Cassandra (DHT)

Dhairya Gala
dmgala@indiana.edu

Department of Computer Science
Indiana University, Bloomington

Abstract:
Cassandra is an open source distributed database management system designed to handle very large amounts of data spread out across many commodity servers while providing a highly available service with no single point of failure.
This paper provides an architectural overview of Cassandra and discusses how the distributed system design and principles are applied to real systems.

1. Introduction:
Cassandra is a NoSQL (Not only SQL) solution initially developed by Facebook and powers their Inbox Search feature. Jeff Hammerbacher, who led the Facebook Data team at the time, described Cassandra as a Big Table data model running on an Amazon Dynamo-like infrastructure. After being submitted to the Apache Software Foundation Incubator in 2009, Cassandra was accepted as a top-level Apache project in February 2010.
Cassandra is a distributed, structured key-value store capable of scaling to arbitrarily large data sets with no single point of failure. With the growth of systems to cover local and wide area networks, the continued smooth functioning of system is important, despite crashed routers, broken links and failed nodes.
Fault-tolerant, structured data stores for working with information at such a scale are in high demand. Also, the importance of data locality is growing as systems span large distances with many network hops.
Cassandra is designed to continue functioning in the face of component failure in a number of user configurable ways. It enables high levels of system availability with tunable consistency. Data in Cassandra is optionally replicated onto N different peers in its cluster. The gossip protocol ensures that each node maintains state regarding each of its peers. This reduces the hard depends-on relationship between any two nodes in the system which in turn increases availability and partition tolerance.
Section 2.1 discusses the fundamental characteristics that the distributed systems must give up to gain resiliency to inevitable failures at sufficiently large scale.
Section 2.2 describes the basic data model Cassandra uses to store data and contrasts that model against traditional relational databases.
Section 2.3 discusses distributed hash table (DHT) and section 2.3.1 discusses how Cassandra's implementation of a DHT enables load balancing with in the cluster.
Section 2.4 discusses Cassandra's architecture. Section 2.4.1 describes anatomy of writes and reads, 2.4.2 discusses data replication within a cluster and 2.4.3 discusses Cassandra's tunable consistency model for reading and writing data. Section 2.4.4 contrasts the consistency and isolation guarantees of Cassandra against traditional ACID-compliant databases. Section 2.4.5 discusses cluster growth.
Section 2.5 discusses client interaction with Cassandra.

2. BODY
2.1 CAP Theorem and PACELC
Cassandra is a distributed, NoSQL solution. NoSQL is a term used to designate database management systems that differ from classic relational database management systems (RDBMS) in some way. These data stores may not require fixed table schemas, usually avoid join operations and typically scale horizontally.
CAP, first conceived in 2000 by Eric Brewer and formalized into a theorem in 2002 by Nancy Lynch, has become a useful model for describing the fundamental behavior of NoSQL systems.
CAP is generally described as following: when you build a distributed system, of three desirable properties you want in your system: consistency, availability and tolerance of network partitions, you can only choose two.
Consistency - all copies of data in the system appear the same to the outside user at all times.
Availability - the system as a whole continues to operate despite of node failure.
Partition-tolerance - the system continue to operate in spite of arbitrary message loss (due to crashing of router, broken network link, etc. preventing communication between nodes).
Depending on usage, the Cassandra user can select either Availability and Partition-tolerance or

Consistency and Partition-tolerance. The main problem with CAP is that it focuses on a consistency/availability tradeoff, resulting in a perception that the reason why NoSQL systems give up consistency is to get availability.

Daniel Abadi of Yale University's Computer Science department has described a refining model referred to as PACELC which he uses to clarify some of the misconceptions about the CAP Theorem and incorporated latency into the trade-off space, which more closely explains the designs of many modern NoSQL systems.

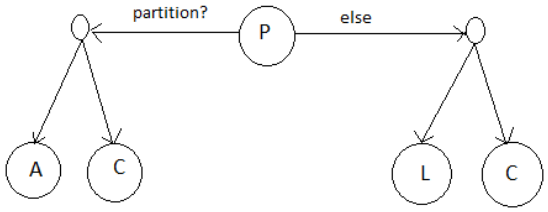

Figure 1: PACELC Tradeoffs for Distributed Data Services

It states that when a system experiences partitioning P, tradeoffs must be made between availability A and consistency C. Else under normal operation it must make tradeoffs between consistency C and latency L. However, when the system experiences partitioning, Cassandra must sacrifice consistency, to remain available. This is because write durability is impossible when a replica is present on a failed node.

2.2 Data Model

Cassandra is a distributed key-value store. Cassandra only allows data to be queried by its key. Additionally, indexes on non-key columns are not allowed. This is unlike SQL queries which allow the client to express arbitrarily complex constraints and joining criteria. Also, the application must implement any joining of related rows of data; Cassandra does not include join engine. Hence, the Cassandra data modeler must choose keys that can be derived or discovered easily and ensure maintenance of referential integrity. Cassandra has adopted abstractions that closely align with the design of Bigtable. The primary units of information in Cassandra are described as follows:

Column - A column is the atomic unit of information and is of the form name: value.

Super Column - Super columns group together columns with a common name and are useful for modeling complex data types.

Row – It is the uniquely identifiable data in the system which groups together columns and super columns. Every row in Cassandra is uniquely identifiable by its key.

Column Family - A Column Family is the unit of abstraction containing keyed rows which group together columns and super columns of highly structured data. They have no defined schema of column names and types supported.

The logic regarding data interpretation is in the application layer, unlike relational databases which requires predefined column names and types. All column names and values are stored as bytes of unlimited size and are usually interpreted as either UTF-8 strings or 64-bit long integer types. In addition, columns within a column family can be sorted either by UTF-8- encoded name, long integer, timestamp, or using a custom algorithm provided by the application. The sorting criteria cannot be changed and hence must be wisely chosen depending upon the semantics of the application.

Keyspace - The Keyspace is the top level unit of information in Cassandra. Column families are subordinate to exactly one keyspace.

The queries for information in Cassandra take the general form:

get (keyspace, column family, row key).

The definition of new keyspaces and column families must be known at the time a Cassandra node starts. Also, the configuration must be common to all nodes in a cluster. Thus, changes to either will require an entire cluster to be rebooted. However, once the appropriate configuration is in place, the only action required for an application to change the structure or schema of its data, is to start using the desired structure. When information structure in Cassandra needs to be updated, the application grows in complexity by maintaining additional code which can interpret old data and migrate it to the new structure.

This at times, underscores the importance of choosing a data model and key strategy carefully early in the design process.

2.3 Distributed Hash Tables

A Distributed Hash Table is a class of a decentralized distributed system that provides a look up service similar to a hash table. (key, value) pairs are stored in a DHT, and any participating node can efficiently retrieve the value associated with a given key.

For look-up of data on a central server, users would connect to the central server, browse for the data they want, and request that data over a point-to-point connection with the server containing the referenced data.

However, there are a few problems with this approach. Any sufficiently large distributed system with a central coordinating node will experience bottlenecks at that node. Large amount of strain in terms of processing power and bandwidth can arise at the central node which can make the entire system appear unavailable. DHTs offer a scalable alternative to the central server lookup which distributes lookup and storage over a number of peers with no central coordination required.

Any read or write operation on a DHT must locate the node containing information about the key. This is done through a system of key-based routing. Each node participating in a DHT contains a range of keys stored along with information about the range of keys available at other nodes in the system.

Any node contacted for information regarding a key forwards that request to the next nearest node according to its lookup table. The more information each node maintains about its neighbors, the fewer number of hops are required to get to the correct node.

While maintaining more state at each node means lower latency lookups due to a reduced number of hops, it also means nodes must exchange a greater amount of information about one other. The tradeoff between lookup latency and internal gossip between nodes is a fundamental driver behind DHT design. The table below shows some pairings of interconnectedness and lookup complexity of common DHT solutions.

Connections	Number of Hops
O (1)	O (n)
O (log n)	O (log n)
O (sqrt n)	O (1)

2.3.1 Balanced Storage

Cassandra's DHT implementation achieves O (1) lookup complexity and is often referred to as a one hop DHT. This is a function of the gossip architecture in Cassandra which ensures each node eventually has state information for every other node. This includes the range of keys it is responsible for, a listing of other nodes and their availability, and other state information.

Like other DHT implementations, nodes in a Cassandra cluster can be thought of as being arranged in a ring. Servers are numbered sequentially around a ring with the highest numbered connecting back to the lowest numbered. Each server is assigned a unique token which represents the range keys for which it will be responsible for. The value of a token t may be any integer such that $0 <= t <= 2^{127}$.

Keys in Cassandra may be a sequence of bytes or a 64-bit integer. However, they are converted into the token domain using a consistent hashing algorithm. MD5 hashing is used by default but an application may supply its own hashing function to achieve specific load balancing goals. If a node n in a token ring of size N has token t_n it is responsible for a key k under the following conditions.

$0 < n < N$

$t_{n-1} < md5\ (k) <= t_n$

The node with the lowest token value, n = 0, completes the token domain and is responsible for all keys k matching the following criteria. It is often referred to as the wrapping range.

$md5\ (k) > t_{N-1}$

$md5\ (k) <= t_0$

Thus, in Cassandra, MD5 hashing algorithm is used for even key distribution around token ring.

This even distribution of data within a cluster is essential to avoiding hotspots that could lead to overburdening a server's storage and capacity to handle queries.

2.4 Architecture

DHTs form the basis of understanding Cassandra's architecture. In the following sections we understand how Cassandra implements reliable, decentralized data storage over DHT internals.

2.4.1 Anatomy of Writes and Reads

To the user, all nodes in a Cassandra cluster appear identical. The fact that each node is responsible for managing a different part of the whole data set is transparent. While each node is responsible for only a subset of the data within the system, each node is capable of servicing any user request to read from or write to a particular key. Such requests are automatically proxied to the appropriate node by checking the key against the local replica of the token table.

Once a write request reaches the appropriate node, it is immediately written to the commit log which is an append-only, crash recovery file in durable storage.

The only I/O for which a client will be blocked is the append operation to the commit log which keeps write latency low. A write request will not return a response until that write is durable in the commit log unless a consistency level of ZERO is specified. Simultaneously an in-memory data structure known as the memtable is updated with this write. Once this memtable reaches a certain size it too is periodically flushed to durable storage known as SSTable.

Reads are much more I/O intensive than writes and typically incur higher latency. Reads for a key at one of the replica nodes will first search the memcache for any requested column. Any SSTables will also be searched. Because the constituent columns for a key may be distributed among multiple SSTables, each

SSTable includes an index to help locate those columns. As the number of keys stored at a node increases, so do the number of SSTables. To help keep read latency under control, SSTables are periodically consolidated.

2.4.2 Replication

Cassandra's storage engine implements a MD5-keyed distributed hash table to evenly distribute data across the cluster. The DHT itself does not provide for fault-tolerance since its design only accounts for a single node at which information regarding a key resides. To allow keys to be read and written even when a responsible node has failed, Cassandra will keep N copies distributed within the cluster. N is also known as the replication factor and it can be configured by the user. The hashing function provides a look up to the server primarily responsible for maintaining the row for a key. However each node keeps a listing of N −1 alternate servers where it will maintain additional copies. This listing is part of the information gossiped to every other node. When a live node in the cluster is contacted to read or write information regarding a key, it will consult the nearest copy if the primary node for that key is not available. Higher values of N contribute to availability and partition tolerance but at the expense of read-consistency of the replicas.

Cassandra provides two basic strategies to determine which nodes should hold replicas for each token. Each strategy is provided both the logical topology (token ring ordering) and physical layout (IP addresses) of the cluster. For each token in the ring, the replication strategy returns N −1 alternate endpoints.

Rack unaware is the default strategy used by Cassandra and ignores the physical cluster topology.

This option begins at the primary node for a token and returns the endpoints of the next N −1 nodes in the token ring.

Rack aware is strategized to improve availability such that it allows the system to continue operating in spite of a rack or whole data center being unavailable.

Nodes are assumed to be in different data centers if the second octet of their IPs differ, and in different racks if the third octet differs. This replication strategy attempts to find a single node in a separate data center, another on a different rack within the same data center, and finds the remaining N − 1 endpoints using the rack unaware strategy.

It is also possible for the users to implement their own replica placement strategy to meet the specific needs of the system.

The ability to tailor replica placement is an important part of architecting a sufficiently large Cassandra cluster because, if not properly tailored, given replica strategy may infect the cluster with unwanted hotspots of activity.

2.4.3 Consistency

Cassandra allows the user to make tradeoffs between consistency and latency by requiring them to specify desired consistency level i.e. ZERO, ONE, QUORUM, ALL, or ANY with each read or write operation.

Lowering consistency requirements means reduced latency and that, the read and write services remain more highly available in the event of a network partition.

ZERO consistency level indicates that a write should be processed completely asynchronously to the client. This offers the lowest possible latency but gives no consistency guarantee. This mode must be used when the write operation can happen at most once and consistency is unimportant.

ONE consistency level indicates that the write request will return only when at least one server where the key is stored has written the new data to its commit log. If no member of the replica group for applicable token is available, the write fails. Even if the server crashes immediately following this operation, the new data is guaranteed to eventually turn up for all reads after being brought back online.

ALL consistency level means, for a write to succeed, all the replicas must be updated durably.

QUORUM requires that N/2 + 1 server must have durable copies where N is the number of replicas.

In **ANY** write consistency, nodes perform hinted handoff to provide for even higher availability at the expense of consistency. When a write request is sent to a node in the cluster, if that node isn't responsible for the key of the write, it will maintain a hint and transparently proxy the request to a replica for that token and asynchronously ensure that the write eventually gets to a correct replica node. Thus, Hinted handoff allows writes to succeed without blocking the client pending handoff to a replica. In the other synchronous write consistency modes ONE, QUORUM and ALL, writes must be committed to durable storage at a node responsible for managing that key.

Reads require coordination among the same number of replicas but have some unique properties. Consistency modes of ZERO and ANY are special and only apply to writes. The consistency levels ONE, QUORUM, ALL, indicate the number of replicas that must be consulted during a read. In all

cases, if any replicas are in conflict the most recent is returned to the client. In addition, any copies that are in conflict are repaired at the time of the read. This is known in Cassandra as read-repair. Whether this read-repair happens synchronously with the caller or asynchronously depends on the stringency of the consistency level specified. If the specified number of replicas cannot be contacted, the read fails.
The following section explains how to balance between consistency and the combination of latency and fault-tolerance. To achieve the lowest latency operations, the most lenient consistency levels may be chosen for reads and writes. If R is given as the number of replicas consulted during a read and W is given as the number consulted during a write, Cassandra can be made fully consistent under the following condition.
R +W > N

QUORUM on reads and writes meets that requirement and is a common starting position which provides consistency without inhibiting performance or fault-tolerance. If lower latency is required, one may choose R +W <= N.
Cassandra does not support the notion of a dynamic quorum in which new quorum criteria are selected when the cluster is partitioned into two or more separate parts unable to communicate with one another. If a partition occurs that prevents the specified number of replicas from being consulted for either a read or write operation, that operation will fail until the partition is repaired.

2.4.4 Isolation and Atomicity

In Cassandra, isolation from other clients, who are working on the same data cannot be achieved. Cassandra guarantees atomicity within a Column Family; so for all columns of a row. There is no notion of a check-and-set operation that executes atomically. Batch updates for multiple keys within a column family are not guaranteed atomicity. Application designers choosing a data store should consider these criteria carefully against their requirements while selecting a data store. In some cases, applications designers may choose to put some subset of data that should be subject to ACID guarantees in a transactional relational database while some other data resides in a data store like Cassandra.

2.4.5 Elastic Storage

Cassandra clusters scale through the addition of new servers rather than requiring the purchase of more powerful servers. This is often referred to as horizontal versus vertical scaling. The process of introducing a new node into a Cassandra cluster is referred to as bootstrapping and is usually accomplished in one of two ways:
1) Configure the node to bootstrap itself to a particular token which dictates its placement within the ring. When a token is chosen specifically, some data from the node with the next highest token will begin migration to this node. Figure 2 shows a new node t_n' being bootstrapped to a token between t_{n-1} and t_n.

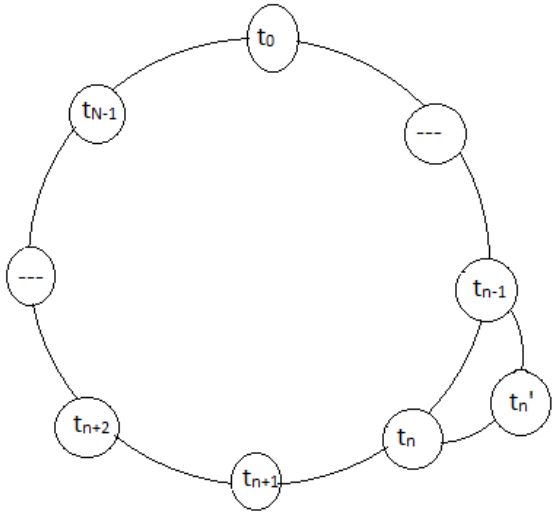

Figure 2: Node t_n' during bootstrap into token ring

The approximate fraction of data that will be migrated from node t_n to t_n' can be calculated as follows:
$(t_n' - t_{n-1}) / (t_n - t_{n-1})$
This calculation can be used in selecting a token value that achieves specific load balancing goals within the cluster. For example if a particular node has limited storage or processing capability, it is assigned responsibility for a smaller slice of the token range.
2) The cluster should select a token dictating the new nodes placement in the ring. The objective behind this is to select a token which would make new node responsible for approximately half of the data on the node with maximum data, which does not have some other node bootstrapping in its range.
To initiate election, the nodes are not contacted in an ad-hoc way. Rather, the new node unilaterally makes this decision depending on storage load data gossiped from other nodes periodically.

2.5 Client Access

Client interaction with Cassandra can be challenging. It is important for the client to know which host it should send request to. Although every node is capable of responding to client request, choosing the correct node having replica of the data to be read or

written will result in reduced communication overhead between nodes to coordinate response. Also, if all the requests are routed to a single node, it may result in a bottleneck. Furthermore, if the node fails or is unreachable, it may perceive to be entire cluster being unavailable.

The Cassandra data store is implemented in Java. However there is no native Java API for communicating with Cassandra from a separate address space. Instead Cassandra implements services using Thrift. Thrift is a framework that includes a high level grammar for defining services, remote objects and types, and a code generator that produces client and server RMI stubs in a variety of languages. Each Cassandra node starts a Thrift server exposing services for interacting with data and introspecting information about the cluster.

The Thrift API to Cassandra leaves open the possibility of a perceived single point of failure and does not intelligently route service invocations to replica nodes.

On the other hand, Hector is a native Java client for Cassandra which has begun to tackle the challenges associated with accessing this decentralized system. Hector is actually a layer on top of the Thrift API and as such it depends on its client and server side bindings. It introspect data about the state of the ring in order to determine the endpoint for each host. This information is then used to implement three modes of client-level failover.

FAIL FAST implements classic behavior of failing; fails if a request of the first node contacted is down.

ON FAIL TRY ONE NEXT AVAILABLE will attempt to contact one more node in the ring before failing.

ON FAIL TRY ALL AVAILABLE will continue to contact nodes, up to all in the cluster, before failing.

By this, it is difficult to find client which can intelligently route requests to replica nodes.

Cassandra's Thrift API would require knowledge of the replica placement strategy to identify nodes where replicas are located, although it supports introspection of the token range for each ring.

A truly intelligent client that requires up-to-date information regarding replica locations, the availability status of nodes in the cluster and token ranges would need to become a receiver of at least a subset of the gossip passed between nodes and thus an actual member of the cluster.

3. Conclusions

This paper provides an architectural overview of Cassandra and discusses how the distributed system design and principles are applied to real systems. The concept of distributed hash tables has also been discussed. We have also seen how the data can be distributed across wide area networks with replication to increase availability in the event of node failure and network partitioning.

4. References

[1**]** F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, *Bigtable: A Distributed Storage System for Structured Data* OSDI'06: Seventh Symposium on Operating System Design and Implementation, 2006, Seattle, WA, 2006.

[2] A. Lakshman, P. Malik, *Cassandra – A Decentralized Structured Storage System*, Cornell, 2009.

[3] Problems with CAP, and Yahoo's little known NoSQL system

http://dbmsmusings.blogspot.com/2010/04/problems-with-cap-and-yahoos-little.html

[4] Kang Wei, Tan Sicong, Xiao Qian, Hadi Amiri - *An Investigation of No-SQL Data Stores.*

[5]http://en.wikipedia.org/wiki/Apache_Cassandra

[6] http://wiki.apache.org/cassandra/API

[7] http://wiki.apache.org/thrift/

[8]http://en.wikipedia.org/wiki/Distributed_hash_table

[9] http://en.wikipedia.org/wiki/NoSQL

[10]http://dfeatherston.com/cassandra-adf-uiuc-su10.pdf