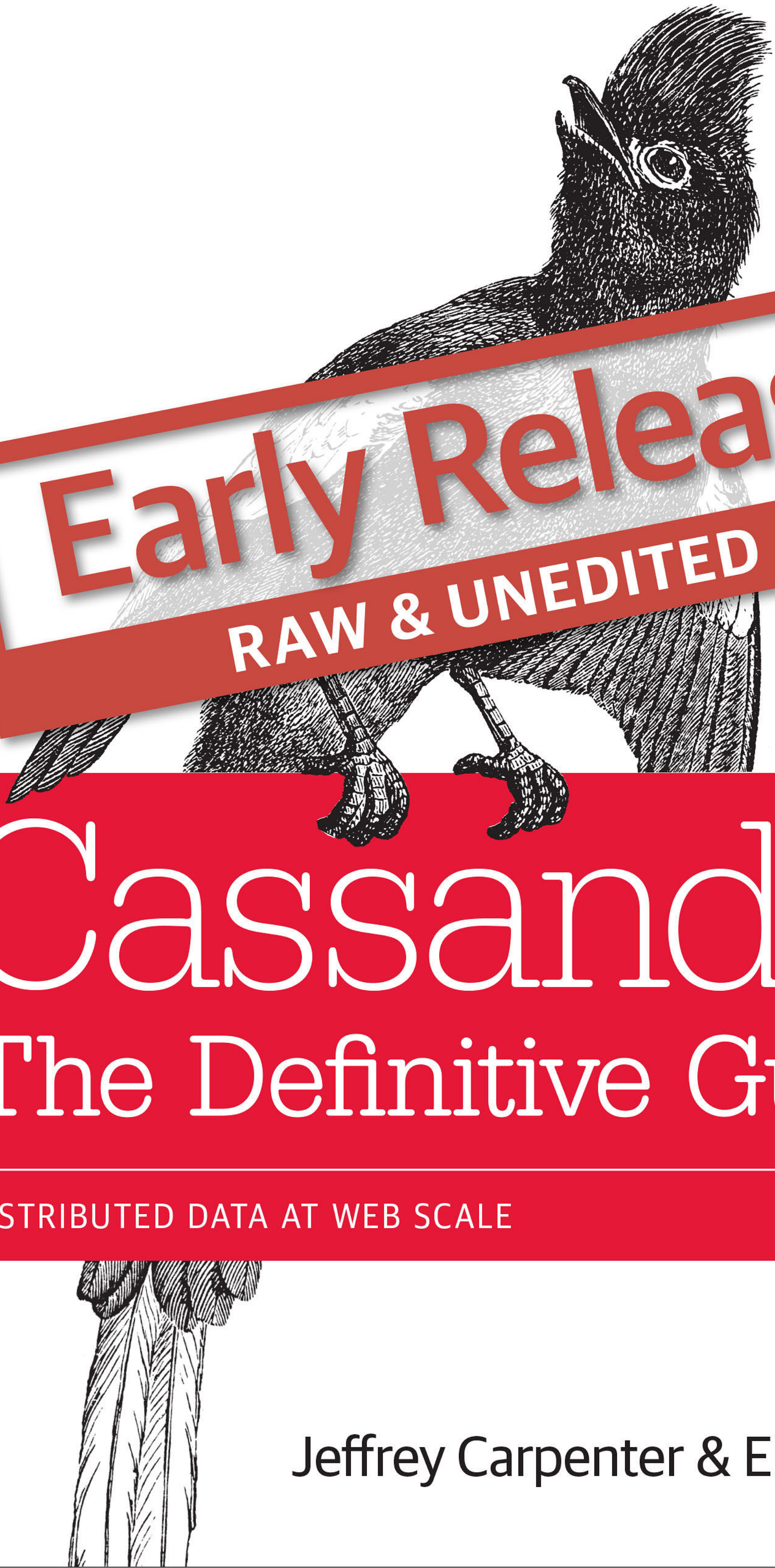
O'REILLY®

2nd Edition



Early Release

RAW & UNEDITED

Cassandra
The Definitive Guide

DISTRIBUTED DATA AT WEB SCALE

Jeffrey Carpenter & Eben Hewitt

SECOND EDITION

Cassandra: The Definitive Guide

Author Name

Beijing · Boston · Farnham · Sebastopol · Tokyo O'REILLY®

Cassandra: The Definitive Guide
by Eben Hewitt

Printed in the United States of America.

Published by O'Reilly Media, Inc. , 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com .

Editor: Mike Loukides
Production Editor: Holly Bauer
Copyeditor: Genevieve d'Entremont
Proofreader: FILL IN PROOFREADER
Indexer: Ellen Troutman Zaig
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

November 2010: First Edition

Revision History for the First Edition
2015-12-17: First Early Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781449390419* for release details.

978-1-449-39041-9

[FILL IN]

Table of Contents

CHAPTER 1

Beyond Relational Databases

If at first the idea is not absurd, then there is no hope for it.
—Albert Einstein

Welcome to *Cassandra: The Definitive Guide*. The aim of this book is to help developers and database administrators understand this important database technology, explore how it compares to traditional relational database management systems, and help you put it to work in your own environment.

What's Wrong with Relational Databases?

If I had asked people what they wanted, they would have said faster horses.
—Henry Ford

I ask you to consider a certain model for data, invented by a small team at a company with thousands of employees. It was accessible over a TCP/IP interface and was available from a variety of languages, including Java and web services. This model was difficult at first for all but the most advanced computer scientists to understand, until broader adoption helped make the concepts clearer. Using the database built around this model required learning new terms and thinking about data storage in a different way. But as products sprang up around it, more businesses and government agencies put it to use, in no small part because it was fast—capable of processing thousands of operations a second. The revenue it generated was tremendous.

And then a new model came along.

The new model was threatening, chiefly for two reasons. First, the new model was very different from the old model, which it pointedly controverted. It was threatening because it can be hard to understand something different and new. Ensuing debates can help entrench people stubbornly further in their views—views that might have been largely inherited from the climate in which they learned their craft and the circumstances in which they work. Second, and perhaps more importantly, as a barrier, the new model was threatening because businesses had made considerable investments in the old model and were making lots of money with it. Changing course seemed ridiculous, even impossible.

Of course I'm talking about the Information Management System (IMS) hierarchical database, invented in 1966 at IBM.

IMS was built for use in the Saturn V moon rocket. Its architect was Vern Watts, who dedicated his career to it. Many of us are familiar with IBM's database DB2. IBM's wildly popular DB2 database gets its name as the successor to DB1—the product built around the hierarchical data model IMS. IMS was released in 1968, and subsequently enjoyed success in Customer Information Control System (CICS) and other applications. It is still used today.

But in the years following the invention of IMS, the new model, the disruptive model, the threatening model, was the relational database.

In his 1970 paper "A Relational Model of Data for Large Shared Data Banks," Dr. Edgar F. Codd, also at IBM, advanced his theory of the relational model for data while working at IBM's San Jose research laboratory. This paper, still available at *http://www.seas.upenn.edu/~zives/03f/cis550/codd.pdf*, became the foundational work for relational database management systems.

Codd's work was antithetical to the hierarchical structure of IMS. Understanding and working with a relational database required learning new terms that must have sounded very strange indeed to users of IMS such as relations, tuples, and normal form. It presented certain key advantages over its predecessor, such as the ability to express complex relationships between multiple entities, well beyond what could be represented by hierarchical databases..

While these ideas and their application have evolved in four decades, the relational database still is clearly one of the most successful software applications in history. It's used in the form of Microsoft Access in sole proprietorships, and in giant multinational corporations with clusters of hundreds of finely tuned instances representing multi-terabyte data warehouses. Relational databases store invoices, customer records, product catalogues, accounting ledgers, user authentication schemes—the very world, it might appear. There is no question that the relational database is a key facet of the modern technology and business landscape, and one that will be with us in its various forms for many years to come, as will IMS in its various forms. The relational model presented an alternative to IMS, and each has its uses.

So the short answer to the question, "What's wrong with relational databases?" is "Nothing."

There is, however, a rather longer answer, which says that every once in a while an idea is born that ostensibly changes things, and engenders a revolution of sorts. And yet, in another way, such revolutions, viewed structurally, are simply history's business as usual. IMS, RDBMS, NoSQL. The horse, the car, the plane. They each build on prior art, they each attempt to solve certain problems, and so they're each good at certain things—and less good at others. They each coexist, even now.

So let's examine for a moment why, at this point, we might consider an alternative to the relational database, just as Codd himself four decades ago looked at the Information Management System and thought that maybe it wasn't the only legitimate way of organizing information and solving data problems, and that maybe, for certain problems, it might prove fruitful to consider an alternative.

We encounter scalability problems when our relational applications become successful and usage goes up. Joins are inherent in any relatively normalized relational database of even modest size, and joins can be slow. The way that databases gain consistency is typically through the use of transactions, which require locking some portion of the database so it's not available to other clients. This can become untenable under very heavy loads, as the locks mean that competing users start queuing up, waiting for their turn to read or write the data.

We typically address these problems in one or more of the following ways, sometimes in this order:

- Throw hardware at the problem by adding more memory, adding faster processors, and upgrading disks. This is known as *vertical scaling*. This can relieve you for a time.
- When the problems arise again, the answer appears to be similar: now that one box is maxed out, you add hardware in the form of additional boxes in a database cluster. Now you have the problem of data replication and consistency during regular usage and in failover scenarios. You didn't have that problem before.
- Now we need to update the configuration of the database management system. This might mean optimizing the channels the database uses to write to the underlying filesystem. We turn off logging or journaling, which frequently is not a desirable (or, depending on your situation, legal) option.
- Having put what attention we could into the database system, we turn to our application. We try to improve our indexes. We optimize the queries. But presumably at this scale we weren't wholly ignorant of index and query optimization, and already had them in pretty good shape. So this becomes a painful process of picking through the data access code to find any opportunities for fine-tuning. This might include reducing or reorganizing joins, throwing out resource-intensive features such as XML processing within a stored procedure, and so forth. Of course, presumably we were doing that XML processing for a

reason, so if we have to do it somewhere, we move that problem to the application layer, hoping to solve it there and crossing our fingers that we don't break something else in the meantime.

- We employ a caching layer. For larger systems, this might include distributed caches such as memcached, Redis, Riak, EHCache, or other related products. Now we have a consistency problem between updates in the cache and updates in the database, which is exacerbated over a cluster.
- We turn our attention to the database again and decide that, now that the application is built and we understand the primary query paths, we can duplicate some of the data to make it look more like the queries that access it. This process, called denormalization, is antithetical to the five normal forms that characterize the relational model, and violate Codd's 12 Rules for relational data. We remind ourselves that we live in this world, and not in some theoretical cloud, and then undertake to do what we must to make the application start responding at acceptable levels again, even if it's no longer "pure."

Codd's Twelve Rules

Codd provided a list of "twelve" rules (there are actually 13, numbered 0 to 12) formalizing his definition of the relational model as a response to the divergence of commercial databases from his original concepts. Codd introduced his rules in a pair of articles in CompuWorld magazine in October 1985, and formalized them in the second edition of his book The Relational Model for Database Management, which is now out of print.

I imagine that this sounds familiar to you. At web scale, engineers legitimately whether this situation isn't similar to Henry Ford's assertion that at a certain point, it's not simply a faster horse that you want. And they've done some impressive, interesting work.

We must therefore begin here in recognition that the relational model is simply a model. That is, it's intended to be a useful way of looking at the world, applicable to certain problems. It does not purport to be exhaustive, closing the case on all other ways of representing data, never again to be examined, leaving no room for alternatives. If we take the long view of history, Dr. Codd's model was a rather disruptive one in its time. It was new, with strange new vocabulary and terms such as "tuples"—familiar words used in a new and different manner. The relational model was held up to suspicion, and doubtless suffered its vehement detractors. It encountered opposition even in the form of Dr. Codd's own employer, IBM, which had a very lucrative product set around IMS and didn't need a young upstart cutting into its pie.

But the relational model now arguably enjoys the best seat in the house within the data world. SQL is widely supported and well understood. It is taught in introductory university courses. There are open source databases that come installed and ready to use with a $4.95 monthly web hosting plan. Cloud-based Platform as a Service (PaaS) providers such as Amazon Web Services, Google Cloud Platform, Rackspace, and Microsoft Azure provide relational database access as a service, including automated monitoring and maintenance features. Often the database we end up using is dictated to us by architectural standards within our organization. Even absent such standards, it's prudent to learn whatever your organization already has for a database platform. Our colleagues in development and infrastructure have considerable hard-won knowledge.

If by nothing more than osmosis—or inertia—we have learned over the years that a relational database is a one-size-fits-all solution.

So perhaps a better question is not, "What's wrong with relational databases?" but rather, "What problem do you have?"

That is, you want to ensure that your solution matches the problem that you have. There are certain problems that relational databases solve very well. But the explosion of the Web, and in particular social networks, means a corresponding explosion in the sheer volume of data we must deal with. When Tim Berners-Lee first worked on the Web in the early 1990s, it was for the purpose of exchanging scientific documents between PhDs at a physics laboratory. Now, of course, the Web has become so ubiquitous that it's used by everyone, from those same scientists to legions of five-year-olds exchanging emoticons about kittens. That means in part that it must support enormous volumes of data; the fact that it does stands as a monument to the ingenious architecture of the Web.

But some of this infrastructure is starting to bend under the weight.

A Quick Review of Relational Databases

Though you are likely familiar with them, let's briefly turn our attention to some of the foundational concepts in relational databases. This will give us a basis on which to consider more recent advances in thought around the trade-offs inherent in distributed data systems, especially very large distributed data systems, such as those that are required at web scale.

RDBMS: The Awesome and the Not-So-Much

There are many reasons that the relational database has become so overwhelmingly popular over the last four decades. An important one is the Structured Query Language (SQL), which is feature-rich and uses a simple, declarative syntax. SQL was first officially adopted as an ANSI standard in 1986; since that time it's gone through several revisions and has also been extended with vendor proprietary syntax such as Microsoft's T-SQL and Oracle's PL/SQL to provide additional implementation-specific features.

SQL is powerful for a variety of reasons. It allows the user to represent complex relationships with the data, using statements that form the Data Manipulation Language (DML) to insert, select, update, delete, truncate, and merge data. You can perform a rich variety of operations using functions based on relational algebra to find a maximum or minimum value in a set, for example, or to filter and order results. SQL statements support grouping aggregate values and executing summary functions. SQL provides a means of directly creating, altering, and dropping schema structures at runtime using Data Definition Language (DDL). SQL also allows you to grant and revoke rights for users and groups of users using the same syntax.

SQL is easy to use. The basic syntax can be learned quickly, and conceptually SQL and RDBMS offer a low barrier to entry. Junior developers can become proficient readily, and as is often the case in an industry beset by rapid changes, tight deadlines, and exploding budgets, ease of use can be very important. And it's not just the syntax that's easy to use; there are many robust tools that include intuitive graphical interfaces for viewing and working with your database.

In part because it's a standard, SQL allows you to easily integrate your RDBMS with a wide variety of systems. All you need is a driver for your application language, and you're off to the races in a very portable way. If you decide to change your application implementation language (or your RDBMS vendor), you can often do that painlessly, assuming you haven't backed yourself into a corner using lots of proprietary extensions.

Transactions, ACID-ity, and two-phase commit

In addition to the features mentioned already, RDBMS and SQL also support *transactions*. A key feature of transactions is that they execute virtually at first, allowing the programmer to undo (using ROLLBACK) any changes that may have gone awry during execution; if all has gone well, the transaction can be reliably committed. As Jim Gray puts it, a transaction is "a transformation of state" that has the ACID properties (see *http://research.microsoft.com/en-us/um/people/gray/papers/theTransactionConcept.pdf*).

ACID is an acronym for Atomic, Consistent, Isolated, Durable, which are the gauges we can use to assess that a transaction has executed properly and that it was successful:

Atomic
: Atomic means "all or nothing"; that is, when a statement is executed, every update within the transaction must succeed in order to be called successful.

There is no partial failure where one update was successful and another related update failed. The common example here is with monetary transfers at an ATM: the transfer requires subtracting money from one account and adding it to another account. This operation cannot be subdivided; they must both succeed.

Consistent
: Consistent means that data moves from one correct state to another correct state, with no possibility that readers could view different values that don't make sense together. For example, if a transaction attempts to delete a Customer and her Order history, it cannot leave Order rows that reference the deleted customer's primary key; this is an inconsistent state that would cause errors if someone tried to read those Order records.

Isolated
: Isolated means that transactions executing concurrently will not become entangled with each other; they each execute in their own space. That is, if two different transactions attempt to modify the same data at the same time, then one of them will have to wait for the other to complete.

Durable
: Once a transaction has succeeded, the changes will not be lost. This doesn't imply another transaction won't later modify the same data; it just means that writers can be confident that the changes are available for the next transaction to work with as necessary.

The debate about support for transactions comes up very quickly as a sore spot in conversations around non-relational data stores, so let's take a moment to revisit what this really means. On the surface, ACID properties seem so obviously desirable as to not even merit conversation. Presumably no one who runs a database would suggest that data updates don't have to endure for some length of time; that's the very point of making updates—that they're there for others to read. However, a more subtle examination might lead us to want to find a way to tune these properties a bit and control them slightly. There is, as they say, no free lunch on the Internet, and once we see how we're paying for our transactions, we may start to wonder whether there's an alternative.

Transactions become difficult under heavy load. When you first attempt to horizontally scale a relational database, making it distributed, you must now account for *distributed transactions*, where the transaction isn't simply operating inside a single table or a single database, but is spread across multiple systems. In order to continue to honor the ACID properties of transactions, you now need a transaction manager to orchestrate across the multiple nodes.

In order to account for successful completion across multiple hosts, the idea of a two-phase commit (sometimes referred to as "2PC") is introduced. But then, because two-phase commit locks all associate resources, it is useful only for operations that can complete very quickly. Although it may often be the case that your distributed operations can complete in sub-second time, it is certainly not always the case. Some use cases require coordination between multiple hosts that you may not control yourself. Operations coordinating several different but related activities can take hours to update.

Two-phase commit *blocks*; that is, clients ("competing consumers") must wait for a prior transaction to finish before they can access the blocked resource. The protocol will wait for a node to respond, even if it has died. It's possible to avoid waiting forever in this event, because a timeout can be set that allows the transaction coordinator node to decide that the node isn't going to respond and that it should abort the transaction. However, an infinite loop is still possible with 2PC; that's because a node can send a message to the transaction coordinator node agreeing that it's OK for the coordinator to commit the entire transaction. The node will then wait for the coordinator to send a commit response (or a rollback response if, say, a different node can't commit); if the coordinator is down in this scenario, that node conceivably will wait forever.

So in order to account for these shortcomings in two-phase commit of distributed transactions, the database world turned to the idea of *compensation*. Compensation, often used in web services, means in simple terms that the operation is immediately committed, and then in the event that some error is reported, a new operation is invoked to restore proper state.

There are a few basic, well-known patterns for compensatory action that architects frequently have to consider as an alternative to two-phase commit. These include writing off the transaction if it fails, deciding to discard erroneous transactions and reconciling later. Another alternative is to retry failed operations later on notification. In a reservation system or a stock sales ticker, these are not likely to meet your requirements. For other kinds of applications, such as billing or ticketing applications, this can be acceptable.

The Problem with Two-Phase Commit

Gregor Hohpe, a Google architect, wrote a wonderful and often-cited blog entry called "Starbucks Does Not Use Two-Phase Commit." It shows in real-world terms how difficult it is to scale two-phase commit and highlights some of the alternatives that are mentioned here. Check it out at *http://www.eaipatterns.com/ramblings/18_starbucks.html*. It's an easy, fun, and enlightening read.

The problems that 2PC introduces for application developers include loss of availability and higher latency during partial failures. Neither of these is desirable. So once you've had the good fortune of being successful enough to necessitate scaling your database past a single machine, you now have to figure out how to handle transactions across multiple machines and still make the ACID properties apply. Whether you have 10 or 100 or 1,000 database machines, atomicity is still required in transactions as if you were working on a single node. But it's now a much, much bigger pill to swallow.

Schema

One often-lauded feature of relational database systems is the rich schemas they afford. You can represent your domain objects in a relational model. A whole industry has sprung up around (expensive) tools such as the CA ERWin Data Modeler to support this effort. In order to create a properly normalized schema, however, you are forced to create tables that don't exist as business objects in your domain. For example, a schema for a university database might require a Student table and a Course table. But because of the "many-to-many" relationship here (one student can take many courses at the same time, and one course has many students at the same time), you have to create a join table. This pollutes a pristine data model, where we'd prefer to just have students and courses. It also forces us to create more complex SQL statements to join these tables together. The join statements, in turn, can be slow.

Again, in a system of modest size, this isn't much of a problem. But complex queries and multiple joins can become burdensomely slow once you have a large number of rows in many tables to handle.

Finally, not all schemas map well to the relational model. One type of system that has risen in popularity in the last decade is the complex event processing system, which represents state changes in a very fast stream. It's often useful to contextualize events at runtime against other events that might be related in order to infer some conclusion to support business decision making. Although event streams could be represented in terms of a relational database, it is an uncomfortable stretch.

And if you're an application developer, you'll no doubt be familiar with the many object-relational mapping (ORM) frameworks that have sprung up in recent years to help ease the difficulty in mapping application objects to a relational model. Again, for small systems, ORM can be a relief. But it also introduces new problems of its own, such as extended memory requirements, and it often pollutes the application code with increasingly unwieldy mapping code. Here's an example of a Java method using Hibernate to "ease the burden" of having to write the SQL code:

```
@CollectionOfElements
@JoinTable(name="store_description",
  joinColumns = @JoinColumn(name="store_code"))
@MapKey(columns={@Column(name="for_store",length=3)})
@Column(name="description")
private Map<String, String> getMap() {
  return this.map;
}
//... etc.
```

Is it certain that we've done anything but move the problem here? Of course, with some systems, such as those that make extensive use of document exchange, as with services or XML-based applications, there are not always clear mappings to a relational database. This exacerbates the problem.

Sharding and shared-nothing architecture

If you can't split it, you can't scale it.
—Randy Shoup, Distinguished Architect, eBay

Another way to attempt to scale a relational database is to introduce *sharding* to your architecture. This has been used to good effect at large websites such as eBay, which supports billions of SQL queries a day, and in other modern web applications. The idea here is that you split the data so that instead of hosting all of it on a single server or replicating all of the data on all of the servers in a cluster, you divide up portions of the data horizontally and host them each separately.

For example, consider a large customer table in a relational database. The least disruptive thing (for the programming staff, anyway) is to vertically scale by adding CPU, adding memory, and getting faster hard drives, but if you continue to be successful and add more customers, at some point (perhaps into the tens of millions of rows), you'll likely have to start thinking about how you can add more machines. When you do so, do you just copy the data so that all of the machines have it? Or do you instead divide up that single customer table so that each database has only some of the records, with their order preserved? Then, when clients execute queries, they put load only on the machine that has the record they're looking for, with no load on the other machines.

It seems clear that in order to shard, you need to find a good key by which to order your records. For example, you could divide your customer records across 26 machines, one for each letter of the alphabet, with each hosting only the records for customers whose last names start with that particular letter. It's likely this is not a good strategy, however—there probably aren't many last names that begin with "Q" or "Z," so those machines will sit idle while the "J," "M," and "S" machines spike. You could shard according to something numeric, like phone number, "member since" date, or the name of the customer's state. It all depends on how your specific data is likely to be distributed.

There are three basic strategies for determining shard structure:

Feature-based shard or functional segmentation

This is the approach taken by Randy Shoup, Distinguished Architect at eBay, who in 2006 helped bring their architecture into maturity to support many billions of queries per day. Using this strategy, the data is split not by dividing records in a single table (as in the customer example discussed earlier), but rather by splitting into separate databases the features that don't overlap with each other very much. For example, at eBay, the users are in one shard, and the items for sale are in another. At Flixster, movie ratings are in one shard and comments are in another. This approach depends on understanding your domain so that you can segment data cleanly.

Key-based sharding

In this approach, you find a key in your data that will evenly distribute it across shards. So instead of simply storing one letter of the alphabet for each server as in the (naive and improper) earlier example, you use a one-way hash on a key data element and distribute data across machines according to the hash. It is common in this strategy to find time-based or numeric keys to hash on.

Lookup table

In this approach, one of the nodes in the cluster acts as a "yellow pages" directory and looks up which node has the data you're trying to access. This has two obvious disadvantages. The first is that you'll take a performance hit every time you have to go through the lookup table as an additional hop. The second is that the lookup table not only becomes a bottleneck, but a single point of failure.

Sharding can minimize contention depending on your strategy and allows you not just to scale horizontally, but then to scale more precisely, as you can add power to the particular shards that need it.

Sharding could be termed a kind of "shared-nothing" architecture that's specific to databases. A *shared-nothing* architecture is one in which there is no centralized (shared) state, but each node in a distributed system is independent, so there is no client contention for shared resources. The term was first coined by Michael Stonebraker at University of California at Berkeley in his 1986 paper "The Case for Shared Nothing."

Shared Nothing was more recently popularized by Google, which has written systems such as its Bigtable database and its MapReduce implementation that do not share state, and are therefore capable of near-infinite scaling. The Cassandra database is a shared-nothing architecture, as it has no central controller and no notion of master/slave; all of its nodes are the same.

More on Shared Nothing Architecture

You can read the 1986 paper "The Case for Shared Nothing" online at *http://db.cs.berkeley.edu/papers/hpts85-nothing.pdf*. It's only a few pages. If you take a look, you'll see that many of the features of shared-nothing distributed data architecture, such as ease of high availability and the ability to scale to a very large number of machines, are the very things that Cassandra excels at.

MongoDB also provides auto-sharding capabilities to manage failover and node balancing. That many nonrelational databases offer this automatically and out of the box is very handy; creating and maintaining custom data shards by hand is a wicked proposition. It's good to understand sharding in terms of data architecture in general, but especially in terms of Cassandra more specifically, as it can take an approach similar to key-based sharding to distribute data across nodes, but does so automatically.

Web Scale

In summary, relational databases are very good at solving certain data storage problems, but because of their focus, they also can create problems of their own when it's time to scale. Then, you often need to find a way to get rid of your joins, which means denormalizing the data, which means maintaining multiple copies of data and seriously disrupting your design, both in the database and in your application. Further, you almost certainly need to find a way around distributed transactions, which will quickly become a bottleneck. These compensatory actions are not directly supported in any but the most expensive RDBMS. And even if you can write such a huge check, you still need to carefully choose partitioning keys to the point where you can never entirely ignore the limitation.

Perhaps more importantly, as we see some of the limitations of RDBMS and consequently some of the strategies that architects have used to mitigate their scaling issues, a picture slowly starts to emerge. It's a picture that makes some NoSQL solutions seem perhaps less radical and less scary than we may have thought at first, and more like a natural expression and encapsulation of some of the work that was already being done to manage very large databases.

Because of some of the inherent design decisions in RDBMS, it is not always as easy to scale as some other, more recent possibilities that take the structure of the Web into consideration. But it's not only the structure of the Web we need to consider, but also its phenomenal growth, because as more and more data becomes available, we need architectures that allow our organizations to take advantage of this data in near-time to support decision making and to offer new and more powerful features and capabilities to our customers.

Data Scale, Then and Now

It has been said, though it is hard to verify, that the 17th-century English poet John Milton had actually read every published book on the face of the earth. Milton knew many languages (he was even learning Navajo at the time of his death), and given that the total number of published books at that time was in the thousands, this would have been possible. The size of the world's data stores have grown somewhat since then.

With the rapid growth in the Web, there is great variety to the kinds of data that need to be stored, processed, and queried, and some variety to the businesses that use such data. Consider not only customer data at familiar retailers or suppliers, and not only digital video content, but also the required move to digital television and the explosive growth of email, messaging, mobile phones, RFID, Voice Over IP (VoIP) usage, and the Internet of Things (IoT). As we have departed from physical consumer media storage, companies that provide content—and the third-party value-add businesses built around them—require very scalable data solutions. Consider too that as a typical business application developer or database administrator, we may be used to thinking of relational databases as the center of our universe. You might then be surprised to learn that within corporations, around 80% of data is unstructured.

The Rise of NoSQL

The recent interest in non-relational databases reflects the growing sense of need in the software development community for web scale data solutions. The term "NoSQL" began gaining popularity around 2009 as a shorthand way of describing these databases. The term historically been the subject of much debate, but a consensus has emerged that the term refers to non-relational databases that support "not only SQL" semantics.

Various experts have attempted to organize these databases in a few broad categories; we'll examine a few of the most common:

Key-Value Stores
: In a key-value store, the data items are keys that have a set of attributes. All data relevant to a key is stored with the key; data is frequently duplicated. Popular key-value stores include Amazon's Dynamo DB, Riak, and Voldemort. Additionally, many popular caching technologies act as key-value stores, including Oracle Coherence, Redis and MemcacheD.

Column Stores
: Column stores are also frequently known as wide-column stores. Google's Bigtable served as the inspiration for implementations including Cassandra, Hypertable, and Apache Hadoop's HBase.

Document Stores
: The basic unit of storage in a document database is the complete document, often stored in a format such as JSON, XML, or YAML. Popular document stores include MongoDB and CouchDB

Graph Databases
: Graph databases represent data as a graph – a network of nodes and edges that connect the nodes. Both nodes and edges can have properties. Because they give heightened importance to relationships, graph databases such as FlockDB, Neo4J and Polyglot have proven popular for building social networking and semantic web applications.

Object Databases
: Object databases store data not in terms of relations and columns and rows, but in terms of the objects themselves, making it straightforward to use the database from an object-oriented application. Object databases such as db4o and InterSystems Caché allow you to avoid techniques like stored procedures and object-relational mapping (ORM) tools.

XML Databases
: XML databases are a special form of document databases, optimized specifically for working with XML. So-called "XML native" databases include Tamino from Software AG and eXist.

For a comprehensive list of NoSQL databases, see the site *http://nosql-database.org/*.

There is wide variety in the goals and features of these databases, but they tend to share a set of common characteristics. The most obvious of these is implied by the name NoSQL – these databases support data models, data definition languages (DDLs) and interfaces beyond the standard SQL available in popular relational databases. In addition, these databases are typically distributed systems without centralized control. They emphasize horizontal scalability and high availability, in some cases at the cost of strong consistency and ACID semantics. They tend to support rapid development and deployment. They take flexible approaches to schema definition, in some cases not requiring any schema to be defined up front. They provide support for Big Data and analytics use cases.

Over the past several years, there have been a large number of open source and commercial offerings in the NoSQL space. The adoption and quality of these have varied widely, but leaders have emerged in the categories above and become mature technologies with large installation bases and commercial support. I'm happy to report that Cassandra is one of those technologies, as we'll dig into more in the next chapter.

Summary

The relational model has served the software industry well over the past four decades, but the level of availability and scalability required for modern applications has stretched relational database technology to the breaking point.

The intention of this book is not to convince you by clever argument to adopt a non-relational database such as Apache Cassandra. It is only my intention to present what Cassandra can do and how it does it so that you can make an informed decision and get started working with it in practical ways if you find it applies.

Perhaps the ultimate question, then, is not “What’s wrong with relational databases?” but rather, “What kinds of things would I do with data if it wasn’t a problem?” In a world now working at web scale and looking to the future, Apache Cassandra might be one part of the answer.

CHAPTER 2

Introducing Cassandra

An invention has to make sense in the world in which it is finished, not the world in which it is started.

—Ray Kurzweil

In the previous chapter, we discussed the emergence of non-relational database technologies in order to meet the increasing demands of modern web scale applications. In this chapter, we'll focus on Cassandra's value proposition and key tenets to show how it rises to the challenge. We'll also learn about Cassandra's history and how you can get involved in the open source community that maintains Cassandra.

The Cassandra Elevator Pitch

Hollywood screenwriters and software startups are often advised to have their "elevator pitch" ready. This is a summary of exactly what their product is all about—concise, clear, and brief enough to deliver in just a minute or two, in the lucky event that they find themselves sharing an elevator with an executive or agent or investor who might consider funding their project. Cassandra has a compelling story, so let's boil it down to an elevator pitch that you can present to your manager or colleagues should the occasion arise.

Cassandra in 50 Words or Less

"Apache Cassandra is an open source, distributed, decentralized, elastically scalable, highly available, fault-tolerant, tuneably consistent, row-oriented database that bases its distribution design on Amazon's Dynamo and its data model on Google's Bigtable. Created at Facebook, it is now used at some of the most popular sites on the Web." That's exactly 50 words.

Of course, if you were to recite that to your boss in the elevator, you'd probably get a blank look in return. So let's break down the key points in the following sections.

Distributed and Decentralized

Cassandra is *distributed*, which means that it is capable of running on multiple machines while appearing to users as a unified whole. In fact, there is little point in running a single Cassandra node. Although you can do it, and that's acceptable for getting up to speed on how it works, you quickly realize that you'll need multiple machines to really realize any benefit from running Cassandra. Much of its design and code base is specifically engineered toward not only making it work across many different machines, but also for optimizing performance across multiple data center racks, and even for a single Cassandra cluster running across geographically dispersed data centers. You can confidently write data to anywhere in the cluster and Cassandra will get it.

Once you start to scale many other data stores (MySQL, Bigtable), some nodes need to be set up as masters in order to organize other nodes, which are set up as slaves. Cassandra, however, is decentralized, meaning that every node is identical; no Cassandra node performs certain organizing operations distinct from any other node. Instead, Cassandra features a peer-to-peer protocol and uses gossip to maintain and keep in sync a list of nodes that are alive or dead.

The fact that Cassandra is *decentralized* means that there is no single point of failure. All of the nodes in a Cassandra cluster function exactly the same. This is sometimes referred to as "server symmetry." Because they are all doing the same thing, by definition there can't be a special host that is coordinating activities, as with the master/slave setup that you see in MySQL, Bigtable, and so many others.

In many distributed data solutions (such as RDBMS clusters), you set up multiple copies of data on different servers in a process called replication, which copies the data to multiple machines so that they can all serve simultaneous requests and improve performance. Typically this process is not decentralized, as in Cassandra, but is rather performed by defining a *master/slave relationship*. That is, all of the servers in this kind of cluster don't function in the same way. You configure your cluster by designating one server as the master and others as slaves. The master acts as the authoritative source of the data, and operates in a unidirectional relationship with the slave nodes, which must synchronize their copies. If the master node fails, the whole database is in jeopardy. The decentralized design is therefore one of the keys to Cassandra's high availability. Note that while we frequently understand master/slave replication in the RDBMS world, there are NoSQL databases such as MongoDB that follow the master/slave scheme as well.

Decentralization, therefore, has two key advantages: it's simpler to use than master/slave, and it helps you avoid outages. It can be easier to operate and maintain a decentralized store than a master/slave store because all nodes are the same. That means that you don't need any special knowledge to scale; setting up 50 nodes isn't much different from setting up one. There's next to no configuration required to support it. Moreover, in a master/slave setup, the master can become a single point of failure (SPOF). To avoid this, you often need to add some complexity to the environment in the form of multiple masters. Because all of the replicas in Cassandra are identical, failures of a node won't disrupt service.

In short, because Cassandra is distributed and decentralized, there is no single point of failure, which supports high availability.

Elastic Scalability

Scalability is an architectural feature of a system that can continue serving a greater number of requests with little degradation in performance. Vertical scaling—simply adding more hardware capacity and memory to your existing machine—is the easiest way to achieve this. Horizontal scaling means adding more machines that have all or some of the data on them so that no one machine has to bear the entire burden of serving requests. But then the software itself must have an internal mechanism for keeping its data in sync with the other nodes in the cluster.

Elastic scalability refers to a special property of horizontal scalability. It means that your cluster can seamlessly scale up and scale back down. To do this, the cluster must be able to accept new nodes that can begin participating by getting a copy of some or all of the data and start serving new user requests without major disruption or reconfiguration of the entire cluster. You don't have to restart your process. You don't have to change your application queries. You don't have to manually rebalance the data yourself. Just add another machine—Cassandra will find it and start sending it work.

Scaling down, of course, means removing some of the processing capacity from your cluster. You might do this for business reasons, such as adjusting to seasonal workloads in retail or travel applications. Or perhaps there will be technical reasons such as moving parts of your application to another platform. As much as we try to minimize these situations, they still happen. But when they do, you won't need to upset the entire apple cart to scale back.

High Availability and Fault Tolerance

In general architecture terms, the availability of a system is measured according to its ability to fulfill requests. But computers can experience all manner of failure, from hardware component failure to network disruption to corruption. Any computer is susceptible to these kinds of failure. There are of course very sophisticated (and often prohibitively expensive) computers that can themselves mitigate many of these circumstances, as they include internal hardware redundancies and facilities to send notification of failure events and hot swap components. But anyone can accidentally break an Ethernet cable, and catastrophic events can beset a single data center. So for a system to be highly available, it must typically include multiple networked computers, and the software they're running must then be capable of operating in a cluster and have some facility for recognizing node failures and failing over requests to another part of the system.

Cassandra is highly available. You can replace failed nodes in the cluster with no downtime, and you can replicate data to multiple data centers to offer improved local performance and prevent downtime if one data center experiences a catastrophe such as fire or flood.

Tuneable Consistency

Consistency essentially means that a read always returns the most recently written value. Consider two customers are attempting to put the same item into their shopping carts on an ecommerce site. If I place the last item in stock into my cart an instant after you do, you should get the item added to your cart, and I should be informed that the item is no longer available for purchase. This is guaranteed to happen when the state of a write is consistent among all nodes that have that data.

But as we'll see later, scaling data stores means making certain trade-offs between data consistency, node availability, and partition tolerance. Cassandra is frequently called "eventually consistent," which is a bit misleading. Out of the box, Cassandra trades some consistency in order to achieve total availability. But Cassandra is more accurately termed "tuneably consistent," which means it allows you to easily decide the level of consistency you require, in balance with the level of availability.

Let's take a moment to unpack this, as the term "eventual consistency" has caused some uproar in the industry. Some practitioners hesitate to use a system that is described as "eventually consistent."

For detractors of eventual consistency, the broad argument goes something like this: eventual consistency is maybe OK for social web applications where data doesn't *really* matter. After all, you're just posting to mom what little Billy ate for breakfast, and if it gets lost, it doesn't really matter. But the data *I* have is actually really important, and it's ridiculous to think that I could allow eventual consistency in my model.

Set aside the fact that all of the most popular web applications (Amazon, Facebook, Google, Twitter) are using this model, and that perhaps there's something to it. Presumably such data is very important indeed to the companies running these applications, because that data is their primary product, and they are multibillion-dollar companies with billions of users to satisfy in a sharply competitive world. It may be possible to gain guaranteed, immediate, and perfect consistency throughout a highly trafficked system running in parallel on a variety of networks, but if you want clients to get their results sometime this year, it's a very tricky proposition.

The detractors claim that some Big Data databases such as Cassandra have merely eventual consistency, and that all other distributed systems have *strict* consistency. As with so many things in the world, however, the reality is not so black and white, and the binary opposition between consistent and not-consistent is not truly reflected in practice. There are instead *degrees* of consistency, and in the real world they are very susceptible to external circumstance.

Eventual consistency is one of several consistency models available to architects. Let's take a look at these models so we can understand the trade-offs:

Strict consistency
: This is sometimes called sequential consistency, and is the most stringent level of consistency. It requires that any read will always return the most recently written value. That sounds perfect, and it's exactly what I'm looking for. I'll take it! However, upon closer examination, what do we find? What precisely is meant by "most recently written"? Most recently to whom? In one single-processor machine, this is no problem to observe, as the sequence of operations is known to the one clock. But in a system executing across a variety of geographically dispersed data centers, it becomes much more slippery. Achieving this implies some sort of global clock that is capable of timestamping all operations, regardless of the location of the data or the user requesting it or how many (possibly disparate) services are required to determine the response.

Causal consistency
: This is a slightly weaker form of strict consistency. It does away with the fantasy of the single global clock that can magically synchronize all operations without creating an unbearable bottleneck. Instead of relying on timestamps, causal consistency instead takes a more semantic approach, attempting to determine the cause of events to create some consistency in their order. It means that writes that are potentially related must be read in sequence. If two different, unrelated operations suddenly write to the same field, then those writes are inferred not to be causally related. But if one write occurs after another, we might infer that they are causally related. Causal consistency dictates that causal writes must be read in sequence.

Weak (eventual) consistency
: Eventual consistency means on the surface that all updates will propagate throughout all of the replicas in a distributed system, but that this may take some time. Eventually, all replicas will be consistent.

Eventual consistency becomes suddenly very attractive when you consider what is required to achieve stronger forms of consistency.

When considering consistency, availability, and partition tolerance, we can achieve only two of these goals in a given distributed system, a trade off known as the CAP Theorem (we explore the CAP Theorem in the section "Brewer's CAP Theorem" on page 23). At the center of the problem is data update replication. To achieve a strict consistency, all update operations will be performed synchronously, meaning that they must block, locking all replicas until the operation is complete, and forcing competing clients to wait. A side effect of such a design is that during a failure, some of the data will be entirely unavailable. As Amazon CTO Werner Vogels puts it, "rather than dealing with the uncertainty of the correctness of an answer, the data is made unavailable until it is absolutely certain that it is correct" ("Dynamo: Amazon's Highly Distributed Key-Value Store": [*http://www.allthingsdistributed.com/2007/10/amazons_dynamo.html*], 207).

We could alternatively take an optimistic approach to replication, propagating updates to all replicas in the background in order to avoid blowing up on the client. The difficulty this approach presents is that now we are forced into the situation of detecting and resolving conflicts. A design approach must decide whether to resolve these conflicts at one of two possible times: during reads or during writes. That is, a distributed database designer must choose to make the system either always readable or always writable.

Dynamo and Cassandra choose to be always writable, opting to defer the complexity of reconciliation to read operations, and realize tremendous performance gains. The alternative is to reject updates amidst network and server failures.

In Cassandra, consistency is not an all-or-nothing proposition. We might more accurately term it "tuneable consistency" because the client can control the number of replicas to block on for all updates. This is done by setting the consistency level against the replication factor.

The *replication factor* lets you decide how much you want to pay in performance to gain more consistency. You set the replication factor to the number of nodes in the cluster you want the updates to propagate to (remember that an update means any add, update, or delete operation).

The *consistency level* is a setting that clients must specify on every operation and that allows you to decide how many replicas in the cluster must acknowledge a write operation or respond to a read operation in order to be considered successful. That's the part where Cassandra has pushed the decision for determining consistency out to the client.

So if you like, you could set the consistency level to a number equal to the replication factor, and gain stronger consistency at the cost of synchronous blocking operations that wait for all nodes to be updated and declare success before returning. This is not often done in practice with Cassandra, however, for reasons that should be clear (it defeats the availability goal, would impact performance, and generally goes against the grain of why you'd want to use Cassandra in the first place). So if the client sets the consistency level to a value less than the replication factor, the update is considered successful even if some nodes are down.

Brewer's CAP Theorem

In order to understand Cassandra's design and its label as an "eventually consistent" database, we need to understand the CAP theorem. The CAP theorem is sometimes called Brewer's theorem after its author, Eric Brewer.

While working at University of California at Berkeley, Eric Brewer posited his CAP theorem in 2000 at the ACM Symposium on the Principles of Distributed Computing. The theorem states that within a large-scale distributed data system, there are three requirements that have a relationship of sliding dependency: Consistency, Availability, and Partition Tolerance.

Consistency
: All database clients will read the same value for the same query, even given concurrent updates.

Availability
: All database clients will always be able to read and write data.

Partition Tolerance
: The database can be split into multiple machines; it can continue functioning in the face of network segmentation breaks.

Brewer's theorem is that in any given system, you can strongly support only two of the three. This is analogous to the saying you may have heard in software development: "You can have it good, you can have it fast, you can have it cheap: pick two."

We have to choose between them because of this sliding mutual dependency. The more consistency you demand from your system, for example, the less partition-tolerant you're likely to be able to make it, unless you make some concessions around availability.

The CAP theorem was formally proved to be true by Seth Gilbert and Nancy Lynch of MIT in 2002. In distributed systems, however, it is very likely that you will have network partitioning, and that at some point, machines will fail and cause others to become unreachable. Networking issues such as packet loss or high latency are nearly inevitable and have the potential to cause temporary partitions. This leads us to the conclusion that a distributed system must do its best to continue operating in the face of network partitions (to be Partition-Tolerant), leaving us with only two real options to compromise on: Availability and Consistency.

Figure 2-1 illustrates visually that there is no overlapping segment where all three are obtainable.

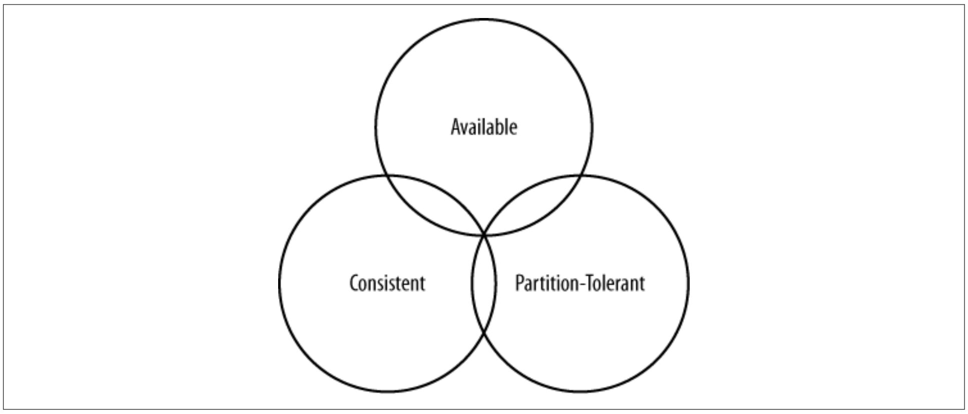


Figure 2-1. CAP Theorem indicates that you can realize only two of these properties at once

It might prove useful at this point to see a graphical depiction of where each of the nonrelational data stores we'll look at falls within the CAP spectrum. The graphic in Figure 2-2 was inspired by a slide in a 2009 talk given by Dwight Merriman, CEO and founder of MongoDB, to the MySQL User Group in New York City (you can watch it online at *http://bit.ly/7r6kRg*). However, I have modified the placement of some systems based on my research.

Figure 2-2 shows the general focus of some of the different databases we discuss in this chapter. Note that placement of the databases in this chart could change based on configuration. As Stu Hood points out, a distributed MySQL database can count as a consistent system only if you're using Google's synchronous replication patches; otherwise, it can only be Available and Partition-Tolerant (AP).

It's interesting to note that the design of the system around CAP placement is independent of the orientation of the data storage mechanism; for example, the CP edge is populated by graph databases and document-oriented databases alike.

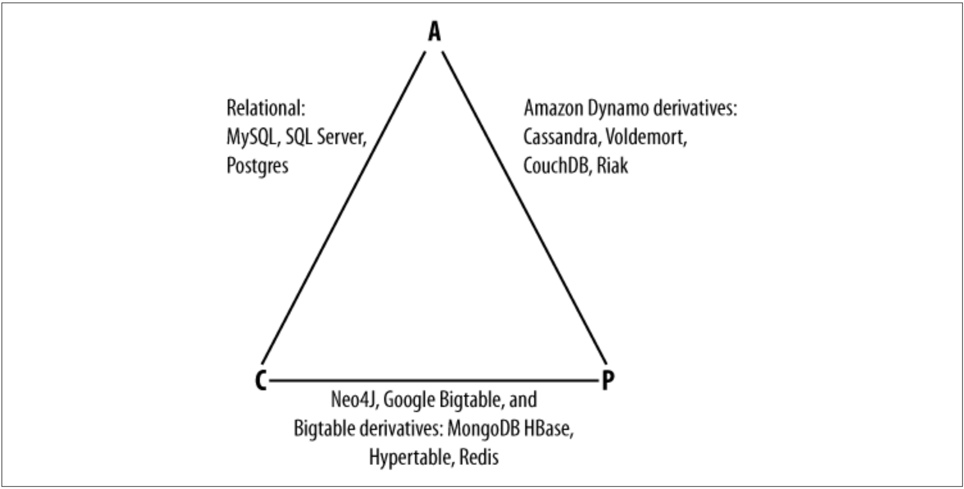


Figure 2-2. Where different databases appear on the CAP continuum

In this depiction, relational databases are on the line between Consistency and Availability, which means that they can fail in the event of a network failure (including a cable breaking). This is typically achieved by defining a single master server, which could itself go down, or an array of servers that simply don't have sufficient mechanisms built in to continue functioning in the case of network partitions.

Graph databases such as Neo4J and the set of databases derived at least in part from the design of Google's Bigtable database (such as MongoDB, HBase, Hypertable, and Redis) all are focused slightly less on Availability and more on ensuring Consistency and Partition Tolerance.

More on NoSQL

If you're interested in the properties of other Big Data or NoSQL databases, see this book's ???.

Finally, the databases derived from Amazon's Dynamo design include Cassandra, Project Voldemort, CouchDB, and Riak. These are more focused on Availability and Partition-Tolerance. However, this does not mean that they dismiss Consistency as unimportant, any more than Bigtable dismisses Availability. According to the Bigtable paper, the average percentage of server hours that "some data" was unavailable is 0.0047% (section 4), so this is relative, as we're talking about very robust systems already. If you think of each of these letters (C, A, P) as knobs you can tune to arrive at the system you want, Dynamo derivatives are intended for employment in the many use cases where "eventual consistency" is tolerable and where "eventual" is a matter of milliseconds, read repairs mean that reads will return consistent values, and you can achieve strong consistency if you want to.

So what does it mean in practical terms to support only two of the three facets of CAP?

CA
: To primarily support Consistency and Availability means that you're likely using two-phase commit for distributed transactions. It means that the system will block when a network partition occurs, so it may be that your system is limited to a single data center cluster in an attempt to mitigate this. If your application needs only this level of scale, this is easy to manage and allows you to rely on familiar, simple structures.

CP
: To primarily support Consistency and Partition Tolerance, you may try to advance your architecture by setting up data shards in order to scale. Your data will be consistent, but you still run the risk of some data becoming unavailable if nodes fail.

AP
: To primarily support Availability and Partition Tolerance, your system may return inaccurate data, but the system will always be available, even in the face of network partitioning. DNS is perhaps the most popular example of a system that is massively scalable, highly available, and partition-tolerant.

Note that this depiction is intended to offer an overview that helps draw distinctions between the broader contours in these systems; it is not strictly precise. For example, it's not entirely clear where Google's Bigtable should be placed on such a continuum. The Google paper describes Bigtable as "highly available," but later goes on to say that if Chubby (the Bigtable persistent lock service) "becomes unavailable for an extended period of time [caused by Chubby outages or network issues], Bigtable becomes unavailable" (section 4). On the matter of data reads, the paper says that "we do not consider the possibility of multiple copies of the same data, possibly in alternate forms due to views or indices." Finally, the paper indicates that "centralized control and Byzantine fault tolerance are not Bigtable goals" (section 10). Given such variable information, you can see that determining where a database falls on this sliding scale is not an exact science.

An updated perspective on CAP

Eric Brewer provided an updated perspective on his CAP theorem in a February 2012 article in IEEE Computer, which has also been made available at: *http://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed*. Brewer now describes the "2 out of 3" axiom as somewhat misleading. He notes that designers only need sacrifice consistency or availability in the presence of partitions, and that advances in partition recovery techniques have made it possible for designers to achieve high levels of both consistency and availability.

These advances in partition recovery certainly would include Cassandra's usage of mechanisms such as hinted handoff and read repair. We'll explore these in #chp-architecture. However, it is important to recognize that these partition recovery mechanisms are not infallible. There is still immense value in Cassandra's tuneable consistency, allowing Cassandra to function effectively in a diverse set of deployments in which it is not possible to completely prevent partitions.

Row-Oriented

Cassandra's data model can be described as a partitioned row store, in which data is stored in sparse multidimensional hashtables. "Sparse" means that for any given row you can have one or more columns, but each row doesn't need to have all the same columns as other rows like it (as in a relational model). "Partitioned" means that each row has a unique key which makes its data accessible, and the keys are used to distribute the rows across multiple data stores.

Row-Oriented vs. Column-Oriented

Cassandra has frequently been referred to as a "column-oriented" database, which has proved the source of some confusion. A column-oriented database is one in which the data is actually stored by columns, as opposed to relational databases, which store data in rows. Part of the confusion that occurs in classifying databases is that there can be a difference between the API exposed by the database and the underlying storage on disk. So Cassandra is not really column-oriented, in that its data store is not organized primarily around columns.

In the relational storage model, all of the columns for a table are defined beforehand and space is allocated for each column whether it is populated or not. In contrast, Cassandra stores data in a multidimensional, sorted hash table. As data is stored in each column, it is stored as a separate entry in the hash table. Column values are stored according to a consistent sort order, omitting columns that are not populated, which enables more efficient storage and query processing. We'll examine Cassandra's data model in more detail in #chp-cql.

Is Cassandra "Schema-Free"?

In its early versions. Cassandra was faithful to the original BigTable whitepaper in supporting a "schema-free" data model in which new columns can be defined dynamically. Schema-free databases such as BigTable and MongoDB have the advantage of being very extensible and highly performant in accessing large amounts of data. The major drawback of schema-free databases is the difficulty in determining the meaning and format of data, which limits the ability to perform complex queries. These disadvantages proved a barrier to adoption for many, especially as startup projects which benefitted from the initial flexibility matured into more complex enterprises involving multiple developers and administrators.

The solution for those users was the introduction of the Cassandra Query Language (CQL), which provides a way to define schema via a syntax similar to the Structured Query Language (SQL) familiar to those coming from a relational background. Initially, CQL was provided as another interface to Cassandra alongside the schema-free interface based on the Apache Thrift project. During this transitional phase the term "Schema-optional" was used to describe that data models could be defined by schema using CQL, but could also be dynamically extended to add new columns via the Thrift API. During this period, the underlying data storage continued to be based on the BigTable model.

Starting with the 3.0 release, the Thrift-based API which supported dynamic column creation is been deprecated, and Cassandra's underlying storage has been re-implemented to more closely align with CQL. Cassandra does not entirely limit the ability to dynamically extend the schema on the fly, but the way it works is significantly different. CQL collections such as lists, sets and especially maps provide the ability to add content in a less structured form that can be leveraged to extend an existing schema. CQL also provides the ability to change the type of columns in certain instances, and facilities to support the storage of JSON-formatted text.

So perhaps the best way to describe Cassandra's current posture is that it supports "flexible schema".

High Performance

Cassandra was designed specifically from the ground up to take full advantage of multiprocessor/multi-core machines, and to run across many dozens of these machines housed in multiple data centers. It scales consistently and seamlessly to hundreds of terabytes. Cassandra has been shown to perform exceptionally well under heavy load. It consistently can show very fast throughput for writes per second on basic commodity computers, whether physical hardware or virtual machines. As you add more servers, you can maintain all of Cassandra's desirable properties without sacrificing performance.

Where Did Cassandra Come From?

The Cassandra data store is an open source Apache project available at *http://cassandra.apache.org*. Cassandra originated at Facebook in 2007 to solve that company's inbox search problem, in which they had to deal with large volumes of data in a way that was difficult to scale with traditional methods. Specifically, the team had requirements to handle huge volumes of data in the form of message copies, reverse indices of messages, and many random reads and many simultaneous random writes.

The team was led by Jeff Hammerbacher, with Avinash Lakshman, Karthik Ranganathan, and Facebook engineer on the Search Team Prashant Malik as key engineers. The code was released as an open source Google Code project in July 2008. During its tenure as a Google Code project in 2008, the code was updateable only by Facebook engineers, and little community was built around it as a result. So in March 2009 it was moved to an Apache Incubator project, and on February 17, 2010 it was voted into a top-level project. You can find a list of the Cassandra committers at: *http://wiki.apache.org/cassandra/Committers*, many of whom have been with the project since 2010/2011. The committers represent companies including Twitter, LinkedIn, Apple, as well as independent developers.

The Paper that Introduced Cassandra to the World

A central paper on Cassandra by Facebook's Lakshman and Malik called "A Decentralized Structured Storage System" is available at: *http://www.cs.cornell.edu/projects/ladis2009/papers/lakshman-ladis2009.pdf*. An updated commentary on this paper was provided by Jonathan Ellis corresponding to the 2.0 release, noting changes to the technology since the transition to Apache. You can find this update at: *http://docs.datastax.com/en/articles/cassandra/cassandra-thenandnow.html*. We'll unpack some of these changes in more detail in #section-release-history.

How Did Cassandra Get Its Name?

In Greek mythology, Cassandra was the daughter of King Priam and Queen Hecuba of Troy. Cassandra was so beautiful that the god Apollo gave her the ability to see the future. But when she refused his amorous advances, he cursed her such that she would still be able to accurately predict everything that would happen—but no one would believe her. Cassandra foresaw the destruction of her city of Troy, but was powerless to stop it. The Cassandra distributed database is named for her. I speculate that it is also named as kind of a joke on the Oracle at Delphi, another seer for whom a database is named.

As commercial interest in Cassandra grew, the need for production support became apparent. Jonathan Ellis, the Apache Project Chair for Cassandra, and his colleague Matt Pfeil formed a services company called DataStax (originally known as Riptano) in April of 2010. DataStax has provided leadership and support for the Cassandra project, employing several Cassandra committers.

DataStax provides free products including Cassandra drivers for various languages and tools for development and administration of Cassandra. Paid product offerings include enterprise versions of the Cassandra server and tools, integrations with other data technologies, and product support. Unlike some other open source projects that have commercial backing, changes are added first to the Apache open source project, and then rolled into the commercial offering shortly after each Apache release.

DataStax also provides the Planet Cassandra website (*http://www.planetcassandra.org*) as a resource to the Cassandra community. This site is a great location to learn about the ever-growing list of companies and organizations that are using Cassandra in industry and academia: *http://www.planetcassandra.org/companies/*. Industries represented run the gamut: financial services, telecommunications, education, social media, entertainment, marketing, retail, hospitality, transportation, healthcare, energy, philanthropy, aerospace, defense and technology. Chances are that you will find a number of case studies here that are relevant to your needs.

Release History

Now that we've learned about the people and organizations that have shaped Cassandra, let's take a look at how Cassandra has matured through its various releases since becoming an official Apache project. If you're new to Cassandra, don't worry if some of these concepts and terms are new to you, we'll dive into them in more depth in due time. You can return to this list later to get a sense of the trajectory of how Cassandra has matured over time and its future directions. If you've used Cassandra in the past, this summary will give you a quick primer on what's changed.

Performance and Reliability Improvements

This list focuses primarily on features that have been added over the course of Cassandra's lifespan. This is not to discount the steady and substantial improvements in reliability and read/write performance.

Release 0.6

This was the first release after Cassandra graduated from the Apache Incubator to a top level project. Releases in this series ran from 0.6.0 in April 2010 through 0.6.13 in April 2011. Features in this series included:

- Integration with Apache Hadoop, allowing easy data retrieval from Cassandra via Map Reduce.
- Integrated row caching, which helped eliminate the need for applications to deploy other caching technologies alongside Cassandra.

Release 0.7

Releases in this series ran from 0.7.0 in January 2011 through 0.7.10 in October 2011. Key features and improvements included:

- Secondary indexes, that is, indexes on non-primary columns
- Support for large rows, containing up to two billion columns
- Online schema changes, including adding, renaming, and removing keyspaces and column families in live clusters without a restart, via the Thrift API
- Expiring columns, via specification of a time-to-live (TTL) per column
- The NetworkTopologyStrategy was introduced to support multi-data center deployments, allowing a separate replication factor per data center, per keyspace
- Configuration files were converted from XML to the more readable YAML format

Release 0.8

This release began a major shift in Cassandra APIs with the introduction of CQL. Releases in this series ran from 0.8.0 in June 2011 through 0.7.10 in February 2012. Key features and improvements included:

- Distributed counters were added as a new data type that incrementally counts up or down
- The sstableloader tool was introduced to support bulk loading of data into Cassandra clusters
- An off heap row cache was provided to allow usage of native memory instead of the JVM heap
- Concurrent compaction allowed for multi-threaded execution and throttling control of sstable compaction
- Improved memory configuration parameters allowed more flexible control over the size of memtables

Release 1.0

In keeping with common version numbering practice, this is officially the first production release of Cassandra, although many companies were using Cassandra in production well before this point. Releases in this series ran from 1.0.0 in October 2011 through 1.0.12 in October 2012. In keeping with the focus on production readiness, improvements focused on performance and enhancements to existing features::

- CQL 2 added improvements including the ability to alter tables and columns, support for counters and TTL, and the ability to retrieve the count of items matching a query
- The leveled compaction strategy was introduced as an alternative to the original size tiered compaction strategy, allowing for faster reads at the expense of more IO on writes
- Compression of sstable files, configurable on a per-table level

Release 1.1

Releases in this series ran from 1.1.0 in April 2011 through 1.1.12 in May 2013. Key features and improvements included:

- CQL 3 added the *timeuuid* type, and the ability to create tables with compound primary keys including clustering keys. Clustering keys support "order by" semantics to allow sorting. This was a much anticipated feature that allowed the creation of "wide rows" via CQL.
- Support for comma separated variable (CSV) import/export in *cqlsh*
- Flexible data storage settings allow the storage of data in SSDs or magnetic storage, selectable by table
- The schema update mechanism was re—implemented to allow concurrent changes and improve reliability. Schema are now stored in tables in the *system* keyspace.
- Caching was updated to provide more straightforward configuration of cache sizes
- A utility to leverage the bulk loader from Hadoop, allowing efficient export of data from Hadoop to Cassandra
- Row level isolation was added to assure that when multiple columns are updated on a write, it is not possible for a read to get a mix of new and old column values

Release 1.2

Releases in this series ran from 1.2.0 in January 2013 through 1.2.19 in September 2014. Notable features and improvements included:

- CQL 3 added collection types (sets, lists and maps), prepared statements, and a binary protocol as a replacement for Thrift.
- Virtual nodes spread data more evenly across the nodes in a cluster, improving performance, especially when adding or replacing nodes
- Atomic batches ensure that all writes in a batch succeed or fail as a unit.
- The *system* keyspace contains the *local* table containing information about the local node and the *peers* table describing other nodes in the cluster.
- Request tracing can be enabled to allow clients to see the interactions between nodes for reads and writes. Tracing provides valuable insight into what is going on behind the scenes and can help developers understand the implications of various table design options.
- Most data structures were moved off of the JVM heap to native memory.
- Disk failure policies allow flexible configuration of behaviors including removing a node from the cluster on disk failure or making a best effort to access data from memory, even if stale.

Release 2.0

The 2.0 release was an especially significant milestone in the history of Cassandra, as it marked the culmination of the CQL capability, as well as a new level of production maturity. This included significant performance improvements and cleanup of the codebase to pay down 5 years of accumulated technical debt. Releases in this series ran from 2.0.0 in September 2013 through 2.0.16 in June 2015. Highlights included:

- Lightweight transactions were added using the Paxos consensus protocol
- CQL3 improvements included the addition of DROP semantics on the ALTER command, conditional schema modifications (IF EXISTS, IF NOT EXISTS), and the ability to create secondary indexes on primary key columns.
- Native CQL protocol improvements began to make CQL demonstrably more performant than Thrift.
- A prototype implementation of triggers was added, providing an extensible way to react to write operations. Triggers can be implemented in any JVM language.
- Java 7 was required for the first time.
- Static columns were added in the 2.0.6 release.

Release 2.1

Releases in this series ran from 2.1.0 in September 2014 through 2.1.8 in June 2015 (the stable production release at the time of writing). Key features and improvements included:

- CQL3 added user defined types (UDT), and the ability to create secondary indexes on collections.
- Configuration options were added to move memtable data off heap to native memory.
- Row caching was made more configurable to allow setting the number of cached rows per partition
- Counters were re-implemented to improve performance and reliability

Release 2.2

Release 2.2.0 became available in July 2015. Notable features and improvements included:

- CQL3 improvements including support for JSON formatted input/output and user defined functions.
- With this release, Windows became a fully supported operating system. Although Cassandra still performs best on Linux systems, improvements in file IO and scripting have made it much easier to run Cassandra on Windows.
- The Date Tiered Compaction Strategy (DTCS) was introduced to improve performance of time series data.
- Role-based access control (RBAC) was introduced to allow more flexible management of authorization

Release 3.0

The 3.0 release was made available in November 2015. It includes features such as:

- The underlying storage engine has been rewritten to more closely match CQL constructs
- Support for materialized views (sometimes also called global indexes)
- Java 8 is now the supported version
- The Thrift-based Command Line Interface (CLI) is removed

Supported releases

There are two officially supported releases of Cassandra at any one time: the latest stable release, which is considered appropriate for production, and the latest development release. For example, as of November 2015, the latest stable release is 2.2.3, while the development release is 3.0.0.

Users of Cassandra are strongly recommended to track the latest stable release in production. Anecdotally, a substantial majority of issues and questions posted to the Cassandra-users email list are with respect to releases that are no longer supported. Cassandra experts are very gracious in answering questions and diagnosing issues with these unsupported releases, but more often than not the recommendation is to upgrade as soon as possible to a release that addresses the issue.

As you will have noticed, the trends in these releases include:

- continuous improvement in the capabilities of CQL
- a growing list of clients for popular languages built on a common set of metaphors
- exposure of configuration options to tune performance and optimize resource usage
- performance and reliability improvements, and reduction of technical debt

Tick-tock releases

In June 2015, the Cassandra team announced plans to adopt a tick-tock release model as part of increased emphasis on improving agility and the quality of Cassandra releases.

The tick-tock release model popularized by Intel was originally intended for chip design, and referred to changing chip architecture and production processes in alternate builds. You can read more about this approach at: *http://www.intel.com/content/www/us/en/silicon-innovations/intel-tick-tock-model-general.html*.

The tick-tock approach has proven to be useful in software development as well. Starting with the Cassandra 3.0 release, even numbered releases are feature releases with some bug fixes, while odd numbered releases are focused on bug fix releases, with the goal of releasing each month.

Is Cassandra a Good Fit for My Project?

We have now unpacked the elevator pitch and have an understanding of Cassandra's advantages. Despite Cassandra's sophisticated design and smart features, it is not the right tool for every job. So in this section let's take a quick look at what kind of projects Cassandra is a good fit for.

Large Deployments

You probably don't drive a semi truck to pick up your dry cleaning; semis aren't well suited for that sort of task. Lots of careful engineering has gone into Cassandra's high availability, tuneable consistency, peer-to-peer protocol, and seamless scaling, which are its main selling points. None of these qualities is even meaningful in a single-node deployment, let alone allowed to realize its full potential.

There are, however, a wide variety of situations where a single-node relational database is all we may need. So do some measuring. Consider your expected traffic, throughput needs, and SLAs. There are no hard and fast rules here, but if you expect that you can reliably serve traffic with an acceptable level of performance with just a few relational databases, it might be a better choice to do so, simply because RDBMS are easier to run on a single machine and are more familiar.

If you think you'll need at least several nodes to support your efforts, however, Cassandra might be a good fit. If your application is expected to require dozens of nodes, Cassandra might be a great fit.

Lots of Writes, Statistics, and Analysis

Consider your application from the perspective of the ratio of reads to writes. Cassandra is optimized for excellent throughput on writes.

Many of the early production deployments of Cassandra involve storing user activity updates, social network usage, recommendations/reviews, and application statistics. These are strong use cases for Cassandra because they involve lots of writing with less predictable read operations, and because updates can occur unevenly with sudden spikes. In fact, the ability to handle application workloads that require high performance at significant write volumes with many concurrent client threads is one of the primary features of Cassandra.

According to the project wiki, Cassandra has been used to create a variety of applications, including a windowed time-series store, an inverted index for document searching, and a distributed job priority queue.

Geographical Distribution

Cassandra has out-of-the-box support for geographical distribution of data. You can easily configure Cassandra to replicate data across multiple data centers. If you have a globally deployed application that could see a performance benefit from putting the data near the user, Cassandra could be a great fit.

Evolving Applications

If your application is evolving rapidly and you're in "startup mode," Cassandra might be a good fit given its support for flexible schemas. This makes it easy to keep your database in step with application changes as you rapidly deploy.

Getting Involved

The strength and relevance of any technology depend on the investment of individuals in a vibrant community environment. Thankfully, the Cassandra community is active and healthy, offering a number of ways for you to participate. We'll start with a few steps in #chp-installing such as downloading Cassandra and building from the source. Here are a few other ways to get involved:

Chat

Many of the Cassandra developers and community members hang out in the `#cassandra` channel on *webchat.freenode.net*. This informal environment is a great place to get your questions answered or offer up some answers of your own.

Mailing Lists

The Apache project hosts several mailing lists to which you can subscribe to learn about various topics of interest:

- *user@cassandra.apache.org* provides a general discussion list for users and is frequently used by new users or those needing assistance.
- *dev@cassandra.apache.org* is used by developers to discuss changes, prioritize work and approve releases.
- *client-dev@cassandra.apache.org* is used for discussion specific to development of Cassandra clients for various programming languages.
- *commits@cassandra.apache.org* tracks Cassandra code commits. This is a fairly high volume list and is primarily of interest to committers.

Releases are typically announced to both the developer and user mailing lists.

Issues

If you encounter issues using Cassandra and feel you have discovered a defect, you should feel free to submit an issue to the Cassandra JIRA at *https://issues.apache.org/jira/browse/cassandra/*. In fact, users who identify defects on the *user@cassandra.apache.org* list are frequently encouraged to create JIRA issues.

Blogs

The DataStax developer blog (*http://www.datastax.com/dev/blog*) features posts on using Cassandra, announcements of Apache Cassandra and DataStax product releases, as well as occasional deep-dive technical articles on Cassandra implementation details and features under development. The Planet Cassandra blog (*http://www.planetcassandra.org/blog/*) provides similar technical content, but has a greater focus on profiling companies using Cassandra.

The Apache Cassandra Wiki (*http://wiki.apache.org/cassandra/*) provides helpful articles on getting started and configuration, but note that some content may not be fully up to date with current releases.

Meetups

A *meetup* group is a local community of people who meet face to face to discuss topics of common interest. These groups provide an excellent opportunity to network, learn, or share your knowledge by offering a presentation of your own. There are Cassandra meetups on every continent, so you stand a good chance of being able to find one in your area: *http://live-pc-development.pantheon.io/join-your-local-meetup/*.

Training and Conferences

DataStax offers online training (*https://academy.datastax.com*), and in June 2015 announced a partnership with O'Reilly Media to produce Cassandra certifications. DataStax also hosts annual Cassandra Summits in locations around the world: *http://cassandrasummit-datastax.com/*.

A Marketable Skill

There continues to be increased demand for Cassandra developers and administrators. A 2015 salary survey by Dice.com placed Cassandra as the second most highly compensated skill set: *http://marketing.dice.com/pdf/Dice_TechSalarySurvey_2015.pdf*.

Summary

In this chapter, we've taken an introductory look at Cassandra's defining characteristics, history, and major features. We have learned about the Cassandra user community and how companies are using Cassandra. Now we're ready to start getting some hands-on experience.

CHAPTER 3

Installing Cassandra

For those among us who like instant gratification, we'll start by installing Cassandra. Because Cassandra introduces a lot of new vocabulary, there might be some unfamiliar terms as we walk through this. That's OK; the idea here is to get set up quickly in a simple configuration to make sure everything is running properly. This will serve as an orientation. Then, we'll take a step back and understand Cassandra in its larger context.

Installing the Apache Distribution

Cassandra is available for download from the Web at *http://cassandra.apache.org*. Just click the link on the home page to download a version as a gzipped tarball. Typically two versions of Cassandra are provided. The *latest release* is recommended for those starting new projects not yet in production. The *most stable release* is the one recommended for production usage. For all releases, the prebuilt binary is named *apache-cassandra-x.x.x-bin.tar.gz*, where *x.x.x* represents the version number. The download is around 23MB.

Extracting the Download

The simplest way to get started is to download the prebuilt binary. You can unpack the compressed file using any regular ZIP utility. On Unix-based systems such as Linux or MacOS, GZip extraction utilities should be preinstalled; on Windows, you'll need to get a program such as WinZip, which is commercial, or something like 7-Zip, which is freeware. You can download the freeware program 7-Zip from http://www.7-zip.org.

Open your extracting program. You might have to extract the ZIP file and the TAR file in separate steps. Once you have a folder on your filesystem called *apache-cassandra-x.x.x*, you're ready to run Cassandra.

What's In There?

Once you decompress the tarball, you'll see that the Cassandra binary distribution includes several directories. Let's take a moment to look around and see what we have.

bin
: This directory contains the executables to run Cassandra and as well as clients including the query language shell (*cqlsh*) and the command-line interface (CLI) client. It also has scripts to run the `nodetool`, which is a utility for inspecting a cluster to determine whether it is properly configured, and to perform a variety of maintenance operations. We look at `nodetool` in depth later. The directory also contains several utilities for performing operations on SSTables, including listing the keys of an SSTable (sstablekeys), bulk extraction and restoration of SSTable contents (sstableloader), and upgrading SSTables to a new version of Cassandra (sstableupgrade).

conf
: This directory contains the files for configuring your Cassandra instance. The required configuration files include: the *cassandra.yaml* file which is the primary configuration for running Cassandra; and the *logback.xml* file, which lets you change the logging settings to suit your needs. Additional files can optionally be used to configure the network topology, archival and restore commands, and triggers. We see how to use these configuration files when we discuss configuration in ???.

interface
: For versions prior to Cassandra 3.0, this directory contains a single file, called *cassandra.thrift*. This file represents the Remote Procedure Call (RPC) client API that Cassandra makes available. The interface is defined using the Thrift syntax and provides an easy means to generate clients. For a quick way to see all of the operations that Cassandra supports, open this file in a regular text editor. You can see that Cassandra supports clients for Java, C++, PHP, Ruby, Python, Perl, and C# through this interface.

javadoc
: This directory contains a documentation website generated using Java's JavaDoc tool. Note that JavaDoc reflects only the comments that are stored directly in the Java code, and as such does not represent comprehensive documentation. It's helpful if you want to see how the code is laid out. Moreover, Cassandra is a

wonderful project, but the code contains relatively few comments, so you might find the JavaDoc's usefulness limited. It may be more fruitful to simply read the class files directly if you're familiar with Java. Nonetheless, to read the JavaDoc, open the *javadoc/index.html* file in a browser.

lib
: This directory contains all of the external libraries that Cassandra needs to run. For example, it uses two different JSON serialization libraries, the Google collections project, and several Apache Commons libraries. This directory includes the Thrift RPC library for interacting with Cassandra.

pylib
: This directory contains Python libraries that are used by *cqlsh*.

tools
: This directory contains tools which are used to maintain your Cassandra nodes. For example, `sstable2json` and `json2sstable` convert Cassandra data files to JSON and back, which can be useful for archive and restore operations or performing a version upgrade. We'll look at these tools in ???.

Additional Directories

If you've already run Cassandra using the default configuration, you will notice two additional directories under the main Cassandra directory: *data* and *log*. We'll discuss the contents of these directories below.

Building from Source

Cassandra uses Apache Ant for its build scripting language and Maven for dependency management.

Downloading Ant

You can download Ant from *http://ant.apache.org*. You don't need to download Maven separately just to build Cassandra.

Building from source requires a complete Java 7 or 8 JDK, not just the JRE. If you see a message about how Ant is missing *tools.jar*, either you don't have the full JDK or you're pointing to the wrong path in your environment variables. Maven downloads files from the internet so if your connection is invalid or Maven cannot determine the proxy, the build will fail.

Downloading Development Builds

If you want to download the most cutting-edge builds, you can get the source from Jenkins, which the Cassandra project uses as its Continuous Integration tool. See *http://cassci.datastax.com* for the latest builds and test coverage information.

If you are a Git fan, you can get a read-only trunk version of the Cassandra source using this command:

```
$ git clone git://git.apache.org/cassandra.git
```

What is Git?

Git is a source code management system created by Linus Torvalds to manage development of the Linux kernel. It's increasingly popular and is used by projects such as Android, Fedora, Ruby on Rails, Perl, and many Cassandra clients (as we'll see in #chp-clients). If you're on a Linux distribution such as Ubuntu, it couldn't be easier to get Git. At a console, just type >*apt-get install git* and it will be installed and ready for commands. For more information, visit *http://git-scm.com/*.

Because Maven takes care of all the dependencies, it's easy to build Cassandra once you have the source. Just make sure you're in the root directory of your source download and execute the `ant` program, which will look for a file called *build.xml* in the current directory and execute the default build target. Ant and Maven take care of the rest. To execute the Ant program and start compiling the source, just type:

```
> ant
```

That's it. Maven will retrieve all of the necessary dependencies, and Ant will build the hundreds of source files and execute the tests. If all went well, you should see a `BUILD SUCCESSFUL` message. If all did not go well, make sure that your path settings are all correct, that you have the most recent versions of the required programs, and that you downloaded a stable Cassandra build. You can check the Jenkins report to make sure that the source you downloaded actually can compile.

More Build Output

If you want to see detailed information on what is happening during the build, you can pass Ant the `-v` option to cause it to output verbose details regarding each operation it performs.

Additional Build Targets

To compile the server, you can simply execute *ant* as shown previously. This command executes the default target, *jar*. This target will perform a complete build including unit tests and output a file into the *build* directory called *apache-cassandra-x.x.x.jar*.

There are a couple of other targets in the build file that you might be interested in:

test
: Users will probably find this the most helpful, as it executes the battery of unit tests. You can also check out the unit test sources themselves for some useful examples of how to interact with Cassandra.

gen-thrift-java
: This target generates the Apache Thrift client interface for interacting with the database in Java.

stress-build
: This target builds the Cassandra stress tool, which we will try out in ???.

clean
: This target removes locally created artifacts such as generated source files and classes and unit test results. The related target *realclean* performs a *clean* and additionally removes the Cassandra distribution jar files and jar files downloaded by Maven.

> If you want to see all of the targets supported by the build file, simply pass Ant the `-p` option to get a description of each target.

Running Cassandra

In earlier versions of Cassandra, before you could start the server there were some required steps to edit configuration files and set environment variables. But the developers have done a terrific job of making it very easy to start using Cassandra immediately. We'll note some of the available configuration options as we go.

Required Java Version

Cassandra requires a Java 7 or 8 JVM, preferably the latest stable version. It has been tested on both the Open JDK and Oracle's JDK. You can check your installed Java version by opening a command prompt and executing `java -version`. If you need a JDK, you can get one at http://www.oracle.com/technetwork/java/javase/downloads/index.html.

On Windows

Once you have the binary or the source downloaded and compiled, you're ready to start the database server.

Setting the `JAVA_HOME` environment variable is recommended. To do this on Windows 7, click the Start button and then right-click on Computer. Click Advanced System Settings, and then click the Environment Variables... button. Click New... to create a new system variable. In the Variable Name field, type `JAVA_HOME`. In the Variable Value field, type the path to your Java installation. This is probably something like *C:\Program Files\Java\jre7* if running Java 7 or *C:\Program Files\Java\jre1.8.0_25.* if running Java 8.

Remember that if you create a new environment variable, you'll need to reopen any currently open terminals in order for the system to become aware of the new variable. To make sure your environment variable is set correctly and that Cassandra can subsequently find Java on Windows, execute this command in a new terminal: `echo %JAVA_HOME%`. This prints the value of your environment variable.

You can also define an environment variable called `CASSANDRA_HOME` that points to the top-level directory where you have placed or built Cassandra, so you don't have to pay as much attention to where you're starting Cassandra from. This is useful for other tools besides the database server, such as `nodetool` and `cassandra-cli`.

Once you've started the server for the first time, Cassandra will add directories to your system to store its data files. The default configuration creates these directories under the `CASSANDRA_HOME` directory.

data
: This directory is where Cassandra stores its data. By default, there are three subdirectories under the *data* directory, corresponding to the various data files Cassandra uses: *commitlog*, *data*, and *saved_caches*. We'll explore the significance of each of these data files in ???. If you've been trying different versions of the database and aren't worried about losing data, you can delete these directories and restart the server as a last resort.

logs
: This directory is where Cassandra stores its logs in a file called *system.log*. If you encounter any difficulties, consult the log to see what might have happened.

Data File Locations

The data file locations are configurable in the *cassandra.yaml* file, located in the *conf* directory. The properties are called `data_file_directories`, `commit_log_directory`, and `saved_caches_directory`. We'll discuss the recommended configuration of these directories in #chp-configuring.

On Linux

The process on Linux and other (*nix operating systems including Mac OS) is similar to that on Windows. Make sure that your `JAVA_HOME` variable is properly set to as described above. Then, you need to extract the Cassandra gzipped tarball using *gunzip*. Many users prefer to use the */var/lib* directory for data storage. If you are changing this configuration you will need to edit the *conf/cassandra.yaml* file and create the referenced directories for Cassandra to store its data and logs, making sure to configure write permissions for the user that will be running Cassandra.

```
$ sudo mkdir -p /var/lib/cassandra
$ sudo chown -R username /var/lib/cassandra
```

Instead of `username`, of course, substitute your own username.

Starting the Server

To start the Cassandra server on any OS, open a command prompt or terminal window, navigate to the *<cassandra-directory>/bin* where you unpacked Cassandra, and run the command `cassandra -f` to start your server.

Starting Cassandra in the Foreground

Using the `-f` switch tells Cassandra to stay in the foreground instead of running as a background process, so that all of the server logs will print to standard out and you can see them in your terminal window, which is useful for testing. In either case, the logs will append to the *system.log* file described above.

In a clean installation, you should see quite a few log statements as the server gets running. Here are a few highlights to look for:

```
INFO  22:20:51 Loading settings from file:/Users/jeff1/
  Cassandra/apache-cassandra-2.1.5/conf/cassandra.yaml
```

```
INFO  22:20:51 Node configuration:[authenticator=
  AllowAllAuthenticator; authorizer=AllowAllAuthorizer; ...
```

These log statements indicate the location of the *cassandra.yaml* file containing the configured settings. The `Node configuration` statement lists out the settings from the config file.

```
INFO  22:20:52 JVM vendor/version: Java HotSpot(TM) 64-Bit Server
  VM/1.8.0_45
INFO  22:20:52 Heap size: 6400507904/6400507904
```

These log statements provide information describing the JVM being used, including memory settings.

```
INFO  22:20:54 Cassandra version: 2.1.5
INFO  22:20:54 Thrift API version: 19.39.0
INFO  22:20:54 CQL supported versions: 2.0.0,3.2.0 (default:
  3.2.0)
```

These log statements identify the server and API versions in use.

```
ERROR 22:20:52 Directory ./../data/data doesn't exist
ERROR 22:20:52 Directory ./../data/commitlog doesn't exist
ERROR 22:20:52 Directory ./../data/saved_caches doesn't exist
INFO  22:20:52 Initializing key cache with capacity of 100 MBs.
INFO  22:20:52 Initializing row cache with capacity of 0 MBs
INFO  22:20:52 Initializing counter cache with capacity of 50 MBs
INFO  22:20:52 Scheduling counter cache save to every 7200 seconds
  (going to save all keys).
INFO  22:20:52 Initializing system.sstable_activity
```

The first time we run our server, the data files Cassandra uses don't yet exist, so the server creates them. The next time we start Cassandra, we'll see log statements indicating the data files that are being used.

```
WARN  22:20:52 JMX is not enabled to receive remote connections.
  Please see cassandra-env.sh for more info.
INFO  22:20:54 Starting up server gossip
INFO  22:20:54 Starting Messaging Service on port 7000
```

These log statements indicate the server is beginning to initiate communications with other servers in the cluster and expose publicly available interfaces. By default, the management interface via the Java Management Extensions (JMX) is disabled. We'll explore the management interface in ???.

```
INFO  22:20:54 This node will not auto bootstrap because it is
  configured to be a seed node.
```

By default, Cassandra is configured to run as a single node cluster. The capability to discover other nodes in the cluster known as *auto bootstrapping* is disabled because the single node is designated as a *seed node*. We'll see how to change this in ???.

```
INFO  22:20:55 Node localhost/127.0.0.1 state jump to normal
INFO  22:20:55 Compacted 4 sstables to [./../data/data/system/
  local-7ad54392bcdd35a684174e047860b377/system-local-ka-5,].
  5,883 bytes to 5,722 (~97% of original) in 150ms = 0.036379MB/s.
  4 total partitions merged to 1.  Partition merge counts were
  {4:1, }
```

Finally, if everything is working correctly you'll see the server transition to a normal operating state. If you continue to monitor the output, you'll begin to see periodic compactions being logged. We'll describe this behavior in TBD.

Congratulations! Now your Cassandra server should be up and running with a new single node cluster called Test Cluster listening on port 9160.

Starting Over

The committers work hard to ensure that data is readable from one minor dot release to the next and from one major version to the next. The commit log, however, needs to be completely cleared out from version to version (even minor versions).

If you have any previous versions of Cassandra installed, you may want to clear out the data directories for now, just to get up and running. If you've messed up your Cassandra installation and want to get started cleanly again, you can delete the data folders.

Stopping Cassandra

Now that we've successfully started a Cassandra server, you may be wondering how to stop it. You may have noticed the `stop-server` command in the `bin` directory. Let's try running that command. Here's what you'll see on Unix systems:

```
$ ./stop-server
please read the stop-server script before use
```

So you see that our server has not been stopped, but instead we are directed to read the script. Taking a look inside with our favorite code editor, you'll learn that the way to stop Cassandra is to kill the JVM process that is running Cassandra. The file suggests a couple of different techniques by which you can identify the JVM process and kill it.

The first technique is to start Cassandra using the `-p` option, which provides Cassandra with the name of a file to which it should write the process identifier (PID) upon starting up. This is arguably the most straightforward approach to making sure we kill the right process.

However, since we did not start Cassandra with he `-p` option, we'll need to find the process ourselves and kill it. The script suggests using pgrep to locate processes for the current user containing the term "cassandra":

```
user=`whoami`
pgrep -u $user -f cassandra | xargs kill -9
```

Stopping Cassandra on Windows

On Windows installations, you can find the JVM process and kill it using the Task Manager.

Other Cassandra Distributions

The instructions above showed us how to install the Apache distribution of Cassandra. In addition to the Apache distribution, there are a couple of other ways to get Cassandra:

DataStax Community Edition
: This free distribution is provided by DataStax via the Planet Cassandra website. Installation options for various platforms include RPM and Debian (Linux), MSI (Windows) and a MacOS library. The community edition provides additional tools including an integrated development environment (IDE) known as DevCenter, and the OpsCenter monitoring tool. Another useful feature is the ability to configure Cassandra as an OS-managed service on Windows. Releases of the community edition generally track the Apache releases, with availability soon after each Apache release.

DataStax Enterprise Edition
: DataStax also provides a fully supported version certified for production use. The product line provides an integrated database platform with support for complementary data technologies such as Hadoop and Apache Spark. We'll explore some of these integrations in ???.

Virtual Machine Images
: An frequent model for deployment of Cassandra is to package one of the above distributions in a virtual machine image. For example, multiple such images are available in the Amazon Web Services (AWS) Marketplace.

We'll take a deeper look at several options for deploying Cassandra in production environments, including cloud computing environments, in ???.

Selecting the right distribution will depend on your deployment environment, your needs for scale, stability and support, and your development and maintenance budgets. Having both open source and commercial deployment options provides the flexibility to make the right choice for your organization.

Running the CQL Shell

Now that you have a Cassandra installation up and running, let's give it a quick try to make sure everything is set up properly. We'll use the CQL shell (`cqlsh`) to connect to our server and have a look around.

Deprecation of the CLI

If you've used Cassandra in releases prior to 3.0, you may also be familiar with the command line client interface known as `cassandra-cli`. The CLI is removed in the 3.0 release because it depends on the Thrift API, which is also removed in 3.0 in favor of the native CQL interface, as discussed further in #chp-clients.

To run the shell, create a new terminal window, change to the Cassandra home directory, and type the command:

```
$ bin/cqlsh
Connected to Test Cluster at 127.0.0.1:9042.
[cqlsh 5.0.1 | Cassandra 2.1.5 | CQL spec 3.2.0 |
  Native protocol v3]
Use HELP for help.
cqlsh>
```

Because we did not specify a node to which we wanted to connect, the shell helpfully checks for a node running on the local host, and finds the node we started earlier. The shell also indicates that you're connected to a Cassandra server cluster called "Test Cluster". That's because this cluster of one node at `localhost` is set up for you by default.

Renaming the Default Cluster

In a production environment, be sure to remove the Test Cluster from the configuration.

To connect to a specific node, specify the host name and port on the command line. For example the following will connect to our local node:

```
$ bin/cqlsh localhost 9042
Connected to Test Cluster at localhost:9042.
[cqlsh 5.0.1 | Cassandra 2.1.5 | CQL spec 3.2.0 |
  Native protocol v3]
Use HELP for help.
cqlsh>
```

Another alternative for configuring the *cqlsh* connection is to set the environment variables `$CQLSH_HOST` and `$CQLSH_PORT`. This approach is useful if you will be frequently connecting to a specific node on another host. The environment variables will be overriden if you specify the host and port on the command line.

Connection Errors

If you see an error like this while trying to connect to a server:

```
Exception connecting to localhost/9160. Reason:
  Connection refused.
```

make sure that a Cassandra instance is started at that host and port and that you can ping the host you're trying to reach. There may be firewall rules preventing you from connecting.

To see a complete list of the command line options supported by *cqlsh*, type the command `cqlsh -help`.

Basic CQLSH Commands

Let's take a quick tour of *cqlsh* to learn what kinds of commands you can send to the server. We'll see how to use the basic environment commands and how to do a round trip of inserting and retrieving some data.

Case in CQLSH

The *cqlsh* commands are all case insensitive. For our examples we'll adopt the convention of uppercase to be consistent with the way the shell describes its own commands in help topics and output.

CQLSH Help

To get help for *cqlsh*, type `HELP` or `?` to see the list of available commands.

```
cqlsh> HELP
Documented shell commands:
===========================
CAPTURE      COPY  DESCRIBE  EXPAND  PAGING  SOURCE
CONSISTENCY  DESC  EXIT      HELP    SHOW    TRACING

CQL help topics:
================
ALTER                        CREATE_TABLE_OPTIONS  SELECT
ALTER_ADD                    CREATE_TABLE_TYPES    SELECT_
```

```
  COLUMNFAMILY
ALTER_ALTER                  CREATE_USER          SELECT_EXPR
ALTER_DROP                   DELETE               SELECT_LIMIT
ALTER_RENAME                 DELETE_COLUMNS       SELECT_TABLE
ALTER_USER                   DELETE_USING         SELECT_WHERE
ALTER_WITH                   DELETE_WHERE         TEXT_OUTPUT
APPLY                        DROP                 TIMESTAMP_INPUT
ASCII_OUTPUT                 DROP_COLUMNFAMILY    TIMESTAMP_OUTPUT
BEGIN                        DROP_INDEX           TRUNCATE
BLOB_INPUT                   DROP_KEYSPACE        TYPES
BOOLEAN_INPUT                DROP_TABLE           UPDATE
COMPOUND_PRIMARY_KEYS        DROP_USER            UPDATE_COUNTERS
CREATE                       GRANT                UPDATE_SET
CREATE_COLUMNFAMILY          INSERT               UPDATE_USING
CREATE_COLUMNFAMILY_OPTIONS  LIST                 UPDATE_WHERE
CREATE_COLUMNFAMILY_TYPES    LIST_PERMISSIONS     USE
CREATE_INDEX                 LIST_USERS           UUID_INPUT
CREATE_KEYSPACE              PERMISSIONS
CREATE_TABLE                 REVOKE
```

CQLSH Help Topics

You'll notice that the help topics listed differ slightly from the actual command syntax. For example, the `CREATE_TABLE` help topic describes how to use the syntax > `CREATE TABLE ....`

To get additional documentation about a particular command, type `HELP <command>`.

Many *cqlsh* commands may be used with no parameters, in which case they print out the current setting. Examples include *consistency*, *expand*, and *paging*.

Describing the Environment in CQLSH

After connecting to your Cassandra instance Test Cluster, if you're using the binary distribution, an empty *keyspace*, or Cassandra database, is set up for you to test with.

To learn about the current cluster you're working in, type:

```
cqlsh> DESCRIBE CLUSTER
Cluster: Test Cluster
Partitioner: Murmur3Partitioner
```

To see which keyspaces are available in the cluster, issue this command:

```
cqlsh> DESCRIBE KEYSPACES
system  system_traces
```

Initially this list will consist of two keyspaces: `system` and `system_traces`. Once you have created your own keyspaces, they will be shown as well. The `system` keyspace is used internally by Cassandra, and isn't for us to put data into. In this way, it's similar to the master and temp databases in Microsoft SQL Server. This keyspace contains the schema definitions and is aware of any modifications to the schema made at runtime. It can propagate any changes made in one node to the rest of the cluster based on timestamps. Similarly, the `system_traces` keyspace is used internally by Cassandra to provide tracing of read and write requests.

You can use the following command to learn the client, server, and protocol versions in use:

```
cqlsh> SHOW VERSION
[cqlsh 5.0.1 | Cassandra 2.1.5 | CQL spec 3.2.0 |
  Native protocol v3]
```

You may have noticed that this version info is printed out when *cqlsh* starts. There are a variety of other commands with which you can experiment. For now, let's add some data to the database and get it back out again.

Creating a Keyspace and Table in CQLSH

A Cassandra keyspace is sort of like a relational database. It defines one or more tables or "column families". When you start *cqlsh* without specifying a keyspace, the prompt will look like this: `cqlsh>`, with no keyspace specified.

Let's create our own keyspace so we have something to write data to. In creating our keyspace, there are some required options. To walk through these options, we could use the command `HELP CREATE_KEYSPACE`, but for this time we'll use the helpful command completion features of *cqlsh*. Type the following and then hit the <TAB> key:

```
cqlsh> CREATE KEYSPACE my_keyspace WITH
```

When you hit the <TAB> key, *cqlsh* begins completing the syntax of our command:

```
cqlsh> CREATE KEYSPACE my_keyspace WITH replication = {'class': '
```

This is informing us that in order to specify a keyspace, we also need to specify a replication strategy. Let's <TAB> again to see what options we have:

```
cqlsh> CREATE KEYSPACE my_keyspace WITH replication = {'class': '
  NetworkTopologyStrategy    SimpleStrategy
  OldNetworkTopologyStrategy
```

Now *cqlsh* is giving us three strategies to choose from. We'll learn more about these strategies in ???. For now we will choose the `SimpleStrategy` by typing the name. We'll indicate we're done with a close quote and <TAB> again:

```
cqlsh> CREATE KEYSPACE my_keyspace WITH replication = {'class':
  'SimpleStrategy', 'replication_factor':
```

The next option we're presented with is a replication factor. For the simple strategy, this indicates how many nodes the data in this keyspace will be written to. For a production deployment, we'd want copies of our data stored on multiple nodes, but since we're just running a single node at the moment, we'll ask for a single copy. Let's specify a value of "1" and <TAB> again:

```
cqlsh> CREATE KEYSPACE my_keyspace WITH replication = {'class':
  'SimpleStrategy', 'replication_factor': 1}
```

We see that *cqlsh* has now added a close bracket, indicating we've completed all of the required options. Let's complete our command with a semicolon and return, and our keyspace will be created.

Keyspace Creation Options

For a production keyspace, we would probably never want to use a value of 1 for the There are additional options on creating a keyspace depending on the replication strategy that is chosen. The command completion feature will walk through the different options.

Let's have a look at our keyspace using the `DESCRIBE KEYSPACE` command:

```
cqlsh> DESCRIBE KEYSPACE my_keyspace
CREATE KEYSPACE my_keyspace WITH replication = {'class':
  'SimpleStrategy', 'replication_factor': '3'}  AND
  durable_writes = true;
```

We see that the table has been created with the *SimpleStrategy*, a `replication_factor` of one, and durable writes, Notice that our keyspace is described in much the same syntax that we used to create it, with one additional option that we did not specify: `durable_writes=true`. Don't worry about these settings now, we'll look at them in detail later.

After you have created your own keyspace, you can switch to it in the shell by typing:

```
cqlsh> USE my_keyspace;
cqlsh:my_keyspace>
```

Notice that the prompt has changed to indicate that we're using the keyspace.

Using Snake Case

You may have wondered why we chose to name our keyspace in "snake case" (`my_keyspace`) as opposed to the "camel case" (`MyKeyspace`) which is familiar to developers using Java and other languages.

As it turns out, Cassandra naturally handles keyspace, table and column names as lowercase. When you enter names in mixed case, Cassandra stores them as all lower case.

This behavior can be overridden by enclosing your names in double quotes, for example `CREATE KEYSPACE "MyKeyspace"....` However, it tends to be a lot simpler to use snake case than to go against the grain.

Now that we have a keyspace, we can create a table in our keyspace. To do this in *cqlsh*, use the following command:

```
cqlsh:my_keyspace> CREATE TABLE user ( first_name text ,
  last_name text, PRIMARY KEY (first_name)) ;
```

This creates a new table called "user" in our current keyspace with two columns to store first and last names, both of type text. The `text` and `varchar` types are synonymous and are used to store strings. We've specified the `first_name` column as our primary key and taken the defaults for other table options.

Using Keyspace Names in CQLSH

We could also have created this table without switching to our keyspace by using the syntax: `CREATE TABLE my_keyspace.user (....`

We can use *cqlsh* to get a description of a the table we just created using the `DESCRIBE TABLE` command:

```
cqlsh:my_keyspace> DESCRIBE TABLE user;
CREATE TABLE my_keyspace.user (
    first_name text PRIMARY KEY,
    last_name text
) WITH bloom_filter_fp_chance = 0.01
    AND caching = '{"keys":"ALL", "rows_per_partition":"NONE"}'
    AND comment = ''
    AND compaction = {'min_threshold': '4', 'class':
      'org.apache.cassandra.db.compaction.
      SizeTieredCompactionStrategy', 'max_threshold': '32'}
    AND compression = {'sstable_compression':
      'org.apache.cassandra.io.compress.LZ4Compressor'}
    AND dclocal_read_repair_chance = 0.1
    AND default_time_to_live = 0
    AND gc_grace_seconds = 864000
    AND max_index_interval = 2048
    AND memtable_flush_period_in_ms = 0
    AND min_index_interval = 128
    AND read_repair_chance = 0.0
    AND speculative_retry = '99.0PERCENTILE';
```

You'll notice that *cqlsh* prints a nicely formattted version of the `CREATE TABLE` command that we just typed in but also includes values for all of the available table options that we did not specify. These values are the defaults that were set since we did not specify them. We'll worry about these settings later. For now, we have enough to get started.

Writing and Reading Data in CQLSH

Now that we have a keyspace and a table, we'll write some data to the database and read it back out again. It's OK at this point not to know quite what's going on. We'll come to understand Cassandra's data model in depth later. For now, you have a keyspace (database), which has a table, which holds columns, the atomic unit of data storage.

To write a value, use the `INSERT` command:

```
cqlsh:my_keyspace> INSERT INTO user (first_name, last_name )
  VALUES ('Bill', 'Nguyen') ;
```

Here we have created a new row with two columns for the key `Bill`, to store a set of related values. The column names are `first_name` and `last_name`. We can use the `SELECT COUNT` command to make sure that the row was written:

```
cqlsh:my_keyspace> SELECT COUNT (*) FROM user;
 count
-------
     1

(1 rows)
```

Now that we know the data is there, let's read it, using the `SELECT` command:

```
cqlsh:my_keyspace> SELECT * FROM user WHERE first_name='Bill';

 first_name | last_name
------------+-----------
       Bill |    Nguyen

(1 rows)
```

In this command we requested to return rows matching the primary key `Bill` including all columns. You can delete a column using the DELETE command. Here we will delete the `last_name` column for the `Bill` row key:

```
cqlsh:my_keyspace> DELETE last_name FROM USER WHERE
  first_name='Bill';
```

To make sure that it's removed, we can query again:

```
cqlsh:my_keyspace> SELECT * FROM user WHERE first_name='Bill';
```

```
 first_name | last_name
------------+-----------
       Bill |      null

(1 rows)
```

Now we'll clean up after ourselves by deleting the entire row. It's the same command, but we don't specify a column name:

```
cqlsh:my_keyspace> DELETE FROM USER WHERE first_name='Bill';
```

To make sure that it's removed, we can query again:

```
cqlsh:my_keyspace> SELECT * FROM user WHERE first_name='Bill';

 first_name | last_name
------------+-----------

(0 rows)
```

CQLSH Command History

Now that you've been using *cqlsh* for a while, you may nave noticed that you can navigate through commands you've executed previously with the up and down arrow key. This history is stored your home directory for a hidden directory called *.cassandra*, in a file called *cqlsh_history*. This acts like your Bash shell history, listing the commands in a plain-text file in the order Cassandra executed them. Nice!

Summary

Now you should have a Cassandra installation up and running. You've worked with the *cqlsh* client to insert and retrieve some data, and you're ready to take a step back and get the big picture on Cassandra before really diving into the details.