Cassandra Design Patterns

Understand and apply Cassandra design and usage patterns, and solve real-world business or technical problems

Sanjay Sharma

Cassandra Design Patterns

Understand and apply Cassandra design and usage patterns, and solve real-world business or technical problems

Sanjay Sharma

[PACKT] open source*
PUBLISHING community experience distilled

BIRMINGHAM - MUMBAI

Cassandra Design Patterns

Copyright © 2014 Packt Publishing

Credits

Author
Sanjay Sharma

Reviewers
William Berg

Mark Kerzner

Acquisition Editors
Pramila Balan

Sam Wood

Commissioning Editor
Sharvari Tawde

Technical Editors
Mario D'Souza

Dennis John

Gaurav Thingalaya

Pankaj Kadam

Copy Editors
Tanvi Gaitonde

Dipti Kapadia

Kirti Pai

Stuti Srivastava

Project Coordinator
Akash Poojary

Proofreader
Simran Bhogal

Indexer
Hemangini Bari

Graphics
Abhinash Sahu

Production Coordinator
Nilesh R. Mohite

Cover Work
Nilesh R. Mohite

About the Author

Sanjay Sharma has been the architect of enterprise-grade solutions in the software industry for around 15 years and using Big Data and Cloud technologies over the past four to five years to solve complex business problems. He has extensive experience with cardinal technologies, including Cassandra, Hadoop, Hive, MongoDB, MPP DW, and Java/J2EE/SOA, which allowed him to pioneer the LinkedIn group, Hadoop India. Over the years, he has also played a pivotal role in many industries, including healthcare, finance, CRM, manufacturing, and banking/insurance. Sanjay is highly venerated for his technological insight and is invited to speak regularly at Big Data, Cloud, and Agile events. He is also an active contributor to open source.

I would like to thank my employer, Impetus and iLabs, and its R&D department, which invests in cutting-edge technologies. This has allowed me to become a pioneer in mastering Cassandra- and Hadoop-like technologies early on.

But, most importantly, I want to acknowledge my family, my beautiful wife and son, who have always supported and encouraged me in all my endeavors in life.

About the Reviewers

William Berg is a software developer for OpenMarket. He helps maintain the Apache Cassandra cluster, which forms part of their internal, distributed file storage solution.

Mark Kerzner holds degrees in Law, Math, and Computer Science. He has been designing software for many years and Hadoop-based systems since 2008. He is President of SHMsoft, a provider of Hadoop applications for various verticals, and a co-founder of the Hadoop Illuminated training and consulting firm, as well as the co-author of *Hadoop Illuminated*, *Hadoop illuminated LLC*. He has also authored and co-authored other books and patents.

> I would like to acknowledge the help of my colleagues, in particular Sujee Maniyam and last but not the least, my multitalented family.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit `www.PacktPub.com` for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

![PacktLib logo]

`http://PacktLib.PacktPub.com`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?
- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

Big Data has already become a buzzword in today's IT world and provides lots of choices for end users. Cassandra is one of the most popular Big Data technologies today and is used as a NoSQL data store to build web-scale applications in the Big Data world.

The idea behind this book is for Cassandra users to learn about Cassandra's strengths and weaknesses and, more importantly, understand where and how to use Cassandra correctly, so as to use its strengths properly and overcome its weaknesses.

One of the most critical decisions taken while writing the book was using the term design pattern to explain Cassandra's where and how usages. Traditionally, design patterns are linked to object-oriented design patterns made popular by Gang of Four (GoF).

However, in the context of this book, patterns refer to general, reusable solutions to commonly recurring software problems and will be used interchangeably for use case patterns, design patterns, execution patterns, implementation strategy patterns, and even applied design patterns.

What this book covers

Chapter 1, *An Overview of Architecture and Data Modeling in Cassandra*, discusses the history and origins of Cassandra. In this chapter, we understand the parentage of Cassandra in Amazon Dynamo and Google BigTable and how Cassandra inherits the best of the abilities as well as evolves some of its own. The importance of reading this chapter lies in the fact that all the subsequent chapters refer to the capabilities mentioned in this chapter that can be used for solving various business use cases in the real world.

Chapter 2, *An Overview of Case and Design Patterns*, offers an overview of all the patterns that will be covered in this book.

Chapter 3, *3V Patterns*, covers the known 3V or Volume, Velocity, and Variety model associated with Big Data and also how Cassandra fulfills all the requirements for building Big Data applications facing the 3V challenge.

Chapter 4, *Core Cassandra Patterns*, describes how Cassandra can be used to solve some interesting business problems due to its unique capabilities covering high availability, wide rows, and counter columns.

Chapter 5, *Search and Analytics Applied Use Case Patterns*, takes the next step and goes beyond using only Cassandra to solve some interesting real-world use cases. This chapter covers how Cassandra can be used easily with other Big Data technologies to solve various business problems.

Chapter 6, *Patterns and Anti-Patterns*, covers some design patterns and anti-patterns that can be used to solve various software problems. The chapter also describes how and where to use these patterns.

What you need for this book

The readers are advised to go through Cassandra basics before starting on the journey of understanding *Cassandra Design Patterns*. A brief and good book to start with is *Instant Apache Cassandra for Developers Starter*, *Packt Publishing* by *Vivek Mishra*.

Though, having prior knowledge of Cassandra is not mandatory, anybody with some background in any application design and implementation, and RDBMS experience will find it easy to relate to this book.

The book is not tied to any specific Cassandra version; however, some of the code examples refer to Cassandra Query Language (CQL). So Cassandra 2.0 and above is a preferred version for references.

Who this book is for

If you are an architect, designer, or developer starting with Cassandra, or an advanced user who is already using Cassandra and looking for a brief summary of the known patterns that Cassandra can be used to solve, this book is ideal for you.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "Dynamo supports the `get()` and `put()` functions."

A block of code is set as follows:

```
SELECT *
FROM temperature_ts
WHERE weather_station_id='Station-NYC-1'
AND date='2013-01-01';
```

> Warnings or important notes appear in a box like this.

> Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title through the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/support`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website, or added to any list of existing errata, under the Errata section of that title.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
An Overview of Architecture and Data Modeling in Cassandra

The demand from web-scale applications in today's social and mobile-driven Internet has resulted in the recent prominence of NoSQL data stores that can handle and scale up to terabytes and petabytes of data.

An interesting fact is that there are more than 100 NoSQL data stores in the software industry today, and clearly, Cassandra has emerged as one of the leaders in this congested arena, thanks to its distinct capabilities that include easy scalability and ease of usage.

Let us look at the Cassandra architecture and data modeling to figure out the key reasons behind Cassandra's success.

Understanding the background of Cassandra's architecture

Cassandra's architecture is based on the best-of-the-world combination of two proven technologies—Google BigTable and Amazon Dynamo. So, it is important to understand some key architectural characteristics of these two technologies before talking about Cassandra's architecture.

Before starting the discussion of these architectures, one of the important concepts to touch upon is the **CAP theorem**, also known as the Brewer Theorem—named after its author Eric Brewer. Without going into the theoretical details of the CAP theorem, it would be simple to understand that CAP stands for Consistency, Availability, and Partition tolerance. Also, the theorem suggests that a true distributed system can effectively cater to only two of the mentioned characteristics.

Amazon Dynamo

Amazon Dynamo is a proprietary key-value store developed at Amazon. The key design requirements are high performance and high availability with continuous growth of data. These requirements mean that firstly, Dynamo has to support scalable, performant architectures that would be transparent to machine failures, and secondly, Dynamo used data replication and autosharding across multiple machines. So, if a machine goes down, the data would still be available on a different machine. The autosharding or automated distribution of data ensures that data is divided across a cluster of machines. A very important characteristic of Amazon Dynamo design is peer-to-peer architecture, which means that there is no master involved in managing the data—each node in a Dynamo cluster is a standalone engine. Another aspect of Dynamo is its simplicity in data modeling, as it uses a simple key-value model.

So where does Dynamo stand with respect to the CAP theorem? Dynamo falls under the category of **Availability and Partition tolerance above Consistency** (**AP**). However, it is not absolutely true that Dynamo does not support Consistency—this cannot be expected from a production-grade, real-world data store. Dynamo uses the concept of **Eventual Consistency**, which means that the data would eventually become more consistent over time. Remember, Dynamo keeps replicas of the same data across nodes, and hence, if the state of the same dataset is not the same as its copies due to various reasons, say in the case of a network failure, data becomes inconsistent. Dynamo uses **gossip protocol** to counter this, which means that each Dynamo node talks to its neighbor for failure detection and cluster membership management without the need for a master node. This process of cluster awareness is further used to enable the passing around of messages in the cluster in order to keep the data state consistent across all of the copies. This process happens over time in an asynchronous way and is therefore termed as Eventual Consistency.

Google BigTable

Google BigTable is the underlying data store that runs multiple popular Google applications that we use daily, ranging from Gmail, YouTube, Orkut, Google Analytics, and much more. As it was invented by Google, BigTable is designed to scale up to petabytes (PB) of data and cater to real-time operations necessary for web-scale applications. So, it has real fast reads and writes, scales horizontally, and has high availability—how many times have we heard Google services failing!

Google BigTable uses an interesting and easy-to-understand design concept for its data storage—the data writes are first recorded in a **commit log** and then the data itself is written to a memory store. The memory store is then persisted in the background to a disk-based storage called **Sorted String Table** (**SSTable**). The writes are super fast because the data is actually being written to a memory store and not directly to a disk, which is still a major bottleneck for efficient reads and writes. The logical question to ask here is what happens if there is a failure when the data is still in memory and not persisted to SSTable. A commit log solves this problem. Remember that the commit log contains a list of all the operations happening on the data store. So, in the case of failures, the commit logs can be replayed and merged with the SSTable to reach a stage where all the commit logs are processed and cleared and become a part of the SSTable. Read queries from BigTable also use the clever approach of looking up the data in a merged view of the memory store and the SSTable store—the reads are super fast because the data is either available in the memory or because SSTable indexing returns the data almost immediately.

> **How fast is "super fast"?**
>
> Reads and writes in the memory are around 10,000 times faster than in the traditional disks. A good guide for every developer trying to understand the read and write latencies is *Latency Numbers Every Programmer Should Know* by *Jeff Dean* from Google Inc.

Google BigTable does have a problem; it falls under the category of **Consistency and Partition tolerance** (**CP**) in the CAP theorem and uses the master-slave architecture. This means that if the master goes down, there are chances that the system might not be available for some time. Google BigTable uses a lot of clever mechanisms to take care of high availability though; but, the underlying principle is that Google BigTable prefers Consistency and Partition tolerance to Availability.

Understanding the background of Cassandra modeling

Dynamo's data modeling consists of a simplistic key-value model that would translate into a table in RDBMS with two columns—a primary key column and an additional value column. Dynamo supports the `get()` and `put()` functions for the reads and the insert/update operations in the following API formats:

- `get(key)`: The datatype of `key` is bytes
- `put(key, value)`: The datatype of `key` and `value` is bytes

Google BigTable has a more complex data model and uses a multidimensional sorted map structure for storing data. The key can be considered to be a complex key in the RDBMS world, consisting of a key, a column name, and a timestamp, as follows:

```
(row:string, column:string, time:int64) -> string
```

The row key and column names are of the string datatype, while the timestamp is a 64-bit integer that can represent real time in microseconds. The value is a simple string.

Google BigTable uses the concept of column families, where common columns are grouped together, so the column key is actually represented as `family: qualifier`.

Google BigTable uses **Google File System (GFS)** for storage purposes. Google BigTable also uses techniques such as **Bloom filters** for efficient reads and compactions for efficient storage.

An overview of architecture and modeling

When Cassandra was first being developed, the initial developers had to take a design decision on whether to build a Dynamo-like or a Google BigTable-like system, and these clever guys decided to use the best of both worlds. Hence, the Cassandra architecture is loosely based on the foundations of peer-to-peer-based Dynamo architecture, with the data storage model based on Google BigTable.

Cassandra uses a peer-to-peer architecture, unlike a master-slave architecture, which is prone to **single point of failure** (**SPOF**) problems. Cassandra is deployed on multiple machines with each machine acting as a node in a cluster. Data is autosharded, that is, automatically distributed across nodes using key-based sharding, which means that the keys are used to distribute the data across the cluster. Each key-value data element in Cassandra is replicated across the cluster on other nodes (the default replication is 3) for high availability and fault tolerance. If a node goes down, the data can be served from another node having a copy of the original data.

Sharding is an old concept used for distributing data across different systems. Sharding can be horizontal or vertical. In horizontal sharding, in case of RDBMS, data is distributed on the basis of rows, with some rows residing on a single machine and the other rows residing on other machines. Vertical sharding is similar to columnar storage, where columns can be stored separately in different locations.

Hadoop Distributed File Systems (**HDFS**) use data-volumes-based sharding, where a single big file is sharded and distributed across multiple machines using the block size. So, as an example, if the block size is 64 MB, a 640 MB file will be split into 10 chunks and placed in multiple machines.

The same autosharding capability is used when new nodes are added to Cassandra, where the new node becomes responsible for a specific key range of data. The details of what node holds what key ranges is coordinated and shared across the cluster using the gossip protocol. So, whenever a client wants to access a specific key, each node locates the key and its associated data quickly within a few milliseconds. When the client writes data to the cluster, the data will be written to the nodes responsible for that key range. However, if the node responsible for that key range is down or not reachable, Cassandra uses a clever solution called **Hinted Handoff** that allows the data to be managed by another node in the cluster and to be written back on the responsible node once that node is back in the cluster.

The replication of data raises the concern of data inconsistency when the replicas might have different states for the same data. Cassandra uses mechanisms such as **anti-entropy** and **read repair** for solving this problem and synchronizing data across the replicas. Anti-entropy is used at the time of compaction, where **compaction** is a concept borrowed from Google BigTable. Compaction in Cassandra refers to the merging of SSTable and helps in optimizing data storage and increasing read performance by reducing the number of seeks across SSTables. Another problem that compaction solves is handling deletion in Cassandra. Unlike traditional RDBMS, all deletes in Cassandra are soft deletes, which means that the records still exist in the underlying data store but are marked with a special flag so that these deleted records do not appear in query results. The records marked as deleted records are called **tombstone** records. Major compactions handle these soft deletes or tombstones by removing them from the SSTable in the underlying file stores. Cassandra, like Dynamo, uses a **Merkle tree** data structure to represent the data state at a column family level in a node. This Merkle tree representation is used during major compactions to find the difference in the data states across nodes and reconciled.

> The Merkle tree or Hash tree is a data structure in the form of a tree where every non-leaf node is labeled with the hash of children nodes, allowing the efficient and secure verification of the contents of the large data structure.

Cassandra, like Dynamo, falls under the AP part of the CAP theorem and offers a *tunable* consistency level. Cassandra provides multiple consistency levels, as illustrated in the following table:

Operation	ZERO	ANY	ONE	QUORUM	ALL
Read	Not supported	Not supported	Reads from one node	Read from a majority of nodes with replicas	Read from all the nodes with replicas
Write	Asynchronous write	Writes on one node including hints	Writes on one node with commit log and Memtable	Writes on a majority of nodes with replicas	Writes on all the nodes with replicas

A summary of the features in Cassandra

The following table summarizes the key features of Cassandra with respect to its origins in Google BigTable and Amazon Dynamo:

Feature	Cassandra implementation	Google BigTable	Amazon Dynamo
Architecture	Peer-to-peer architecture, ring-based deployment architecture	No	Yes
Data model	Multidimensional map `(row,column, timestamp) -> bytes`	Yes	No
CAP theorem	AP with tunable consistency	No	Yes
Storage architecture	SSTable, Memtables	Yes	No
Storage layer	Local filesystem storage	No	No
Fast reads and efficient storage	Bloom filters, compactions	Yes	No

Feature	Cassandra implementation	Google BigTable	Amazon Dynamo
Programming language	Java	No	Yes
Client programming language	Multiple languages supported, including Java, PHP, Python, REST, C++, .NET, and so on	Not known	Not known
Scalability model	Horizontal scalability; multiple nodes deployment than a single machine deployment	Yes	Yes
Version conflicts	Timestamp field (not a vector clock as usually assumed)	No	No
Hard deletes/ updates	Data is always appended using the timestamp field—deletes/updates are soft appends and are cleaned asynchronously as part of major compactions	Yes	No

Summary

Cassandra packs the best features of two technologies proven at scale—Google BigTable and Amazon Dynamo. However, today Cassandra has evolved beyond these origins with new unique and enterprise-ready features such as **Cassandra Query Language** (**CQL**), support for collection columns, lightweight transactions, and triggers.

In the next chapter, we will talk about the design and use case patterns that are used in the world of Cassandra and utilize its architectural and modeling strengths.

2
An Overview of Case and Design Patterns

The Wikipedia definition of Software Design Pattern states:

> *In software engineering, a design pattern is a general repeatable solution to a commonly occurring problem in software design. A design pattern isn't a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations.*

Most of us have grown up using or listening about **Gang of Four** (**GoF**) patterns for **Object-Oriented Design and Programming** (**OOD/OOP**). These object-oriented design patterns are used to solve known problems of objects and are differentiated as creational, structural, and behavioral patterns.

On the same lines, we have also been using architectural patterns in software, which range from **Extract/Transform/Load** (**ETL**), **Service Oriented Architecture** (**SOA**), **Enterprise Application Integration/Enterprise Service Bus** (**EAI/ESB**), to **Transactional Data Store/Online Transaction Processing** (**TDS/OLTP**), **Operational Data Store** (**ODS**), **Master Data Hub** (**MDH**), and many more.

So, what do we mean by "Cassandra Design Patterns" in the context of this book? This book is an attempt to capture the common design patterns and use case patterns that can be solved using Cassandra. We will go through some design patterns that follow the traditional definition of design patterns. We will also include applied use case patterns that Cassandra solves better than other solutions in the software world.

As we move forward with the various design patterns and applied use case patterns, it is important to understand the 3V model as well as other key aspects such as high availability, schemaless design modeling, counting features, and document storage.

Understanding the 3V Model

The **3V Model** (**Volume**, **Velocity**, and **Variety**) is the most common model that defines the need for Big Data technologies ranging from Cassandra to Hadoop.

Simply speaking, the 3V model suggests that the traditional RDBMS or relational databases are ineffective and are not capable of handling huge data volumes, high ingestion rates, and different formats of data effectively. Hence, the need for alternate solutions defined under the category of Big Data Technologies.

RDBMS or relational tables cannot really be scaled for both read/write functions and huge data volumes—most of the application and SQL developers spend a lot of time normalizing and then denormalizing data, indexing for reads, and removing indexes for fast writes. Finally, they end up throwing the *relations* out of the relational database to remove the bottleneck of constrains.

> **How do we decide when RDBMS solutions become ineffective for huge data volumes, ingestion rates, or different formats?**
>
> There is no good answer to this except the fact that whenever we start throwing out *relations* in RDBMS (which means that you start removing constraints and indexes for faster writes or start denormalizing for faster reads by duplicating data), we should start considering alternate solutions. Another indication to start thinking about alternate solutions is the data volumes becoming too big and starting to impact query or write throughput SLA. Or, we start thinking about purging data so as to limit the data managed by our RDBMS solution. Adding more hardware such as RAM or CPU for vertical scalability is another indicator that alternate solutions such as NoSQL might be a better solution. Another indicator is the management of rapid schema changes, where changing the schema in RDBMS is nontrivial since it needs managing constraints, relationships, and indexes.

Most of the Big Data Technologies, especially NoSQL data stores, do not come with any relational support. Cassandra also does not support relationships, so it can avoid the limitations associated with a relational model.

Broadly speaking, the following use case patterns can be applied for Cassandra:

- **Massive Store Pattern**: This handles the Volume part of the 3V model
- **Ultra fast data Sink Pattern**: This handles the Velocity part of the 3V model
- **Flexi Schema Pattern**: This handles the Variety part of the 3V model

High availability

Most modern software solutions require 365x24x7 availability and Cassandra, inheriting its origin from Dynamo, strongly supports high availability.

In the traditional RDBMS world, the most common practice for high availability is database replication, usually in a **Master-Slave** mode. The master node is used for write operations while one or more slave nodes might be used for scaling read operations. **Master-Master** replication is usually avoided in RDBMS due to complexities around data synchronization and strict consistency requirements.

Cassandra uses **P2P** (**peer-to-peer**) architecture with auto-sharding and replication so that the master does not lead to a single point of failure, as mentioned earlier. Cassandra also has advanced capabilities for high availability across multirack/data center deployment such as replica factor and replica placement strategy for detecting and using network configuration efficiently.

These high availability features of Cassandra enable new use case patterns:

- Always ON pattern
- Using Cassandra as a nonstop data store

For Oracle users, the following table is an illustration of the complexities and differences between Oracle Replication, **Oracle RAC** (**Real Application Clusters**), and Cassandra:

Operation	Oracle Replication	Oracle RAC	Cassandra
Load balancing	Reads over multiple databases	Read/write over multiple instances	Automated over the entire Cassandra data store in/across data center(s)
Survivability	Good protection—natural disasters, power outages or sabotage, or both because the remaining replication sites can be in a different region geographically	Cannot protect against physical problems as RAC operates on a cluster or other massively parallel systems and is located in the same physical environment	Great protection within a single data center or multiple data centers using the replication strategy
Cost	High	High	No additional cost

Columns on the fly!

Many real-world applications today can fall under the category of time series applications that manage and store data generated over time, such as sever logs, performance metrics, sensor data, stock ticker, and transport tracking.

Some of the key characteristics of this dataset are that it grows over time and usually requires data to be queried or analyzed on time as the main criteria for that dataset. Usually, this data is used for showing trends and boundaries of data on a line graph using time as the x axis. Cassandra is a perfect fit for such kind of applications as it can store fast-ingesting high volume data using the concept of wide rows.

Wide rows are not used too much in the RDBMS world and might seem an alien concept for somebody coming from the traditional SQL modeling world. Cassandra offers the flexibility of adding columns on the fly being schemaless in nature, and allows columns to be used as an ever-growing placeholder for new data as it arrives. For this kind of usage, each new event is added as a new column rather than as a new row. This design also allows easy retrieval of data since all events against a single event source can be accessed as a single row lookup because all events are in a single row.

There are multiple strategies to store time series data in Cassandra and give us our next use case pattern:

- **Time Series Analytics Pattern**: With this, we can build time series applications using Cassandra

Count and count!

Cassandra supports a special kind of column used to store a number that can be used to count the occurrences of events incrementally and can work in a distributed atomic mode. This feature is very powerful if used correctly, as building a distributed counter solution is one of the not-so-easy changes to solve.

The operations available with counter columns are the increment and decrement counter values and form the basis of the following use case pattern:

- **Distributed Counters Pattern**: Building applications requiring distributed counters

Streaming analytics!

Cassandra is being successfully used along with other solutions such as Apache Storm to build streaming and complex event processing applications. Cassandra's fast ingestion and read capabilities are useful for building such solutions, and hence the reason for including the following use case pattern:

- **Streaming/CEP Analytics Pattern**: Building a social media network like graph applications

Needle in a haystack!

Cassandra can scale to millions and billions of rows over TBs and PBs of data. Its architecture allows it to find and return the data for a single row key blazingly fast, usually within milliseconds, irrespective of the scale of data volumes.

> Did you know that Cassandra was created as part of the Inbox Search for Facebook?

So, the origins of Cassandra actually started as a solution for Inbox Search for over 100 million users in the late 2000s.

Paying respect to the search origins of Cassandra, we define the following use case pattern:

- **Needle in a Haystack Pattern**: Using Cassandra for search applications

Graph problems!

Cassandra has also been used to solve Graph problems, which involves representing entities and relationships as nodes, edges, and properties to represent and store data.

Titan, an open source graph database built over Cassandra, is the reason for including the following use case pattern:

- **Graph Solution Pattern**: Building a social media network such as graph applications

Analytics

Since Cassandra can store massive amounts of data, running analytical queries is an important requirement to harness the huge amount of data.

Cassandra supports the leader (Hadoop, since its inception days) in Big Data analytics, and can be easily integrated with it. In fact, Cassandra has been used as a part of commercial distribution by DataStax—the company that commercially supports Cassandra to have many Hadoop components such as Hive, Sqoop, and Mahout, supported in Cassandra.

The use case patterns for analytics done over Cassandra systems are featured under the following pattern:

- Advanced Analytics Pattern

Blob store

Cassandra schema flexibility has multiple advantages, given that the only native datatypes that it supports are bytes. Although the recent addition of **Cassandra Query Language** (**CQL**) in Cassandra does provide high level datatypes including blob datatypes, the native datatype internally is still raw bytes. This same feature is useful for using Cassandra as a content or document store in certain cases when this blob size is in line with the following current limitations in Cassandra:

- Single column value may not be larger than 2 GB
- Single digits of MB are a more reasonable limit as there is no streaming or random access available for blob values

Even with these limitations, the huge scalability and read/write performance enables the following use case patterns:

- **Content/Document Store Pattern**: Building a content management service
- **Object/Entity Store Pattern**: Building an object store

Design patterns

There are some specific design patterns that have emerged as part of the discussion and are used across the Cassandra community. They mostly cover best practices around transaction management and data modeling designing in Cassandra. These core design patterns can be defined as:

- **CAP the ACID**: This applied design pattern covers how to use transactions correctly in Cassandra
- **Materialized View**: Designing query-driven tables to provide index-like capabilities using denormalization as the key concept

- **Composite Key**: Designing a composite row key to include more than one logical key
- **Anti-pattern Messaging Queue**: Why Cassandra should not be used as a messaging queue solution

We will also briefly cover some other design patterns such as Valueless columns, Collection fields, Sorting, Pagination, Temporal data, Triggers, and Unique records around some of the interesting and new features available in Cassandra since some recent releases.

We will windup our compilation of Cassandra patterns by including some patterns and anti-patterns around Cassandra's infrastructure/deployment problems.

Summary

We discussed some key use case and design patterns that we will cover in greater detail in the subsequent chapters. Cassandra's key capabilities enable a solution to be designed for these use cases in the form of some design patterns. We should also note that Cassandra has a lot of interesting use cases and can solve multiple challenges in the software world today.

We will start with the 3V model, with the use case and design patterns around it, in the next chapter.

3
3V Patterns

3V patterns are the foundational patterns that can be used as deciding factors on whether Cassandra should be used as a solution in the first place.

Strict believers of the design pattern definition might even disagree in considering these patterns as design patterns; however, these patterns represent the common recurring problems that Cassandra solves repeatedly and can also be used as smoke tests to figure out whether Cassandra is actually a good fit or not for a given use case.

We will be using the following standard pattern definition structure to define our patterns:

- Pattern name
- Problem/Intent
- Context/Applicability
- Forces/Motivation
- Solution
  - ° Optional consequences
  - ° Optional code samples
- Related patterns

Interestingly, the 3V patterns are applicable for many other NoSQL solutions.

These foundational patterns would most probably occur together, since the problems that these patterns solve also occur together.

Pattern name – Web scale store

A web scale store refers to web applications that serve and manage millions of Internet users, such as Facebook, Instagram, Twitter, and banking websites, or even online retail shops, such as Amazon.com or eBay, among many others. We will see if Cassandra can be a possible candidate for building such web scale applications.

Problem/Intent

Which is the data store that can be used to manage a huge amount of data on a web scale?

Context/Applicability

In today's Internet era, web applications are required to manage a lot of user traffic, given the fact that the number of Internet users are increasing at an exponential rate. This problem of managing data for the growing number of users is further aggravated by the fact that more than 85 percent of the human population today has mobile access, and these mobile users are slowly switching to smart phones. As the usage of smart phones becomes more prevalent, Internet usage and hence data movement will further increase. The current web and real-time applications require a data store that can manage a huge amount of data running into terabytes and petabytes.

> Wikipedia lists around 6.8 billion out of a 7 billion population as having access to mobile phones today! This amounts to around 87 phones per 100 people.

Forces/Motivations

- The traditional model using RDBMS as the web applications store is being challenged by the sheer data volume growth that RDBMS cannot scale to cost effectively.

- End users are becoming more demanding and are disapproving the traditional RDBMS approach of purging and archiving data after some time.
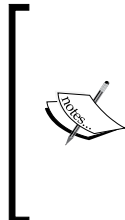
> Ever wondered why your bank statements are only available online for the last six months or a few years at the most? The obvious reason might be rooted in the fact that the RDBMS store being used is not efficient enough to store your transactions online beyond those durations. Therefore, you have to make special requests for historical statements, which are then served from an archive or data warehouse store.

- RDBMS relies on normalization and relationships for storing data efficiently. This allows lesser file storage to be used, which was the need of the hour when RDBMS principles were modeled 10 to 15 years ago when file storage was very expensive. RDBMS was also designed to run on single machines to easily manage **ACID** transaction compliance. However, this model does not allow data to be scaled beyond a single machine.

> RDBMS usually uses master-slave configuration and, in rare cases, master-master strategies for horizontal scalability. These strategies perform full replication of data; hence, the data size in such architecture would still be bound by the file storage available on a single machine instance.
>
> Manual sharding can be used to manually distribute data across multiple RDBMS stores, but it would require a lot of application coding and effort to take care of the arising issues.
>
> *Not everybody can afford Facebook's engineering teams capable of using a manually sharded MySQL store as its user profiles data store!*
>
> *-Sanjay Sharma*

- Relationships in RDBMS require constraints to be used for ensuring data consistency. Also, indexes are used as a powerful aid for fast queries. However, constraints and indexes become a bottleneck for fast writes. Thus, as data volumes grow, read and write tuning and optimizations slowly become a balancing act, where only one out of the many reads or writes can be kept happy.

- There are plenty of open source scalable NoSQL solutions available. However, which is the solution that one can really scale to huge volumes of data with lots of production use cases and is developer-friendly?

Solution

Cassandra is being used as a successful web scale data store solution, since it is proven at scale and used across hundreds of production sites. It has a very good read and write performance, and it is very versatile in solving most use cases, which is expected from any real-time web application.

Cassandra has an easy-to-use API and lots of clients in multiple languages, including enterprise-prominent Java. It also has support for **CQL** (**Cassandra Query Language**) for ease of usage and can be easily integrated with the existing web applications using high-level interfaces such as **Kundera**, **Spring Data**, **Playorm**, and many more interfaces, for easy migration from older RDBMS stores.

Cassandra is also easy to install and deploy and is cloud-deployment-friendly. Cassandra has been used in production applications deployed in AWS and Rackspace-like cloud providers. It is also supported on most **IaaS** (**Infrastructure as a Service**) cloud vendors such as Cloud Foundry, Google Compute Cloud, Azure, GoGrid, Terremark, and Savvis, because Cassandra can run on commodity-grade hardware, and is resilient to the network issues usually associated with public cloud environments.

> NetflixTM is the poster child for Cassandra usage as a scalable cloud-based web store, handling 60+ TBs of data. Interestingly, Netflix is the biggest single Internet traffic source as well as the largest pure cloud service for North America. Netflix has also shared the publicly available benchmark proving Cassandra's near-linear scalability.
>
> Today, Cassandra powers hundreds of big web scale applications for big names such as Netflix, eBay, Facebook, Twitter, Cisco, Comcast, Disney, Ericsson, Instagram, IHG, Intuit, NASA, PBS Kids, Travelocity, and others, which which the majority of us use, today.
>
> We can see many more examples of the production users of Cassandra at `PlannetCassandra.org`—a website dedicated to Cassandra information that lists more than 400 Cassandra users.

Consequences

Cassandra does come with certain nuances and possible considerations that apply to successful Cassandra usage:

- **Query-first-driven schema design**: Unlike traditional RDBMS schema modeling, all the possible queries should be identified first, and the schema model designed accordingly. This is necessary, because Cassandra supports only row key-based queries (the primary key in RDBMS world); hence, the row key design is the best way to allow the accomplishment of all the queries. It is also worthwhile mentioning that even though secondary index support is now available in Cassandra, secondary indexes are nothing but inverted index tables. So, as data volumes increase, the problems similar to read versus write optimization for indexes in RDBMS would also appear in Cassandra.

When to use or not use an index

Cassandra's secondary indexes work best on a table having many rows that contain the indexed values. Also, the indexing works best when the indexed column has a less number of unique values. In case there are too many unique values for the indexed column, there will be a lot of overhead on querying as well as managing the index.

Also, we would not use an index to query a huge volume of records for a small number of results, because if we create an index on a high-cardinality column having many distinct values, a query between the fields will incur many seeks for very few results. In these cases, it might be more efficient to manually maintain the table in the form of an index instead of using secondary indexes. For columns containing unique data, we might decide to use an index for convenience, as long as the query volume to the table having an indexed column is moderate and not under constant load.

This also means that we should not be creating an index for extreme low cardinality columns, such as a Boolean column, as each value in the index will become a single row in the index. For example, for Boolean values, we will end up with two big rows—one for true and the other for false values.

Cassandra does not have support for joins and hence denormalization is the norm. Similarly, there is no support for GROUP BY and should therefore be taken care of in the schema designing phase.

Cassandra also does not have direct support for sorting; hence, it is important to model the column design carefully to take care of sorting in query results. CQL does have ORDER BY support, but it has lots of restrictions; for example, it is supported only on a single column and this single column has to be the second column in a composite primary key.

- **Deployment across multiple machines**: Cassandra is designed for deployment across multiple machines and provides the best performance in more than three nodes. The reason why multiple machine deployment is important is because the number of nodes that can be used in parallel is a factor of the read and write performances in Cassandra. So, the more the number of machines, the more they can work in parallel for simultaneous reads and writes. This is a bit different from traditional RDBMS, where a single database engine is handling multiple reads and writes. While in Cassandra, each node is a data store engine capable of handling multiple requests across a single node. So, the cumulative requests that Cassandra can handle can be calculated by multiplying the number of machines in the Cassandra cluster with the number of requests per machine.

- **Use of Commodity hardware and JBOD (Just a Bunch Of Disks)**: Cassandra is best deployed across a cluster of commodity hardware nodes and with simple disk drives rather than using expensive hardware and storage solutions such as **SAN**. This is again different from traditional database usage, where expensive SAN is used for high availability. However, Cassandra has inbuilt support for high availability and relies on local hard drives for avoiding network-related issues present with SAN. Thus, local hard drives are faster and cheaper than SAN. Also, Cassandra scales horizontally, which means that instead of having a resource heavy machine with expensive hardware, it is better to use horizontal scalability by having multiple commodity machines. Having multiple machines also helps in better performance.

Our subsequent chapters will cover patterns on how to take care of the previously mentioned nuances and solve business problems successfully.

Pattern name – Ultra fast data sink

In today's world, we are not only talking about huge data volumes. Another problem to solve is how to ingest these high volumes as fast as possible so that the end users can be provided with a real-time experience. Let us see if Cassandra can serve as a fast ingest store.

Problem/Intent

Which data store to use that can handle high ingestion rates?

Context/Applicability

As the number of Internet users increase, there is also an increasing demand for capturing data from sensors and other machine-generated data sources. Today, data needs to captured from various sources at a mind boggling ingest speed of millions of events or transactions per second.

> We are sitting on the brink of a major data and IT revolution (Internet of Things), which talks about the near future where every object in the world can be RFID-tagged and can become a source of far bigger informational datasets, unlike today's world with human beings as the only source of data.
>
> We already see this happening all around us today with machines such as cars sending sensor data to centralized data stores for predictive maintenance-driven analytics.

Forces/Motivations

- Traditional RDBMS has not been designed for ultra fast ingestion of millions of inserts per second.

- A single machine deployment model in RDBMS usually restricts ingests from scaling horizontally. Hence, the limiting factor becomes disk I/O on a single machine.

- Web application middleware has since long been using stateless architectures that are capable of scaling horizontally. In traditional multitiered architecture, it is usually the RDBMS data tier that cannot be scaled horizontally and causes a bottleneck.
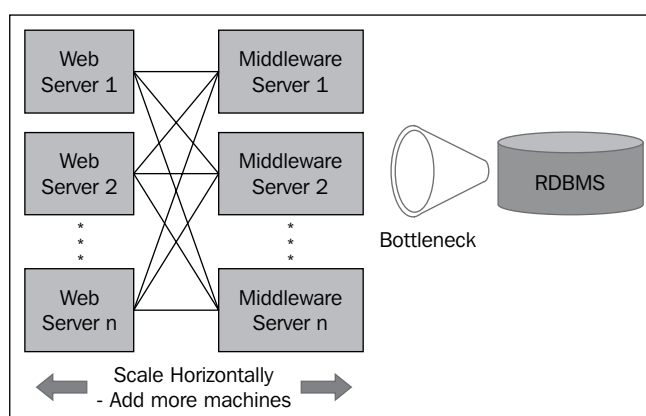

Traditional RDBMS-based web-tiered architecture

- Traditional RDBMS writes become more problematic as the data sizes increase due to the time spent in maintaining constraints and indexes.

Solution

Cassandra is unmatched with respect to fast ingestion of data, which is possible due to its memory store, sequential writes, tunable consistency, and parallel writes across cluster feature sets. Unlike RDBMS, Cassandra supports parallel writes where each node in a cluster is responsible for a specific key range. Also, writes in Cassandra are first written to a memory store and purged to a disk drive asynchronously. This memory-backed design allows the writes per node capacity to be easily scaled to 10k+ for normal complexity data sets. This, combined with parallel writes to multiple nodes in a cluster, theoretically allow a cumulative throughput of millions of inserts per second.
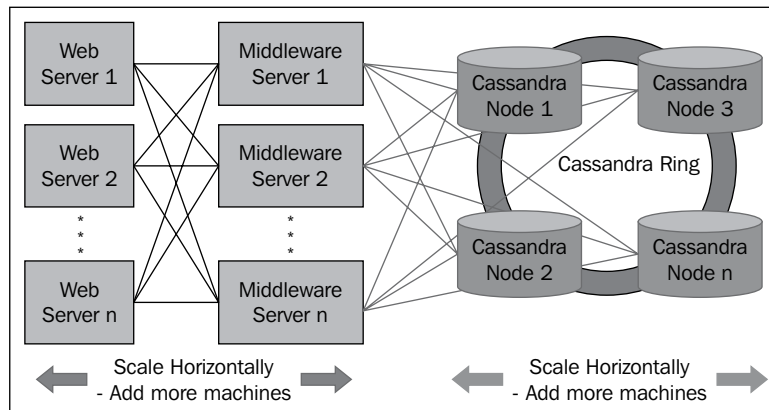
Cassandra-based web-tiered architecture

Netfix has shared some interesting benchmarks for Cassandra deployment in Amazon EC2 cloud with the open source community easily accessible through any Internet search. Netflix was able to achieve around 10k writes per second per node with mean latencies in milliseconds on 16 GB RAM extra large EC2 machines. The results were proven for 48, 96, 144, and 288 nodes, and 288 nodes could achieve 1 million writes per second.

Consequences

Enabling high ingestion rates in Cassandra does require careful planning around consistency and partitioning.

- Let us start with understanding tunable consistency considerations. Cassandra has a very powerful feature set of tunable consistency. A lower write consistency level starting with **ZERO**, **ANY**,
  ONE, **QUORUM**, and **ALL** usually have decreasing ingestion rates support.
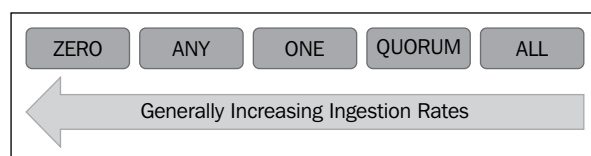
Write consistency level impact

- **Another consideration is scaling ingestion rates by adding more nodes and proper partitioning**: Cassandra reads and writes can be scaled easily by adding more nodes to a cluster. However, it is important to ensure that both reads and writes are spread along the cluster by the proper designing of the row key, which is the partitioning mechanism for Cassandra. Similarly, proper rebalancing of data to take care of data skews and ensuring a homogenous spread is important for both reads and writes.

Related patterns

This usage pattern is very closely related to the "web scale store" pattern and is usually used together.

Pattern name – Flexi schema

One of the challenges that business applications face today is rapid changes in requirements as well as new forms of end-user interactions with these applications. These rapid changes are still a challenge in the RDBMS world, especially semi-structured or flexible structure data in the form of JSON or XML, as it is difficult to convert these flexi-schema structures into the strict schema world of RDBMS. Let us see if Cassandra can provide some possible solutions for handling such flexi-schema problems.

Problem/Intent

Which data store to use that allows schema changes and multiple formats?

Context/Applicability

As Internet access becomes easily available, another major change happening in the Internet users' world is the use of an Omni device and Omni channel. This means that end users are using multiple channels and multiple devices to access the same services, and the same data is being generated from different sources. Agility, personalization, and rapid socio-driven changes are also enforcing the data structure to change more frequently than during the traditional era, when data schema changes were not prevalent.

It is also important to mention that there is a recent revelation in the world of data analytics, which was the data collected from silo databases and brought together has much more value than the silos of data. A loose structure is also required to join this data on multiple varying parameters for maximizing the analytical value add out of this data.

Forces/Motivations

- The RDBMS schema design follows a strict schema-first design. This means that it is difficult to change the schema once designed. Changes in a schema usually means including new fields, or modifying the existing fields, or changing the datatypes. Queries and analytics against such a static schema are bounded by the constraints of SQL syntax and modifying the existing schema for alternative analytics is difficult.

- RDBMS schemas are difficult to be altered since relationship constraints and indexes resist even minor changes in the schema.

- RDBMS also has a problem of storing null values, and most real-life data has lots of such null values.

- JSON and XML formats are increasingly becoming the standard for web and **SOA** (**Service oriented architecture**) applications. These formats have flexible schemas; hence, they are not very friendly to be managed in RDBMS schemas.

Solution

Cassandra uses a column-family-driven schema design approach, which means that columns can be added on the fly. In fact, each row in Cassandra is actually a flexible data structure in the form of a map where multiple key-value pairs can be stored against a single row key, and each row in a Cassandra table can have its own set of key-value pairs. An easy way of understanding this schema flexibility in the Cassandra data model is by knowing that the column name is also stored as part of the row data.
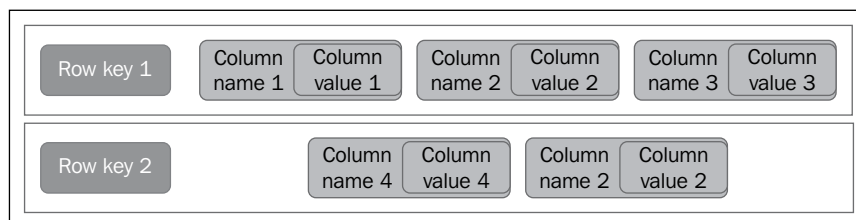
Flexi schema with column names as part of the data row

XML and JSON formats can be easily stored in the Cassandra data model by storing each key-value pair in in separate columns JSON/XML documents.

Consequences

Flexible schema management in Cassandra does require the following consideration:

- **Programming interface extensibility**: The middleware tier should be extensible to accommodate and use the changes in the schema without major code, preferably using configuration-driven design and coding style

Related patterns

This usage pattern is very closely related to the "web scale store" and "ultra fast data sink" patterns and is usually used together.

Summary

We covered the foundational patterns that can be used for assessing whether Cassandra is a good fit for solving some recurring problems that the web and real-time application designers face.

These patterns are applicable for the NoSQL world in general and talk about how to use Cassandra for solving these challenges.

In the next chapter, we will look at other important patterns such as highly available stores, time series analytics, content/document stores, and an atomic distributed counter service that Cassandra solves in ways better than any other solutions out there.

4
Core Cassandra Patterns

We will continue our journey in understanding technical use cases and how Cassandra solves the following problems:

- Highly available store
- Time series analytics
- Atomic distributed counter service

Pattern name – Highly available store

Building highly available applications has always been a challenge for both the hardware and the software world. This high availability refers to seamless end user experience even if the underlying application fails. This is usually achieved by transparently passing user access to a standby application providing the same service. Let us learn how Cassandra can enable you to build such highly available applications in this pattern.

Problem/Intent

Which data store should we use such that it is highly available even across multiple data centers at web scale?

Context/Applicability

Today's Internet era web applications require a lot of user traffic to be managed, given that the number of Internet users is increasing at an exponential rate. Also, the end users are truly global and need high availability to ensure competitiveness against possible competitors. One of the traditional, proven approaches for high availability is the deployment of applications across different data centers in an active-active mode, where each active data center deployment is independent but is always in a ready state to replace the other data center deployment as well as to work in parallel. This approach allows traditional web applications to be deployed across multiple data centers across geographical locations to ensure high availability.

An interesting aspect of today's web applications, especially social media applications, is deployments in public clouds, that usually have different high availability **SLA** (**Service level agreement**) than the web application itself. This is especially true for a web/middleware application tier where a machine instance that deploys the web/middleware tier can be restarted or reinstantiated but can take time running into minutes or more, which might be beyond the application SLAs of a couple of seconds or less. Cloud vendors typically provide hosting across geographically located data centers for faster access similar to **CDN** (**Content Delivery Network**) and also provide data replication services across data centers for disaster recovery. Hence, deployment across multiple data centers in a Cloud vendor is commonplace, where the web application is deployed in geographically located data centers.

Forces/Motivations

The traditional model of using **RDBMS** in a highly available mode generally involves deploying multiple machines in a master-slave and sometimes in a master-master architecture. The machines in the cluster usually replicate data between the different machines. This data copy is sometimes problematic due to various issues in networking.

> GoldenGate is a standard tool for data synchronization between Oracle machines in a cluster. One of the biggest challenges that enterprises face with multidatacenter cluster deployments is that the data synchronization process becomes slow between nodes across different data centers.

Web applications with a huge user base in the Cloud particularly run into issues as the hardware and network is usually shared in Cloud environments. This means that although the web application might not be impacted significantly due to shared CPU or RAM, there might be issues with respect to shared drive or network usage. So, most web applications in the Cloud should ideally be designed to counter the possible issues with network inconsistencies and high availability problems.

Solution

Cassandra was designed from the ground up to be deployable in multidatacenter and cloud-like environments, given its origins in Amazon Dynamo and Google BigTable. It has unmatched advanced features for data center deployment and provides fine-grained control over how the data is spread and replicated across different data centers.

Cassandra uses the concept of a **snitch** to allow how and what data is written to data centers and racks. The following snitch implementations are available out of the box in Cassandra:

- **Simple snitch**: This is used for a single data center or a single rack-based deployment.
- **Dynamic snitch**: This is used to dynamically find the best read replica based on the history of performance of the earlier reads of the replicas.
- **Rack inferring snitch**: This uses the IP address' third and second octets to determine the location of racks and data centers.
- **Property file snitch**: This uses a property file to define and name data centers, racks, and IP addresses of each node.
- **Gossiping property file snitch**: This uses a property file to define the data center and rack information for a local node and a gossip protocol to propagate this information to other nodes.
- **EC2 snitch**: This is used for Amazon Web Services (AWS) EC2-based cloud deployment in a single AWS EC2 region. It uses private IPs; hence, it is not usable across data centers.
- **EC2 multi region snitch**: This is used for multi region (multidatacenter) AWS EC2 deployments.

A replication strategy can be used to control where and how many replicas are placed in a data center across a multidatacenter deployment.

Cassandra is highly available as it uses a peer-to-peer architecture, so it does not have a single point of failure as in the case of a master-slave architecture. This, combined with advanced support for multidatacenter deployments, fine-grained replica placement, asynchronous replication, and read/write consistency level control, ensures that Cassandra can provide high availability for 4-nines more easily.

Example

A case study: An enterprise customer was facing issues with slow replication in an Oracle database backed web application with Oracle 11*i* servers deployed across multiple geographical locations using GoldenGate for replication. One of the data centers was solely for disaster recovery, and the high availability requirements were 99.99 percent. The latencies in data synchronization across the data centers were reaching 5 minutes or so under peak loads. The system was replaced by a Cassandra-based multidatacenter-based deployment with a data center-aware snitch. The new system showed data synchronization being achieved in 2-3 seconds even under peak loads. This new system also had very high availability, thanks to a higher replication factor.

> An interesting fact in the preceding case study was that the replication factor could be used to affect high availability. For example, for a 4 data center deployment with three nodes each per data center, a replication factor of 12 would mean that even if 11 machines go down, the system would still be available for reads.

Pattern name – Time series analytics

Time series refers to the data generated at regular intervals and constitutes a sizable number of business problems faced in enterprises. A stock ticker is a classic example of time series data, where stock prices keep on changing continuously. Today, in most stock markets across the world, stock prices are captured at millisecond levels since multiple buy/sell decisions might be happening at the same time, and even a small time lapse can result in millions of dollars gained or lost. Cassandra has lots of feature sets that make it a perfect candidate for time series analytics, and this is what we will be covering in this pattern.

Problem/Intent

Which data store should we use for time series analytics?

Context/Applicability

Time series data is the data captured at regular time intervals and is a standard way for capturing information of log captures and applications like stock prices. Time series data is useful for predicting and forecasting analytics over time. One of the bigger challenges with managing time series data is that while smaller intervals of time help in better analytics, the sheer amount of data that needs to be captured becomes too huge.

> Sensor data generated in manufacturing processes is a classic example of time series data where millisecond or even nanosecond time series data analytics can be used for predictive preventive maintenance.
>
> *Consider your car becoming intelligent to caution you to take it to a repair shop as the time series forecast analyzed from the car's sensors are predicting an engine failure!*

Computer infrastructure monitoring is a multibillion dollar industry and is highly characterized by time series data capture and the ability to provide analytical capabilities around trend and root cause analysis. This infrastructure monitoring of computer systems (operating systems, applications, hardware, network, and so on) usually requires data to be captured at second or millisecond levels. And, depending on the number of deployed machines, systems, and metrics, this data capture can run into millions to hundreds of billions of data points per day!

> Websites like BOX and TUMBLR capture tens of billions of data points per day!

Forces/Motivations

Time series data capture involves two distinct challenges—possibly millions of inserts per second as well as huge data volumes resulting from storage of billions of data points per day. Both these challenges are difficult to solve using traditional RDBMS technologies.

Also, the usual way of accessing time series data is by using graphical interfaces; hence, a relational model of time series data might become cumbersome for building queries required for graphs.

Solution

Cassandra writes data sequentially to a disk and in a sorted way in SSTable files. This allows data retrieval by row key and range, which is really efficient, thanks to minimal disk seek. Also, remember that Cassandra is designed to use wide rows. This means that a row in Cassandra can potentially have around 2 billion columns. This property is highly applicable for capturing and building time series data applications.

Example

Let us use the classic example of a weather forecasting system where temperature readings need to be captured for each and every weather station. The schema for such an application can be modeled in the following way:
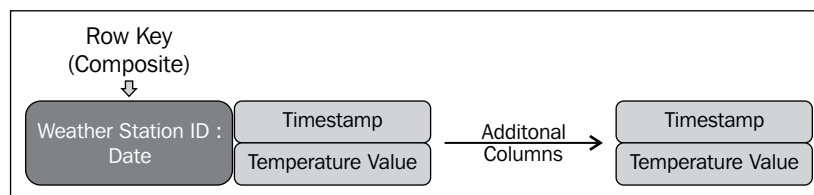
The CQL (Cassandra Query Language) DDL (Data Definition Language) table creates scripts that represent this data model as shown in the following query:

```
CREATE TABLE temperature_ts (
weather_station_id text,
capture_date text,
capture_time timestamp,
temperature int,
PRIMARY KEY ((weather_station_id,capture_date),capture_time)
);
```

The following screenshot is an example of the data captured using the previous data model schema:


---

[38]

The following insert commands represent the CQL equivalent for this data capture:

```
INSERT INTO temperature_ts (weather_station_id, capture_date,
capture_time, temperature)
VALUES ('Station-SFO-1','2013-01-01','2013-01-01 00:00:00','65');
INSERT INTO temperature_ts (weather_station_id, capture_date,
capture_time, temperature)
VALUES ('Station-SFO-1','2013-01-01','2013-01-01 00:10:00','64');
..
..
INSERT INTO temperature_ts (weather_station_id, capture_date,
capture_time, temperature)
VALUES ('Station-SFO-1','2013-01-01','2013-01-01 23:50:00','62');
INSERT INTO temperature_ts (weather_station_id, capture_date,
capture_time, temperature)
VALUES ('Station-NYC-1','2013-01-01','2013-01-01 00:00:00','28');
INSERT INTO temperature_ts (weather_station_id, capture_date,
capture_time, temperature)
VALUES ('Station-NYC-1','2013-01-01','2013-01-01 00:10:00','30');
..
..
INSERT INTO temperature_ts (weather_station_id, capture_date,
capture_time, temperature)
VALUES ('Station-NYC-1','2013-01-01','2013-01-01 23:50:00','32');
```

Data retrieval is simple and can be represented by the following query:

```
SELECT *
FROM temperature_ts
WHERE weather_station_id='Station-NYC-1'
AND date='2013-01-01';
```

Another example of a time series application for log monitoring is an open source project **kariosdb** hosted on Google Code. This application supports Cassandra as the backend store and allows millisecond level events to be stored. The row key in this data store consists of a metric name, row timestamp, and concatenated string of tags. A single row can store three weeks of data for a single metric, which amounts to 1,814,400,000 columns.
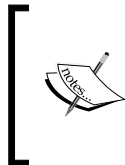
> OpenTSDB is another open source time series data store. The next generation of OpenTSDB started as a rewrite of some of the core features and was initially named OpenTSDB2. However, it was later renamed to kariosdb—hence, kariosdb is actually the next generation version of OpenTSDB.

Pattern name – Atomic distributed counter service

Counters are simple mechanisms to keep a track of events or an inventory of items in any business use case. Counters enable the avoidance of expensive aggregation queries by preaggregation at an accessible location. Building a distributed counter has traditionally been a daunting software design task, and hence, the reason to include this pattern in this chapter in order to see how Cassandra can solve this problem.

Problem/Intent

How to build a distributed counter service?

Context/Applicability

Distributed counters are a major requirement for many applications, particularly e-commerce applications. These services are required for quickly showing and tracking user events or actions. Keeping a track of videos or website hits is a popular use case for counters, where running a "count" query in the underlying database would be too expensive and time consuming. Hence, video or web page hits are managed as distributor counters for quick presentation on the user websites.

Forces/Motivations

Designing and implementing a counter service is a complicated task due to the dependency on atomic updates, where at a given time, only one process/thread can update the counter.

Solution

Cassandra has support for a special type—**counter**. A counter is a special kind of column used to store a number that incrementally counts the occurrences of a particular event or process. Counter column tables must use the counter datatype. Counters may only be stored in dedicated tables.

Data is loaded into a counter column using the UPDATE command, instead of INSERT. UPDATE is also used for increasing or decreasing the value of the counter.

Example

Let us use an example for keeping a track of the videos viewed/downloaded on a media website.

The keyspace for this counter service can be created as shown in the following query:

```
CREATE KEYSPACE counterkeyspace WITH REPLICATION =
{ 'class' : 'SimpleStrategy', 'replication_factor' : 3 };
```

The table for this counter service can be created as shown in the following query:

```
CREATE TABLE counterkeyspace.video_view_counts
  (counter_value counter,
  url_name varchar,
  video_name varchar,
  PRIMARY KEY (url_name, video_name)
);
```

Data can be loaded into this counter table as shown in the following query:

```
UPDATE counterkeyspace.video_view_counts
 SET counter_value = counter_value + 1
 WHERE url_name='www.youtube.com' AND video_name='cassandra';
```

The counter can be retrieved as shown in the following query:

```
SELECT * FROM counterkeyspace.video_view_counts;
```

The output obtained is shown in the following listing:

```
 url_name          | video_name | counter_value
-------------------+------------+---------------
 www.youtube.com   |  cassandra |             1

(1 rows)
```

The counter value can be incremented as shown in the following query:

```
UPDATE counterkeyspace.video_view_counts
  SET counter_value = counter_value + 2
  WHERE url_name='www.youtube.com' AND video_name='cassandra';
```

Summary

We covered some of the core use case patterns that can be easily catered to by some core capabilities available as part of Cassandra.

In the next chapter, we will look at other important patterns such as highly available store, time series analytics, content/document store, and atomic distributed counter service that Cassandra solves in ways better than any other competitive solutions out there.

5
Search and Analytics Applied Use Case Patterns

So far, we have seen some core and applied design patterns using Cassandra's unique capabilities as a leading NoSQL store. Now, let us look at some cases where other technologies and solutions can be used along with Cassandra for additional business use cases.

Cassandra is being used in conjunction with other leading solutions to solve many challenging use cases including the following:

- Streaming/CEP analytics
- Needle in a haystack/search engine
- Graph problems
- Advanced analytics

Pattern name – Streaming/CEP analytics

Streaming analytics and **Complex Event Processing** (**CEP**) have traditionally been used as standalone applications. However, combining streaming analytics with real-time analytics provided by Cassandra offers a new way of enabling an all-round analytical view of data throughout its lifecycle as it flows into a data store as well as after it gets persisted in the data store.

Problem/Intent

How can Cassandra be used for streaming/CEP analytics?

Context/Applicability

Streaming analytics is the process of extracting information from continuous data records and is very helpful in building low-latency systems, especially applications where early, real-time alerts and responses are a major requirement. An example can be a manufacturing process stopped in case of some alert raised as a part of sensor feed analysis.

Forces/Motivations

Traditional streaming and CEP engines are usually difficult to scale as CEP engines work on rolling time windows and are restricted by CPU/memory constraints on a single machine. The possible combination of Cassandra along with a Streaming/CEP analytic platform could be potentially useful for solving multiple use cases. This combination would allow us to use Cassandra's real-time analytics capability along with streaming analytics.

Solution

Cassandra can easily be used in conjunction with popular parallel processing streaming solutions for building applications that can handle streaming analytics as well as real-time analytics. Cassandra is also great for high ingest rates, so multiple data streams can be easily supported by Cassandra for fast storage.

The following figure shows a high-level architecture for representing such a solution:
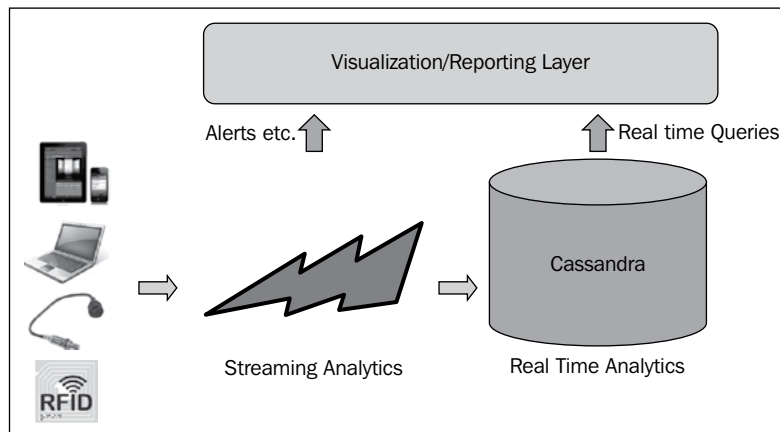
The reference architecture depicts machine data being captured from multiple sources and being ingested and processed though a streaming analytics platform. The streaming solution works as a pass-through for the streaming data as it is finally ingested in the Cassandra data store for long-term storage and analytics.

This solution can be used in the following ways:

- A streaming solution for alert management using a rule engine and CEP along with Cassandra for real-time analytics and long term storage.
- A streaming solution for alerts as well as an aggregation solution for using Cassandra as an aggregator/summarization data store. This pattern solves the problem of aggregate functions being difficult to implement in Cassandra.

For example, Apache Storm is an open source popular streaming solution that can be easily used in conjunction with Cassandra to provide an implementation of the reference architecture involving a streaming engine with a real-time engine.

Some of the characteristics of such a solution are as follows:

- Storm can run across multiple machines as a distributed grid engine for streaming analytics
- It can be scaled linearly by adding more machines in a way similar to Cassandra
- It is easy to integrate with other solutions such as Esper, which is an open source SQL supporting the CEP engine
- It easily integrates with Cassandra

Storm-Cassandra is a standard bolt available in open source and offers an easy implementation for integration with Cassandra. The `CassandraBolt` class provides a convenience constructor that takes a column family name and row key field value as arguments, as shown in the following code:

```
IRichBolt cassandraBolt = new CassandraBolt("myColumnFamily",
"myRowKey");
```

This statement creates a `CassandraBolt` object that writes to the `myColumnFamily` column family, and will look for/use a field named `myRowKey` in the `backtype.storm.tuple.Tuple` objects that it receives as the Cassandra row key.

For each field received in the `backtype.storm.tuple.Tuple` object, the `CassandraBolt` object will write a column name/value pair. For example, for a tuple value of {rowKey: 54321, col1: "col1val", col2: "col2val"}, the following Cassandra row would be seen in `cassandra-cli`:

```
RowKey: 54321
=> (column=col1, value=col1val, timestamp=1921236503081009)
=> (column=col2, value=col2val, timestamp=1921236503081102)
```

Cassandra counter columns are also supported by this Storm-Cassandra implementation.

So, a possible design pattern can involve using Storm for range window aggregations such as keeping count of the URL or videos for the last-hour window and storing hour-wise counters in Cassandra using a roll-up aggregator.

The following Cassandra bolt implementation can be used for counters:

```
CassandraCounterBatchingBolt myRollUpBolt = new
CassandraCounterBatchingBolt(
          "myColumnFamily", "myRowKeyField",
"myIncrementAmountField" );
```

This statement will create a bolt that will write to the `myColumnFamily` column family, and will use a field named `myRowKeyField` in the tuples that it receives.

Given a tuple {rowKey: 12345, IncrementAmount: 1L, IncrementColumn: 'SomeCounter'}, the `myCounter` counter column will be incremented by 1L.

Pattern name – Needle in a haystack/ search engine

Search applications are becoming an important tool across enterprises, enabling business users to access the right information at the right time. Since the data volumes across enterprises are becoming huge, searching for the right data is seemingly similar to searching for a needle in a haystack. Also, more importantly, the results of this search should be available in real time for the information to be used as early as possible. Cassandra can offer some great options for enabling such search capabilities.

Problem/Intent

How can Cassandra be used for search engine-like solutions?

Context/Applicability

Enterprise search is becoming a key business requirement across organizations as a means of centralized knowledge management. A search engine usually comprises an indexing mechanism and some means of querying capability over that index.

Forces/Motivations

Cassandra is great for storing huge amounts of data and fast reads, but it does face a challenge in handling complex queries on multiple fields.

An interesting fact is that Cassandra was created for Facebook as a search engine and was used to store the indexes to allow searches for keywords. The Facebook challenge was about storing reverse indices of Facebook messages that Facebook users send and receive while communicating with their friends on the Facebook network.

Also, proven search engines such as Solr and Elastic Search are built over Lucene technologies that have specialized search capabilities such as wild, faceted search.

So, a combination of Cassandra with search capabilities can be a useful solution for multiple search-related use cases across enterprises.

Solution

As mentioned earlier, Cassandra's origin is rooted in being created as a search engine for Facebook messaging. This involved creating an inverted index with values as row keys against primary keys in the following way:
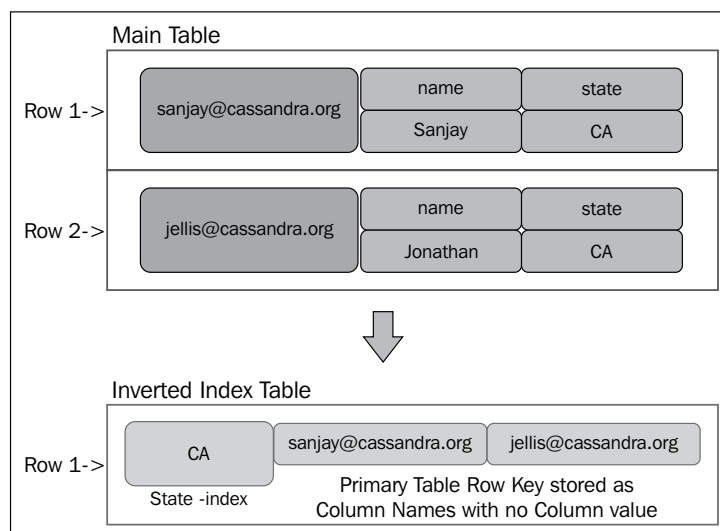
The schema example depicts an inverted index table for a query on a state field. The inverted index table schema uses the design pattern of using column names only, without the need for column values. The problem with such an inverted index approach is that complex queries on multiple fields are still a problem.

Please note that Cassandra now has inbuilt secondary indexing that supports inverted indexing capability since some recent releases. So instead of the manual implementation of inverted index creation, a secondary index can be created easily as follows:

```
CREATE INDEX state_key ON users (state);
```

This index allows the query to be executed on the state field directly as follows:

```
SELECT * FROM users where state="CA";
```

For advanced search capabilities, it is advisable to use Cassandra in combination with a full-fledged indexing and search engine as creating too many secondary indexes in Cassandra will, in the end, result in the same problems faced by the traditional RDBMS.

A typical deployment of Cassandra with an indexing/search engine would involve the source application sending data to be stored in Cassandra while the index would be created and stored in the search engine's index store.

The following solutions are possible examples of implementing a search engine solution along with Cassandra:

- **Cassandra along with SOLR integration**: This approach would require the middle tier of the business application to write data to Cassandra along with creating a SOLR document that can be submitted to the SOLR search engine. The deployment would consist of Cassandra servers deployed alongside SOLR servers. This approach does have the complexity of maintaining data sync between the Cassandra store and the SOLR index.

- **DataStax Enterprise**: DataStax is a commercial organization providing enterprise distribution and support for Cassandra. The enterprise distribution includes advanced search capability and includes a tight integration between Cassandra and Lucene indexes. A distinct advantage of this approach is that the DataStax solution takes care of the data state sync between the Cassandra store and the Lucene search indexes. In fact, the tight integration abstracts the complexity of creation of the index document by enabling auto-indexing of Cassandra columns as defined by the end user.

- **Kundera**: This is an open source, high level client library that supports JPA-compliant **Object Relational Mapping** (**ORM**) over Cassandra. Kundera also allows indexing on entity fields and runs Lucene queries on the indexed fields. Kundera has recently added support for a popular search engine (Elastic Search) and it would be interesting to see if this integration can be used to simplify Cassandra's integration with Elastic Search.

Pattern name – Graph problems

Another area of interest facing the software world today due to rising social media growth is graph analysis. Graph analytics has been a part of statistical and analytical solutions, traditionally in the scientific and research community. However, social media is playing a powerful role in understanding customer profile better; graph analytics is becoming a part of solving business problems. We will now see how Cassandra can be used for building such graph analytical solutions.

Problem/Intent

How can Cassandra be used to solve graph problems?

Context/Applicability

Graph problems are very pertinent in this world of social media where social interactions can be easily designed in the form of social graphs. A graph structure uses nodes, edges, and properties to represent and store data.

Forces/Motivations

Cassandra is a document store and is not directly suited for storage as a graph store. However, Cassandra does have the capability of storing huge amounts of data and their fast retrieval. This fast retrieval capability over large volumes of data along with the wide row feature can be used to build a graph data store over Cassandra.

Solution

Cassandra can be used to store graph entities as vertexes with node properties and relationships as edges with wide rowed columns. This pattern of storing edges and properties against a node as a vertex has been used in a data store called Titan, which is an open source implementation of a graph database using Cassandra as shown in the following figure. Titan supports Cassandra as a persistent store since it relies on Cassandra's high availability and linear scalability features.
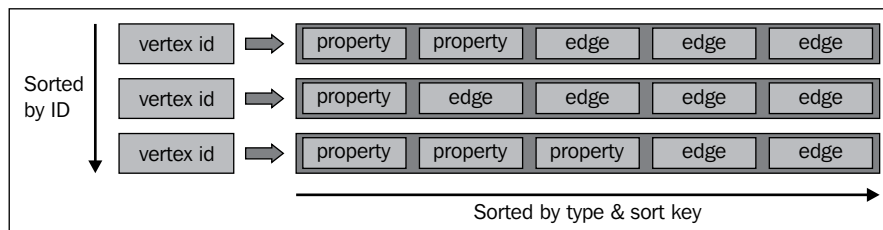
Pattern name – Advanced analytics

In today's world, advanced analytics refers to using statistics and machine learning to get more insights into the data. Predictive analytics or forecasting has many uses for solving real-world business use cases, such as customer acquisition, preventive maintenance, and campaign planning. Let us now look at how Cassandra can be used alongside other advanced analytical solutions.

Problem/Intent

How can Cassandra be used for advanced analytics?

Context/Applicability

Cassandra is predominately a real-time data store used for **OLTP** applications. Advanced analytics usually involves running complex batch queries and is useful for solving multiple business use cases.

Forces/Motivations

Cassandra is a real-time store and does not have a strong aggregation framework that is usually required for advanced analytics. There has been a recent addition to Cassandra's capabilities in the form of trigger support, but having a full batch-oriented analytical mechanism is mandatory for solving all analytical challenges.

Hadoop has recently emerged as the standard batch analytical solution for Big Data and hence Cassandra's integration with Hadoop is important for solving business problems using real as well as batch analytics.

Solution

Cassandra's integration with Hadoop is a successful pattern applied multiple times across production use cases of Cassandra to enable advanced analytical capabilities.

Cassandra supports integration with Hadoop in various ways and we can choose between the following options depending on the technical and business requirements:

- **Cassandra's direct integration with Hadoop**: Cassandra provides `ColumnFamilyInputFormat` and `ColumnFamilyOutputFormat` classes that can be used for direct integration of Cassandra with Hadoop from map reduce programs. This approach involves data being read from Cassandra column families in map reduce mappers directly and does include data movement.

- **DataStax Enterprise**: DataStax includes an implementation of Hadoop **Distributed File System** (**HDFS**) in the form of **Cassandra File System** (**CFS**), where data is stored in Cassandra column families. DataStax does provide support for Hive, PIG, and Sqoop, which are Hadoop ecosystem components. DataStax deployment does not require data movement as the data in HDFS-equivalent is also stored in Cassandra.

- **Data warehousing approach**: This is similar to the traditional data warehousing approach, where data is copied from OLTP stores to data warehouses. Here, in a similar fashion, data from the Cassandra OLTP store is copied and/or moved to the Cassandra store. This approach should be used only if the first approach does not work due to various issues such as network restrictions. The approach might involve exporting data from Cassandra using export tools such as the new support for the `COPY TO` CQL command with the following syntax:

```
COPY <tablename> [ ( column [, ...] ) ]
     TO ( '<filename>' | STDOUT )
     [ WITH <option>='value' [AND ...] ];
```

Summary

In this chapter, we saw some powerful combinations of using Cassandra along with other solutions and tools to solve a variety of business challenges. We saw how Cassandra can be utilized for building search applications and also for playing a strong role in an upcoming technology area—streaming analytics.

Cassandra can also be easily integrated with today's de facto leader of big data batch analytics—Hadoop. This allows us to build advanced complex analytical applications, providing both real time as well as batch analytical capabilities.

As Cassandra's popularity grows, many other solutions might also provide compatibility and integration with Cassandra to use its core capabilities around fast ingestions, fast reads, and high scalability and availability.

We covered some of the core use case patterns, which can be easily catered by some core capabilities available as a part of Cassandra.

In the next chapter, we will look at some patterns that should be used carefully and also some anti-patterns that would tell us how not to use Cassandra.

6
Patterns and Anti-patterns

So far, we've looked at use cases and design patterns where Cassandra is a really good candidate. However, as with any other solution, Cassandra can be used in the wrong ways. So, in this chapter, we will be covering some patterns that should be used carefully and those that could result in the wrong usage of Cassandra.

The software industry also uses a term called **anti-patterns**, which are basically repetitive usage of certain patterns that are actually wrong and should be avoided at all costs.

We will now look at some other interesting patterns that can be easily turned into anti-patterns if misused as well as some known anti-patterns.

Pattern name – Content/Document store

Document or content stores are used in the software world for storing artifacts and documents in the formats of Word, Excel, PDF, HTML, and so on, so that they can be searched, used, read, and managed in a better way than if we rely on traditional filesystems. Also, centralized content stores are becoming the need of the hour; individuals can collaborate more easily using this central store. Let us see whether or not Cassandra can be a potential candidate for building a content or document store.

Problem/Intent

Which data store to use as a content/document store?

Context/Applicability

In today's world, content management systems face challenges related to huge data volumes and high reads. Content management systems' data stores need to store the document content and also the metadata related to the content. The metadata associated with the content usually requires search capabilities on tags as well as workflow management in complex systems.

Forces/Motivations

Content management systems are usually "write once and read multiple times", but content updates are also an important requirement. The updates of the content are managed through version management. This requires metadata management through a fast read/write store.

Solution

Cassandra's data modeling provides a solution to this content storage by allowing the storage of raw bytes. This byte storage model allows content or documents to be stored as raw bytes as column values, while the metadata associated with the content can be stored as key-value pairs in wide rows against the document itself. In fact, CQL now has support for the **BLOB** (**binary large object**) type.

However, this pattern of storing bytes at the column level should be used with caution. This is because Cassandra is not optimized specifically for large-file or BLOB storage. So, storing large objects in Cassandra has to be done carefully, since it can cause excessive heap pressure and hotspots. However, files of around 64 Mb and smaller can be easily stored in the database without having to be split into smaller chunks.

Example

A client API for Cassandra called **Astyanax** has special support for storing large objects. This client API provides utility classes that address this issue by splitting up large objects into multiple keys and handles, which fetch those objects in random order to reduce hotspots.

An object can be stored using the following approach:

```
ObjectMetadata meta = ChunkedStorage.newWriter(provider, objName,
    someInputStream)
.withChunkSize(0x1000)    // Optional chunk size to override
    // the default for this provider
.withConcurrencyLevel(8)  // Optional.Upload chunks in 8 threads
.withTtl(60)              // Optional TTL for the entire object
    .call();
```

An object can be read using the following approach:

```
ObjectMetadata meta = ChunkedStorage.newInfoReader(provider, objName).
call();
ByteArrayOutputStream os = new ByteArrayOutputStream(meta.
getObjectSize().intValue());

// Read the file
meta = ChunkedStorage.newReader(provider, objName, os)
.withBatchSize(11) // Randomize fetching blocks within a batch.
.withRetryPolicy(new ExponentialBackoffWithRetry(250,20))
                            // Retry policy for when a chunk isn't
available. This helps implement retries in a cross region setup where
replication may be slow
.withConcurrencyLevel(2) // Download chunks in 2 threads
    .call();
```

Caution

This pattern can easily be misused if the document/content size is bigger than a few kilobytes. As advised, Astyanax-like APIs have advanced support for breaking bigger content into smaller, manageable chunks to reduce the risks involved with bigger column values.

Pattern name – Object/Entity store

Entity or object stores are more suitable for managing objects in an object-oriented architecture and design paradigm, compared to the previous pattern. There was a rise in object databases in the early 2000s, but these data stores did not really gain popularity. One of the reasons behind the apparent failure of such data stores was the popularity of **ORM** (**Object Relational Mapper**) tools, which allowed **OOP**-managed (**Object-oriented Programming**) objects to be stored in the RDBMS. **Hibernate** is one of the popular examples of such an ORM tool, which allows Java objects to be stored in RDBMS. Let us see what the best practices are for handling objects or entities in Cassandra.

Problem/Intent

How do we use Cassandra as an Entity/Object store?

Context/Applicability

Object-oriented programming is a prominent standard of enterprise-grade development and implementation of software. The Java language is the poster child of the OOP model's success.

Forces/Motivations

The storage of objects in Cassandra can be handled in multiple ways, such as the following:

- A BLOB store, such as a JSON object, in one column
- Multiple columns
- Different programming models can also be used for interacting with the Cassandra store
- A Cassandra native model using columns and rows
- An object-oriented model:
    - ° Using high level API interfaces
    - ° Using ORM tools

Solution

Cassandra should always be used for storing objects in multiple columns and not as BLOBs such as JSON or XML representations. The solution is to represent the object in key-value pairs and store these as columns against the object identifier or store the primary key as the row key.

Cassandra's schemaless nature allows columns to be added on the fly so that developers do not have to worry about modifying schema for storing new columns. Cassandra recommends denormalization for managing relationships between different data entities and now also has support for advanced structures, such as maps and collections, as part of CQL.

The ORM usage of Cassandra should be used only in cases where OOP-based development and the easy migration of existing Java applications is more important than redesigning the application from scratch. Caution should be observed in selecting the right ORM tool, so as to use the Cassandra feature set as well as OOP.

Kundera—an open-source ORM supporting Cassandra—has some capabilities that ensure that users are using the nest principles of Cassandra design modeling, such as denormalization using embedded tags, and also ease of development, since Kundera is a **JPA** (**Java Persistence API**). JPA compliance means that any existing Hibernate developer can easily start developing Cassandra applications as well as migrate existing RDBMS applications to Cassandra.

Caution

Using an ORM does not mean that the developer does not care about the best practices and principles for Cassandra schema modeling. So, using the ORM incorrectly can still result in the same bottlenecks that traditional RDBMS run into.

Pattern name – CAP the ACID

Traditional RDBMS applications have transactional or **ACID** (**Atomicity**, **Consistency**, **Isolation**, and **Durability**) support as an integral part of the solution. However, NoSQL solutions, including Cassandra, use the **CAP** (**Consistency**, **Availability**, and **Partition tolerance**) theorem as their foundation. Let us see how real-world applications can be tackled using Cassandra's CAP-based architecture.

Problem/Intent

How can we use Cassandra for seemingly transactional problems?

Context/Applicability

Many enterprise-grade applications seem to be dependent on transactional ACID support usually provided by the underlying RDBMS. How can we build such applications using Cassandra?

Forces/Motivations

Cassandra does not support ACID in the true RDBMS sense, but does provide support for row-level transactions.

Cassandra sacrificed support joins or foreign keys for scalability and performance and consequently does not offer consistency in the ACID sense. Cassandra supports atomicity and isolation at the row level but trades transactional isolation and atomicity for high availability and fast write performance. Cassandra writes are durable.

Solution

Enterprise-grade applications can certainly be designed and used today using Cassandra. One of the recommended approaches is to design the data model in such a way that any update operation can be avoided using the INSERT statements. An example can be treating all updates as new inserts as new columns added against the same row. Cassandra has atomicity enabled at the row level, which means that inserting or updating columns in a row is treated as one write operation.

Also, Cassandra supports full row-level isolation, which means that writes to a row are isolated to the client performing those writes and not visible to any other user until they are complete. Similarly, writes in Cassandra are completely durable. This means that all writes to a replica node are recorded on disk before being acknowledged as successful. The commit log can be replayed on restart to receive any lost writes in case there is a crash or server failure before the memory tables are flushed to disks.

Cassandra 1.2 and above also have support for atomic batch operations. In this batch-statement context, atomic means that if even one of the batches succeeds, all of them will. Cassandra internally uses a batchlog system table to achieve such atomicity. Cassandra first writes the serialized batch to this batchlog system table that consumes the serialized batch as BLOB data; when the rows in the batch have been successfully written and persisted (or hinted), the batchlog data is removed.

One of the recent solutions to some parts of this problem is the introduction of the lightweight transaction as part of Cassandra 2.0. This transaction support is similar to RDBMS support for certain use cases, which require **linearizable** consistency or in ACID terms, a **serial** isolation level. A serial consistency level allows us to read the current (and possibly uncommitted) state of data without proposing a new addition or update. If a serial read finds an uncommitted transaction in progress, it will commit it as part of the read.

An example of such a scenario might be assigning a defective ticket to a user, so that the ticket gets assigned only once. Cassandra uses a Paxos consensus-protocol-based implementation that can provide distributed transaction in a masterless environment.
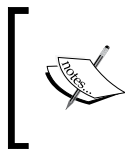
> Paxos details would require a separate book, but the following is a link to a simple explanation of this protocol: `http://www.cs.utexas.edu/users/lorenzo/corsi/cs380d/past/03F/notes/paxos-simple.pdf`.

The lightweight transaction support in Cassandra is similar to the "compare and set" operation and involves the new `IF` clause added to the `INSERT` and `UPDATE` commands in CQL. This can be illustrated in the following example code:

```
INSERT INTO tickets_assigned (ticketID, userID)
VALUES ('1001', 'smith@cassandra.org')
IF NOT EXISTS;
```

DML modifications via `UPDATE` can also use the new `IF` clause by comparing one or more columns to various values:

```
UPDATE tickets_assigned
SET    userID= 'john@cassandra.org'
IF     userID= 'smith@cassandra.org';
```

Caution

In fact, Cassandra does not support transactions in the sense of bundling multiple row updates into one all-or-nothing operation. It will also not roll back when a write succeeds on one replica, but fails on other replicas. It is also possible to have a write operation report a failure to the client, but still actually persist the write to a replica. Also note that Cassandra internally uses timestamps to determine the most recent update to a column. The timestamp is provided by the client application. The latest timestamp always wins when requesting data, so if multiple client sessions update the same columns in a row concurrently, the most recent update is the one that will eventually persist.

Cassandra's support for batch statements has a performance overhead for atomicity. This penalty can be avoided using the `UNLOGGED` option in batch statements. However, it should be noted that no other transactional enforcement is done at the batch level. For example, there is no batch isolation, and other clients will be able to read the first updated rows from the batch while the updating of other rows is in progress. However, transactional row updates within a single row are isolated; a partial row update cannot be read.

Please also note that the usage of lightweight transactions involves quorum-based operations and updates will incur a performance hit with degradation by one-third of the normal operation.

Pattern name – Materialized view

A materialized view is usually a data-store object that contains the result of a database query. Let us see how this pattern is useful as well as important for Cassandra.

Problem/Intent

How can we enable advanced queries in Cassandra using materialized views?

Context/Applicability

The materialized view pattern allows queries to be served faster in RDBMS, since the query results are already cached as a snapshot representation. Cassandra has a simple query-first data model, where queries are constrained around row-key lookups only or rely on secondary indexes. A materialized view-like implementation would allow end users to query Cassandra in more advanced ways than just relying on row-key and secondary index-based queries.

Forces/Motivations

Cassandra's highly scalable architecture for supporting high reads and writes and huge data volumes had to sacrifice support for aggregation, or join-like queries. So, how do we solve the problem of queries that are difficult to resolve using just row key or secondary indexes?

Solution

The solution lies in designing query-driven tables to provide index-like capabilities using denormalization as the key concept. This means that the required query results are created as separate tables in Cassandra and usually contain denormalized data that needs to be updated when the underlying data in the main tables changes.

Please note that materialized views are *not* supported out-of-the-box in Cassandra as in, say, RDBMS, where we can use `create materialized view ….. as [select query]`. Instead, we will be creating new tables in Cassandra that will contain the query results to be served by this materialized view.

Also note that challenges with `JOIN` queries in Cassandra can also be resolved using this pattern. So, a join query between two or more tables in Cassandra can be represented as a materialize view table to let us overcome the absence of the `JOIN` queries in Cassandra.

There are two ways of implementing materialized views in Cassandra:

- **Application-tier-driven materialized view**: Here, the application layer will be writing to the main table(s) as well as the materialized view table in Cassandra. This can provide both real-time read/write capabilities:

  - A Python or Java middleware application can be an example of such an implementation, where the business layer is also calculating the query results and then updating the table that represents the materialized view.

  - Another example can be Apache Storm being used as an aggregation processing layer. Please refer to *Chapter 5*, *Search and Analytics Applied Use Case Patterns*, for more details regarding this example.

- **Analytics-driven materialized view**: Here, an analytical solution or aggregation framework will be reading data from the main table(s) and then updating the materialized view table in Cassandra. This can be useful in scenarios where the aggregation or complex queries are time consuming and the only way to run them is through batch frameworks or fast-processing engines:

  - An example can be Hadoop being used as a batch-processing engine for pulling data from the main table(s) in Cassandra, running the query, and finally, updating the results in the materialized view table

  - An external aggregation engine such as Apache Storm can also be used for computing intensive query-processing operations in the same way as the Hadoop example does

Interestingly, Cassandra can also be used along with a search engine, such as Solr, and this search capability can be a potential way of implementing materialized views in the search engine itself.

Caution

The materialized view pattern uses denormalization as the underlying concept, and denormalization always comes with data-consistency problems with various copies of the same data. Given that Cassandra has limited support for transactions, materialized views should be used carefully, ensuring that there are sufficient mechanisms for data consistency between the main table(s) and the materialized view.

Pattern name – Composite key

The Composite key or Compound key is a well-known RDBMS design pattern. Let us see if it makes sense to use this in Cassandra.

Problem/Intent

How can we build advanced data models in Cassandra with multiple keys?

Context/Applicability

Composite keys allow two or more keys to represent and uniquely identify an entity occurrence.

> *In database design, a compound key is a key that consists of two or more attributes that uniquely identify an entity occurrence. Each attribute that makes up the compound key is a simple key in its own right.*
>
> *This is often confused with a composite key whereby even though this is also a key that consists of two or more attributes that uniquely identify an entity occurrence, at least one attribute that makes up the composite key is not a simple key in its own right.*
>
> *-Wikipedia*

Forces/Motivations

Cassandra initially only had support for a single primary key, which allowed simple data models to be designed.

Some ways of handling complex data structures traditionally involved using super column family or aggregated keys while designing Cassandra's data model.

Super column family is a special type of data table structure supported in Cassandra that allows a column to be complex and contain other columns within it, thus providing a deeper nested structure represented as follows:

```
Map<RowKey, SortedMap<SuperColumnKey, SortedMap<ColumnKey,
ColumnValue>>>
```

Aggregated keys involved creating concatenated keys that are composed of two or more primary keys using some delimiter.

Super column family has an intrinsic problem involving memory loading of the entire row in cache; hence, it is not recommended that you use super column families in Cassandra anymore. Similarly, aggregated keys are difficult to manage from a coding perspective and usually need expert data-modeling skills due to its complexities.

Solution

Cassandra added support for composite primary as part of CQL support, and there have been some advanced capabilities added to it as part of CQL1.2 and its subsequent versions.

This composite key support allows developers to apply traditional primary key concepts to Cassandra data modeling and still use Cassandra's strengths internally for high scalability and fast querying.

The composite key support in CQL is illustrated as follows:
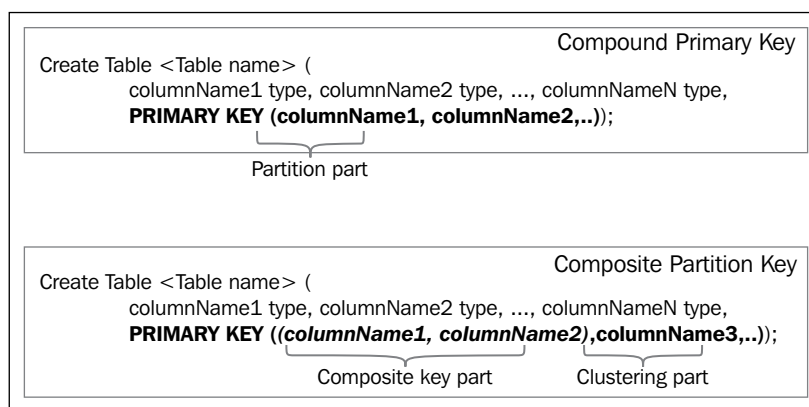
Cassandra uses the first column name in the primary key definition as the partition key, which is the same as the row key, to the underlying storage engine. The data for each partition key gets clustered by the remaining columns of the primary key definition. Clustering means that the storage engine creates an index and always keeps the data clustered in that index.

Cassandra 1.2 documentation states that CQL3 transposes data partitions (sometimes called "wide rows") into familiar, row-based result sets, which dramatically simplifies data modeling.

Also, according to the documents, legacy tables from Cassandra 1.1 are supported in 1.2 and Cassandra 1.1 style tables can still be created using the `COMPACT STORAGE` attribute in CQL 3. However, the `COMPACT STORAGE` directive prevents the addition of more than one column that is not part of the primary key.

Let us illustrate the use of the compound primary key using the following example:

```
CREATE TABLE employee (
  empID int,
  deptID int,
  first_name varchar,
  last_name varchar,
  PRIMARY KEY (empID, deptID));
```

The compound primary key is made up of the `empID` and `deptID` columns in this example. The `empID` attribute acts as a partition key for distributing data in the table among the various nodes that comprise the cluster. The remaining component of the primary key, `deptID`, acts as a clustering mechanism and ensures that the data is stored in ascending order on the disk. On a physical node, when rows for a partition key are stored in order of clustering column, the retrieval of rows is very efficient. For example, because `empID` in the employee table is the partition key, all `deptID` attributes for an employee are clustered in the order of the `deptID` column.

It is also worthwhile to note that we can perform sorting or ordering operations on the second key of our compound key. So, we can use this pattern to add the `ORDER BY` support in Cassandra:

```
CREATE TABLE employee (
  empID int,
  deptID int,
  first_name varchar,
  last_name varchar,
  PRIMARY KEY (empID, deptID))
  WITH CLUSTERING ORDER BY (deptID);
```

In this example, `WITH CLUSTERING ORDER` allows Cassandra to store the data in descending order of `deptID`.

Additional interesting patterns

The following are a few other interesting patterns worth mentioning, which are useful for simplifying Cassandra usage:

Pattern name	Context/problem	Solution/usage
Valueless columns	In a Cassandra data model, the column name is stored against a column value. This ability allows schemaless data modeling in Cassandra, where each row in a Cassandra table can have its own set of columns.	Valueless column usage means having columns with just names and *no* values. This can sometimes be useful for inverted index tables where column names would contain the required information. An easy way to implement this is using `byte[0]` as the value against a column. This pattern is useful for simplifying data modeling.
Collection fields	Collections are usually useful in the database design for managing relationships between entities. Cassandra does not have any join support. However, storing collections such as Set, List, or Map in Cassandra has always been possible using its flexible, column-based representations. However, this implementation has always been complicated as these structures need to be handled at the application end.	Cassandra CQL1.2 has added support for handling collection objects, such as Map, Set, and List, and should be used instead of implementing custom solutions. Internally, Cassandra uses internal column structures for efficient storage of these collections; so, it is a scalable solution. Please note that there are still some precautions to be observed for these new datatypes: A collection has to be retrieved as a whole. This means that collections are not meant to be excessively large or a replacement for proper modeling into tables. Collections are typed, but cannot currently be nested. This means that it is possible to have `list<text>` or `list<int>`, but not `list<list<int>>`. There is no support for secondary indexes in collections.

Pattern name	Context/problem	Solution/usage
Sorting	Ordering or sorting is one of the important functions required by any data store and it has not been a trivial task to provide these capabilities in Cassandra. Unlike with traditional RDBMS applications, we need to use design time sorting rather than query time ordering. This means that the ordering needs to be designed as part of the data model.	There are multiple ways of enabling the sorting of results in Cassandra. One of the easiest ones is using an ordered partition, which allows row/primary keys to be stored in a sorted order. However, ordered partitions are not recommended for use as these can easily skew data. Skewed data is dangerous, because it can result in only a few Cassandra nodes being used while the rest of the nodes are not being used at all. Another approach is using column-name natural sorting, which can be controlled using comparators to fine-tune the sorting. This approach involves using wide rows and the storing of the values that need to be sorted as column names. CQL3 provides sorting support using compound key and, internally, would be using the same mechanism as that in the column-name sorting approach.
Pagination	Pagination for web applications is the process of dividing the results to be shown on the web page over multiple pages. In Cassandra, the traditional approach has been manual pagination, usually using column slicing. However, this approach has always been nontrivial and requiring application-based page limits, state management, and complex coding.	CQL3 has inbuilt support for pagination in the form of cursors, available as part of the SELECT query. This provides an easy and simple approach to implementing pagination, as shown in the following Java example: ```Statement stmt = new SimpleStatement("SELECT * FROM employees"); stmt.setFetchSize(100); ResultSet rs = session.execute(stmt);``` This approach eliminates the need to use the token function to page through the results available in CQL2.0, illustrated as follows: ```SELECT * FROM employees WHERE token(k) > token(100);```

Pattern name	Context/problem	Solution/usage
Temporal data	Some applications require data that is temporal in nature, which means that the data has some life associated with it. **TTL** (**time to live**) is the term used for finding the life duration of a record in Cassandra.	Cassandra allows setting TTL at the column level and, after this time to live is surpassed, the column is marked for deletion as a tombstone record. Please note that the precaution with deletes in Cassandra holds true here too, as the tombstone records might result in performance degradation. Greater degradation happens where there are many tombstone records to be processed during either query or compaction process.
Triggers	Cassandra traditionally did not have support for DDL- or DML-triggered events handling, as in RDBMS or some NoSQL stores such as HBase, which have co-processor support.	Cassandra 2.0 and above now have experimental support for triggers that allow Java-based triggers to be coded and used on all changes to a Cassandra table using `RowMutations`. Please note that such a prototype implementation is subject to change in future Cassandra releases, but is certainly an important capability being added to Cassandra.
Unique records	Cassandra did not have support for server-side, unique, or distinct operations.	Cassandra 2.0 now has support for the `DISTINCT` clause, but it is available only for partition keys. Please remember that the partition key, here, refers to the primary key in a simple key or the first key in a composite key.

Anti-pattern name – Messaging queue

Messaging queues are commonly used to build scalable applications, because these provide interprocess communication by enabling asynchronous communication protocol. Messaging usually involves producers generating messages, while consumers listen to a queue from which the messages can be read and processed. The queue that stores messages has to be fast, efficient, and reliable. So, let us see whether or not Cassandra can be used as a scalable storage backbone required for building a message queue implementation.

Problem/Intent

Can Cassandra be used as a messaging queue?

Context/Applicability

Cassandra offers high ingest and read capabilities; hence, it can easily be considered as a candidate for messaging-queue solutions.

Liability/Issue

Cassandra should not be used as a durable messaging queue due to a major problem with the way that it handles deletes.

Cassandra uses a log-structured storage engine, so, deletes do not remove the rows and columns immediately. Instead, Cassandra writes a special marker called a tombstone, which indicates that a row, column, or range of columns was deleted.

In case of a message queue, the typical operation includes queuing and dequeuing, which translate into inserts and subsequent deletes. These deletes would result in multiple tombstone markers. So, a new message dequeue would need us to go through all the tombstones and then finally the newly queued message.

Patterns and anti-patterns – Cassandra infrastructure/deployment problems

The following is a list of recommendations from a Cassandra infrastructure and deployment perspective:

- Cassandra on SAN is not recommended.
- Commit log and data directories should ideally be on separate hard drives in case of SATA or SAS drives. This is due to the fact that commit log usage is sequential, while data operations are sequential in addition to random updates in some cases, during compactions or flush operations.
- Batch statements should be used carefully and, ideally, the size of the batch should be carefully tested and set. A large batch will result in memory overheads and bottlenecks, while a small batch can result in network overheads; hence, benchmarking and testing is a reliable way of figuring and defining the batch size based on the schema used in the batch statements.
- Placing the load balancer in front of the Cassandra cluster is not recommended.

- Row cache, especially for large rows, should be used only in rare cases and after proper testing.

- Ordered partitioning should be avoided as this can result in unmanageable, skewed data and unbalanced clusters.

- In fact, `OrderPreservingPartitioner` and `CollatingOrderPreservingPartitioner` have been deprecated in recent Cassandra releases.

- Super columns should not be used and have been deprecated due to challenges in memory management.

- Read repairs usually happen across all data centers in a multidatacenter deployment. So, DC local read repair should be configured for efficiency.

- In case single nodes are slow, partition-aware clients such as Astyanax can deal with this situation.

- Compression might become a bottleneck for fast reads, so it should be used carefully. This is because of some code issue in Cassandra that disables the optimization of some fast paths when using compression. Hopefully, these bugs will be resolved in the near future. It would also be worthwhile to note that LZ4 compression can now be used with Cassandra, which is around 50 percent faster than default snappy compression.

- SSD can also be beneficial for Cassandra. Cassandra is one of the few data stores optimized for SSD. Cassandra can use SSD's high IOPS capabilities for faster random reads and also minimize the undesirable effects of write amplification, which is often associated with SSD.

Summary

In this chapter, we covered a lot of patterns, usage precautions, and anti-patterns for using Cassandra successfully to solve various business challenges. Despite the various usages and patterns that we have gone though, this journey is not complete. The list of topics to discuss will keep evolving along with Cassandra's growth and its emergence as the leader in the Big Data store arena.

Index

L

linearizable 58

M

Master Data Hub (MDH) 13
Master-Master replication 15
Master-Slave mode 15
Materialized View pattern
  about 18, 60
  caution 61
  context/applicability 60
  forces/motivations 60
  problem/intent 60
  solution 60
materialized views
  about 61
  analytics-driven materialized view 61
Merkle tree data structure 9, 10
Messaging queue anti-pattern
  about 67
  context/applicability 68
  liability/issue 68
  problem/intent 68

N

Needle in a Haystack pattern
  about 17, 46
  context/applicability 47
  forces/motivations 47
  problem/intent 46
  search engine solution, implementing 48
  solution 47, 48
Netflix 24

O

Object/Entity Store pattern
  about 18, 55
  caution 57
  context/applicability 56
  forces/motivations 56
  problem/intent 55
  solution 56
Operational Data Store (ODS) 13
ORM tool 55

P

P2P (peer-to-peer) 15
Pagination pattern 66
Playorm 24
property file snitch 35

R

rack inferring snitch 35
RDBMS or relational databases 14

S

Savvis 24
search engine solution, implementing
  Cassandra along with SOLR integration 48
  DataStax Enterprise 49
  Kundera 49
serial isolation level 58
Service Oriented Architecture (SOA) 13
Sharding 9
simple snitch 35
single point of failure (SPOF) 8
snitch implementations
  dynamic snitch 35
  EC2 multi region snitch 35
  EC2 snitch 35
  gossiping property file snitch 35
  property file snitch 35
  rack inferring snitch 35
  simple snitch 35
Sorted String Table (SSTable) 7
Sorting pattern 66
Spring Data 24
Storm-Cassandra 45
Streaming/CEP Analytics pattern
  about 17, 43
  context/applicability 44
  forces/motivations 44
  problem/intent 43
  solution 44, 45

T

TDS/OLTP 13
Temporal data pattern 67
Terremark 24

[PACKT] PUBLISHING · open source*
community experience distilled

Thank you for buying
Cassandra Design Patterns

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.
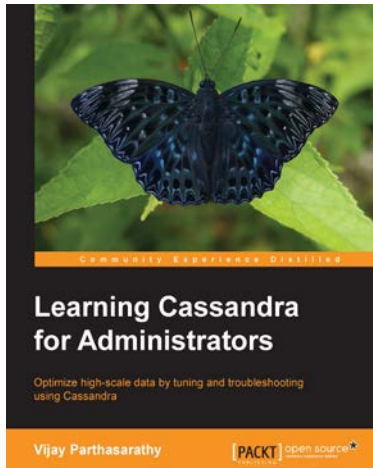
Learning Cassandra for Administrators

ISBN: 978-1-78216-817-1   Paperback: 120 pages

Optimize high-scale data by tuning and troubleshooting using Cassandra

1. Install and set up a multi datacenter Cassandra

2. Troubleshoot and tune Cassandra

3. Covers CAP tradeoffs, physical/hardware limitations, and helps you understand the magic

4. Tune your kernel, JVM, to maximize the performance
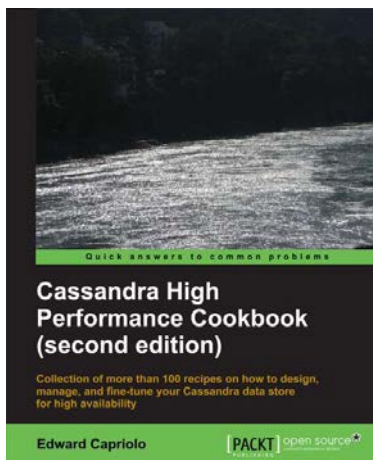
## Cassandra High Performance Cookbook: Second Edition

ISBN: 978-1-78216-180-6   Paperback: 350 pages

Collection of more than 100 recipes on how to design, manage, and fine-tune your Cassandra data store for high availability

1. Understand the Column Family data model and other Big Data schema modeling techniques from practical real-world examples

2. Write applications to store and access data in Cassandra using both the RPC and Cassandra Query Language interfaces

3. Deploy multi-node, multi-data center Cassandra clusters for high availability

Please check **www.PacktPub.com** for information on our titles
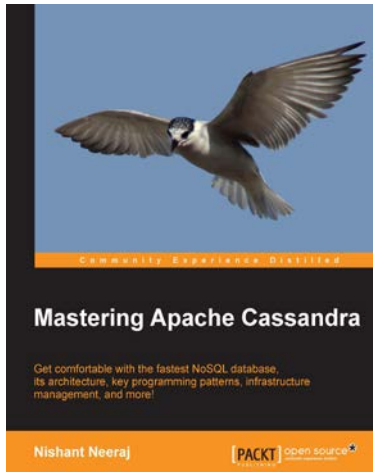
Mastering Apache Cassandra

ISBN: 978-1-78216-268-1   Paperback: 340 pages

Get comfortable with the fastest NoSQL database, its architecture, key programming patterns, infrastructure management, and more!

1. Complete coverage of all aspects of Cassandra

2. Discusses prominent patterns, pros and cons, and use cases

3. Contains briefs on integration with other software
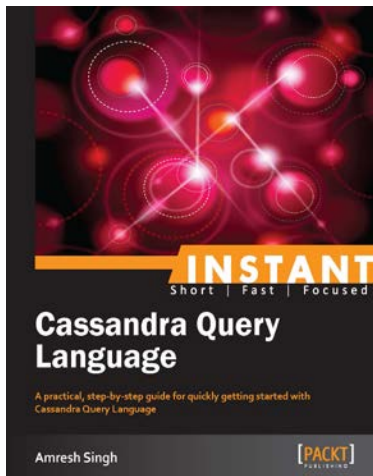
Instant Cassandra Query Language

ISBN: 978-1-78328-271-5   Paperback: 54 pages

A practical, step-by-step guide for quickly getting started with Cassandra Query Language

1. Learn something new in an Instant! A short, fast, focused guide delivering immediate results

2. Covers the most frequently used constructs using practical examples

3. Dive deeper into CQL, TTL, batch operations, and more

4. Learn how to shed Thrift and adopt a CQL-based binary protocol

Please check **www.PacktPub.com** for information on our titles