Introduction To Cassandra

1 . NoSQL

NoSQL is a database technology designed to support the requirements of cloud applications and architected to overcome the scale, performance, data model, and data distribution limitations of relational databases (RDBMS's).

1.1 NoSQL Explained

A NoSQL (Not-only-SQL) database is one that has been designed to store, distribute and access data using methods that differ from relational databases (RDBMS's). NoSQL technology was originally created and used by Internet leaders such as Facebook, Google, Amazon, and others who required database management systems that could write and read data anywhere in the world, while scaling and delivering performance across massive data sets and millions of users.

1.2 How NoSQL Differs from Relational Databases

NoSQL and RDBMS's are designed to support different application requirements and typically they co-exist in most enterprises. The key decision points on when to use which include the following:

Today, almost every company and organization has to deliver cloud applications that personalize their customer's experience with their business, with NoSQL being the database technology of choice for powering such systems.

Use an RDBMS when you need / have	Use NoSQL when you need / have
Centralized applications (e.g. ERP)	Decentralized applications (e.g. Web, mobile and IOT)
Moderate to high availability	Continuous availability; no downtime
Moderate velocity data	High velocity data (devices, sensors, etc.)
Data coming in from one/few locations	Data coming in from many locations
Primarily structured data	Structured, with semi/unstructured
Complex/nested transactions	Simple transactions
Primary concern is scaling reads	Concern is to scale both writes and reads

2 . Cassandra

2.1 What is Apache Cassandra?

Apache Cassandra, a top level Apache project born at Facebook and built on Amazon's Dynamo and Google's BigTable, is a distributed database for managing large amounts of structured data across many commodity servers, while providing highly available service and no single point of failure. Cassandra offers capabilities that relational databases and other NoSQL databases simply cannot match such as: continuous availability, linear scale performance, operational simplicity and easy data distribution across multiple data centers and cloud availability zones.

Ring Architecture:

Cassandra's architecture is responsible for its ability to scale, perform, and offer continuous uptime. Rather than using a legacy master-slave or a manual and difficult-to-maintain sharded architecture, Cassandra has a masterless "ring" design that is elegant, easy to setup, and easy to maintain.

In Cassandra, all nodes play an identical role; there is no concept of a master node, with all nodes communicating with each other equally. Cassandra's built-for-scale architecture

means that it is capable of handling large amounts of data and thousands of concurrent users or operations per second—even across multiple data centers—as easily as it can manage much smaller amounts of data and user traffic. Cassandra's architecture also means that, unlike other master-slave or sharded systems, it has no single point of failure and therefore is capable of offering true continuous availability and uptime — simply add new nodes to an existing cluster without having to take it down.

Many companies have successfully deployed and benefited from Apache Cassandra including some large companies such as: Apple, Comcast, Instagram, Spotify, eBay, Rackspace, Netflix, and many more. The larger production environments have PB's of data in clusters of over 75,000 nodes. Cassandra is available under the Apache 2.0 license.

2.2 Architecture in brief

Cassandra is designed to handle big data workloads across multiple nodes with no single point of failure. Its architecture is based on the understanding that system and hardware failures can and do occur. Cassandra addresses the problem of failures by employing a peer-to-peer distributed system across homogeneous nodes where data is distributed among all nodes in the cluster. Each node exchanges information across the cluster every second. A sequentially written commit log on each node captures write activity to ensure data durability. Data is then indexed and written to an in-memory structure, called a memtable, which resembles a write-back cache. Once the memory structure is full, the data is written to disk in an SSTable data file. All writes are automatically partitioned and replicated throughout the cluster. Using a process called compaction Cassandra periodically consolidates SSTables, discarding obsolete data and tombstone (an indicator that data was deleted).

Cassandra is a row-oriented database. Cassandra's architecture allows any authorized user to connect to any node in any data center and access data using the CQL language. For ease of use, CQL uses a similar syntax to SQL. From the CQL perspective the database consists of tables. Typically, a cluster has one keyspace per application. Developers can access CQL through cqlsh as well as via drivers for application languages.

Client read or write requests can be sent to any node in the cluster. When a client connects to a node with a request, that node serves as the coordinator for that particular client operation. The coordinator acts as a proxy between the client application and the

nodes that own the data being requested. The coordinator determines which nodes in the ring should get the request based on how the cluster is configured.

2.3 Key structures

Node

Where we store our data. It is the basic infrastructure component of Cassandra.

Data center

A collection of related nodes. A data center can be a physical data center or virtual data center. Different workloads should use separate data centers, either physical or virtual. Replication is set by data center. Using separate data centers prevents Cassandra transactions from being impacted by other workloads and keeps requests close to each other for lower latency. Depending on the replication factor, data can be written to multiple data centers. However, data centers should never span physical locations.

Cluster

A cluster contains one or more data centers. It can span physical locations.

Commit log

All data is written first to the commit log for durability. After all its data has been flushed to SSTables, it can be archived, deleted, or recycled.

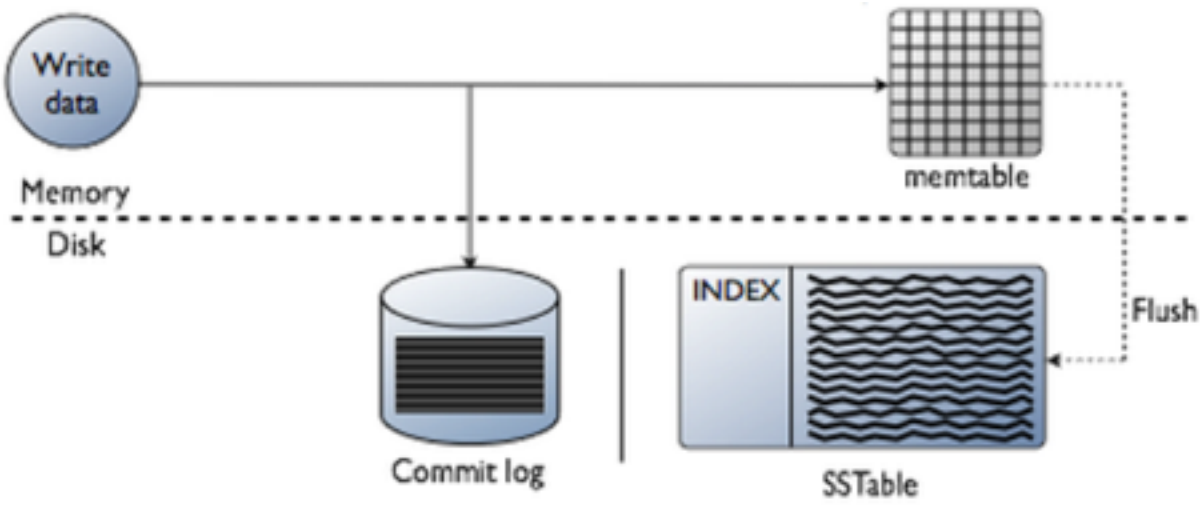
Table

A collection of ordered columns fetched by row. A row consists of columns and have a primary key. The first part of the key is a column name.

SSTable

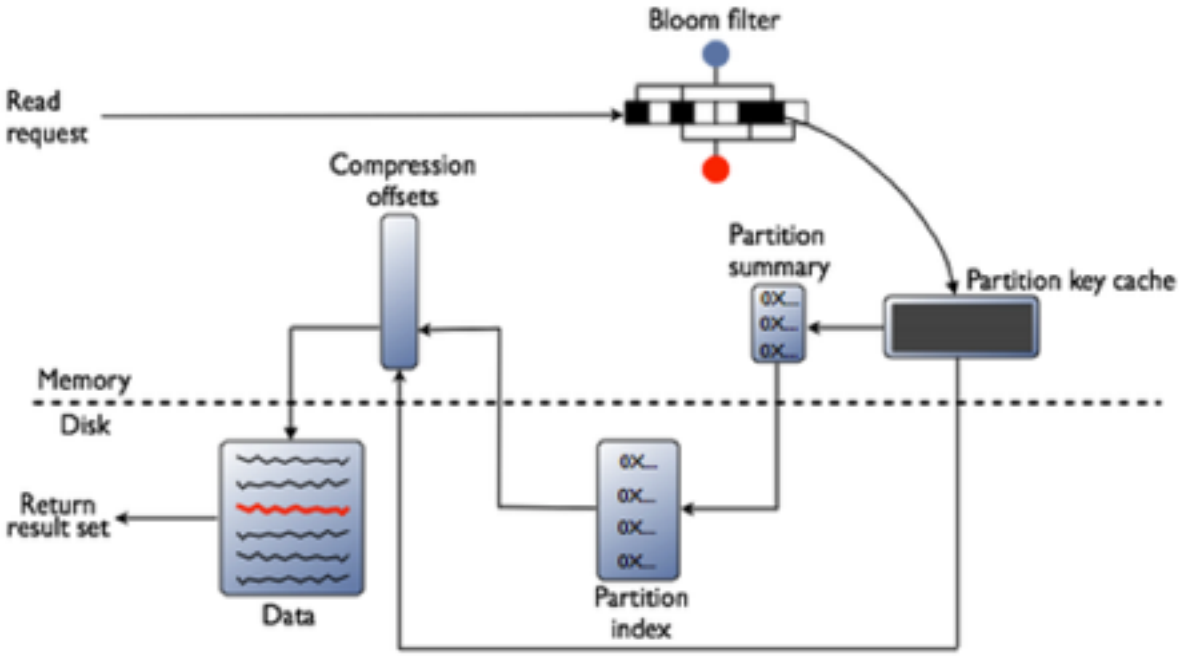
A sorted string table (SSTable) is an immutable data file to which Cassandra writes memtables periodically. SSTables are append only and stored on disk sequentially and maintained for each Cassandra table.

2.4 Writing and Reading Data



Cassandra is well known for its impressive performance in both reading and writing data. Data is written to Cassandra in a way that provides both full data durability and high performance. Data written to a Cassandra node is first recorded in an on-disk commit log and then written to a memory-based structure called a memtable. When a memtable's size exceeds a configurable threshold, the data is written to an immutable file on disk called an SSTable. Buffering writes in memory in this way allows writes always to be a fully sequential operation, with many megabytes of disk I/O happening at the same time, rather than one at a time over a long period. This architecture gives Cassandra its legendary write performance.

2.5 Reading data from Cassandra



Reading data from Cassandra involves a number of processes that can include various memory caches and other mechanisms designed to produce fast read response times.

For a read request, Cassandra consults an in-memory data structure called a Bloom filter that checks the probability of an SSTable having the needed data. The Bloom filter can tell very quickly whether the file probably has the needed data, or certainly does not have it. If answer is a tenative yes, Cassandra consults another layer of in-memory caches, then fetches the compressed data on disk. If the answer is no, Cassandra doesn't trouble with reading that SSTable at all, and moves on to the next.

2.6 Data Management Concepts

This section provides a brief introduction to Cassandra's data model, along with the data structures and query language Cassandra uses to manage data.

2.7 Data Model Overview

Cassandra is a wide-row-store database that uses a highly denormalized model designed to capture and query data performantly. Although Cassandra has objects that resemble a relational

database (e.g., tables, primary keys, indexes, etc.), Cassandra data modeling techniques necessarily depart from the relational tradition. For example, the legacy entity-relationship-attribute data modeling paradigm is not appropriate in Cassandra the way it is with a relational database.

Success with Cassandra almost always comes down to getting two things right: the data model and the selected hardware—especially the storage subsystem. The topic of data modeling is of prime importance.

Unlike a relational database that penalizes the use of many columns in a table, Cassandra is highly performant with tables that have thousands or even tens of thousands of columns. Cassandra provides helpful data modeling abstractions to make this paradigm approachable for the developer.

2.8 Cassandra Data Objects

The basic objects you will use in Cassandra include:

Keyspace – a container for data tables and indexes; analogous to a database in many relational databases. It is also the level at which replication is defined.

Table – somewhat like a relational table, but capable of holding vastly large volumes of data. A table is also able to provide very fast row inserts and column level reads.

Primary key – used to identity a row uniquely in a table and also distribute a table's rows across multiple nodes in a cluster.

Index – similar to a relational index in that it speeds some read operations; also different from relational indices in important ways.

2.9 Cassandra Query Language

Early versions of Cassandra exclusively used the programmatic Thrift interface to create database objects and manipulate data. While Thrift is still supported and maintained in Cassandra, the Cassandra Query Language (CQL) has become the primary API used for interacting with a Cassandra cluster today. This represents a substantial improvement in Cassandra's usability.

CQL resembles the standard SQL used by all relational databases. Because of that similarity, the learning curve for those coming from the relational world is reduced. DDL (CREATE, ALTER, DROP), DML (INSERT, UPDATE, DELETE, TRUNCATE), and query (SELECT) operations are all supported.

CQL datatypes also reflect RDBMS syntax with numerical (int, bigint, decimal, etc.), character (ascii, varchar, etc.), date (timestamp, etc.), unstructured (blob, etc.), and specialized datatypes (set, list, map, etc.) being supported.

Various CQL command line utilities like cqlsh and graphical tools like DataStax DevCenter can be used to interact with a Cassandra cluster, and client drivers for Cassandra (Java, C#, etc.) also support CQL for developing applications.

2.10 Transaction Management

While Cassandra does not support ACID transactions like most legacy relational databases, it does offer the “AID” portion of ACID. Writes to Cassandra are atomic, isolated, and durable. The “C” of ACID—consistency—does not apply to Cassandra, as there is no concept of referential integrity or foreign keys.

Cassandra offers tunable data consistency across a database cluster. This means you can decide whether you want strong or eventual consistency for a particular transaction. You might want a particular request to complete if just one node responds, or you might want to wait until all nodes respond. Tunable data consistency is supported across single or multiple data centers, and you have a number of different consistency options from which to choose.

Consistency is configurable on a per-query basis, meaning you can decide how strong or eventual consistency should be per SELECT, INSERT, UPDATE, and DELETE operation. For example, if you need a particular transaction to be available on all nodes throughout the world, you can specify that all nodes must respond before a transaction is marked complete. On the other hand, a less critical piece of data (e.g., a social media update) may only need to be propagated eventually, so in that case, the consistency requirement can be greatly relaxed.

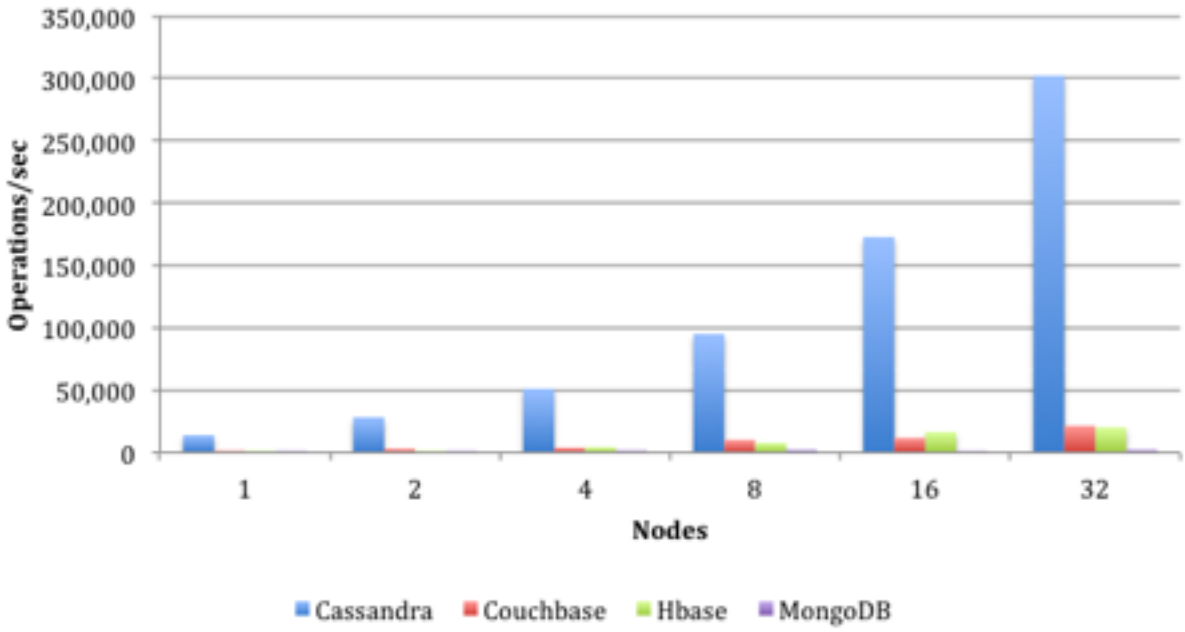
Cassandra also supplies lightweight transactions, or a compare-and-set mechanism. Using and extending the Paxos consensus protocol (which allows a distributed system to agree on proposed data modifications with a quorum-based algorithm, and without the need for any one "master" database or two-phase commit), Cassandra offers a way to ensure a transaction isolation level similar to the serializable level offered by relational database.

2.10 Migrating Data to Cassandra

Moving data from an RDBMS or other database to Cassandra is fairly easy depending on the state of the existing data. The following options currently exist for migrating data to Cassandra:

COPY command – The cqlsh utility provides a copy command that is able to load data from an operating system file into a Cassandra table. Note that this is not recommended for very large files.

SSTable loader – this utility is designed for more quickly loading a Cassandra table with a file



that is delimited in some way (e.g. comma, tab, etc).

Sqoop – Sqoop is a utility used in Hadoop to load data from relational databases into a Hadoop cluster. DataStax supports pipelining data from a relational databases table directly into a Cassandra table in its production certified Cassandra platform (DataStax Enterprise).

ETL tools – there are a variety of ETL tools that support Cassandra as both a source and target data platform. Many of these tools not only extract and load data but also provide transformation routines that can manipulate the incoming data in many ways. A number of these tools are also free to use (e.g. Pentaho, Jaspersoft, Talend).

3 . Comparison with other NoSQL's

NoSQL databases are challenging relational technologies by delivering the flexibility required of modern applications. But, which NoSQL database is best architected to handle performance demands of today's workloads?

Benchmarks recently run by End Point an independent database firm, stress-tested Apache Cassandra, HBase, MongoDB, and Couchbase on operations typical to real-world applications. Results showed that Cassandra outperformed its NoSQL counterparts.

In fact, for mixed operational and analytic workloads typical to modern Web, Mobile and IOT applications, Cassandra performed six times faster than HBase and 195 times faster than MongoDB.

Operations Per Second Per Node Cluster

4 . Who uses Cassandra

5 . Use Cases :

5.1 Bazaarvoice

Bazaar voice chooses Cassandra over MySQL,HBase and MongoDB to power content analytics platform—Shawn Smith Software Engineer at Bazaarvoice

"We started out by using MySQL in the classic master/slave horizontal scale out way. We found it just impossible to scale and grow write capacity with MySQL."

Bazaarvoice collects user generated content that deal with reviews, questions/answers, stories and other such things for various retailers and brands, and then we analyze and serve that information back up to our clients. We've been doing this for about seven years now, and our customers include very well known companies such as those shown on our website.

Horizontal Scaling

We started out by using MySQL in the classic master/slave horizontal scale out way. We found it just impossible to scale and grow write capacity with MySQL. So we started looking for something that was cloud friendly, which was very important to us. We needed something where, if any one machine goes down, it's not a big deal, meaning our systems aren't affected and that it can recover without human intervention.

Next, we needed a database that allowed for easy capacity expansion (especially write capacity) by simply adding new machines online. Having multiple data center support was also a very big deal, especially where we can write to multiple data centers at the same time. We didn't want master/slave data centers but peer to peer data centers.

These things were key to why we chose Cassandra.

NoSQL Landscape

We experimented with MongoDB and we do use Hadoop for our analytics, but when we did our architecture comparisons with HBase and Mongo, and wrote a number of development prototypes, and we became convinced that Cassandra was the right way to go.

The multi-data center support, the masterless architecture, ease of administration, and the no single point of failure and constant availability in the cloud were the things that were key for us.

Cataloging AcrossAZ

Cassandra is what we use for our primary datastore, but we're big enough where other databases are also used for other things.

We use Cassandra for two main classes of data. One use case is that we take lots of product feeds from our customers and we maintain a big master catalog of all our customer's products, names, categories and brands. So all of our customer metadata is maintained in Cassandra.

The second use case for Cassandra is one where we store all of the user generated content from all our customer's sites. Whenever users submit something on a customer's website, that's fed into Cassandra, with feeds coming into different data centers. That's all analyzed and then returned back to our customers.

We do have a data center in Dallas, but nearly all of our new development is being carried out on Amazon, spanning multiple cloud availability zones.

Get The Model Right

The biggest thing to get your head around is the data model differences. You need to think about how you're going to read the data. We pretty much do all of our writing in Cassandra and then replicate that data over to ElasticSearch for various search and read operations. So for us, getting the schema right in Cassandra was very important.

5.2 Outbrain

Outbrain touches over 80% of all US online users with help from Cassandra

Outbrain helps people discover the most interesting, relevant and trusted content out there. We are the leading company for content discovery. Folks like CNN, Fox News, and lots of other premium publishers and companies trust us on thier sites. The company was founded in 2006 and we do somewhere around 90 billion recommendations on over 10 billion page views per month. Right now we reach ~86% of the online US population.

MULTI-DC

A number of years ago we were looking at what would be the best way to deploy our data across multiple data centers. A number of other databases we tried just wouldn't work, and were operationally immature. Our primary reason for using Cassandra over something like HBase is the ease with which we can span multiple data centers; it's very simple. We're talking very little if any operational overhead in doing what we do today with Cassandra.

By contrast, try doing something like that with MySQL and have network partition or other failure can be expensive in terms of engineering time to repair in many cases.

HURRICANE SANDY

During Hurricane Sandy, we lost an entire data center. Completely. Lost. It.

Our application fail-over resulted in us losing just a few moments of serving requests for a particular region of the country, but our data in Cassandra never went offline.

Then, when that data center came back online, we spent a lot of time manually rebuilding other databases that we use, shipping drives across country and what not. But for Cassandra, all we did was turn it back on in that data center and ran a single command, and it brought itself back up to speed with no other manual work.

SINCE 0.4

We've been using Cassandra for quite a while. We started way back with version 0.4. As an ops guy, I like Cassandra because, after you do your data modeling, you can turn Cassandra on and pretty much leave it alone except when you need to add new nodes online to increase capacity.

We've got a number of Cassandra clusters. We have a general use cluster, an 'on line' cluster that is directly in the path of how fast we serve up recommendations (which is critical), a few clusters tied to specific algorithms or processes, and then a number of other production and development clusters.

We like standing up different Cassandra clusters that are devoted to a specific application domain vs. having one big multi-tenant cluster that services many different apps. That seems to give us the best performance and makes finding problems easier, plus it's also simpler to manage from an automation perspective.

We also run these clusters across multiple data centers; we have three data centers now and more on the way.

SELECTING A DB

Top to bottom we use open source technologies on the front and back end. On the back end we employ a menagerie of technologies and data stores. Cassandra is one of the central pieces of our back end and serves as one of our primary data stores We also use Hadoop and Hive heavily and rely on Solr, MogileFS, MySQL, Storm, Kafka for various products and algorithms. We even have a few MongoDB small instances for various projects.

We look at how the data will be queried, its size, and how it needs to be distributed. We might use things like MySQL for historical reasons and MongoDB for smaller tasks, and then Cassandra for situations where data doesn't all fit into memory or where it spans multiple machines and possibly data centers.

Cassandra is something we have a pretty good handle on now and know what to expect from it. When we architect and design properly we trust it to not fold under pressure.

A NEW DATA MODEL

The primary thing is to do your data modeling in a way that fits with Cassandra. Don't treat it like MySQL, Mongo, or a pure KV store and you'll get the full benefits of it's

replication and caching models. Get that right and your management and maintenance isn't hard. Any real problem we've had with Cassandra has occurred because we tried to treat the Cassandra data model like some other data store and you can't do that.

5.3 Eventbrite

Eventbrite Moves to Cassandra for Customer Experience, Scalability and Availability

Eventbrite is an event-ticketing marketplace. We have two main focuses: One, providing tools and services to our organizers so that it's super easy for them to use our services by creating events and selling tickets. Two, making the marketplace super easy for attendees to access our services, find events that are interesting to them and attend those events.

Personalized Recommandations To Millions

Primarily we are using Cassandra for our consumer experience features, which is basically delivering recommendations and various discovery funnels out to our website and to our mobile apps and some parts of our API. The way we have scaled our architecture is we have MySQL as our transaction data store and we have decoupled services from that transaction data store to different parts of our infrastructure. One of the key parts is being able to actually calculate a large amount of data, upload it quickly, and serve it to our tens of millions of users. That's where we use Cassandra.Eventbriteiphone

Another place we use Cassandra is that we are building a spam and fraud specific data warehouse; this warehouse will be used to serve a large group of queries that have random access. These are the two places we are primarily using Cassandra.

Consistency Requirements

We were primarily doing everything out of MySQL before and, as we are growing rapidly, we needed to decouple our services based on the consistency requirements.

Cassandra provides us an easy way of providing a highly available store and not have to worry about things like sharding, multi-datacenter support and things like that. That's why Cassandra was very attractive to us, where we didn't have high consistency requirements.

Hacking within Minutes

I think one of the reasons why Cassandra has taken off and is doing so well is because of its awesome community; I have had the pleasure of interacting with a few committers. I have worked with a few people in the past, specifically where there was a lot of contribution done to Cassandra.It's an awesome group of really smart people. All the improvements that we are seeing at Cassandra are amazing; especially with DataStax, in terms of all the documentation that DataStax has written it is just incredible for anybody to get on and start hacking within minutes.

Lessons Learned

I think that we have been very careful about using data stores where their strengths are; I think that many problems occur when you have to go back and redo things. For example, if I was to build a solution where I need range queries out of Cassandra, that would be a bad thing because Cassandra's not built for that. We were wise in actually determining where Cassandra's strength lies and what exactly we need out of these features, and that's the only place where we are going to use Cassandra. We do the same thing with out other solutions as well, like HBase ,MySQL, and Redis.

We have a bunch of different kinds of data stores in our infrastructure but all of them are serving to their strengths. If I had to go back and do something different: Since we run Cassandra on the cloud, we do see some IO bottlenecks; in the future, we might actually think about using SSDs or our own data centers to dissolve some of those issues.

Out of the Box

The only advice I would give you is use it where the strengths are; I don't feel comfortable in saying that you should use Cassandra with high consistency requirements, if you have performance on the back of your mind. If performance is critical, I like to use Cassandra with low consistency requirements and not do things for which a data service is not made for. For example, doing range queries or running heavy map reduce jobs and stuff like that.

Cassandra is an amazing store. Some of the features that you get out of the box are pretty incredible: high availability, multi-datacenter support, etc. With the performance, you're able to do a ton of writes and a ton of reads.

6. Sample Application

We create a complete sample application so we can see how all the parts fit together.

6.1 Data Design

When you set out to build a new data-driven application that will use a relational database, you might start by modeling the domain as a set of properly normalized tables and use foreign keys to reference related data in other tables. Now that we have an understanding of how Cassandra stores data, let's create a little domain model that is easy to understand in the relational world, and then see how we might map it from a relational to a distributed hashtable model in Cassandra.

Relational modeling, in simple terms, means that you start from the conceptual domain and then represent the nouns in the domain in tables. You then assign primary keys and foreign keys to model relationships. When you have a many-to-many relationship, you create the join tables that represent just those keys. The join tables don't exist in the real world, and are a necessary side effect of the way relational models work. After you have all your tables laid out, you can start writing queries that pull together disparate data using the relationships defined by the keys. The queries in the relational world are very much secondary. It is assumed that you can always get the data you want as long as you have your tables modeled properly. Even if you have to use several complex subqueries or join statements, this is usually true.

By contrast, in Cassandra you don't start with the data model; you start with the query model.

For this example, let's use a domain that is easily understood and that everyone can relate to: a hotel that wants to allow guests to book a reservation.

Our conceptual domain includes hotels, guests that stay in the hotels, a collection of rooms for each hotel, and a record of the reservation, which is a certain guest in a certain room for a certain period of time (called the "stay"). Hotels typically also maintain a collection of "points of interest," which are parks, museums, shopping galleries, monuments, or other places near the hotel that guests might want to visit during their stay. Both hotels and points of interest need to maintain geolocation data so that they can be found on maps for mashups, and to calculate distances.

Obviously, in the real world there would be many more considerations and much more complexity. For example, hotel rates are notoriously dynamic, and calculating them involves a wide array of factors. Here we're defining something complex enough to be interesting and touch on the important points, but simple enough to maintain the focus on learning Cassandra.

Here's how we would start this application design with Cassandra.

First, determine your queries. We'll likely have something like the following:

Find hotels in a given area.

Find information about a given hotel, such as its name and location.

Find points of interest near a given hotel.

Find an available room in a given date range.

Find the rate and amenities for a room.

Book the selected room by entering guest information.

Hotel App RDBMS Design

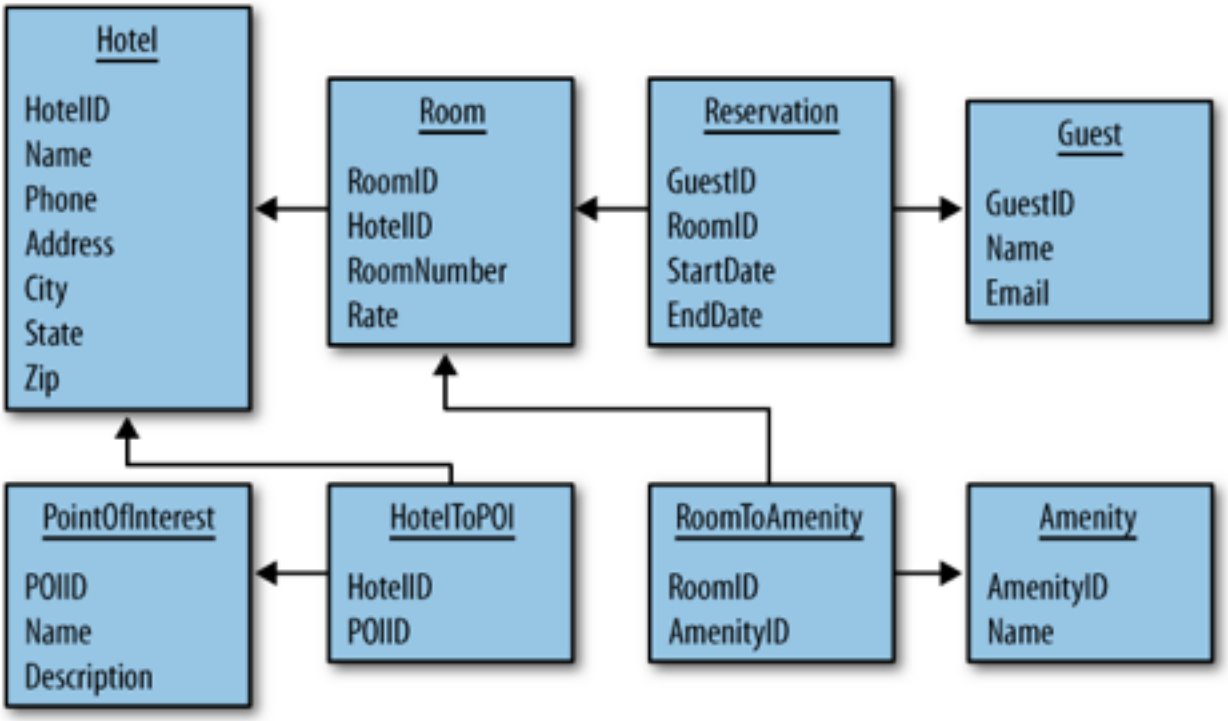


Figure : shows how we might represent this simple hotel reservation system using a relational database model. The relational model includes a couple of “join” tables in order to resolve the many-to-many relationships of hotels-to-points of interest, and for rooms-to-amenities.

A simple hotel search system using RDBMS

Figure : A simple hotel search system using RDBMS

Hotel App Cassandra Design

Although there are many possible ways to do it, we could represent the same logical data model using a Cassandra physical model such as that shown in Figure 4-2.

The hotel search represented with Cassandra's model

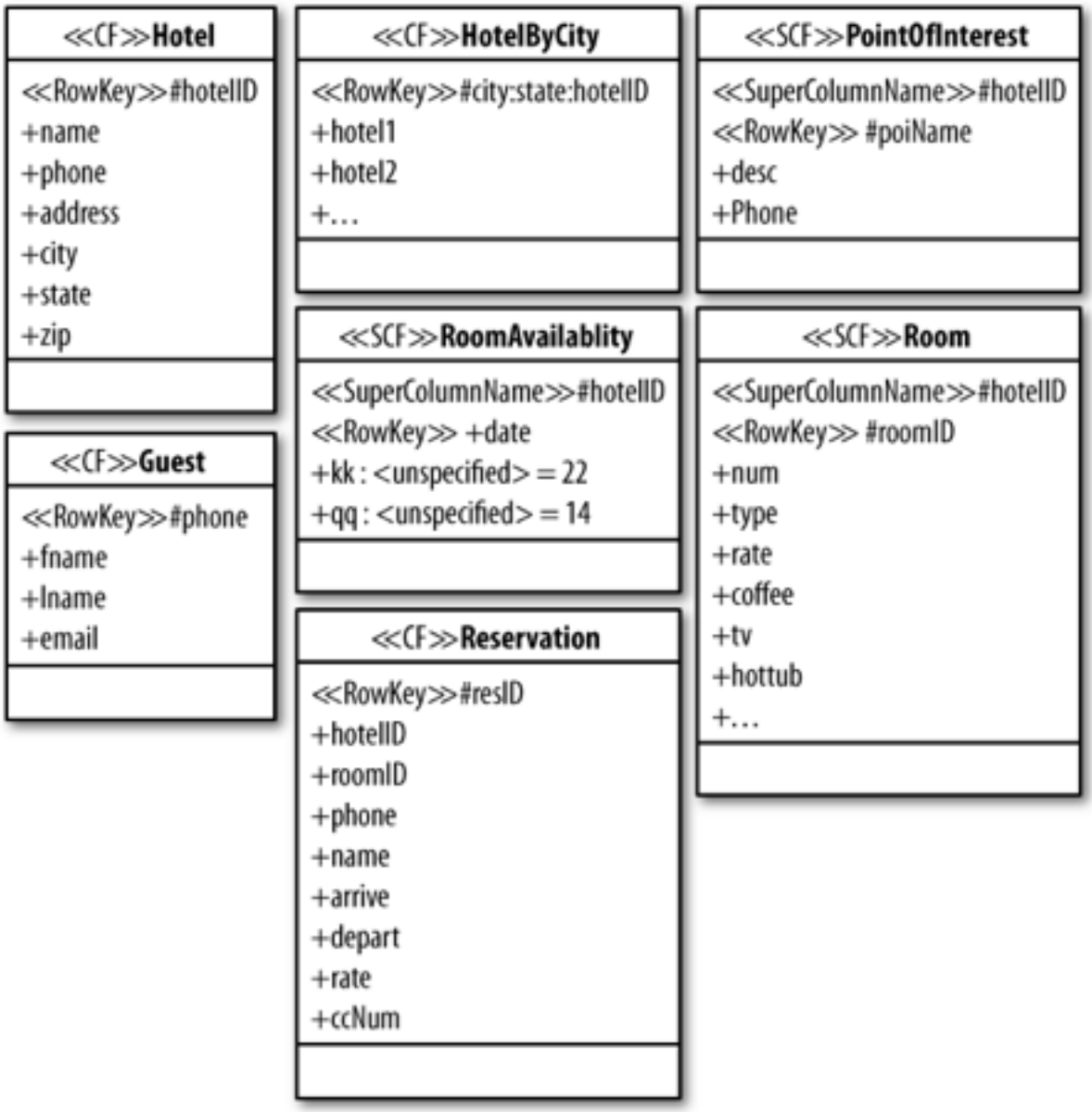


Figure : The hotel search represented with Cassandra's model

In this design, we're doing all the same things as in the relational design. We have transferred some of the tables, such as Hotel and Guest, to column families. Other tables, such as PointOfInterest, have been denormalized into a super column family. In the relational model, you can look up hotels by the city they're in using a SQL statement. But because we don't have SQL in Cassandra, we've created an index in the form of the HotelByCity column family.

NOTE

I'm using a stereotype notation here, so <<CF>> refers to a column family, <<SCF>> refers to a super column family, and so on.

We have combined room and amenities into a single column family, Room. The columns such as type and rate will have corresponding values; other columns, such as hot tub, will just use the presence of the column name itself as the value, and be otherwise empty.