Query Driven implementation of Twitter base using Cassandra

Conference Paper · March 2018

DOI: 10.1109/ICCTCT.2018.8551136

2 authors:

Dharavath Ramesh
Indian Institute of Technology (ISM) Dhanbad
102 PUBLICATIONS **278** CITATIONS
SEE PROFILE

Anand Kumar
Indian Institute of Technology (ISM) Dhanbad
3 PUBLICATIONS **2** CITATIONS
SEE PROFILE

Query Driven implementation of Twitter base using Cassandra

Dharavath Ramesh
Department of Computer Science and Engineering
Indian Institute of Technology (ISM),
Dhanbad, India
ramesh.d.in@ieee.org

Anand Kumar
Department of Computer Science and Engineering
Indian Institute of Technology (ISM),
Dhanbad, India
Anand.15JE001505@cse.ism.ac.in

***Abstract*— With big data challenges rising in the recent years, NoSQL databases have witnessed a surge in popularity, thus challenging the dominance of relational databases. These next generation databases focus on being non-relational, distributed, open source and horizontally scalable. Cassandra being one of them is a high performance database proficient in handling large volumes of data stored across data centers, providing high availability, fault tolerance and tunable data consistency. However, unlike SQL and MongoDB, Cassandra doesn't support adhoc queries and hence requires an efficient data model largely driven by the kind of query intended to be performed majorly on the database. Social Networking sites like Twitter stores millions of tweets per day leading to large number of read and write operations on a regular basis. This paper suggests a query driven data model in Cassandra to store tweets and maintain user's timeline so that read and write operations can be performed in an efficient way.**

Keywords—RDBMS; Scalability; NoSQL; Consistency; Cluster

I.INTRODUCTION

The term NoSQL was first coined in the late 1990s, however, it has come to prominence with the rise of big data and the need for web scale processing. The needs of web 2.0 companies e.g. facebook, google, amazon and twitter triggered a surge in demand of NoSQL databases in the recent past. The tectonic shift from relational data models to non-relational ones can be attributed to the following reasons:

- The relational model does not provide enough flexibility to handle operations on big data.
- The static nature of the relational data models makes them inefficient in handling time series applications as well as high velocity data.
- RDBMS is vertically scalable i.e. its limits are defined by hardware as compared to horizontal scalability [10] of NoSQL databases which allows increase in capacity by simply connecting multiple hardware or software entities.

Twitter juggles around 10,000 tweets per second (500 million per day) and hence efficient storage of these tweets is of primary importance to them. Cassandra's ability to handle the discrete time series data efficiently raises its stake to be an ideal fit for this use case. To accommodate this instance, in this paper, a Cassandra data model is suggested primarily for storing the tweets and maintaining the timeline of each and every user on Twitter.

The paper organization is as follows: Section II underlines the use cases of Cassandra; Section III discusses the databases which have been used by Twitter; Section IV presents the query driven data model for Twitter; Section V talks about the scope and the future work of this model.

II.CASSANDRA

Cassandra was designed to run on cheap community hardware. Since it stores data in append only format, it provides the most efficient write operation among all other NoSQL databases [5]. Cassandra is being used in managing some of the world's largest clusters having thousand nodes and that too deployed across multiple data centers [3]. In fact, for operations typical to modern Mobile and IOT applications [15], Cassandra performs six times faster than HBase and 195 times faster than MongoDB.

Fig. 1 compares the number of operations per second per node cluster performed by the various NoSQL databases and it's quite apparent that performance of Cassandra better as the number of nodes grow in the cluster and hence Cassandra emerges as a clear choice for managing huge workloads among all NoSQL databases.

Cassandra follows the CAP theorem also known as Brewer's theorem. This theorem states that any distributed data store can maximally ensure two of the following three features:

Consistency: The read operation should return the most recent writes on the database.

Availability: No request to the server should be devoid of response.

Partition tolerance: System continues to operate even if there are arbitrary partitions due to network failures.

However, it is not that one of the above three has to be abandoned all the time. There has to be a tradeoff between consistency and availability only when a network failure occurs. Cassandra gives upper hand to availability over consistency in such situations and this feature only makes Cassandra an apt NoSQL database in dealing with big data [12].

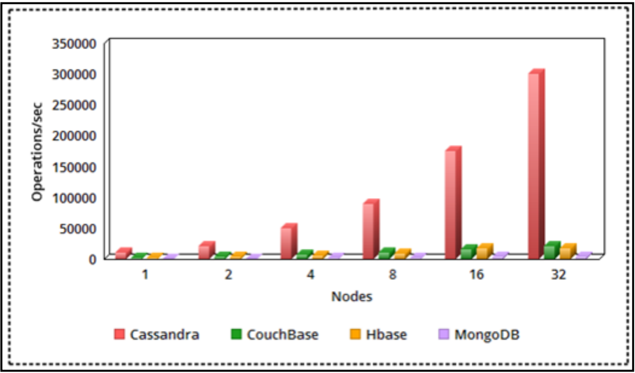

Fig. 1. Performance comparison of NoSQL databases

For web 2.0 websites, eventual consistency [11] is the main source to perform related operations i.e. it is not necessary to load the data posted milliseconds before on the user's timeline. However, the system needs to be available even in the case of multiple network failures.

Since data are stored in a sequential fashion in Cassandra, it offers an obvious advantage in storing huge chunks of data which form some sort of sequence [1]. Hence, the number of data retrievals per second increases many fold leading to fast read operation. Cassandra offers much higher scalability when data is heavily concentrated in a small time interval (e.g. discrete time series data).

To increase the reliability of Cassandra data model, a mechanism has been designed to introduce transparency in managing consistency between different data replicas [9]. The query driven model of Cassandra has additionally been used in implementing a health data model [2].

III. Twitter and Cassandra

Twitter was built on MySQL and originally all data was stored on it. The designers of Twitter expanded from a small database instance to large one, and eventually many large database clusters. However, because of the vertical scalability of MySQL, it became increasingly difficult to distribute data as the network traffic increased million folds.

In 2010, the designers of Twitter introduced framework named Gizzard for creating distributed data stores [4]. The ecosystem at that time was replicated using MySQL clusters and Gizzard based sharded MySQL clusters. They alao introduced FlockDB, a graph storage solution on top of Gizzard and MySQL. In 2010, the methodology of Hadoop has been adopted to store MySQL backups. However, it is now heavily used for analytics. On the other hand, they added Cassandra as a storage solution. However, they didn't store tweets in the Cassandra database. As traffic grew exponentially they launched Manhattan in 2014.

IV. The Query Driven Model

Any data model designed in Cassandra should achieve the following two goals:

Spread data evenly among the nodes in the cluster: We want every node to have almost same amount of data so that formation of hotspots can be avoided in the cluster. A token is derived from the partition key (first element of primary key) value in each row, which determines the node on which that particular row will be stored. Hence a good primary key should be chosen [6].

Minimize the number of partition reads: Since different partitions may reside on different nodes, the coordinator node will have to issue separate commands to separate nodes for each partition you request leading to large overhead cost [13].

Keeping these two goals in context, we will design the data model for Twitter base. The objective is to create a column family database, which will have single tweet stored per row and the columns contain information regarding each tweet e.g. time of the tweet, name of the user publishing the tweet, text in the tweet and an id which uniquely identifies the tweet. Let us assume the name of this column family to be tweets.

CREATE TABLE *tweets (tweetid text, tweet- time text, tweet text, postedby text,* PRIMARY KEY*(tweetid))*;

Choosing *tweetid* column as the partition key ensures that the tweets are distributed evenly across the nodes in a cluster. Apart from the *tweets* column family, each user will have a unique column family having all the tweets which are to be shown on the twitter timeline of that particular user. This column family will have column values like time of the tweet, *id* of the tweet. Let the user's name be $user_1$ then the column family having timeline of $user_1$ will be named *user1timeline*. Let *usertimeline* be the column family unique to a random user.

The *tweets* of column family stores complete information about the tweet so that only *tweetid* and *tweettime* needs to be stored in *usertimeline* column family of the followers of the user who posts the tweet. In this manner *tweets* column family avoids multiple writes of the same data in other column families and for publishing timeline of any user, we can refer to the complete tweet information via the *tweetid stored* in his/her column family.

When a tweet is posted by a user, the complete information about that tweet will be stored as a row in the *tweets* column family. The tweet id and the time of the tweet will be stored in *usertimeline* column family of all the followers of that particular user. The timeline of any user can now be published by simply querying all the rows from the *usertimeline* column family corresponding to that user.

The functionality of *user1timeline* is a column family having time series data since the tweets must be ordered in descending order of time and hence modeling it efficiently is important to get the reasonable performance in querying. So, the main challenge is to pick the most optimal primary key for this use case. The functionality of *tweettime* can't be chosen as the primary key as data is stored on each node depending on the token generated from the partition key and the range assigned to that node. The order of retrieval of partitioned rows is determined by order in which tokens are hashed and not by the values of partition key [7]. Since we want the tweets to be sorted in descending order by time, we need a different primary key.

PRIMARY KEY *(tweettid,tweettime)* WITH CLUSTERING ORDER BY(*tweettime* DESC) will not give the desired results. The tweets will be distributed on different nodes according to the token generated from the *tweetid* value and will be stored in descending order of time in each partition.

```
SELECT * FROM user1timeline;
```

This CQL command will read the timeline of *user$_1$*. However, the timeline may not have tweets sorted by descending order of time as the tweets stored on node$_1$ are sorted among themselves, tweets stored on node$_2$ are sorted among themselves, but the tweets on node$_1$ and node$_2$ combined may not be sorted at all.

Using dummy column (column having same value for all rows) as partition key and *tweettime* as clustering column is not an ideal method. The hash value corresponding to the partition key remains unchanged on each insert leading to storage of all the data on a single node and a hotspot is formed in the cluster [8]. Since Cassandra supports only two billion column per partition, this limit will expire easily for a networking site as big as twitter [14].

We introduce a column *tweetnode* in the *user1timeline* column family which takes the value like 201707 or 201708 i.e. the year in which the tweet was posted concatenated with the month. This column will be the partition key and *tweetime* will be the clustering column. We run the following CQL command to create the column family for user$_1$.

```
CREATE TABLE user1timeline (tweetnode int, tweettime
text, tweetid text, PRIMARY KEY (tweetnode, tweettime))
WITH CLUSTERING ORDER BY (tweettime DESC,
tweetid ASC);
```

The tweets in *user1timeline* column family which were posted in the same month will have the same partition key and hence same hash value. As a result, those tweets will be assigned the same node and will be sorted by descending order of time in their partition. The tweets in *user1timeline* having same *tweetnode* value will be sorted among themselves. We first publish the tweets of the newest month (all those tweets at least are stored in descending order of time), then of the second newest month and likewise the complete timeline of *user$_1$* can be published in the desired order.

Fig. 2 gives a diagrammatic preview of how data are being stored in the database as well as retrieved from the database. The complete process is explained using the following example:

Let *user$_1$* posts a tweet and *user$_2$* and *user$_3$* are following *user$_1$*. We store the complete information about the tweet in *tweets* column family. The *tweetid* and *tweettime* along with *tweetnode* value will be stored in *user2timeline* and *user3timeline* column families.

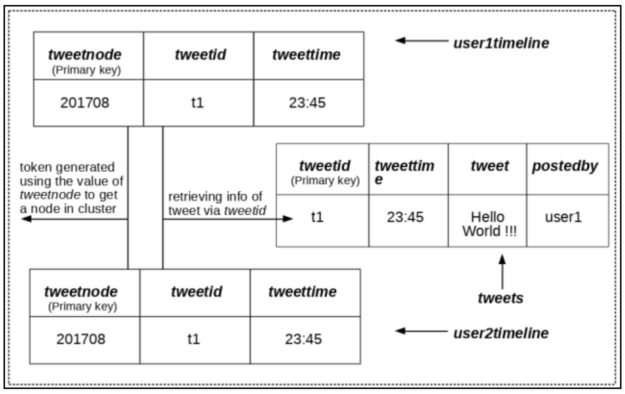

Fig. 2. Storage and retrieval of tweets

We run the following CQL commands to carry out the write operation:

```
INSERT INTO tweets (tweetid, tweettime, tweet,
postedby) VALUES ('t1', '23:45', 'Hello World!!!',
'user1');
  INSERT INTO user2timeline (tweetnode, tweet- time,
tweetid) VALUES (201708, '23:45','t1');
  INSERT INTO user3timeline (tweetnode, tweet- time,
tweetid) VALUES (201708, '23:45', 't1');
```

The read operation can be carried out by running the following CQL command:

```
SELECT * FROM user2timeline;
 SELECT * FROM user3timeline;
```

The above commands read the timeline of *user*$_2$ and *user*$_3$ from the database and the results can be stored in php variables for further use. The *tweetnode* column's value is just used as an example for demonstrating the best pick for primary key. If the number of tweets gets higher and breaches the Cassandra limit for a single partition then we can use *tweetnode* to have value like 3011 or 0812 i.e. the date on which tweet was posted concatenated with the month. Thus, each node in the cluster will have tweets of a single day and a limit of 2 billion rows per partition is large enough to handle this case.

V.CONCLUSION AND FUTURE WORK

The Cassandra data model is the best choice to store huge volume of time series data with high availability and performance. Cassandra usually provides very efficient read operations because of sequential insertion of data in the database and this query driven data model adds to its querying efficiency in implementing the tweet storage system. Cassandra also gives flexibility in removing older data from database automatically, which is not there in RDBMS. TTL (Time to Live) function can be used while inserting data to expire it after a certain period of time. In case of tweets, that period can be set to 10 to 12 months ideally as users tend to check out the latest tweets only.

The future scope of this paper can be to use Cassandra data model to implement other functionalities of twitter like maintaining relationship between user and his/her followers.

ACKNOWLEDGMENT

This work is supported by the Indian Institute of Technology (ISM), Dhanbad that comes under the administrative and financial control of the Ministry of Human Resource Development (MHRD), Government of India. The authors express their gratitude towards the Department of Computer Science and Engineering at IIT(ISM) for providing all necessary support to carry out the research work.

REFERENCES

[1]. Ramesh, D., Sinha, A., & Singh, S. (2016, March). Data modelling for discrete time series data using Cassandra and MongoDB. In Recent Advances in Information Technology (RAIT), 2016 3rd International Conference on (pp. 598-601). IEEE.

[2]. Naguri, K., Sil, P., & Mukherjee, N. (2015, October). Design of a health-data model and a query-driven implementation in Cassandra. In E-health Networking, Application & Services (HealthCom), 2015 17th International Conference on (pp. 144- 148). IEEE.

[3]. Chebotko, A., Kashlev, A., & Lu, S. (2015, June). A big data modeling methodology for Apache Cassandra. In Big Data (BigData Congress), 2015 IEEE International Congress on (pp. 238-245). IEEE.

[4]. The infrastructure behind Twitter- https:// blog.twitter. com/ engineering/ en us/ topics/ infrastructure/ 2017/the-infrastructure-behind-twitter-scale.html

[5]. Apache Cassandra Documentation on Cassandra architecture- *http:// cassandra.apache.org/ doc/ latest/ architecture/ index.html*

[6]. Datastax documentation for data modeling in Cassandra *https:// www.datastax.com/ dev/ blog/ basic-rules-of-cassandra-data-modeling*

[7]. Jay Patel "Data Modeling best practices" Ebay tech blog- *http:// www.ebaytechblog.com/ 2012/ 07/ 16/ cassandra-data-modeling-best-practices-part-1/.*

[8]. Jay Patel "Data Modeling best practices" Ebay tech blog- *http:// www.ebaytechblog.com/ 2012/ 08/ 14/ cassandra-data-modeling-best-practices-part-2/.*

[9]. Hernandez, R., Becerra, Y., Torres, J., & Ayguad, E. (2015). Automatic query driven data modelling in Cassandra. Procedia Computer Science, 51, 2822-2826.

[10]. F. Bugiotti, L. Cabibbo, P. Atzeni, and R. Torlone, Database design for NoSQL systems, in Proceedings of the 33rd Interna- tional Conference on Conceptual Modeling, 2014, pp. 223-231.

[11]. Wang, G., & Tang, J. (2012, August). The nosql principles and basic application of cassandra model. In Computer Science & Service System (CSSS), 2012 International Conference on (pp. 1332-1335). IEEE.

[12]. Prez-Miguel, C., Mendiburu, A., & Miguel-Alonso, J. (2015). Modeling the availability of Cassandra. Journal of Parallel and Distributed Computing, 86, 29-44.

[13]. Lakshman, A., & Malik, P. (2010). Cassandra: a decentralized structured storage system. ACM SIGOPS Operating Systems Review, 44(2), 35-40.

[14]. Schram, A., & Anderson, K. M. (2012, October). MySQL to NoSQL: data modeling challenges in supporting scalability. In Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity (pp. 191- 202). ACM.

[15]. Barbierato, E., Gribaudo, M., & Iacono, M. (2014). Perfor- mance evaluation of NoSQL big-data applications using multi- formalism models. Future Generation Computer Systems, 37, 345-353.