Cassandra High Availability

Harness the power of Apache Cassandra to build scalable, fault-tolerant, and readily available applications

Robbie Strickland
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 "Cassandra's Approach to High Availability"
- A synopsis of the book’s content
- More information on Cassandra High Availability

About the Author

Robbie Strickland got involved in the Apache Cassandra project in 2010, and he initially went into production with the 0.5 release. He has made numerous contributions over the years, including his work on drivers for C# and Scala, and multiple contributions to the core Cassandra codebase. In 2013, he became the very first certified Cassandra developer, and in 2014, DataStax selected him as an Apache Cassandra MVP.

While this is Robbie's first published technical book, he has been an active speaker and writer in the Cassandra community and is the founder of the Atlanta Cassandra Users Group. Other examples of his writing can be found on the DataStax blog, and he has conducted numerous webinars and spoken at many conferences over the years.

I would like to thank my wife for encouraging me to go forward with this project and for continuing to be supportive throughout the significant time commitment required to write a book. Also, I am truly appreciative of my excellent reviewers: Richard Low, Jimmy Mårdell, Rob Murphy, and Russell Spitzer. They helped keep me honest, and their deep expertise added materially to the quality of the content. I would also like to thank the entire staff at Packt Publishing who were involved in the book's publishing process. Lastly, I want to thank Logan Johnson who initially pointed me toward Cassandra. The risk has paid off, and Logan is responsible for starting me off on this path.
Cassandra High Availability

Cassandra is a fantastic data store and is certainly well suited as the foundation for a highly available system. In fact, it was built for such a purpose: to handle Facebook's messaging service. However, it hasn't always been so easy to use, with its early Thrift interface and unfamiliar data model causing many potential users to pause—and in many cases for a good reason.

Fortunately, Cassandra has matured substantially over the last few years. I used to advise people to use Cassandra only if nothing else would do the job because the learning curve for it was quite high. However, the introduction of newer features such as CQL and vnodes has changed the game entirely.

What once appeared complex and overly daunting now comes across as deceptively simple. A SQL-like interface masks the underlying data structure, whose familiarity can lure an unsuspecting new user into dangerous traps. The moral of this story is that it's not a relational database, and you still need to know what it's doing under the hood.

Imparting this knowledge is the core objective of this book. Each chapter attempts to demystify the inner workings of Cassandra so that you no longer have to work blindly against a black box data store. You will learn to configure, design, and build your system based on a fundamentally solid foundation.

The good news is that Cassandra makes the task of building massively scalable and incredibly reliable systems relatively straightforward, presuming you understand how to partner with it to achieve these goals.

Since you are reading this book, I presume you are either already using Cassandra or planning to do so, and that you're interested in building a highly available system on top of it. If so, I am confident that you will meet with success if you follow the principles and guidelines offered in the chapters that follow.

What This Book Covers

Chapter 1, Cassandra's Approach to High Availability, is an introduction to concepts related to system availability and the problems that have been encountered historically while trying to make data stores highly available. This chapter outlines Cassandra's solutions to these problems.

Chapter 2, Data Distribution, outlines the core mechanisms that underlie Cassandra's distributed hash table model, including consistent hashing and partitioner implementations.
Chapter 3, Replication, offers an in-depth look at the data replication architecture used in Cassandra, with a focus on the relationship between consistency levels and replication factors.

Chapter 4, Data Centers, enables you to thoroughly understand Cassandra's robust data center replication capabilities, including deployment on EC2 and building separate clusters for analysis using Hadoop or Spark.

Chapter 5, Scaling Out, is a discussion on the tools, processes, and general guidance required to properly increase the size of your cluster.

Chapter 6, High Availability Features in the Native Java Client, covers the new native Java driver and its availability-related features. We'll discuss node discovery, cluster-aware load balancing, automatic failover, and other important concepts.

Chapter 7, Modeling for High Availability, explains the important concepts you need to understand while modeling highly available data in Cassandra. CQL, keys, wide rows, and denormalization are among the topics that will be covered.

Chapter 8, Antipatterns, complements the data modeling chapter by presenting a set of common antipatterns that proliferate among inexperienced Cassandra developers. Some patterns include queues, joins, high delete volumes, and high cardinality secondary indexes among others.

Chapter 9, Failing Gracefully, helps the reader to understand how to deal with various failure cases, as failure in a large distributed system is inevitable. We'll examine a number of possible failure scenarios, and discuss how to detect and resolve them.
Cassandra's Approach to High Availability

What does it mean for a data store to be "highly available"? When designing or configuring a system for high availability, architects typically hope to offer some guarantee of uptime even in the presence of failure. Historically, it has been sufficient for the vast majority of systems to be available for less than 100 percent of the time, with some attempting to achieve the "five nines", or 99.999, percent uptime.

The exact definition of high availability depends on the requirements of the application. This concept has gained increasing significance in the context of web applications, real-time systems, and other use cases that cannot afford any downtime. Database systems must not only guarantee system uptime, the ability to fulfill requests, but also ensure that the data itself remains available.

Traditionally, it has been difficult to make databases highly available, especially the relational database systems that have dominated the scene for the last couple of decades. These systems are most often designed to run on a single large machine, making it challenging to scale out to multiple machines.

Let's examine some of the reasons why many popular database systems have difficulty being deployed in high availability configurations, as this will allow us to have a greater understanding of the improvements that Cassandra offers. Exploring these reasons can help us to put aside previous assumptions that simply don't translate to the Cassandra model.
Cassandra's Approach to High Availability

Therefore, in this chapter, we'll cover the following topics:

- The atomicity, consistency, isolation and durability (ACID) properties
- Monolithic architecture
- Master-slave architecture, covering sharding and leader election
- Cassandra's approach to achieve high availability

ACID

One of the most significant obstacles that prevents traditional databases from achieving high availability is that they attempt to strongly guarantee the ACID properties:

- **Atomicity**: This guarantees that database updates associated with a transaction occur in an all-or-nothing manner. If some part of the transaction fails, the state of the database remains unchanged.
- **Consistency**: This assures that the integrity of data will be preserved across all instances of that data. Changes to a value in one location will definitely be reflected in all other locations.
- **Isolation**: This attempts to ensure that concurrent transactions that manipulate the same data do so in a controlled manner, essentially isolating in-process changes from other clients. Most traditional relational database systems provide various levels of isolation with different guarantees at each level.
- **Durability**: This ensures that all writes are preserved in nonvolatile storage, most commonly on disk.

Database designers most commonly achieve these properties via write masters, locks, elaborate storage area networks, and the like—all of which tend to sacrifice availability. As a result, achieving some semblance of high availability frequently involves bolt-on components, log shipping, leader election, sharding, and other such strategies that attempt to preserve the original design.

The monolithic architecture

The simplest design approach to guarantee ACID properties is to implement a monolithic architecture where all functions reside on a single machine. Since no coordination among nodes is required, the task of enforcing all the system rules is relatively straightforward.
Increasing availability in such architectures typically involves hardware layer improvements, such as RAID arrays, multiple network interfaces, and hot-swappable drives. However, the fact remains that even the most robust database server acts as a single point of failure. This means that if the server fails, the application becomes unavailable. This architecture can be illustrated with the following diagram:

A common means of increasing capacity to handle requests on a monolithic architecture is to move the storage layer to a shared component such as a storage area network (SAN) or network attached storage (NAS). Such devices are usually quite robust with large numbers of disks and high-speed network interfaces. This approach is shown in a modification of the previous diagram, which depicts two database servers using a single NAS.
You'll notice that while this architecture increases the overall request handling capacity of the system, it simply moves the single failure point from the database server to the storage layer. As a result, there is no real improvement from an availability perspective.

The master-slave architecture

As distributed systems have become more commonplace, the need for higher capacity distributed databases has grown. Many distributed databases still attempt to maintain ACID guarantees (or in some cases only the consistency aspect, which is the most difficult in a distributed environment), leading to the master-slave architecture.

In this approach, there might be many servers handling requests, but only one server can actually perform writes so as to maintain data in a consistent state. This avoids the scenario where the same data can be modified via concurrent mutation requests to different nodes. The following diagram shows the most basic scenario:

![Diagram of master-slave architecture]

However, we still have not solved the availability problem, as a failure of the write master would lead to application downtime. It also means that writes do not scale well, since they are all directed to a single machine.
Sharding
A variation on the master-slave approach that enables higher write volumes is a technique called **sharding**, in which the data is partitioned into groups of keys, such that one or more masters can own a known set of keys. For example, a database of user profiles can be partitioned by the last name, such that A-M belongs to one cluster and N-Z belongs to another, as follows:

An astute observer will notice that both master-slave and sharding introduce failure points on the master nodes, and in fact the sharding approach introduces multiple points of failure—one for each master! Additionally, the knowledge of where requests for certain keys go rests with the application layer, and adding shards requires manual shuffling of data to accommodate the modified key ranges.

Some systems employ shard managers as a layer of abstraction between the application and the physical shards. This has the effect of removing the requirement that the application must have knowledge of the partition map. However, it does not obviate the need for shuffling data as the cluster grows.
Master failover

A common means of increasing availability in the event of a failure on a master node is to employ a master failover protocol. The particular semantics of the protocol vary among implementations, but the general principle is that a new master is appointed when the previous one fails. Not all failover algorithms are equal; however, in general, this feature increases availability in a master-slave system.

Even a master-slave database that employs leader election suffers from a number of undesirable traits:

- Applications must understand the database topology
- Data partitions must be carefully planned
- Writes are difficult to scale
- A failover dramatically increases the complexity of the system in general, and especially so for multisite databases
- Adding capacity requires reshuffling data with a potential for downtime

Considering that our objective is a highly available system, and presuming that scalability is a concern, are there other options we need to consider?

Cassandra's solution

The reality is that not every transaction in every application requires full ACID guarantees, and ACID properties themselves can be viewed as more of a continuum where a given transaction might require different degrees of each property.

Cassandra's approach to availability takes this continuum into account. In contrast to its relational predecessors—and even most of its NoSQL contemporaries—it original architects considered availability as a key design objective, with the intent to achieve the elusive goal of 100 percent uptime. Cassandra provides numerous knobs that give the user highly granular control of the ACID properties, all with different trade-offs.

The remainder of this chapter offers an introduction to Cassandra's high availability attributes and features, with the rest of the book devoted to help you to make use of these in real-world applications.
Cassandra's architecture

Unlike either monolithic or master-slave designs, Cassandra makes use of an entirely peer-to-peer architecture. All nodes in a Cassandra cluster can accept reads and writes, no matter where the data being written or requested actually belongs in the cluster. Internode communication takes place by means of a gossip protocol, which allows all nodes to quickly receive updates without the need for a master coordinator.

This is a powerful design, as it implies that the system itself is both inherently available and massively scalable. Consider the following diagram:
Note that in contrast to the monolithic and master-slave architectures, there are no special nodes. In fact, all nodes are essentially identical, and as a result Cassandra has no single point of failure—and therefore no need for complex sharding or leader election. But how does Cassandra avoid sharding?

**Distributed hash table**

Cassandra is able to achieve both availability and scalability using a data structure that allows any node in the system to easily determine the location of a particular key in the cluster. This is accomplished by using a distributed hash table (DHT) design based on the Amazon Dynamo architecture.

As we saw in the previous diagram, Cassandra’s topology is arranged in a ring, where each node owns a particular range of data. Keys are assigned to a specific node using a process called consistent hashing, which allows nodes to be added or removed without having to rehash every key based on the new range.

The node that owns a given key is determined by the chosen partitioner. Cassandra ships with several partitioner implementations or developers can define their own by implementing a Java interface.

These topics will be covered in greater detail in the next chapter.

**Replication**

One of the most important aspects of a distributed data store is the manner in which it handles replication of data across the cluster. If each partition were only stored on a single node, the system would effectively possess many single points of failure, and a failure of any node could result in catastrophic data loss. Such systems must therefore be able to replicate data across multiple nodes, making the occurrence of such loss less likely.

Cassandra has a sophisticated replication system, offering rack and data center awareness. This means it can be configured to place replicas in such a manner so as to maintain availability even during otherwise catastrophic events such as switch failures, network partitions, or data center outages. Cassandra also includes a mechanism that maintains the replication factor during node failures.
Replication across data centers

Perhaps the most unique feature Cassandra provides to achieve high availability is its multiple data center replication system. This system can be easily configured to replicate data across either physical or virtual data centers. This facilitates geographically dispersed data center placement without complex schemes to keep data in sync. It also allows you to create separate data centers for online transactions and heavy analysis workloads, while allowing data written in one data center to be immediately reflected in others.

*Chapters 3, Replication, and Chapter 4, Data Centers, will provide a complete discussion of Cassandra's extensive replication features.*

Tunable consistency

Closely related to replication is the idea of consistency, the C in ACID that attempts to keep replicas in sync. Cassandra is often referred to as an eventually consistent system, a term that can cause fear and trembling for those who have spent many years relying on the strong consistency characteristics of their favorite relational databases. However, as previously discussed, consistency should be thought of as a continuum, not as an absolute.

With this in mind, Cassandra can be more accurately described as having tunable consistency, where the precise degree of consistency guarantee can be specified on a per-statement level. This gives the application architect ultimate control over the trade-offs between consistency, availability, and performance at the call level—rather than forcing a one-size-fits-all strategy onto every use case.

The CAP theorem

Any discussion of consistency would be incomplete without at least reviewing the CAP theorem. The CAP acronym refers to three desirable properties in a replicated system:

- **Consistency**: This means that the data should appear identical across all nodes in the cluster
- **Availability**: This means that the system should always be available to receive requests
- **Partition tolerance**: This means that the system should continue to function in the event of a partial failure
Cassandra's Approach to High Availability

In 2000, computer scientist Eric Brewer from the University of California, Berkeley, posited that a replicated service can choose only two of the three properties for any given operation.

The CAP theorem has been widely misappropriated to suggest that entire systems must choose only two of the properties, which has led many to characterize databases as either AP or CP. In fact, most systems do not fit cleanly into either category, and Cassandra is no different.

Brewer himself addressed this misguided interpretation in his 2012 article, CAP Twelve Years Later: How the "Rules" Have Changed:

... all three properties are more continuous than binary. Availability is obviously continuous from 0 to 100 percent, but there are also many levels of consistency, and even partitions have nuances, including disagreement within the system about whether a partition exists.

In that same article, Brewer also pointed out that the definition of consistency in ACID terms differs from the CAP definition. In ACID, consistency refers to the guarantee that all database rules will be followed (unique constraints, foreign key constraints, and the like). The consistency in CAP, on the other hand, as clarified by Brewer refers only to single-copy consistency, a strict subset of ACID consistency.

When considering the various trade-offs of Cassandra's consistency level options, it's important to keep in mind that the CAP properties exist on a continuum rather than as binary choices.

The bottom line is that it's important to bear this continuum in mind when designing a system based on Cassandra. Refer to Chapter 3, Replication, for additional details on properly tuning Cassandra's consistency level under a variety of circumstances.
Summary
By now, you should have a solid understanding of Cassandra's approach to availability and why the fundamental design decisions were made. In the later chapters, we'll take a deeper look at the following ideas:

- Configuring Cassandra for high availability
- Designing highly available applications on Cassandra
- Avoiding common antipatterns
- Handling various failure scenarios

By the end of this book, you should possess a solid grasp of these concepts and be confident that you've successfully deployed one of the most robust and scalable database platforms available today.

However, we need to take it a step at a time, so in the next few chapters, we will build a deeper understanding of how Cassandra manages data. This foundation will be necessary for the topics covered later in the book. We'll start with a discussion of Cassandra's data placement strategy in the next chapter.
Where to buy this book
You can buy Cassandra High Availability from the Packt Publishing website.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.
Click here for ordering and shipping details.