Analysis of Facebook's Use of Apache Cassandra

NoSQL Database Management, Spring 2020

1st Rhea Qianqi Rao
Carnegie Mellon University
qianqir@andrew.cmu.edu

***Abstract*—As one of the fastest growing social media platforms, Facebook has over 2.5 billion monthly active user worldwide. Storing the huge amount of data each user generates through all the different services the platform provides, is just as big a challenge as making sure these data are readily retrievable by users of various geographical locations and time-zones. There is no doubt that choosing the appropriate database architecture is one of the most critical business decisions for Facebook. From the early use of MySQL, through the development and phase-out of TAO, to the current employment of ten different database tools, Facebook has been through several database transformations. This paper aims to briefly introduce the evolution of database management at Facebook, and explore in depth Facebook's current utilization of Apache Cassandra for its Messenger service.**

***Index Terms*—NoSQL, Database, Facebook, Cassandra**

I. The Evolution of Facebook

From its founding in 2004, Facebook has witnessed the growth and the burgeoning of Big Data, as well as the industry's migration from relational database to NoSQL databases. As the world's largest social media network, Facebook offers invaluable lessons in its creation and handling of a globally distributed architecture.

A. Relational Database Era - MySQL

As with most of the enterprise apps in early 2000s, Facebook started off employing the popular relation database. Facebook was built as a PHP application whose persistent database is MySQL. To better aid performance, Facebook implemented look-aside caching using Memcache. Together, the almost simplistic two-database structure supported all the early Facebook functionalities.

Soon after founding, Facebook experienced meteoric success, and the overwhelming amount of data coming into the system put enormous strain on the previous database architecture. As data volumes grow exponentially, MySQL's inability to scale became increasingly obvious and painful. To scale horizontally, an application-level sharding would have to be enforced. So the application has to keep track of which nodes are keeping whose profiles, not only did this increased the operational difficulties, but it was also hard for the applications to perform any cross-shard join operations, which is what made SQL so powerful.

The divide between the master and slave replicated copies also led to other problems like asynchronicity between the master and the salves, and between database and caches. For example, a consistent read could be difficult for a cache for remote region, and instead, stale data would be served to certain users, leading to different state of data for users in different regions.

Fig. 1. Early Database Structure at Facebook

B. Improvement upon relational database - TAO

Starting in 2009, Facebok started exploring the possibilities of a non-relational database by developing TAO (short for The Associations and Objects), a graph database that runs on sharded MySQL. Much like the graph data structure, TAO serves up data points as nodes, and associations between different data points as edges. TAO is hence great for portraying and querying a web-like data structure not unlike that of people's relationships to each other. On Facebook's end, developers would no longer need to be proficient in both MySQL and Memcache, TAO alone would be sufficient to update and query the database.

Such convenience did not come without caveats - cross-shard ACID transactions were disallowed in TAO. While this ensured the low-latency of user access, it does not ensure atomicity. To fix this issue, Facebook implemented an "asynchronous repair job" in the event when one shard is updated and the others have not yet [1]. Even though TAO made managing distributed database easier without changing its initial investments in sharded MySQL, Facebook has grown to become a multi-service platform with each service demanding a more suitable data structure for that specific unit. Meanwhile, as Big Data era approached, other types of NoSQL databases started to get developed for specific database needs. For example, MongoDB of document-oriented database and its schema-less design is great for business units that care about

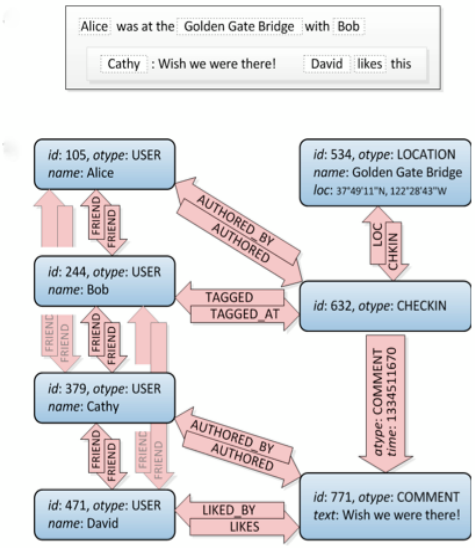

Fig. 2. TAO's Graph-Based API [2]

performances and scaling. Facebook joined fellow tech firms in implementing database architectures that could handle the modern challenges of over-flowing data, and multi-service needs.

C. Big Data Era

Entering 2010s, Facebook started to experiences tremendous business growth and became a conglomerate of various social media applications, rather than a monolithic app as what it started out to be. As different business areas of Facebook was identified and separated, Facebook's database architecture also responded by becoming increasingly polyglot. As Facebook entered the modern era of Big Data, it started using different tools for different components of the company. For example, photo-sharing, messenger, timeline management all became a micro-service of its own rather than the relational model it initially employed. To easier manage these different components, each component takes on a separate layer.

This type of system is called a Polyglot Persistence System, meaning a different database for each business unit, since each of them has its own business use cases. Each micro-service run in conjunction of each other, making it easy to scale, and can be easily navigated for different user cases.

Given the Polyglot Persistence Architecture Facebook employs, it utilizes different tools for its many needs in the application. For inbox search, Apache Cassandra is initially used, before Facebook moved to Apache HBase. Facebook also used to use Cassandra for Messenger, until it switched to HBase in 2010, and again MyRocks engine in 2018. Apache Hadoop, HBase, Hive, Apache Thrift, PrestoDB are used for data ingestion and data analytics. Beringei Gorilla for infrastructure monitoring, and LogDevice to store logs.

Employing different NoSQL micro-services has its merits and curses. ACID requirements, for example, are difficult to

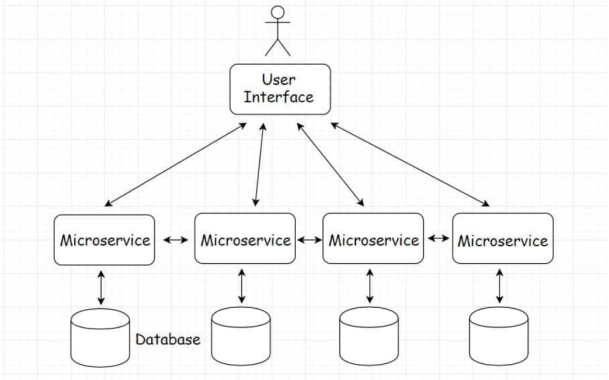

Fig. 3. Pologlot Persistence Architecture. [3]

meet. While diving business logic into different micro-services in part ensures the consistency of data within that specific business logic, ACID is not fully supported by most NoSQL database. Instead, NoSQL databases follow BASE properties (Basically Available, Soft State, Eventual Consistency), which give up on the hard requirement on consistency for better availability, which definitely differs from the ever-consistent SQL and highly available caching.

In this paper, the focus is going to be on Cassandra, the open-sourced distributed database Facebook developed for inbox search. Even though Facebook switched to HBase from Cassandra for inbox search in early 2010s (the exact date undisclosed), Cassandra went on to be a core database for companies like Apple, Netflix, Reddit, etc. Starting 2012, Instagram, one of Facebook's subsidiaries, started using Cassandra for multiple user cases, including fraud detection, feed, and the direct inbox. Instagram is current one of the world's largest deployments of Cassandra [8]. After a brief introduction to Cassandra, the paper will delve deeper into the different use cases of Cassandra by Facebook, and explored the reasons why Facebook migrated over to HBase.

II. Apache Cassandra

Written in house, Cassandra runs across many data centers, and is built to maintain data persistency in case of node failures. Before Cassandra came into being, Facebook was facing the problem of an increasing inbox message growth, the challenge is to not only find an effective and scalable way to store all the fast-growing messages, but store reverse indices of the messages users send and receive. Born out of a Hackathon, Cassandra is created as a distributed storage system that aims to scale without any single point of failure. On the other hand, the relational data model aspect is not as emphasized – it was envisioned as a simple data model that supports high availability, scalability, and high applicability.

A. Column-Family Stores

Cassandra model does not utilize just one type of NoSQL model like Key-Value pair or document storage, rather, it's a hybrid of the key-value model and a tabular database management system. While Cassandra has its own query language, it

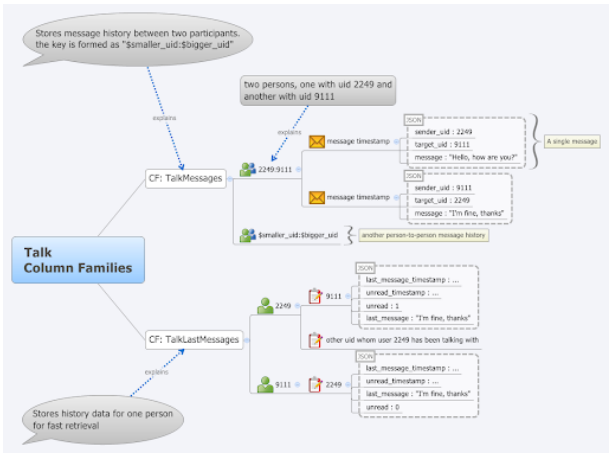

Fig. 4. Cassandra's Column-Family Store Model [4]

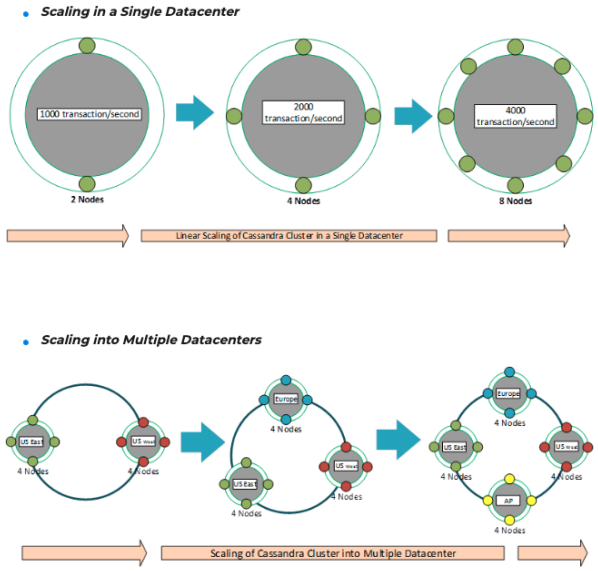

Fig. 5. Scaling in Cassandra [7]

can't do joins or sub-queries operations, which can be difficult for normalized tables. Similar to MongoDB, Cassandra focuses on denormalization through collections. A table in an RDBMS is called a column family in Cassandra, and a column family contains multiple rows and columns. Rows are represented as tables, and each row has multiple columns with different variables. While Cassandra might sound similar to a relational database, unlike a table in an RDBMS, however, each row or table does not have to share the same set of columns. As of 2020, Cassandra is ranked as the most popular Column-Family Store Database By DB Engine [5].

B. Key Features

- Peer-to-peer architecture
- Sharding and auto-sharding support
- Tunable consistency

Since Cassandra is built on top of an infrastructure of different distributed nodes, and it follows a peer-to-peer architecture. Between nodes, communication follows the "Gossip Protocol", which means when one node communicates with another, not only does it get the current status of said node, but also the information of every node said node has talked to before. This avoids chaotic communicates in a peer-to-peer architecture and ensures latest information for different nodes.

High availability is achieved with replication across different data centers. A client would be served by the closest replica, similar to TAO, an "asynchronous repair job" is done in the background for increased read throughput.

To ensure fast and easy scalability, nodes could be added or dropped anytime. Any number of nodes could be added to any cluster, or any data centers. Cassandra could thus be scaled up horizontally by adding more data centers, or vertically by adding more nodes. A failure detector is also employed to monitor failures of nodes within the cluster, thus ensuring that node would be one single point of failure.

The cherry-on-top feature of Cassandra is perhaps its tunable consistency characteristic. There are two types of consistency – eventual consistency and strong consistency. Eventual consistency, as the name indicates, only ensures the updated data ultimately would be updated on every node, regardless of the time it took. If a request is made to a specific node before the replica is updated, stale data will be returned. On the other hand, strong consistency ensures quite updates at the cost of availability. The tunable feature allows Cassandra to opt for eventual consistency when the data center is remote and latency could be high, and opt for strong consistency during other times. Note that Cassandra still mostly opts for eventual consistency to achieve high availability.

Due to these features, Cassandra promises high write-performances, high availability, and no single point of failure.

C. Major Trade-offs

The obvious trade off for the high availability, easy scalability and no single point of failure is weaker consistency. Even though Cassandra could "tune" for strong consistency, it will have to sacrifice its availability. Cassandra would be a poor choice for any operation that demands high data consistency.

Even though Cassandra is great for writing, it's also append-oriented, meaning the performance for updates are not as great. For every update request, instead of updating the current copy, Cassandra adds a new data version to the data with the same primary key, which could be confusing for read requests [6].

Given the focus on denormalization, Cassandra is not the best for complicated ad-hoc querying. CQL, the Cassandra query language, does not allow embedded queries, which makes it messy and difficult for operations that need to aggregate different information across.

Being a typical NoSQL database, Cassandra has all the performance advantages that other NoSQL databases have, and it has done a great job addressing scalability, performance, availability and maintainability requirements. Such benefits did not come without drawbacks. Cassandra's level of data

consistency suffers, and due to the denormalized storage structure, the business operations applicable with Cassandra is limited.

III. Facebook's use of Apache Cassandra

With its key features, Cassandra seems to do well with operations that requires fast write, fast read, and few updates. During the mid 2000s when Facebook was moving away from MySQL into NoSQL databases, Cassandra was developed and applied to different areas of Facebook's business. One of the most notable uses of Cassandra, which is also the use that Cassandra was developed for in the first place, was inbox search. Even though Facebook had later moved the inbox search functionalities to be supported by Apache HBase, for a long time, Cassandra had been considered as the best-in-class database within Column-Family Stores. Cassandra's original usage for inbox search had since inspired the development of other Column-Family Store databases, and it's worth exploring the use cases of Cassandra for inbox searches, and the associated trade-offs that eventually made Facebook choose other databases over Cassandra.

A. A Database that fits the Business Need – Inbox Search

Ever since Cassandra was created, it had been in production-serving traffic for inbox search. The phase "inbox search" is essentially referring to two types of actions: message search and term search [11]. Message search is returning all previous message interactions with a user when that user's name is looked up as a key, while term search, as the name indicates, is returning all messages pertaining to the term searched for.

For term search, the user id is used as the partition key, and all the words that made up the messages are clustered as rows in said partition (or super-column as it was called back then, or column-family as it has come to be known). The message ids become the column keys within the said column-family, and the messages that contain the word indicated by the row are the column values. On the other hand, for message search, user id is the partition key, but the searched user's id is the column-family, with individual messages as the cells within.

The term "column" is unlike the definition we commonly know of in a relational database, the "column" within the column-family in Cassandra acts more like key-value pairs, and hence column-family is more analogous to a nested sorted map in terms of underlying data structure. Since maps are efficient for key lookups, and the sorted nature returned ordered data, the nested sorted map structure is perfect for searches and lookups of messages like Facebook's inbox search.

To make the search even faster, Cassandra has hooks that supported caching of data. Every time a user clicks on the search bar, a message with the user id is sent to a nearby Cassandra cluster to fetch the cache related to that user's index, so when the actual search occurs, chances are the results of that search are already fetched to local memory. This process is called the "index prefetching".

Moreover, the term "wide rows" used in Cassandra is referring to the fact that a column-family can have countless

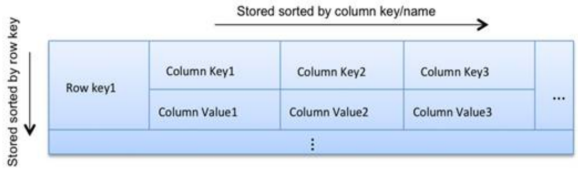

Fig. 6. Cassandra Column-Family [9]

amount of column keys within, which makes it great for scaling and storing more data. As data for existing user increases, the columns grow. As more users join, Cassandra can easily add more nodes to the cluster.

Due to the fast search nature of Cassandra's structure and Cassandra's high availability, even with more than 300 million users at the time when Cassandra came around, the latency levels for the searches were able to be kept really low. In 2010, Term search had a median latency of 18.27ms, while message searches had a median latency of 15.69ms [11].

B. Secondary Indexing and Inverted Indexing

Since search and fast lookup are of essence in inbox search, indexing and inverted indexing are key to Cassandra's operations.

A key features in Cassandra is secondary indexing. Secondary indexing is essentially building indexes over traditional column values. For example, in a user's table, the user id might be used as the primary key, and hence the primary index. A user's name could be easily found with the primary key. If we want to look up the primary key with the user's name, however, we would have to conduct a reverse query. This operation – fetching a user's primary key with said user's name, would require secondary indexing. Cassandra allows storage for both primary and secondary key, hence enabling better and faster searches. The problem with secondary indexing, however, is that it slows down read performances. Secondary indexes are stored differently in Cassandra – the primary index is stored globally within every node, while the secondary index is only stored on local nodes. So while querying by primary key can be served up quickly by any of the nodes, querying by a secondary index means every node in the cluster will have to do a search, which could lead to slow reading performances as the amount of data grows.

To achieve the same search result of secondary indexing without compromising read performances, Facebook engineers decided to go for manual inverted index. The inverted index is stored in a separate column family, enabling the same look up for inbox search as the primary key. Instead of secondary index being stored locally by Cassandra, reverse index storage is like primary index storage – it is distributed globally to every node, thus maintaining the fast read performances.

C. Associated Trade-offs and Alternatives - HBase

While theoretically inverted indexing enables faster searches for Facebook, in practice it induced a large amount of write

workload. Since Facebook's messages were indexed by its words, and since every user has to have its own user index, for every message sent, there has to be a reverse index entry for the recipient.

Facebook also found out that in production, there were a lot more writes involved in inbox search than reads, making inverted indexing an expensive operation [12].

Since there seemed to be a lot more writes than reads, and read throughout was not a concern, Facebook started to consider compromising data availability for data consistency. Facebook turned to Apache HBase, a database that had a similar Column-Family Stores structure as Cassandra. By 2013, Facebook had already employed HBase for Facebook Messengers, and the engineers were fairly familiar with the pros and cons of HBase. HBase was thus being considered and Facebook contemplated the move away from Cassandra.

Compared to Cassandra, HBase, as a fellow Column-Family Store candidate, also offers great scalability, and utilizes replication to avoid node failures. However, HBase operates much differently under the hood. There are numerous differences and nuances between the two databases, but the biggest one is perhaps that Cassandra has a peer-to-peer architect, while HBase operates on a master-slave model. Due to the master-based architecture of HBase, HBase ensures strong data consistency, as every write has to go to the master machine. On the contrary, Cassandra operates on a peer-to-peer architect that ensures high availability and only eventual consistency. In an over simplified term of CAP Theorem – Cassandra is choosing AP, and HBase is choosing CP.

For inbox search, since empirical evidences suggested that read throughput was not as big a concern as writing consistency, Facebook decided to switch to HBase. HBase also offered other great features like auto load balancing, and multiple shards per server, etc. HDFS, the file system underlying HBase, was already in use by Facebook in data processing, so it was also an easy transition for the engineers at the company. At the end of the day, however, the eventual consistency feature was the major reason why Facebook moved away from Cassandra and why Cassandra had not been used more of across Facebook's platform.

By 2013, it was not the first time Facebook had chosen HBase over Cassandra. For example, Cassandra was considered as a database choice for Messengers. Back in the 2000s, Messenger had not developed into its own app as it is now, rather, it operated as a messaging feature on the Facebook website. As Facebook entered 2010, it decided to incorporate a pop-up chat feature into the message functionalities. Even then, it had to find a new infrastructure that could handle 300 million users who sent over 120 billion messages per month [13]. The database had to be able to scale fast and handle volatile temporary data. In 2010, HBase was chosen over Cassandra for better consistency. HBase also made it easier to repaid missing replicas, due to the feature of the underlying HDFS. Cassandra, at the time, did not have the "repair job" (a workable merkel tree underneath the hood) feature that it has now, hence it was harder to replace replicas.

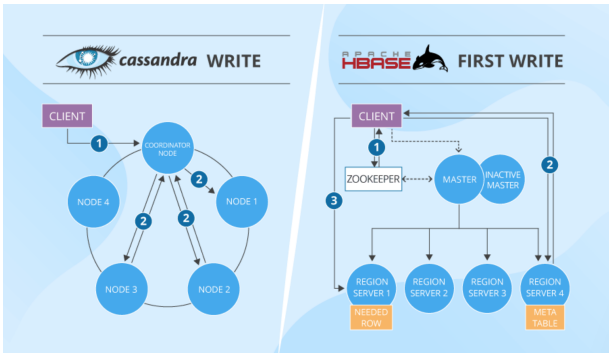

Fig. 7. Cassandra vs HBase Architecture [10]

IV. Conclusion

As the amount of data accumulated in the modern era experienced an explosive growth in the past two decades, Facebook's journey of moving from a simplistic application structure based on relational database to a Polyglot Persistence System employing various NoSQL databases along with RDBMS mirrors that of the industry. As the world's largest social media network, Facebook offers invaluable lessons in its development and employment of different NoSQL databases, and it handling of a globally distributed architecture. As seemed from Facebook's development of the once best-in-class Column-Family Store database Apache Cassandra for its inbox search functionality, to Facebook migration to Apache HBase after a mere couple of years, the lesson learned here is that the field of NoSQL is ever-evolving. Companies continue to strive to improve its current tools and databases to better suit its business needs. Facebook's database journey and continuous improvement of architectural design also illustrates the importance of prioritizing business goals, and choosing the right database accordingly. From MongoDB, to Cassandra, to HBase, there is no one single database that is perfect in every aspect. The evolutional Cassandra has high availability, high scalability and avoids one single point of failure, which was perfect for Facebook's then urgent need for a scalable system for inbox search with fast writes and reads, but as databases and business situations continued to evolve, Facebook eventually chose the scalable and more consistent HBase. Facebook's journey is perhaps the best real-life illustration of the famous oxymoron that the only constant is change, at least in the realm of NoSQL database.

REFERENCES

[1] K. Ranganathan, "Facebook's User DB—Is It SQL or NoSQL?", July 2019.

[2] Facebook Engineering Blog, "TAO: The power of the graph", June 2013.

[3] Shivang, 8bitmen.com, "Facebook Database–A Thorough Insight Into The Databases Used @Facebook", published date unkown.

[4] J. Makinen, "Example how to model your data into nosql with cassandra", September 2010.

[5] DB-Engines.com, "DB-Engines Ranking-Trend of Wide Column Stores Popularity", February 2020.

[6] A. Bekker, Medium Blog, "When to use Cassandra and when to steer clear", August 2018.

[7] S. Tapadar, "Cassandra – The Right Data Store for Scalability, Performance, Availability and Maintainability", July 2015.

[8] Instagram Engineering Blog, "Open-sourcing a 10x reduction in Apache Cassandra tail latency", March 2018.

[9] T. Ma, "Learn Cassandra=", Gitbook 2015.

[10] A. Bekker, "Cassandra vs. HBase: twins or just strangers with similar looks?", June 2018.

[11] A. Lakshman, P. Malik, "Cassandra - A Decentralized Structured Storage System", LADIS Facebook's Cassandra paper 2009.

[12] K. Ranganathan, "How does the searching and indexing for Facebook Inbox search work?", February 2013.

[13] Facebook Engineering Blog, "The Underlying Technology of Messages", November 2010.