O'REILLY®

Compliments of MESOSPHERE

2nd Edition

FREE CHAPTERS

Cassandra The Definitive Guide

DISTRIBUTED DATA AT WEB SCALE

Jeff Carpenter & Eben Hewitt

THE BEST PLACE TO RUN CASSANDRA WITH CONTAINERS.

Mesosphere DC/OS makes it easy to build and elastically scale data-rich applications.

- 1-Click Install & Operation of Data Services
- Run on Any Cloud, Public or Private
- Secure & Proven in Production
- Dramatically Cuts Infrastructure Costs

LEARN MORE →

MESOSPHERE

SECOND EDITION

Cassandra: The Definitive Guide

This Excerpt contains Chapters 1, 2, 6, and 14 of the book *Cassandra: The Definitive Guide*, 2nd Edition. The complete book is available at oreilly.com and through other retailers.

Jeff Carpenter and Eben Hewitt

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

Cassandra: The Definitive Guide
by Jeff Carpenter and 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://oreilly.com/safari*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Mike Loukides and Marie Beaugureau
Production Editor: Colleen Cole
Copyeditor: Jasmine Kwityn
Proofreader: James Fraleigh
Indexer: Ellen Troutman-Zaig
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

June 2016: Second Edition

Revision History for the Second Edition

2010-11-12: First Release
2016-06-27: Second Release
2017-04-07: Third Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491933664* for release details.

978-1-491-99565-5

[LSI]

Table of Contents

Foreword

Data is truly the new oil. The ability to easily access tremendous amounts of computing power has made data the new basis of competition. We've moved beyond traditional big data, where companies gained historical insights using batch analytics—in today's digital economy, businesses must learn to extract value from data and build modern applications that serve customers with personalized services, in real time, and at scale.

At Mesosphere, we are seeing two major technology trends that enable modern applications to handle users and data at scale: containers and data services. Microservice-based applications that are deployed in containers speed time to market and reduce infrastructure overhead, among other benefits. Data services capture, process, and store data being used by the containerized microservices.

A de facto architecture is emerging to build and operate fast data applications, with a set of leading technologies used within this architecture that are scalable, enable real-time processing, and open source. This set of technologies is often referred to as the "SMACK" stack:

- Apache Kafka™: A distributed, highly available messaging system to ensure millions of events per second are captured through connected endpoints with no loss
- Apache Spark™: A large-scale analytics engine that supports streaming, machine learning, SQL, and graph computation
- Apache Cassandra™: A distributed database that is highly available and scalable
- Akka: A toolkit and runtime to simplify development of data-driven apps
- Apache Mesos™: A cluster resource manager that serves as a highly available, scalable, and efficient platform for running data services and containerized microservices.

Cassandra: The Definitive Guide describes how and why to apply Cassandra in your application in a production environment. It describes the recent rise of non-relational database technologies like Cassandra, outlines Cassandra's distributed structure, explains how to integrate Cassandra with other technologies including Spark, and provides practical examples.

Mesosphere is proud to offer this excerpt, as we were founded with the goal of making cutting edge technologies such as Cassandra and the SMACK stack easy to use. Mesos serves as an elastic and proven foundation for building and elastically scaling data-rich, modern applications. We created DC/OS to make Mesos simple to use, including the automation of data services.

We hope you enjoy the excerpt, and that you consider Mesosphere DC/OS to jump start your journey to building, deploying, and scaling a data-intensive application that helps your business.

— Tobi Knaup
Chief Technology Officer, Mesosphere

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. During the course of this book, we will explore how Cassandra 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

We 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, we are 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 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, including "relations," "tuples," and "normal form," all of which must have sounded very strange indeed to users of IMS. 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, RDBMSs, 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 violates 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 12 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.

This likely sounds familiar to you. At web scale, engineers may legitimately ponder 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.

RDBMSs: 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 RDBMSs 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, RDBMSs 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 "The Transaction Concept: Virtues and Limitations").

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 associated 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. 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 the site's 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 the University of California at Berkeley in his 1986 paper "The Case for Shared Nothing."

Shared-nothing architecture 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 non-relational 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 RDBMSs. 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 RDBMSs 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 RDBMSs, it is not always as easy to scale as some other, more recent possibilities that take the structure of the Web into consideration. However, 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 real 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 has 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 just discussed, and many have become mature technologies with large installation bases and commercial support. We're 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 our 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. You'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, 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 this theorem in more depth in "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."[1]

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

1 "Dynamo: Amazon's Highly Distributed Key-Value Store", 207.

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 the 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
: 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.

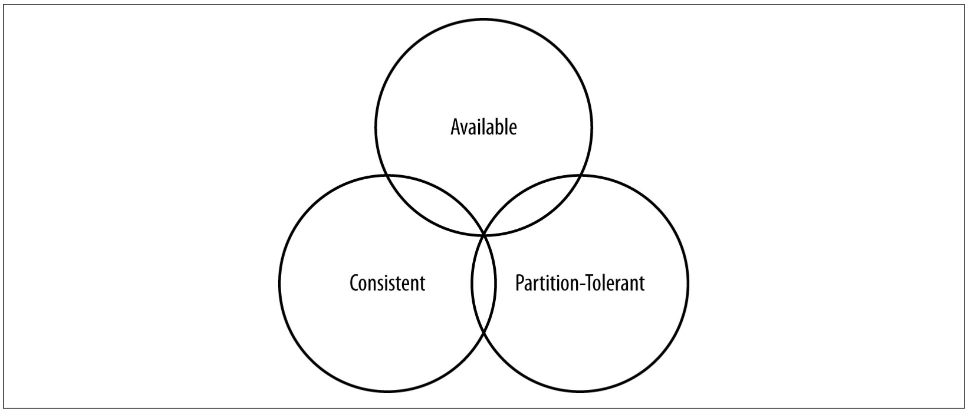

Figure 2-1. CAP theorem indicates that you can realize only two of these properties at once

It might prove useful at this point to see a graphical depiction of where each of the non-relational 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. However, we have modified the placement of some systems based on research.

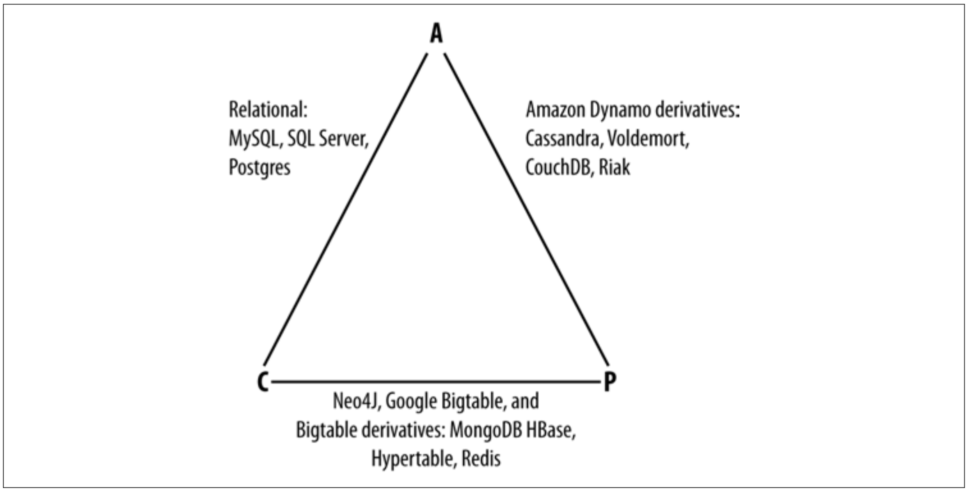

Figure 2-2. Where different databases appear on the CAP continuum

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.

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.

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

In February 2012, Eric Brewer provided an updated perspective on his CAP theorem in the article "CAP Twelve Years Later: How the 'Rules' Have Changed" in IEEE's *Computer*. 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 Chapter 3. 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 Versus Column-Oriented

Cassandra has frequently been referred to as a "column-oriented" database, which has proved to be 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.

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 that supported dynamic column creation has 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. Cassandra originated at Facebook in 2007 to solve its inbox search problem—the company 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 updatable 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. On the Apache Cassandra Wiki, you can find a list of the 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 Decentralized Structured Storage System" by Facebook's Lakshman and Malik was a central paper on Cassandra. 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. We'll unpack some of these changes in more detail in "Release History" on page 30.

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. We 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 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. 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 MapReduce
- 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 0.8.10 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 several 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 I/O 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 importing and exporting comma-separated values (CSV) files via `cqlsh`
- Flexible data storage settings allow the storage of data in SSDs or magnetic storage, selectable by table

- The schema update mechanism was reimplemented 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. 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

The original release plan outlined by the Cassandra developers did not contain a 2.2 release. The intent was to do some major "under the covers" rework for a 3.0 release to follow the 2.1 series. However, due to the amount and complexity of the changes involved, it was decided to release some of completed features separately in order to make them available while allowing some of the more complex changes time to mature. Release 2.2.0 became available in July 2015, and support releases are scheduled through fall 2016. Notable features and improvements in this series 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 I/O 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

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 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 fixes, with the goal of releasing each month.

Release 3.0 (Feature release - November 2015)
- The underlying storage engine was rewritten to more closely match CQL constructs
- Support for materialized views (sometimes also called global indexes) was added
- Java 8 is now the supported version
- The Thrift-based command-line interface (CLI) was removed

Release 3.1 (Bug fix release - December 2015)

Release 3.2 (Feature release - January 2016)
- The way in which Cassandra allocates SSTable file storage across multiple disk in “just a bunch of disks” or JBOD configurations was reworked to improve reliability and performance and to enable backup and restore of individual disks
- The ability to compress and encrypt hints was added

Release 3.3 (Bug fix release - February 2016)

Release 3.4 (Feature release - March 2016)
- `SSTableAttachedSecondaryIndex`, or “SASI” for short, is an implementation of Cassandra’s `SecondaryIndex` interface that can be used as an alternative to the existing implementations.

Release 3.5 (Bug fix release - April 2016)

The 4.0 release series is scheduled to begin in Fall 2016.

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

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. You can see the officially supported versions on the project's download page.

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 pertain 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.

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 semitruck 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 RDBMSs 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 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. 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 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 provides similar technical content, but has a greater focus on profiling companies using Cassandra.

The Apache Cassandra Wiki 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.

Training and conferences
: DataStax offers online training, 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.

A Marketable Skill

There continues to be increased demand for Cassandra developers and administrators. A 2015 Dice.com salary survey placed Cassandra as the second most highly compensated skill set.

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

The Cassandra Architecture

3.2 Architecture - fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution.
—ISO/IEC/IEEE 42010

In this chapter, we examine several aspects of Cassandra's architecture in order to understand how it does its job. We'll explain the topology of a cluster, and how nodes interact in a peer-to-peer design to maintain the health of the cluster and exchange data, using techniques like gossip, anti-entropy, and hinted handoff. Looking inside the design of a node, we examine architecture techniques Cassandra uses to support reading, writing, and deleting data, and examine how these choices affect architectural considerations such as scalability, durability, availability, manageability, and more. We also discuss Cassandra's adoption of a Staged Event-Driven Architecture, which acts as the platform for request delegation.

As we introduce these topics, we also provide references to where you can find their implementations in the Cassandra source code.

Data Centers and Racks

Cassandra is frequently used in systems spanning physically separate locations. Cassandra provides two levels of grouping that are used to describe the topology of a cluster: data center and rack. A *rack* is a logical set of nodes in close proximity to each other, perhaps on physical machines in a single rack of equipment. A *data center* is a logical set of racks, perhaps located in the same building and connected by reliable network. A sample topology with multiple data centers and racks is shown in Figure 3-1.

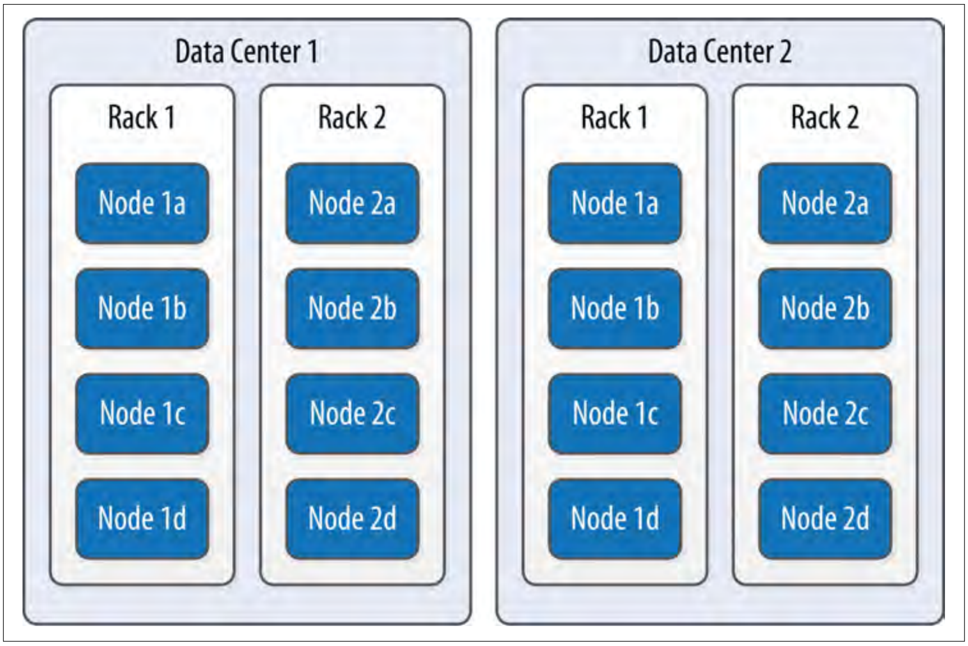

Figure 3-1. Topology of a sample cluster with data centers, racks, and nodes

Out of the box, Cassandra comes with a default configuration of a single data center (`"DC1"`) containing a single rack (`"RAC1"`).

Cassandra leverages the information you provide about your cluster's topology to determine where to store data, and how to route queries efficiently. Cassandra tries to store copies of your data in multiple data centers to maximize availability and partition tolerance, while preferring to route queries to nodes in the local data center to maximize performance.

Gossip and Failure Detection

To support decentralization and partition tolerance, Cassandra uses a gossip protocol that allows each node to keep track of state information about the other nodes in the cluster. The gossiper runs every second on a timer.

Gossip protocols (sometimes called "epidemic protocols") generally assume a faulty network, are commonly employed in very large, decentralized network systems, and are often used as an automatic mechanism for replication in distributed databases. They take their name from the concept of human gossip, a form of communication in which peers can choose with whom they want to exchange information.

The Origin of "Gossip Protocol"

The term "gossip protocol" was originally coined in 1987 by Alan Demers, a researcher at Xerox's Palo Alto Research Center, who was studying ways to route information through unreliable networks.

The gossip protocol in Cassandra is primarily implemented by the `org.apache.cassandra.gms.Gossiper` class, which is responsible for managing gossip for the local node. When a server node is started, it registers itself with the gossiper to receive endpoint state information.

Because Cassandra gossip is used for failure detection, the `Gossiper` class maintains a list of nodes that are alive and dead.

Here is how the gossiper works:

1. Once per second, the gossiper will choose a random node in the cluster and initialize a gossip session with it. Each round of gossip requires three messages.
2. The gossip initiator sends its chosen friend a `GossipDigestSynMessage`.
3. When the friend receives this message, it returns a `GossipDigestAckMessage`.
4. When the initiator receives the `ack` message from the friend, it sends the friend a `GossipDigestAck2Message` to complete the round of gossip.

When the gossiper determines that another endpoint is dead, it "convicts" that endpoint by marking it as dead in its local list and logging that fact.

Cassandra has robust support for failure detection, as specified by a popular algorithm for distributed computing called Phi Accrual Failure Detection. This manner of failure detection originated at the Advanced Institute of Science and Technology in Japan in 2004.

Accrual failure detection is based on two primary ideas. The first general idea is that failure detection should be flexible, which is achieved by decoupling it from the application being monitored. The second and more novel idea challenges the notion of traditional failure detectors, which are implemented by simple "heartbeats" and decide whether a node is dead or not dead based on whether a heartbeat is received or not. But accrual failure detection decides that this approach is naive, and finds a place in between the extremes of dead and alive—a *suspicion level*.

Therefore, the failure monitoring system outputs a continuous level of "suspicion" regarding how confident it is that a node has failed. This is desirable because it can take into account fluctuations in the network environment. For example, just because one connection gets caught up doesn't necessarily mean that the whole node is dead. So suspicion offers a more fluid and proactive indication of the weaker or stronger possibility of failure based on interpretation (the sampling of heartbeats), as opposed to a simple binary assessment.

Phi Threshold and Accrual Failure Detectors

Accrual Failure Detectors output a value associated with each process (or node). This value is called Phi. The value is output in a manner that is designed from the ground up to be adaptive in the face of volatile network conditions, so it's not a binary condition that simply checks whether a server is up or down.

The Phi convict threshold in the configuration adjusts the sensitivity of the failure detector. Lower values increase the sensitivity and higher values decrease it, but not in a linear fashion.

The Phi value refers to a level of *suspicion* that a server might be down. Applications such as Cassandra that employ an AFD can specify variable conditions for the Phi value they emit. Cassandra can generally detect a failed node in about 10 seconds using this mechanism.

You can read the original Phi Accrual Failure Detection paper by Naohiro Hayashibara et al. at *http://www.jaist.ac.jp/~defago/files/pdf/IS_RR_2004_010.pdf*.

Failure detection is implemented in Cassandra by the `org.apache.cassandra.gms.FailureDetector` class, which implements the `org.apache.cassandra.gms.IFailureDetector` interface. Together, they allow operations including:

`isAlive(InetAddress)`
: What the detector will report about a given node's alive-ness.

`interpret(InetAddress)`
: Used by the gossiper to help it decide whether a node is alive or not based on suspicion level reached by calculating Phi (as described in the Hayashibara paper).

`report(InetAddress)`
: When a node receives a heartbeat, it invokes this method.

Snitches

The job of a snitch is to determine relative host proximity for each node in a cluster, which is used to determine which nodes to read and write from. Snitches gather information about your network topology so that Cassandra can efficiently route requests. The snitch will figure out where nodes are in relation to other nodes.

As an example, let's examine how the snitch participates in a read operation. When Cassandra performs a read, it must contact a number of replicas determined by the consistency level. In order to support the maximum speed for reads, Cassandra selects a single replica to query for the full object, and asks additional replicas for hash values in order to ensure the latest version of the requested data is returned. The role of the snitch is to help identify the replica that will return the fastest, and this is the replica which is queried for the full data.

The default snitch (the `SimpleSnitch`) is topology unaware; that is, it does not know about the racks and data centers in a cluster, which makes it unsuitable for multi-data center deployments. For this reason, Cassandra comes with several snitches for different cloud environments including Amazon EC2, Google Cloud, and Apache Cloudstack.

The snitches can be found in the package `org.apache.cassandra.locator`. Each snitch implements the `IEndpointSnitch` interface.

While Cassandra provides a pluggable way to statically describe your cluster's topology, it also provides a feature called *dynamic snitching* that helps optimize the routing of reads and writes over time. Here's how it works. Your selected snitch is wrapped with another snitch called the `DynamicEndpointSnitch`. The dynamic snitch gets its basic understanding of the topology from the selected snitch. It then monitors the performance of requests to the other nodes, even keeping track of things like which nodes are performing compaction. The performance data is used to select the best replica for each query. This enables Cassandra to avoid routing requests to replicas that are performing poorly.

The dynamic snitching implementation uses a modified version of the Phi failure detection mechanism used by gossip. The "badness threshold" is a configurable parameter that determines how much worse a preferred node must perform than the best-performing node in order to lose its preferential status. The scores of each node are reset periodically in order to allow a poorly performing node to demonstrate that it has recovered and reclaim its preferred status.

Rings and Tokens

So far we've been focusing on how Cassandra keeps track of the physical layout of nodes in a cluster. Let's shift gears and look at how Cassandra distributes data across these nodes.

Cassandra represents the data managed by a cluster as a *ring*. Each node in the ring is assigned one or more ranges of data described by a *token*, which determines its position in the ring. A token is a 64-bit integer ID used to identify each partition. This gives a possible range for tokens from -2^{63} to $2^{63}-1$.

A node claims ownership of the range of values less than or equal to each token and greater than the token of the previous node. The node with lowest token owns the range less than or equal to its token and the range greater than the highest token, which is also known as the "wrapping range." In this way, the tokens specify a complete ring. Figure 3-2 shows a notional ring layout including the nodes in a single data center. This particular arrangement is structured such that consecutive token ranges are spread across nodes in different racks.

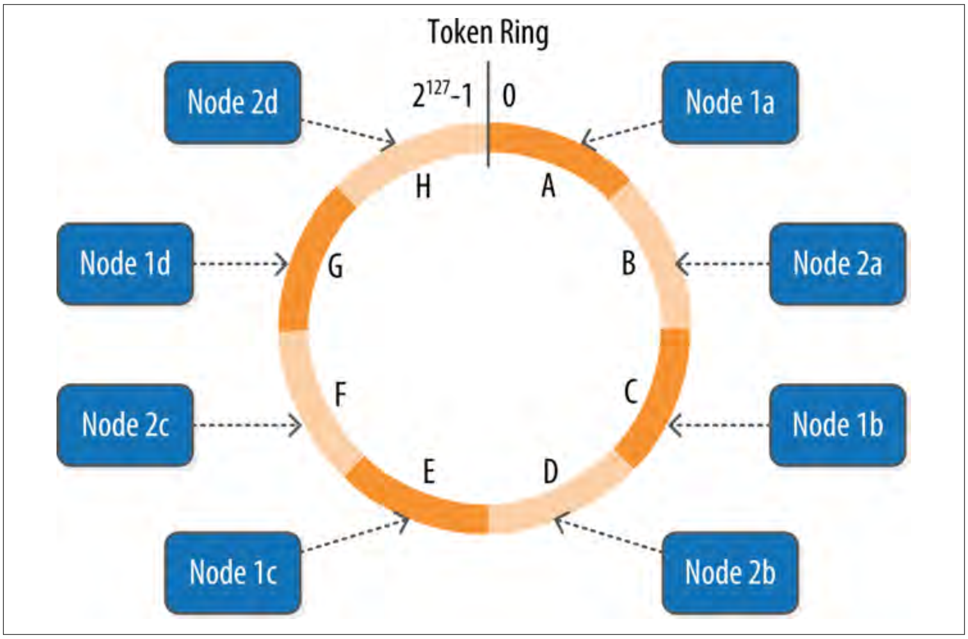

Figure 3-2. Example ring arrangement of nodes in a data center

Data is assigned to nodes by using a hash function to calculate a token for the partition key. This partition key token is compared to the token values for the various nodes to identify the range, and therefore the node, that owns the data.

Token ranges are represented by the `org.apache.cassandra.dht.Range` class.

Virtual Nodes

Early versions of Cassandra assigned a single token to each node, in a fairly static manner, requiring you to calculate tokens for each node. Although there are tools available to calculate tokens based on a given number of nodes, it was still a manual process to configure the `initial_token` property for each node in the *cassandra.yaml* file. This also made adding or replacing a node an expensive operation, as rebalancing the cluster required moving a lot of data.

Cassandra's 1.2 release introduced the concept of *virtual nodes*, also called *vnodes* for short. Instead of assigning a single token to a node, the token range is broken up into multiple smaller ranges. Each physical node is then assigned multiple tokens. By default, each node will be assigned 256 of these tokens, meaning that it contains 256 virtual nodes. Virtual nodes have been enabled by default since 2.0.

Vnodes make it easier to maintain a cluster containing heterogeneous machines. For nodes in your cluster that have more computing resources available to them, you can increase the number of vnodes by setting the `num_tokens` property in the *cassandra.yaml* file. Conversely, you might set `num_tokens` lower to decrease the number of vnodes for less capable machines.

Cassandra automatically handles the calculation of token ranges for each node in the cluster in proportion to their `num_tokens` value. Token assignments for vnodes are calculated by the `org.apache.cassandra.dht.tokenallocator.ReplicationAwareTokenAllocator` class.

A further advantage of virtual nodes is that they speed up some of the more heavyweight Cassandra operations such as bootstrapping a new node, decommissioning a node, and repairing a node. This is because the load associated with operations on multiple smaller ranges is spread more evenly across the nodes in the cluster.

Partitioners

A *partitioner* determines how data is distributed across the nodes in the cluster. As we have learned, Cassandra stores data in wide rows, or "partitions." Each row has a partition key that is used to identify the partition. A partitioner, then, is a hash function for computing the token of a partition key. Each row of data is distributed within the ring according to the value of the partition key token.

Cassandra provides several different partitioners in the `org.apache.cassandra.dht` package (DHT stands for "distributed hash table"). The `Murmur3Partitioner` was added in 1.2 and has been the default partitioner since then; it is an efficient Java implementation on the murmur algorithm developed by Austin Appleby. It generates 64-bit hashes. The previous default was the `RandomPartitioner`.

Because of Cassandra's generally pluggable design, you can also create your own partitioner by implementing the `org.apache.cassandra.dht.IPartitioner` class and placing it on Cassandra's classpath.

Replication Strategies

A node serves as a *replica* for different ranges of data. If one node goes down, other replicas can respond to queries for that range of data. Cassandra replicates data across nodes in a manner transparent to the user, and the *replication factor* is the number of nodes in your cluster that will receive copies (replicas) of the same data. If your replication factor is 3, then three nodes in the ring will have copies of each row.

The first replica will always be the node that claims the range in which the token falls, but the remainder of the replicas are placed according to the *replication strategy* (sometimes also referred to as the *replica placement strategy*).

For determining replica placement, Cassandra implements the Gang of Four Strategy pattern, which is outlined in the common abstract class `org.apache.cassandra.locator.AbstractReplicationStrategy`, allowing different implementations of an algorithm (different strategies for accomplishing the same work). Each algorithm implementation is encapsulated inside a single class that extends the `AbstractReplicationStrategy`.

Out of the box, Cassandra provides two primary implementations of this interface (extensions of the abstract class): `SimpleStrategy` and `NetworkTopologyStrategy`. The `SimpleStrategy` places replicas at consecutive nodes around the ring, starting with the node indicated by the partitioner. The `NetworkTopologyStrategy` allows you to specify a different replication factor for each data center. Within a data center, it allocates replicas to different racks in order to maximize availability.

Legacy Replication Strategies

A third strategy, `OldNetworkTopologyStrategy`, is provided for backward compatibility. It was previously known as the `RackAwareStrategy`, while the `SimpleStrategy` was previously known as the `RackUnawareStrategy`. `NetworkTopologyStrategy` was previously known as `DataCenterShardStrategy`. These changes were effective in the 0.7 release.

The strategy is set independently for each keyspace and is a required option to create a keyspace.

Consistency Levels

In Chapter 2, we discussed Brewer's CAP theorem, in which consistency, availability, and partition tolerance are traded off against one another. Cassandra provides tuneable consistency levels that allow you to make these trade-offs at a fine-grained level. You specify a consistency level on each read or write query that indicates how much consistency you require. A higher consistency level means that more nodes need to respond to a read or write query, giving you more assurance that the values present on each replica are the same.

For read queries, the consistency level specifies how many replica nodes must respond to a read request before returning the data. For write operations, the consistency level specifies how many replica nodes must respond for the write to be reported as successful to the client. Because Cassandra is eventually consistent, updates to other replica nodes may continue in the background.

The available consistency levels include `ONE`, `TWO`, and `THREE`, each of which specify an absolute number of replica nodes that must respond to a request. The `QUORUM` consistency level requires a response from a majority of the replica nodes (sometimes expressed as "replication factor `/ 2 + 1`"). The `ALL` consistency level requires a response from all of the replicas.

For both reads and writes, the consistency levels of `ANY`, `ONE`, `TWO`, and `THREE` are considered weak, whereas `QUORUM` and `ALL` are considered strong. Consistency is tuneable in Cassandra because clients can specify the desired consistency level on both reads and writes. There is an equation that is popularly used to represent the way to achieve strong consistency in Cassandra: $R + W > N$ = *strong consistency*. In this equation, R, W, and N are the read replica count, the write replica count, and the replication factor, respectively; all client reads will see the most recent write in this scenario, and you will have strong consistency.

Distinguishing Consistency Levels and Replication Factors

If you're new to Cassandra, the replication factor can sometimes be confused with the consistency level. The replication factor is set per keyspace. The consistency level is specified per query, by the client. The replication factor indicates how many nodes you want to use to store a value during each write operation. The consistency level specifies how many nodes the client has decided must respond in order to feel confident of a successful read or write operation. The confusion arises because the consistency level is based on the replication factor, not on the number of nodes in the system.

Queries and Coordinator Nodes

Let's bring these concepts together to discuss how Cassandra nodes interact to support reads and writes from client applications. Figure 3-3 shows the typical path of interactions with Cassandra.

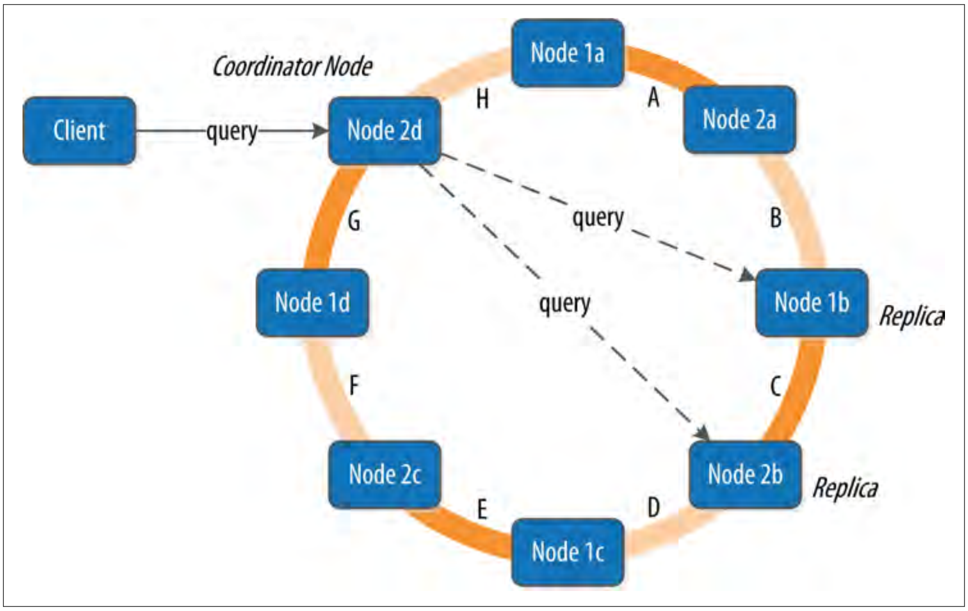

Figure 3-3. Clients, coordinator nodes, and replicas

A client may connect to any node in the cluster to initiate a read or write query. This node is known as the *coordinator node*. The coordinator identifies which nodes are replicas for the data that is being written or read and forwards the queries to them.

For a write, the coordinator node contacts all replicas, as determined by the consistency level and replication factor, and considers the write successful when a number of replicas commensurate with the consistency level acknowledge the write.

For a read, the coordinator contacts enough replicas to ensure the required consistency level is met, and returns the data to the client.

These, of course, are the "happy path" descriptions of how Cassandra works. We'll soon discuss some of Cassandra's high availability mechanisms, including hinted handoff.

Memtables, SSTables, and Commit Logs

Now let's take a look at some of Cassandra's internal data structures and files, summarized in Figure 3-4. Cassandra stores data both in memory and on disk to provide both high performance and durability. In this section, we'll focus on Cassandra's use of constructs called *memtables*, *SSTables*, and *commit logs* to support the writing and reading of data from tables.

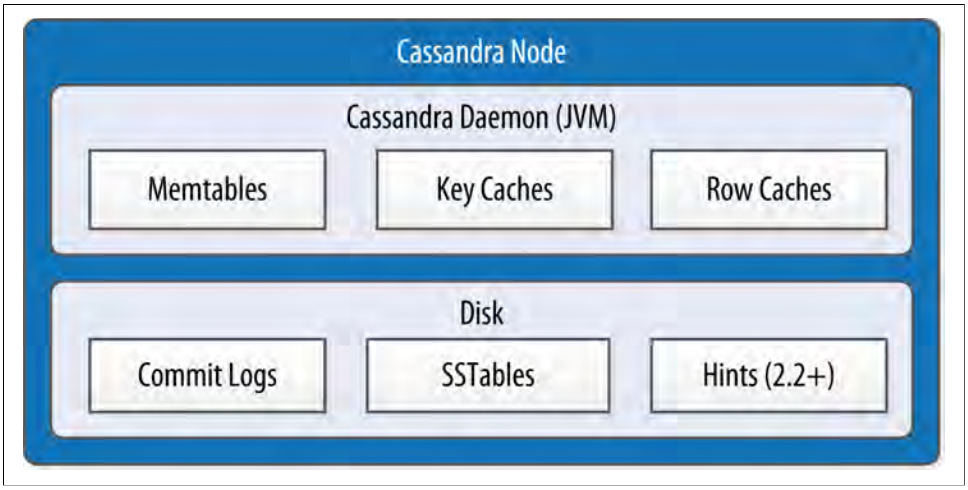

Figure 3-4. Internal data structures and files of a Cassandra node

When you perform a write operation, it's immediately written to a *commit log*. The commit log is a crash-recovery mechanism that supports Cassandra's durability goals. A write will not count as successful until it's written to the commit log, to ensure that if a write operation does not make it to the in-memory store (the memtable, discussed in a moment), it will still be possible to recover the data. If you shut down the database or it crashes unexpectedly, the commit log can ensure that data is not lost. That's because the next time you start the node, the commit log gets replayed. In fact, that's the only time the commit log is read; clients never read from it.

After it's written to the commit log, the value is written to a memory-resident data structure called the *memtable*. Each memtable contains data for a specific table. In early implementations of Cassandra, memtables were stored on the JVM heap, but improvements starting with the 2.1 release have moved the majority of memtable data to native memory. This makes Cassandra less susceptible to fluctuations in performance due to Java garbage collection.

When the number of objects stored in the memtable reaches a threshold, the contents of the memtable are flushed to disk in a file called an *SSTable*. A new memtable is then created. This flushing is a non-blocking operation; multiple memtables may exist for a single table, one current and the rest waiting to be flushed. They typically should not have to wait very long, as the node should flush them very quickly unless it is overloaded.

Each commit log maintains an internal bit flag to indicate whether it needs flushing. When a write operation is first received, it is written to the commit log and its bit flag is set to `1`. There is only one bit flag per table, because only one commit log is ever being written to across the entire server. All writes to all tables will go into the same commit log, so the bit flag indicates whether a particular commit log contains anything that hasn't been flushed for a particular table. Once the memtable has been properly flushed to disk, the corresponding commit log's bit flag is set to `0`, indicating that the commit log no longer has to maintain that data for durability purposes. Like regular logfiles, commit logs have a configurable rollover threshold, and once this file size threshold is reached, the log will roll over, carrying with it any extant dirty bit flags.

The SSTable is a concept borrowed from Google's Bigtable. Once a memtable is flushed to disk as an SSTable, it is immutable and cannot be changed by the application. Despite the fact that SSTables are compacted, this compaction changes only their on-disk representation; it essentially performs the "merge" step of a mergesort into new files and removes the old files on success.

Why Are They Called "SSTables"?

The idea that "SSTable" is a compaction of "Sorted String Table" is somewhat inaccurate for Cassandra, because the data is not stored as strings on disk.

Since the 1.0 release, Cassandra has supported the compression of SSTables in order to maximize use of the available storage. This compression is configurable per table. Each SSTable also has an associated Bloom filter, which is used as an additional performance enhancer (see "Bloom Filters" on page 54).

All writes are sequential, which is the primary reason that writes perform so well in Cassandra. No reads or seeks of any kind are required for writing a value to Cassandra because all writes are append operations. This makes one key limitation on performance the speed of your disk. Compaction is intended to amortize the reorganization of data, but it uses sequential I/O to do so. So the performance benefit is gained by splitting; the write operation is just an immediate append, and then compaction helps to organize for better future read performance. If Cassandra naively inserted values where they ultimately belonged, writing clients would pay for seeks up front.

On reads, Cassandra will read both SSTables and memtables to find data values, as the memtable may contain values that have not yet been flushed to disk. Memtables are implemented by the `org.apache.cassandra.db.Memtable` class.

Caching

As we saw in Figure 3-4, Cassandra provides three forms of caching:

- The *key cache* stores a map of partition keys to row index entries, facilitating faster read access into SSTables stored on disk. The key cache is stored on the JVM heap.
- The *row cache* caches entire rows and can greatly speed up read access for frequently accessed rows, at the cost of more memory usage. The row cache is stored in off-heap memory.
- The *counter cache* was added in the 2.1 release to improve counter performance by reducing lock contention for the most frequently accessed counters.

By default, key and counter caching are enabled, while row caching is disabled, as it requires more memory. Cassandra saves its caches to disk periodically in order to warm them up more quickly on a node restart.

Hinted Handoff

Consider the following scenario: a write request is sent to Cassandra, but a replica node where the write properly belongs is not available due to network partition, hardware failure, or some other reason. In order to ensure general availability of the ring in such a situation, Cassandra implements a feature called *hinted handoff*. You might think of a *hint* as a little Post-it note that contains the information from the write request. If the replica node where the write belongs has failed, the coordinator will create a hint, which is a small reminder that says, "I have the write information that is intended for node B. I'm going to hang onto this write, and I'll notice when node B comes back online; when it does, I'll send it the write request." That is, once it detects via gossip that node B is back online, node A will "hand off" to node B the "hint" regarding the write. Cassandra holds a separate hint for each partition that is to be written.

This allows Cassandra to be always available for writes, and generally enables a cluster to sustain the same write load even when some of the nodes are down. It also reduces the time that a failed node will be inconsistent after it does come back online.

In general, hints do not count as writes for the purposes of consistency level. The exception is the consistency level `ANY`, which was added in 0.6. This consistency level means that a hinted handoff alone will count as sufficient toward the success of a write operation. That is, even if only a hint was able to be recorded, the write still counts as successful. Note that the write is considered durable, but the data may not be readable until the hint is delivered to the target replica.

Hinted Handoff and Guaranteed Delivery

Hinted handoff is used in Amazon's Dynamo and is familiar to those who are aware of the concept of guaranteed delivery in messaging systems such as the Java Message Service (JMS). In a durable guaranteed-delivery JMS queue, if a message cannot be delivered to a receiver, JMS will wait for a given interval and then resend the request until the message is received.

There is a practical problem with hinted handoffs (and guaranteed delivery approaches, for that matter): if a node is offline for some time, the hints can build up considerably on other nodes. Then, when the other nodes notice that the failed node has come back online, they tend to flood that node with requests, just at the moment it is most vulnerable (when it is struggling to come back into play after a failure). To address this problem, Cassandra limits the storage of hints to a configurable time window. It is also possible to disable hinted handoff entirely.

As its name suggests, `org.apache.cassandra.db.HintedHandOffManager` is the class that manages hinted handoffs internally.

Although hinted handoff helps increase Cassandra's availability, it does not fully replace the need for manual repair to ensure consistency.

Lightweight Transactions and Paxos

As we discussed in Chapter 2, Cassandra provides tuneable consistency, including the ability to achieve strong consistency by specifying sufficiently high consistency levels. However, strong consistency is not enough to prevent race conditions in cases where clients need to read, then write data.

To help explain this with an example, imagine we are building a client that wants to manage user records as part of an account management application. In creating a new user account, we'd like to make sure that the user record doesn't already exist, lest we unintentionally overwrite existing user data. So we do a read to see if the record exists first, and then only perform the create if the record doesn't exist.

The behavior we're looking for is called *linearizable consistency*, meaning that we'd like to guarantee that no other client can come in between our read and write queries with their own modification. Since the 2.0 release, Cassandra supports a *lightweight transaction* (or "LWT") mechanism that provides linearizable consistency.

Cassandra's LWT implementation is based on Paxos. Paxos is a consensus algorithm that allows distributed peer nodes to agree on a proposal, without requiring a master to coordinate a transaction. Paxos and other consensus algorithms emerged as alternatives to traditional two-phase commit based approaches to distributed transactions (reference the note on Two-Phase Commit in The Problem with Two-Phase Commit).

The basic Paxos algorithm consists of two stages: prepare/promise, and propose/accept. To modify data, a coordinator node can propose a new value to the replica nodes, taking on the role of leader. Other nodes may act as leaders simultaneously for other modifications. Each replica node checks the proposal, and if the proposal is the latest it has seen, it promises to not accept proposals associated with any prior proposals. Each replica node also returns the last proposal it received that is still in progress. If the proposal is approved by a majority of replicas, the leader commits the proposal, but with the caveat that it must first commit any in-progress proposals that preceded its own proposal.

The Cassandra implementation extends the basic Paxos algorithm in order to support the desired read-before-write semantics (also known as "check-and-set"), and to allow the state to be reset between transactions. It does this by inserting two additional phases into the algorithm, so that it works as follows:

1. Prepare/Promise
2. Read/Results
3. Propose/Accept
4. Commit/Ack

Thus, a successful transaction requires four round-trips between the coordinator node and replicas. This is more expensive than a regular write, which is why you should think carefully about your use case before using LWTs.

More on Paxos

Several papers have been written about the Paxos protocol. One of the best explanations available is Leslie Lamport's "Paxos Made Simple".

Cassandra's lightweight transactions are limited to a single partition. Internally, Cassandra stores a Paxos state for each partition. This ensures that transactions on different partitions cannot interfere with each other.

You can find Cassandra's implementation of the Paxos algorithm in the package `org.apache.cassandra.service.paxos`. These classes are leveraged by the `StorageService`, which we will learn about soon.

Tombstones

In the relational world, you might be accustomed to the idea of a "soft delete." Instead of actually executing a delete SQL statement, the application will issue an update statement that changes a value in a column called something like "deleted." Programmers sometimes do this to support audit trails, for example.

There's a similar concept in Cassandra called a *tombstone*. This is how all deletes work and is therefore automatically handled for you. When you execute a delete operation, the data is not immediately deleted. Instead, it's treated as an update operation that places a tombstone on the value. A tombstone is a deletion marker that is required to suppress older data in SSTables until compaction can run.

There's a related setting called Garbage Collection Grace Seconds. This is the amount of time that the server will wait to garbage-collect a tombstone. By default, it's set to 864,000 seconds, the equivalent of 10 days. Cassandra keeps track of tombstone age, and once a tombstone is older than `GCGraceSeconds`, it will be garbage-collected. The purpose of this delay is to give a node that is unavailable time to recover; if a node is down longer than this value, then it is treated as failed and replaced.

Bloom Filters

Bloom filters are used to boost the performance of reads. They are named for their inventor, Burton Bloom. Bloom filters are very fast, non-deterministic algorithms for testing whether an element is a member of a set. They are non-deterministic because it is possible to get a false-positive read from a Bloom filter, but not a false-negative. Bloom filters work by mapping the values in a data set into a bit array and condensing a larger data set into a digest string using a hash function. The digest, by definition, uses a much smaller amount of memory than the original data would. The filters are stored in memory and are used to improve performance by reducing the need for disk access on key lookups. Disk access is typically much slower than memory access. So, in a way, a Bloom filter is a special kind of cache. When a query is performed, the Bloom filter is checked first before accessing disk. Because false-negatives are not possible, if the filter indicates that the element does not exist in the set, it certainly doesn't; but if the filter thinks that the element is in the set, the disk is accessed to make sure.

Bloom filters are implemented by the `org.apache.cassandra.utils.BloomFilter` class. Cassandra provides the ability to increase Bloom filter accuracy (reducing the number of false positives) by increasing the filter size, at the cost of more memory. This false positive chance is tuneable per table.

Other Uses of Bloom Filters

Bloom filters are used in other distributed database and caching technologies, including Apache Hadoop, Google Bigtable, and Squid Proxy Cache.

Compaction

As we already discussed, SSTables are immutable, which helps Cassandra achieve such high write speeds. However, periodic compaction of these SSTables is important in order to support fast read performance and clean out stale data values. A compaction operation in Cassandra is performed in order to merge SSTables. During compaction, the data in SSTables is merged: the keys are merged, columns are combined, tombstones are discarded, and a new index is created.

Compaction is the process of freeing up space by merging large accumulated datafiles. This is roughly analogous to rebuilding a table in the relational world. But the primary difference in Cassandra is that it is intended as a transparent operation that is amortized across the life of the server.

On compaction, the merged data is sorted, a new index is created over the sorted data, and the freshly merged, sorted, and indexed data is written to a single new SSTable (each SSTable consists of multiple files including: *Data*, *Index*, and *Filter*). This process is managed by the class `org.apache.cassandra.db.compaction.CompactionManager`.

Another important function of compaction is to improve performance by reducing the number of required seeks. There are a bounded number of SSTables to inspect to find the column data for a given key. If a key is frequently mutated, it's very likely that the mutations will all end up in flushed SSTables. Compacting them prevents the database from having to perform a seek to pull the data from each SSTable in order to locate the current value of each column requested in a read request.

When compaction is performed, there is a temporary spike in disk I/O and the size of data on disk while old SSTables are read and new SSTables are being written.

Cassandra supports multiple algorithms for compaction via the strategy pattern. The compaction strategy is an option that is set for each table. The compaction strategy extends the `AbstractCompactionStrategy` class. The available strategies include:

- `SizeTieredCompactionStrategy` (STCS) is the default compaction strategy and is recommended for write-intensive tables
- `LeveledCompactionStrategy` (LCS) is recommended for read-intensive tables
- `DateTieredCompactionStrategy` (DTCS), which is intended for time series or otherwise date-based data.

We'll revisit these strategies to discuss selecting the best strategy for each table.

One interesting feature of compaction relates to its intersection with incremental repair. A feature called *anticompaction* was added in 2.1. As the name implies, anticompaction is somewhat of an opposite operation to regular compaction in that the result is the division of an SSTable into two SSTables, one containing repaired data, and the other containing unrepaired data.

The trade-off is that more complexity is introduced into the compaction strategies, which must handle repaired and unrepaired SSTables separately so that they are not merged together.

What About Major Compaction?

Users with prior experience may recall that Cassandra exposes an administrative operation called *major compaction* (also known as *full compaction*) that consolidates multiple SSTables into a single SSTable. While this feature is still available, the utility of performing a major compaction has been greatly reduced over time. In fact, usage is actually discouraged in production environments, as it tends to limit Cassandra's ability to remove stale data.

Anti-Entropy, Repair, and Merkle Trees

Cassandra uses an *anti-entropy* protocol, which is a type of gossip protocol for repairing replicated data. Anti-entropy protocols work by comparing replicas of data and reconciling differences observed between the replicas. Anti-entropy is used in Amazon's Dynamo, and Cassandra's implementation is modeled on that (see Section 4.7 of the Dynamo paper).

Anti-Entropy in Cassandra

In Cassandra, the term *anti-entropy* is often used in two slightly different contexts, with meanings that have some overlap:

- The term is often used as a shorthand for the replica synchronization mechanism for ensuring that data on different nodes is updated to the newest version.
- At other times, Cassandra is described as having an anti-entropy *capability* that includes replica synchronization as well as hinted handoff, which is a write-time anti-entropy mechanism we read about in "Hinted Handoff" on page 51.

Replica synchronization is supported via two different modes known as *read repair* and *anti-entropy repair*. Read repair refers to the synchronization of replicas as data is read. Cassandra reads data from multiple replicas in order to achieve the requested consistency level, and detects if any replicas have out of date values. If an insufficient number of nodes have the latest value, a read repair is performed immediately to update the out of date replicas. Otherwise, the repairs can be performed in the background after the read returns. This design is observed by Cassandra as well as by straight key/value stores such as Project Voldemort and Riak.

Anti-entropy repair (sometimes called *manual repair*) is a manually initiated operation performed on nodes as part of a regular maintenance process. This type of repair is executed by using a tool called *nodetool*. Running `nodetool repair` causes Cassandra to execute a *major compaction* (see "Compaction" on page 55). During a major compaction, the server initiates a TreeRequest/TreeReponse conversation to exchange Merkle trees with neighboring nodes. The Merkle tree is a hash representing the data in that table. If the trees from the different nodes don't match, they have to be reconciled (or "repaired") to determine the latest data values they should all be set to. This tree comparison validation is the responsibility of the `org.apache.cassandra.service.AbstractReadExecutor` class.

What's a Merkle Tree?

A Merkle tree, named for its inventor, Ralph Merkle, is also known as a "hash tree." It's a data structure represented as a binary tree, and it's useful because it summarizes in short form the data in a larger data set. In a hash tree, the leaves are the data blocks (typically files on a filesystem) to be summarized. Every parent node in the tree is a hash of its direct child node, which tightly compacts the summary.

In Cassandra, the Merkle tree is implemented in the `org.apache.cassandra.utils.MerkleTree` class.

Merkle trees are used in Cassandra to ensure that the peer-to-peer network of nodes receives data blocks unaltered and unharmed. They are also used in cryptography to verify the contents of files and transmissions.

Both Cassandra and Dynamo use Merkle trees for anti-entropy, but their implementations are a little different. In Cassandra, each table has its own Merkle tree; the tree is created as a snapshot during a major compaction, and is kept only as long as is required to send it to the neighboring nodes on the ring. The advantage of this implementation is that it reduces network I/O.

Staged Event-Driven Architecture (SEDA)

Cassandra's design was influenced by Staged Event-Driven Architecture (SEDA). SEDA is a general architecture for highly concurrent Internet services, originally proposed in a 2001 paper called "SEDA: An Architecture for Well-Conditioned, Scalable Internet Services" by Matt Welsh, David Culler, and Eric Brewer (who you might recall from our discussion of the CAP theorem). You can read the original SEDA paper at *http://www.eecs.harvard.edu/~mdw/proj/seda*.

In a typical application, a single unit of work is often performed within the confines of a single thread. A write operation, for example, will start and end within the same thread. Cassandra, however, is different: its concurrency model is based on SEDA, so a single operation may start with one thread, which then hands off the work to another thread, which may hand it off to other threads. But it's not up to the current thread to hand off the work to another thread. Instead, work is subdivided into what are called *stages*, and the thread pool (really, a `java.util.concurrent.ExecutorService`) associated with the stage determines execution.

A stage is a basic unit of work, and a single operation may internally state-transition from one stage to the next. Because each stage can be handled by a different thread pool, Cassandra experiences a massive performance improvement. This design also means that Cassandra is better able to manage its own resources internally because different operations might require disk I/O, or they might be CPU-bound, or they might be network operations, and so on, so the pools can manage their work according to the availability of these resources.

A stage consists of an incoming event queue, an event handler, and an associated thread pool. Stages are managed by a controller that determines scheduling and thread allocation; Cassandra implements this kind of concurrency model using the thread pool `java.util.concurrent.ExecutorService`. To see specifically how this works, check out the `org.apache.cassandra.concurrent.StageManager` class. The following operations are represented as stages in Cassandra, including many of the concepts we've discussed in this chapter:

- Read (local reads)
- Mutation (local writes)
- Gossip
- Request/response (interactions with other nodes)
- Anti-entropy (`nodetool` repair)
- Read repair
- Migration (making schema changes)
- Hinted handoff

You can observe the thread pools associated with each of these stages by using the `nodetool tpstats` command.

A few additional operations are also implemented as stages, such as operations on memtables including flushing data out to SSTables and freeing memory. The stages implement the `IVerbHandler` interface to support the functionality for a given verb.

Because the idea of mutation is represented as a stage, it can play a role in both insert and delete operations.

A Pragmatic Approach to SEDA

Over time, developers of Cassandra and other technologies based on the SEDA architecture article have encountered performance issues due to the inefficiencies of requiring separate thread pools for each stage and event queues between each stage, even for short-lived stages. These challenges were acknowledged by Matt Welsh in the follow-up blog post "A Retrospective on SEDA".

Over time, Cassandra's developers have relaxed the strict SEDA conventions, collapsing some stages into the same thread pool to improve throughput. However, the basic principles of separating work into stages and using queues and thread pools to manage these stages are still in evidence in the code.

Managers and Services

There is a set of classes that form Cassandra's basic internal control mechanisms. We've encountered a few of them already in this chapter, including the `HintedHandOffManager`, the `CompactionManager`, and the `StageManager`. We'll present a brief overview of a few other classes here so that you can become familiar with some of the more important ones. Many of these expose `MBeans` via the Java Management Extension (JMX) in order to report status and metrics, and in some cases to allow configuration and control of their activities.

Cassandra Daemon

The `org.apache.cassandra.service.CassandraDaemon` interface represents the life cycle of the Cassandra service running on a single node. It includes the typical life cycle operations that you might expect: `start`, `stop`, `activate`, `deactivate`, and `destroy`.

You can also create an in-memory Cassandra instance programmatically by using the class `org.apache.cassandra.service.EmbeddedCassandraService`. Creating an embedded instance can be useful for unit testing programs using Cassandra.

Storage Engine

Cassandra's core data storage functionality is commonly referred to as the storage engine, which consists primarily of classes in the `org.apache.cassandra.db` package. The main entry point is the `ColumnFamilyStore` class, which manages all aspects of table storage, including commit logs, memtables, SSTables, and indexes.

Major Changes to the Storage Engine

The storage engine was largely rewritten for the 3.0 release to bring Cassandra's in-memory and on-disk representations of data in alignment with the CQL. An excellent summary of the changes is provided in the CASSANDRA-8099 JIRA issue.

The storage engine rewrite was a precursor for many other changes, most importantly, support for materialized views, which was implemented under CASSANDRA-6477. These two JIRA issues make for interesting reading if you want to better understand the changes required "under the hood" to enable these powerful new features.

Storage Service

Cassandra wraps the storage engine with a service represented by the `org.apache.cassandra.service.StorageService` class. The storage service contains the node's token, which is a marker indicating the range of data that the node is responsible for.

The server starts up with a call to the `initServer` method of this class, upon which the server registers the SEDA verb handlers, makes some determinations about its state (such as whether it was bootstrapped or not, and what its partitioner is), and registers an `MBean` with the JMX server.

Storage Proxy

The `org.apache.cassandra.service.StorageProxy` sits in front of the `StorageService` to handle the work of responding to client requests. It coordinates with other nodes to store and retrieve data, including storage of hints when needed. The `StorageProxy` also helps manage lightweight transaction processing.

Direct Invocation of the Storage Proxy

Although it is possible to invoke the `StorageProxy` programmatically, as an in-memory instance, note that this is not considered an officially supported API for Cassandra and therefore has undergone changes between releases.

Messaging Service

The purpose of `org.apache.cassandra.net.MessagingService` is to create socket listeners for message exchange; inbound and outbound messages from this node come through this service. The `MessagingService.listen` method creates a thread. Each incoming connection then dips into the `ExecutorService` thread pool using `org.apache.cassandra.net.IncomingTcpConnection` (a class that extends `Thread`) to deserialize the message. The message is validated, and then routed to the appropriate handler.

Because the `MessagingService` also makes heavy use of stages and the pool it maintains is wrapped with an `MBean`, you can find out a lot about how this service is working (whether reads are getting backed up and so forth) through JMX.

Stream Manager

Streaming is Cassandra's optimized way of sending sections of SSTable files from one node to another via a persistent TCP connection; all other communication between nodes occurs via serialized messages. The `org.apache.cassandra.streaming.StreamManager` handles these streaming messages, including connection management, message compression, progress tracking, and statistics.

CQL Native Transport Server

The CQL Native Protocol is the binary protocol used by clients to communicate with Cassandra. The `org.apache.cassandra.transport` package contains the classes that implement this protocol, including the `Server`. This native transport server manages client connections and routes incoming requests, delegating the work of performing queries to the `StorageProxy`.

There are several other classes that manage key features of Cassandra. Here are a few to investigate if you're interested:

Key feature	Class
Repair	`org.apache.cassandra.service.ActiveRepairService`
Caching	`org.apache.cassandra.service.CachingService`
Migration	`org.apache.cassandra.service.MigrationManager`
Materialized views	`org.apache.cassandra.db.view.MaterializedViewManager`
Secondary indexes	`org.apache.cassandra.db.index.SecondaryIndexManager`
Authorization	`org.apache.cassandra.auth.CassandraRoleManager`

System Keyspaces

In true "dogfooding" style, Cassandra makes use of its own storage to keep track of metadata about the cluster and local node. This is similar to the way in which Microsoft SQL Server maintains the meta-databases `master` and `tempdb`. The `master` is used to keep information about disk space, usage, system settings, and general server installation notes; the `tempdb` is used as a workspace to store intermediate results and perform general tasks. The Oracle database always has a tablespace called SYSTEM, used for similar purposes. The Cassandra `system` keyspaces are used much like these.

Let's go back to `cqlsh` to have a quick peek at the tables in Cassandra's `system` keyspace:

```
cqlsh> DESCRIBE TABLES;

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

Keyspace system_schema
----------------------
materialized_views  functions  aggregates  types            columns
tables              triggers   keyspaces   dropped_columns

Keyspace system_auth
--------------------
resource_role_permissons_index  role_permissions  role_members
roles

Keyspace system
---------------
available_ranges                       sstable_activity    local
range_xfers                            peer_events         hints
materialized_views_builds_in_progress  paxos
"IndexInfo"                            batchlog
peers                                  size_estimates
built_materialized_views               compaction_history

Keyspace system_distributed
---------------------------
repair_history  parent_repair_history
```

Seeing Different System Keyspaces?

If you're using a version of Cassandra prior to 2.2, you may not see some of these keyspaces listed. While the basic `system` keyspace has been around since the beginning, the `system_traces` keyspace was added in 1.2 to support request tracing. The `system_auth` and `system_distributed` keyspaces were added in 2.2 to support role-based access control (RBAC) and persistence of repair data, respectively. Finally, tables related to schema definition were migrated from `system` to the `system_schema` keyspace in 3.0.

Looking over these tables, we see that many of them are related to the concepts discussed in this chapter:

- Information about the structure of the cluster communicated via gossip is stored in `system.local` and `system.peers`. These tables hold information about the local node and other nodes in the cluster including IP addresses, locations by data center and rack, CQL, and protocol versions.
- The `system.range_xfers` and `system.available_ranges` track token ranges managed by each node and any ranges needing allocation.
- The `system_schema.keyspaces`, `system_schema.tables`, and `system_schema.columns` store the definitions of the keyspaces, tables, and indexes defined for the cluster.
- The construction of materialized views is tracked in the `system.materialized_views_builds_in_progress` and `system.built_materialized_views` tables, resulting in the views available in `system_schema.materialized_views`.
- User-provided extensions such as `system_schema.types` for user-defined types, `system_schema.triggers` for triggers configured per table, `system_schema.functions` for user-defined functions, and `system_schema.aggregates` for user-defined aggregates.
- The `system.paxos` table stores the status of transactions in progress, while the `system.batchlog` table stores the status of atomic batches.
- The `system.size_estimates` stores the estimated number of partitions per table, which is used for Hadoop integration.

Removal of the system.hints Table

Hinted handoffs have traditionally been stored in the `system.hints` table. As thoughtful developers have noted, the fact that hints are really messages to be kept for a short time and deleted means this usage is really an instance of the well-known anti-pattern of using Cassandra as a queue. Hint storage was moved to flat files in the 3.0 release.

Let's go back to `cqlsh` to have a quick peek at the attributes of Cassandra's `system` keyspace:

```
cqlsh> USE system;

cqlsh:system> DESCRIBE KEYSPACE;

CREATE KEYSPACE system WITH replication =
  {'class': 'LocalStrategy'} AND durable_writes = true;

...
```

We've truncated the output here because it lists the complete structure of each table. Looking at the first statement in the output, we see that the `system` keyspace is using the replication strategy `LocalStrategy`, meaning that this information is intended for internal use and not replicated to other nodes.

Immutability of the System Keyspace

Describing the `system` keyspaces produces similar output to describing any other keyspace, in that the tables are described using the `CREATE TABLE` command syntax. This may be somewhat misleading, as you cannot modify the schema of the `system` keyspaces.

Summary

In this chapter, we examined the main pillars of Cassandra's architecture, including gossip, snitches, partitioners, replication, consistency, anti-entropy, hinted handoff, and lightweight transactions, and how the use of a Staged Event-Driven Architecture maximizes performance. We also looked at some of Cassandra's internal data structures, including memtables, SSTables, and commit logs, and how it executes various operations, such as tombstones and compaction. Finally, we surveyed some of the major classes and interfaces, pointing out key points of interest in case you want to dive deeper into the code base.

CHAPTER 4

Deploying and Integrating

In this, our final chapter, it's time to share a few last pieces of advice as you work toward deploying Cassandra in production. We'll discuss options to consider in planning deployments and explore options for deploying Cassandra in various cloud environments. We'll close with a few thoughts on some technologies that complement Cassandra well.

Planning a Cluster Deployment

A successful deployment of Cassandra starts with good planning. You'll want to consider the amount of data that the cluster will hold, the network environment in which the cluster will be deployed, and the computing resources (whether physical or virtual) on which the instances will run.

Sizing Your Cluster

An important first step in planning your cluster is to consider the amount of data that it will need to store. You will, of course, be able to add and remove nodes from your cluster in order to adjust its capacity over time, but calculating the initial and planned size over time will help you better anticipate costs and make sound decisions as you plan your cluster configuration.

In order to calculate the required size of the cluster, you'll first need to determine the storage size of each of the supported tables. This calculation is based on the columns within each table as well as the estimated number of rows and results in an estimated size of one copy of your data on disk.

In order to estimate the actual physical amount of disk storage required for a given table across your cluster, you'll also need to consider the replication factor for the table's keyspace and the compaction strategy in use. The resulting formula for the total size T_t is as follows:

$$T_t = S_t * RF_k * CSF_t$$

Where S_t is the size of the table calculated using the formula referenced above, RF_k is the replication factor of the keyspace, and CSF_t is a factor representing the compaction strategy of the table, whose value is as follows:

- 2 for the `SizeTieredCompactionStrategy`. The worst case scenario for this strategy is that there is a second copy of all of the data required for a major compaction.
- 1.25 for other compaction strategies, which have been estimated to require an extra 20% overhead during a major compaction.

Once we know the total physical disk size of the data for all tables, we can sum those values across all keyspaces and tables to arrive at the total for our cluster.

We can then divide this total by the amount of usable storage space per disk to estimate a required number of disks. A reasonable estimate for the usable storage space of a disk is 90% of the disk size.

Note that this calculation is based on the assumption of providing enough overhead on each disk to handle a major compaction of all keyspaces and tables. It's possible to reduce the required overhead if you can ensure such a major compaction will never be executed by an operations team, but this seems like a risky assumption.

Sizing Cassandra's System Keyspaces

Alert readers may wonder about the disk space devoted to Cassandra's internal data storage in the various `system` keyspaces. This is typically insignificant when compared to the size of the disk. We created a three-node cluster and measured the size of each node's data storage at about 18 MB with no additional keyspaces.

Although this could certainly grow considerably if you are making frequent use of tracing, the `system_traces` tables do use TTL to allow trace data to expire, preventing these tables from overwhelming your data storage over time.

Once you've made calculations of the required size and number of nodes, you'll be in a better position to decide on an initial cluster size. The amount of capacity you build into your cluster is dependent on how quickly you anticipate growth, which must be balanced against cost of additional hardware, whether it be physical or virtual.

Selecting Instances

It is important to choose the right computing resources for your Cassandra nodes, whether you are running on physical hardware or in a virtualized cloud environment. The recommended computing resources for modern Cassandra releases (2.0 and later) tend to differ for development versus production environments:

Development environments
: Cassandra nodes in development environments should generally have CPUs with at least two cores and 8 GB of memory. Although Cassandra has been successfully run on smaller processors such as Raspberry Pi with 512 MB of memory, this does require a significant performance-tuning effort.

Production environments
: Cassandra nodes in production environments should have CPUs with at least eight cores (although four cores are acceptable for virtual machines), and anywhere from 16 MB to 64 MB of memory.

Docker and Other Container Deployments

Another deployment option that has been rapidly gaining in popularity, especially in the Linux community, is software containers such as Docker.

The value proposition of containers is often phrased as "build once, deploy anywhere." Container engines support the ability to create lightweight, portable containers for software that can be easily moved and deployed on a wide variety of hardware platforms.

In Docker, each container is a lightweight virtual machine that provides process isolation, and a separate filesystem and network adapter. The Docker engine sits between the application and the OS.

The primary challenges in deploying Cassandra in Docker have to do with networking and data directories. The default deployment in Docker uses software-defined routing. At the time of writing, the performance of this layer currently reduces Cassandra throughput by somewhere around 50%, so the recommended configuration is to use the host network stack directly. This limits the deployment to a single Cassandra node per container.

If you do attempt to run multiple Cassandra instances in a container, this is really only appropriate for a development environment. You'll want to carefully consider the memory settings.

From a data management perspective, you will want to configure data directories to be outside of the container itself. This will allow you to preserve your data when you want to upgrade or stop containers.

These same considerations apply in other container technologies, although the details will be slightly different. This is an emerging area that is likely to see continuing change over the coming months and years.

Storage

There are a few factors to consider when selecting and configuring storage, including the type and quantities of drives to use:

HDDs versus SSDs
: Cassandra supports both hard disk drives (HDDs, also called "spinning drives") and solid state drives (SSDs) for storage. Although Cassandra's usage of append-based writes is conducive to sequential writes on spinning drives, SSDs provide higher performance overall because of their support for low-latency random reads.

  Historically, HDDs have been the more cost-effective storage option, but the cost of using SSDs has continued to come down, especially as more and more cloud platform providers support this as a storage option.

Disk configuration
: If you're using spinning disks, it's best to use separate disks for data and commit log files. If you're using SSDs, the data and commit log files can be stored on the same disk.

JBOD versus RAID
: Traditionally, the Cassandra community has recommended using multiple disks for data files, with the disks configured as a Redundant Array of Independent Disks (RAID). Because Cassandra uses replication to achieve redundancy across multiple nodes, the RAID 0 (or "striped volume") configuration is considered sufficient.

  More recently, Cassandra users have been using a Just a Bunch of Disks (JBOD) deployment style. The JBOD approach provides better performance and is a good choice if you have the ability to replace individual disks.

Avoid shared storage
: When selecting storage, avoid using Storage Area Networks (SAN) and Network Attached Storage (NAS). Neither of these storage technologies scale well—they consume additional network bandwidth in order to access the physical storage over the network, and they require additional I/O wait time on both reads and writes.

Network

Because Cassandra relies on a distributed architecture involving multiple networked nodes, here are a few things you'll need to consider:

Throughput
: First, make sure your network is sufficiently robust to handle the traffic associated with distributing data across multiple nodes. The recommended network bandwidth is 1 GB or higher.

Network configuration
: Make sure that you've correctly configured firewall rules and IP addresses for your nodes and network appliances to allow traffic on the ports used for the CQL native transport, inter-node communication (the `listen_address`), JMX, and so on. This includes networking between data centers (we'll discuss cluster topology momentarily).

: The clocks on all nodes should be synchronized using Network Time Protocol (NTP) or other methods. Remember that Cassandra only overwrites columns if the timestamp for the new value is more recent than the timestamp of the existing value. Without synchronized clocks, writes from nodes that lag behind can be lost.

Avoid load balancers
: Load balancers are a feature of many computing environments. While these are frequently useful to spread incoming traffic across multiple service or application instances, it's not recommended to use load balancers with Cassandra. Cassandra already provides its own mechanisms to balance network traffic between nodes, and the DataStax drivers spread client queries across replicas, so strictly speaking a load balancer won't offer any additional help. Besides this, putting a load balancer in front of your Cassandra nodes actually introduces a potential single point of failure, which could reduce the availability of your cluster.

Timeouts
: If you're building a cluster that spans multiple data centers, it's a good idea to measure the latency between data centers and tune timeout values in the *cassandra.yaml* file accordingly.

A proper network configuration is key to a successful Cassandra deployment, whether it is in a private data center, a public cloud spanning multiple data centers, or even a hybrid cloud environment.

Cloud Deployment

Now that we've learned the basics of planning a cluster deployment, let's examine our options for deploying Cassandra in three of the most popular public cloud providers.

There are a couple of key advantages that you can realize by using commercial cloud computing providers. First, you can select from multiple data centers in order to maintain high availability. If you extend your cluster to multiple data centers in an active-active configuration and implement a sound backup strategy, you can avoid having to create a separate disaster recovery system.

Second, using commercial cloud providers allows you to situate your data in data centers that are closer to your customer base, thus improving application response time. If your application's usage profile is seasonal, you can expand and shrink your clusters in each data center according to the current demands.

You may want to save time by using a prebuilt image that already contains Cassandra. There are also companies that provide Cassandra as a managed service in a Software-as-a-Service (SaaS) offering.

Don't Forget Cloud Resource Costs

In planning a public cloud deployment, you'll want to make sure to estimate the cost to operate your cluster. Don't forget to account for resources including compute services, node and backup storage, and networking.

Amazon Web Services

Amazon Web Services (AWS) has been a popular deployment option for Cassandra, as evidenced by the presence of AWS-specific extensions in the Cassandra project such as the `Ec2Snitch`, `Ec2MultiRegionSnitch`, and the `EC2MultiRegionAddressTranslator` in the DataStax Java Driver.

Cluster layout

AWS is organized around the concepts of regions and availability zones, which are typically mapped to the Cassandra constructs of data centers and racks, respectively. A sample AWS cluster topology spanning the us-east-1 (Virginia) and eu-west-1 (Ireland) regions is shown in Figure 4-1. The node names are notional—this naming is not a required convention.

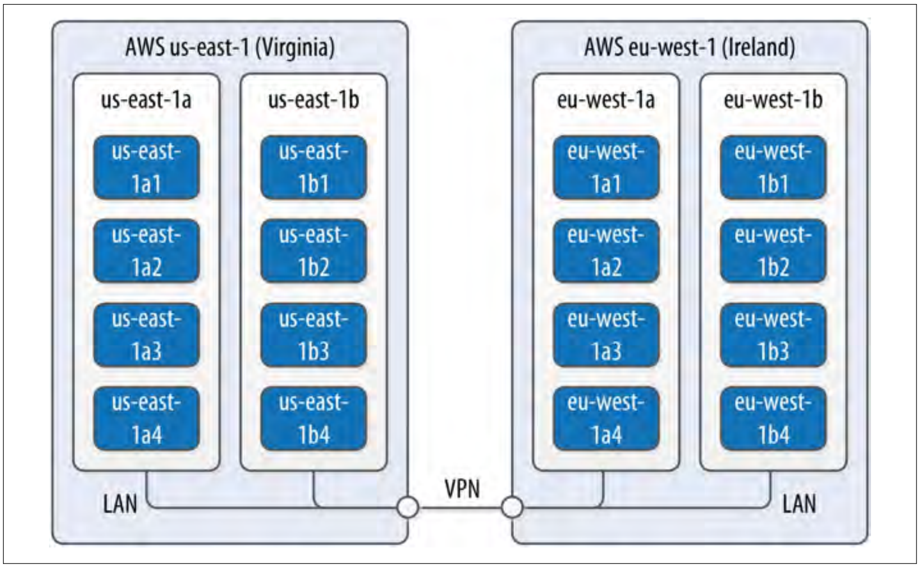

Figure 4-1. Topology of a cluster in two AWS regions

EC2 instances

The Amazon Elastic Compute Cloud (EC2) provides a variety of different virtual hardware instances grouped according to various classes. The two classes most frequently recommended for Cassandra deployments are the M-class and the I-class.

M-class instances are general-purpose instances that balance compute, memory, and network resources and are generally acceptable for development and light production environments. Later M-class instances such as M3 and M4 are SSD-backed.

The I-class instances are all SSD-backed and designed for high I/O. These instances come at a higher cost that generally pays off for heavily loaded production clusters.

You can find more information about the various instance types available at *https://aws.amazon.com/ec2/instance-types*.

DataStax Enterprise provides a prebuilt Amazon Machine Image (AMI) to simplify deployment (an AMI for DataStax Community Edition was discontinued after the Cassandra 2.1 release).

Data storage

Historically, the recommended disk option in AWS was to use ephemeral storage attached to each instance. The drawback of this is that if an instance on which a node is running is terminated (as happens occasionally in AWS), the data is lost.

By late 2015, the network-attached storage known as Elastic Block Store (EBS) has been proven to provide a reliable place to store data that doesn't go away when EC2 instances are dropped, at the cost of additional latency.

AWS Services such as S3 and Glacier are a good option for short- to medium-term and long-term storage of backups, respectively.

Networking

If you're running a multi-region configuration, you'll want to make sure you have adequate networking between the regions. Many have found that using elements of the AWS Virtual Private Cloud (VPC) provides an effective way of achieving reliable, high-throughput connections between regions. AWS Direct Connect provides dedicated private networks, and there are virtual private network (VPN) options available as well. These services of course come at an additional cost.

If you have a single region deployment or a multi-region deployment using VPN connections between regions, use the `Ec2Snitch`. If you have a multi-region deployment that uses public IP between regions, use the `Ec2MultiRegionSnitch`. For either snitch, increasing the `phi_convict_threshold` value in the *cassandra.yaml* file to 12 is generally recommended in the AWS network environment.

Microsoft Azure

DataStax and Microsoft have partnered together to help improve deployment of Cassandra in Microsoft's Azure on both Windows and Ubuntu Linux OSs.

Cluster layout

Azure provides data centers in locations worldwide, using the same term "region" as AWS. The concept of *availability sets* is used to manage collections of VMs. Azure manages the assignment of the VMs in an availability set across *update domains*, which equate to Cassandra's racks.

Virtual machine instances

Similar to AWS, Azure provides several classes of VMs. The D series, D series v2, and G series machines are all SSD-backed instances appropriate for Cassandra deployments. The G series VMs provide additional memory as required for integrations such as those described next. You can find more information about Azure VM types at *https://azure.microsoft.com/en-us/pricing/details/virtual-machines*.

Data storage

Azure provides a standard SSD capability on the previously mentioned instances. In addition, you can upgrade to use Premium Storage, which provides network-attached SSDs in larger sizes up to 1 TB.

Networking

Public IP addresses are recommended for both single-region and multi-region deployments. Use the `GossipingPropertyFileSnitch` to allow your nodes to detect the cluster topology. Networking options between regions include VPN Gateways and the Express Route, which provides up to 2 GB/s throughput.

Google Cloud Platform

Google Cloud Platform provides cloud computing, application hosting, networking, storage, and various Software-as-a-Service (SaaS) offerings such as Google's Translate and Prediction APIs.

Cluster layout

The Google Compute Environment (GCE) provides regions and zones, corresponding to Cassandra's data centers and racks, respectively.

Virtual machine instances

GCE's *n1-standard* and *n1-highmemory* machine types are recommended for Cassandra deployments. You can launch Cassandra quickly on the Google Cloud Platform using the Cloud Launcher. For example if you search the launcher at *https://cloud.google.com/launcher/?q=cassandra*, you'll find options for creating a cluster in just a few button clicks.

Data storage

GCE provides both spinning disk (*pd-hdd)* and solid state disk options for both ephemeral drives (*local-ssd*) and network-attached drives (*pd-ssd*). There are three storage options that can be used to store Cassandra backups, each with different cost and availability profiles: Standard, Durable Reduced Availability (DRA), and Nearline.

Networking

The `GoogleCloudSnitch` is a custom snitch designed just for the GCE. VPN networking is available between regions.

Integrations

As we learned in Chapter 2, Cassandra is a great solution for many applications, but that doesn't guarantee that it provides everything you need for your application or enterprise. In this section, we'll discuss several technologies that can be paired with Cassandra in order to provide more robust solutions for features such as searching and analytics.

Apache Lucene, SOLR, and Elasticsearch

One of the features that is commonly required in applications built on top of Cassandra is full text searching. This capability can be added to Cassandra via Apache Lucene, which provides an engine for distributed indexing and searching, and its subproject, Apache Solr, which adds REST and JSON APIs to the Lucene search engine.

Elasticsearch is another popular open source search framework built on top of Apache Lucene. It supports multitenancy and provides Java and JSON over HTTP APIs.

Stratio has provided a plugin that can be used to replace Cassandra's standard secondary index implementation with a Lucene index. When using this integration, there is a Lucene index available on each Cassandra node that provides high performance search capability.

Apache Hadoop

Apache Hadoop is a framework that provides distributed storage and processing of large data sets on commodity hardware. This work originated at Google in the early 2000s. Google found that several internal groups had been implementing similar functionality for data processing, and that these tools commonly had two phases: a map phase and a reduce phase. A map function operates over raw data and produces intermediate values. A reduce function distills those intermediate values, producing the final output. In 2006, Doug Cutting wrote open source implementations of the Google File System, and MapReduce, and thus, Hadoop was born.

Similar to Cassandra, Hadoop uses a distributed architecture with nodes organized in clusters. The typical integration is to install Hadoop on each Cassandra node that will be used to provide data. You can use Cassandra as a data source by running a Hadoop Task Tracker and Data Node on each Cassandra node. Then, when a MapReduce job is initiated (typically on a node external to the Cassandra cluster), the Job Tracker can query Cassandra for data as it tracks map and reduce tasks, as shown in Figure 4-2.

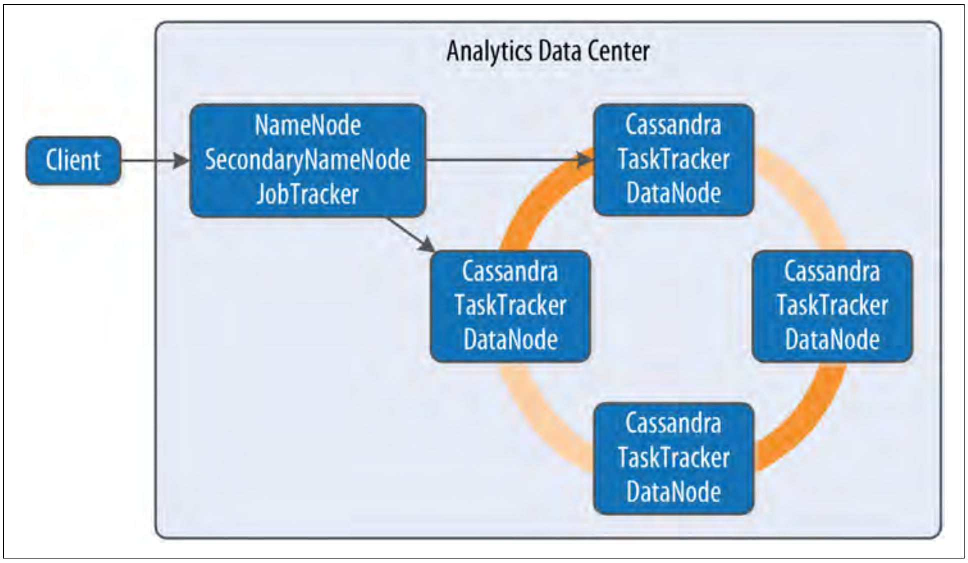

Figure 4-2. Topology of a Hadoop-Cassandra cluster

Starting in the late 2000s, Hadoop has been a huge driver of interest and growth in the Big Data community. Due to its popularity, a large ecosystem of extensions has developed around Hadoop, such as Apache Pig, which provides a framework and language for data analysis, and Apache Hive, a data warehouse tool. The Hadoop community has begun moving away from MapReduce because of its relatively slow performance (all data is written to disk, even for intermediate processing steps), high memory use, non-intuitive API, and batch-only processing mode. For this reason, the emergence of the Apache Spark project has been a significant development.

Apache Spark

with Patrick McFadin

Apache Spark is an emerging data analytics framework that provides a massively parallel processing framework to enable simple API calls across large volumes of data. Originally developed in 2009 at UC Berkeley as an improvement to MapReduce, Spark was open sourced in 2010, and became an Apache project in 2014.

Unlike Hadoop, which writes intermediate results to disk, the Spark core processing engine is designed to maximize memory usage while minimizing disk and network access. Spark uses streaming instead of batch-oriented processing to achieve processing speeds up to 100 times faster than Hadoop. In addition, Spark's API is much simpler to use than Hadoop.

The basic unit of data representation in Spark is the Resilient Distributed Dataset (RDD). The RDD is a description of the data to be processed, such as a file or data collection. Once an RDD is created, the data contained can be transformed with API calls as if all of the data were contained in a single machine. However, in reality, the RDD can span many nodes in the network by partitioning. Each partition can be operated on in parallel to produce a final result. The RDD supports the familiar `map` and `reduce` operations plus additional operations such as `count`, `filter`, `union`, and `distinct`. For a full list of transformations, see the Spark documentation.

In addition to the core engine, Spark includes further libraries for different types of processing requirements. These are included as subprojects that are managed separately from Spark core, but follow a similar release schedule:

- SparkSQL provides familiar SQL syntax and relational queries over structured data.
- MLlib is Spark's machine learning library.
- SparkR provides support for using the R statistics language in Spark jobs.
- GraphX is Spark's library for graph and collection analytics.
- Spark Streaming provides near real-time processing of live data streams.

Use cases for Spark with Cassandra

As we've discussed in this book, Apache Cassandra is a great choice for transactional workloads that require high scale and maximum availability. Apache Spark is a great choice for analyzing large volumes of data at scale. Combining the two enables many interesting use cases that exploit the power of both technologies.

An example use case is high-volume time-series data. A system for ingesting weather data from thousands of sensors with variable volume is a perfect fit for Cassandra. Once the data is collected, further analysis on data stored in Cassandra may be difficult given that the analytics capabilities available using CQL are limited. At this point, adding Spark to the solution will open many new uses for the collected data. For example, we can pre-build aggregations from the raw sensor data and store those results in Cassandra tables for use in frontend applications. This brings analytics closer to users without the need to run complex data warehouse queries at runtime.

Or consider the hotel application we've been using throughout this book. We could use Spark to implement various analytic tasks on our reservation and guest data, such as generating reports on revenue trends, or demographic analysis of anonymized guest records.

One use case to avoid is using Spark-Cassandra integration as an alternative to a Hadoop workload. Cassandra is suited for transactional workloads at high volume and shouldn't be considered as a data warehouse. When approaching a use case where both technologies might be needed, first apply Cassandra to solving a problem suited for Cassandra, such as those we discuss in Chapter 2. Then consider incorporating Spark as a way to analyze and enrich the data stored in Cassandra without the cost and complexity of extract, transform, and load (ETL) processing.

Deploying Spark with Cassandra

Cassandra places data per node based on token assignment. This existing data distribution can be used as an advantage to parallelize Spark jobs. Because each node contains a subset of the cluster's data, the recommended configuration for Spark-Cassandra integrations is to co-locate a Spark Worker on each Cassandra node in a data center, as shown in Figure 4-3.

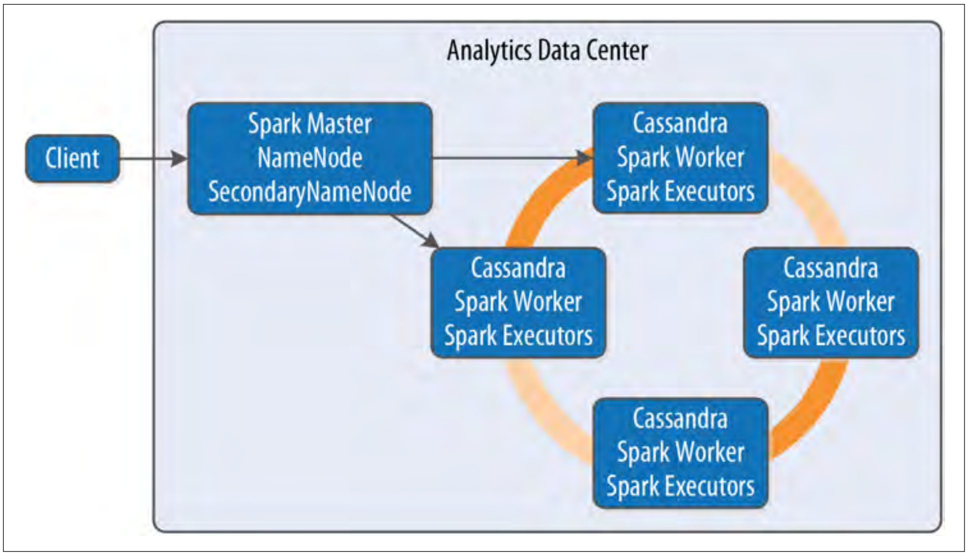

Figure 4-3. Topology of a Spark-Cassandra cluster

When a job is submitted to the Spark Master, the Spark Workers on each node spawn Spark Executors to complete the work. Using the `spark-cassandra-connector` as a conduit, he data required for each job is sourced from the local node as much as possible. We'll learn more about the connector momentarily.

Because each node contains a portion of the entire data in the cluster, each Spark Worker will only need to process that local subset of data. An example is a count action on a table. Each node will have a range of the table's data. The count is calculated locally and then merged from every node to produce the total count.

This design maximizes data locality, resulting in improved throughput and lower resource utilization for analytic jobs. The Spark Executors only communicate over the network when data needs to be merged from other nodes. As cluster sizes get larger, the efficiency gains of this design are much more pronounced.

Using a Separate Data Center for Analytics

A common deployment model for Cassandra and analytics toolsets such as Spark is to create a separate data center for analytic processing. This has the advantage of isolating the performance impact of analytics workloads from the rest of the cluster. The analytics data center can be constructed as a "virtual" data center where the actual hardware exists in the same physical location as another data center within the cluster. Using the `NetworkTopologyStrategy`, you can specify a lower replication factor for the analytics data center, as your required availability in this data center will typically be lower.

The spark-cassandra-connector

The `spark-cassandra-connector` is an open source project sponsored by DataStax on GitHub. The connector can be used by clients as a conduit to read and write data from Cassandra tables via Spark. The connector provides features including SQL queries and server-side filtering. The connector is implemented in Scala, but a Java API is available as well. API calls from the `spark-cassandra-connector` provide direct access to data in Cassandra in a context related to the underlying data. As Spark accesses data, the connector translates to and from Cassandra as the data source.

To start using the `spark-cassandra-connector`, you'll need to download both the connector and Spark. Although a detailed installation guide is beyond our scope here, we'll give a quick summary. For a more fulsome introduction, we suggest the O'Reilly book *Learning Spark*. You can download either a pre-built version of Spark, or you can build Spark yourself from the source.

If you're building an application in Java or Scala and using Maven, you'll want to add dependencies such as the following to your project's *pom.xml* file to access the Spark core and connector:

```
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-core_2.10</artifactId>
    <version>1.5.2</version>
</dependency>
<dependency>
    <groupId>com.datastax.spark</groupId>
    <artifactId>spark-cassandra-connector_2.10</artifactId>
    <version>1.5.0</version>
</dependency>
```

Spark supports two modes of deployment: *local* and *cluster*. Cluster mode involves a central Spark Master and many compute nodes. Local mode runs entirely on the local host; this mode is best suited for development. For this example, we will run in local mode, but clustering requires only a few more steps.

Let's review the common API elements used for most Spark jobs accessing data in Cassandra. In this section, we've chosen to write our examples in Scala because of its simplicity and readability, as well as the fact that Spark and many Spark applications are written in Scala. The Java API is similar but quite a bit more verbose; we've provided a Java version of this example code in the GitHub repository for this book. To connect your Spark application to Cassandra, you will first need to create a `SparkContext` containing connection parameters:

```
val conf = new SparkConf(true)
   .set("spark.cassandra.connection.host", "127.0.0.1")
   .setMaster("local[*]")
   .setAppName(getClass.getName)
   // Optionally
   .set("cassandra.username", "cassandra")
   .set("cassandra.password", "cassandra")

val sc = new SparkContext(conf)
```

Establishing a connection between Cassandra and Spark is simply the process of pointing to the running Cassandra cluster and Spark Master. This example configuration shows how to connect to a local Cassandra node and Spark Master. You can also provide Cassandra login credentials if required.

Once the `SparkContext` is created, you can then operate on Cassandra data by creating an RDD representing a Cassandra table. For example, let's create an RDD representing the `reservations_by_hotel_date` table from the `reservation` keyspace:

```
val rdd = sc.cassandraTable("reservation",
  "reservations_by_hotel_date")
```

Once you've created an RDD, you can perform transformations and actions on it. For example, to get the total number of reservations, create the following action to count every record in the table:

```
println("Number of reservations: " + rdd.count)
```

Because this is running as an analytics job in parallel with Cassandra, it is much more efficient than running a `SELECT count(*) FROM reservations` from `cqlsh`.

As the underlying structure of the RDD is a Cassandra table, you can use CQL to filter the data and select rows. In Cassandra, filter queries using native CQL require a partition key to be efficient, but that restriction is removed when running queries as Spark jobs.

For example, you might derive a use case to produce a report listing reservations by end date, so that each hotel can know who is checking out on a given day. In this example, `end_date` is not a partition key or clustering column, but you can scan the entire cluster's data looking for reservations with a checkout date of September 8, 2016:

```
val rdd = sc.cassandraTable("reservation",
  "reservations_by_hotel_date")
  .select("hotel_id", "confirm_number")
  .where("end_date = ?", "2016-09-08")

// Invoke the action to run the spark job
rdd.toArray.foreach(println)
```

Finding and retrieving data is only half of the functionality available—you can also save data back to Cassandra. Traditionally, data in a transactional database would require extraction to a separate location in order to perform analytics. With the `spark-cassandra-connector`, you can extract data, transform in place, and save it directly back to a Cassandra table, eliminating the costly and error-prone ETL process. Saving data back to a Cassandra table is amazingly easy:

```
// Create a collection of guests with simple identifiers
val collection = sc.parallelize(Seq(("1", "Delaney", "McFadin"),
  ("2", "Quinn", "McFadin")))

// Save to the guests table
collection.saveToCassandra("reservation", "guests",
  SomeColumns("guest_id", "first_name", "last_name"))
```

This is a simple example, but the basic syntax applies to any data. A more advanced example would be to calculate the average daily revenue for a hotel and write the results to a new Cassandra table. In a sensor application, you might calculate high and low temperatures for a given day and write those results back out to Cassandra.

Querying data is not just limited to Spark APIs. With SparkSQL, you can use familiar SQL syntax to perform complex queries on data in Cassandra, including query options not available in CQL. It's easy to create enhanced queries such as aggregations, ordering, and joins.

To embed SQL queries inside your code, you need to create a `CassandraSQLContext`:

```
// Use the SparkContext to create a CassandraSQLContext
val cc = new CassandraSQLContext(sc)

// Set the keyspace
cc.setKeyspace("reservation")
val rdd = cc.cassandraSql("
SELECT hotel_id, confirm_number
FROM reservations_by_hotel_date
WHERE end_date = '2016-09-08'
```

```
    ORDER BY hotel_id")

// Perform action to run SQL job
rdd.collect().foreach(println)
```

The SQL syntax is similar to the Spark job from before, but is more familiar to users with a traditional database background. To explore data outside of writing Spark jobs, you can also use the `spark-sql` shell, which is available under the *bin* directory in your Spark installation.

Integrations in DataStax Enterprise

DataStax Enterprise is a productized version of Cassandra that supports many of the integrations described in this chapter. Specifically, DSE Search provides integration with Solr, while DSE Analytics provides integration with Apache Spark and elements of the Hadoop ecosystem such as MapReduce, Hive, and Pig.

Additional DSE features include additional security provider plugins and an in-memory configuration suitable for applications that require extremely fast response times.

Summary

In this chapter, we've just scratched the surface of the many deployment and integration options available for Cassandra. Hopefully we've piqued your interest in the wide range of directions you can take your applications using Cassandra and related technologies.

And now we've come to the end of our journey together. If we've achieved our goal, you now have an in-depth understanding of the right problems to solve using Cassandra, and how to design, implement, deploy, and maintain successful applications.

About the Authors

Eben Hewitt is Chief Technology Officer for Choice Hotels International, one of the largest hotel companies in the world. He is the author of several books on architecture, distributed systems, and programming. He has consulted for venture capital firms, and is a frequently invited speaker on technology and strategy.

Jeff Carpenter is a Systems Architect for Choice Hotels International, with 20 years of experience in the hospitality and defense industries. Jeff's interests include SOA/microservices, architecting large-scale systems, and data architecture. He has worked on projects ranging from complex battle planning systems to a cloud-based hotel reservation system. Jeff is passionate about disruptive projects that change industries, mentoring architects and developers, and the next challenge.

Colophon

The bird on the cover of *Cassandra: The Definitive Guide* is a paradise flycatcher. Part of the family of monarch flycatchers (*Monarchidae*), paradise flycatchers are passerine (perching) insectivores. They're the most widely distributed of the monarch flycatchers and can be found from sub-Saharan Africa to Southeast Asia and on many Pacific islands. While most species are resident, others, including the Japanese paradise flycatcher and the Satin flycatcher, are migratory.

Most species of paradise flycatcher are sexually dimorphic, meaning that males and females look different. Females of most species tend to be less brilliantly colored than their male counterparts, which are also characterized by long tail feathers that vary in length according to species. For example, the male Asian paradise flycatcher's tail streamers can be approximately 15 inches long. Female flycatchers are believed to select their mate based on tail length. Paradise flycatchers are monogamous, which makes their distinctive coloring and plumage unusual, as this form of sexual display is usually reserved for non-monogamous species.

Because they're so widely distributed, paradise flycatchers can be found in a variety of habitats, including savannas, bamboo groves, rain forests, deciduous forests, and even cultivated gardens. Most species catch their food on the wing, thanks in part to their quick reflexes and sharp eyesight.

The cover image is from *Cassell's Natural History, Vol. IV*. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.