Workload Aware Replicated Datapartitioning for Twitter

Shanty S.R.
Dept. of Computer Science & Engineering
MACE, Kothamangalam
Kerala, India

Aby Abahai T.
Assistant Professor
Dept of Computer Science & Engineering
MACE, Kothamangalam
Kerala, India

Eldo P. Elias
Assistant Professor
Dept of Computer Science & Engineering
MACE, Kothamangalam
Kerala, India

ABSTRACT
Most of the queries in twitter include multiuser operations. When a user login to twitter it requests the most recent tweets of whom he follows. These data may be present in different servers. The expense of these queries depends on how the data is partitioned. Existing solution for data partitioning involve hash or graph based partition. In this paper a new method for reducing the interaction between the servers are proposed. For this the data is partitioned such that most of the users that a user interacts are placed on the same partition. In addition to data partition selective replication is also implemented in the proposed approach. The data about the users that are requested most are replicated more than the other users. Experimental analysis indicates that the proposed technique provides significant improvements in the quality of the partitions, especially under low replication ratios.

General Terms
Data partitioning; Selective replication; Social network; Twitter; Cassandra.

Keywords
Data partitioning; Selective replication; Social network; Twitter; Cassandra.

1. INTRODUCTION
The availability requirements of social networks are high due to its fast growing nature and it has a dynamic structure. These challenges are partially handled by using NoSQL (Not Only SQL) systems, which use data partitioning and replication to achieve scalability and availability. These systems use hash-based partitioning and random replication of data. It ignores the relationship between the users. So it often leads to redundant replication which often leads to huge number of writes on the replicas. This leads to performance degradation.

Besides dealing with large amount of data, massive read and write requests have to be responded without any latency. In order to deal with these requirements, most of the companies maintain clusters with thousands of commodity hardware machines. Relational databases are not suitable in this domain, because joins and locks influence performance in distributed systems negatively. In addition to high performance, high availability is a fundamental requirement of many companies. Therefore, databases have to provide a failover mechanism to deal with failures and should be easily replicable. They also must be able to balance read requests on multiple slaves to cope with access peaks which can exceed the capacity of a single server. Since replication techniques in relational databases are limited and these databases focuses on consistency than availability, these requirements can only be achieved with additional efforts.

Data partitioning is a data distribution technique used for improving the query response time performances of I/O intensive applications. The aim in partitioning is to optimize the processing time of each query requested from distributed servers. This is achieved by reducing the number of servers required for a single user while answering a single query

In addition to partitioning, replication of data items to achieve higher I/O parallelism has started to gain attention. There are many replicated partitioning schemes proposed for optimizing range queries. Recently, there are a few studies that address this problem for arbitrary queries as well.

Data replication is widely applied in various application areas for achieving fault tolerance and fault recovery. Data replication can also be used to achieve higher I/O parallelism in a partitioning system. However, while performing replication consistency must also be considered, which arise in update and delete operations. Furthermore, write operations tend to slow down when there is replication. Finally, replication means extra storage requirement and there are applications with very large data sizes where even two-copy replication is not feasible. Thus, if possible, unnecessary replication has to be avoided and techniques that enable replication under given size constraints must be studied.

The primary focus of this paper is to introduce a workload aware replication and partitioning method for twitter. The remaining sections are organized as follows: A review of the related works on data partitioning and replication on social network is provided in Section 1. Section 2 will present the proposed model in details. Section 3 presents the experiments and the results. Finally, section 4 concludes the paper.

2. RELATED WORKS
Several studies have been made for showing the problems related to partitioning and replication in social networks. In [7] a graph partitioning based database replication and partitioning scheme called SCHISM is proposed for OLTP type Web applications. The problem with this scheme is that it requires generation of larger graph from the transaction graph. Another disadvantage of this method is that it is not possible to set the amount of replication.

In [5], for managing large graphs a new method called sedge is proposed. For static primary partition it uses graph and complementary partitioning and then it uses workload-aware dynamic secondary partitions. [4] proposes graph partitioning based models for processing time-dependent social network queries. The problem with this approach is that it does not solves the replication problem.

In [5] and [20], data partitioning and replication strategy for social networks has been proposed. In [20] the WEPAR partitioning and replication system is proposed. The main idea

in WEPAR is that the master copies of related records are placed in the same node and then slave copies are generated for records that receive more read queries. [5] is extended to support selective replication and the algorithm generates placements with respect to the replication constraints. When a read miss occurs new replicas are added and removing replicas when a read is not performed for a while and a write occurs. It also uses temporal prediction of future request to avoid undesired operations.

Based on graph-partitioning, modular-optimization and random partitioning Pujol et al. [2] proposes social network partitioning schemes. The performance is measured via metrics such as the number of internal messages. For small partitions, graph-based approaches are shown to perform superior, whereas for large partition, modular optimization algorithms perform slightly better. Pujol et al. [3] extended the work in [2] to include replication. This scheme replicates all data items that are in partition boundaries. That is, data items that might be required in many servers are replicated to all of those servers. But this replication scheme creates too much replication which often lead to high I/O.

[19] focuses on the creation of feed pages, pages containing recent activities of followed/following. The activities of news-sources are broadcasted to many users and feed pages contain data collected from many news-sources. The proposed scheme overcome the disadvantages mentioned above by partitioning the users by considering the interactions and selectively replicating the users by considering the logs.

3. PROPOSED APPROACH

The input to the partitioning system is a database, representative workload, and the number of partitions that are desired. The output is a partitioning and replication strategy that balances the size of the partitions while minimizing the interaction with external partition.

The basic approach consists of following steps:

Data pre-processing: The system computes read and write sets for each request.

Creating the hypergraph: A node is created for each user. Edges represent the usage of user within a request. When a user login it requests the tweets from the users whom he follows. This request is modeled by connecting the users by a net. It also allows us to account for replicated users. For simplicity the cost of a net is assumed to be 1. Each user in the graph contains a state. The state of a user is A if it belongs to partition A or B if it is partition B or AB if it is replicated in both the partition. $\sigma A(n_j)$ denotes the number of users in partition A for the net n_j. $\sigma B(n_j)$ denotes the number of users in partition B for net n_j and $\sigma AB(n_j)$ denotes the number of users which are replicated in both the partitions A and B for the net n_j. A net is said to be internal if all the connections are in the same server. If a request requires data from multiple servers the net is said to be external. An example of a hypergraph is shown in fig 1. ui, uj, uk, up, uq ,un, um represents users. nj is the net.S1, S2, S3 are servers. When user ui requests its home page it interacts with the other users and gets the data from them. These request is connected by the net n_j.

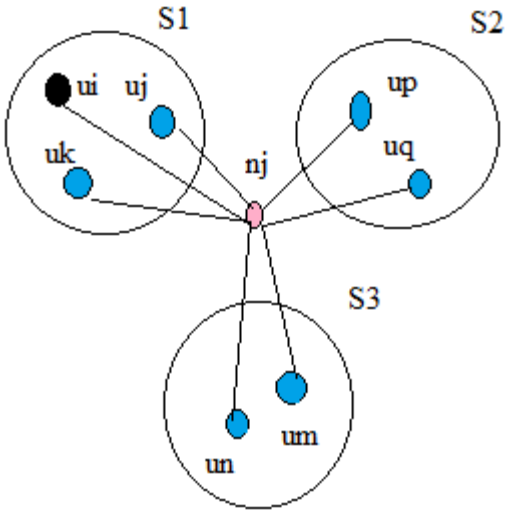

Fig 1: Hypergraph model

A graph partitioning algorithm is used to produce a balanced minimum-cut partitioning of the graph into k partitions. Each partition is assigned to one physical node. The replicated graph partitioning algorithm is described in section 2.3

3.1 Twitter on Cassandra

The proposed method is implemented by developing a twitter clone called Twissandra which works on Cassandra Nosql database. This twitter clone contains the basic functionalities of twitter. Twissandra data model consists of six column families: USER: Stores user information; key for each row is username and columns contain user details such as passwords, profile image url, banner image url. FOLLOWINGS: Stores the users that are followed by a user. FOLLOWERS: Stores the followers of a user; TWEET: Stores the tweets; HOMETIMELINE: Stores the tweets of a the user which a user follows; USERLINE: Stores all the tweets of a user;

Even if it is a twitter clone, it is possible to implement most of the existing functionalities in Twitter. The focus is on the operations performed when a user posts a tweet (which is propagated to his followers), and when a user checks his homepage (the latest tweets of his followings should be loaded). These operations involve multiple users which may be present on the same server or in another server. The former involves multi-write operations where as the latter requires multi-read operations.

Twissandra is designed such that, both multi-reads and multi-writes require multi-way interactions. This is because the actual tweet data is stored only in the servers where the tweeting user is stored, and it is not stored in follower HOMETIMELINEs. Replicating actual tweet data on all followers can be expensive as it contains large pieces of data such as videos or pictures.

3.2 Two Way Replicated Partitioning

In this the users are randomly partitioned into two partitions. Some of the users may be replicated. These uses are selected randomly or the users with higher number of followers can also be chosen. Even though the initial bipartition is random it is giving a good initial distribution to the partitioning problem. These partitions are further improved by performing three different operations such as move, replicate and unreplicate.

move gain (gm(u_i)): the reduction to be observed in the overall query processing cost, if user u_i is moved to the other partition,

replication gain (gr(u_i)): the reduction to be observed in the overall query processing cost, if user u_i is replicated to the other partition,

unreplication-from A gain (gu,A(u_i)): the reduction to be observed in the overall query processing cost, if a replica of user u_i is deleted from partition A , P^A,

unreplication-from B gain (gu,B(u_i)): the reduction to be observed in the overall query processing cost, if a replica of user u_i is deleted from partition B,P^B

Unreplication gains are only meaningful for users that are replicated. Similarly, in a two-way partition, move, and replication gains are only meaningful for users that are not replicated. Thus, for any user, only two gain values need to be maintained.

The overall two-way replicated partitioning works as a sequence of two-way refinement passes performed over all users. In each pass, the initial operation gains of all users are computed. Then iteratively perform the following computations: find the user and the operation that produces the highest reduction in the cost; perform that operation; update gain values of neighboring user; lock the selected user to further processing to prevent thrashing. These computations are performed until there are no remaining users to process. This process is repeated until the state where the best reduction is obtained during the pass or if the obtained improvement in the current pass is above a threshold or if the number of passes performed is below some predetermined number. After obtaining a two-way partition, the two-way partitioning algorithm is recursively applied on each of these partitions to obtain any number of partitions.

After initializing the gains, retrieve the highest gains and the associated user for each operation type and by comparing these gains select the best operation to perform. If there are any possible unreplication operations which do not increase the total cost of the system (i.e., with zero unreplication gain), those unreplication operations are performed first. After finishing possible unreplications, compare the gains to be obtained by move and replication operations. If the gains are the same, perform move operations.

After performing an operation (move, replication, or unreplication) on a user, update the gains of operations related with the users that are neighbors of the given user.

3.3 Algorithms

In this section,detailed explanations of the algorithms used in replicated data partitioning is presented. For that a temporal activety hypergraph is constructed. This hypergraph contains nets which represents the interaction between users when a user requests tweets.The input to the algorithms is this temporal activity hypergraph.

2.3.1 Initial gain computation.

The initial gain computation consists of two main loops. The first loop computes the initial gain values by traversing users and the second loop updates the initialization of gain values by traversing all nets. The move and replication gains are computed according to the nets that connect these users which are critical and external, whereas the unreplication gains are modified according to the internal and critical nets that connect these users. The move and replication gains of the non-replicated users are initially set to their minimum possible values (lines 3–4). If a net n_j is external and move critical or replication critical, the move and replication gains of the users connected by n_j must be incremented by c(n_j) (lines 12–13), since it can be saved from the cut with either one of these operations. In contrast to move and replication gains, unreplication gains are initially set to their maximum possible values (lines 6–7). If a net n_j is internal and thus unreplication critical, the unreplication gains of the replicas of the replicated users connected by n_j may need to be updated. If n_j connects at least one non-replicated users that is in the same part with this net then the unreplication gains of the replicas that are in the same partition with the internal net should be decremente by c(n_j) (lines 14–18).

Algorithm 1: Initial move, replication, and unreplication gain computation.

H is the hypergraph and Π^R is the replicated partitioning of the hypergraph. P^A,P^B are the partitions. U is the set of users N is the set of nets.

Input: H = (U,N), Π^R = {P^A,P^B}

1 for each $u_i \in$ U do

2 if State(u_i) $\neq$ AB then

3 gm(u_i) $\leftarrow -$c(InternalNets(u_i))

4 gr (u_i) $\leftarrow$ 0

5 else

6 gu,A(u_i) $\leftarrow$ 0

7 gu,B(u_i) $\leftarrow$ 0

8 for each $n_j \in$ N do

9 for each $u_i \in$ Users(n_j) do

10 if State(u_i) $\neq$ AB and n_j is external then

11 if ($\sigma A(n_j)$) = 1 and State(u_i) = A) or ($\sigma B(n_j)$) = 1 and State(u_i) = B) then n_j is critical to P^A or P^B

12 gm(u_i) ← gm(u_i) + c(n_j)

13 gr (u_i) ← gr (u_i) + c(n_j)

14 else if State(u_i) = AB and n_j is internal then

15 if $\sigma A(n_j) > 0$ and $\sigma B(n_j) = 0$ then n_j is critical to P^A

16 gu,A(u_i) ← gu,A(u_i) − c(n_j)

17 else if $\sigma B(n_j) > 0$ and $\sigma A(n_j) = 0$ then n_j is critical to P^B

18 gu,B(u_i) ← gu,B(u_i) − c(n_j)

2.3.2 Gain updates after a move operation

Algorithm 2: Gain updates after moving u∗ from P^A to P^B.

Input: H = (U,N), Π^R = { P^A, P^B }, u∗ ∈ P^B

1 State(u∗) ← B

2 Lock u∗

3 for each n_j ∈ Nets(u∗) do

4 $\sigma A(n_j) \leftarrow \sigma A(n_j) - 1$

5 if $\sigma A(n_j) = 0$ then n_j becomes critical to P^B

6 for each unlocked u_i ∈ Users(n_j) do

7 if State(u_i) = B then

8 gm(u_i) ← gm(u_i) − c(n_j)

9 else if State(u_i) = AB then

10 gu,B(u_i) ← gu,B(u_i) − c(n_j)

11 else if $\sigma A(n_j) = 1$ then n_j becomes critical to P^A

12 for each unlocked u_i ∈ Users(n_j) do

13 if State(u_i) = A then

14 gm(u_i) ← gm(u_i) + c(n_j)

15 gr (u_i) ← gr (u_i) + c(n_j)

16 $\sigma B(n_j) \leftarrow \sigma B(n_j) + 1$

17 if $\sigma B(n_j) = 1$ then n_j was critical to P^A

18 for each unlocked u_i ∈ Users(n_j) do

19 if State(u_i) = A then

20 gm(u_i) ← gm(u_i) + c(n_j)

21 else if State(u_i) = AB then

22 gu,A(u_i) ← gu,A(u_i) + c(n_j)

23 else if $\sigma B(n_j) = 2$ then n_j was critical to P^B

24 for each unlocked u_i ∈ Users(n_j) do

25 if State(u_i) = B then

26 gm(u_i) ← gm(u_i) − c(n_j)

27 gr (u_i) ← gr (u_i) − c(n_j)

Algorithm 2 shows the procedure for performing gain updates after moving a given user u∗ from P^A to P^B. The algorithm includes updating fields of u∗ (lines 1–2), the users of Nets (u∗) (lines 4 and 16), and the gain values of neighbors of u∗ (lines 5–15 and 17–27). The necessary field updates on u∗ are performed by updating the state and locked fields of u∗ to reflect the move operation. The users of each net n_j ∈ Nets (u∗) needs to be updated by decrementing $\sigma A(n_j)$ by 1 and incrementing $\sigma B(n_j)$ by 1. When the users of n_j changes, its criticality may change. The change in the criticality of n_j may require various gain updates on the unlocked users connected by n_j.

After decrementing the number of users of n_j in P^A (line 4), check the value of $\sigma A(n_j)$ to see if the criticality of n_j has changed (lines 5 and 11). If $\sigma A(n_j) = 0$, n_j becomes internal to P^B by becoming move critical and unreplication critical to this part, and if $\sigma A(n_j) = 1$, n_j becomes move critical and replication critical to P^A.

Similarly, after incrementing the number of users connected by n_j in P^B (line 16), check the value of $\sigma B(n_j)$ to see if the criticality of n_j has changed (lines 17 and 23). If $\sigma B(n_j) = 1$, it means that n_j was internal and hence was move critical and unreplication critical to P^A, and if $\sigma B(n_j) = 2$, it means that n_j was move critical and replication critical to P^B. Under these conditions for n_j, the

gains of the users connected by n_j should be checked for any update with respect to the corresponding part.

2.3.3 Gain updates after a replication operation

Algorithm 3: Gain updates after replicating u∗ from P^A to P^B.

Input: H = (U,N), Π^R = { P^A, P^B }, u∗ ∈ P^B

1 State(u∗) ← AB

2 Lock u∗

3 for each n_j ∈ Nets(u∗) do

4 $\sigma A(n_j)$) ← $\sigma A(n_j)$ − 1

5 $\sigma B(n_j)$ ← $\sigma B(n_j)$ + 1

6 if $\sigma A(n_j)$ = 0 then n_j becomes critical to P^B

7 for each unlocked u_i ∈ Users(n_j) do

8 if State(u_i) = B then

9 gm(u_i) ← gm(u_i) − c(n_j)

10 if $\sigma B(n_j)$ = 1 then

11 gr (u_i) ← gr (u_i) − c(n_j)

12 else if State(u_i) = AB then

13 if $\sigma B(n_j)$ = 0 then

14 gu,A(u_i) ← gu,A(u_i) + c(n_j)

15 else if $\sigma B(n_j)$ > 0 then

16 gu,B(u_i) ← gu,B(u_i) − c(n_j)

17 else if $\sigma A(n_j)$ = 1 then n_j becomes critical to P^A

18 for each unlocked u_i ∈ Users(n_j) do

19 if State(u_i) = A then

20 gm(u_i) ← gm(u_i) + c(n_j)

21 if $\sigma B(n_j)$ > 0 then

22 gr (u_i) ← gr (u_i) + c(n_j)

Algorithm 3 shows the procedure for performing gain updates after replicating a given user u∗ from P^A to P^B. The procedure starts with changing the state of u∗ to AB and locking both replicas of u∗ (lines 1–2). Then, for each net n_j that connects u∗, the users of n_j are updated and checked for criticality condition changes (lines 6 and 17). Since u∗ was in P^A before replication, $\sigma A(n_j)$) is decremented by 1 and $\sigma AB(n_j)$ is incremented by 1 to reflect that u∗ is now a replicated user (lines 4–5). The replication of u∗ from P^A does not change the $\sigma B(n_j)$ value of any n_j ∈ Nets (u∗); thus the criticality conditions that include $\sigma B(n_j)$ need not be checked.

After the value of $\sigma A(n_j)$ is decremented (line 4), n_j must be checked for criticality condition changes to see if there are any necessary gain updates for the neighbors of u∗ (lines 6 and 17). If $\sigma A(n_j)$ = 0, n_j becomes move critical and unreplication critical to P^B. In this condition, the move gains of the unlocked users and the unreplication gains of the unlocked replicas that are connected by n_j need to be decremented by c(n_j) since n_j is internal now, and the move of any user or the unreplication of any replica connected by n_j would bring it to cut. If $\sigma A(n_j)$ = 1, n_j becomes move critical and replication critical to P^A. The move or the replication of the only non-replicated user u_i connected by n_j in P^A can now save n_j from the cut, and thus the move and replication gains of this user must be incremented by c(n_j).

2.3.4 Gain updates after an unreplication operation

Algorithm 4: Gain updates after unreplicating u* from P^A.

Input: H = (U,N), Π^R = { P^A, P^B }, u* ∈ P^B

1 State(u*) ← B

2 Lock u*

3 For each n_j ∈ Nets(u*) do

4 $\sigma B(n_j)$ ← $\sigma B(n_j)$ + 1

5 $\sigma AB(n_j$ ← $\sigma AB(n_j$ − 1

6 if $\sigma B(n_j)$ = 1 then n_j was critical to P^A

7 for each unlocked u_i ∈ Users(n_j) do

8 if State(u_i) = A then

9 gm(u_i) ← gm(u_i) + c(n_j)

10 if $\sigma A(n_j)$ = 1 then

11 gr (u_i) ← gr (u_i) + c(n_j)

12 else if State(u_i) = AB then

13 if $\sigma A(n_j)$ = 0 then

14 gu,B(u_i) ← gu,B(u_i) − c(n_j)

15 else if $\sigma A(n_j)$ > 0 then

16 gu,A(u_i) ← gu,A(u_i) + c(n_j)

17 else if $\sigma B(n_j)$ = 2 then $\in n_j$ was critical to P^B

18 foreach unlocked $u_i \in$ Users (n_j) do

19 if State(u_i) = B then

20 gm(u_i) ← gm(u_i) − c(n_j)

21 if $\sigma A(n_j)$ > 0 then

22 gr (u_i) ← gr (u_i) − c(n_j)

Algorithm 4 shows the procedure for performing updates after unreplication of a given replica u∗ from P^A. The procedure starts with changing the state of u∗ to B and locking it (lines 1–2). Then, for each net n_j that connects u∗, the pin distributions of n_j are updated and checked for criticality condition changes (lines 6 and 17). Since u∗ was a replicated user before unreplication from P^A, $\sigma B(n_j)$ is incremented by 1 and $\sigma AB(n_j)$ is decremented by 1 to reflect that u∗ is now a non replicated users in P^B (lines 4–5). The unreplication of u∗ from P^A does not change the $\sigma A(n_j)$ value of any $n_j \in$ Nets (u∗); thus the criticality conditions that include $\sigma A(n_j)$ need not be checked. After the value of $\sigma B(n_j)$ is incremented (line 4), n_j must be checked for criticality condition changes to see if there are any necessary gain updates for the neighbors of u∗ (lines 6 and 17). If $\sigma B(n_j)$ = 1, it means that n_j was move critical and unreplication critical to P^A. In this case, the move and replication gains of the unlocked users and replicas that are in P^A and connected by n_j are incremented by c(n_j), since n_j is not an internal net anymore.

If $\sigma B(n_j)$ = 2, it means that n_j was move critical and replication critical to P^B. The net n_j connects two users in P^B and one of them, u*, is already locked, and thus the move and replication gains of the other user, u_i, need to be decremented by c(n_j), since this users can no longer save n_j from the cut.

4. EXPERIMENTS AND RESULTS

Experiments were conducted by obtaining the real world data from twitter. Temporal activity hypergraph of the obtained data is constructed. .The hypergraph is partitioned by selectively replicating the users. The number of partition is set to 4. The maximum replication ration is set to 0.5.

The selectively replicated partition of a few users whose data has been collected is shown in fig 2 . Each partition contains the followers and the followings of a user. Only a few users will be present in another server. As the number of servers required to process a request is small a considerable reduction in the processing time of a request is obtained.

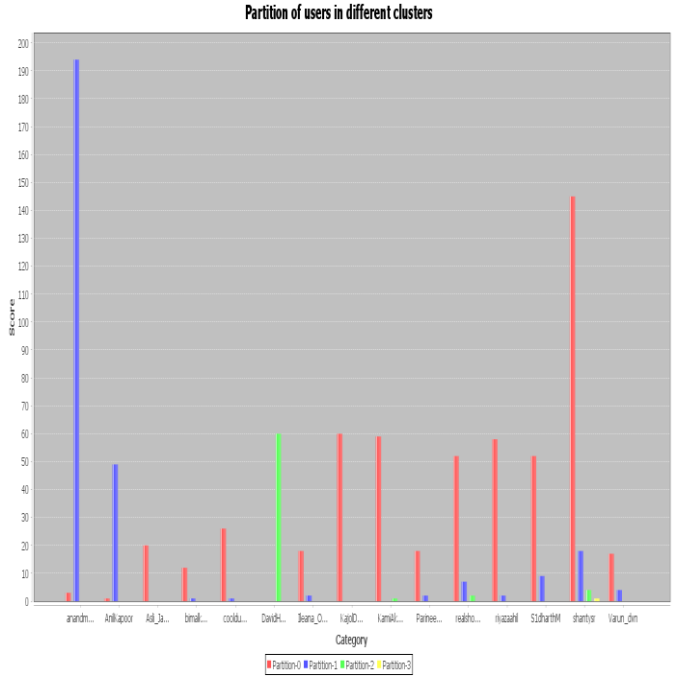

Fig 2: User partition

The proposed method is compared with hash based partition and random replication. Experimental analysis shows that a considerable reduction in the processing time is observed by using selective replication.

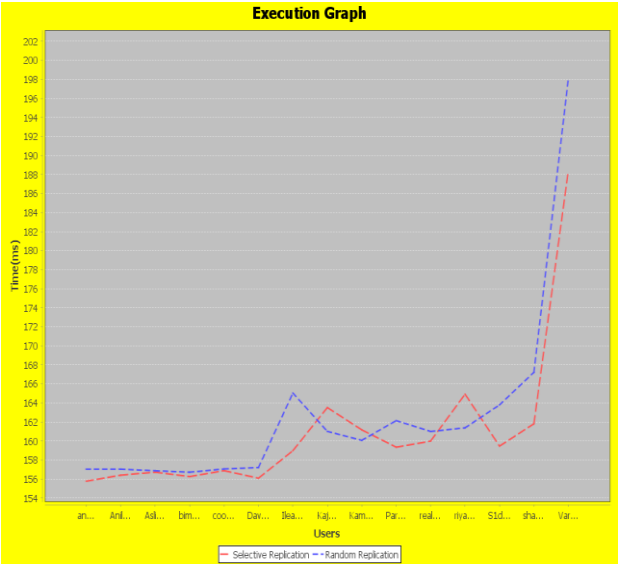

Fig 3: Comparison with hash based replicated partitioning

The following table shows the number of read and write requests issued in different partitioning schemes.

Table 1. Read Write Comparison

Partition Technique	**Average number of reads**	**Average number of writes**
Random Partition and random replication	75	23
Hash based partition and two hop replication	60	25

While comparing the balancing performance of these schemes it is observed that random replication and random partitioning have the worst balancing performance for both read and write requests. Hash based replication and two hop replication has comparatively better performance than random replication and random partitioning but poor performance than selective replicated partitioning. This is because the other replication schemes does not take the balancing constraint into account during replication On the other hand, RHP can simultaneously perform objective optimization and balancing under replication in a single replicated partitioning phase and thus has superior read and write balancing performance. Selectively replicated partitioning strikes a balance on the performance metrics by trading locality with load balancing and I/O load minimization, which leads to its superior query processing performance.

5. CONCLUSION

In this work, a selectively replicated partitioning using temporal activity hypergraph is proposed for data partitioning and replication in twitter. The proposed model uses multi-way interactions incurred by the read and write operations in twitter.The users are randomly partitioned into two partitions and the partitions are further improved by performing move, replication and unreplication operations. The process is performed recursively until the desired number of partition is obtained.

Hash-based approaches distribute workload and enhance parallelism but suffer from communication overhead. Graph-partitioning-based approaches enhance read locality at the expense of increasing I/O loads and load balance. This approach performs partitioning and replication simultaneously and reduce the number of servers required to process a request with a limit amount of replication.

6. REFERENCES

[1] Ata Turk, R. Oguz Selvitopi, Hakan Ferhatosmanoglu, and Cevdet Aykanat, "Temporal Workload-Aware Replicated Partitioning for Social Networks," IEEE transactions on knowledge and data engineering, vol. 26, no. 11, november 2014

[2] R. Hecht and S. Jablonski, "NoSQL Evaluation: A Use Case Oriented Survey," Proc. Int'l Conf. Cloud and Service Computing (CSC),pp. 336-341, Dec. 2011.

[3] J.M. Pujol, G. Siganos, V. Erramilli, and P. Rodriguez, "Scaling Online Social Networks Without Pains," Proc. Fifth Int'l Workshop Networking Meets Databases (NeTDB), 2009.

[4] M. Yuan, D. Stein, B. Carrasco, J.M. F. da Trindade, and Y. Lu,"Partitioning Social Networks for Fast Retrieval of Time-Dependent Queries," Proc. IEEE 28th Int'l Conf. Data Eng. Workshop. t10.1145/2213836.2213895, 2012.

[5] J.M. Pujol, V. Erramilli, G. Siganos, X. Yang, N. Laoutaris, P. Chhabra, and P. Rodriguez, "The Little Engine (s) that Could: Scaling Online Social Networks," ACM SIGCOMM Computer Comm. Rev., vol. 40, no. 4, pp. 375-386, 2010.

[6] C.Curino, E. Jones, Y. Zhang, and S. Madden, "Schism: A Workload-Driven Approach to Database Replication and Partitioning," Proc. VLDB Endowment, vol. 3, no. 1-2, pp. 48-57.

[7] A. Lakshman and P. Malik, "Cassandra: A Decentralized Structure Storage System," SIGOPS Operating System Rev., vol. 44, no. 2, pp. 35-40,Apr. 2010.

[8] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazons Highly Available Key-Value Store.," Proc. 21st ACM SIGOPS Symp. Operating Systems Principles, pp. 205-220, 2007.

[9] O.R.M. Thomae, "Database Partitioning Strategies for Social Network Data," master's thesis, Massachusetts Inst. of Technology, 2012.

[10] G. Karypis and V. Kumar, "Multilevel k-Way Hypergraph Partitioning," Proc. ACM/IEEE 36th Ann. Design Automation Conf. pp. 343-348, 1999.

[11] U.V. Atalyurek and C. Aykanat, "PaToH: A Multilevel Hypergraph Partitioning Tool, Version 3.0," technical report, Dept. of Computer Eng., Bilkent Univ., 1999.

[12] U. Catalyurek and C. Aykanat, "Hypergraph-Partitioning-Based Decomposition for Parallel Sparse-Matrix Vector Multiplication," IEEE Trans. Parallel and Distributed System, vol. 10, no. 7, pp. 673- 693,2010.

[13] R.O. Selvitopi, A. Turk, and C. Aykanat, "Replicated Partitioning for Undirected Hypergraphs," J. Parallel and Distributed Computing, vol. 72, no. 4, pp. 547-563. j.jpdc.2012.01.004, Apr. 2012.

[14] D.S. Johnson, "Approximation Algorithms for Combinatorial Problems," Proc. ACM Fifth Ann.Symp. Theory of Computing (STOC '73), pp. 38-49.

[15] Y. Qiu-yan, "A Novel Time Streams Prediction Approach Based on Exponential Smoothing," Proc. Second Int'l Conf. MultiMedia and Information Technology (MMIT '10), pp. 20-23, 2010.

[16] M. De Choudhury, Y.-R. Lin, H. Sundaram, K.S. Candan, L. Xie, and A. Kelliher, "How Does the Data Sampling Strategy Impact the Discovery of Information Diffusion in Social Media?" Proc. Fourth Int'l AAAI Conf. Weblogs and Social Media, 2010.

[17] G. Karypis and V. Kumar, "Metis—Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 2.0," technical report, Dept. of Computer Science and Eng., Univ. of Minnesota, 1995.

[18] A. Tatarowicz, C. Curino, E. Jones, and S. Madden, "Lookup Tables: Fine-Grained Partitioning for Distributed Databases," Proc. IEEE 28th Int'l Conf. Data Eng. (ICDE), pp. 102-113, Apr. 2012.

[19] A. Silberstein, J. Terrace, B.F. Cooper, and R. Ramakrishnan, "Feeding Frenzy: Selectively Materializing Users' Event Feeds," Proc. ACM SIGMOD Int'l Conf. Management of Data, pp. 831-842, 2010.

[20] Y. Huang, Q. Deng, and Y. Zhu, "Differentiating Your Friends for Scaling Online Social Networks," Proc. IEEE Int'l Conf. ClusterComputing (CLUSTER), pp. 411-419, Sept. 2012.