Lab 1. Bird's Eye View of Cassandra

Imagine that we have turned back the clock to the 1990s and you an application architect. Whenever you were required to select a suitable database technology for your applications, what kind of database technology would you choose? I bet 95 percent (or more) of the time you would select relational databases.

Relational databases have been the most dominating data management solution since the 1970s. At that time, the application system was usually silo. The users of the application and their usage patterns were known and under control. The workload that had to be catered for by the relational database could be determined and estimated. Apart from the workload consideration, the data model can also be structured in normalized forms as recommended by the relational theory. Moreover, relational databases provide many benefits such as support of transactions, data consistency, and isolation. Relational databases just fit perfectly for the purposes. Therefore, it is not difficult to understand why the relational database has been so popular and why it is the de facto standard for persistent data stores in application development.

Nonetheless, with the proliferation of the Internet and the numerous web applications running on it, the control of the users and their usage patterns (hence the scale), the workload generated, and the flexibility of the data model were gone. Typical examples of these web applications were global e-commerce websites, social media sites, video community websites, and so on. They generated a tremendous amount of data in a very short period of time. It should also be noted that the data generated by these applications were not only structured, but also semi-structured and even unstructured. Since relational databases were the de facto standard at that time, developers and architects did not have many alternatives but were forced to tweak them to support these web applications, even though they knew that relational databases were suboptimal and had many limitations. It became apparent that a different kind of enabling technology should be found to break through the challenges.

We are in an era of information explosion, as a result of the ever-increasing amount of user-generated data and content on the Web and mobile applications. The generated data is not only large in volume and fast in velocity but it is also diversified in variety. Such rapidly growing data of different varieties is often termed as **Big Data**.

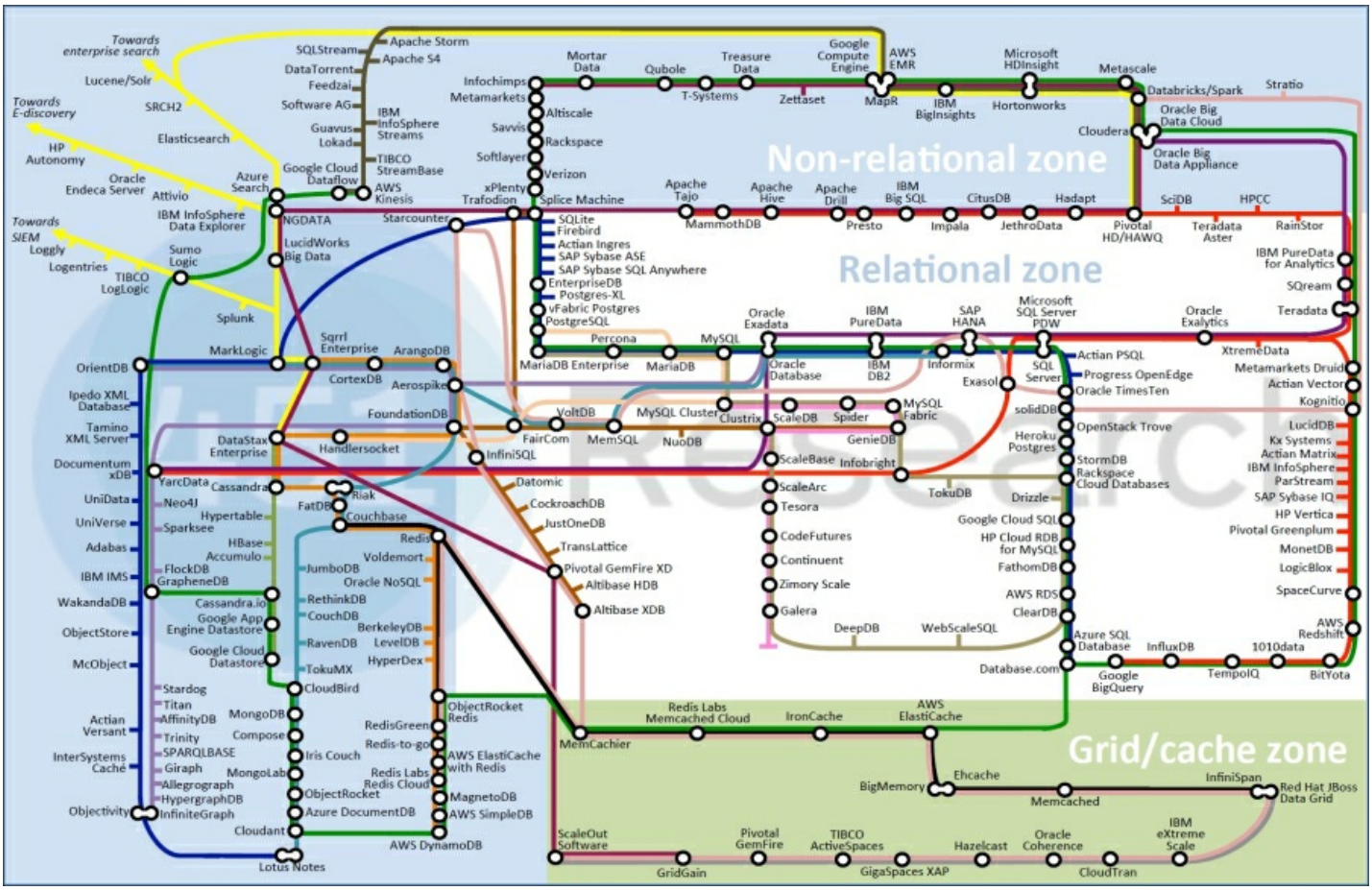
No one has a clear, formal definition of Big Data. People, however, unanimously agree that the most fundamental characteristics of Big Data are related to large volume, high velocity, and great variety. Big Data imposes real, new challenges to the information systems that have adopted traditional ways of handling data. These systems are not designed for web-scale and for being enhanced to do so, cost effectively. Due to this, you might find yourself asking whether or not we have any alternatives.

Challenges come with opportunities on the flip side. A new breed of data management products was born. The most recent answer to the question in the last paragraph is NoSQL.

What is NoSQL?

The need to tackle the Big Data challenges has led to the emergence of new data management technologies and techniques. Such technologies and techniques are rather different from the ubiquitous relational database technology that has been used for over 40 years. They are collectively known as **NoSQL**.

NoSQL is an umbrella term for the data stores that are not based on the relational data model. It encompasses a great variety of many different database technologies and products. As shown in the following figure, The **Data Platforms Landscape Map**, there are over 150 different database products that belong to the non-relational school as mentioned in http://nosql-database.org/. Cassandra is one of the most popular ones. Other popular NoSQL database products are, just to name a few, MongoDB, Riak, Redis, Neo4j, so on and so forth.



::: The Data Platforms Landscape Map (Source: 451 Research) :::

So, what kinds of benefits are provided by NoSQL? When compared to the relational database, NoSQL overcomes the weaknesses that the relational data model does not address well, which are as follows:

- Huge volume of structured, semi-structured, and unstructured data

- Flexible data model (schema) that is easy to change
- Scalability and performance for web-scale applications
- Lower cost
- Impedance mismatch between the relational data model and object-oriented programming
- Built-in replication
- Support for agile software development

Note

Limitations of NoSQL Databases

Many NoSQL databases do not support transactions. They use replication extensively so that the data in the cluster might be momentarily inconsistent (although it is eventually consistent). In addition, the range queries are not available in NoSQL databases. Furthermore, a flexible schema might lead to problems with efficient searches.

The huge volume of structured, semi-structured, and unstructured data was mentioned earlier. What I want to dive deeper into here is that different NoSQL databases provide different solutions for each of them. The primary factor to be considered is the NoSQL database type, which will be introduced in the subsequent section.

All NoSQL databases provide a flexible data model that is easy to change and some might be even schemaless. In a relational database, the relational data model is called schema. You need to understand the data to be stored in a relational database, design the data model according to the relational database theory, and define the schema upfront in the relational database before you can actually store data inside it. It is a very structured approach for structured data. It is a prescriptive data modeling process. It is absolutely fine if the data model is stable, because there are not many changes required. But what if the data model keeps changing in the future and you do not know what needs to be changed? You cannot prescribe comprehensively in advance. It leads to many inevitable remedies; say, data patching for example, to change the schema.

Conversely, in NoSQL databases, you need not prescribe comprehensively. You only need to describe what is to be stored. You are not bound by the relational database theory. You are allowed to change the data model whenever necessary. The data model is schemaless and is a living object. It evolves as life goes on. It is a descriptive data modeling process.

Scalability and performance for web-scale applications refer to the ability of the system to be scaled, preferably horizontally, to support web-scale workloads without considerably deteriorating system performance. Relational databases can only be scaled out to form a cluster consisting of a very small number of nodes. It implies the rather low ceiling imposed on these web-scale applications using relational databases. In addition, changing the schema in a clustered relational database is a big task of high complexity. The processing power required to do this is so significant that the system performance cannot be unaffected. Most NoSQL databases were created to serve web-scale applications. They natively support horizontal scaling without very little degrade on the performance.

Now let us talk about money. Traditionally, most high-end relational databases are commercial products that demand their users to pay huge software license fees. Besides, to run these high-end relational databases, the underlying hardware servers are usually high-end as well. The result is that the hardware and software costs of running a powerful relational database are exceptionally large. In contrast, NoSQL databases are open source and community-driven in a majority, meaning that you need to pay the software license cost, which is an order of magnitude less than other databases. NoSQL databases are able to run on commodity machines that will lead to a possible churn, or crashes. Therefore, the machines are usually configured to be a cluster. High-end hardware servers are not needed and so the hardware cost is tremendously reduced. It should be noted that when NoSQL databases are put into production, some cost of the support is still required but it is definitely much less when compared to that of commercial products.

There exists a generation gap between the relational data model and object-oriented programming. The relational data model was the product of 1970s, whereas object-oriented programming became very popular in 1990s. The root cause, known as impedance mismatch, is an inherent difficulty of representing a record or a table in a relational data model with the object-oriented model. Although there are resolutions for this difficulty, most application developers still feel very frustrated to bring the two together.

Note

Impedance Mismatch

Impedance mismatch is the difference between the relational model and the in-memory data structures that are usually encountered in object-oriented programming languages.

Built-in replication is a feature that most NoSQL databases provide to support high availability in a cluster of many nodes. It is usually automatic and transparent to the application developers. Such a feature is also available in relational databases, but the database administrators must struggle to configure, manage, and operate it by themselves.

Finally, relational databases do not support agile software development very well. Agile software development is iterative by nature. The software architecture and data model emerge and evolve as the project proceeds in order to deliver the product incrementally. Hence, it is conceivable that the need of changing the data model to meet the new requirements is inevitably frequent. Relational databases are structured and do not like changes. NoSQL can provide such flexibility for agile software development teams by virtue of its schemaless characteristic. Even better, NoSQL databases usually allow the changes to be implemented in real time without any downtime.

NoSQL Database types

Now you know the benefits of NoSQL databases, but the products that fall under the NoSQL databases umbrella are quite varied. How can you select the right one for yourself among so many NoSQL databases? The selection criteria of which NoSQL database fits your needs is really dependent on the use cases at hand. The most important factor to consider here is the NoSQL database type, which can be subdivided into four main categories:

- Key/value pair store
- Column-family store
- Document-based repository
- Graph database :::

The NoSQL database type dictates the data model that you can use. It is beneficial to understand each of them deeper.

Key/value pair store

Key/value pair is the simplest NoSQL database type. Key/value store is similar to the concept of Windows registry, or in Java or C#, a map, a hash, a key/value pair. Each data item is represented as an attribute name, also a key, together with its value. It is also the basic unit stored in the database. Examples of the NoSQL databases of key/value pair type are **Amazon Dynamo**, **Berkeley DB**, **Voldemort** and **Riak**.

Internally, key/value pairs are stored in a data structure called **hashmap**. Hashmap is popular because it provides very good performance on accessing data. The key of a key/value pair is unique and can be searched very quickly.

Key/value pair can be stored and distributed in the disk storage as well as in memory. When used in memory, it can be used as a cache, which depends on the caching algorithm, can considerably reduce disk I/O and hence boost up the performance significantly.

On the flip side, key/value pair has some drawbacks, such as lack of support of range queries, no way to operate on multiple keys simultaneously, and possible issues with load balancing.

Column-family store

A column in this context is not equal to a column in a relational table. In the NoSQL world, a column is a data structure that contains a key, value, and timestamp. Thus, it can be regarded as a combination of key/value pair and a timestamp. Examples are **Google BigTable**, **Apache Cassandra**, and **Apache HBase**. They provide optimized performance for queries over very large datasets.

Column-family store is basically a multi-dimensional map. It stores columns of data together as a row, which is associated with a row key. This contrasts with rows of data in a relational database. Column-family store does not need to store null columns, as in the case of a relational database and so it consumes much less disk space. Moreover, columns are not bound by a rigid schema and you are not required to define the schema upfront.

The key component of a column is usually called the primary key or the row key. Columns are stored in a sorted manner by the row key. All the data belonging to a row key is stored together. As such, read and write operations of the data can be confined to a local node, avoiding unnecessary inter-node network traffic in a cluster. This mechanism makes the data lookup and retrieval extremely efficient.

Obviously, a column-family store is not the best solution for systems that require ACID transactions and it lacks the support for aggregate queries provided by relational databases such as `SUM()`.

Document-based repository

Document-based repository is designed for documents or semi-structured data. The basic unit of a document-based repository associates each key, a primary identifier, with a complex data structure called a document. A document can contain many different key-value pairs, or key-array pairs, or even nested documents. Therefore, document-based repository does not adhere to a schema. Examples are **MongoDB** and **CouchDB**.

In practice, a document is usually a loosely structured set of key/value pairs in the form of **JavaScript Object Notation** (**JSON**). Document-based repository manages a document as a whole and avoids breaking up a document into fragments of key/value pairs. It also allows document properties to be associated with a document.

As a document does not adhere to a fixed schema, the search performance is not guaranteed. There are generally two approaches to query a document database. The first is to use materialized views (such as CouchDB) that are prepared in advance. The second is to use indexes defined on the document values (such as MongoDB) that behave in the same way as a relational database index.

Graph database

Graph databases are designed for storing information about networks, such as a social network. A graph is used to represent the highly connected network that is composed of nodes and their relationships. The nodes and relationships can have individual properties. The prominent graph databases include **Neo4J** and **FlockDB**.

Owing to the unique characteristics of a graph, graph databases commonly provide APIs for rapid traversal of graphs.

Graph databases are particularly difficult to be scaled out with sharding because traversing a graph of the nodes on different machine does not provide a very good performance. It is also not a straightforward operation to update all or a subset of the nodes at the same time.

So far, you have grasped the fundamentals of the NoSQL family. Since this course concentrates on Apache Cassandra and its data model, you need to know what Cassandra is and have a basic understanding of what its architecture is, so that you can select and leverage the best available options when you are designing your NoSQL data model and application.

What is Cassandra?

Cassandra can be simply described in a single phrase: a massively scalable, highly available open source NoSQL database that is based on peer-to-peer architecture.

Cassandra is now 5 years old. It is an active open source project in the Apache Software Foundation and therefore it is known as Apache Cassandra as well. Cassandra can manage huge volume of structured, semi-structured, and unstructured data in a large distributed cluster across multiple data centers. It provides linear scalability, high performance, fault tolerance, and supports a very flexible data model.

Note

Netflix and Cassandra

One very famous case study of Cassandra is Netflix's move to replace their Oracle SQL database to Cassandra running on cloud. As of March 2013, Netflix's Cassandra deployment consists of 50 clusters with over 750 nodes. For more information, please visit the case study at http://www.datastax.com/wp-content/uploads/2011/09/CS-Netflix.pdf.

In fact, many of the benefits that Cassandra provides are inherited from its two best-of-breed NoSQL parents, Google BigTable and Amazon Dynamo. Before we go into the details of Cassandra's architecture, let us walk through each of them first.

Google BigTable

Google BigTable is Google's core technology, particularly addressing data persistence and management on web-scale. It runs the data stores for many Google applications, such as Gmail, YouTube, and Google Analytics. It was designed to be a web-scale data store without sacrificing real-time responses. It has superb read and write performance, linear scalability, and continuous availability.

Google BigTable is a sparse, distributed, persistent, multidimensional sorted map. The map is indexed by a row key.

Despite the many benefits Google BigTable provides, the underlying design concept is really simple and elegant. It uses a persistent commitlog for every data write

request that it receives and then writes the data into a memory store (acting as a cache). At regular intervals or when triggered by a particular event, the memory store is flushed to persistent disk storage by a background process. This persistent disk storage is called **Sorted String Table**, or **SSTable**. The SSTable is immutable meaning that once it has been written to a disk, it will never be changed again. The word *sorted* means that the data inside the SSTable is indexed and sorted and hence the data can be found very quickly. Since the write operation is log-based and memory-based, it does not involve any read operation, and therefore the write operation can be extremely fast. If a failure happens, the commitlog can be used to replay the sequence of the write operations to merge the data that persists in the SSTables.

Read operation is also very efficient by looking up the data in the memory store and the indexed SSTables, which are then merged to return the data.

All the above-mentioned Google BigTable brilliances do come with a price. Because Google BigTable is distributed in nature, it is constrained by the famous *CAP theorem*, stating the relationship among the three characteristics of a distributed system, namely Consistency, Availability, and Partition-tolerance. In a nutshell, Google BigTable prefers Consistency and Partition-tolerance to Availability.

Note

The CAP theorem

CAP is an acronym of the three characteristics of a distributed system: Consistency, Availability, and Partition-tolerance. Consistency means that all the nodes in a cluster see the same data at any point in time. Availability means that every request that is received by a non-failing node in the cluster must result in a response. Partition-tolerance means that a node can still function when communication with other groups of nodes is lost. Originating from Eric A. Brewer, the theorem states that in a distributed system, only two out of the three characteristics can be attained at the most.

Google BigTable has trouble with Availability while keeping Consistency across partitioned nodes when failures happen in the cluster.

Amazon Dynamo

Amazon Dynamo is a proprietary key-value store developed by Amazon. It is designed for high performance, high availability, and continuous growth of data of huge volume. It is the distributed, highly available, fault-tolerant skeleton for Amazon. Dynamo is a peer-to-peer design meaning that each node is a peer and no one is a master who manages the data.

Dynamo uses data replication and auto-sharding across multiple nodes of the cluster. Imagine that a Dynamo cluster consists of many nodes. Every write operation in a node is replicated to two other nodes. Thus, there are three copies of data inside the cluster. If one of the nodes fails for whatever reason, there are still two copies of data that can be retrieved. Auto-sharding ensures that the data is partitioned across the cluster.

Note

Auto-sharding

NoSQL database products usually support auto-sharding so that they can natively and automatically distribute data across the database cluster. Data and workload are automatically balanced across the nodes in the cluster. When a node fails for whatever reason, the failed node can be quickly and transparently replaced without service interruptions.

Dynamo focuses primarily on the high availability of a cluster and the most important idea is eventual consistency. While considering the CAP Theorem, Dynamo prefers Partition-tolerance and Availability to Consistency. Dynamo introduces a mechanism called **Eventual Consistency** to support consistency. Temporary inconsistency might occur in the cluster at a point in time, but eventually all the nodes will receive the latest consistent updates. Given a sufficiently long period of time without further changes, all the updates can be expected to propagate throughout the cluster and the replicas on all the nodes will be consistent eventually. In real life, an update takes only a fraction of a second to become eventually consistent. In other words, it is a trade-off between consistency and latency.

Note

Eventual consistency

Eventual consistency is not inconsistency. It is a weaker form of consistency than the typical Atomic-Consistency-Isolation-Durability (ACID) type consistency is found in the relational databases. It implies that there can be short intervals of inconsistency among the replicated nodes during which the data gets updated among these nodes. In other words, the replicas are updated asynchronously.

Cassandra's high-level architecture

Cassandra runs on a peer-to-peer architecture which means that all nodes in the cluster have equal responsibilities except that some of them are seed nodes for other non-seed nodes to obtain information about the cluster during startup. Each node holds a partition of the database. Cassandra provides automatic data distribution and replication across all nodes in the cluster. Parameters are provided to customize the distribution and replication behaviors. Once configured, these operations are processed in the background and are fully transparent to the application developers.

Cassandra is a column-family store and provides great schemaless flexibility to application developers. It is designed to manage huge volume of data in a large cluster without a single point of failure. As multiple copies of the same data (replicas) are replicated in the cluster, whenever one node fails for whatever reason, the other replicas are still available. Replication can be configured to meet the different physical cluster settings, including data center and rack locations.

Any node in the cluster can accept read or write requests from a client. The node that is connected to a client with a request serves as the coordinator of that particular request. The coordinator determines which nodes are responsible for holding the data for the request and acts as a proxy between the client and the nodes.

Cassandra borrows the commitlog mechanism from Google BigTable to ensure data durability. Whenever a write data request is received by a node, it is written into the commitlog. The data that is being updated is then written to a memory structure, known as memtable. When the memtable is full, the data inside the memtable is flushed to a disk storage structure, SSTable. The writes are automatically partitioned by the row key and replicated to the other nodes holding the same partition.

Cassandra provides linear scalability, which means that the performance and capacity of the cluster is proportional to the number of nodes in it.

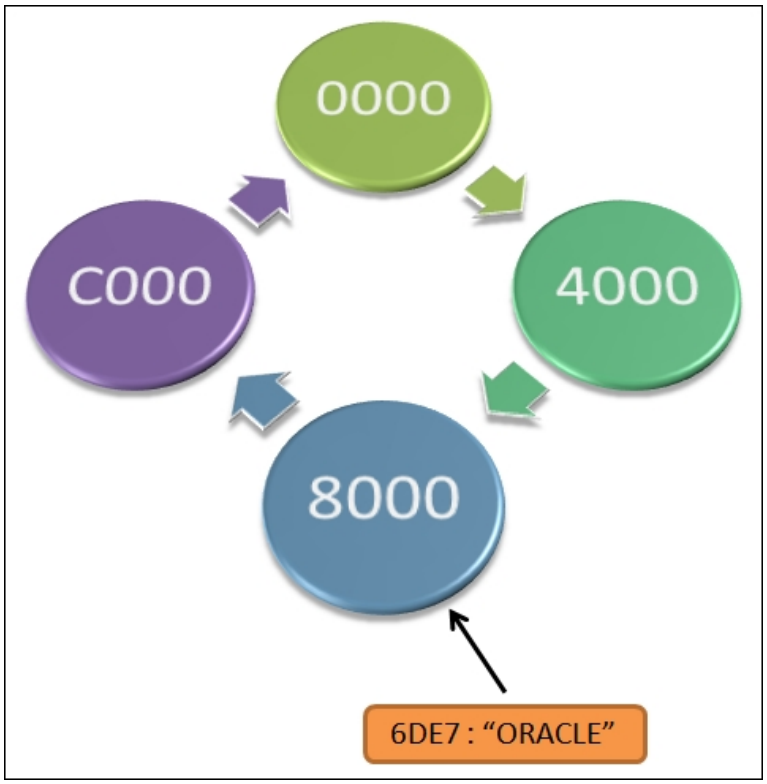
Partitioning

The ability to scale horizontally and incrementally is a Cassandra key design feature. To achieve this, Cassandra is required to dynamically partition the data over the set of nodes in the cluster.

A cluster is the outermost structure which is composed of nodes in Cassandra. It is also a container of keyspace. A keyspace in Cassandra is analogous to a schema in a relational database. Each Cassandra cluster has a system keyspace to keep system-wide metadata. It contains the replication settings which controls how the data is distributed and replicated in a cluster. Typically, one keyspace is assigned to one cluster but one cluster might contain more than one keyspace.

The smallest cluster in the theory contains a single node and a cluster of three or more nodes, which is much more practical. Each node holds a replica for the different range of data in partitions, and exchanges information across the cluster every second.

A client issues read or write requests to any node. The node that receives the request becomes a coordinator that acts as a proxy of the client to do the things as explained previously. Data is distributed across the cluster and the node addressing mechanism is called consistent hashing. Therefore, a cluster can be viewed as a ring of hash as each node in the cluster or the ring is assigned a single unique token so that each node is responsible for the data in the range from its assigned token to that of the previous node. For example, in the following figure, a cluster contains four nodes with unique tokens:



::: Cassandra's consistent hashing :::

Before Version 1.2, tokens were calculated and assigned manually and from Version 1.2 onwards, tokens can be generated automatically. Each row has a row key used by a partitioner to calculate its hash value. The hash value determines the node which stores the first replica of the row. The partitioner is just a hash function that is used for calculating a row key's hash value and it also affects how the data is distributed or balanced in the cluster. When a write occurs, the first replica of the row is always placed in the node with the key range of the token. For example, the hash value of a row key `ORACLE` is `6DE7` that falls in the range of 4,000 and 8,000 and so the row goes to the bottom node first. All the remaining replicas are distributed based on the replication strategy.

Note

Consistent hashing

Consistent hashing allows each node in the cluster to independently determine which nodes are replicas for a given row key. It just involves hashing the row key, and then compares that hash value to the token of each node in the cluster. If the hash value falls in between a node's token, and the token of the previous node in the ring (tokens are assigned to nodes in a clockwise direction), that node is the replica for that row.

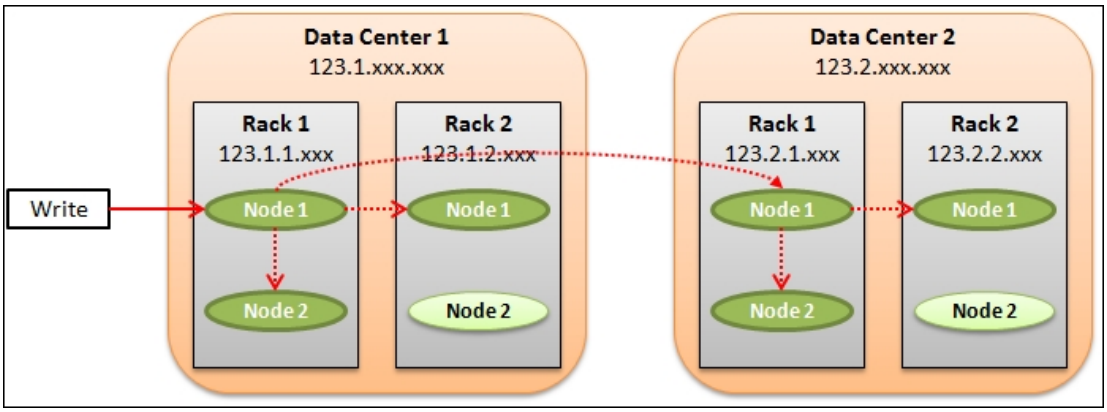
Replication

Cassandra uses replication to attain high availability and data durability. Each data is replicated at a number of nodes that are configured by a parameter called replication factor. The coordinator commands the replication of the data within its range. It replicates the data to the other nodes in the ring. Cassandra provides the client with various configurable options to see how the data is to be replicated, which is called replication strategy.

Replication strategy is the method of determining which nodes the replicas are placed in. It provides many options, such as rack-aware, rack-unaware, network-topology-aware, so on and so forth.

Snitch

A snitch determines which data centers and racks to go for in order to make Cassandra aware of the network topology for routing the requests efficiently. It affects how the replicas can be distributed while considering the physical setting of the data centers and racks. The node location can be determined by the rack and data center with reference to the node's IP address. An example of a cluster across two data centers is shown in the following figure, in order to illustrate the relationship among replication factor, replication strategy, and snitch in a better way:



::: Multiple data center cluster :::

Each data center has two racks and each rack contains two nodes respectively. The replication factor per data center is set to three here. With two data centers, there are six replicas in total. The node location that addresses the data center and rack locations are subject to the convention of IP address assignment of the nodes.

Seed node

Some nodes in a Cassandra cluster are designated as seed nodes for the others. They are configured to be the first nodes to start in the cluster. They also facilitate the bootstrapping process for the new nodes joining the cluster. When a new node comes online, it will talk to the seed node to obtain information about the other nodes in the cluster. The talking mechanism is called **gossip**. If a cluster is across multiple data centers, the best practice is to have more than one seed node per data center.

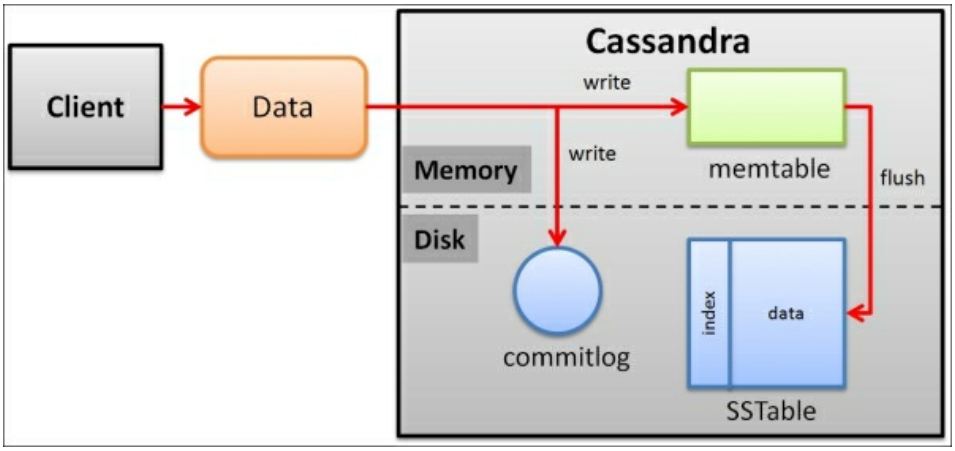
Gossip and Failure detection

Nodes need to communicate periodically (every second) to exchange state information (for example, dead or alive), about themselves and about other nodes they know about. Cassandra uses a gossip communication protocol to disseminate the state information, which is also known as epidemic protocol. It is a peer-to-peer communication protocol that provides a decentralized, periodic, and an automatic way for the nodes in the cluster to exchange the state information about themselves, and about other nodes they know about with up to three other nodes. Therefore, all nodes can quickly learn about all the other nodes in the cluster. Gossip information is also persisted locally by each node to allow fast restart.

Cassandra uses a very efficient algorithm, called *Phi Accrual Failure Detection Algorithm*, to detect the failure of a node. The idea of the algorithm is that the failure detection is not represented by a Boolean value stating whether a node is up or down. Instead, the algorithm outputs a value on the continuous suspicion level between dead and alive, on how confident it is that the node has failed. In a distributed environment, false negatives might happen due to the network performance, fluctuating workload, and other conditions. The algorithm takes all these factors into account and provides a probabilistic value. If a node has failed, the other nodes periodically try to gossip with it to see if it comes back online. A node can then determine locally from the gossip state and its history and adjust routes accordingly.

Write path

The following figure depicts the components and their sequence of executions that form a write path:



::: Cassandra write path :::

When a write occurs, the data will be immediately appended to the commitlog on the disk to ensure write durability. Then Cassandra stores the data in memtable, an in-memory store of hot and fresh data. When memtable is full, the memtable data will be flushed to a disk file, called SSTable, using sequential I/O and so random I/O is avoided. This is the reason why the write performance is so high. The commitlog is purged after the flush.

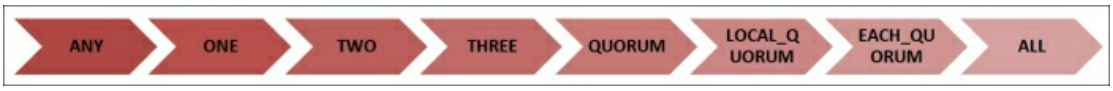
Due to the intentional adoption of sequential I/O, a row is typically stored across many SSTable files. Apart from its data, SSTable also has a primary index and a *bloom filter*. A primary index is a list of row keys and the start position of rows in the data file.

Note

Bloom filter

Bloom filter is a sample subset of the primary index with very fast nondeterministic algorithms to check whether an element is a member of a set. It is used to boost the performance.

For write operations, Cassandra supports tunable consistency by various write consistency levels. The write consistency level is the number of replicas that acknowledge a successful write. It is tunable on a spectrum of write consistency levels, as shown in the following figure:



::: Cassandra write consistency levels :::

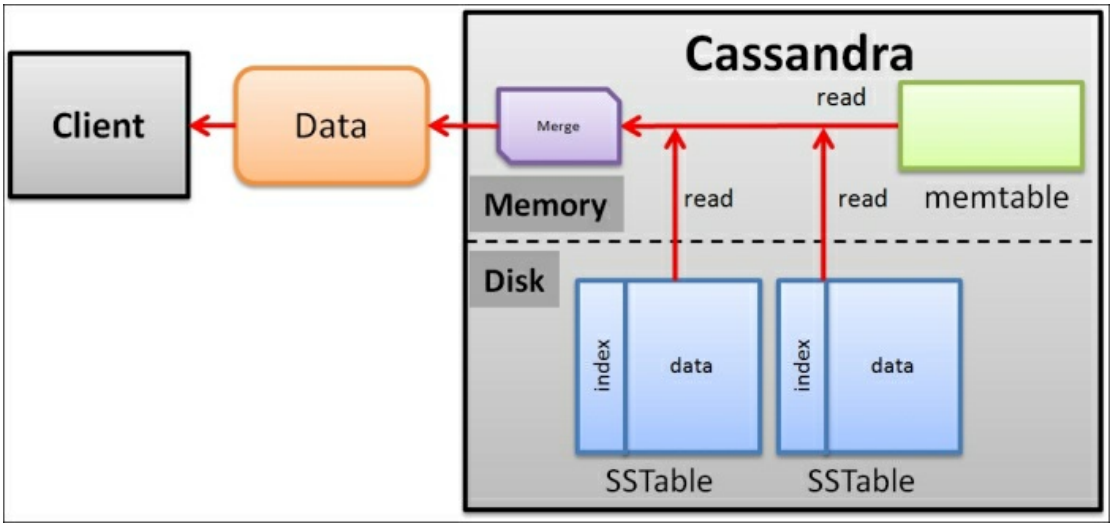
The following describes the terms in the figure:

- **ANY**: This is the lowest consistency (but highest availability)
- **ALL**: This is the highest consistency (but lowest availability)
- **ONE**: This gives at least one replica
- **TWO**: This gives at least two replicas
- **THREE**: This gives at least three replicas
- **QUORUM**: This ensures strong consistency by tolerating some level of failure, which is determined by *(replication_factor / 2) + 1* (rounded down to the nearest integer)
- **LOCAL_QUORUM**: This is for multi-data center and rack-aware without inter-data center traffic
- **EACH_QUORUM**: This is for multi-data center and rack-aware :::

The two extremes are the leftmost **ANY** which means weak consistency and the rightmost **ALL** means strong consistency. A consistency level of **THREE** is very common in practice. **QUORUM** can be chosen to be an optimum value, as calculated by the given formula. Here, the replication factor is the number of replicas of data on multiple nodes. Both **LOCAL QUORUM** and **EACH QUORUM** support multiple data centers and rack-aware write consistency with a slight difference as shown earlier.

Read path

On the flip side, the following figure shows the components and their sequence of executions that form a read path:



::: Cassandra read path :::

When a read request comes in to a node, the data to be returned is merged from all the related SSTables and any unflushed memtables. Timestamps are used to determine which one is up-to-date. The merged value is also stored in a write-through row cache to improve the future read performance.

Similar to the write consistency levels, Cassandra also provides tunable read consistency levels, as shown in the following figure:

::: Cassandra read consistency levels :::

The following describes the terms in the figure:

- **ALL**: This is the highest consistency (but lowest availability)
- **ONE**: This gives at least one replica
- **TWO**: This gives at least two replicas
- **THREE**: This gives at least three replicas
- **QUORUM**: This ensures strong consistency by tolerating some level of failure, which is determined by *(replication_factor / 2) + 1* (rounded down to the nearest integer)
- **LOCAL_QUORUM**: This is for multi-data center and rack-aware without inter-data center traffic
- **EACH_QUORUM**: This is for multi-data center and rack-aware :::

Read consistency level is the number of replicas contacted for a successful, consistent read, almost identical to write consistency levels, except that **ANY** is not an option here.

Repair mechanism

There are three built-in repair mechanisms provided by Cassandra:

- Read repair
- Hinted handoff
- Anti-entropy node repair :::

During a read, the coordinator that is just the node connects and services the client, contacts a number of nodes as specified by the consistency level for data and the fastest replicas will return the data for a consistency check by in-memory comparison. As it is not a dedicated node, Cassandra lacks a single point of failure. It also checks all the remaining replicas in the background. If a replica is found to be inconsistent, the coordinator will issue an update to bring back the consistency. This mechanism is called **read repair**.

Hinted handoff aims at reducing the time to restore a failed node when rejoining the cluster. It ensures absolute write availability by sacrificing a bit of read consistency. If a replica is down at the time a write occurs, another healthy replica stores a hint. Even worse, if all the relevant replicas are down, the coordinator stores the hint locally. The hint basically contains the location of the failed replica, the affected row key, and the actual data that is being written. When a node responsible for the token range is up again, the hint will be handed off to resume the write. As such, the update cannot be read before a complete handoff, leading to inconsistent reads.

Another repair mechanism is called **anti-entropy** which is a replica synchronization mechanism to ensure up-to-date data on all nodes and is run by the administrators manually.

Features of Cassandra

In order to keep this lab short, the following bullet list covers the great features provided by Cassandra:

- Written in Java and hence providing native Java support
- Blend of Google BigTable and Amazon Dynamo
- Flexible schemaless column-family data model
- Support for structured and unstructured data
- Decentralized, distributed peer-to-peer architecture
- Multi-data center and rack-aware data replication
- Location transparent
- Cloud enabled
- Fault-tolerant with no single point of failure
- An automatic and transparent failover
- Elastic, massively, and linearly scalable
- Online node addition or removal
- High Performance
- Built-in data compression
- Built-in caching layer
- Write-optimized
- Tunable consistency providing choices from very strong consistency to different levels of eventual consistency
- Provision of **Cassandra Query Language** (**CQL**), a SQL-like language imitating `INSERT`, `UPDATE`, `DELETE`, `SELECT` syntax of SQL
- Open source and community-driven

Summary

In this lab, we have gone through a bit of history starting from the 1970s. We were in total control of the data models that were rather stable and the applications that were pretty simple. The relational databases were a perfect fit in the old days. With the emergence of object-oriented programming and the explosion of the web applications on the pervasive Internet, the nature of the data has been extended from structured to semi-structured and unstructured. Also, the application has become more complex. The relational databases could not be perfect again. The concept of Big Data was created to describe such challenges and NoSQL databases provide an alternative resolution to the relational databases.

NoSQL databases are of a wide variety. They provide some common benefits and can be classified by the NoSQL database type. Apache Cassandra is one of the NoSQL databases that is a blend of Google BigTable and Amazon Dynamo. The elegance of its architecture inherits from the DNA of these two parents.

In the next lab, we will look at the flexible data model supported by Cassandra.