Cassandra

J O E L B O H M A N
a n d J O H A N H I L D I N G

KTH Computer Science
and Communication

Bachelor of Science Thesis
Stockholm, Sweden 2010

Cassandra

J O E L B O H M A N
a n d J O H A N H I L D I N G

Bachelor's Thesis in Computer Science (15 ECTS credits)
at the School of Computer Science and Engineering
Royal Institute of Technology year 2010
Supervisor at CSC was Mads Dam
Examiner was Mads Dam

URL: www.csc.kth.se/utbildning/kandidatexjobb/datateknik/2010/
bohman_joel_OCH_hilding_johan_K10057.pdf

Royal Institute of Technology
School of Computer Science and Communication

KTH CSC
100 44 Stockholm

URL: www.kth.se/csc

Abstract

Cassandra is a distributed data storage system. This report explains why there is a need for non-relational databases and the current problems that relational databases face. It also describes what Cassandra does different and how those decisions affect performance, scaling and data consistency.

Referat

Cassandra är ett distributerat system för datalagring. Rapporten förklarar behovet för datalagringslösningar som inte är relationsdatabaser och vilka problem som relationsdatabaser ofta stöter på i större installationer. Rapporten förklarar hur Cassandra skiljer sig från relationsdatabaser. Den innehåller även en analys av hur Cassandras design påverkar systemets skalbarhet och prestanda.

Contents

Chapter 1

Introduction

With the introduction of Web 2.0 [1] sites on the Internet during the late 2000s, especially the new social medias [2], many new technical challenges have arisen. Of these challenges, one has received more attention than the others, namely data storage.

The problem is two-fold, there are more people browsing the Internet and also contributing to the website's content in a different way than earlier. Before the Web 2.0 revolution, there were a handful writers on a site and maybe a few people who commented on articles. Now sites like Facebook and Twitter are letting anyone post massive amount of content.

From a data storage perspective, this means an increasing amount of reads and writes to and from the database system. This report will outline how this creates problems for the data storage solutions of yesterday and the new data storage solutions that have emerged over the last few years.

One of these new solutions [3] to solve the problems mentioned is Cassandra. Cassandra is a massively scalable, decentralized structured database. Cassandra was at first developed by Facebook and during 2008 the source was released under an open source license [20]. Unlike most databases today, it is not a relational database, but a NoSQL [4] database. This report will explain what a NoSQL database is, what it implies for Cassandra and why Cassandra is one of the most powerful data storage system of 2010.

Chapter 2

Background

To fully understand this report, the reader should be familiar with the topics in this chapter.

2.1 Relational Database Management System

The Relational Database Management System (RDBMS) is a collective word for all database implementations based on the relational model which was developed by E.F. Codd in 1969 [10][11]. Data in the relational model is represented as relations which in turn are made up of a head and a body. The head consists of n attributes and the body consists of a set of n-tuples and every n-tuple contains n attribute values. Figure 2.1 illustrates a relation.

As the relational model began as a theoretical academic endeavor the naming scheme was very strict and conformed to mathematical and logic terms. When implementations of the relational model started emerging a new and much simpler naming scheme evolved based on tables. In Figure 2.2 the new naming scheme can be seen and it will be used throughout this report.

Every row in a table can be uniquely identified by a key which in turn is comprised of one or more columns. Another table can then reference this key and create a relationship. The concept of relationships is fundamental in relational databases. Every RDBMS supports at least one query language to extract data, the most com-

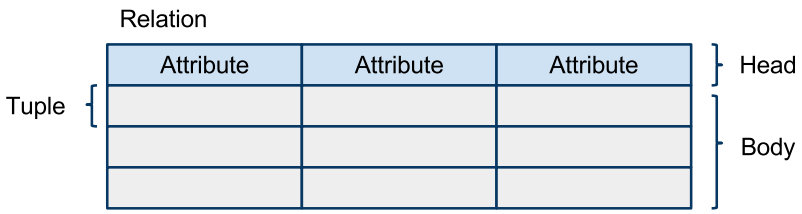

Figure 2.1. A relation and its components.

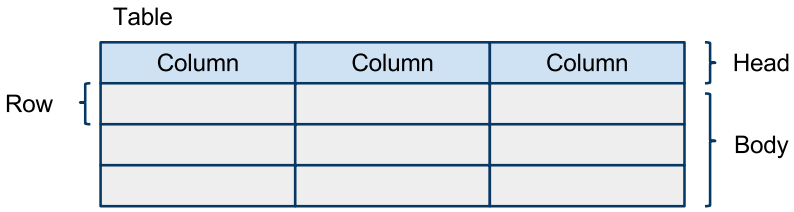

Figure 2.2. The table naming scheme.

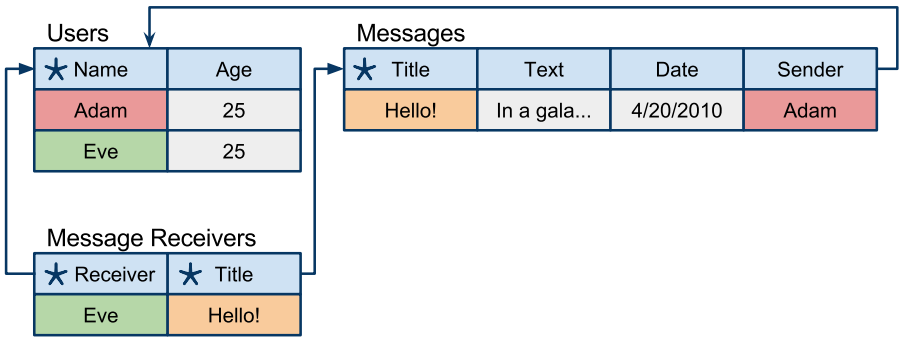

Figure 2.3. An example of relationships in a relational database.

mon being Structured Query Language (SQL). SQL supports join queries which is used to query data from relationships in a consistent way.

Figure 2.3 is an example data model that shows how relationships and keys work. The example consists of three tables: User, Message and Message Receiver. The User table contains two columns, name and age, where the name is a key and uniquely identifies every row in the table. The Message table contains four columns, title, text, date and sender. The Title is the key. The column sender is referencing the User table which creates a relationship between the two tables. The third one, Message Receiver, contains two columns, receiver and title, both of which are keys and referencing the two other tables.

Another important concept in relational databases is normalization, also introduced by E.F. Codd. The idea is that the data only exists in one table and this minimizes or eliminates anomalies when inserting, updating or deleting data.

2.2 Scalability in RDBMS

Scalability is the ability for a system to be able to handle more requests by adding more nodes or upgrading the hardware of existing nodes, without shutting it down. In 2010 scalability is a desired property in any database because of the ever-increasing quantities of data stored and processed every day. This was not always the case as most RDBMS were designed to be used on a single server.

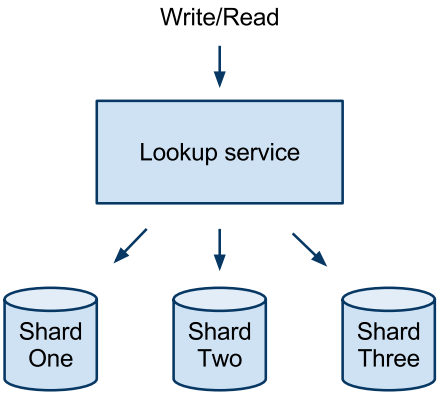

Figure 2.4. An example of a sharding technique with a lookup service.

But what happens when there is not enough hardware in a single server to handle thousands and millions of users? What happens when there is not enough disk space to handle the large quantities of data? These two problems have come about after the invention of the relational model and therefore the techniques to solve them are not optimal. Sharding and replication are two examples of the techniques solving these scalability problems sufficiently.

2.2.1 Sharding

Sharding is a technique in which data is partitioned across different servers (or shards, thereof the name). The partitioning can be done in several different ways. The least complex solution is simply to move a heavily used table to another server. This solution is not possible if the table in question has outgrown a server. The table must then be split up by its rows using a lookup service [18] to decide which servers get which rows. It is a more complex solution but will provide a higher level of scalability in the long run.

An example can be used to better explain the latter solution. Let us take the table named Message from the earlier example in this chapter. A large amount of messages are expected and one server alone cannot handle them. Therefore, three servers are used, Shard One, Shard Two and Shard Three and a lookup service is created. The lookup service checks the title of each message. If the title starts with A to H it will be sent to Shard One, I-P to Shard Two and Q-Z to Shard Three. This can be seen in Figure 2.4.

The sharding technique has a few problems. Mainly, features relating to table relationships such as SQL join queries might not work, it depends on the implementation details of the sharding technique. There is also the problem of making sure that all the different applications that use the database know how the sharding works. This is not very effective for developers nor for the operations team taking care of the database if there are a large number of shards but it is effective

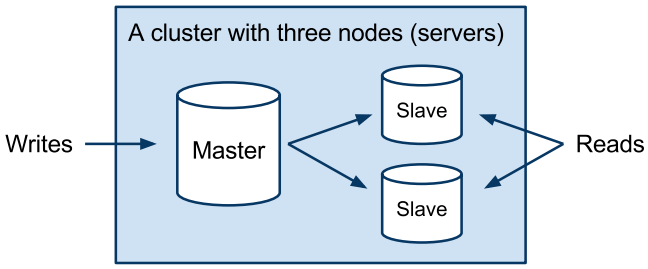

Figure 2.5. Master and slave database replication strategy.

performance wise.

2.2.2 Replication

The easiest technique to make an RDBMS handle an increasing amount of reads is to replicate the data across different servers. A standard replication strategy is to create a master and slave cluster [17]. Writes are sent to the master and from there pushed through to the slaves. Reads are sent to the slaves and handled by them. Figure 2.5 illustrates this strategy.

This replication strategy is not desired in an environment with more or equal amounts of writes compared to reads, because the writes will be pushed to the slave by the master anyway. A nearly optimal replication strategy would be one in which reads and writes can be sent to any server in the cluster, but that is not possible with RDBMS because of the CAP theorem.

2.3 CAP theorem

The CAP theorem [12] (Consistency, Availability, Partition tolerance theorem, also referred to as Brewer's theorem) is a famous theorem about trade-offs that has to be made when designing highly scalable systems. The theorem is not directly related to Cassandra, but understanding this theorem helps with the understanding of why Cassandra is designed the way it is.

It lists three properties that are interesting to highly scalable systems, consistency, availability and partition tolerance:

2.3.1 Consistency

For the system to be consistent according to the CAP theorem, querying all the nodes in the system should return the same data. This means that a write to one node has to be replicated to the other nodes in the system. Coming from an RDBMS background, this might sound silly not to have. However, this is not the case, as enforcing consistency means that the system will have to sacrifice other aspects, such as speed.

2.3.2 Availability

Availability means that either the system is available or not. This is most commonly sacrificed when the system needs to lock itself to make a write, because then it will be unavailable until the lock is released.

2.3.3 Partition tolerance

If a system is partition tolerant, it will continue to function (respond with correct data) regardless of failure to communicate within the system. The a common occurrence of this is when a network cable is cut and the nodes in the system are divided into two partitions that cannot communicate with each other. When this happens, querying either of the two partitions should return the correct data.

Another way to describe partition tolerance is "No set of failures less than total network failure is allowed to cause the system to respond incorrectly." [5].

2.3.4 Choosing two of these

The CAP theorem says that a distributed system cannot have all three of these properties, it can have at most two. This does not mean that the system has to sacrifice caring about the last property, but that it will not handle it as well as the property is stated in the theorem.

For a reader familiar with RDBMS, it might be insightful to know that most, if not all, RDBMS's have chosen CA, consistency and availability. That means they lack P, partition tolerance. However, this does not mean that RDBMS do not handle network failures, because they do, with a master/slave approach and/or sharding [8]. To reiterate, a system that has chosen two of the three does not mean that it has ignored the last, but solved the issue in a weaker way than the CAP theorem defines it.

2.4 Google BigTable

Google BigTable is Google's attempt to create a high-performance distributed data storage. It was one of the first [3] NoSQL databases and the research paper describing BigTable [7] is still an important paper in the field of distributed systems. The implementation of Google BigTable is proprietary, but there is an open source project, Apache HBase [9], aiming to implement BigTable. At Google, BigTable is used in Google Search, Google Earth, Google Finance and is available to all applications that run on the Google App Engine [7].

Cassandra has borrowed ideas from BigTable, such as the data model and ways to handle data in memory and on disk in an efficient way. They also share the implementation detail that they are both supposed to run on commodity hardware. In CAP terms, Google BigTable focuses on consistency and availability.

2.5 Amazon Dynamo

Around the same time that Google started developing BigTable, Amazon set out to create their own distributed data storage, Amazon Dynamo [14]. Their goals differed slightly from Google's and that has resulted in a system that is more fault-tolerant and reliable but has a simpler data model. Google BigTable's data model is similar to a 5-dimensional hash map, whereas Amazon Dynamo's is a 1-dimensional hash map (also known as a key-value map).

However, BigTable has a single point of failure in the node that manages the BigTable cluster. In a Dynamo cluster, all the nodes are equally important, which means that any one of them can fail and the cluster will keep on functioning. Cassandra borrows this fail-proof design and focuses on the same CAP properties as Dynamo, A and P. Cassandra is often described as BigTable on top of Dynamo, merging the best parts of both.

Chapter 3

Cassandra

Like Amazon Dynamo and Google BigTable, Cassandra is a high-performance distributed data storage. Cassandra is decentralized, every node is identical and there are no single points of failure. It is also elastic, the read and write throughput increases linearly as new nodes are added to the cluster. It has a rich data model, but can also be used as a key-value storage.

Cassandra has all of these great features, but what are its weaknesses? It is not a relational database and it cannot be queried using SQL queries. It does not allow arbitrary indexing of data and searching through the data is limited to range queries over columns.

It is a top-level Apache open-source project [23] with contributors from large companies such as Facebook, Digg and Rackspace. Cassandra was designed with performance in mind and to be run on commodity hardware. It does not require expensive hardware such as SANs or SSDs to achieve great performance.

Cassandra should be used in applications where large quantities of data are expected, in the terabyte range. If strong data integrity is needed Cassandra is not the best alternative, it is up to the application using Cassandra to enforce integrity.

3.1 Data model

In Cassandra, data is not stored in tables but in a 5-dimensional hash map. This means that to get a single value from the database, you have to specify each of the five dimensions which will be introduced in this chapter. Initially, this might seem like a lot of work to fetch data, but the hash map-like data model is very flexible and easy to work with in most programming languages.

3.1.1 Keyspace

Keyspace is the outer most dimension. It is a term well known in programming, variables and functions live in a keyspace to avoid name conflicts. Cassandra's keyspaces are like the schemas (databases) in RDBMS.

3.1.2 Column Family and Row

Inside the keyspaces, there are column families, the second dimension. Every column family has a name and an unspecified number of rows. A row consists of a row key, the third dimension, and an unbounded number of columns. These rows are nothing like the relational database rows. A row in a relational database is a n-tuple and in Cassandra a row consists of infinitely many columns.

3.1.3 Column

A column is the smallest piece in Cassandra's data model. It is a triplet (a 3-tuple) that contains a name, a value and a timestamp. The name and value are self-explanatory. The timestamp is used to determine if the value should be updated or not. If a write to a column is done with a timestamp that is less than the previous timestamp it will be discarded. The timestamp is set by the application, not by Cassandra itself.

3.1.4 Super column

As the data model has been explained so far, there are only four dimensions, keyspace, column family, row and column, but Cassandra's data model is five dimensional. There is a fifth dimension, the super column. It consists of a name and an infinite number of columns. A column family can either contain columns or super columns, not both.

3.1.5 Querying the data model

Besides direct access to the 5-dimensional hash map, Cassandra also supports range-queries to fetch columns. A range-query returns an interval of columns from a sorted sequence of columns. The column family can only be sorted in a single predetermined way, limiting the kind of range-queries that can be performed.

Cassandra does not have any arbitrary indices to support arbitrary questions as a relational database has. The modeling of a keyspace in Cassandra therefore need to account for what questions it should be able to answer. Using the previous example from the RDBMS chapter, what questions should it be able to answer?

- What messages were sent by a specific user?
- What messages were received by a specific user?
- What messages were sent or received during a specific date or period?

To answer the above questions two column families are needed, both containing only super columns. The first column family contains all messages sent by a specific user and each super column is sorted by date. Inside the super column there is a column which contains the receiver among other columns. It is up to the application

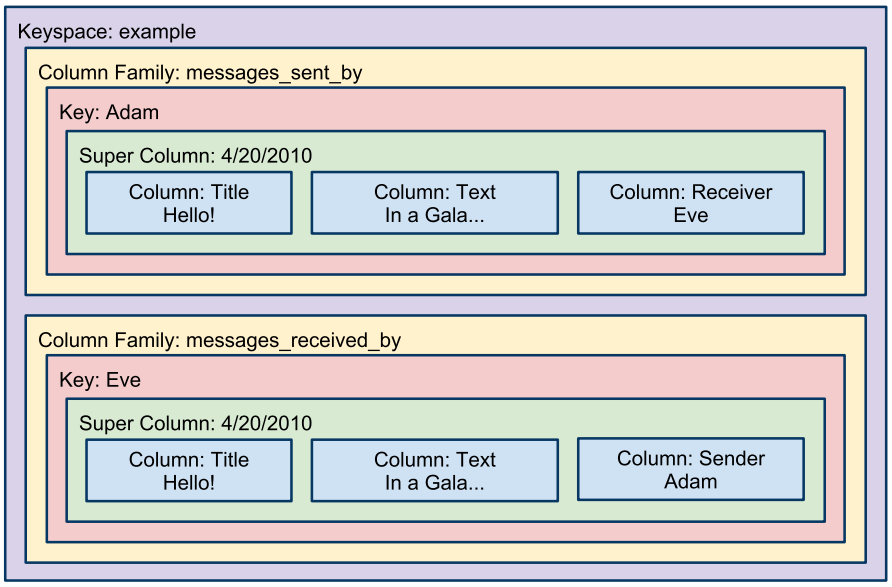

Figure 3.1. An example of a data model in Cassandra.

to define the relationship of the receiver column. The second column family contains the same columns except that the receiver has been replaced with a sender column. Figure 3.1 shows the full keyspace.

An example of an API request to list all messages sent by Adam between two dates might look like this (in pseudo code):

```
get_columns('example.messages_sent_by.Adam', 'between 4/10/2010 and now');
```

3.2 Internal data storage

To achieve high performance per node, Cassandra has to minimize the amount of read and write accesses to the secondary memory (hard disk drive). Cassandra does this by using a lot of primary memory (RAM) which is fast compared to the secondary memory. But data in Cassandra cannot only rely on the primary memory because of its volatile property and size. Therefore, data written to Cassandra need to go through three checkpoints to be safely stored on a node. Those three checkpoints are: Commit Log, Memtable and SSTable [19].

3.2.1 Commit Log

The responsibility of the Commit Log is to write the data onto the secondary memory. This is done to protect the data in the event of a power failure or a computer

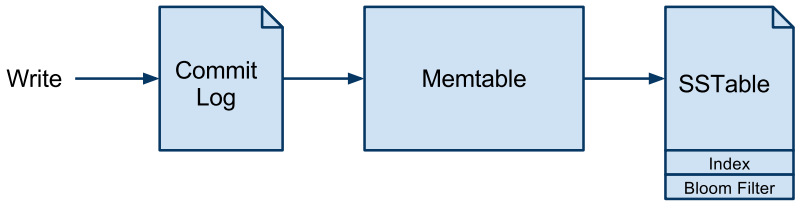

Figure 3.2. A diagram of a write taking place in Cassandra.

crash. The Commit Log does this efficiently by writing the data in a sequential order and therefore only requires a fast write instruction. No read or seek instructions are needed to be called on the secondary memory.

3.2.2 Memtable

The Memtable is an in-memory key-value lookup table. When data is safely stored on a non-volatile memory by the Commit Log the data is written to the primary memory in a Memtable. It is worth noting that there exists a Memtable for every column family but the Commit Log is keyspace wide.

3.2.3 SSTable

The Sorted String Table (SSTable) is the final storage for data written to Cassandra. An SSTable is an immutable structure that represents the Memtable as a file on the secondary memory. It is a key-value table sorted by the key.

The process of writing data from the primary memory to the secondary memory is called flushing. A flush occurs per column family when a Memtable reaches a predefined threshold. In the flush process three files are written to the secondary memory: the SSTable file, an index file and a filter file.

As the SSTable is immutable it can be written to the secondary memory in sequential order which means no read or seek instructions. The same goes for the index file and the filter file.

Index file

The index file is also sorted by key but holds no actual data, only offsets to where the data can be found in the SSTable file. The purpose of this file is to minimize the number of seek instructions to find the data for a specific row key.

Filter file

The filter file is actually a Bloom Filter [16] and is used to test if a row key exists or not without having to read the index file. False positives can occur but not false negatives, which means that in the worst case the index file also needs to be tested if the row key exists or not.

Compaction

Over time data files accumulate and read performance will decrease because more seek instructions is needed to find a specific key and its value. The process to prevent this degrading performance is called compaction. When four or more SSTable files exist they will be merged into a new larger SSTable file with a new index file and a new filter file. This process can continue as long as the secondary memory has enough capacity.

3.3 Gossip

Any distributed system needs a way to keep track of nodes, to let new nodes find other nodes and to let the nodes know about the state of the other nodes. The naive solution to this is using a single master node that is responsible for keep track of the rest of the nodes. But, since Cassandra is designed with high availability in mind, introducing a single point of failure is not OK.

Instead Cassandra uses a gossip style protocol [13], where the nodes chatter among themselves. This is not a protocol in the same way that FTP or HTTP is, but rather a concept. Assume that we have a running Cassandra cluster with nodes A, B, C and D. Assume that we have a node E that we want to add.

We let node E know about one (or more) of the existing nodes, called a seed(s). E then asks the seed (let's assume A is the seed) for more nodes, and A replies with B and C. Then E announces itself to B and C, asks for more nodes and C replies with D. Then E announces itself to D, and the introduction is complete.

This design assures that the work of introducing a new node to a running instance is done by the new node, to make sure that the already running nodes use their resources to respond to requests. For a more technical overview, see [6].

3.4 Failure detection and prevention

To determine whether or not to try and contact a node, local nodes need to detect if other nodes are running or not. To determine the nodes status, Cassandra uses a modified version of the Φ Accrual Failure Detection [24]. It uses heartbeats (similar to gossip) to determine if a node is trusted to be up or suspected to be down. The failure detection does not emit a boolean value for a certain node, but instead returns a suspicion value, Φ. If this suspicion value is above a certain threshold, no more reads or writes will happen to that node. The implementation details are available in [6].

If a node receives a write that is supposed to go to a node that is flagged as down, it will store the write for later redirection. This is called hinted handoff [21], because when the node that is flagged down starts to send out heartbeats again, the write will be sent to it. This means that the node that returns from failure will be returning correct faster than if the node was going to be brought back to correct

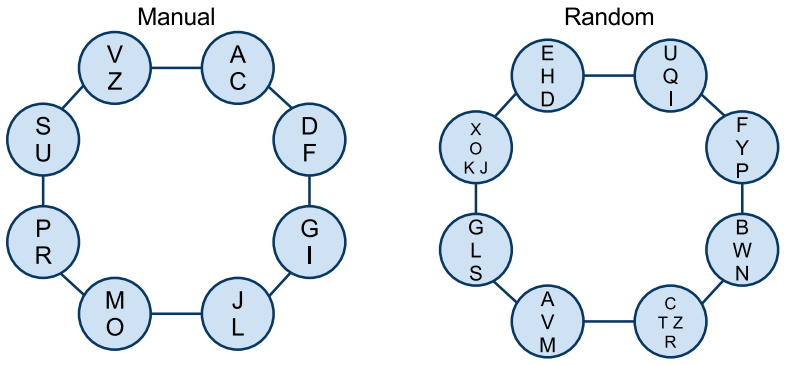

Figure 3.3. Data spread out on nodes using manually and randomly assigned ranges.

data through read repair. This is done to make sure that from a client's point of view, the write will always be fast.

If data was written with a consistency level greater than one, and the read request was sent with consistency level one, a read request will only fail when all the nodes containing the requested data are unreachable. This means that failure prevention can be controlled by the client, by increasing the consistency level of the write. However, the higher the consistency level, the longer it takes to make the write (since the write will be sent to more nodes with a high consistency level).

3.5 Why and how Cassandra scales

Data storage systems grow in two ways, nodes are upgraded or more nodes are added. If the system handles these additions well, it is said to scale well. In this case, handling the addition well means that the system's performance should increase in a linear fashion. Both Cassandra and RDBMS's are greatly helped by upgrading the hardware of the already existing nodes, adding better disks or more primary memory.

In the background chapter of this report, the way that RDBMS's store more data than a single node can handle were sharding. In this section, Cassandra's handling of the same problem will be explained.

In Cassandra, there are two ways to partition your data across the nodes in the cluster. One way is to randomly spread out the data (using the RandomPartitioner) by randomizing the ranges every node takes care of and the other one is to manually assign ranges to nodes. Manually assigning ranges to nodes is similar to the sharding technique used by RDBMS's, but there is no need to have an external system to keep track of which node to query for data.

3.5.1 Scaling writes

Is random assignment of the ranges any good? It is often the case that it is very difficult to foresee the usage patterns of the database before the application is up and running. With manually assigned ranges there is a big risk of overloading a single node, either by writes or by reads. Using a random assignment of equally sized ranges, it is much easier (from an operational perspective) to "load balance" the ranges.

The load balance procedure makes sure that the hard drive load (the data stored) is balanced across all available nodes, by adjusting the ranges and gossiping this to the other nodes. This helps when a single node is getting lots of writes, but not when the same columns are read over and over again.

3.5.2 Scaling reads

To scale the reading, Cassandra makes use of the possibility to replicate writes and a feature called read repair. When writing a column to Cassandra, a column and a consistency level is sent to any node in the cluster. The consistency level, in combination with the replication factor setting (a cluster wide setting), decides how many nodes the column should be written to. If the consistency level is greater than one, there will be more than one node to read the data from.

As with writes, the requests for data also contain a consistency level. If the consistency level is low and the node receiving the request does not have the data, it will send out a request for the data and reply to the client with the data received first. Then, in the background, it will receive the responses from the rest of the nodes. If these new responses contain a newer timestamp than the data sent to the client, the node will let the nodes with old data know that there is new data.

This means that sometimes old data will be returned to clients, and this is why Cassandra in CAP terms is a AP system and not a CA one. In Cassandra, the client has a choice. Either get a quick response with potentially old data or get the newest data with a slight delay.

Chapter 4

How Cassandra differs from RDBMS

Cassandra has taken a completely different approach to database design than the one of relational databases. Instead of building on the relational model from 1969, the team behind Cassandra has taken a look at what problems exists for current RDBMS and how these problems can be solved. Most of the problem solving has been done by the teams behind BigTable and Dynamo. Cassandra is in a way a prodigy of the two.

From an architect's or a system designer's perspective, the core difference between Cassandra and a relational database is what to think about when modeling your data. For a relational database, the model is constructed from how the data relates to itself. For example, the users get put in one table and the messages in another. In Cassandra the main concern is in which way your data will be accessed. Will most of the requests be for a single column? Do we need the possibility to list all the messages sent by a certain user?

From an application developer's point of view, using SQL queries to fetch data from an RDBMS is often a bit complex to integrate into the applications code. This stems mostly from SQL being a declarative language and not a functional or object oriented language like most modern programming languages. In comparison to this, working with Cassandra is like working with a hash map, which is very natural in most programming languages.

From the perspective of the operational team (the team that takes care of the Cassandra cluster) Cassandra provides great and easy ways to replicate data to different data centers compared to most (but not all) RDBMS solutions.

Cassandra cannot really be compared to a RDBMS for just one server. Cassandra is not built to be used on one server, but it is still possible. Performance wise an RDBMS and Cassandra are close for a single server solution but with a RDMBS you get consistency and strong data integrity. The storage will be more space efficient because of the normalized data and due to arbitrary indices a RDBMS can answer any question it is asked in the domain of the data.

In an environment where data rapidly change or is added, Cassandra is a good fit. It is easy to add new nodes to a Cassandra cluster. There is no need to

use external tools to complicate it. In the background chapter of this report we mentioned sharding. This is built into Cassandra, the same goes with replication. Both of these feature are hard to implement with a RDBMS especially because of the CAP theorem. You have CA with RDBMS and start to add new servers and want P too, either way you lose C or A which is not good. Cassandra is designed not to have C, but this can be tweaked at the expense of response time (and/or failure prevention) which is to a degree A.

Chapter 5

Real world usage

Since Cassandra's release as an open source project, it has been put to use in many different projects. The most famous and biggest (in terms of cluster size) user of Cassandra is still Facebook, where it powers the inbox search [20]. Two early adopters are Twitter, the world's largest micro-blogging service and Digg, a social news site. The core feature at Digg, digging a link, is backed by Cassandra. As of September 2009, they stored over 76 billion columns at a total of about 3 TB [22].

We, the authors of this report, think it is the performance and the inexpensive scalability of Cassandra together with the active community that makes Cassandra such an attractive choice. Until recently, there was no commercial support for Cassandra. But a startup named Riptano [15] now provides service and support for Cassandra, which might open new doors.

Cassandra is still young and has not, as far as we know, been used in critical applications such as banking, customer relationship management, or research data storage. Remembering that Cassandra is only a few years old and looking at the adoption rate of relational databases, there is a great chance that Cassandra will expand into new usage areas in the future.

However, it must be remembered that most applications do not need the scalability or performance that Cassandra provides. Cassandra is a niche database for use with large quantities of data or a high user count. But on the other hand, data is getting more valuable and it needs to be stored and processed somewhere.

Chapter 6

Conclusion

As the saying goes, "If all you have is a hammer, everything looks like a nail". Cassandra does not provide a better hammer (a better way to store relational data), but another tool to the toolkit of developers.

In this report, we have shown that Cassandra has a sound and proven theoretical foundation. Together with the smart tricks used to implement the ideas, Cassandra is a serious contender in the NoSQL space. It is definitely a project to keep your eye on in the future!

Bibliography

[1] O'Reilly, T. (2005). What is Web 2.0 [www]. Retrieved from <http://oreilly.com/web2/archive/what-is-web-20.html> on March 9th, 2010.

[2] Wikipedia (2010). Social media [www]. Retrieved from <http://en.wikipedia.org/wiki/Social_media> on March 9th, 2010.

[3] Vineet Gupta (2010). NoSql Databases - Landscape [www]. Retrieved from <http://www.vineetgupta.com/2010/01/nosql-databases-part-1-landscape.html> on March 9th, 2010.

[4] The NoSQL Community (2010). NoSQL Databases [www]. Retrieved from <http://nosql-database.org> on April 19th, 2010.

[5] Browne, J. (2009). Brewer's CAP Theorem [www]. Retrieved from <http://www.julianbrowne.com/article/viewer/brewers-cap-theorem> on April 13th, 2010.

[6] Lakshman A., Malik P. (2009). Cassandra - A Decentralized Structured Storage System [www]. Retrieved from <http://www.cs.cornell.edu/projects/ladis2009/papers/lakshman-ladis2009.pdf> on April 19th, 2010.

[7] Jeffery Dean et. al. (2006). Bigtable: A Distributed Storage System for Structured Data. In Proceedings of OSDI 2006, Seattle, WA.

[8] Gautier, T. (2010). Characterizing Enterprise Systems using the CAP theorem [www]. Retrieved from <http://javathink.blogspot.com/2010/01/characterizing-enterprise-systems-using.html> on April 25th, 2010.

[9] Apache Foundation (2010). Apache HBase [www]. Retrieved from <http://hadoop.apache.org/hbase/> on April 25th, 2010.

[10] Codd, E. F. (1969). Derivability, Redundancy, and Consistency of Relations Stored in Large Data Banks. San Jose, IBM Research Report RJ599.

[11] Codd, E. F. (1970). A Relational Model of Data for Large Shared Data Banks. Communications of the ACM, Volume 13, Issue 6.

[12] Brewer, E. (2004). Towards Robust Distributed Systems [www]. Retrieved from <http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf> on April 27th, 2010.

[13] Renesse, v. R. et. al. (2008). Efficient Reconciliation and Flow Control for Anti-Entropy Protocols. In proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware.

[14] Vogels, W. et. al. (2007). Dynamo: Amazon's highly-available key-value store. In proceedings of the SIGOPS, p. 205-220.

[15] Riptano (2010). Riptano - Service and Support for Apache Cassandra [www]. Retrieved from <http://www.riptano.com> on April 29th, 2010.

[16] Burton H. Bloom (1970). Space/time trade-offs in hash coding with allowable errors. Communications of the ACM, Volume 13, Issue 7.

[17] MySQL AB. (2005). Replication in MySQL. Retrieved from <http://dev.mysql.com/doc/refman/5.0/en/replication.html> on May 2nd, 2010.

[18] Tocker, M. (2009). MySQL Performance Blog: Why you don't want to shard. Retrieved from <http://www.mysqlperformanceblog.com/2009/08/06/why-you-dont-want-to-shard/> on May 2nd, 2010.

[19] Apache Foundation (2010). Architecture Overview. Retrieved from <http://wiki.apache.org/cassandra/ArchitectureOverview> on May 2nd, 2010.

[20] A. Lakshman (2008). Cassandra - A structured storage system on a P2P Network. Retrieved from <www.facebook.com/note.php?note_id=24413138919> on May 2nd, 2010.

[21] Cassandra Wiki (2010). Hinted Handoff. Retrieved from <http://wiki.apache.org/cassandra/HintedHandoff> on May 2nd, 2010.

[22] Eure, I. (2009). Looking to the future with Cassandra. Retrieved from <http://about.digg.com/blog/looking-future-cassandra> on May 2nd, 2010.

[23] Riou, M. (2010). Cassandra is an Apache top level project. Retrieved from <http://www.mail-archive.com/cassandra-dev@incubator.apache.org/msg01518.html> on May 2nd, 2010.

[24] Naohiro Hayashibara , Xavier Defago , Rami Yared , Takuya Katayama. (2004). The Φ Accrual Failure Detector. Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems (SRDS'04), p.66-78.