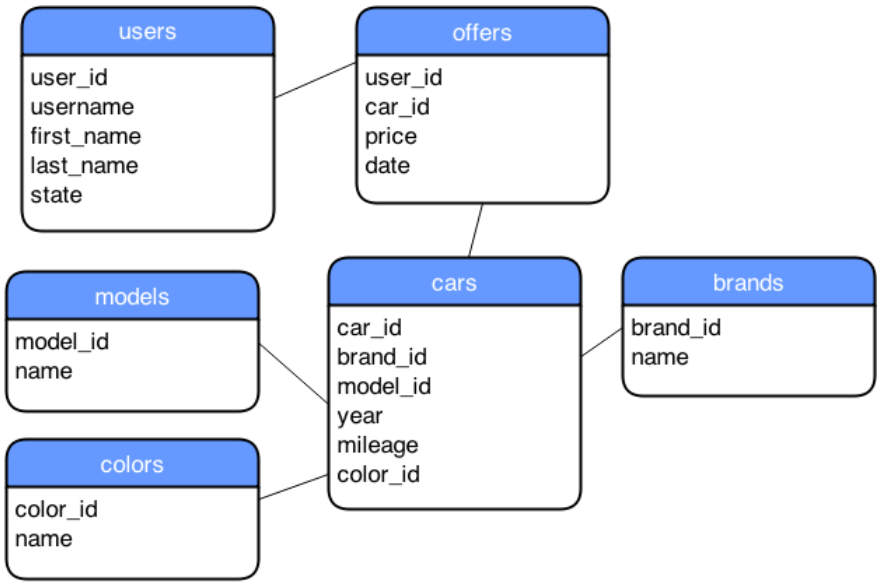

Figure 35: Simple relational model of an online car market

Figure 35 shows a simple relational model for an online used car market. Every box in the figure would become a separate table in a relational database. Elements inside the boxes would become columns in the tables. Every stored row would have each column defined in the model saved to the disk. If the creator of the table specifies it, the columns can have an undefined or a null value, but they would still be stored on the disk. When relational data from the model in Figure 35 is stored, it might look something like the following figure.

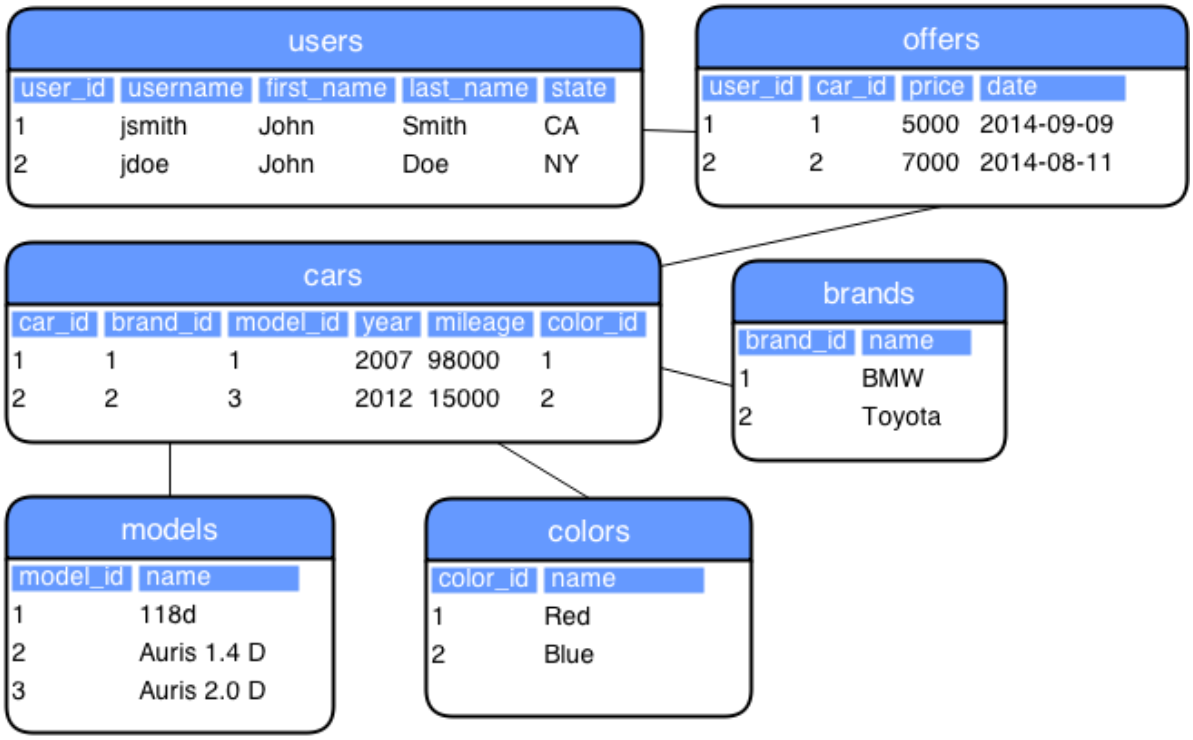

Figure 36: Example of stored relational data

To examine Cassandra's way of storing data, we first have to look at how the data is actually structured.

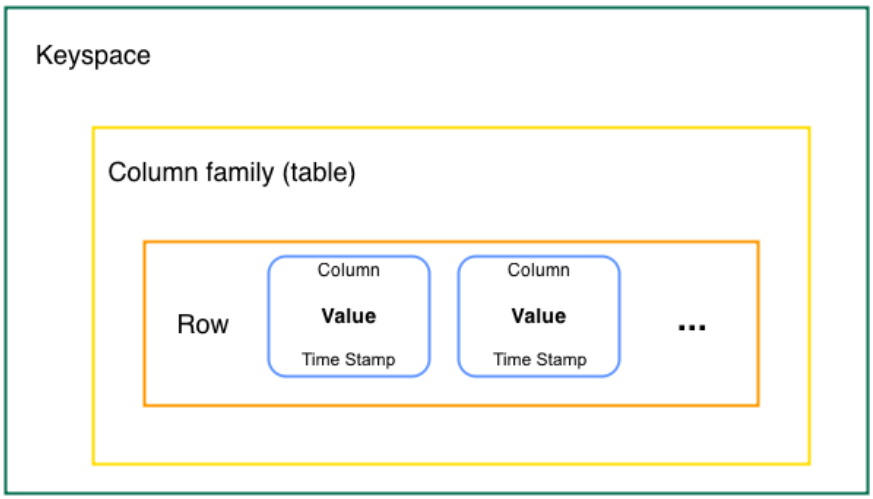

Figure 37: Cassandra Data Structuring

We already talked about the keyspace and the column family. Column family is also often simply called a table. A row in Cassandra can have all of the columns defined or just some of them. Cassandra will save only actual values to the row. Cassandra's limitation for a row is that it must fit on a single node. The second limitation for a row is that it can have at most two billion columns. In most cases, this is more than enough, and if you exceed that number, the data model you defined probably needs some tweaking.

The previous figure shows columns in a row storing a value and a time stamp. The columns are always sorted within a row by the column name. We'll discuss how to model the example of the used car market with Cassandra in the next sections, but for now, let's just show how the user's table would be stored in Cassandra. Note that we didn't use an auto-increment key as is often the case with relational data modeling; rather, we used a natural key username. For now, let's skip the CQL stuff and just look at the stored data at an abstract level:

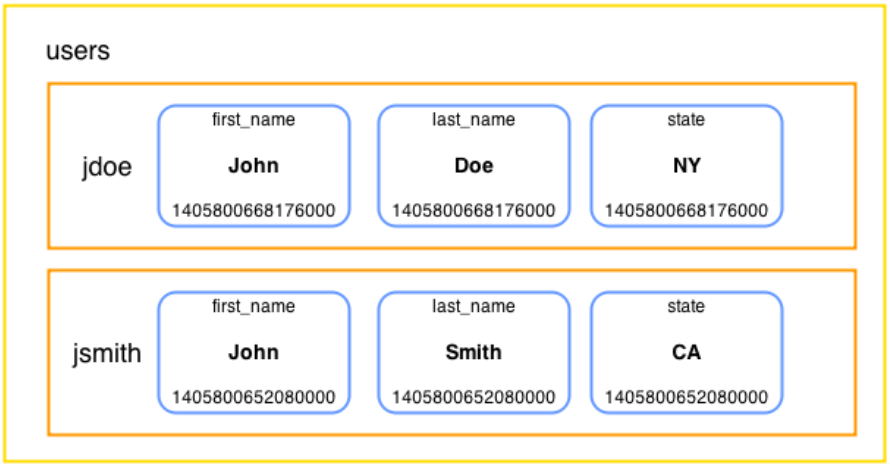

Figure 38: Cassandra storing example user data

The previous figure shows the low-level, data-storing technique Cassandra uses. The example is pretty similar to the way the relational database stores data. Now, let's look at the way the data is stored if the row keys were the states and the clustering columns were the usernames. Also, let's add John Q. Public to NY to illustrate the difference.

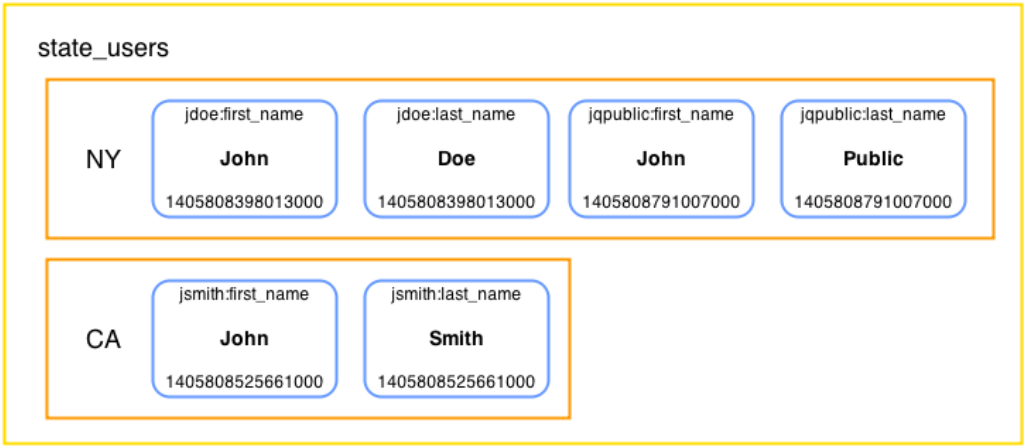

Figure 39: Cassandra storing example user data with state as partition key

The data storage in the previous figure might seem a little strange at the moment because column names are fairly fixed in relational database storage systems. It uses a username to group the columns belonging to a single user into a group. Remember, columns are always sorted by their names. Also note that there is no column for the username and that it's stored only in the column name. However, this approach to storage does not allow direct fetching of users by their username. To access any of the data contained within the row (partition), we need to provide a state name.

Now that we know how Cassandra is handling the data internally, let's try it in practice. By now, we have talked about a lot of Cassandra concepts and how they differ from relational databases. In the next section, we'll dive into the CQL, the language used for interacting with Cassandra.

CQL

CQL, or Cassandra Query Language, is not the only way of interacting with Cassandra. Not long ago, Thrift API was the main way of interacting with it. The usage, organization, and syntax of this API were oriented toward exposing the internal mechanisms of Cassandra storage directly to the user.

CQL is the officially recommended way to interact with Cassandra. The current version of CQL is CQL3. CQL3 was a major update from CQL2. It added **`CREATE TABLE`** syntax to allow multicolumn primary keys, comparison operators in **`WHERE`** clauses on columns besides the partition key, **`ORDER BY`** syntax, and more. The main difference from standard SQL is that CQL does not support joins or subqueries. One of the main goals of CQL is, in fact, giving users the familiar feeling of working with SQL.

CQL Shell

In the previous chapter we looked at how to install and run DataStax DevCenter. If you don't want to use the DataStax DevCenter, go to the directory where you installed Cassandra, and then go to the **bin** directory and run the **`cqlsh`** utility.

```
# ./cqlsh
Connected to Test Cluster at localhost:9160.
[cqlsh 4.1.1 | Cassandra 2.0.9 | CQL spec 3.1.1 | Thrift protocol 19.39.0]
Use HELP for help.
cqlsh>
```

Code Listing 3

After running the CQL shell, you can start issuing instructions to Cassandra. The CQL shell is fine for most of the day-to-day work with Cassandra. In fact, it is the admin's preferred tool because of its command-line nature and availability. Remember, it comes bundled with Cassandra. DevCenter is more for developing complex scripts and developer efficiency.

Keyspace

Before doing any work with the tables in Cassandra, we have to create a container for them, otherwise known as a keyspace. One of the main uses for keyspaces is defining a replication mechanism for a group of tables. We'll start by defining a keyspace for our used car market example.

```
CREATE KEYSPACE used_cars
       WITH replication = {
              'class': 'SimpleStrategy',
              'replication_factor' : 1};
```

Code Listing 4

When creating a keyspace, we have to specify the replication. Two components defining the replication are **`class`** and **`replication_factor`**. In our example, we have used **`SimpleStrategy`** for the class option; it replicates the data to the next node on the ring without any network-aware mechanisms.

At the moment, we are in the development environment with only one node in the cluster so a replication factor of one is sufficient. In fact, if we specified a higher replication factor—3, for instance—and then used the **`quorum`** option when performing selects, the select commands would fail because **`quorum`** cannot be reached with a replication factor 3 because 2 nodes have to respond, and we have only one node in our cluster at the moment. If we added more nodes to the cluster and would like the data to be replicated, we can update the replication factor with the **`ALTER KEYSPACE`** command as shown in the following code example.

```
ALTER KEYSPACE used_cars
       WITH REPLICATION = {
              'class' : 'SimpleStrategy',
              'replication_factor' : 3};
```

Code Listing 5

`SimpleStrategy` will not be enough in some cases. The first chapter mentioned that Cassandra uses a component called snitch to determine the network topology and then uses different replication mechanisms that are network-aware. To enable this behavior on the keyspace, we have to use the **`NetworkTopologyStrategy`** replication class, as shown in the following example.

```
CREATE KEYSPACE used_cars
       WITH replication = {
              'class': 'NetworkTopologyStrategy',
              'DC1' : 1,
              'DC2' : 3};
```

Code Listing 6

The previous example replicates the **used_cars** keyspace with replication factor 1 in DC1 and replication factor 3 in DC2. Data center names DC1 and DC2 are defined in node configuration files. We don't have any data centers specified in our current environment but the command will still run and create a keyspace with the options that we specified.

On some occasions, we will need to delete the complete keyspace. This is achieved with the **DROP KEYSPACE** command. The command is pretty simple and has only one parameter: the name of the keyspace that we want to delete. Keyspace removal is irreversible and issuing the command causes immediate removal of the keyspace and all of its column families and their data.

```
DROP KEYSPACE used_cars;
```

Code Listing 7

The **DROP KEYSPACE** command is primarily used in development scenarios, migration scenarios, or both. If you drop a keyspace, the data is removed from the system and can only be restored from a backup.

When issuing commands in the CQL shell, we usually have to provide a keyspace and table. The following **SELECT** command lists all users in the **used_cars** keyspace.

```
SELECT * FROM used_cars.users;

 username | first_name | last_name | state
----------+------------+-----------+-------
     jdoe |       John |       Doe |    NY
   jsmith |       John |     Smith |    CA
```

Code Listing 8

This becomes a bit tedious over time, and most of the operations issued in sequence are usually within a single keyspace. So, to reduce the writing overhead, we can issue the **USE** command.

```
USE used_cars;
```

Code Listing 9

After running the **USE** command, all of the subsequent queries are run within the specified keyspace. This makes writing queries a bit easier.

Table

Defining and creating tables is the foundation for building any application. Although the syntax of CQL is similar to SQL, there are important differences when creating tables. Let's start with the most basic table for storing users.

```
CREATE TABLE users (
       username text,
       password text,
       first_name text,
       last_name text,
       state text,
       PRIMARY KEY (username)
);
```

Code Listing 10

PRIMARY KEY is required when defining a table. It can have one or more component columns, but it cannot be a counter column. If the **PRIMARY KEY** consists of only one column, you can place **PRIMARY KEY** into the column definition after specifying the column type.

```
CREATE TABLE users (
       username text PRIMARY KEY,
       password text,
       first_name text,
       last_name text,
       state text
);
```

Code Listing 11

If we were using SQL and a relational database to specify columns in the **PRIMARY KEY**, all of the columns would be treated the same. In the previous chapter we discussed how Cassandra stores data, and that all data is saved in wide rows. Each row is identified by an ID. In Cassandra terminology, this ID is called the partition key. The partition key is the first column from the list of columns in the **PRIMARY KEY** definition.

Note: The first column in the PRIMARY KEY list is the partition (row) key.

Let's use an example to see what is going on with the columns specified after the first place in the **PRIMARY KEY** list. We will model the offers shown when we compared relational and Cassandra data storage. We mentioned earlier that normalization is not common when working with Cassandra. The relational example we used stored the data about car offerings into five tables. Here, we are going to use just one table, at least for now.

```
CREATE TABLE offers (
       username text,
       date timestamp,
       price float,
       brand text,
       model text,
       year int,
       mileage int,
       color text,
```

```
      PRIMARY KEY (username, date)
);
```

Code Listing 12

The **PRIMARY KEY** definition consists of two columns. The first column in the list is the partition key. All of the subsequent listed columns after the partition key are clustering keys. The clustering keys influence how Cassandra is organizing the data on the storage level. To keep things simple, we'll just concentrate on brand and color to show what the clustering key does. The table with offers might look something like the following when queried.

```
 username | date                       | brand  | color
----------+----------------------------+--------+-------
     jdoe | 2014-08-11 17:12:32+0200 | Toyota |  Blue
   jsmith | 2014-09-09 11:35:20+0200 |    BMW |   Red
   jsmith | 2014-09-19 11:35:20+0200 |    BMW | Black
```

Code Listing 13

This looks pretty similar to what we would expect from a table in relational storage.

When it comes down to physically storing the data, the situation is a bit different. The clustering key **date** is combined with the names of the other columns. This causes the offers to be sorted by date within the row.

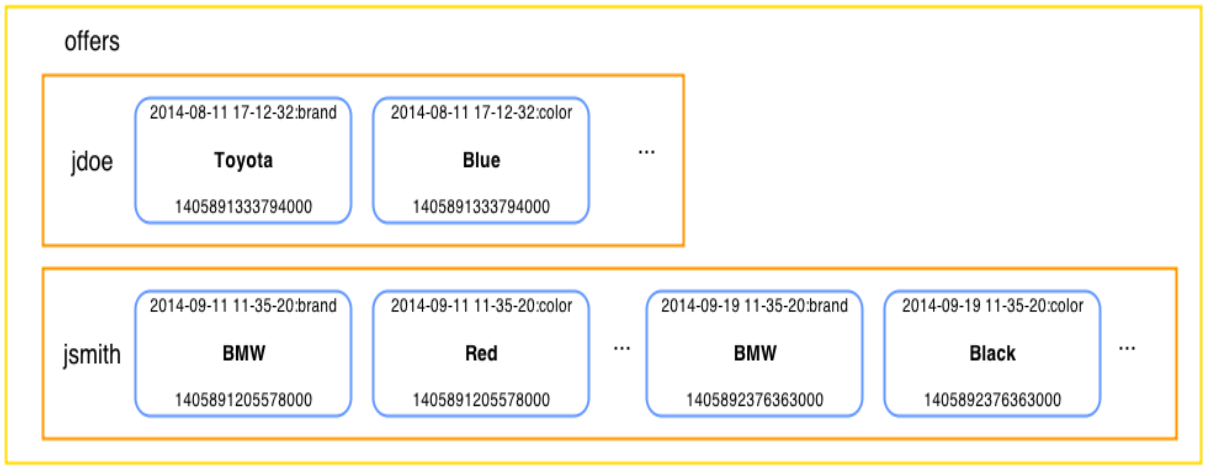

Figure 40: Cassandra storing offers data with date as clustering key

Now the question is, what would happen if we didn't add **date** to the second place in the **PRIMARY KEY** list, causing it to become a clustering key? If you look more carefully, the column names in the previous figure are combinations of date and column names such as brand and color. If we removed **date** as a clustering key, every user would have just one used car offer and that's it. **date** would become just another column and we would lose the possibility to add more offers per user.

Sometimes data will be too large for a single row. In that case, we combine the row ID with other data to split the data further into smaller chunks. For instance, if our site became mega popular and if all car retailers offered their cars on our site, we would somehow have to split the data. Perhaps the most reasonable way to do it would be to split the data by the car brand.

```
CREATE TABLE offers_by_brand (
       username text,
       date timestamp,
       price float,
       brand text,
       model text,
       year int,
       mileage int,
       color text,
       PRIMARY KEY ((username, brand), date)
);
```

Code Listing 14

Note that if a row is a combination of username and brand, we will be unable to access the offers without specifying the username and brand when fetching the row. Sometimes when such a technique is used, the application has to combine the data to present it to the user. A partition (row) key that consists of more than one column is called a composite partition key. A composite partition key is defined by listing the columns in parentheses. Remember, only the first column in the list is the partition key. If we want more columns to go into the partition key, we have to put them in parentheses.

CQL Data Types

At this point we have used different types to specify column data but have yet to discuss them. The following table provides a short overview of the data types in Cassandra.

Table 1: CQL Data Types

Type	Description
ascii	US-ASCII character string
bigint	64-bit signed long
blob	Arbitrary bytes (no validation), expressed as hexadecimal in CQL shell. DevCenter only shows <<blob>>.

Type	Description
boolean	true or false
counter	64-bit distributed counter value
decimal	Variable-precision decimal
double	64-bit IEEE-754 floating point
float	32-bit IEEE-754 floating point
inet	IP address string (supports both IPV4 and IPV6 formats)
int	32-bit signed integer
list	A collection of ordered elements
map	Associative array
set	Unordered collection of elements
text	UTF-8 encoded string
timestamp	Date and time, encoded as 8-byte integers starting from 1.1.1970
uuid	UUID in standard format
timeuuid	UUID with a time stamp in its value
varchar	UTF-8 encoded string

Type	Description
varint	Arbitary-precision integer

The column type is defined by specifying it after the column name. Changing column type is possible if the types are compatible. If the types are not compatible, the query will return an error. Also note that the application might stop functioning because the types may not be compatible on the application level. Changing the clustering column or columns that have indexes defined on them is not possible. For instance, changing the **`year`** column type from **`int`** to **`text`** is not allowed and causes the following error.

```
ALTER TABLE offers_by_hand ALTER year TYPE text;
Bad Request: Cannot change year from type int to type text: types are
incompatible.
```

Code Listing 15

Adding columns to Cassandra tables is a common and standard operation.

```
ALTER TABLE offers ADD airbags int;
```

Code Listing 16

The previous code listing adds an integer type column called **`airbags`** to the table. In our example this column stores the number of airbags the vehicle has. Dropping the column is done with the **`DROP`** subcommand from the **`ALTER TABLE`** command.

```
ALTER TABLE offers DROP airbags;
```

Code Listing 17

To complete the life cycle of a table, we have to run two more commands. The first is for emptying all of the data from the table.

```
TRUNCATE offers;
```

Code Listing 18

The final command is for dropping the entire table.

```
DROP TABLE offers;
```

Code Listing 19

Table Properties

Besides column names and types, CQL can also be used to set up the properties of a table. Some properties such as comments are used for easier maintenance and development, and some go really deep into the inner workings of Cassandra.

Table 2: CQL Table Properties

Property	Description
bloom_filter_fp_chance	False-positive probability for SSTable Bloom filter. This value ranges from 0, which produces the biggest possible Bloom filter, to 1.0, which disables the Bloom filter. The recommended value is 0.1. Default values depend on the compaction strategy. SizeTieredCompaction has a default vaule of 0.01 and LeveledCompaction has a default value of 0.1.
caching	Cache memory optimization. The available levels are all, keys_only, rows_only, and none. The rows_only option should be used with caution because Cassandra puts a lot of data into memory when that option is enabled.
comment	Used primarily by admins and developers to make remarks and notes about the tables.
compaction	Sets the compaction strategy for the table. There are two: the default SizeTieredCompactionStrategy and LeveledCompactionStrategy. SizeTiered triggers compaction when the SSTable passes a certain limit. The positive aspect of this strategy is that it doesn't degrade write performance. The negative is that it occasionally uses double the data size on the disk and has potentially poor read performance. LeveledCompaction has multiple levels of SSTables. At the lowest level there are 5 MB tables. Over time, the tables are merged into a table that is 10 times bigger; this causes very good read performance.

Property	Description
compression	Determines how the data is going to be compressed. Users can choose speed or space savings. The greater the speed, the less disk space is saved. In order from fastest to slowest, the compressions are LZ4Compressor, SnappyCompressor, DeflateCompressor.
dclocal_read_repair_chance	Probability of invoking read repairs.
gc_grace_seconds	Time to wait for removal of data with tombstones. The default is 10 days.
populate_io_cache_on_flush	This value is disabled by default; enable this only if you expect all the data to fit into memory.
read_repair_chance	Number between 0 and 1.0 specifying the probability to repair data when quorum is not reached. Default value is 0.1
replicate_on_write	This applies only to counter tables. When set, replicas write to all affected replicas, ignoring the specified consistency level.

Defining tables with comments is pretty easy and represents a positive practice for maintenance and administration of the database.

```
CREATE TABLE test_comments (
       a text,
       b text,
       c text,
       PRIMARY KEY (a)
) WITH comment = 'This is a very useful comment';
```

Code Listing 20

Most of the options are simple to define. Adding a comment is a nice example of such an option. On the other hand, compression and compaction options have subproperties. Defining the subproperties is done with the help of a JSON-like syntax. We'll look at an example shortly.

Table 3: CQL Table Compression Options

Property	Description
sstable_compression	Specifies the compression algorithm to use. The available algorithms were listed in the previous table: LY4Compressor, SnappyCompressor, and DeflateCompressor. To disable compression, simply use an empty string.
chunk_length_kb	SSTables are compressed by block. Larger values generally provide better compression rates but increase the data size for reading. By default, this option is set to 64KB.
crc_check_chance	All compressed data in Cassandra has a checksum block. This value is used to check that the data is not corrupt so that it isn't sent to other replicas. By default, this option is set to 1.0 so that every time the data is read, the node checks the checksum value. Setting this value to 0 disables the checksum checking and setting it to 0.33 causes checksum to be checked every third time the data is read.

Manipulating compression options can lead to significant performance gains, and many Cassandra solutions out there run with compression options set to non-default values. Indeed, tuned compression options are sometimes very important for a successful Cassandra deployment, but most of the time, especially if you are just starting to use Apache Cassandra, you will be just fine using the default **`SnappyCompressor`** settings as the compression option.

Compaction has many subproperties as well.

Table 4: CQL Table Compaction Options

Property	Description
enabled	Determines whether compaction will run on the table. By default, all tables have compaction enabled.
tombstone_threshold	Ratio value from 0 to 1 specifying how many columns have to be marked with tombstones to begin compaction. The default value is 0.2.

Property	Description
tombstone_compaction_interval	Minimum time to wait after SSTable creation to start compaction, but only after the tombstone_threshold has been reached. The default setting is one day.
unchecked_tombstone_compaction	Enables aggressive compaction, which runs compaction on the checked interval even if the SSTable hasn't reached the threshold. By default, this is set to false.
min_sstable_size	Used with SizeTieredCompactionStrategy. This option is used to prevent grouping SSTables into too-small chunks. It is set to 50MB by default.
min_threshold	Available in SizeTieredCompactionStrategy. Represents a minimum number of SSTables needed to start a minor compaction process. It is set to 4 by default.
max_threshold	Only available in SizeTieredCompactionStrategy. Sets the maximum number of tables processed by minor compaction. It is set to 32 by default.
bucket_low	For SizeTieredCompactionStrategy only. Checks for tables with a difference in size below the group average. The default value is 0.5, meaning only tables whose sizes diverge by a maximum of 50 percent.
bucket_high	For SizeTieredCompactionStrategy only. Checks for compaction on tables whose sizes are larger than the group average. The default setting is 1.5, meaning all tables 50 percent larger than the group average.

Property	Description
sstable_size_in_mb	Available only in the LeveledCompactionStrategy. Represents the targeted SSTable size, but the size may be slightly larger or smaller because row data is never split between two SSTables. It is set to 5MB by default.

The options for compression and compaction options are defined with JSON-like syntax.

```
CREATE TABLE inventory (
  id uuid,
  name text,
  color text,
  count varint,
  PRIMARY KEY (id)
) WITH
      compression = {
          'sstable_compression' : 'DeflateCompressor',
          'chunk_length_kb' : 64
        }
    AND
      compaction = {
          'class' : 'SizeTieredCompactionStrategy',
          'min_threshold' : 6
        };
```

Code Listing 21

Cluster Ordering the Data

Previously we noted that columns within a row on the disk are sorted. When fetching the data from physical rows in the table, we get the data presorted by the specified clustering keys. If the data is, by default, in ascending order, then fetching in descending order causes performance issues over time. To prevent this from happening, we can instruct Cassandra to keep the data in a row of a table in descending order with **`CLUSTERING ORDER BY`**.

```
CREATE TABLE latest_offers (
      username text,
      date timestamp,
      price float,
      brand text,
      model text,
      year int,
      mileage int,
      color text,
      PRIMARY KEY (username, date)
```

```
) WITH CLUSTERING ORDER BY (date DESC);
```

Code Listing 22

It is important to remember that the previous **`CREATE TABLE`** example keeps the data sorted in descending order but only within the row. This means that selecting the data from a specific user row with a username will show sorted data for that user, but if you select all the data from the table without providing the row key (**`username`**), the data will not be sorted by **`username`**.

Tip: Clustering sorts the data within the partition, not the partitions.

But just to remember all this even better, let's consider an example. Let's go back to our **`offers`** table. The **`offers`** table is sorted in ascending order by default, but this is fine for demonstrating what the previous tip is all about. First, we'll select the offers for the **`jsmith`** user.

```
SELECT username, date, brand, color
    FROM offers WHERE username = 'jsmith';

 username | date                         | brand | color
----------+--------------------------+-------+-------
   jsmith | 2014-09-09 11:35:20+0200 |   BMW |   Red
   jsmith | 2014-09-19 11:35:20+0200 |   BMW | Black
   jsmith | 2014-09-20 17:12:32+0200 |  Audi | White
```

Code Listing 23

Notice that these offers are sorted in ascending order, with the oldest record being in the first place. And that's what we would expect if the data were sorted by **`date`** since this column is the clustering key. But some Cassandra users may be confused when they try to access the whole table without providing the row key (**`username`**):

```
SELECT username, date, brand, color
    FROM offers;

 username | date                         | brand  | color
----------+--------------------------+--------+--------
     jdoe | 2014-08-11 17:12:32+0200 | Toyota |   Blue
     jdoe | 2014-08-25 11:13:22+0200 |   Audi | Orange
   jsmith | 2014-09-09 11:35:20+0200 |    BMW |    Red
   jsmith | 2014-09-19 11:35:20+0200 |    BMW |  Black
   jsmith | 2014-09-20 17:12:32+0200 |   Audi |  White
     adoe | 2014-08-26 10:11:10+0200 |     VW |  Black
```

Code Listing 24

Notice that the data has no sorting whatsoever by the **`username`** (row ID, partition key) column. Once again, the data is sorted within a username but not between usernames.

Manipulating the Data

Up until now, we have been concentrating mostly on the data structure and the various options and techniques for organizing stored data. We covered the basic data reading but we'll look at more advanced examples of data retrieval soon. Inserting data looks pretty much like it does in standard SQL databases; we define the table name and the columns and then specify the values. One of the simplest examples is entering data into the **users** table.

```
INSERT INTO
    users (username, password, first_name, last_name, state)
       VALUES ('jqpublic', 'hello1',  'John', 'Public', 'NY');
```

Code Listing 25

The previous code shows you how to insert data into the **users** table. It has only text columns so all of the inserted values are strings. A more complex example would be to insert values into the **offers** table, as in the following example.

```
INSERT INTO
       offers (username, date, price, brand, model, year,
            mileage, color)
       VALUES ('jsmith', '2014-09-19 11:35:20', 6000, 'BMW', '120i',
              2010, 40000, 'Black');
```

Code Listing 26

Working with strings and numbers is pretty straightforward. Numbers are more or less simple; the decimal separator is always a period because insert parameters are split with a comma. A number cannot begin with a plus sign, only a minus sign. The numbers can also be entered with scientific notation, so both "e" and "E" are valid when specifying a floating-point number. Note that DataStax DevCenter marks numbers defined with an "e" as incorrect syntax usage, while the CQL shell interprets both signs as valid. After the "e" that specifies the exponent, a plus or minus sign can follow. The following code example inserts a car that costs $8,000 specified in scientific notation with a negative exponent prefix.

```
INSERT INTO
       offers (username, date, price, brand, model, year,
            mileage, color)
         VALUES ('jsmith', '2014-05-11 01:22:11', 80000.0E-1, 'FORD',
               'Orion', 206, 200000, 'White');
```

Code Listing 27

Strings are always placed inside single quotation marks. If you need the single quotation mark in the data, escape it with another single quotation mark before placing it in the string as in the following example.

```
INSERT INTO test_data(stringval) VALUES ('O''Hara');
```

Code Listing 28

Besides numbers and strings, the previous queries also contain dates. Dates are defined as the **timestamp** type in Cassandra. A **timestamp** can simply be entered as an integer, representing the number of milliseconds that have passed since January 1, 1970 at 00:00:00 GMT. The string literals for entering **timestamp** data in Cassandra are entered in the following formats.

Table 5: Timestamp String Literal Formats

Format	Example
yyyy-mm-dd HH:mm	'2014-07-24 23:23'
yyyy-mm-dd HH:mm:ss	'2014-07-24 23:23:40'
yyyy-mm-dd HH:mmZ	'2014-07-24 23:23+0200'
yyyy-mm-dd HH:mm:ssZ	'2014-07-24 23:23:40+0200'
yyyy-mm-dd'T'HH:mm	'2014-07-24T23:23'
yyyy-mm-dd'T'HH:mmZ	'2014-07-24T23:23+0200'
yyyy-mm-dd'T'HH:mm:ss	'2014-07-24T23:23:40'
yyyy-mm-dd'T'HH:mm:ssZ	'2014-07-24T23:23:40+0200'
yyyy-mm-dd	'2014-07-24'
yyyy-mm-ddZ	'2014-07-24+0200'

The format is pretty standard in many programming languages and database environments. Besides using a space to separate the date and time parts, sometimes they are separated with a T. The time zone is specified in a four-digit format and is noted with the letter Z in the format examples in the previous table. The time zone can begin with a plus or a minus sign, depending on the position of the time zone compared to GMT (Greenwich Mean Time). The first two digits represent the difference in hours and the last two digits represent the difference in minutes.

Most time zones are an integer difference when compared to GMT. Some regions have an additional half hour offset, such as Sri Lanka, Afghanistan, Iran, Myanmar, Newfoundland, Venezuela, Nepal, the Chatham Islands, and some regions of Australia. If no time zone is specified, the time zone from the coordinator node is used. Most of the documentation recommends specifying the time zone instead of relying on the coordinator node time zone. If the time zone is not specified, it is assumed that you wanted to enter '00:00:00' as the time zone.

The default format for displaying **timestamp** values in the CQL shell is "yyyy-mm-dd HH:mm:ssZ". This format shows all of the available information about the **timestamp** value to the user.

Sometimes you will want to update the data that has been written. As in standard SQL database systems, it is possible to update records, but the inner workings of the **UPDATE** command in Cassandra are a bit different. Let's change the brand and the model of an offer that user **jdoe** made on **2014-08-11**.

```
UPDATE offers SET
    brand = 'Ford', model = 'Mustang'
  WHERE
    username = 'jdoe' AND date='2014-08-11 17:12:32+0200';
```

Code Listing 29

Updating the value actually adds a new column within the row with the new time stamp and marks the old columns with tombstones. The tombstone is not set immediately, the data is just written. With the first read or compaction process starts, Cassandra will compare the two columns. If the column names are the same, the one with a newer time stamp wins. The other columns are marked with tombstones. This is easier to visualize in the following figure.

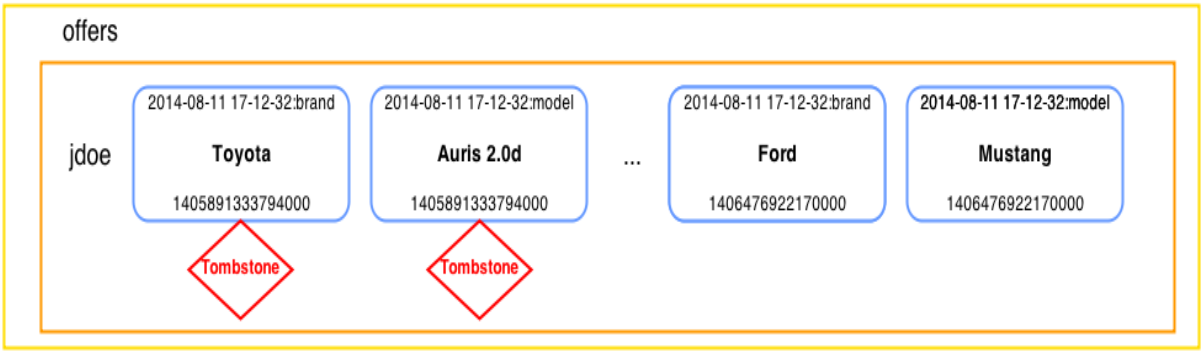

Figure 41: Updating columns: The old columns get tombstones, and new columns are added with new values

When manipulating data, there is another important difference when comparing Cassandra to SQL solutions. Let's review the current table data.

```
SELECT username, date, brand, model FROM offers;

 username | date                         | brand | model
```

```
----------+--------------------------+-------+---------
     jdoe | 2014-08-11 17:12:32+0200 |  Ford | Mustang
     jdoe | 2014-08-25 11:13:22+0200 |  Audi |      A3
   jsmith | 2014-05-11 01:22:11+0200 |  Ford |   Orion
   jsmith | 2014-09-09 11:35:20+0200 |   BMW |    118d
   jsmith | 2014-09-19 11:35:20+0200 |   BMW |    120i
   jsmith | 2014-09-20 17:12:32+0200 |  Audi |      A6
     adoe | 2014-08-26 10:11:10+0200 |    VW |    Golf
```

Code Listing 30

Now, let's try to insert the first offer from the **jdoe** user but with a different brand and model.

```
INSERT INTO offers (
      username, date, price, brand, model,
      year, mileage, color)
      VALUES (
            'jdoe', '2014-08-11 17:12:32',
            7000, 'Toyota', 'Auris 2.0d', 2012, 15000, 'Blue');
```

Code Listing 31

What do you think would be the result of the previous **INSERT** statement? Well, if this were a classical SQL database, we would probably get an error because the system already has data with the same username and date. Cassandra is a bit different. As mentioned before, **INSERT** adds new columns and that's it. The new columns will have the same names as the old ones, but they will have new time stamps. So when reading the data, Cassandra will return the newest columns. After running the previous **INSERT** statement, running the same query as in Code Listing 30 would return the following.

```
SELECT username, date, brand, model FROM offers;

 username | date                     | brand  | model
----------+--------------------------+--------+-----------
     jdoe | 2014-08-11 17:12:32+0200 | Toyota | Auris 2.0d
     jdoe | 2014-08-25 11:13:22+0200 |   Audi |         A3
   jsmith | 2014-05-11 01:22:11+0200 |   Ford |      Orion
   jsmith | 2014-09-09 11:35:20+0200 |    BMW |       118d
   jsmith | 2014-09-19 11:35:20+0200 |    BMW |       120i
   jsmith | 2014-09-20 17:12:32+0200 |   Audi |         A6
     adoe | 2014-08-26 10:11:10+0200 |     VW |       Golf
```

Code Listing 32

On some occasions you will want to delete data. This is done with the **DELETE** statement. Usual SQL systems allow the **DELETE** statement to be called without specifying which rows to delete. This then deletes all the rows in the table. With Cassandra, you must provide the **WHERE** part of the **DELETE** statement for it to work, but there are some options. We can always specify just the partition key in the **WHERE** part. This removes the entire row and, in effect, all of the offers made by the user.

```
DELETE FROM offers WHERE username = 'maybe';
```

Code Listing 33

The previous statement is a rather dangerous one because it removes the entire row, and it is done with a username that is not present in our example. Usually, when deleting the data, all of the primary key columns are specified, not just the partition (first) key.

```
DELETE FROM offers
   WHERE username = 'jdoe' AND date = '2014-08-11 17:12:32+0200';
```

Code Listing 34

Data Manipulation Corner Cases

In the previous section, we used an **INSERT** statement using the same partition and clustering key with new data set in the columns instead of updating the row. This technique is called upserting.

Upserting is a very popular technique in Cassandra and is actually the recommended way to update data. It also makes system maintenance and implementation easier because we don't have to design and implement additional update queries.

Tip: Upsert data by inserting new data with the old key whenever possible.

Upserting data is what you should be doing most of the time in Cassandra, but let's compare it with inserting. Let's assume that, for some reason, users can change all of the offers in our example at any time. We will analyze the fields and see what changes are even possible.

It would not be wise to let the system change the **username** on the offer. This would actually represent a serious security issue if left to any user. For our comparison, we'll assume that the administrator of the system will be able to change the **username**. Let's try changing it.

```
UPDATE offers
    SET username = 'maybe'
        WHERE username = 'jdoe'
            AND date = '2014-08-11 17:12:32+0200';

Bad Request: PRIMARY KEY part username found in SET part
```

Code Listing 35

OK, this didn't work but while we are at it, let's try to change the offer **date** and set it to the day after the offer date with the **UPDATE** command. The **date** field is not a partition key. In actuality, it is a clustering key, so it might just work.

```
UPDATE offers
    SET date = '2014-08-12 17:12:32+0200'
        WHERE username = 'jdoe'
            AND date = '2014-08-11 17:12:32+0200';

Bad Request: PRIMARY KEY part username found in SET part
```

Code Listing 36

OK, this didn't work either. **date** is also a part of the primary key. You may have expected it because of the error message received in the first try. Updating primary key columns is not possible.

This example is very important to remember. Cassandra does not allow updates to any of the primary key fields.

Note: Cassandra does not allow updates to primary key fields.

If you have some data that might require this kind of update, and if it is data that changes a lot, you might want to consider other storage technologies or move this logic to the application level. This could be done in two steps. The steps are interchangeable; the choice is left to the system designer. Let's consider an example where the first step is to insert the new data and the second is to delete the old data.

```
INSERT INTO offers (
       username, date, price, brand, model,
       year, mileage, color)
       VALUES ('maybe', '2014-08-11 17:12:32',
           7000, 'Toyota', 'Auris 2.0d', 2012, 15000, 'Blue');

DELETE FROM offers
    WHERE username = 'jdoe' AND date = '2014-08-11 17:12:32+0200';
```

Code Listing 37

The database will be in an inconsistent state before the **DELETE** statement is issued, but the situation would be the same if we reversed the steps. If the **INSERT** happens before the **DELETE**, two offers might be visible in the system for a while. If the **DELETE** happens before the **INSERT**, an offer might be missing. As mentioned before, it's up to the system designers to decide which is lesser of the two evils.

The previous case will work just fine in most situations. Still, sometimes it's possible due to various concurrency techniques that the previous queries won't run in an order we intended for them. Issuing the previous commands from a shell or from DevCenter will always run in this order, but when the application issues multiple requests, it might lead to inconsistency.

Also, some systems might have clocks that are not 100 percent in sync. If, for instance, the clocks have offsets even in the milliseconds range, it is very important that the commands are issued exactly one after the other. Simply add **USING TIMESTAMP <integer>** to the statements that you want to run in a specific order. The Cassandra clusters will then use the provided **USING TIMESTAMP** instead of the time when they received the command. Using this option on an application level will, in most cases, prevent the caching of prepared statements, so use it wisely. Also, note that this option is usually not issued if users give the commands sequentially by typing.

Playing with the **TIMESTAMP** option might be a little tricky. For instance, if by some mistake we issue a **DELETE** that is in the future, we might work with an entry normally, and later wonder why it disappeared without having issued a **DELETE** command.

Anyway, the following code example issues the previous **INSERT** and **DELETE** statement combination with the **TIMESTAMP** option.

```
INSERT INTO offers (
       username, date, price, brand, model,
       year, mileage, color)
       VALUES ('jdoe', '2014-08-11 17:12:32+0200',
              7000, 'Toyota', 'Auris 2.0d', 2012, 15000, 'Blue')
              USING  TIMESTAMP 1406489822417000;


DELETE FROM offers USING TIMESTAMP 1406489822417001
    WHERE username = 'jdoe' AND date = '2014-08-11 17:12:32+0200';
```

Code Listing 38

The **INSERT** statement has the **TIMESTAMP** option specified at the end. In the **DELETE** statement, it's specified before the **WHERE** part of the statement. The **TIMESTAMP** option is rarely used, but it is included here because knowing about this option might save you some time when encountering weird issues when manipulating data in Cassandra.

Querying

So far, we've spent a lot of time exploring data manipulation because to work with queries, we have to create some data in the database. The command for retrieving data is called **SELECT**, and it has appeared in a couple of examples in previous sections of the book. For those of you with experience in classical SQL databases, you won't have any problems adapting to the syntax of the command, but the behavior of the command will probably seem a bit strange. Remember, with Cassandra, the emphasis is on scalability.

There are two key differences when comparing CQL to SQL:

- No **JOIN** operations are supported.
- No aggregate functions are available other than **COUNT**.

With classical SQL solutions, it's possible to join as many tables together as you like and then combine the data to produce various views. In Cassandra, joins are avoided because of scalability. It is a good practice for developers and architects think up-front about what they will want to query and then populate the tables accordingly during the write phase. This will cause a lot of data redundancy, but this is actually a common pattern with Cassandra. Also, we mentioned in a previous chapter that the disk is considered to be the cheapest resource when building applications with Cassandra.

The only allowed aggregate function is **COUNT**, and it is often used when implementing paging solutions. The important thing to remember with the **COUNT** function is that it won't return an actual count value in some cases, and also that this function is restricted with a limit. The most common limit value is 10000, so if the expected count is larger but the function returns exactly 10000, you may need to increase the limit to get the exact value. At the same time, be very careful and bear in mind that this might cause performance issues.

In the previous sections, we discussed the basic **SELECT** statements and limited ourselves to displaying specific columns because not enough space was available to show all of the table columns nicely. Building the queries was easy because we knew the keyspace names, table names, and column names in the tables. DevCenter makes this easier because of the graphical interface, but for those of you using the CQL shell, the situation requires a couple of commands to find your way around. The most useful command is **DESCRIBE**. As soon as you connect to Cassandra with the CQL shell, you can issue the following command.

```
cqlsh> DESCRIBE keyspaces;

system  used_cars  system_traces

cqlsh>
```

Code Listing 39

Now we see our **used_car** keyspace is available and we can issue the **USE** command on it. Now that we are in the **used_cars** keyspace, it would be nice if we could see what tables are available. We can use the **DESCRIBE** command with the **tables** parameter to do just that.

```
cqlsh> DESCRIBE tables;

Keyspace system
---------------
IndexInfo                hints        range_xfers
NodeIdInfo               local        schema_columnfamilies
batchlog                 paxos        schema_columns
compaction_history       peer_events  schema_keyspaces
compactions_in_progress  peers        schema_triggers
sstable_activity

Keyspace used_cars
------------------
offers_by_brand  test_comments        offers
test_data        users
```

```
Keyspace system_traces
----------------------
events  sessions

cqlsh>
```

Code Listing 40

Using the **`DESCRIBE`** command with the **`tables`** parameter gave us a list of tables. Most of the tables in the previous listing are system tables; we usually won't have any direct interaction with them through CQL. Now that we have a list of tables in the **`used_cars`** keyspace, it would be very interesting for us to see the columns in the table and their types. Let's look it up for the **`offers`** table.

```
cqlsh:used_cars> DESCRIBE TABLE offers;

CREATE TABLE offers (
  username text,
  date timestamp,
  brand text,
  color text,
  mileage int,
  model text,
  price float,
  year int,
  PRIMARY KEY ((username), date)
) WITH
  bloom_filter_fp_chance=0.010000 AND
  caching='KEYS_ONLY' AND
  comment='' AND
  dclocal_read_repair_chance=0.100000 AND
  gc_grace_seconds=864000 AND
  index_interval=128 AND
  read_repair_chance=0.000000 AND
  replicate_on_write='true' AND
  populate_io_cache_on_flush='false' AND
  default_time_to_live=0 AND
  speculative_retry='99.0PERCENTILE' AND
  memtable_flush_period_in_ms=0 AND
  compaction={'class': 'SizeTieredCompactionStrategy'} AND
  compression={'sstable_compression': 'LZ4Compressor'};

cqlsh:used_cars>
```

Code Listing 41

The most interesting result of the previous command is column names and their types. If you want to optimize and tweak the table behavior, you will probably use this command to check if a setting was successfully applied to the table. Let's check the current state in the **offers** table.

```
SELECT username, date, brand, model
    FROM offers;

 username | date                       | brand | model
----------+----------------------------+-------+-------
     jdoe | 2014-08-25 11:13:22+0200 |  Audi |    A3
   jsmith | 2014-05-11 01:22:11+0200 |  Ford | Orion
   jsmith | 2014-09-09 11:35:20+0200 |   BMW |  118d
   jsmith | 2014-09-19 11:35:20+0200 |   BMW |  120i
   jsmith | 2014-09-20 17:12:32+0200 |  Audi |    A6
     adoe | 2014-08-26 10:11:10+0200 |    VW |  Golf
```

Code Listing 42

It would be very interesting for us to see only the offers from a specific user. Since the user **jsmith** has the most offers, let's concentrate on them. Picking their offers will be done with the **WHERE** clause and specifying the **jsmith** username.

```
SELECT username, date, brand, model
    FROM offers WHERE username = 'jsmith';

 username | date                       | brand | model
----------+----------------------------+-------+-------
   jsmith | 2014-05-11 01:22:11+0200 |  Ford | Orion
   jsmith | 2014-09-09 11:35:20+0200 |   BMW |  118d
   jsmith | 2014-09-19 11:35:20+0200 |   BMW |  120i
   jsmith | 2014-09-20 17:12:32+0200 |  Audi |    A6
```

Code Listing 43

You might want to filter the **offers** table for multiple users and build a query that would return all of the offers for users **jsmith** and **adoe**.

```
SELECT username, date, brand, model
    FROM offers WHERE username = 'jsmith' OR username = 'adoe';

Bad Request: line 1:74 missing EOF at 'OR'
```

Code Listing 44

The **OR** operator does not exist in Cassandra. If we want to select multiple offers from the users, we have to use the **IN** operator.

```
SELECT username, date, brand, model
    FROM offers WHERE username IN ('jsmith', 'adoe');
```

```
 username | date                       | brand | model
----------+----------------------------+-------+-------
   jsmith | 2014-05-11 01:22:11+0200 |  Ford | Orion
   jsmith | 2014-09-09 11:35:20+0200 |   BMW |  118d
   jsmith | 2014-09-19 11:35:20+0200 |   BMW |  120i
   jsmith | 2014-09-20 17:12:32+0200 |  Audi |    A6
     adoe | 2014-08-26 10:11:10+0200 |    VW |  Golf
```

Code Listing 45

We'll concentrate on the offers from the **jsmith** user again and select the offer that the user made on a specific date. Remember, the offer is identified by the combination of the username and the date information for the offer. If you go back to the previous section, you will find that the username and date information form a primary key. The query is:

```
SELECT username, date, brand, model FROM offers
    WHERE username= 'jsmith'
        AND date = '2014-09-09 11:35:20+0200';

 username | date                       | brand | model
----------+----------------------------+-------+-------
   jsmith | 2014-09-09 11:35:20+0200 |   BMW |  118d
```

Code Listing 46

If we would like to get two specific orders for the user from the table, we could again use the **IN** operator as shown in the following example.

```
SELECT username, date, brand, model
    FROM offers WHERE username= 'jsmith'
AND date IN ('2014-09-09 11:35:20+0200', '2014-09-19 11:35:20+0200');

 username | date                       | brand | model
----------+----------------------------+-------+-------
   jsmith | 2014-09-09 11:35:20+0200 |   BMW |  118d
   jsmith | 2014-09-19 11:35:20+0200 |   BMW |  120i
```

Code Listing 47

But listing all of the dates to get the offers from the user is not the most efficient way to query the data. Think about it; we would actually have to know every piece of data in the list in order to get the listing. A more practice-oriented query would be to show all of the offers made by the user since a certain time or all of the offers made within a period of time. Let's look at the offers **jsmith** did in September.

```
SELECT username, date, brand, model
    FROM offers WHERE username= 'jsmith'
        AND date > '2014-09-01' AND date < '2014-10-01';

 username | date                       | brand | model
```

```
----------+--------------------------+-------+-------
   jsmith | 2014-09-09 11:35:20+0200 |   BMW |  118d
   jsmith | 2014-09-19 11:35:20+0200 |   BMW |  120i
   jsmith | 2014-09-20 17:12:32+0200 |  Audi |    A6
```

Code Listing 48

Some might set the end date to be '2014-09-30', but this would not return offers made on the last day of the month because CQL, by default, assumes the values for hours, minutes, and seconds are set to zero. So, specifying the 30th would, in effect, ignore all of the offers after midnight, or in other words, all the offers for that day.

The simplest way to approach this is to specify the first day of the following month as the end date with the less than operator. The previous query is not 100 percent correct, actually. The offers made between the stroke of midnight and the expiration of the first millisecond will not be included in the results. To cover this case, simply add the equal sign after the greater-than sign. The comparison operators are not always possible in Cassandra. Cassandra will allow comparison operators only in the situations where it can retrieve data sequentially.

In previous examples, all of the offers from the user are automatically sorted by their date by Cassandra. That's why we could use the greater-than and less-than operators. We did this on the clustering part of the primary key. Now, just for a moment, let's take a small step back and try some sort of comparison with the first part of the primary key. We will create a whole new table to do that, just to show the situation for the numbers:

```
CREATE TABLE test_comparison_num_part (
    a int,
    b text,
    PRIMARY KEY (a)
);

INSERT INTO test_comparison_num_part (a, b) VALUES (1, 'A1');
INSERT INTO test_comparison_num_part (a, b) VALUES (2, 'A2');
INSERT INTO test_comparison_num_part (a, b) VALUES (3, 'A3');
INSERT INTO test_comparison_num_part (a, b) VALUES (4, 'A4');

SELECT * FROM test_comparison_num_part WHERE a >= 3;

Bad Request: Only EQ and IN relation are supported on the partition key (unless
you use the token() function)
```

Code Listing 49

As the previous error message says, the first part of the primary key can only be retrieved with the equal sign and an **IN** operator. This is one of Cassandra's limitations when it comes to querying the data. Let's look into the other parts of the primary key. We will create a new and independent example to show Cassandra's behavior in this case.

```
CREATE TABLE comp_num_clustering (
      a text,
```

```
      b int,
      c text,
      PRIMARY KEY (a, b)
);

INSERT INTO comp_num_clustering (a, b, c) VALUES ('A', 1, 'A1');
INSERT INTO comp_num_clustering (a, b, c) VALUES ('A', 2, 'A2');
INSERT INTO comp_num_clustering (a, b, c) VALUES ('A', 3, 'A3');
INSERT INTO comp_num_clustering (a, b, c) VALUES ('A', 4, 'A4');

SELECT * FROM comp_num_clustering WHERE a = 'A' AND b > 2;

 a | b | c
---+---+----
 A | 3 | A3
 A | 4 | A4
```

Code Listing 50

The previous results look just fine. We specified what row we wanted to read and provided the range. Cassandra can find the row, do the sequential read of it, and then return it to us. This is possible because the values from column **b** actually become columns. So, in row **A** we don't have columns called **b**, but we have columns called **1:c**, **2:c**, etc., and the values in those columns are **A1**, **A2**, etc.

I'm reviewing how Cassandra stores tables because it is very important to understand the internals of Cassandra in order to use them as efficiently as possible.

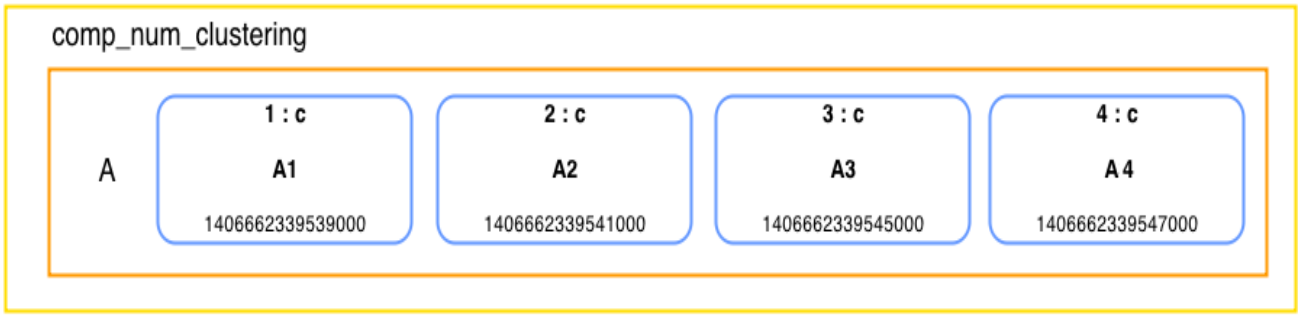

Figure 42: Cassandra storage of the table in Code Listing 50

Everything is fine with the ranges, but what would happen if we tried this on multiple rows? What if we wanted all of the data where **b** is greater than some number, let's say, 3?

```
SELECT * FROM comp_num_clustering WHERE b >= 3;

Bad Request: Cannot execute this query as it might involve data filtering and thus
may have unpredictable performance. If you want to execute this query despite the
performance unpredictability, use ALLOW FILTERING
```

Code Listing 51

This might seem a bit unexpected, especially to people coming from the relational world. The numbers are ordered so why we are not getting all of the data out? Well, the previous table might have multiple rows. In order to find all the data, the client would have to potentially contact other nodes, wait for their results, go through every row that it receives, and then read the data out of them. This simple query could very easily contact every node in the cluster and cause serious performance degradation. In Cassandra, everything is oriented toward performance. So, that kind of querying is not possible out of the box or even allowed in some cases. One way of making this kind of querying possible is with the **`ALLOW FILTERING`** option. The following query would return the results just as we might expect them.

```
SELECT * FROM comp_num_clustering WHERE b >= 3 ALLOW FILTERING;

 a | b | c
---+---+----
 A | 3 | A3
 A | 4 | A4
```

Code Listing 52

`ALLOW FILTERING` will let you run some queries that might require filtering. Use this option with extreme caution. It significantly degrades performance and should be avoided in production environments. The main disadvantage of the filtering is that it usually involves a lot of queries to other nodes and, if done on big tables, it may cause very long processing times that are pretty nondeterministic. While we are at it, let's try the filtering option for column **c**.

```
SELECT * FROM comp_num_clustering WHERE c = 'A4' ALLOW FILTERING;

Bad Request: No indexed columns present in by-columns clause with Equal operator
```

Code Listing 53

Indexes

The filtering option does not give us the ability to search the table by the data in the **c** column. The error message warns us that there are no indexed columns, so let's create an index in the table for the **c** column.

```
CREATE INDEX ON comp_num_clustering(c);

SELECT * FROM comp_num_clustering WHERE c = 'A4';

 a | b | c
---+---+----
 A | 4 | A4
```

Code Listing 54

The previous query returned the results once we created the index on the column. Creating indexes might seem the easy way to add searching capabilities to Cassandra. Theoretically, we could create an index on the table any time we needed to search for any kind of data, but creating indexes has some potentially dangerous downsides.

Cassandra stores the indexes just as it stores the other tables. The indexed value becomes the partition key. Usually, the more unique values there are, the better the index performance. On the other hand, maintaining this index will be a significant overhead for the system if the number of rows grows because every node will have to have most of the information that the other nodes store. The other possibility is that we have some binary index with some sort of true-false values. This would be an index with a very small number of unique values. That kind of index quickly gets many columns in just a couple of rows and becomes unresponsive over time because it is searching within the two very large rows. One important point is that if the node fails and we restore the data, the index will have to be rebuilt from scratch. As a rule of thumb, the best use for indexes is on relatively small tables with queries that return results in tens or hundreds but not more.

The index that we created in the previous example returned an automatic name because we simply forgot to specify a name for it. This is a very common mistake. Indexes are updated on every write to Cassandra and there are techniques that eliminate the need for them, so if we at some point in time decide to remove the index, we will run into trouble because we have to know its name in order to remove it. The following query makes it possible for us to find the index name together with other information that is relevant for identifying it.

```
SELECT column_name, index_name, index_type
   FROM system.schema_columns
     WHERE keyspace_name='used_cars'
       AND columnfamily_name='comp_num_clustering';


 column_name | index_name                | index_type
-------------+---------------------------+------------
           a |                      null |       null
           b |                      null |       null
           c | comp_num_clustering_c_idx | COMPOSITES
```

Code Listing 55

We have made an index in the **c** column of the **comp_num_clustering** table, so the index that we are looking for must be the last index in the table shown after running the previous query. The following example shows how to remove the index.

```
DROP INDEX comp_num_clustering_c_idx;
```

Code Listing 56

The previous index name seems pretty reasonable, and from the name we can determine what this index is all about. If you don't like the naming Cassandra does automatically, or you have some special naming convention that you have to enforce, then you can do name the index with the following command.

```
CREATE INDEX comp_num_clustering_c_idx ON comp_num_clustering(c);
```

Code Listing 57

This way, you can keep the index name in the initialization scripts for other nodes or development environments and avoid depending on the automatic index naming that might change in future versions of Cassandra.

Now that you know how Cassandra fetches data and that it's not always possible to get data with any column value out, let's look at the **ORDER BY** clause. Using this clause is not always possible. Most relational databases allow combinations of multiple columns on almost all queries. Cassandra limits this for performance gains. **ORDER BY** can be specified on one column only, and that column has to be the second key from the primary key specification. On the **comp_num_clustering** table, it's the **b** column; in the **offers**, it is the **date** column. Two possible orders are **ASC** and **DESC**. As an example, we will reverse the **jsmith** user's offers in time and put them in descending order. They are ascending by default because Cassandra sorts by column names, so there is no need to use **ASC** on the **offers** table.

```
SELECT username, date, brand, model
    FROM offers WHERE username= 'jsmith' ORDER BY date DESC;

 username | date                     | brand | model
----------+--------------------------+-------+-------
   jsmith | 2014-09-20 17:12:32+0200 |  Audi |    A6
   jsmith | 2014-09-19 11:35:20+0200 |   BMW |  120i
   jsmith | 2014-09-09 11:35:20+0200 |   BMW |  118d
   jsmith | 2014-05-11 01:22:11+0200 |  Ford | Orion
```

Code Listing 58

ORDER BY is not used often. It's more common to specify the clustering option when creating the table so that the data is automatically retrieved in the order that the system designers intended it. Most of the time, ordering is based on some kind of time-based column like a sensor, user activity, weather, or some other readings.

Collections

The primary use for collections in Cassandra is storing a small amount of denormalized data together with the basic set of information. The most common usages for collections are storing e-mails, storing properties of device readings, storing properties of events that are variable from occurrence to occurrence, and so on. Cassandra limits the amount of elements in a collection to 65,535 entries. You can insert data past this limit but the data is preserved and managed by Cassandra only up to the limit. There are three basic collection types:

- map
- set
- list

These structures are pretty common in most programming languages today, and most readers will have at least a basic understanding of them. We'll show each structure with a couple of examples, and while we're at it, we'll expand our data model a bit and make use of the collections.

We'll start with the map collection in our used car example. By now, we have defined the **offers** table and specified some basic offer attributes such as date, brand, color, mileage, model, price, and year. Now, we know that different cars have very different accessories in them. Some cars might run on different fuels, have different transmissions, or have different numbers of doors. We could cover all these extra properties without adding new columns to our **offers** table if we specified a **map** type column in the **offers** table as shown in the following example.

```
ALTER TABLE offers ADD equipment map<text, text>;
```

Code Listing 59

Now that we have the **equipment** map on our offers, let's talk a bit about the properties of the map collection. Long story short, a map is a typed set of key-value pairs. The keys in the map are always unique. One interesting property of the values in a map when stored in Cassandra is that the keys are always sorted.

There are two ways to define map key-value pairs. One is to update just the specific key in the map. The other is to define a whole map at once. Both ways are possible with the **UPDATE** command, but when using **INSERT**, you can specify just the whole map. Let's see what an **INSERT** would look like.

```
INSERT INTO offers (
       username, date, price, brand, model,
       year, mileage, color, equipment)
       VALUES ('adoe', '2014-09-01 12:02:52',
              12000, 'Audi', 'A4 2.0T', 2008, 99000, 'Black',
          {
          'transmission' : 'automatic',
          'doors' : '4',
          'fuel' : 'petrol'
          });
```

Code Listing 60

The syntax for defining the maps is pretty similar to the very popular JSON format. The comma separates the key-value pairs and the key is separated from the value by a colon. The saved row looks like the following.

```
SELECT equipment FROM offers WHERE username= 'adoe';

 equipment
------------------------------------------------------------
{'doors': '4', 'fuel': 'petrol', 'transmission': 'automatic'}
```

Code Listing 61

Notice that the stored map values are sorted and that they don't depend on the order we defined them. If we ever decide to change the values in the map or add a new one without discarding the old values, we can use the following **UPDATE** commands.

```
UPDATE offers SET
       equipment['doors'] = '5',
       equipment['turbo'] = 'yes'
  WHERE username = 'adoe'
        AND date='2014-09-01 12:02:52';
```

Code Listing 62

The previous query will update the **doors** key in the map and will add a new **turbo** key to it. This syntax is used when doing updates because we don't have to care if the key was created or not. We simply specify its value and Cassandra will update the key with the new value if it is able to find it, or it will define a new key if it's unable to find it. Let's see what the data looks like at the moment.

```
SELECT equipment FROM offers WHERE username= 'adoe';

 equipment
------------------------------------------------------------------
{'doors': '5', 'fuel': 'petrol', 'transmission': 'automatic', 'turbo': 'yes'}
```

Code Listing 63

If we wanted to redefine the complete map, we would simply set **equipment** to be equal to a whole new map definition as it was done in Code Listing 60. Still, sometimes we will want to remove just specific keys from the map. This is done with the **DELETE** command.

```
DELETE equipment['turbo'] FROM offers
    WHERE username = 'adoe'
        AND date='2014-09-01 12:02:52+0200';
```

Code Listing 64

Deleting a key that does not exist in the map is not a problem. You can run the previous example as many times as you like but it will delete the **turbo** key only on the first run. We have yet to cover the concept of time to live (TTL). For now, just remember that Cassandra can automatically delete data after it has lived for a specified period of time. For instance, if you insert or update a key-value with a time to live parameter, the time the data will actually live refers only to the changed, inserted key-value pair. All the other data have their own times to live. We will discuss the TTL concept more in depth later. For now, it's enough to know that collections specify a TTL on an element level, not on the row or the column level.

The next collection type we are going to dive into is the set. A set is a collection of unique values. Cassandra always orders sets by their value. Creating a column that is a set is done with the **set** keyword followed by the type of key elements in angled brackets. One of the most common uses for sets is saving user contact data such as e-mail addresses and phone numbers. Let's add a set of e-mails to the **users** table.

```
ALTER TABLE users ADD emails set<text>;
```

Code Listing 65

Now the **users** table can store more e-mails for every user. Inserting a new user is shown in the following example.

```
INSERT INTO users (
   username, password, first_name, last_name, state, emails)
    VALUES ('jqpublic', 'secret',  'John', 'Public', 'NY',
             {'j@example.com', 'p@example.com'});
```

Code Listing 66

The difference in sets when compared to maps is that there are no key elements, there are just values. If you add duplicates in the previous e-mail list, the duplicated values will not be stored in Cassandra. The values in the set are always sorted just as the keys in maps are.

The updating syntax is based on the plus and minus operators. If we wanted to first remove an e-mail address from the set and then add a new one, we would do it with the following queries.

```
UPDATE users SET
  emails = emails - {'j@example.com'} WHERE username = 'jqpublic';

UPDATE users SET
  emails = emails + {'jq@example.com'} WHERE username = 'jqpublic';
```

Code Listing 67

The previous queries result in the following emails for the user **jqpublic**.

```
SELECT username, emails
  FROM users WHERE username = 'jqpublic';
```

```
 username | emails
----------+-------------------------------------
 jqpublic | {'jq@example.com', 'p@example.com'}
```

Code Listing 68

The last collection type in Cassandra is a list. A list is a typed collection of non-unique values. The values in the list are ordered by their position. Map and set values are always sorted. The list elements always remain in the position where we put them. The most basic example would be a to-do list. We'll show how to use the list in the next examples. Before doing anything with the list, we have to define one.

```
CREATE TABLE list_example (
      username text,
      to_do list<text>,
      PRIMARY KEY (username)
);
```

Code Listing 69

Maps and sets are set in braces, but the list is in brackets. Other than that, the list is pretty similar to the previous two collection types. Let's see how to define a list when inserting data into a table.

```
INSERT INTO list_example (username, to_do)
    VALUES ('test_user', ['buy milk', 'send mail', 'make a call']);
```

Code Listing 70

The previous **INSERT** statement generated a simple to-do list for a user, with three elements in it. Let's have a look at what's in the table.

```
SELECT * FROM list_example;

 username  | to_do
-----------+-----------------------------------------
 test_user | ['buy milk', 'send mail', 'make a call']
```

Code Listing 71

When referencing the elements in the list, we do so by their position. The first element in the list has the position 0, the second 1, and so on. The following example shows how to update the first element in the to-do list of the user.

```
UPDATE list_example
    SET to_do[0] = 'buy coffee'
        WHERE username = 'test_user';
```

Code Listing 72

In some situations, we will want to add elements to the list.

```
UPDATE list_example
    SET to_do = to_do + ['go to a meeting', 'visit parents']
        WHERE username = 'test_user';
```

Code Listing 73

Cassandra can also remove occurrences of items from the list.

```
UPDATE list_example
    SET to_do = to_do - ['send mail', 'make a call']
        WHERE username = 'test_user';
```

Code Listing 74

It's also possible to delete elements with a specific index in the list.

```
DELETE to_do[1]
    FROM list_example
        WHERE username = 'test_user';
```

Code Listing 75

All of the previous changes in the list should result in the following.

```
SELECT * FROM list_example;

 username  | to_do
-----------+---------------------------------
 test_user | ['buy coffee', 'visit parents']
```

Code Listing 76

Now we know how to use collections in Cassandra. Collections are a relatively new concept in Cassandra. They are very useful in many day-to-day situations. Because they are very easy to work with, they might sometimes be overused. Be very careful and don't fill up collection columns with data that might be in the tables. All of the collection types are limited to 65,535 elements; having more would definitely be a sign that you are doing something wrong.

Time Series Data in Cassandra

Knowing how something changes over time is always important. By having the historical data about some kind of phenomenon, we can draw conclusions about the inner workings of that phenomenon and then predict how it's going to behave in the future. When talking about historical data, it usually consists of a measurement and a time stamp of when the measurement was taken. In statistics, these measurements are called data points. If the data points are in sequence, and if the measurements are spaced at uniform time intervals, then we are talking about time series data.

Time series data is very important in many fields of human interest. It is often used in:

- Performance metrics
- Fleet tracking
- Sensor data
- System logging
- User activity and behavior tracking
- Financial markets
- Scientific experiments

The importance of time series data is growing even more in today's world of interconnected devices that are getting smarter and, therefore, generate more and more data. Time series data is often associated with terms such as the Internet of Things (IoT). Every day, there are more devices generating more data. Until recently, most of this time series data was gathered on very specialized and expensive machines which, more or less, supported only vertical scaling. There aren't a lot of storage solutions out there that can handle large amounts of data and allow horizontal scaling at the same time. As a storage technology, Cassandra's performance is very comparable to expensive and specialized solutions; it even outperforms them in some areas.

So far we have gone pretty deep into some concepts of Cassandra, and we know that columns in Cassandra rows are always sorted by the column name. We also learned that Cassandra, in some cases, uses logical row values to form column names for the data that is going to be stored. Cassandra usually stores time series data presorted and fetches it with very little seeking on the hard drive.

We have also looked at modeling examples in our online used car market. When interacting with time series data, however, most tutorials use weather station data. We will also be using weather stations to introduce time series data because they use easy to understand concepts. Weather stations usually measure barometric pressure and the amount of precipitation, but we will limit all of our examples to temperature only to keep things simple.

Basic Time Series

Cassandra can support up to two billion columns in a single row. If you have a temperature measurement captured on a daily or hourly basis, a single row for every weather station will be enough. Each row in this table would probably be marked with some kind of weather station ID. The column name would be the time stamp of when the measurement was taken, and the data would be the temperature.

Let's place the time series examples into a separate keyspace. We will come back to the used cars in the later sections.

```
CREATE KEYSPACE weather
      WITH replication = {
                  'class': 'SimpleStrategy',
                  'replication_factor' : 1};
```

Code Listing 77

In its simplest form, basic weather station data could be gathered with the following table.

```
CREATE TABLE temperature (
   weatherstation_id text,
   measurement_time timestamp,
   temperature float,
   PRIMARY KEY (weatherstation_id,  measurement_time)
);
```

Code Listing 78

If the data is gathered every hour per year, one weather station would generate around 8,760 columns. Even if we added additional data to the previous table, the column number would not pass Cassandra's row length limitation in even the most optimistic predictions for the system's expected lifetime. Adding temperature readings to this table is shown in the following example.

```
INSERT INTO temperature
    (weatherstation_id, measurement_time, temperature)
        VALUES ('A', '2014-09-12 18:00:00', 26.53);
```

Code Listing 79

Feel free to add as many readings to the **`temperature`** table as you like. All of the inserted data will be saved under a weather station row. Column names will be the measurement time stamps, and temperatures will be the stored values. Let's look at how Cassandra stores this data internally.

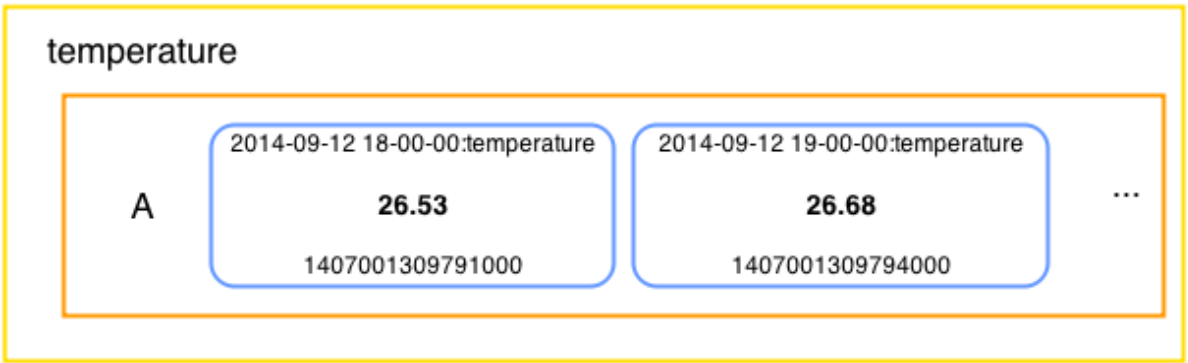

Figure 43: Single weather station data

In our temperature example, Cassandra will automatically sort all of the inserted data by the measurement time stamp. This makes reading the data very fast and efficient. When loading the data, Cassandra reads portions on the disk sequentially and this is where most of the efficiency and speed comes from.

Sooner or later, we will need to perform some kind of analysis on time series data. To do it in our example, we will fetch the temperature measurement data station by station. Fetching all of the temperature readings from weather station A would be done with the following query.

```
SELECT * FROM temperature WHERE weatherstation_id = 'A';

 weatherstation_id | measurement_time         | temperature
-------------------+--------------------------+-------------
                 A | 2014-09-12 18:00:00+0200 |       26.53
                 A | 2014-09-12 19:00:00+0200 |       26.68
                 A | 2014-09-12 20:00:00+0200 |       26.98
                 A | 2014-09-12 21:00:00+0200 |       22.11
```

Code Listing 80

If the application gathers data for a very long time, the amount of data in a row might become too impractical to be analyzed in a timely manner. Also, Cassandra is, in most cases, primarily used for storing and organizing the data. Almost any kind of analysis would have to be done on an application level since Cassandra does not have any aggregating functions other than **COUNT**. In reality, most of the analysis will not require you to fetch all of the data that was ever recorded by the weather station. In fact, most analysis will be for certain periods of time, such as a day, week, month, or year. To limit the results down to a specified period, the following query is used.

```
SELECT measurement_time, temperature
    FROM temperature
        WHERE weatherstation_id = 'A'
            AND measurement_time >= '2014-09-12 18:00:00'
            AND measurement_time <= '2014-09-12 20:00:00';

 weatherstation_id | measurement_time         | temperature
-------------------+--------------------------+-------------
                 A | 2014-09-12 18:00:00+0200 |       26.53
                 A | 2014-09-12 19:00:00+0200 |       26.68
                 A | 2014-09-12 20:00:00+0200 |       26.98
```

Code Listing 81

Note that if we ran the previous query without the greater than or equal to operator and used just the greater than operator, we would not include the readings for the time stamps that we specified in the query conditions. Sometimes this might lead to a faulty analysis or unexpected results.

Our weather station example will handle most basic time series usages. As you might have noticed, Cassandra gets along with time series data pretty easily, and running it enables horizontal scaling and reasonable running costs.

Reverse Order Time Series

In some cases, applications will be oriented toward recent data, and fetching historical data will not be as important. We know that Cassandra always sorts columns. So, if we selected just a few results from our weather station row, we would get the first results the weather station ever recorded. This is not very useful in dashboard-like applications where we just want to see the latest results. Furthermore, the old data might even be irrelevant to us, and we would want to remove it completely from our storage.

Cassandra handles this case very well because we can influence the way Cassandra stores and sorts a row in the table with the help of the **`CLUSTERING`** directive when defining the table. We talked about clustering in earlier sections of the book and we also looked at examples of it. Clustering is very important to creating a reverse order time series because it helps increase efficiency, especially if the data is kept for a longer period of time. In our example, we simply need to define a descending clustering on the **`temperatures`** table.

```
CREATE TABLE latest_temperatures (
   weatherstation_id text,
   measurement_time timestamp,
   temperature float,
   PRIMARY KEY (weatherstation_id, measurement_time)
) WITH CLUSTERING ORDER BY (measurement_time DESC);
```

Code Listing 82

Inserting the data is the same as it was in the **`temperatures`** table.

```
INSERT INTO latest_temperatures
    (weatherstation_id, measurement_time, temperature)
        VALUES ('A', '2014-09-12 18:00:00', 26.53);
```

Code Listing 83

When we retrieve the values, the latest data will automatically be on top.

```
SELECT * FROM latest_temperatures WHERE weatherstation_id = 'A';

 weatherstation_id | measurement_time         | temperature
-------------------+--------------------------+-------------
                 A | 2014-09-12 21:00:00+0200 |       22.11
                 A | 2014-09-12 20:00:00+0200 |       26.98
                 A | 2014-09-12 19:00:00+0200 |       26.68
                 A | 2014-09-12 18:00:00+0200 |       26.53
```

Code Listing 84

A query that returns the latest reading of a temperature from the station would look like the following.

```
SELECT *
    FROM latest_temperatures
        WHERE weatherstation_id = 'A'
            LIMIT 1;

 weatherstation_id | measurement_time         | temperature
-------------------+--------------------------+-------------
                 A | 2014-09-12 21:00:00+0200 |       22.11
```

Code Listing 85

We used a **`LIMIT`** option to fetch just one result. If we were interested in more than one of the latest readings, we would adjust the limit. All of the other queries are the same as in the basic time series. We could even use the basic time series and then use the **`ORDER`** clause to fetch the latest result. The following query uses **`temperature`** table, not **`latest_temperatures`**.

```
SELECT * FROM temperature
    WHERE weatherstation_id = 'A'
        ORDER BY measurement_time DESC
            LIMIT 1;

 weatherstation_id | measurement_time         | temperature
-------------------+--------------------------+-------------
                 A | 2014-09-12 21:00:00+0200 |       22.11
```

Code Listing 86

The previous query looks cumbersome when compared to querying a cluster ordered table. Besides the cumbersome queries, Cassandra will take longer to process this query because it will start at the beginning and then go through the records until it reaches the last one. With the cluster ordered table, Cassandra will simply get the first value from the row and return the result.

Tip: Use CLUSTERING ORDER BY COLUMN to reverse columns in row.

If only the latest data is of interest to us, we wouldn't want to fill up the database with data that we are never going to use. Cassandra has a mechanism that can give data an expiration date as a number of seconds when inserting.

```
INSERT INTO latest_temperatures
    (weatherstation_id, measurement_time, temperature)
        VALUES ('A', '2014-09-12 22:00:00', 26.88) USING TTL 20;

SELECT * FROM latest_temperatures
    WHERE weatherstation_id = 'A';

 weatherstation_id | measurement_time         | temperature
-------------------+--------------------------+-------------
                 A | 2014-09-12 22:00:00+0200 |       26.88
```

```
                A | 2014-09-12 21:00:00+0200 |       22.11


[wait for 20 seconds or more ...]

SELECT * FROM latest_temperatures
    WHERE weatherstation_id = 'A';

 weatherstation_id | measurement_time         | temperature
-------------------+--------------------------+-------------
                A | 2014-09-12 21:00:00+0200 |       22.11
```

Code Listing 87

After inserting data with a defined time to live (TTL), you just have to wait the specified number of seconds and Cassandra will mark that data with a tombstone and remove it in the next compaction process. In classical relational databases, we would have to write complex jobs that run in the background and remove the data. This is usually a very resource-hungry process because the databases often have to reorganize their indexes, etc., so most organizations do it overnight or during times when the system is under a lighter load.

TTL definition is always done in seconds, so if we need a TTL that is on the scale of months or years, we have to do a little bit of calculation. In most cases, data in Cassandra will be written from multiple application sources or will be changed manually by multiple operators. In effect, sometimes we can't really be sure how long the TTL is for some data. Cassandra provides two very useful functions for checking how long the data has to live and when the data was written: **`ttl`** and **`writetime`**.

```
INSERT INTO latest_temperatures
    (weatherstation_id, measurement_time, temperature)
        VALUES ('A', '2014-09-12 22:00:00', 26.88) USING TTL 20;

SELECT weatherstation_id AS w_id,
      temperature AS temp,
       ttl(temperature) AS ttl,
       writetime(temperature) AS wt
    FROM latest_temperatures WHERE weatherstation_id = 'A';

 w_id | temp  | ttl  | wt
------+-------+------+------------------
    A | 26.88 |    7 | 1407072070093000
    A | 22.11 | null | 1407063783604000
```

Code Listing 88

The **`ttl`** column from the previous query will have an integer value that is, in effect, a countdown timer. The data that have a **`null`** value as their TTL will remain there forever. The first row from the previous query has seven seconds until Cassandra will delete it automatically. Sometimes the write time is also important because the measurement time and the time when the data came into Cassandra will probably not match, and when searching for issues, the time the data came into Cassandra might turn out to be very important.

Anyway, there will most certainly be times when we will want to keep the data for a longer period of time than its current TTL specifies. The application requirements might change, we might have some faulty configuration in our application and we inserted the values with invalid TTLs, or we may find out that some of the data we are storing suddenly became operationally crucial for us. We can very easily change the TTL by updating the data and setting a new TTL value. Sometimes we will want to remove the TTL completely; this is also possible with CQL.

```
INSERT INTO latest_temperatures
    (weatherstation_id, measurement_time, temperature)
        VALUES ('A', '2014-09-12 22:00:00', 26.88)
            USING TTL 20;

SELECT weatherstation_id as w_id,
      temperature as temp,
       ttl(temperature) as ttl,
       writetime(temperature) as wt
    FROM latest_temperatures
       WHERE weatherstation_id = 'A';

 w_id | temp  | ttl  | wt
------+-------+------+------------------
    A | 26.88 |   11 | 1407073690443000
    A | 22.11 | null | 1407063783604000

UPDATE latest_temperatures
    USING TTL 60
        SET temperature = 26.88
            WHERE weatherstation_id = 'A'
                AND measurement_time = '2014-09-12 22:00:00';

[run the previous select again ...]

 w_id | temp  | ttl  | wt
------+-------+------+------------------
    A | 26.88 |   55 | 1407073766175000
    A | 22.11 | null | 1407063783604000

UPDATE latest_temperatures
    USING TTL 0
        SET temperature = 26.88
            WHERE weatherstation_id = 'A'
                AND measurement_time = '2014-09-12 22:00:00';

[run the previous select again ...]

 w_id | temp  | ttl  | wt
------+-------+------+------------------
    A | 26.88 | null | 1407073798438000
    A | 22.11 | null | 1407063783604000
```

Code Listing 89

Tip: Use TTL if you want specific data to expire automatically.

Handling Large-Scale Time Series

Cassandra's two billion columns per row limitation might seem like a lot, and it is. Most everyday usages will be just fine with this limit, but two billion is not a lot when we start generating data on a millisecond level. In fact, if some sensor generates data at a millisecond pace, the row in Cassandra would fill up in around a month's time. Whatever application we make, it will probably need to remain operational for more than a month.

The solution to this is to split the data into more rows. The most common way to split this data is by making a row for every combination of day and source ID (the weather station ID in our case). The data is then later assembled at the application level. Let's see how our temperature measuring weather station would look if the weather station generated the data at a millisecond pace.

```
CREATE TABLE temperature_by_day (
    weatherstation_id text,
    date text,
    measurement_time timestamp,
    temperature float,
    PRIMARY KEY ((weatherstation_id, date), measurement_time)
);
```

Code Listing 90

With this model, each day monitored by a weather station is stored in a separate row. The previous query shows that the partition key is a combination of weather station ID and the data. Data is inserted with the following query.

```
INSERT INTO temperature_by_day
    (weatherstation_id, date, measurement_time, temperature)
        VALUES ('A', '2014-09-12', '2014-09-12 18:00:00', 26.53);
```

Code Listing 91

The **`date`** column is usually generated at the application level automatically. The application usually takes the measurement time and then determines which partition the data that we want to insert will go into. In the previous example, we used a **`text`** column to make the partitions. This **`text`** column is set to the date part of the measurement time. There is no limitation for inserting measurement times into a partition. Times from the day before, the day after, or any random day can be inserted into a partition if done directly with CQL; it is simply a convention that enables our application to scale. Besides the textual date column, we could have used a simple integer column to represent the number of days since some day in the past, like 1970-01-01 or the day in the current year if we won't need the data for periods longer than a year. It all depends on the specific situation and varies from system to system.

This partitioning is very light on the application side when inserting data, but it makes the handling a lot easier. There might even be situations in which the data is gathered on a submillisecond level which is, most of the time, specific to scientific experiments. In that scenario, we would need to make partitions that are even smaller than the day.

A good practice would be to partition the data on an hourly level. We might even adhere to the following rule of thumb: the bigger partitions we expect, the smaller granularity of the partition key is. Let's look at how Cassandra would store the previously inserted data.

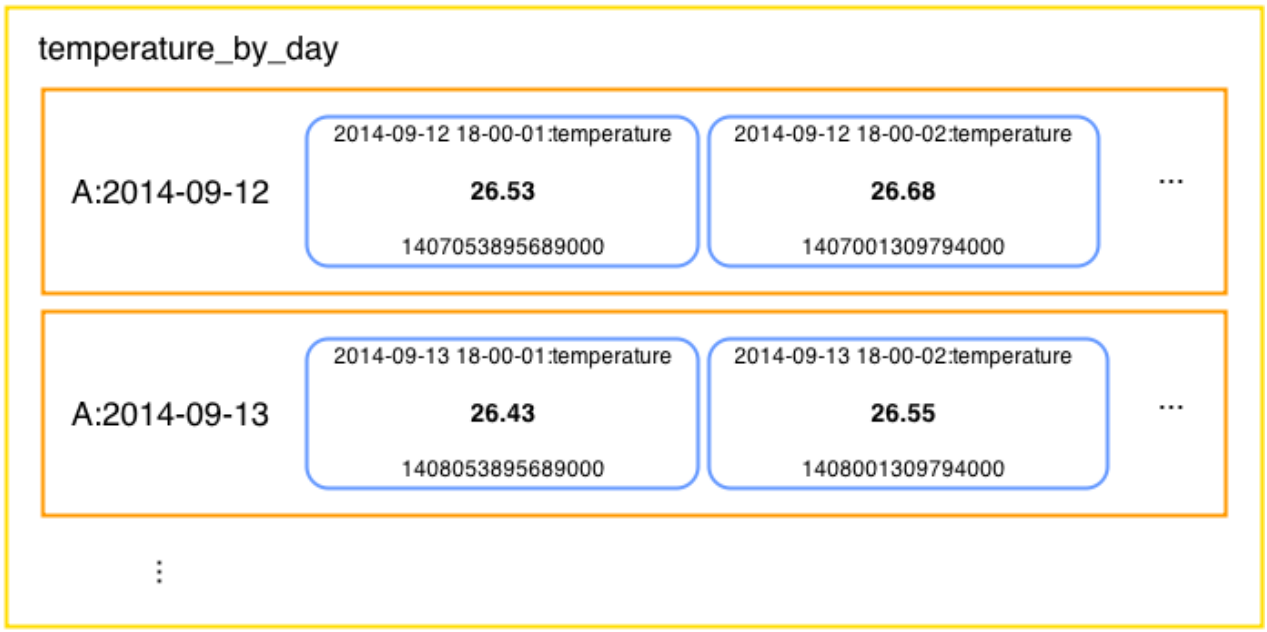

Figure 44: Day-level partitions of a single weather station's data

Getting the data out will not be as easy as it is with the basic example. As shown in the previous figure, the rows of single weather station data are now partitioned by the day the readings were made. To access data from a specific day, we have to specify the day.

```
SELECT weatherstation_id AS w_id, date,
      measurement_time AS t,
      temperature AS temp
    FROM temperature_by_day
        WHERE weatherstation_id = 'A' AND date = '2014-09-12';

 w_id | date       | t                         | temp
------+------------+---------------------------+-------
    A | 2014-09-12 | 2014-09-12 18:00:00+0200 | 26.45
    A | 2014-09-12 | 2014-09-12 18:00:01+0200 | 26.53
    A | 2014-09-12 | 2014-09-12 18:00:02+0200 | 26.68
    A | 2014-09-12 | 2014-09-12 18:00:03+0200 | 26.64
    A | 2014-09-12 | 2014-09-12 18:00:05+0200 | 26.77
```

Code Listing 92

Column names are a bit longer in this example, so aliases were used in displaying the columns so that the table could fit into the previous code listing. The alias syntax is relatively simple. To change the column names in the output, all you have to do is to specify the **AS** keyword after the column or function name and then provide the name to be displayed in the table when the results are fetched. As with the basic example, it is possible to navigate through the readings by specifying the boundary measurement times within the partition (a day in our case). The syntax is pretty similar to the basic use case.

```
SELECT weatherstation_id as w_id,
      date,
      measurement_time as t,
      temperature as temp
    FROM temperature_by_day
        WHERE weatherstation_id = 'A'
            AND date = '2014-09-12'
            AND measurement_time >= '2014-09-12 18:00:01'
            AND measurement_time <= '2014-09-12 18:00:03';

 w_id | date       | t                         | temp
------+------------+---------------------------+-------
    A | 2014-09-12 | 2014-09-12 18:00:01+0200 | 26.53
    A | 2014-09-12 | 2014-09-12 18:00:02+0200 | 26.68
    A | 2014-09-12 | 2014-09-12 18:00:03+0200 | 26.64
```

Code Listing 93

The examples up to now actually had precision to the seconds level. We used the formatted time stamps to make everything easier to understand. The CQL shell provides an easy way to insert time stamps on the milliseconds level. The numbers that we are going to use are not easily understandable by humans, so it's advisable to use some kind of converter from seconds and milliseconds to time stamps and vice versa. It's easy enough to search the Internet for something such as "convert timestamp online".

We'll insert a millisecond-level time stamp measurement for weather station **B** that occurred at **"2014-09-12 18:00:00"**—that would be a time stamp with a milliseconds value of 1410537600000. Let's insert three readings one after another.

```
INSERT INTO temperature_by_day
    (weatherstation_id, date, measurement_time, temperature)
        VALUES ('B', '2014-09-12', 1410537600000, 30.00);

INSERT INTO temperature_by_day
    (weatherstation_id, date, measurement_time, temperature)
        VALUES ('B', '2014-09-12', 1410537600001, 30.01);

INSERT INTO temperature_by_day
    (weatherstation_id, date, measurement_time, temperature)
        VALUES ('B', '2014-09-12', 1410537600002, 30.02);
```

Code Listing 94

Let's have a look at the **"2014-09-12"** partition for weather station B. To access it, we will need to specify the **WHERE** part of the **SELECT** to the desired partition.

```
SELECT weatherstation_id as w_id, date,
      measurement_time as t, temperature as temp
    FROM temperature_by_day
        WHERE weatherstation_id = 'B'
            AND date = '2014-09-12';


 w_id | date       | t                         | temp
------+------------+---------------------------+-------
    B | 2014-09-12 | 2014-09-12 18:00:00+0200 |    30
    B | 2014-09-12 | 2014-09-12 18:00:00+0200 | 30.01
    B | 2014-09-12 | 2014-09-12 18:00:00+0200 | 30.02
```

Code Listing 95

With the current formatting, it might seem as if all of the temperature readings happened at the same point in time. We know for a fact that we inserted the temperatures with spacing of one millisecond between the readings, and we incremented the temperature by a small amount for every temperature reading just so we could identify them later. Cassandra sorts the reading by the measurement time stamp automatically, so the previous listing seems fine from that point of view. Still, we would like to see the milliseconds instead of the time stamp just to make sure the data is actually stored with the time precision that we require.

To do this, we are going to use a little trick. Cassandra handles binary data pretty well; to do so it is equipped with a lot of functions that enable converting any kind of Cassandra native type to its binary representation and converting binary data to any of the basic types. In Cassandra documentation, these functions are often called the blob functions. You may remember that the time stamp in Cassandra is saved as a 64-bit signed integer representing the number of milliseconds that have passed since midnight 1970-01-01. We could convert the time stamp to its binary representation and then this binary representation back to a number as shown in the following example.

```
SELECT weatherstation_id as w_id, date,
       blobAsBigint(timestampAsBlob(measurement_time)) as t,
      temperature as temp
    FROM temperature_by_day
        WHERE weatherstation_id = 'B'
            AND date = '2014-09-12';

 w_id | date       | t             | temp
------+------------+---------------+-------
    B | 2014-09-12 | 1410537600000 |    30
    B | 2014-09-12 | 1410537600001 | 30.01
    B | 2014-09-12 | 1410537600002 | 30.02
```

Code Listing 96

In the previous example, we took the time stamp column and converted it to a binary value with the **timestampAsBlob** function. After that, we converted this binary value to an integer and then displayed it in the column **t**, all with the help of the **blobAsBigint** function. Knowing the millisecond value of a time stamp is very useful in situations where you are not sure what time stamp value is actually stored in the column.

For instance, if a result does not fall into a queried range when we are making a **SELECT** statement or something similar, we will wonder why the row is not displayed in the results. If we look at the millisecond level, we might see that the time stamp is actually a couple of milliseconds greater than zero, and because of that it is not falling within the expected range. We could then adjust the query to cover the omitted results or fix the time stamp value on the row. In Cassandra, a lot of things can and will happen one after another, and the default time stamp precision on the seconds level may be inadequate. We will discuss this further in the following section.

Millisecond is a Long Time

As information systems evolve more and more, things start to happen in a smaller and smaller time frame. Nowadays, there are more systems than ever, where hundreds of events occur within the same millisecond. Most of the time, it's very hard to synchronize information systems' clocks to a greater precision than a millisecond. So, the majority of devices, components, and systems don't keep track of the events on a submillisecond level. Even if they did, the timers would probably be in some kind of offset because the clocks probably wouldn't be perfectly synchronized.

Now imagine that we are using Cassandra to gather measurements that come from multiple devices. While we're at it, imagine that there are literally thousands of sources sending information to Cassandra, and that the information the sources are generating comes at a millisecond pace. Now what would happen if, let's say, ten messages come in within the same millisecond?

The initial answer might be that we need to increase the resolution of a clock and then simply make the measurement mechanisms more precise, but remember that the synchronization on a submillisecond level might reveal itself as pretty impractical. If this solution is not possible, then we could perhaps add additional bytes to the time stamp that we are saving and fill the additional bytes with some pseudorandom values so that we could effectively squeeze more events into a single millisecond. In essence, the Cassandra **timeuuid** type does exactly this. In fact, the **timeuuid** type can squeeze so much data into the same millisecond that if we did a billion writes per second over a hundred years, we would have just a 50 percent chance of getting a single **timeuuid** value collision.

Tip: Use the timeuuid type when multiple events occur within the same millisecond.

There are multiple CQL functions that enable us to work with **timeuuid** more easily:

- **now()**
- **dateOf()**

- **unixTimestampOf()**
- **minTimeuuid() and maxTimeuuid()**

We will describe how to use these functions through examples. Our weather measurements are not changing every millisecond. To make the examples more practice-oriented, we will use a sample table that monitors high-volume stock exchange trading. Let's start by defining a new keyspace.

```
CREATE KEYSPACE high_volume_trading
       WITH replication = {
                    'class': 'SimpleStrategy',
                    'replication_factor' : 1};
```

Code Listing 97

Stock ticks usually consist of the transaction price and a time stamp, something like the following example.

```
CREATE TABLE stocks_ticks (
    symbol text,
    day int,
    time timeuuid,
    details text,
    PRIMARY KEY ((symbol, day), time)
) WITH CLUSTERING ORDER BY (time DESC);
```

Code Listing 98

Now that we have the table with the **timeuuid** column set up, let's see how the **timeuuid** functions are used. Let's start with inserting the data. The **timeuuid** has 32 hexadecimal digits in it. It would be very hard for a human to generate these values, so in effect, we wouldn't even be able to insert the data into tables. To insert the current **timeuuid**, we use the **now** function. This function creates a **timeuuid** with the current time stamp. If we wanted to insert a stock tick, we would do it as shown in the following code listing.

```
INSERT INTO stocks_ticks (symbol, day, time, details)
    VALUES ('KTCG', 220, now(), 'BUY:2000');

INSERT INTO stocks_ticks (symbol, day, time, details)
    VALUES ('KTCG', 220, now(), 'SELL:2001');

INSERT INTO stocks_ticks (symbol, day, time, details)
    VALUES ('KTCG', 220, now(), 'BUY:2000');

INSERT INTO stocks_ticks (symbol, day, time, details)
    VALUES ('KTCG', 220, now(), 'BUY:2002');
```

Code Listing 99

To extract the date from a **timeuuid**, we use the **dateOf** function.

```
SELECT time, dateOf(time) FROM stocks_ticks;

 time                                 | dateOf(time)
--------------------------------------+--------------------------
 24f13590-1f06-11e4-b22e-55bb596001d5| 2014-08-08 16:13:17+0200
 24f0c060-1f06-11e4-b22e-55bb596001d5| 2014-08-08 16:13:17+0200
 24f02420-1f06-11e4-b22e-55bb596001d5| 2014-08-08 16:13:17+0200
 24efaef0-1f06-11e4-b22e-55bb596001d5| 2014-08-08 16:13:17+0200
```

Code Listing 100

This is an example where every millisecond is very important and we will probably want to extract the exact milliseconds from a **`timeuuid.`** To do that, we will use the **`unixTimestampOf`** function.

```
SELECT time, unixTimestampOf(time) FROM stocks_ticks;


 time                                 | unixTimestampOf(time)
--------------------------------------+-----------------------
 24f13590-1f06-11e4-b22e-55bb596001d5 |         1407507197801
 24f0c060-1f06-11e4-b22e-55bb596001d5 |         1407507197798
 24f02420-1f06-11e4-b22e-55bb596001d5 |         1407507197794
 24efaef0-1f06-11e4-b22e-55bb596001d5 |         1407507197791
```

Code Listing 101

Although the millisecond values from the previous listing are not the same because the client I was using couldn't insert them fast enough, it could easily happen that the millisecond part is exactly the same. When retrieving the results, this would mean that within the same millisecond there could be multiple results. CQL provides two functions that effectively enable us to get all of the data that have the same millisecond time stamp. They are **`minTimeuuid`** and **`maxTimeuuid`**, and they return minimum and maximum **`timeuuid`** value for any given time stamp. For instance, to get all stock ticks in a single second we would use the following query.

```
SELECT * FROM stocks_ticks
    WHERE symbol = 'KTCG' AND day = 220
        AND time > minTimeuuid('2014-08-08 16:13:17')
        AND time < maxTimeUUID('2014-08-08 16:13:18');

symbol | day | time                                 | details
-------+-----+--------------------------------------+-----------
  KTCG | 255 | 24f13590-1f06-11e4-b22e-55bb596001d5 |  BUY:2002
  KTCG | 255 | 24f0c060-1f06-11e4-b22e-55bb596001d5 |  BUY:2000
  KTCG | 255 | 24f02420-1f06-11e4-b22e-55bb596001d5 | SELL:2001
  KTCG | 255 | 24efaef0-1f06-11e4-b22e-55bb596001d5 |  BUY:2000
```

Code Listing 102

Summary

This is the longest chapter in the book and we covered many things in it. CQL is the most important way of interacting with Cassandra so we had a lot of everyday usages to cover. We started off by modeling a used car market in the relational world and then examined how Cassandra would handle the same data, and what would be the best practices for using Cassandra to handle the data modeled by the relational approach.

Next, we described how Cassandra is a rather specific storage engine, so we looked closely at how Cassandra physically stores the data. We talked about tables and their properties, and covered Cassandra data types. While explaining these concepts, we covered the common cases in everyday use. We also mentioned that Cassandra has very limited searching capabilities, at least out of the box, so we described indexes and how to use them.

Since Cassandra is often used for time series data, we discussed the time to live (TTL) concept because time series data is usually compacted or assigned an expiration time in most systems. We also looked into handling data on a submillisecond level. CQL and data modeling are at the core of any Cassandra-based application, so this chapter is the most important one in the book.

Chapter 4 Using Cassandra with Applications

Up until now, we have been concentrating mostly on Cassandra and CQL. Understanding how Cassandra works and knowing CQL is essential to be able to use Cassandra with applications. Cassandra has drivers for most of the popular languages available today.

Table 6: Cassandra Drivers

Language	Driver
Java	https://github.com/datastax/java-driver
C#	https://github.com/datastax/csharp-driver
Python	https://github.com/datastax/python-driver
Clojure	https://github.com/clojurewerkz/cassaforte
Erlang	https://github.com/iamaleksey/seestar
Go	https://github.com/tux21b/gocql
Haskell	https://github.com/Soostone/cassy
Node.js	https://github.com/jorgebay/node-cassandra-cql
Perl	https://github.com/mkjellman/perlcassa
PHP	http://github.com/thobbs/phpcassa
Ruby	https://github.com/iconara/cql-rb
Scala	https://github.com/websudos/phantom

In this chapter, we will show how to use Cassandra from Node.js, C#, and Java. Most of this chapter will be oriented toward the Node.js application.

Cassandra with Node.js

Node.js is a very popular technology today and is currently used by companies such as Microsoft, VMWare, eBay, Yahoo, and many others. Node.js became popular partly because most web developers didn't have to learn a new language in order to start using it. Some people describe Node.js as server-side JavaScript. JavaScript has a very interesting event model and is a good fit for asynchronous programming. There are a lot of other factors that contribute to the increasing popularity of Node.js.

Some important Node.js features include:

- Configuration is done with JSON.
- Code can be reused on the client and server-side.
- Vibrant community and very easy package sharing.
- Simple and dependable package manager.
- Minimalistic and lightweight environment; users include only what they want to use.
- Several tools to support production usage of the technology.

To follow along in this section, you will need to install Node.js. Installing Node.js is a straightforward and well-documented process; all of the information can be found at nodejs.org. In this section, we will build an app based on our used car market example. The provided example will not be 100 percent production-ready; it is more oriented toward showing how to interact with Cassandra from Node.js applications. Most of the time, we will ignore security best practices and user input validation, but we will use the most common and popular Node.js libraries to build our example.

Initial Setup

Before doing any work, we have to determine the directory where we will hold the sources. Choose any folder you like but always avoid using spaces when naming folders and choosing locations as it might lead to undesired behavior across various platforms. To start everything up, we need to create an empty folder called "cass_node_used_cars" and then position ourselves in this folder. The first command we will run will be **`npm init`**; this command creates the **`package.json`** file. This file contains all the metadata for the project. To create it, simply issue the following command and answer the questions the command will ask you.

```
# npm init
```

Code Listing 103

The generated **`package.json`** file should look something like the following.

```
{
    "name": "cass_node_used_cars",
    "version": "0.0.1",
    "description": "Cassandra used cars example in node.js",
    "main": "index.js",
    "scripts": {
        "test": "echo \"Error: no test specified\" && exit 1"
    },
    "author": "Marko Švaljek (http//msvaljek.blogspot.com/)",
    "license": "ISC"
}
```

Code Listing 104

Besides the **package.json** file generated with the help of the **npm init** command, we need to create additional files.

```
.
├── cassandra.js
├── config.js
├── index.js
├── node_modules
├── package.json
├── public
├── routes.js
└── views
    ├── add_offer.html
    ├── add_user.html
    ├── home.html
    ├── search.html
    └── layouts
        └── main.html
```

Code Listing 105

The **node_modules** directory will be created automatically in the later steps so you don't have to create it right away. The next step is to install the libraries. To interact with Cassandra, we are going to use a library called **node-cassandra-cql**. By default, the Node.js CQL library is installed to be compatible with previous versions of Cassandra.

In our demonstration app, we are going to show how to use lightweight transactions and batch statements. To use these Cassandra capabilities, we have to use the **v2** protocol which is not used by default. To install it, we will simply command **npm** to install the **protocol12** version of the module. To use a specific package, we have to specify its name after an **@** symbol when installing the module. We will install the modules and use the **save** option so that **npm** updates the **package.json** file. The **public** folder contains publically available assets such as images, and JavaScript and CSS files.

Remember that this folder is visible to everybody. The **views** folder contains the template files for the application. The files with a **.js** extension contain the logic of the application. Later, we will use the **index.js** file to start the application. At the moment, it would be best if we installed the required modules by issuing the following commands.

```
# npm install node-cassandra-cql@protocol2 -save
# npm install express –save
# npm install express3-handlebars –save
# npm install moment –save
# npm install body-parser -save
# npm install promise –save
```

Code Listing 106

We are going to build a minimal web interface for our application. The most popular web application framework for Node.js is Express, so we are going to use it. For writing the template logic, we are going to use Handlebars. To work with dates, we will use the Moment library. Our application will process HTTP **POST** requests. In earlier versions, Express handled such requests out of the box, but now we have to use a middleware component called **body-parser**.

We are also going to demonstrate a searching technique used in Cassandra that doesn't require defining Cassandra indexes on tables or using some kind of search technology on top of Cassandra. This technique will use the Promise library. After running the previous **install** commands, the **package.json** file should have a **dependencies** section.

```
  "dependencies": {
    "node-cassandra-cql": "^0.4.4",
    "express": "^4.8.2",
    "express3-handlebars": "^0.5.2",
    "moment": "^2.8.1",
    "body-parser": "^1.6.2",
    "promise": "^5.0.0"
  }
```

Code Listing 107

The tables from the previous chapters are not 100 percent tailored to the example application, and we used them to show various techniques, so we need to create two more tables.

```
CREATE TABLE car_offers (
       country text,
       date timeuuid,
       username text,
       brand text,
       color text,
       equipment map<text, text>,
       mileage int,
       model text,
       price float,
       year int,
```

```
      PRIMARY KEY (country, date)
) WITH CLUSTERING ORDER BY (date DESC);

CREATE TABLE car_offers_search_index (
      brand text,
      color text,
      price_range text,
      creation_time timeuuid,
      date timeuuid,
      country text,
      PRIMARY KEY ((brand, color, price_range), creation_time)
);
```

Code Listing 108

In the example, we are going to partition the data by the country from which the offer is made. The app will have the country fixed to USA. It only adds complexity if we add more countries and doesn't make explaining easier. Besides the new **offers** table, we will create an index for searching the offers. Building a more complex searching index would also require a lot of time and provide very little additional insight into using Cassandra in an application. In the following sections, we will look closely at example files in our application.

index.js

```
var express = require('express');

var app = express();

require('./config')(app);
require('./routes')(app);

app.listen(8080);
console.log('Application is running on port 8080');
```

Code Listing 109

This module initializes the Express framework. After **express** has been initialized, we initialize the **config.js** and **routes.js** modules. The web application is running on port 8080. If everything works fine, you should see the message from **console.log**. If you plan to follow along with building this example, it would be best if we didn't have to restart the Node application just to see if some change has taken effect. The best practice is to install the **nodemon** tool and then run the example with it as shown in the following code.

```
# npm install -g nodemon
# nodemon index.js
```

Code Listing 110

After starting the application with **nodemon**, it will track changes and automatically reload the application when something changes. The example we are building is not fully functional yet. Don't be surprised if you receive error messages after trying to run the index.js file as shown in previous listing. Just continue editing the files.

config.js

This file contains the entire configuration for Express and the other frameworks we are using. We also have to load the Express module so that we can define the static directory. The Handlebars module has a bit more configuration that we need to do. The Handlebars framework doesn't come with out-of-the-box support for formatting dates and comparing strings, so we need to define **formatDate** and **if_eq** helpers. For date formatting, we will use Moment.js. The Express framework externalized support for parsing post requests, so we have to use the **body-parser** module. We also have to define the layouts directory and configure the view engine. All of these changes are shown in the following code listing.

```
var express = require('express'),
    bodyParser = require('body-parser'),
    handlebars = require('express3-handlebars'),
    moment = require('moment');

module.exports = function(app){
    app.engine('html', handlebars({
        defaultLayout: 'main',
        extname: ".html",
        layoutsDir: __dirname + '/views/layouts',
        helpers: {
            formatDate: function(value) {
                return moment(value).format('YYYY-MM-DD HH:mm:ss');
            },
            if_eq : function(a, b, opts) {
                if(a == b) {
                    return opts.fn(this);
                }
                else {
                    return opts.inverse(this);
                }
            }
        }
    }));

    app.use(bodyParser.urlencoded({extended: false}));

    app.set('view engine', 'html');

    app.set('views', __dirname + '/views');

    app.use(express.static(__dirname + '/public'));
};
```

Code Listing 111

Express supports more than 20 other view technologies, with the most popular being Jade, Haml, and Underscore. In our example, we will stick to plain old HTML.

cassandra.js

```
var cql = require('node-cassandra-cql');
var client = new cql.Client({
      hosts: ['127.0.0.1:9042'], keyspace: 'used_cars'
});
module.exports = {
      client: client,
      cql: cql
};
```

Code Listing 112

This file initializes the Cassandra driver. The **`hosts`** value and **`keyspace`** parameter would normally be loaded from a configuration file, but in this example application, it's sufficient to do it directly in the file. This module exports the **`client`** to enable other modules to query Cassandra and **`cql`** to make type handling in later examples easier.

routes.js

This is the most complex file, so we are going to review each function separately. Normally the logic would be split into multiple modules, but in this example application the logic is simply put in the routes file. You can jump to view sources and add them to the example as you define the route logic.

```
var cassandra = require('./cassandra'),
      client = cassandra.client,
      cql = cassandra.cql,
      Promise = require('promise');

module.exports = function (app) {
    // all of the following methods go here
};
```

Code Listing 113

Now we'll start with adding the routes and defining the logic for them. Let's define the logic for the home page. The home page should display all of the car offers right away. The request will be in the **`get`** form, and there will be no specified URL or page in the parameter. We do this by simply defining an **`app.get`** with the route and a callback for the request processing.

```
app.get('/', function (req, res) {
    client.execute('SELECT dateOf(date) as date, username,\
                    brand, color, mileage, model, price, year \
                    FROM car_offers', [],
```

```
        function(err, result) {
            if (err) {
                res.render('500');
            } else {
                res.render('home', {rows: result.rows});
            }
        });
});
```

Code Listing 114

To display the home page, we simply select all of the offers from Cassandra. To use the **`execute`** method from the Cassandra client, we have to provide the following parameters:

- Query string
- Parameters
- Callback function

The **`SELECT`** query we used returns a **`timeuuid`** type for the date. To make handling the data easier, we converted the **`timeuuid`** to **`date`** directly in the query. If the Cassandra query doesn't throw any errors, we will redirect the client to the home view. On the home view, the application user will have buttons to create user profiles in the application. The button for creating the user profile will lead to the first route in the following code listing.

```
app.get('/show/adduser', function (req, res) {
    res.render('add_user', {});
});

app.post('/adduser', function (req, res) {
    client.execute('INSERT INTO users (\
        username, emails, first_name, last_name,\
        password, state)\
        VALUES (?, null, ?, ?, null, null) IF NOT EXISTS',
        [req.param('username'), req.param('first_name'),
        req.param('last_name')],
        function(err) {
            if (err) {
                res.render('add_user', {
                    username: req.param('username'),
                    first_name: req.param('first_name'),
                    last_name: req.param('last_name'),
                    errorOccured: true
                });
            } else {
                res.redirect('/');
            }
        });
});
```

Code Listing 115

Showing the **add_user** view is relatively simple; we just need to show the view and don't have to prepare any data for it. When the form from the view sends a post request to the **adduser** route, we extract the parameters from the post and fill in the parameters for inserting into the **users** table. We won't use all of the fields in our example, so we simply set them to **null** in the query. We also use a lightweight transaction with the **IF NOT EXISTS** so no two users with the same username will ever be created.

This is a neat Cassandra feature, but to use it, we had to install the **protocol12** version of the driver. If an error occurs, we display the **add_user** page again, but this time we pass in the parameters so that the user doesn't lose values in the form, and we also provide the view with the error flag so that the error message is displayed to the user. If there were no errors on **INSERT**, we simply redirect the user back to the home page.

Now that we have our users added to the application, we want to create offers for them. The offers form will display the **users** list in a selection box, so before displaying the **add_offer** page, we need to fetch the users from Cassandra. The **SELECT** statement is pretty simple; we just list the columns that we need. Note that we only use the username and the first and last name. If we are unable to fetch the users from Cassandra, we will simply render an internal error page. If we get the users, we will render the **add_offer** page.

```
app.get('/show/addoffer', function (req, res) {
    client.execute('SELECT username, first_name, last_name \
                    FROM users', [],
        function(err, users) {
            if (err) {
                res.render('500');
            } else {
                res.render('add_offer', {users: users.rows});
            }
        });
});
```

Code Listing 116

After the user is finished providing details in the **add_offer** view, the form will be posted to the **addoffer** route.

```
app.post('/addoffer', function (req, res) {
    var offer_timeuuid = cql.types.timeuuid();

    client.execute('INSERT INTO car_offers (\
        country, date, brand, color, equipment,\
        mileage, model, price, username, year)\
        VALUES (?, ?, ?, ?, null, ?, ?, ?, ?, ?)',
        [req.param('country'),
        offer_timeuuid,
        req.param('brand'),
        req.param('color'),
        parseInt(req.param('mileage'), 10),
        req.param('model'),
        {
```

```
            value: parseFloat(req.param('price')),
            hint: cql.types.dataTypes.float
        },
        req.param('username'),
        parseInt(req.param('year'), 10)],
        function(err) {
            if (err) {
                res.render('add_offer', {
                    brand: req.param('brand'),
                    color: req.param('color'),
                    mileage: req.param('mileage'),
                    model: req.param('model'),
                    price: req.param('price'),
                    username: req.param('username'),
                    year: req.param('year'),
                    errorOccured: true
                });
            } else {
                makeSearchIndexEntries(
                    req.param('brand'),
                    req.param('color'),
                    parseFloat(req.param('price')),
                    offer_timeuuid, req.param('country'));

                res.redirect('/');
            }
        });
});
```

Code Listing 117

We didn't let Cassandra generate the **timeuuid** date field automatically with the help of the **now** function because we need this value to create search index entries. Instead, we generated one on the application side with the **timeuuid()** function. The parameters are sent to the server as strings. Most of the time this is fine, but some parameters have to be converted to other types. We used the **parseInt** and **parseFloat** JavaScript functions to do that. If we simply passed the parsed float to the parameters, the driver would insert some very strange values with large negative exponents instead. So we inserted the price as an object with two fields. The first one is the **value**, and the second is the **hint** in which we say to the driver that this is a **float** value. The driver then behaves fine and inserts the floats normally.

If the **INSERT** statement throws an error, we return to the **add_offer** page and propagate the posted data so that the user doesn't lose the entered values. In cases where everything works as expected, we make index search entries and return to the home page. We built the search index for the field's brand, color, and price range. The search index we are going to build doesn't require any modifications to Cassandra's installation, and the good side is that we can back it up and it will be distributed among the nodes in the cluster. The downside of this search index is that it generates a lot of writes to Cassandra, but that's just fine because in Cassandra philosophy, the disk is cheap and the writes are fast.

```
function makeSearchIndexEntries(brand, color, price, date, country) {
```

```
    var insertQuery = 'INSERT INTO car_offers_search_index\
    (brand, color, price_range, creation_time, date, country)\
    VALUES (?, ?, ?, now(), ?, ?)';

    brand = brand.toLowerCase();
    color = color.toLowerCase();
    var price_range = convertPriceToRange(price);

    var queries = [
     {query: insertQuery, params: ['', '', '', date, country]},
     {query: insertQuery, params: ['', '', price_range, date, country]},
     {query: insertQuery, params: ['', color, '', date, country]},
     {query: insertQuery, params: ['', color, price_range, date, country]},
     {query: insertQuery, params: [brand, '', '', date, country]},
     {query: insertQuery, params: [brand, '', price_range, date, country]},
     {query: insertQuery, params: [brand, color, '', date, country]},
     {query: insertQuery, params: [brand, color, price_range, date, country]}
    ];

    var consistency = cql.types.consistencies.one;

    client.executeBatch(queries, consistency, function(err) {
        if (err) {
            console.log('error inserting ');
        }
    });
}

function convertPriceToRange(price) {
    if (price > 0) {
        if (price < 1000) {
            return '1';
        }
        if (price < 3000) {
            return '2';
        }
        if (price < 5000) {
            return '3';
        }
        if (price < 10000) {
            return '4';
        }
        return '5';
    }

    return '0';
}
```

Code Listing 118

To make the index building as fast as possible, we are going to use a batch statement. **executeBatch** is available in **version12** of the driver. The batch statement is also going to make inserting the values a bit easier because we need to insert all of the combinations that are possible for an offer. Also, the **brand** and the **color** values are always saved and queried in lowercase so that we don't have to worry about results not showing up in a search because of incorrect casing.

In every statement, the **date** and **country** parameters are always inserted. The **brand**, **color**, and **price_range** are combined in every possible manner. For simplicity, we used three variables resulting in eight **INSERT** statements, which is enough for demonstration purposes. Setting the quorum level to **one** also speeds things up because it requires an acknowledgement from just one node to continue. The price is not stored directly in the index. Instead, we use a price range. Building the price ranges is done with the **convertPriceToRange** function which maps the price to a simple string.

Now, we will move to the search form. First, we need to display the search form to the user.

```
app.get('/show/search', function (req, res) {
    res.render('search', {});
});
```

Code Listing 119

The search form will then post the search parameters to the search route. The **search** route will process the parameters and then make a **SELECT** to the index table.

```
app.post('/search', function (req, res) {
    var accumulator = [];

    var brand = req.param('brand').toLowerCase();
    var color = req.param('color').toLowerCase();
    var price_range = req.param('price_range');

    client.execute('SELECT country, date \
                    FROM car_offers_search_index \
                    WHERE brand=? AND color=? AND price_range = ?',
        [brand, color, price_range],
        function(err, result) {
            if (!err) {
                var prevPromise = Promise.resolve();

                result.rows.forEach(function (row) {
                    prevPromise = prevPromise.then(function () {
                        return new Promise(function (resolve, reject) {
                            client.execute(
                                'SELECT dateOf(date) as date, username,\
                                  brand, color, mileage, model, price, year \
                                    FROM car_offers \
                                      WHERE country = ? AND date = ?',
                                [row.get('country'), row.get('date')],
                                function(err, result) {
```

```
                                resolve(result.rows[0]);
                            });
                        });
                    }).then(function (value) {
                        accumulator.push(value);
                    });
                });

                prevPromise.then(function () {
                    res.render('search', {
                        brand: req.param('brand'),
                        color: req.param('color'),
                        price_range: req.param('price_range'),
                        results: accumulator
                    });
                });
            }
            else {
                res.render('search', {
                    brand: req.param('brand'),
                    color: req.param('color'),
                    price_range: req.param('price_range')
                });
            }
        });
});
```

Code Listing 120

Querying the search index will return a list of **`country`** fields and **`timeuuid`** fields named **`date`**. With the combination of the two, we can uniquely identify all the data stored in the **`offers`** table. Node.js is by its nature a very asynchronous environment, but sometimes we want things to happen in order or just wait for something to finish before continuing other tasks. Our problem in the previous module is loading the data for every search result from the **`offers`** table. This is relatively hard to do with Node.js itself, so we use the Promise library to do that. For every row in the results, we will create a new promise. The promises are then chained by the **`prevPromise`** variable. Every promise will query the **`offers`** table and fetch the result. When the promise is done fetching the result, it pushes the result to the **`accumulator`** array. The last promise then renders the view with the loaded search results. Every rendering of the search form passes in the posted values so that the user doesn't lose the values after submitting the form. With that, we have finished the logic behind the example. Now we will move to the presentation part.

main.html

This file serves as a container for generating all of the other views. It's the best place to include all of the header, footer, and navigation scripts. To make our example a bit more eye-catching, we are going to use Twitter Bootstrap. It is simple to include and relatively easy to use but makes the examples look much better:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Used Cars</title>

    <link rel="stylesheet"
href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap.min.css">
    <link rel="stylesheet"
href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap-theme.min.css">

  </head>
  <body>
    <div class="container">
        <h1>Used Cars</h1>
        {{{body}}}
    </div>
    <script
src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.1/jquery.min.js">
    </script>
    <script
src="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/js/bootstrap.min.js"></script>
  </body>
</html>
```

Code Listing 121

All of the other views are going to be rendered inside the **{{{body}}}** tag in the previous code listing. Most of the time, the scripts are the longest to load so it's best practice to put the scripts before the closing **body** tag. We'll continue with other views.

home.html

The home screen will show all of the available offers. If there are no offers, a warning message will be shown. From the home screen, users can add an offer, add a user, or start searching.

```
{{#if rows}}
<table class="table table-striped">
    <thead>
    <tr>
        <td>date</td>
        <td>username</td>
        <td>brand</td>
        <td>color</td>
        <td>mileage</td>
        <td>model</td>
        <td>price</td>
        <td>year</td>
    </tr>
```

```
    </thead>
    <tbody>
    {{#each rows}}
        <tr>
            <td>{{formatDate date}}</td>
            <td>{{username}}</td>
            <td>{{brand}}</td>
            <td>{{color}}</td>
            <td>{{mileage}}</td>
            <td>{{model}}</td>
            <td>{{price}}</td>
            <td>{{year}}</td>
        </tr>
    {{/each}}
    </tbody>
</table>
{{else}}
<div class="alert alert-warning" role="alert">
    <strong>Warning!</strong>
    There are no car offers at the moment
</div>
{{/if}}
<p>
    <a href="./show/addoffer" class="btn btn-primary">Add Offer</a>
    <a href="./show/adduser" class="btn btn-info">Add User</a>
    <a href="./show/search" class="btn btn-success">Search</a>
</p>
```

Code Listing 122

add_user.html

The following template is for adding users.

```
<p class="text-center bg-info">Add User</p>
{{#if errorOccured}}
<div class="alert alert-danger" role="alert">
    <strong>Warning!</strong> An error occured.
</div>
{{/if}}
<form class="form-horizontal" role="form" action="/adduser" method="post">
  <div class="form-group">
    <label class="col-sm-2 control-label">username</label>
    <div class="col-sm-10">
      <input type="text" class="form-control" name="username"
        value="{{username}}">
    </div>
  </div>
  <div class="form-group">
    <label class="col-sm-2 control-label">first_name</label>
    <div class="col-sm-10">
```

```
      <input type="text" class="form-control" name="first_name"
        value="{{first_name}}">
    </div>
  </div>
  <div class="form-group">
    <label class="col-sm-2 control-label">last_name</label>
    <div class="col-sm-10">
      <input type="text" class="form-control" name="last_name"
        value="{{last_name}}">
    </div>
  </div>
  <div class="form-group">
    <div class="col-sm-offset-2 col-sm-10">
      <button type="submit" class="btn btn-info">Add</button>
      <a href="/" class="btn btn-default">Cancel</a>
    </div>
  </div>
</form>
```

Code Listing 123

add_offer.html

The following template is for adding offers. This form has the most input fields in our example application.

```
<p class="text-center bg-primary">Add Offer</p>
{{#if errorOccured}}
<div class="alert alert-danger" role="alert">
    <strong>Warning!</strong> An error occured.
</div>
{{/if}}
<form class="form-horizontal" role="form" action="/addoffer" method="post">
  <input type="hidden" name="country" value="USA">
  {{#if username}}
    <input type="hidden" name="username" value="{{username}}">
  {{else}}
  <div class="form-group">
    <label class="col-sm-2 control-label">username</label>
    <div class="col-sm-10">
      <select name="username" class="form-control">
      {{#each users}}
        <option value="{{username}}">
          {{first_name}} {{last_name}}
        </option>
      {{/each}}
      </select>
    </div>
  </div>
  {{/if}}
  <div class="form-group">
```

```
    <label class="col-sm-2 control-label">brand</label>
    <div class="col-sm-10">
      <input type="text" class="form-control" name="brand"
        value="{{brand}}">
    </div>
  </div>
  <div class="form-group">
    <label class="col-sm-2 control-label">color</label>
    <div class="col-sm-10">
      <input type="text" class="form-control" name="color"
        value="{{color}}">
    </div>
  </div>
  <div class="form-group">
    <label class="col-sm-2 control-label">mileage</label>
    <div class="col-sm-10">
      <input type="text" class="form-control" name="mileage"
        value="{{mileage}}">
    </div>
  </div>
  <div class="form-group">
    <label class="col-sm-2 control-label">model</label>
    <div class="col-sm-10">
      <input type="text" class="form-control" name="model"
        value="{{model}}">
    </div>
  </div>
  <div class="form-group">
    <label class="col-sm-2 control-label">price</label>
    <div class="col-sm-10">
      <input type="text" class="form-control" name="price"
        value="{{price}}">
    </div>
  </div>
  <div class="form-group">
    <label class="col-sm-2 control-label">year</label>
    <div class="col-sm-10">
      <input type="text" class="form-control" name="year"
        value="{{year}}">
    </div>
  </div>
  <div class="form-group">
    <div class="col-sm-offset-2 col-sm-10">
      <button type="submit" class="btn btn-primary">Add</button>
      <a href="/" class="btn btn-default">Cancel</a>
    </div>
  </div>
</form>
```

Code Listing 124

search.html

This is the last template in our example. It uses custom directive for string comparison and the view logic for displaying the search results.

```
<p class="text-center bg-success">Search Cars</p>

<form class="form-horizontal" role="form" action="/search" method="post">
  <div class="form-group">
    <label class="col-sm-2 control-label">brand</label>
    <div class="col-sm-10">
      <input type="text" class="form-control" name="brand"
        value="{{brand}}">
    </div>
  </div>
  <div class="form-group">
    <label class="col-sm-2 control-label">color</label>
    <div class="col-sm-10">
      <input type="text" class="form-control" name="color"
        value="{{color}}">
    </div>
  </div>
  <div class="form-group">
    <label class="col-sm-2 control-label">price_range</label>
    <div class="col-sm-10">
      <select name="price_range" class="form-control">
        <option value=""></option>
        <option value="1" {{#if_eq price_range "1"}}selected{{/if_eq}}>
            &lt; 1000</option>
        <option value="2" {{#if_eq price_range "2"}}selected{{/if_eq}}>
            &gt;= 1000 &lt; 3000</option>
        <option value="3" {{#if_eq price_range "3"}}selected{{/if_eq}}>
            &gt;= 3000 &lt; 5000</option>
        <option value="4" {{#if_eq price_range "4"}}selected{{/if_eq}}>
            &lt;= 5000 &lt; 10000</option>
        <option value="5" {{#if_eq price_range "5"}}selected{{/if_eq}}>
            &gt; 10000</option>
      </select>
    </div>
  </div>
  <div class="form-group">
    <div class="col-sm-offset-2 col-sm-10">
      <button type="submit" class="btn btn-success">Search</button>
      <a href="/" class="btn btn-default">Back</a>
    </div>
  </div>
</form>

{{#if results}}
<table class="table table-striped">
    <thead>
    <tr>
        <td>date</td>
```

```
        <td>username</td>
        <td>brand</td>
        <td>color</td>
        <td>mileage</td>
        <td>model</td>
        <td>price</td>
        <td>year</td>
    </tr>
    </thead>
    <tbody>
    {{#each results}}
        <tr>
            <td>{{formatDate date}}</td>
            <td>{{username}}</td>
            <td>{{brand}}</td>
            <td>{{color}}</td>
            <td>{{mileage}}</td>
            <td>{{model}}</td>
            <td>{{price}}</td>
            <td>{{year}}</td>
        </tr>
    {{/each}}
    </tbody>
</table>
{{else}}
<div class="alert alert-success" role="success">
    There are no search results
</div>
{{/if}}
```

Code Listing 125

Used Car Market in Action

The following screenshots will give you an idea what the application should look like in the end.

Used Cars

date	username	brand	color	mileage	model	price	year
2014-08-09 20:54:44	jqpublic	Audi	Black	99000	A4	12000	2008
2014-08-09 20:54:16	jqpublic	Volkswagen	Black	100000	Golf	2000	2008
2014-08-09 20:53:46	jsmith	Audi	White	150000	A6	3000	2005
2014-08-09 20:53:12	jsmith	BMW	Black	40000	120i	6000	2010
2014-08-09 20:52:22	jsmith	BMW	Red	98000	118d	5000	2007
2014-08-09 20:51:44	jsmith	Ford	White	200000	Orion	800	2006
2014-08-09 20:49:44	jdoe	Audi	Orange	10000	A3	7000	2011

Add Offer Add User Search

Figure 45: Demo application home screen

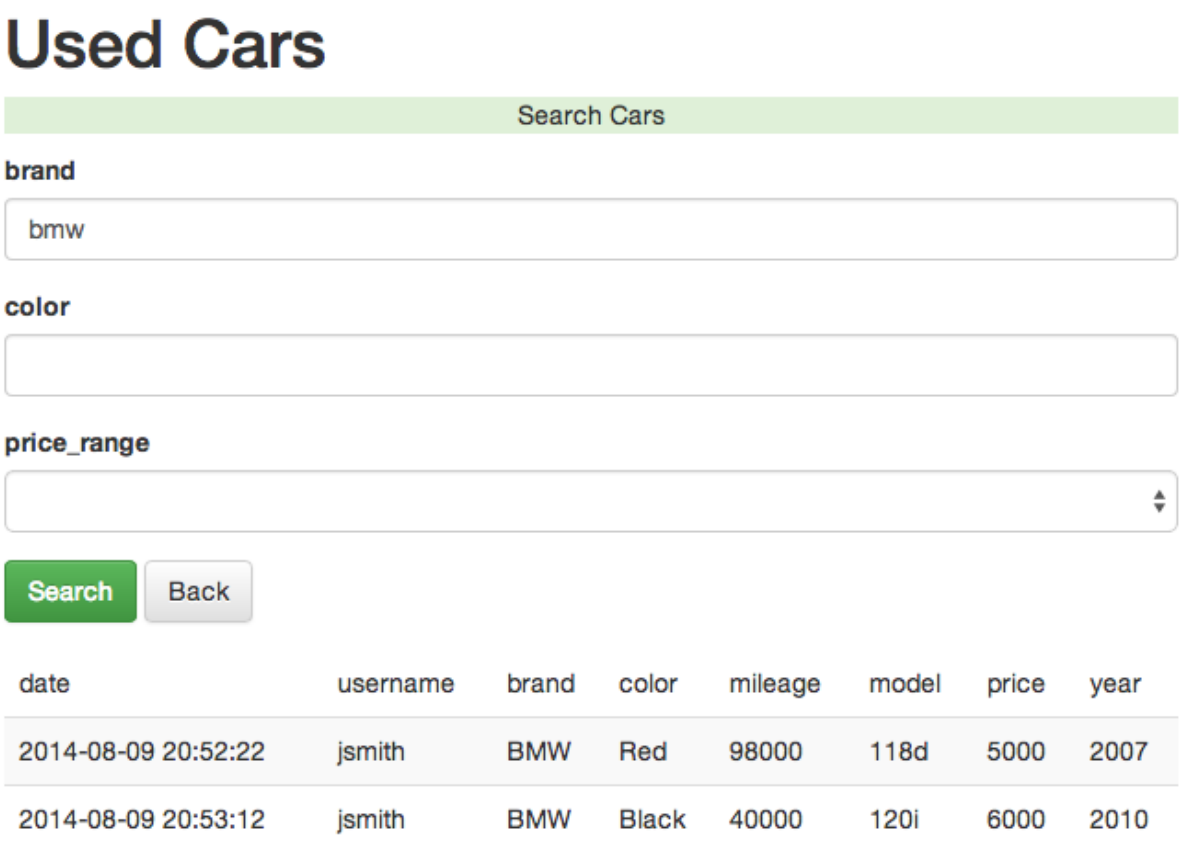

Figure 46: Demo application search screen

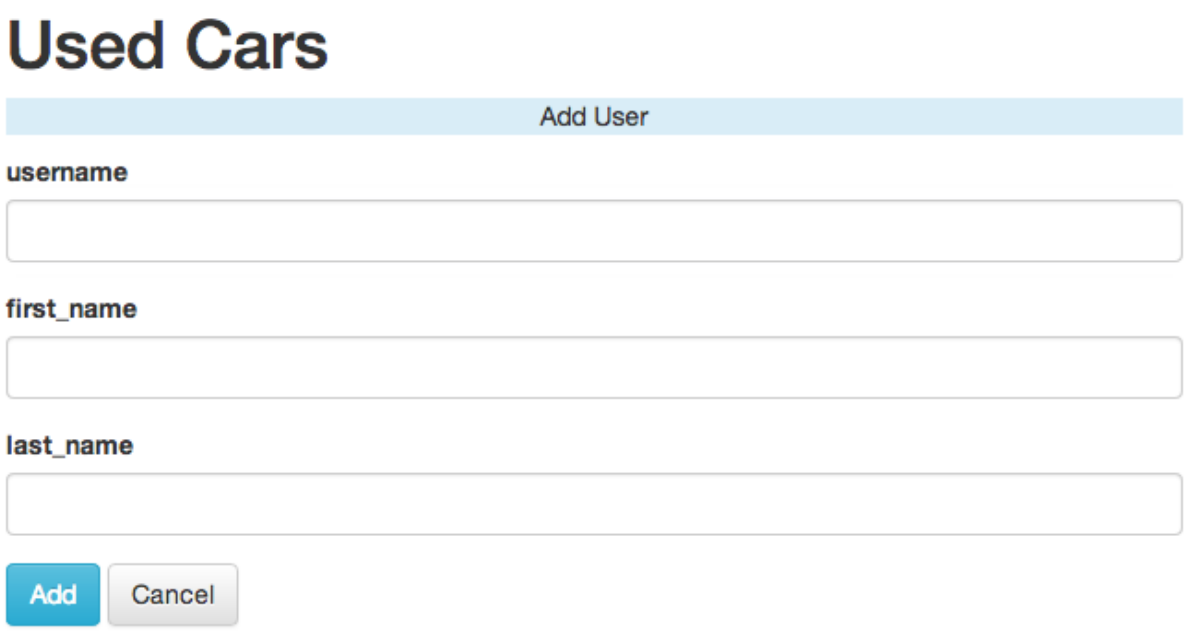

Figure 47: Demo application Add User form

Cassandra with Java

Java is a very mature, stable, and battle-tested technology. In fact, Cassandra was written in Java. In the beginning, most developers using Cassandra used Java on a daily basis. The situation has changed a lot from the early days, but still, Java remains a very important language when it comes to Cassandra's platform.

The techniques that we covered in the Node.js section apply to Java, too. Java has its own set of frameworks; some of them are similar to Express, but we won't go into them. To use Cassandra from Java, use your preferred IDE and make sure that you add the **`com.datastax.driver`** as a dependency to the project. In this section, we will build a Java client that lists all of the offers from the previously used **`used_cars`** keyspace.

```
package com.cassandra.example;

import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.UUID;

import com.datastax.driver.core.BoundStatement;
import com.datastax.driver.core.Cluster;
import com.datastax.driver.core.PreparedStatement;
import com.datastax.driver.core.ResultSet;
import com.datastax.driver.core.Row;
import com.datastax.driver.core.Session;

public class CassandraExample {

    public static void main(String[] args) {
        String host = "127.0.0.1";

        Cluster cluster = Cluster.builder().addContactPoint(host).build();

        Session session = cluster.connect("used_cars");

        PreparedStatement statement = session
                .prepare("SELECT * FROM car_offers WHERE country = ?");

        BoundStatement boundStatement = statement.bind("USA");

        ResultSet results = session.execute(boundStatement);

        System.out.println(String.format(
                "%-23s\t%-15s\t%-7s\t%-7s\t%-7s\t%-7s\t%-10s\t%-7s", "date",
                "brand", "color", "mileage", "model", "price", "username",
                "year"));

        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss.S");

        for (Row row : results.all()) {
            UUID dateUUID = row.getUUID("date");
            Date date = new Date(getTimeFromUUID(dateUUID));
```

```
            String brand = row.getString("brand");
            String color = row.getString("color");
            Integer mileage = row.getInt("mileage");
            String model = row.getString("model");
            Float price = row.getFloat("price");
            String username = row.getString("username");
            Integer year = row.getInt("year");

            System.out.println(String.format(
                    "%-23s\t%-15s\t%-7s\t%-7s\t%-7s\t%-7s\t%-10s\t%-7s",
                    sdf.format(date), brand, color, mileage, model, price,
                    username, year));
        }

        session.close();
        cluster.close();
    }

    static final long INTERVALS_SINCE_UUID_EPOCH_100NS = 0x01b21dd213814000L;

    public static long getTimeFromUUID(UUID uuid) {
        return (uuid.timestamp() - INTERVALS_SINCE_UUID_EPOCH_100NS) / 10000;
    }
}
```

Code Listing 126

The previously listed module has a dependency on the DataStax Cassandra driver. If you are going to try the example without any dependency management system, be aware that the driver needs the **sl4j-api, sl4j-simple**, **com.google.gson**, **guava**, **jboss netty**, and **metrics** modules to run.

The example is relatively easy to follow. The only tricky part is to extract the time stamp from the **timeUUID** column. The most common mistake is taking the **timestamp** and pasting it to a **Date** constructor. Before taking the time stamp, we need to adjust it to match to the commonly used milliseconds since midnight of January 1st 1970. UUIDs are measured from midnight October 15th 1582 in 100-nanosecond intervals. Conversion between the two epochs is done in the previously listed **getTimeFromUUID** function.

Cassandra with C#

C# is also an extremely popular technology. Before doing any work with the driver, we need to import it with the package manager into our project. It's sufficient to type **`Install-Package CassandraCSharpDriver`** in the standard Package Manager Console window. If you are running .NET on platforms other than Windows, add a package to the project by searching for "DataStax C# Driver for Apache Cassandra". The following example for C# is going to be identical to the Java example and it will display the output in the console. As with the Java example, the most complex part of the code is extracting the time stamp from the **`GUID`** type. The rest is pretty similar in any of the previous languages and is more Cassandra-specific than application or computer language-specific. The C# example is shown in the following listing.

```
using Cassandra;
using System;

namespace CassandraExample
{
    class ExampleApp
    {
        private Cluster cluster;
        private ISession session;

        public void Connect (String node)
        {
            cluster = Cluster.Builder ().AddContactPoint (node).Build ();
            session = cluster.Connect ("used_cars");
        }

        public void PrinResults ()
        {
            RowSet results = session.Execute ("SELECT * FROM car_offers");

            Console.WriteLine (String.Format (
"{0, -23}\t{1, -15}\t{2, -7}\t{3, -7}\t{4, -7}\t{5, -7}\t{6, -10}\t{7, -7}",
 "date", "brand", "color", "mileage", "model", "price", "username", "year"));

            foreach (Row row in results.GetRows()) {
                Guid date = new Guid (row ["date"].ToString ());
                Console.WriteLine (String.Format (
"{0, -23}\t{1, -15}\t{2, -7}\t{3, -7}\t{4, -7}\t{5, -7}\t{6, -10}\t{7, -7}",
GetDateTime (date), row ["brand"], row ["color"], row ["mileage"],
row ["model"], row ["price"], row ["username"], row ["year"]));
            }
        }

        public void Close ()
        {
            cluster.Shutdown ();
        }

        static void Main (string[] args)
        {
```

```
            ExampleApp client = new ExampleApp ();
            client.Connect ("127.0.0.1");
            client.PrinResults ();
            client.Close ();
        }

        private const int VersionByte = 7;
        private const int VersionByteMask = 0x0f;
        private const int VersionByteShift = 4;
        private const byte TimestampByte = 0;
        private static readonly DateTimeOffset GregorianCalendarStart =
            new DateTimeOffset (1582, 10, 15, 0, 0, 0, TimeSpan.Zero);

        private static DateTimeOffset GetDateTimeOffset (Guid guid)
        {
            byte[] bytes = guid.ToByteArray ();

            bytes [VersionByte] &= (byte)VersionByteMask;
            bytes [VersionByte] |= (byte)((byte)0x01 >> VersionByteShift);

            byte[] timestampBytes = new byte[8];
            Array.Copy (bytes, TimestampByte, timestampBytes, 0, 8);

            long timestamp = BitConverter.ToInt64 (timestampBytes, 0);
            long ticks = timestamp + GregorianCalendarStart.Ticks;

            return new DateTimeOffset (ticks, TimeSpan.Zero);
        }

        private static DateTime GetDateTime (Guid guid)
        {
            return GetDateTimeOffset (guid).DateTime;
        }
    }
}
```

Code Listing 127

Summary

In this chapter, we have shown how to use Cassandra with today's most popular development environments. Most of the chapter was dedicated to Node.js, where we saw how to build a complete web application that relies on Cassandra underneath. The completed application focuses on Cassandra as an underlying data platform and not so much on the application using it. Repeating the same application for Java and C# wouldn't provide additional insight to Cassandra, so in these final two sections we have looked only at how to connect to a cluster, retrieve the offers, and print them to the console.

Conclusion

Although Cassandra was a bit of a problem child in its infancy, it has become the NoSQL star in recent years. More and more companies are starting to use it every day. Even Facebook, which stopped using Cassandra after open-sourcing it in 2008, started to use it again to power their Instagram acquisition. Cassandra outperforms most SQL and NoSQL solutions when it comes to data throughput, and it is becoming more and more popular every day.

Cassandra is also popular because it is easy to maintain when compared to other solutions. Installing Cassandra usually takes just a couple of minutes. There are even videos available that demonstrate installing a whole Cassandra cluster in less than two minutes. Doing complex operations on the database doesn't require the staff to be awake all night; some say they can add new nodes and take the old one out in the time it takes to go to lunch. There are more and more big data jobs out there, and one of the most sought-after skills is definitely Apache Cassandra.

This book has tried to introduce you to Cassandra in about one hundred pages. The first part of the book covered the basic theoretical concepts on which Cassandra relies. Although this book is more practice-oriented, it makes very little sense to jump into Cassandra without understanding what's under the hood. Cassandra is in a class of its own when it comes to internal mechanisms, and as you progress it becomes very important to understand them.

After exploring the theory, the book demonstrated how to make a Cassandra setup sufficient to get you going. It then focused on data modeling from Cassandra's perspective and pointed out many guiding principles. A primary one is that the disk is cheap. This is often repeated because most developers are confused by the amount of writes Cassandra applications usually do and the use of denormalized data, especially if they are coming from the relational world where almost everything is normalized.

The final chapter of the book demonstrated how to use Cassandra with popular application technologies such as C#, Node.js, and Java.

I hope this book has provided you a solid foundation for getting started with Apache Cassandra, and I hope you had as much fun reading it as I did writing it.