Apache Cassandra™ Architecture

Inside DataStax Distribution of Apache Cassandra™

DATASTAX®

TABLE OF CONTENTS

INTRODUCTION

Fueled by the internet revolution, mobile devices, and ecommerce, modern applications have outgrown relational databases. Out of necessity, a new generation of databases has emerged to address large-scale, globally distributed data management challenges. In this paper, we discuss the motivations for these changes and provide an architectural and technical overview of DataStax Distribution of Apache Cassandra™, a commercially supported distribution of the highly successful Cassandra database. Cassandra powers online services and mobile backends for some of the world's most recognizable brands, including Apple, Netflix, and Facebook.

MOTIVATIONS FOR CASSANDRA

Dramatic changes in data management

Relational database management systems (RDBMS) have long been used for everything from managing financial transactions to manufacturing systems to employee data. Popular relational databases include Oracle®, IBM® DB2®, and Microsoft® SQL Server, as well as open source solutions such as MySQL® and PostgreSQL®. Generations of computer science students have been educated in relational databases, and Structured Query Language (SQL) has become a standard way of accessing and manipulating data.

Despite the success of relational databases, they've proven difficult to scale to meet the challenges of modern large-scale applications. Internet and mobile applications frequently serve millions of customers and manage hundreds of terabytes of data. For internet-scale applications, capacity alone is not enough; companies also need applications to be continuously available, lightning fast, and globally accessible.

In an increasingly connected world, it's not uncommon for applications to collect and store hundreds of terabytes or even petabytes of data. A popular definition of big data refers to data that is "too large or complex to be dealt with using conventional means." By the early 2000s, data volumes were already becoming too large to store on a single machine. Attempts were made to scale RDBMS solutions by splitting databases across multiple hosts, but inevitably relational models ran into problems. Simultaneously addressing scale, availability, performance, and strong consistency proved to be too difficult, and compromises had to be made. The race was on to find better solutions.

NoSQL databases

Diverse data management challenges resulted in a variety of solutions. Examples include various key-value stores (Redis and Memcached), column-oriented databases (Bigtable and HBase®), document databases (MongoDB™ and CouchDB®) and graph databases (Neo4j and DataStax Enterprise Graph).

In 2009 the term “NoSQL” started to appear. While NoSQL originally referred to non-SQL databases, many of these solutions began to support SQL-like query languages to make them more accessible to users. Over time, the term has evolved to mean “not only SQL.”

About Cassandra

Cassandra was first developed at Facebook by Avinash Lakshman and Prashant Malik to support inbox searching. Facebook open sourced Cassandra in 2008 as an Apache incubator project, and it was accepted as a top-level Apache project in 2010. The design of Cassandra was influenced by both Amazon’s Dynamo—a highly available key store and a predecessor to Amazon’s DynamoDB that was outlined in a research paper[i]— and Google’s Bigtable.[ii]

Among the goals of Dynamo was to build a reliable globally distributed database to support Amazon’s growing ecommerce business. Accordingly, Cassandra was designed from the ground up for efficient distributed operation across multiple data centers and elastic scaling with exceptionally high performance and reliability.

WHERE CASSANDRA EXCELS

While Cassandra can be used for a variety of applications, it excels in the following areas:

- **Large-scale storage:** Cassandra scales to hundreds of terabytes running on commodity clusters that deliver excellent performance.
- **Ease of management:** Cassandra clusters are easy to manage at scale and can be resized dynamically across clouds as requirements change.
- **Continuous availability:** Cassandra is designed to be “always on” and has supported features such as zero-downtime upgrades for over a decade.
- **Write-intensive applications:** Cassandra is especially suited to write-intensive applications such as time-series streaming data, sensor log data, and IoT applications.
- **Statistics and analytics:** Cassandra serves as a data store for distributed analytic frameworks such as Spark. Users can leverage Spark’s powerful in-memory analytic operators using the DataStax Spark Cassandra Connector.[iii]
- **Geographic distribution:** Cassandra supports geographical distribution and efficient data replication across multiple clouds and data centers.

Cassandra is deployed in all these areas, supporting applications in online retail, internet portals, time-series databases, customer 360, industrial applications, and mobile application backends.

ARCHITECTURAL OVERVIEW

Highlights

Cassandra is a free, open source database written in Java. A Cassandra cluster is made up of multiple nodes. Cassandra nodes typically run on Linux® and the only requirement to participate in a cluster is that the nodes are able to communicate with one another via a few well-known TCP/IP ports. Linux administrators will appreciate Cassandra's elegance and simplicity:

- The same software runs on every Cassandra node, making it easy for administrators to automate installation and management using familiar tools.
- The Cassandra software is approximately 50 MB (compressed)—tiny by modern standards—and installation is localized to just two main disk locations.
- Cassandra has few prerequisites. It runs on almost any hardware or cloud platform. The only requirement is that a Cassandra node runs a supported Java Virtual Machine.
- Other than a few configuration files related to topology, all configurable items are localized in a single, well-documented *cassandra.yaml* file, making Cassandra easy to configure.
- Cassandra also provides storage flexibility, with data localized to a single data directory and administrators free to choose the underlying disk technology.

Cluster topology

A good way to understand Cassandra is to look at how clusters are physically deployed. Cassandra clusters are organized into nodes, racks, and data centers as shown in Figure 1.

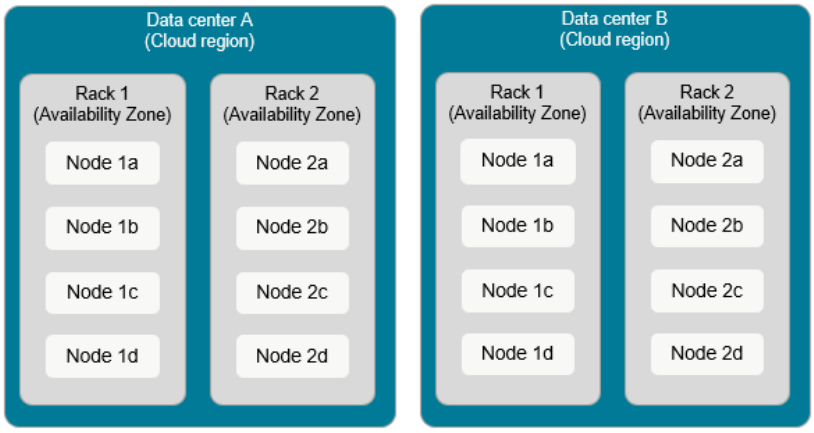

Figure 1: Cassandra cluster topology across racks and data centers

- A **node** refers to a computer system running an instance of Cassandra. A node can be a physical host, a machine instance in the cloud, or even a Docker container.

- A **rack** refers to a set of Cassandra nodes near one another. A rack can be a physical rack containing nodes connected to a common network switch. In cloud deployments, however, a rack often refers to a collection of machine instances running in the same availability zone.
- A **data center** refers to a collection of logical racks, generally residing in the same building and connected by a reliable network. In cloud deployments, data centers generally map to a cloud region—for example, on AWS, *us-west-1,* and *us-west-2*.

Cassandra typically stores copies of data across multiple data centers to ensure availability, while preferring to route queries to other nodes in the same data center to maximize performance. To achieve this, Cassandra uses two internal protocols to manage data placement based on cluster topology: *gossip* and *snitches*.

Gossip

The Gossip protocol[iv] in Cassandra allows each node to keep track of the state of other nodes in the cluster. A "Gossiper" inside each Cassandra instance runs every second and chooses up to three random nodes to initiate a gossip session with. The nodes exchange information with one another about other nodes each has previously gossiped about so that all nodes quickly learn the overall cluster state. Cassandra decides whether a node is up or down based on whether a Gossip session can connect, helping it route requests optimally within a cluster.

Snitches

The other key input to request routing in Cassandra is something called a "snitch." The job of a snitch is to inform each node about the relative proximity of other nodes. This information is used to determine which nodes to read from and write to, and how best to distribute replicas to maximize availability in case a node fails or a rack or data center becomes unreachable.

There are multiple snitch types in Cassandra. The recommended snitch in most environments is something called the "GossipingPropertyFileSnitch." Cassandra also provides snitches optimized for multi-region cloud environments. For example, when deploying a Cassandra cluster that spans AWS regions, users can select the "Ec2MultiRegionSnitch."

Logical ring structure

Just as nodes, racks, and data centers describe how Cassandra clusters are physically deployed, the concept of a "ring" is commonly used to explain how data is organized logically.

Cassandra Query Language (CQL) is much like SQL, making it easy to learn. Consider the example of a pet store with many franchises. A table called "stores" contains information about each affiliated business. We can create a table using the following CQL commands:

```
cqlsh> USE petstore;
cqlsh:petstore> CREATE TABLE stores (
            ...    store_id uuid PRIMARY KEY,
            ...    store_name text,
            ...    country_code text,
            ...    state_province text,
            ...    city text,
            ...    phone_number text
            ... );
```

To decide where data is stored, the partition key (the store_id in this example) is hashed to determine a *token*. A token is a 64-bit integer ID ranging from -2^63 to +2^63 that is used to identify each partition. The token value determines which Cassandra nodes the data will reside on.

Each Cassandra node in the ring shown in Figure 2 is assigned a range of token values. In earlier versions of Cassandra (before version 1.2), token ranges were manually assigned to nodes. A node claimed ownership of the range of values less than or equal to each token and greater than the token of the previous node. The example illustrates the ring layout for a cluster that consists of a single data center with two racks. Notice how the arrangement is structured such that consecutive token ranges are spread across nodes in different racks.

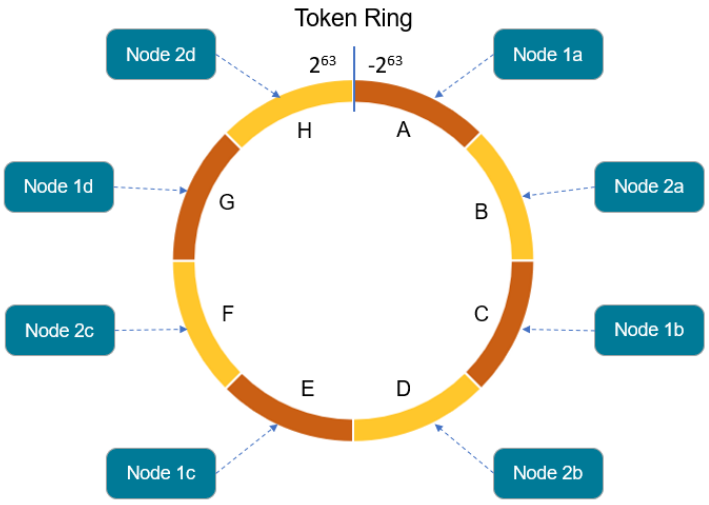

Figure 2: Ring layout in Cassandra

Current versions of Cassandra support the notion of virtual nodes, or *vnodes*, where each Cassandra node supports multiple token ranges distributed throughout the token ring. The number of vnodes per host is configurable. Virtual nodes make Cassandra easier to manage because the generation and assignment of token ranges are handled automatically.

Cassandra's *nodetool* command provides visibility into how token ranges are managed. In this example, a three-node DataStax Distribution of Apache Cassandra cluster is running on Amazon Elastic Compute Cloud (EC2). Because an EC2-specific snitch is used, the data center name

maps to the AWS region (us-east) and the rack maps to the availability zone (1d). Each Cassandra node is managing 256 tokens in this example and each token defines a token range. The three-node cluster has a replication factor of two configured for the petstore keyspace, so each node owns approximately two-thirds of the logical ring.

```
$ ./nodetool status
Datacenter: us-east
===================
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address         Load        Tokens    Owns (effective)  Host ID                               Rack
UN  172.31.49.238   195.39 KiB  256       65.2%             3a6cc136-a277-4e0f-b837-92b53c95c469  1d
UN  172.31.51.1     252.02 KiB  256       68.2%             663d6d82-79ed-4e5d-8361-12bcd24ae79a  1d
UN  172.31.48.193   238.95 KiB  256       66.6%             edf2dfed-f8ed-4c91-9b9a-b328af56cc6a  1d
```

Queries, cluster-level replication

In Cassandra, a client can be a user running CQL commands or a program that connects to Cassandra using a language-specific Cassandra driver.[v] DataStax provides the majority of the de facto standard open source drivers for the Cassandra community. One of the main benefits of Cassandra's masterless architecture is that a client can connect to any Cassandra node. The Cassandra driver implements a load balancing function to distribute client requests across nodes. The node that a client connects to is known as the coordinator node for the query.

The coordinator is responsible for interacting with other nodes sending requests, collecting results, and returning the result of the query to the client as shown in Figure 3. To ensure that data is available even if nodes fail or are unreachable, Cassandra stores data redundantly across multiple nodes, depending on the replication factor specified when the keyspace was created.

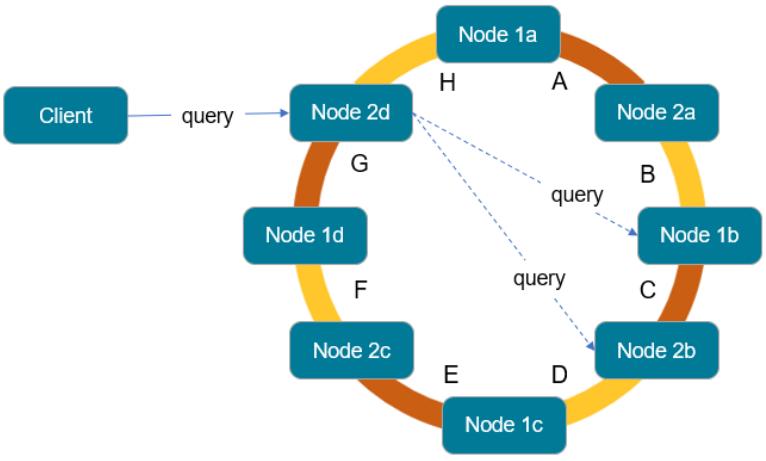

Figure 3: Clients connect to a coordinator node that passes queries to replicas

Cassandra takes the topology of the cluster into account and attempts to distribute data across racks (or cloud availability zones) and data centers, if appropriate. The partition key determines which cluster nodes hold replicas.

Tunable consistency in Cassandra

In distributed databases, it takes time for updates to propagate across networks to remote cluster nodes. Unlike an RDBMS, which can guarantee consistency (at the expense of performance and scalability), distributed databases are usually "eventually consistent." An underrated feature of Cassandra is that it provides tunable consistency, allowing users to manage the trade-off between data consistency and performance. Consistency can be managed globally or can be adjusted for individual read and write operations.

Users can set preferences related to data consistency using the CONSISTENCY command in CQL or programmatically using the Cassandra client drivers. The consistency refers to the number of replicas required to agree on a result (ONE, TWO, or THREE, for example) or keywords with specific meanings such as ANY, ALL, QUORUM, LOCAL_QUORUM, etc. If a quorum of nodes agree on a result, the user will have higher confidence that the result is correct. A quorum is calculated using the formula (sum_of_replication_factors / 2) + 1 and rounding the result up to the nearest integer. In a two-data center cluster, where each data center has a replication factor of three (meaning six replicas) a quorum would be four nodes, meaning that two nodes could be down and the query could still succeed ([6 / 2] + 1 = 4).

These options give application developers the flexibility to manage trade-offs between data availability, consistency, and application performance. When updating important data (such as a financial account balance, for example), strong consistency will be required. For less critical applications, such as showing the number of customer reviews for an item on an online store, it may be appropriate to relax consistency to achieve better performance.

So far, we've been treating each Cassandra node as a black box. To understand how Cassandra works under the hood, it's useful to look inside a node to understand the various components housed within the Cassandra daemon on each node.

Node-level architecture

The Cassandra daemon manages various in-memory and disk-based data structures, as shown in Figure 4. Commit logs are used to record writes to disk as a crash recovery mechanism. One of the reasons that writes in Cassandra are so fast is that all keyspaces share a common commit log, so as soon as a write appends data to the commit log on a replica, as far as the coordinator node is concerned, the write is considered complete.

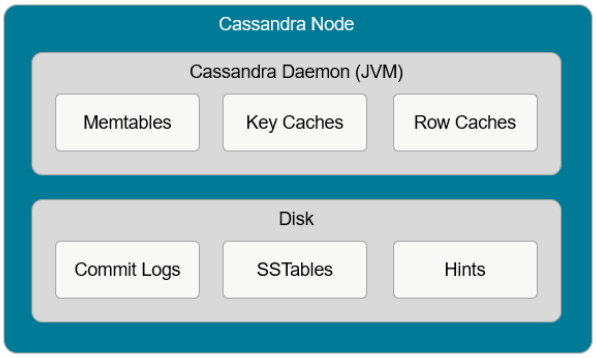

Figure 4: Cassandra log-structured merge storage engine

Sorted String Tables (SSTables) provide permanent on-disk storage for Cassandra. When Cassandra writes data, SSTables aren't stored right away. This is because writes are stored in Memtables to maximize performance and are only flushed periodically to disk.

Row caches and key caches are used to cache frequently accessed data to help with performance. For small tables, the entire table may be cached in memory.

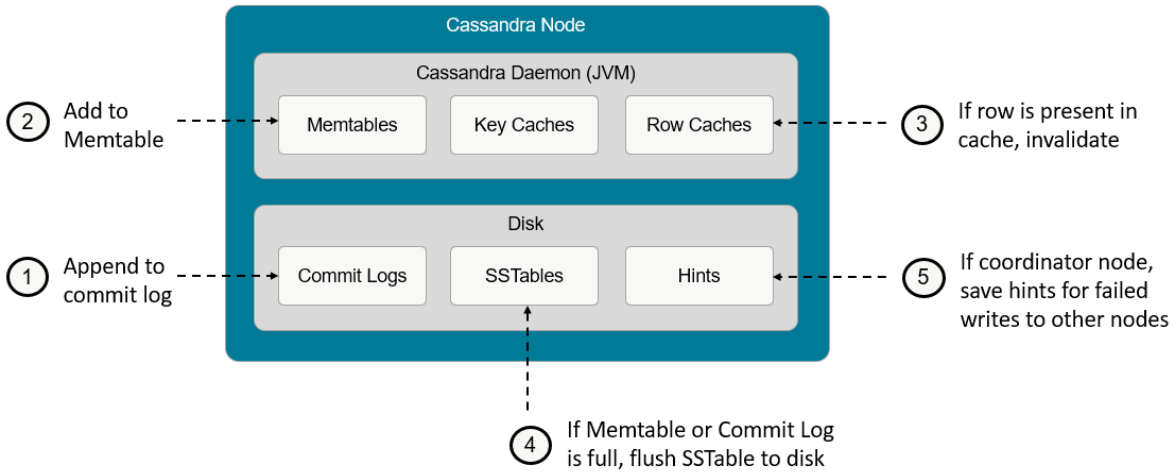

Figure 5: The anatomy of a write operation in Cassandra

How Cassandra writes data

Figure 5 illustrates how these components work together to facilitate a write operation:

1. When data is written to a node, it is first stored to the commit log so that the write can be recovered if the node fails.
2. A copy of the data is also stored in the memtable where it is accessible for subsequent read operations or future updates without the need to go to disk.
3. Next, if the row cache is in use and there is an older copy of the row already in the cache, the old copy is invalidated and replaced with the new data value.
4. In the background, Cassandra monitors the size of the memtable. If the memtable reaches a certain threshold size, Cassandra writes the memtable data to SSTables which are never deleted. If different columns are updated in the same row via different write operations, there will be multiple SSTable files. Cassandra has a mechanism called compaction that runs periodically to consolidate SSTables.
5. Finally, if the "hinted handoff" feature is enabled, and if the coordinator detects that a node has become unresponsive during a write (due to network issues, overload conditions, garbage collection pauses, etc.) the missed write operation will be stored as a "hint" on the coordinator. The hint contains the identifier of the node that failed to accept the write as well as the data to be written. When the Gossip protocol discovers that the failed node has come back online, the coordinator node will replay hints for operations that had previously failed and remove the hints from the coordinator. Hints expire after a configurable period to prevent them from building up on the coordinator. It's important to monitor the health of the cluster so that a downed node can be restarted or replaced before hints expire.

Write performance

One of the most impressive features of Cassandra is its exceptional write performance. Cassandra can complete a write as soon as data is logged to the commit log while other operations happen asynchronously. Also, performance scales directly with the number of nodes in the cluster. In our simple example, we've written only a few records to our table and measured the average write latency to be 0.69 milliseconds or approximately 70 microseconds. Cassandra has been shown to deliver up to one million writes per second in production-scale clusters.[vi]

```
$ bin/nodetool tablestats petstore
Total number of tables: 36
----------------
Keyspace : petstore
        ..
        Write Count: 6
        Write Latency: 0.06933333333333333 ms
```

How Cassandra reads data

Reads in Cassandra are more complicated than writes. The read operation begins when a client connects to a coordinator node with a read query. Like a write request, the coordinator node will use the partitioner to determine which nodes hold replicas of the data.

Like writes, the performance of a read depends on the consistency required for the query. As a reminder, read consistency refers to the number of replicas that need to agree before a result is considered valid. If there is no consensus on a result, Cassandra will internally run a "read repair" operation, forcing Cassandra to update pending changes lingering on replicas before returning a result to the client.

This is an example of how requiring strong consistency can affect performance. For each replica contacted during a read, Cassandra needs to perform several steps and combine results from the active memtable and potentially multiple SSTables as well.

Figure 6 describes the sequence of operations on each replica node.

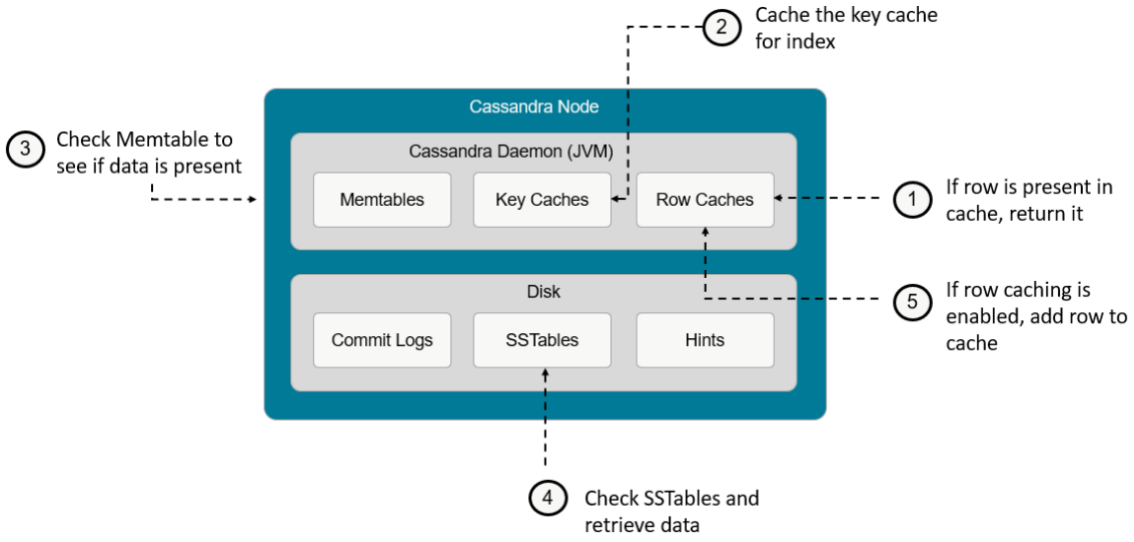

Figure 6: The anatomy of a database read operation in Cassandra

1. When querying a replica, the first place to look is in the row cache. If the needed data is available in the row cache, it can be returned immediately.
2. Next, Cassandra will check the key cache (if enabled). If the partition key is found in the key cache, Cassandra can use the key to learn where data is stored by reading an in-memory compressed offset map.
3. Next, Cassandra will check the memtable to see if the required data is present.
4. After this, Cassandra fetches data from SSTables on disk and combines it with data from the memtable to construct an up-to-date view of the data queried.
5. Finally, if row caching is enabled, Cassandra will store the data in the row cache (to accelerate subsequent reads of the same data) and return results to the coordinator node.

These steps are slightly simplified. Cassandra has additional optimizations such as in-memory bloom filters that can speed up the process of partition key lookups by narrowing the pool of keys to search. A complete description of the read process can be found in the DataStax Cassandra Database Internals Documentation.[vii]

Understanding Cassandra's data model

So far, we've been looking at Cassandra from an architectural perspective and explaining how it works internally. In this section, we'll look at Cassandra's data model and explain why it is so well-suited to handling very large distributed databases.

At first glance, Cassandra looks a lot like a relational database. It has tables with rows and columns, and the CQL command syntax looks much like SQL. However, Cassandra behaves and stores data differently than a relational database, so in this sense the data model can be confusing to new users.

A good way to explain Cassandra's data model is to look at a typical internet-scale application and examine how Cassandra handles the problem versus how a relational database handles it.

A typical customer problem

Consider the example of an online service provider or retailer managing millions of customer accounts. For illustration purposes, we'll focus on just two tables:

- A **customer table** maintains a list of anyone who uses the online service. It contains details like the customer's unique ID, login, name, encrypted password, etc.
- A **customer_events table** logs a record every time a customer has a significant interaction with the system—web logins, purchases, page views, etc.

Now imagine a service with ten million user profiles where each user has logged an average of 1,000 events, each approximately 1 KB in size. This works out to ten terabytes of data in the *customer_events* table alone, and that's before replication and assuming no overhead.

Challenges with relational databases

With a relational database, this problem would typically be handled using two tables. The customer table would hold information about customers and would be linked to the customer events table through a common customer ID (the relation) present in both tables. The primary key for the customer events table might be a timestamp indicating when the event occurred, or some other composite key guaranteed to be unique for each transaction.

There are several problems with a relational approach for this type of application:

- Performance would be an enormous challenge. Any query requiring both customer and event information would involve joining two very large tables.
- Traffic from a potential community of millions of users would almost certainly overwhelm a single database instance. Developers could deploy multiple regional databases, but this would complicate the design and make it difficult to aggregate data.
- Ensuring availability and data integrity across cloud regions would also be a challenge. Users would experience periods of downtime and administrators would need to worry about balancing recovery point objectives (RPO) and recovery time objectives (RTO) when relational databases inevitably need to failover to standby instances.

- Finally, in the relational model, it may be necessary to have many null fields. For example, the name of a customer representative may only be relevant for specific types of customer interactions. In the relational model, fields would need to exist in each row regardless of whether they were relevant to the type of event.

Cassandra provides a better way

It appears that customer names and email addresses are being stored redundantly in Figure 7, but as we'll see shortly, this is just a matter of how CQL presents tabular data views. Cassandra supports static columns[viii] where a single value is associated with each partition key.

customer_events table

cust_id	cust_name	cust_email	event_time	event_type	agent_name	..
Aj50nT63	Barney Rubble	rubble@hotmail.com	2018-06-02T19:42:28	pageview	-	xxx
kTe82rn2	Fred Flinstone	fred@gmail.com	2018-06-02T19:42:31	web login	-	xxx
rTq59vd6	Joe Rockhead	joer@yahoo.com	2018-06-02T19:42:38	refund	-	xxx
N6pDgQ5G	Wilma Flinstone	wilma@bedrock.org	2018-06-02T19:43:02	help line	Phillips	xxx
kTe82rn2	Fred Flinstone	fred@gmail.com	2018-06-02T19:43:55	purchase	-	xxx
Aj50nT63	Barney Rubble	rubble@hotmail.com	2018-06-02T19:44:01	timeout	-	xxx
..			..			
rTq59vd6	Joe Rockhead	joer@yahoo.com	2019-03-11T06:24:03	web login	-	xxx

partition key | static columns | clustering key | additional columns

Figure 7: The customer events table in Cassandra

It's important to remember that Cassandra is not storing data in traditional rows and columns. Under the hood, Cassandra stores data as a map of key-value pairs. For example, an individual data item in a row will have a key ("cust_email" for example) and an associated value (rubble@hotmail.com). Also, not every column needs to be represented in each row. If a column value isn't provided, Cassandra simply doesn't store the key-value pair for that record.

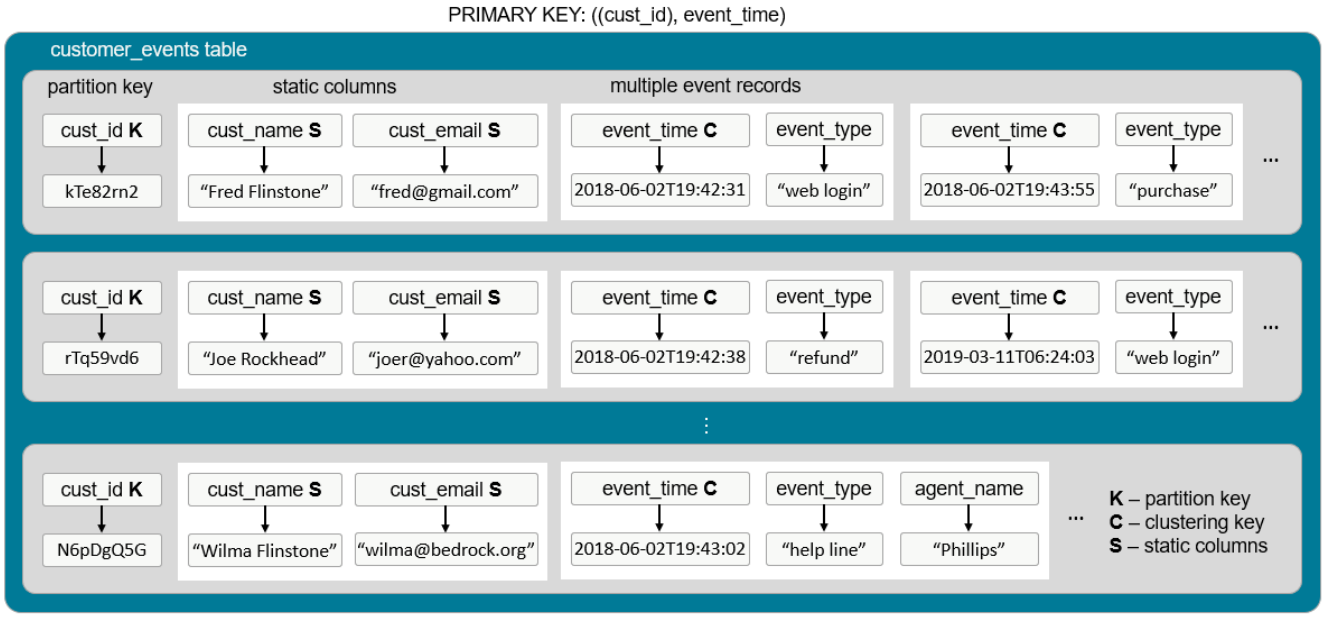

Figure 8: How Cassandra stores customer data in a wide row

Figure 8 shows how our customer_events table is logically stored in Cassandra. The primary key is comprised of both the *cust_id* (the partition key) and the *event_time* (the clustering column). The customer name and email address are stored as static columns, meaning that there is only one copy stored for each associated partition key.

As explained previously, the partition key is hashed to a token value that determines the cluster nodes holding replicas. This partitioning strategy ensures that customer information and associated customer events all reside on the same cluster node. Also, because customer records will be distributed across the cluster, Cassandra can deal with transactions involving many customers simultaneously because it distributes the work across physical nodes.

This use of cluster keys is described as a "wide row" design because there can be thousands of event entries associated with each customer. By partitioning data and allowing users to define how data is stored and grouped using static columns and cluster keys, Cassandra avoids the need for multiple tables, avoids storing data redundantly, and delivers excellent performance.

Cassandra solves all the problems we saw with the relational database approach:

- There is no single-instance bottleneck because any Cassandra node can act as a coordinator for a query. As nodes are added to a cluster, transaction throughput increases.

- Cross-cloud, cross-region integrity issues are resolved as well. Customers can simply access a node close to their location, and Cassandra automatically manages replication to other availability zones, data centers, or cloud providers.
- Data availability is assured because each of the wide rows is written to multiple replicas. If a host becomes unavailable, Cassandra transparently gets data from a replica instead.
- Cassandra avoids the overhead of storing null field values because it only stores key-value pairs for data items that exist.

Scaling Cassandra

One of the biggest challenges that database administrators inevitably run into is the need to scale their database. Databases may need to grow for several reasons:

- The amount of data is growing, and data partitions on cluster nodes are filling up.
- Transaction volumes are growing, and it would be useful to distribute transactions across more cluster nodes for added performance.
- Data access patterns are changing—for example, if you are launching your service in Asia and need to extend your cluster to a new data center or cloud region.

Fortunately, Cassandra provides simple procedures to address any scaling scenario. Administrators can do things like add or remove nodes, migrate nodes to different availability zones, or add or remove data centers or cloud regions. Cassandra can automatically accommodate changes and rebalance tokens while the database continues to service requests.

This ability to easily scale in multiple dimensions has many advantages. Unlike a relational database where customers may need to size a database in anticipation of future growth requirements, Cassandra databases can start small and be easily expanded, allowing the investment in technology to scale with the growth of the business.

MULTIPLE DEPLOYMENT OPTIONS

Another key strength of Cassandra is that customers have a variety of deployment options. Unlike cloud-specific databases such as Microsoft's CosmoDB, Amazon's DynamoDB, or Google Cloud Spanner, customers can run Cassandra on-premises or on any cloud platform.

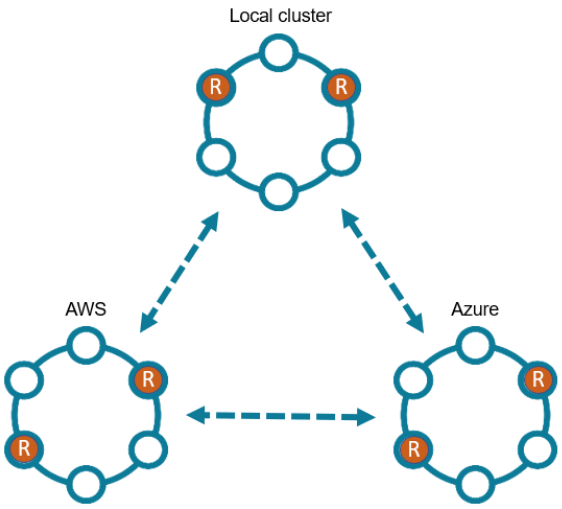

Figure 9: Cassandra cluster configured with two replicas per data center

This topology model consisting of nodes, racks, and data centers together with flexible replication policies provides many deployment options. Cassandra clusters can run in a single cloud or data center or optionally be distributed across multiple locations (data centers, cloud availability zones, cloud regions, or even cloud providers) as shown in Figure 9.

Control over how data is replicated across availability zones and data centers turns out to be very important in practice. With Cassandra's "SimpleStrategy" used for replication within a single data center, replicas are simply placed on the next clockwise node around the Cassandra ring.

The "NetworkTopologyStrategy" provides more precise control over replication, satisfying reads and writes locally (to avoid cross-data center latency) and ensuring that data is replicated across regions or data centers for data availability. In a multi-cloud or multi-data center configuration, administrators will define a replication strategy for each keyspace. The strategy shown in Figure 9 provides two replicas per data center (to ensure a local copy of data is always available) for a total of six replicas across three sites. Asymmetric replication strategies are possible as well.

In addition to running Cassandra across local data centers or cloud machine instances, Cassandra can be deployed in a Docker container, providing even more deployment flexibility.

ABOUT DATASTAX DISTRIBUTION OF APACHE CASSANDRA

DataStax has provided the majority of the commits to Cassandra and is the main provider of open source Cassandra drivers for the most popular development languages. DataStax also supports the Cassandra community by running a variety of events, workshops, and developer initiatives, and by providing free online training, tools, and documentation.

To make it easier for customers to get started with Cassandra and deploy it in enterprise environments, DataStax provides a tested, 100% open source compatible distribution of Cassandra. DataStax Distribution of Apache Cassandra is completely storage and application-compatible with Cassandra and is fully tested and supported by the Cassandra experts at DataStax.

Using DataStax Distribution of Apache Cassandra provides many advantages. It offers multiple support options providing organizations with flexibility and taking the risk out of open source deployments. In addition to production certification, DataStax Distribution of Apache Cassandra provides key features not available in open source Cassandra, including the DataStax Bulk Loader™, the DataStax Kafka™ Connector, and support for Docker containers.

Feature	Open Source Apache Cassandra	DataStax Distribution of Apache Cassandra
Free, open source compatible	✓	✓
DataStax Cassandra drivers	✓	✓
Production certified		✓
DataStax Bulk Loader		✓
DataStax Kafka Connector		✓
Production Docker images		✓
Technical support		✓
Support SLAs		✓
Professional services		✓
Hot fixes		✓
Bug escalation		✓

DataStax Bulk Loader

Users frequently need to load or export data for a variety of reasons that include ingesting data from other sources, taking off-line copies of databases or key tables, and integrating with workflow management tools to move data in and out of Cassandra. The DataStax Bulk Loader[ix] provided with DataStax Distribution of Apache Cassandra supports a variety of file formats, handles parse errors and database insertion errors gracefully, and can load data up to 4x faster than the native CQL COPY command.

DataStax Kafka Connector

Apache Kafka is a distributed queuing system widely used for streaming applications and event-driven applications. It is often used as a buffer layer between external data sources and databases, including Cassandra. The DataStax Kafka Connector provided with DataStax Distribution of Apache Cassandra supports a variety of Kafka data formats and makes it easy to move data efficiently between Kafka and open source Cassandra. Like the bulk loader, the Kafka Connector benefits from DataStax experience in driver development and delivers superior performance.

GETTING STARTED WITH DATASTAX DISTRIBUTION OF APACHE CASSANDRA

DataStax makes it easy to get started with open source Cassandra. There are two primary installation paths for the DataStax Distribution of Apache Cassandra:

1. **Traditional installation** - Users can download a compressed tar file containing DataStax Distribution of Apache Cassandra by visiting http://academy.datastax.com and creating a free account for the DataStax Academy. With this free account, you can download DataStax Distribution of Apache Cassandra as well as access installation guides, documentation, and additional materials.
2. **Docker installation** - Users can also download production-certified DataStax Distribution of Apache Cassandra from Docker Hub by visiting https://hub.docker.com/r/datastax/ddac.

DataStax provides a variety of additional resources, including access to drivers, additional software and documentation, online self-paced computer-based training, and certification exams for Apache Cassandra developers and administrators.

SUMMARY

Over the past few decades, database technologies have changed dramatically. Apache Cassandra has emerged as one of the most successful database technologies for the internet age. Cassandra meets all customer requirements around scale, performance, and availability,

making it a preferred choice among IT architects, developers, administrators, and decision-makers. Cassandra is also open, easy to deploy and manage, and unlike proprietary cloud databases, Cassandra can be deployed on-premises or on your choice of cloud platform.

DataStax Distribution of Apache Cassandra is the most advanced and best-supported Cassandra distribution, providing important features not found in open source Cassandra.

To learn more about the DataStax Distribution of Apache Cassandra, visit www.datastax.com or send an email to info@datastax.com

REFERENCES

[i] Dynamo: Amazon's Highly Available Key-value Store - https://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf
[ii] Bigtable: A Distributed Storage System for Structured Data - https://static.googleusercontent.com/media/research.google.com/en//archive/bigtable-osdi06.pdf
[iii] Spark Cassandra Connector - https://docs.datastax.com/en/dse/6.0/dse-dev/datastax_enterprise/spark/sparkJavaApi.html
[iv] Gossip protocol in Cassandra - https://docs.datastax.com/en/cassandra/3.0/cassandra/architecture/archGossipAbout.html
[v] Cassandra drivers are available for a variety of programming languages - http://cassandra.apache.org/doc/latest/getting_started/drivers.html
[vi] Cassandra delivers one million writes per second at Netflix - https://medium.com/netflix-techblog/benchmarking-cassandra-scalability-on-aws-over-a-million-writes-per-second-39f45f066c9e
[vii] Cassandra database internals documentation - https://docs.datastax.com/en/cassandra/3.0/cassandra/dml/dmlDatabaseInternalsTOC.html
[viii] Static columns in Cassandra - https://docs.datastax.com/en/dse/6.7/cql/cql/cql_using/refStaticCol.html
[ix] DataStax Bulk Loader - https://www.datastax.com/2018/05/introducing-datastax-bulk-loader

ABOUT DATASTAX

DataStax delivers the always-on, active everywhere distributed hybrid cloud database built on Apache Cassandra™. The foundation for personalized, real-time applications at scale, DataStax Enterprise makes it easy for enterprises to exploit hybrid and multi-cloud environments via a seamless data layer that eliminates the issues that typically come with deploying applications across multiple on-premises data centers and/or multiple public clouds.

Our product also gives businesses full data visibility, portability, and control, allowing them to retain strategic ownership of their most valuable asset in a hybrid/multi cloud world. We help many of the world's leading brands across industries transform their businesses through an enterprise data layer that eliminates data silos and cloud vendor lock-in while powering modern, mission-critical applications. For more information, visit www.DataStax.com and follow us on Twitter @DataStax.

Last Rev: MAY2019

DATASTAX