Temporal Workload-Aware Replicated Partitioning for Social Networks

Ata Turk, R. Oguz Selvitopi, Hakan Ferhatosmanoglu, and Cevdet Aykanat

Abstract—Most frequent and expensive queries in social networks involve multi-user operations such as requesting the latest tweets or news-feeds of friends. The performance of such queries are heavily dependent on the data partitioning and replication methodologies adopted by the underlying systems. Existing solutions for data distribution in these systems involve hash- or graph-based approaches that ignore the multi-way relations among data. In this work, we propose a novel data partitioning and selective replication method that utilizes the temporal information in prior workloads to predict future query patterns. Our method utilizes the social network structure and the temporality of the interactions among its users to construct a hypergraph that correctly models multi-user operations. It then performs simultaneous partitioning and replication of this hypergraph to reduce the query span while respecting load balance and I/O load constraints under replication. To test our model, we enhance the Cassandra NoSQL system to support selective replication and we implement a social network application (a Twitter clone) utilizing our enhanced Cassandra. We conduct experiments on a cloud computing environment (Amazon EC2) to test the developed systems. Comparison of the proposed method with hash- and enhanced graph-based schemes indicate that it significantly improves latency and throughput.

Index Terms—Cassandra, social network partitioning, selective replication, replicated hypergraph partitioning, twitter, NoSQL

1 Introduction

Social networks have fast-growing, ever-changing dynamic structures and strict availability requirements. These challenges are partially handled by emerging solutions such as NoSQL (Not Only SQL) systems [1], which use data partitioning and replication to achieve scalability and availability. In these systems, the general approach is to use hash-based partitioning and random replication of data. This approach ignores the relations among the data and often leads to redundant replications and significant communication overheads during query processing, which in turn leads to performance degradation [2], [3], [4], [5].

Recently, a number of approaches based on modeling the social network structure and user interactions have been proposed [4], [6], [7] to alleviate the shortcomings of hash-based partitioning and random replication schemes. These approaches try to capture the interactions between social network users via two-way relations, e.g., edges in graphs. On the other hand, most common social network operations such as propagating a user's tweets to all of his followers, requesting the latest tweets of followed users, collecting the latest news-feeds of Facebook friends, or sharing/commenting/liking a news-feed are all expand-/gather-like operations that require multi-casting/gathering of data to/from multiple users in a single operation. These popular social network operations generally tend to be more expensive than operations that only involve bilateral interactions. In this study, we claim and show that hypergraphs are more suitable for modeling these multi-user operations, since they can inherently model multi-way interactions via hyperedges. We also show that performing partitioning and replication in a single phase enables more accurate cost prediction and better load balancing.

There are a number of problems with graph-partitioning-based models and we adress these problems using our replicated hypergraph partitioning (RHP) model. We first present a simple example in Fig. 1 to illustrate why hypergraphs are more suitable for capturing multi-user operations. In this example, user u_i tweets and this tweet is propagated to his followers. This can be done by communicating the tweet data to servers S_1, S_2, and S_3. The graph representation of this operation in Fig. 1a observes a cut of six, wrongly assuming that u_i's tweet has to be sent to S_2 and S_3 three times each, where only a single message suffices for each server. As shown in Fig. 1b, hypergraph model captures this operation's cost via a net with a connectivity of three, correctly modeling the number of activated servers (or the span).

More importantly, graph model cannot accurately model the effect of replication on the span of multi-user operations. For example, in Fig. 1, replicating u_p from S_3 to S_1 does not reduce the span of the query, but it removes an edge in the graph model, wrongly giving an impression of improvement. To alleviate this deficiency, one-hop replication schemes are proposed [2], [3]. In these schemes, by replicating all boundary vertices, the span of all queries are reduced to one, but in turn, overheads associated with write operations increase severely.

It is not vital to enforce the span of all queries to one. Solutions that benefit from reasonable span reductions with low update overheads can provide better system performance. Such solutions can be obtained by performing partitioning and replication decisions at the same time in

- A. Turk is with Yahoo Labs, Barcelona. E-mail: ata@yahoo-inc.com.
- R.O. Selvitopi, H. Ferhatosmanoglu, and C. Aykanat are with Bilkent University. E-mail: {reha, hakan, aykanat}@cs.bilkent.edu.tr.

Manuscript received 17 Apr. 2013; revised 28 Nov. 2013; accepted 10 Jan. 2014. Date of publication 22 Jan. 2014; date of current version 26 Sept. 2014. Recommended for acceptance by Y. Tao.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TKDE.2014.2302291

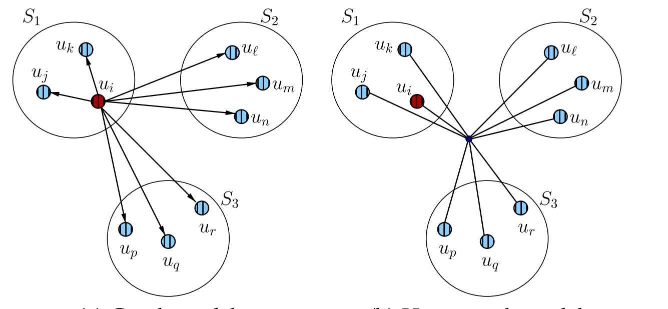


Fig. 1. User u_i's tweet is propogated to his followers. (a) Graph model and (b) Hypergraph model for this operation.

harmony. In fact, studies like [7] try to achieve this by augmenting graph models. However, the deficiency of graph models in representing multi-way relations hinders their effectiveness in exploring replication solutions as well. For example, in Fig. 1, replicating u_p, u_q, u_r from S_3 to S_2 does reduce the span of the query by one and hypergraph model can correctly observe this reduction, whereas, the graph model cannot foresee to perform such a replication, since it is not aware of the relation between S_2 and S_3 due to this query. If partitioning and replication are performed in separate phases, load balancing efforts made during partitioning can be futile, since based on replica selection decisions, certain servers can be highly loaded. To alleviate such problems, we consider a close-to-optimal query scheduling algorithm while performing replicated partitioning, and hence observe true balancing and cost estimations at the end of our replicated partitioning scheme.

Query processing performance of social networks has a direct influence on their success. In our empirical analysis, we observe that (i) server load imbalance, (ii) the total number of I/O operations (read and write operations), and (iii) the number of servers processing a query (query span), have direct correlation with the performance of the system. Thus, we focus on these metrics for possible improvements in query performance.

In this work, we propose a selective partitioning and replication method for data distribution in social networks by utilizing the workload and time information. Our method uses a novel hypergraph model (called the temporal activity hypergraph model) to represent the social network structure and interactions among its users. This model values the time of interactions between users and predicts the interactions that are likely to occur in the near future. We show that simultaneous partitioning and replication (replicated partitioning) of this hypergraph model can accurately capture the objective of reducing the span of multi-user queries, subject to load balance and replication constraints. After performing a replicated partitioning of this hypergraph model, we decode the obtained result as a data-to-server mapping. This scheme greatly reduces the average query span while balancing the server loads. It also limits the amount of increase in I/O load due to replications by respecting to a user-provided threshold on the replication amount and by performing selective replication.

To test the proposed data distribution method, we first introduced selective replication capabilities to Cassandra NoSQL system [8]. Then we implemented a Twitter clone via adapting the Twissandra project [9] to make use of this enhanced system. We tested our Twitter clone on Amazon EC2 cluster under social network loads derived from actual Twitter data (including connections and interactions over time) to show the benefits of the proposed data distribution method. Even though we validate our approach on a Twitter-like system, the proposed method is applicable to other social network applications that frequently utilize multi-user operations.

The rest of the paper is organized as follows. Section 2 presents a background on technologies used in social networks and replicated hypergraph partitioning. Motivating insights and problem definition are presented in Section 3. Section 4 presents and discusses the proposed temporal activity hypergraph model. Replicated partitioning of the proposed hypergraph model is discussed in Section 5. In Section 6, we compare the proposed approach against the state-of-the-art approaches. Section 7 covers related studies. Finally we conclude in Section 8.

2 Background and System Architecture

2.1 Partitioning and Replication in NoSQL Systems

Most NoSQL systems use either hash-based or range-based (or a blend of the two) partitioning schemes. In range-based partitioning, the keyspace is divided into ranges and each range is assigned to a server and potentially replicated to others. The main advantage of range-partitioning is that two consecutive keys are likely to appear in the same partition, which is beneficial when range scan type queries are frequent. Range-based partitioning schemes generally maintain a map that stores information about which servers are responsible for which key ranges. Hash-based partitioning simply uses the hash of data to determine the responsible server for storing that data. Consistent hash rings are a blend of range- and hash-based partitioning schemes and many NoSQL systems such as Cassandra [8], Dynamo [10], Voldemort [11], and Riak [12] adopt this scheme.

2.1.1 Partitioning and Replication in Cassandra

Servers in a Cassandra cluster can be considered to be located around a ring and the data stored is distributed according to this ring analogy. The ring is divided into ranges and each server is responsible for one or more ranges. When a new server joins Cassandra, it is assigned a new token, which determines its position on the ring and the range of data it is responsible for. Column family (cf) data is partitioned across servers based on the row key (horizontal partitioning). Each server is responsible for the region of the ring between itself and its predecessor in token order.

There are two basic partitioning strategies employed in Cassandra: *Random Partitioning*, which partitions the data using the MD5 hash of each row key, and *Ordered Partitioning*, which stores the row keys in sorted order across servers. Once a Cassandra cluster is initialized with a partitioning strategy, this strategy cannot be changed without reloading all the data to the system. When a data is inserted and assigned a row key, a copy of this data is replicated for a fixed number of times (replication factor) across servers based on the preferred replication strategy. Default replication strategy in Cassandra is *RackUnawareStrategy*, which places the original data on the server determined by the partitioner. Additional replicas are placed on the following servers in the ring with respect to token order.

2.1.2 Writes, Reads, and Consistency in Cassandra

Cassandra is designed for providing fast and available writes. A write is first inserted into a commit log for durability, then to an in-memory table structure called *memtable*, and then it is acknowledged as successful. Periodically, the writes collected in the memory are dumped to the disk in a format called *SSTable* (sorted string table). SSTables are immutable and thus different columns of the same row can be stored in different SSTables. Due to this fast write mechanism, reads in Cassandra are costlier. When a read for a row is issued to Cassandra, the row must be collected from the unflushed memtables and the SSTables containing the columns of the requested row. In background, Cassandra periodically merges SSTables to form larger SSTables. During this merge process, row fragments are collected together and deleted columns are removed.

In a distributed system with replicas, consistency issues arise in realizing write and read operations. Cassandra is eventually consistent, i.e., in sufficient time, all writes are applied to all replicas making all replicas eventually consistent. The write consistency level specifies the number of replicas a write must succeed before returning an acknowledgement. Similarly, the read consistency level specifies the minimum number of replicas for which the result of the read must be agreed upon before generating a response. Write/read consistency levels can be determined according to the sensitivity of the used application to reading stale data and its need for fast query processing.

2.1.3 Twitter on Cassandra

To test our proposed method and enhanced Cassandra system on a real-world application, we modified Twissandra [9], a project that provides a fully-working Twitter clone. In Twissandra, the data is stored in Cassandra and in terms of data partitioning and replication decisions, scaling Twissandra carries most fundamental problems observed in scaling 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, gender, phone, email, etc. `FRIENDS`: Stores the users that are followed by a user (friends); key for each row is the username and columns are the usernames of the friends, which are the users followed by the user in the row key. `FOLLOWERS`: Stores the followers of a user; key for each row is the username and columns are the usernames of the users that follow the user in the row key. `TWEET`: Stores the tweets; key for each row is a unique tweet ID and columns are the tweet body and the username of the tweeting user. `TIMELINE`: Stores the tweets of a user's friends; key for each row is the username, column names are time stamps, and column values are tweet IDs. `USERLINE`: Stores all the IDs of a given user's tweets; key for each row is the username, column names are time stamps and column values are tweet IDs.

Using this data model, it is possible to implement most of the existing functionalities in Twitter. We mainly investigate the operations performed when a user tweets (which is propagated to his followers), and when a user checks his homepage for the latest tweets of his friends (here, a "friend" is somebody that a user follows). These are multi-user operations. The former operation is a multi-write request (also referred as a write request) and requires (i) insertion of a tweet to the `TWEET` cf, (ii) addition of the unique tweet ID into the `USERLINE` cf of the tweeting user, and (iii) addition of the unique tweet ID into the `TIMELINE` column families of the followers of the tweeting user. The latter operation is a multi-read request (also referred as a read request) and requires (i) a lookup for the latest tweet IDs in a user's respective row at the `TIMELINE` cf and then (ii) the retrieval of the tweets for those tweet IDs from the `TWEET` cf. We choose to model these operations since they are representative of the most frequent multi-user operations in social networks [13]. Note that a read request and a write request consist of a set of individual read and write operations, respectively.

Our Cassandra-based Twitter-clone is designed such that, both multi-reads and multi-writes require multi-way interactions. This is because, in our data model we assume that actual tweet data is only stored in the servers where the tweeting user is stored, and it is not replicated in follower `TIMELINE`s. There can be other application implementations which choose to do only "pushes" (by storing the actual tweet data instead of the tweet id on all followers' `TIMELINE`s) or only "pulls" (by not storing tweet id's at all on followers' `TIMELINE`s, which would necessitate a sorting and selection for finding recent tweets). We opted to use a "mixed" strategy since, first we believe that replicating actual tweet data (or feed data) on all followers can be quite expensive (considering tweets or feeds may contain large pieces of data such as pictures or videos), and second we want to show that our approach can capture the underlying interactions for both pull- and push-based applications.

2.2 Hypergraph Partitioning (HP) and Replication

A hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ is defined as a set $\mathcal{V}$ of vertices and a set $\mathcal{N}$ of nets (hyperedges), where each net connects a number of distinct vertices. The vertices connected by a net n_j are said to be its pins ($\mathrm{Pins}(n_j)$). A cost $c(n_j)$ and a weight $w(v_i)$ may be associated with a net $n_j \in \mathcal{N}$ and a vertex $v_i \in \mathcal{V}$, respectively.

$\Pi = \{\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_K\}$ is said to be a K-way partition of a given hypergraph $\mathcal{H}$, if parts are mutually disjoint and collectively exhaustive. In Π, a net is said to connect a part if it has at least one pin in that part. Connectivity set $\Lambda(n_j)$ of a net n_j is the set of parts connected by n_j:

$$\Lambda(n_j) = \{\mathcal{V}_k : \mathcal{V}_k \cap Pins(n_j) \neq \emptyset\}. \tag{1}$$

The connectivity $\lambda(n_j) = |\Lambda(n_j)|$ of a net n_j is the number of parts connected by n_j. A net n_j is said to be cut if $\lambda(n_j) > 1$ and uncut otherwise.

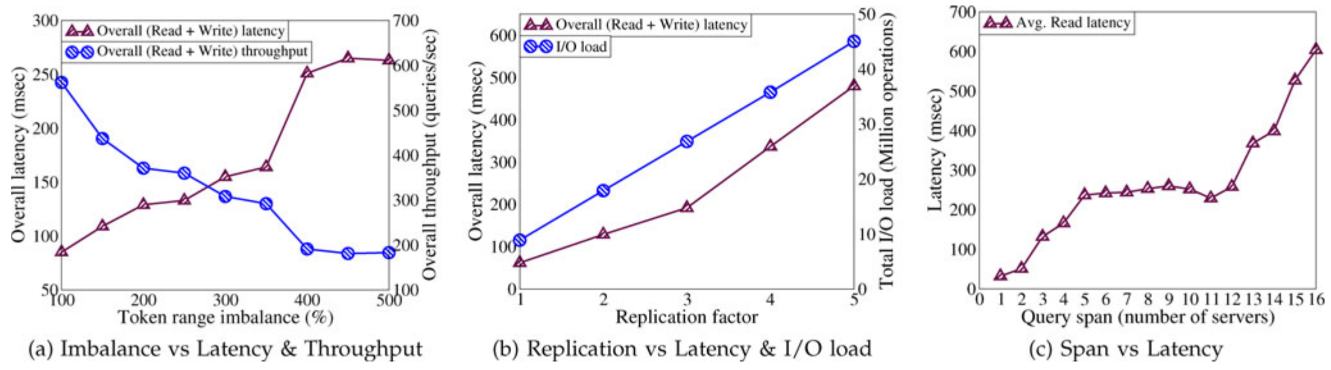


Fig. 2. Investigated metrics and their effects on latency.

In the *Hypergraph Partitioning Problem*, given a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ and an imbalance ratio ϵ, we want to find a K-way vertex partition Π of $\mathcal{V}$ that optimizes a partitioning objective defined over the nets, while satisfying a given partitioning constraint. The partitioning constraint is to maintain the balance criteria on part weights, i.e., $W(\mathcal{V}_k) \leq W_{\text{avg}}(1+\epsilon)$, for $k = 1, \ldots, K$. Here, the weight $W(\mathcal{V}_k)$ of a part $\mathcal{V}_k$ is defined as the sum of the weights $w(v_i)$ of the vertices in $\mathcal{V}_k$, W_{avg} is the average part weight ($W_{avg} = W(\mathcal{V})/K$), and ϵ is the maximum allowed imbalance ratio.

In the HP problem, the partitioning objective is to minimize the cutsize based on the connectivity metric

$$\chi(\Pi) = \sum_{n_j \in \mathcal{N}} c(n_j)\lambda(n_j), \tag{2}$$

defined over the set of nets $\mathcal{N}$.

The HP problem is known to be NP-hard [14], [15]. Fortunately, there are successful HP tools (e.g., hMETIS [16] and PaToH [17], [18]) that implement efficient and effective heuristics.

$\Pi^R = \{\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_K\}$ is said to be a K-way replicated partition of a given hypergraph $\mathcal{H}$, if vertex parts are collectively exhaustive. Note that parts need not be pairwise disjoint.

In the *Replicated Hypergraph Partitioning Problem* [19], given a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$, an imbalance ratio ϵ, and a replication ratio ρ, we want to find a K-way replicated partition Π^R that minimizes the cutsize defined in Eq. (2), while satisfying the following constraints:

- Balancing constraint: $W_{max} \leq (1+\epsilon)W_{avg}$, where $W_{max} = max_{1\leq k\leq K} W(\mathcal{V}_k)$ and $W_{avg} = (1+\rho)W(\mathcal{V})/K$.
- Replication constraint: $\sum_{k=1}^{K} W(\mathcal{V}_k) \leq (1+\rho)W(\mathcal{V})$

In RHP, using Eq. (1) for calculating connectivity of a net may not be exact due to vertex replications. The replicated connectivity $\Lambda^R(n_j)$ of a net n_j can only be defined after solving a pin selection (or replica selection) problem [19]. $\Lambda(n_j)$ is a superset of $\Lambda^R(n_j)$. So, $\lambda^R(n_j) = |\Lambda^R\ (n_j)| \leq \lambda(n_j)$. Pin (replica) selection for a net corresponds to selecting a set of parts whose vertices cover all pins of that net. In RHP, connectivity of a net is defined as the set of covering parts. Note that finding the minimum set of covering parts is NP-hard [20].

The cutsize definition for the RHP problem can be obtained by replacing $\lambda(n_j)$ with $\lambda^R(n_j)$ in Eq. (2). We explain how we address the pin selection problem in detail in Section 5.2.

3 Motivation and Problem Definition

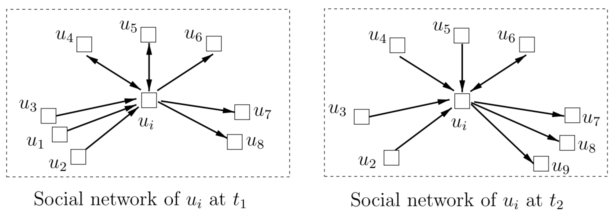
3.1 Motivating Insights

We devised a number of experiments to understand the effects of various metrics on multi-user query performance. In these experiments, more than two million read/write requests are directed to a 16-node Cassandra system using the standard hash-based data distribution. As a result, we observed three critical metrics for further exploration of their effects on system performance.

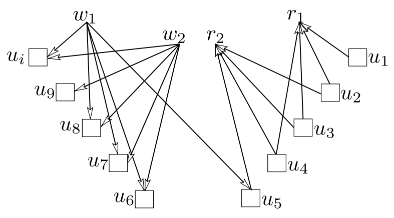
Metric 1, Server load imbalance: Load imbalance has a negative impact on the performance of a distributed system. In Fig. 2a, we display this negative impact on system latency and throughput. Note that high imbalances in token ranges are usual in NoSQL systems such as Cassandra due to random server-token generation. Even though there are ways to achieve more uniform range distributions, due to skewed query distributions, imbalance in randomized partitioning methods that do not utilize query logs is pretty common. As seen in the figure, as the imbalance increases, the overall query latency tends to increase and the overall system throughput tends to decrease.

Metric 2, I/O load: Increasing replication can cause increases in read/write latencies due to consistency requirements. Even in an eventually consistent system, whenever a user data is replicated, all write requests to that user's data must be (eventually) propagated to all replicas, which causes an increase in the total amount of I/O operations performed by the system when compared with an unreplicated scenario. This may have a negative effect not only on the write performance but also on the overall system performance as well. In Fig. 2b, we display the impact of increased replication on the I/O load and system latency. As seen in the figure, as the replication factor increases, the I/O load and the overall query latency tend to increase as well.

Metric 3, Number of servers processing a query (query span): Several studies already indicate that minimizing query span also minimizes query latency [2], [3], [4], [5], [6], [7]. In Fig. 2c, we display the correlation between the query span and latency. As seen in the figure, as the number of servers

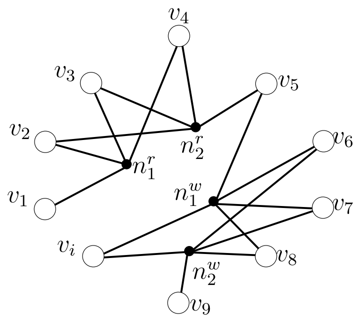

Social network of u_i at t_1
Social network of u_i at t_2

(a) Social network of u_i. $u_x \rightarrow u_y$ implies u_y follows u_x.



$user(w_1) = user(w_2) = user(r_1) = user(r_2) = u_i$
$time(w_1) = time(r_1) = t_1$, $time(w_2) = time(r_2) = t_2$
$participants(r_1) = \{u_1, u_2, u_3, u_4\}$, $participants(w_1) = \{u_5, u_6, u_7, u_8\}$
$participants(r_2) = \{u_2, u_3, u_4, u_5\}$, $participants(w_2) = \{u_6, u_7, u_8, u_9\}$

(b) User u_i tweets (w_1 and w_2) and visits his/her homepage (r_1 and r_2) at t_1 and t_2. $w_x \rightarrow u_y$ implies $user(w_x)$ tweets to user u_y. $u_x \rightarrow r_y$ implies $user(r_y)$ reads the latest tweet(s) of user u_x.



(c) Temporal hypergraph model.

Fig. 3. Modelling read and write requests.

processing a query increases, the latency tends to increase. Similar experiments with different number of nodes and with different query loads exhibit similar patterns.

3.2 Problem Definition

In this study, we utilize the social network and interactions between users to predict the user actions that are likely to occur in the near future. Using these predictions, we perform a selective replicated partition of the user data. Given the metrics presented in Section 3.1, our problem definition is as follows:

Definition (Selective replicated partitioning for minimized server interaction problem): *Given a set $\mathcal{U}$ of users, a set $\mathcal{Q}$ of queries, K homogeneous servers, an imbalance ratio ϵ, and a maximum allowed replication percent ρ, find a user-to-server placement that minimizes the average query span in $\mathcal{Q}$, while balancing the number of queries processed by each server (within the imbalance ratio ϵ) and increasing the I/O load (when compared to an unreplicated scenario) by at most ρ percent.*

In order to address this problem, we perform a replicated partitioning of the constructed hypergraph and interpret the partition result as a solution to the selective replicated partitioning for minimized server interaction problem. To test our solution, we enhance the Cassandra NoSQL system, design a Twitter clone utilizing the enhanced Cassandra, and compare the performance of our solution on Amazon EC2 with existing solutions. Note that we assume a single datacenter setting.

4 Temporal Activity Hypergraph Model

4.1 Model Input

We are given a log $\mathcal{Q}$ of queries and the log contains information on the timing of the activities. We divide the activities into time periods and utilizing the activities in the previous periods, we aim to identify the pattern and frequency of the activities that are likely to occur in the next period. We then partition and replicate data according to this prediction. The time periods can be months, weeks, days, or even hours. The model appraises the activities in recent periods more than the activities in older periods and appraises all activities that occur in the same period equally. The selection of these time periods also determines the frequency of partitioning actions.

We assume that the time span $\mathcal{T}$ of the activities in the log is divided into $T = |\mathcal{T}|$ time spans, that is $\mathcal{T} = \{t_1, t_2, \ldots, t_T\}$, where t_1 denotes the earliest time period and t_T denotes the most recent time period in the log. We also assume that the log $\mathcal{Q}$ consists of a set $\mathcal{R} = \{r_1, r_2, \ldots\}$ of read requests and a set $\mathcal{W} = \{w_1, w_2, \ldots\}$ of write requests. That is, $\mathcal{Q} = \mathcal{R} \cup \mathcal{W}$. A read request necessitates the retrieval of a fixed number (e.g., m) of latest tweets of a user's friends. These type of requests are issued whenever a user checks his homepage in Twitter and thus they are pretty common. A write request necessitates the update of a user's and his followers' data. These type of requests are issued whenever a user tweets and thus they are also pretty common.

Each read request $r_j \in \mathcal{R}$ has attributes: $user(r_j)$, $time(r_j)$, and $participants(r_j)$. Here, $user(r_j)$ denotes the user that issues r_j, and $time(r_j)$ denotes the time of r_j. $participants(r_j)$ denotes the set of users whose tweets are returned to $user(r_j)$ in response to r_j. That is, $participants(r_j)$ corresponds to the set of users followed by $user(r_j)$ and has at least one tweet in the set of most recent m tweets $user(r_j)$ can retrieve at $time(r_j)$.

Each write request $w_j \in \mathcal{W}$ has attributes: $user(w_j)$, $time(w_j)$, and $participants(w_j)$. Again, $user(w_j)$ denotes the user that performs w_j, and $time(w_j)$ denotes the time of w_j. $participants(w_j)$ denotes the set of users who receive the tweet made by $user(w_j)$ at $time(w_j)$. That is, $participants(w_j)$ corresponds to the set of users who follow $user(w_j)$ at $time(w_j)$.

4.2 Model Construction

For a given log $\mathcal{Q} = \mathcal{R} \cup \mathcal{W}$, the construction of the temporal activity hypergraph $\mathcal{H}(\mathcal{Q}) = (\mathcal{V}, \mathcal{N} = \mathcal{N}^r \cup \mathcal{N}^w)$ is performed as follows. For each user u_i, there exists a vertex v_i in $\mathcal{V}$. For each read request $r_j \in \mathcal{R}$, there exists a read net n_j^r in $\mathcal{N}^r$. For each write request $w_j \in \mathcal{W}$, there exists a write net n_j^w in $\mathcal{N}^w$. A read net n_j^r connects the vertices corresponding to the users in $participants(r_j)$. A write net n_j^w

connects the vertex for $user(w_j)$ and the vertices corresponding to the users in $participants(w_j)$. That is,

$$Pins(n_j^r) = \{v_i : u_i \in participants(r_j)\}.$$
$$Pins(n_j^w) = \{v_i : u_i \in participants(w_j)\} \cup \{v_i : u_i = user(w_j)\}.$$

Note that a write net n_j^w connects the vertex for $user(w_j)$ since w_j must be propagated to the server storing $user(w_j)$, whereas a read net n_j^r does not connect the vertex for $user(r_j)$ since r_j gathers data only from friends of $user(r_j)$. Also note that the degree of each read net n_j^r can be at most m under the assumption that upon visiting their homepages users are served the latest m tweets of their friends.

The relations between read/write requests and nets are depicted in Fig. 3. In Figs. 3a and 3b, squares represent users. Fig. 3a shows the social network of a sample user u_i at two different time periods t_1 and t_2. At time t_1, u_i follows users u_1-u_5 and is followed by users u_4-u_8, and at time t_2, u_i follows users u_2-u_6 and is followed by users u_6-u_9. Fig. 3b illustrates two sample write scenarios, where u_i tweets at t_1 and t_2. After the tweet, the data of u_i and his followers are updated. Note that the set of users that receive u_i's tweets may change in time due to the changes in the social network structure. Fig. 3b also illustrates two sample reads, where u_i checks his homepage at t_1 and t_2 and receives the latest tweets of his friends. We assume only the latest m tweets of the friends of a user are returned as a response to a read request ($m = 4$ in this example). Depending on the activities of friends and social network of u_i at the time of the request, the set of users whose tweets are returned to u_i can change.

Fig. 3c shows the temporal activity hypergraph that models the read and write requests of Fig. 3b. As seen in the figure, the temporal hypergraph successfully distinguishes the read and write requests performed by the same user in different time periods by placing separate nets for such requests.

Apart from forming the structure of the hypergraph model, temporality also comes into play in setting vertex weights and net costs. We use a decay factor $\alpha(t)$ to impose an order of precedence among read/write requests in different time periods so that requests in recent periods have higher importance. Hence, the costs of nets representing these recent queries and the weights of vertices representing users who are active in the recent periods are assigned higher values. In this study we use the decay function proposed in [4]. For time period t, the decay factor $\alpha(t)$ is computed as $\alpha(t) = \frac{|\mathcal{Q}_t \cap \mathcal{Q}_T|}{|\mathcal{Q}_T|}$, where $\mathcal{Q}_t$ denotes the set of queries in time period t. The decay function only affects the costs of nets and weights of vertices, thus our model can be coupled with any other decay function (e.g., exponential smoothing [21]). The cost $c(n_j)$ of a net n_j associated with a read request r_j or a write request w_j is set equal to the decay factor for the time period $time(r_j)$ or $time(w_j)$ to reflect the closeness of the associated request to the current time, i.e., $c(n_j^r) = \alpha(time(r_j))$ and $c(n_j^w) = \alpha(time(w_j))$.

The weight $w(v_i)$ of a vertex v_i is set to reflect the total amount of activity u_i is expected to perform and is computed as $w(v_i) = \sum_{n_j \in Nets(v_i)} c(n_j)$.

5 Replicated Partitioning of Temporal Activity Hypergraph

5.1 Replicated Partitioning of $\mathcal{H}(\mathcal{Q})$

A K-way replicated partition Π^R of the temporal activity hypergraph $\mathcal{H}(\mathcal{Q})$ can be used in replicated placement of user data in a distributed system. That is, a K-way replicated partition $\Pi^R = \{\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_K\}$ of $\mathcal{H}(\mathcal{Q})$ is decoded to induce a K-way user-to-server mapping as follows: The set of users and their data corresponding to the set of vertices in $\mathcal{V}_k$ are assigned to server S_k. In other words, for each vertex $v_i \in \mathcal{V}_k$, S_k is held responsible for storing the data associated with user u_i.

With the cost and weight schemes described in Section 4.2, maintaining the partitioning constraint of balanced part weights is expected to balance the number of read and write requests that will be processed by the servers in the next time period (*Metric 1*). Maintaining the replication constraint is expected to limit the amount of increase in I/O load due to replicated data (*Metric 2*). Optimizing the partitioning objective of reducing cutsize is expected to minimize the average query span in the next time period (*Metric 3*), thus minimizing the distributed query processing overhead.

Consider a read net n_j^r with $\Lambda^R(n_j^r)$ for a given Π^R. If $\Lambda^R(n_j^r) = \{\mathcal{V}_k\}$, then n_j^r is uncut and internal to $\mathcal{V}_k$. This implies that all users contributing to the latest tweets that are retrieved by the read request r_j are grouped in server S_k, and S_k can process r_j by using only local data. On the other hand, if a net n_j^r is cut with connectivity set $\Lambda^R(n_j^r)$, this implies that, due to replica selection, the servers corresponding to the parts in $\Lambda^R(n_j^r)$ will process r_j. So, $\lambda^R(n_j^r) = |\Lambda^R(n_j^r)|$ denotes the number of distinct servers that will process r_j. Thus, minimizing the objective in Eq. (2) with $\lambda(n_j)$ replaced by $\lambda^R(n_j^r)$ corresponds to minimizing the span of read requests.

Consider a write net n_j^w with $\Lambda(n_j^w)$ and $\Lambda^R(n_j^w)$ for a given Π^R. Even though acknowledging a write request on one server is enough for processing it, the write is propagated to all replicas eventually causing an increase in the I/O loads of all servers storing a replica. Thus, unlike a read net n_j^r which contributes to the loads of the servers corresponding to the parts in $\Lambda^R(n_j^r)$, a write net n_j^w contributes to the loads of the servers corresponding to the parts in $\Lambda(n_j^w)$. So, the discussion given for a read net n_j^r in the previous paragraph can be made for a write net n_j^w by replacing $\Lambda^R(n_j^r)$ with $\Lambda(n_j^w)$ as follows. w_j enforces the system to perform write operations on all servers that store replicas of the users corresponding to the pins of n_j^w. Thus, the servers corresponding to the parts in $\Lambda(n_j^w)$ are involved in processing w_j. That is, minimizing the objective in Eq. (2) with $\lambda(n_j) = \lambda(n_j^w)$ corresponds to minimizing the span of write requests.

Consequently, the cutsize definition that covers both read and write nets can be formulated as

$$\chi(\Pi^R) = \sum_{n_j^w \in \mathcal{N}^w} c(n_j^w)\lambda(n_j^w) + \sum_{n_j^r \in \mathcal{N}^r} c(n_j^r)\lambda^R(n_j^r). \tag{3}$$

Fig. 4 shows the temporal activity hypergraph $\mathcal{H}(\mathcal{Q}_{sample})$ corresponding to the sample log $\mathcal{Q}_{sample}$ of read and write

$\mathcal{Q}_{sample} = \{r_1, w_1, r_2, w_2, r_3, r_4\}$
r_1: u_6 reads the tweets of users u_1, u_2, and u_3
w_1: u_1 tweets to his followers u_3 and u_6
r_2: u_4 reads the tweets of users u_3 and u_5
w_2: u_2 tweets to his followers u_4, u_5, and u_6
r_3: u_5 reads the tweets of users u_2, u_6, u_7, and u_8
r_4: u_1 reads the tweets of users u_3, u_4, and u_6

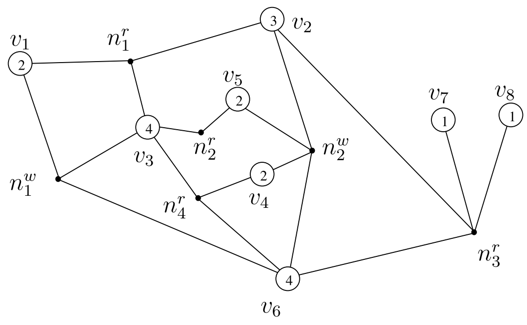


Fig. 4. Temporal activity hypergraph model $\mathcal{H}(\mathcal{Q}_{sample})$.

requests given in the figure. In this example, we assume that all requests are assumed to be performed in the same time period for simplicity.

Fig. 5 shows a four-way replicated partition Π^R of the hypergraph in Fig. 4 after pin selection is applied on read nets. In the figures, empty circles represent unreplicated vertices, shaded and dashed circles represent replicated vertices and small black dots represent nets. Numbers in circles indicate vertex weights. Since we assume that all requests are performed in the same time period, the nets can be considered to have unit costs. Due to the costs of the nets they are connected, the weights of vertices are $w(v_1) = 2$, $w(v_2) = 3$, $w(v_3) = 4$, $w(v_4) = 2$, $w(v_5) = 2$, $w(v_6) = 4$, $w(v_7) = 1$, and $w(v_8) = 1$. Considering these vertex weights, the part weights for the four parts in Fig. 5 are $W(\mathcal{V}_1) = 8$, $W(\mathcal{V}_2) = 7$, $W(\mathcal{V}_3) = 10$, and $W(\mathcal{V}_4) = 6$.

Write nets n_1^w and n_2^w are cut with connectivity $\lambda(n_1^w) = \lambda(n_2^w) = 4$, thus each incurring a cost of four to the cutsize. Among read nets, n_1^r, n_2^r, and n_4^r are internal and thus each incur a cost of one, whereas net n_3^r is cut with connectivity $\lambda^R(n_3^r) = 2$ and thus incurs a cost of two. The cutsize according to Eq. (3) is $4 + 4 + 1 + 1 + 1 + 2 = 13$.

If we decode the four-way replicated partition Π^R as a four-way user-to-server mapping, then both write requests w_1 and w_2 necessitate writes on servers S_1, S_2, S_3, and S_4. Replication of v_2 and v_3 to $\mathcal{V}_1$ makes net n_1^r internal to $\mathcal{V}_1$ enabling r_1 to be processed locally on S_1. Similarly, replication of v_3 to $\mathcal{V}_4$ enables read request r_2 to be processed locally on S_4, and replication of v_3 and v_6 to $\mathcal{V}_3$ enables read request r_4 to be processed locally on S_3. Also, replication of v_2 and v_6 to $\mathcal{V}_2$ enables r_3 to be processed only on two servers (S_2 and S_4). Thus, total number of servers processing queries in $\mathcal{Q}_{sample}$ is equal to 13, which is equal to the cutsize.

The resultant replicated partition provides a user-to-server mapping. In our evaluations, we utilize this mapping to generate a horizontal partition of the Twissandra column families. To be more precise, the user partition induces a partition of all cfs of Twissandra. It is clear that a partition of users implies a row-based partition of the

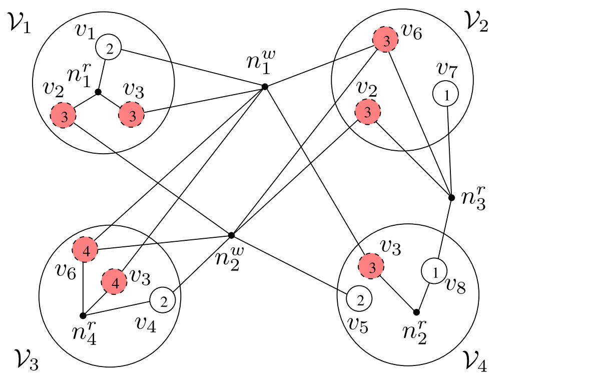


Fig. 5. A four-way replicated partition of $\mathcal{H}(\mathcal{Q}_{sample})$. The dashed, red-filled circles indicate replicated vertices.

USER, FRIENDS, FOLLOWERS, TIMELINE, and USERLINE column families since the row keys for all these cfs are the username (Section 2.1.3). The partitioning of TWEET cf is performed according to the username of the tweeting user. In the end, each user's personal information, friends, followers, userline, timeline and tweets are stored on the same server(s).

5.2 Replica Selection

When user data is replicated, the problem of selecting which replicas to use arises during query processing. The objective of replica selection is to minimize the span of a read query in a replicated environment. In HP-theoretic view the replica selection problem corresponds to the pin selection problem.

The pin selection problem can be formulated as a set cover problem. An instance $(\mathcal{X}, \mathcal{F})$ of the *set-cover problem* consists of a finite set $\mathcal{X}$ and a family $\mathcal{F} = \{X_1, X_2, \ldots,\}$ of subsets of $\mathcal{X}$, which cover $\mathcal{X}$ (i.e., $\mathcal{X} = \bigcup_{X_i \in \mathcal{F}} X_i$), and the problem is to find a minimum-size subset $\mathcal{C} \subseteq \mathcal{F}$ whose members cover all of $\mathcal{X}$. The transition from the pin selection problem to the set cover problem can be done as follows: For a net n_j in a given replicated partition Π^R of $\mathcal{H}(\mathcal{Q})$, $\mathcal{X}$ is set equal to the set of pins of n_j, and the family of subsets $\mathcal{F}$ is set equal to the subsets of the pins of n_j that reside in the parts of $\Lambda(n_j)$, i.e.,

$$\mathcal{X} = Pins(n_j)$$
$$\mathcal{F} = \{X_k : X_k = Pins(n_j) \cap \mathcal{V}_k,\ \text{where } \mathcal{V}_k \in \Lambda(n_j)\}.$$

Note that the number of subsets in $\mathcal{F}$ is equal to $\lambda(n_j)$. For a net n_j, after the above transition, the solution $\mathcal{C}$ to the defined set-cover problem is then decoded as a solution $\Lambda^R(n_j)$ of the replica selection problem.

The set-cover problem is known to be NP-hard [20]. We use a simple $(ln(n) + 1)$-approximation algorithm [19], [20] to solve the pin selection problem. For a net n_j, this greedy heuristic first selects the part/subset, say X_k, in $\Lambda(n_j)$ that contains the largest number of uncovered pins of n_j, then includes this X_k into $\mathcal{C}(n_j)$, and then eliminates the vertices covered by X_k from X. Selection and elimination processes are repeated till there remains no uncovered vertices. We should note that $\mathcal{X}$ may contain unreplicated vertices. So prior to executing the above heuristic, the unreplicated vertices and their respective vertex parts (subsets) are pre-selected for the sake of runtime efficiency. The resulting set cover $\mathcal{C}(n_j)$ for net n_j is decoded as follows to induce $\Lambda^R(n_j)$:

$$\Lambda^R(n_j) = \{\mathcal{V}_k : X_k \in \mathcal{C}(n_j)\}.$$

After pin selection, $\Lambda^R(n_j)$ determines the set of servers that will process r_j. That is, for each $\mathcal{V}_k \in \Lambda^R(n_j)$, server S_k will process r_j. Replica selection can be performed at a gateway node accepting user queries on behalf of Twitter and directing them to Cassandra.

6 Experimental Results

To evaluate the proposed method for achieving replicated partitioning of social networks, we embedded a selective partitioning and replication scheme into Apache Cassandra version 0.8.7. We also modified Twissandra [9] to utilize our enhanced Cassandra, and obtained a Twitter-like system, where the social network structure and user tweets are stored by Cassandra. We tested this Twitter clone on the Amazon Elastic Computing Cloud (EC2), which provides Linux-based virtual machines (instances) running on top of the Xen virtualization engine.

6.1 Experimental Setup

6.1.1 Data Set

In our experiments we made use of the Twitter data set from [22]. This data set was crawled from Twitter between October 2006–November 2009 and contains tweets of 465,107 distinct users. The crawl was seeded from a set of genuine (or authoritative) users collected from Mashable.[1] The social graph of the seed set was expanded by following friendships of the users. The tweets of these users were collected every 24 hours. Among these users there are 836,541 social relationships and the data set contains a total of 25,378,846 tweets. Within this data set, we selected one year's worth of queries, used the first eleven months for modeling, and the last month for evaluations. Specifically, we made use of 8,105,164 tweets made between November 2008–September 2009 for constructing the temporal activity hypergraph model and the alternative graph models. Since social relationships of some users were missing in the original data set, we recrawled these social relationships between 10–14 of September 2012. This recrawl added 80,523 new social relationships to our data set making a total of 917,064 social relationships. The investigated data distribution methods are tested with the tweets made in October 2009, which contains 1,882,256 tweets.

Since the Twitter data set from [22] only contains tweets and social relations, and does not contain any read queries, we designed the following method to generate read queries. For each user, we go over the log of tweets in increasing time order to count the number of new tweets received by that user and whenever this count exceeds a certain number (set to two in our experiments), a read request for the last 40 tweets of that user is generated, if the number of distinct users in the last 40 tweets is greater than a threshold (set to two in our experiments). There are two main motivations behind this query generation method. First, we expect there to be a correlation between the number of tweets a user receives and the number of times he checks his homepage. Second, by default, Twitter sends reminder/informer emails to a user who has a number of unread tweets and has not logged-in to the system for a while, which occasionally causes users to login to Twitter upon receipt of such emails.

1. http://mashable.com/2008/10/20/25- celebrity-twitter-users/.

Generated read queries are interleaved with write queries based on time information. The interleaved query set for November 2008–September 2009 period contains 5,300,407 read and 8,105,164 write queries and the interleaved query set for October 2009 contains 1,429,303 read queries and 1,882,256 write queries.

We assume that a user visiting his homepage retrieves the latest $m = 40$ tweets of his friends. On average, a read query contains 39.8 individual read and a write query contains 3.9 individual write operations.

TABLE 1
Properties of Used EC2 Instance Types

Instance Type	`Micro`	`Medium`
Memory	613 MB	3.75 GB
CPU (EC2 Comp. Unit)	1	2
Storage	EBS storage	410 GB
I/O Performance	Low	Moderate
API Name	t1.micro	m1.medium

6.1.2 Amazon EC2 Setup

EC2 instances are classified based on their memory, CPU, network, and I/O characteristics. In our experiments, we used the `m1.medium` instances for Cassandra servers, and the `t1.micro` instances for the Twissandra system and submitting queries (called submitters). The properties of these instance types are given in Table 1. One EC2 Compute Unit provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor [23]. Since EC2 instances are virtual servers, they share resources with other virtual servers.

We used instance store volumes instead of EBS volumes for storage in Cassandra servers to avoid variations that might occur in EBS during I/O operations. In order to model a single-datacenter setting, we selected all Cassandra servers from the same region (us-east-1a). Instead of using multiple submitter threads from a single machine, we opted to create multiple EC2 submitter instances to more closely emulate distributed multiple users, since threads from the same machine would not have great variations in terms of network overhead.

6.2 Alternative Methods for Comparison

We compare the performance of the proposed replicated hypergraph-partitioning-based data distribution method (`RHP`) with the standard hash-based mapping of Cassandra (`HASH`), a graph-partitioning-based scheme (`GP/L`) that extends the approach in [3] to utilize query logs, a further extended graph-partitioning-based scheme (`GP/T`) that utilizes the temporal information in user interactions in the same way we do in `RHP`, and a third

TABLE 2
Preprocessing Overheads (Secs)

K	GP/L	GP/T	SCH	RHP
8	634.6	636.1	2687.6	882.6
16	641.6	642.3	3064.1	1130.3
32	650.5	651.6	3778.8	1436.2

graph-partitioning-based scheme called SCHISM [7] (SCH), which achieves data distribution with a unified partitioning and replication approach.

Details of HASH and RHP are given in Sections 2.1.1 and 4, respectively. Both GP/L and GP/T construct a graph model from the social network structure [3]. In GP/L, we extend the graph model to utilize the query logs by weighting its edges. For each write query w_j, we increment the weight of each edge $(user(w_j), u_k)$, where $u_k \in participants(w_j)$, by one. Similarly, for each read query r_j, we increment the weight of each edge $(user(r_j), u_k)$, where $u_k \in participants(r_j)$, by one. In GP/T we further extend GP/L so that contributions of interactions to edge weights are scaled utilizing the decay factor used in RHP (Section 4.2). In both GP/L and GP/T, after partitioning, one-hop replication strategy [3], [6] is utilized to achieve replication.

In SCH, each user is represented by a vertex and the edges are used to capture the relations among the users of a query. The users that interact in a query are modeled with a completely connected subgraph, which is then incorporated into the original graph by adding new edges/vertices. Replication in SCH is achieved by exploding vertices into star-shaped structures in the graph prior to partitioning. All three graph models are partitioned with the multilevel graph partitioning tool MeTiS [24]. After partitioning, replicas of a vertex that belong to the same part are collapsed into a single vertex.

In RHP, for replicated partitioning of the proposed hypergraph model, we used the rpPaToH tool [19]. rpPaToH is capable of replicating vertices of a hypergraph during the partitioning process in order to improve a target objective under given balancing and replication constraints. During the RHP runs, since rpPaToH supports the same connectivity metric for all nets, we included only the read nets as read requests are more expensive than write requests.

To integrate GP/L, GP/T, SCH, and RHP into Cassandra, we implemented a new replication strategy called selective replicated partitioning strategy (*SRPStrategy*), which extends the abstract AbstractReplicationStrategy class of the locator package. SRPStrategy initially loads a lookup table that describes the user-to-server mapping. For any user whose mapping is not provided, SRPStrategy acts the same as RackUnawareStrategy. We should note that, utilizing lookup tables for data placement necessitates the use of efficient schemes for maintaining such structures in highly distributed settings. This issue is beyond the scope of this work but there exist studies addressing this problem [25].

All five schemes are tested using 100 percent replication. This corresponds to two-copy replication for HASH, whereas GP/L, GP/T, SCH, and RHP perform selective replication.

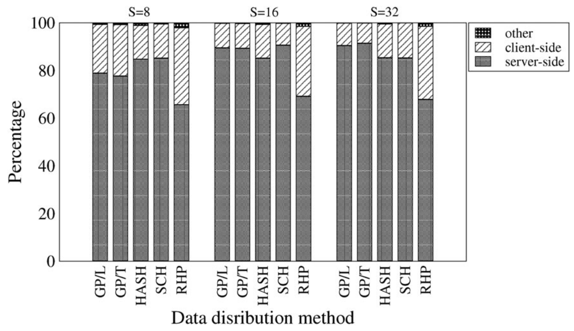


Fig. 6. Dissection of overall query processing time.

The models used in GP/L, GP/T, SCH and RHP are constructed using the query logs between November 2008–September 2009. All five schemes are evaluated using the query logs of October 2009.

6.3 Preprocessing Costs and Time Dissections

In Table 2, the preprocessing overheads of GP/L, GP/T, SCH, and RHP are presented. The overheads of GP/L and GP/T are lower than RHP, whereas SCH has the longest preprocessing time due to its much larger graph size. To give a perspective, the number of edges in SCH are 80 times the number of pins in RHP.

As seen in Table 2, the preprocessing times are in the order of minutes. Since we assume that the preprocessing is to be performed in relatively long intervals such as days or weeks, they are within acceptable limits. Also note that, in social networks, only a fraction of users are extremely active and they generate a significant portion of the total workload (e.g., see the 90-9-1 rule [26]). Given this knowledge, the size of the processed graphs/hypergraphs can be significantly reduced by eliminating vertices corresponding to inactive users. That is, it is possible to eliminate infrequent queries from the workload, which enables workload-aware approaches to keep their preprocessing overhead low. Data of relatively inactive users can be handled by using the default partitioning/replication scheme of the underlying NoSQL system, e.g., HASH. This also reduces the size of the lookup tables utilized by the graph/hypergraph models.

In Fig. 6, dissections of the overall query processing times of GP/L, GP/T, HASH, SCH, and RHP are illustrated for $S = 8, 16$, and 32 servers. The query processing times are composed of three portions. The server-side portion includes the I/O overheads associated with read/write operations and intra-server communication overheads required for server-side coordination. The client-side portion includes the communication overheads between clients and servers such as the times spent during query submission, retrieval of results/acknowledgments, as well as overheads associated with query preparation. The other portion includes the time spent during replica selection process and server-side threading.

As seen in Fig. 6, the percentages of the server-side portion of RHP are the lowest among all schemes. We note that the times spent during the operations depicted in the client-side portion of all schemes would roughly be equal, since the submitted queries and the gathered

TABLE 3
Comparison of Server-Side Performance Metrics

Metric	Scheme	$S=8$	$S=16$	$S=32$
Percent Read Imbalance	GP/L	287.6	483.2	410.3
	GP/T	320.8	428.0	412.0
	HASH	216.2	417.3	343.1
	SCH	201.4	271.3	486.1
	RHP	**6.3**	**54.3**	**35.4**
Percent Write Imbalance	GP/L	185.5	392.4	623.8
	GP/T	175.3	398.7	600.4
	HASH	235.3	308.1	539.3
	SCH	297.3	295.0	652.8
	RHP	**27.9**	**32.5**	**63.1**
Total I/O Load (Write)	GP/L	26.22M	34.56M	45.39M
	GP/T	27.66M	35.23M	49.34M
	HASH	17.97M	17.84M	**16.78M**
	SCH	**14.56M**	20.08M	28.79M
	RHP	17.71M	**17.20M**	16.82M
Average Read Span	GP/L	**1.0**	**1.0**	**1.0**
	GP/T	**1.0**	**1.0**	**1.0**
	HASH	4.5	8.6	15.9
	SCH	2.7	3.0	3.6
	RHP	3.9	5.9	8.3
Average Write Span	GP/L	**1.0**	**1.0**	**1.0**
	GP/T	**1.0**	**1.0**	**1.0**
	HASH	1.8	2.3	2.7
	SCH	1.2	1.2	1.2
	RHP	1.5	1.7	2.0
Average # of Messages per Write	GP/L	3.3	5.1	8.1
	GP/T	3.4	5.3	8.5
	HASH	3.7	4.7	5.6
	SCH	**2.1**	**3.1**	5.7
	RHP	3.0	3.5	**4.0**

results are the same for all schemes. Given that, the low percentages in the server-side portion of RHP indicate that it is successful in its objective of reducing server-side overhead. As also seen in the figure, with increasing number of servers, the server-side portion of GP/L and GP/T increase, whereas they remain roughly the same for HASH and RHP. This suggests that HASH and RHP scale better than GP/L and GP/T.

6.4 Evaluation

In our experiments, we set the number of Cassandra servers to $S = 8$, 16, and 32. The number of submitters is set to $U = S \times 2$, $S \times 3$, and $S \times 4$. The metrics used for performance evaluation are separated into two as server-side and client-side metrics. The server-side metrics include (i) server read and write load imbalance, (ii) total and average number of I/O operations performed by the system for writes, (iii) average read and write query span, and (iv) average number of messages per write. The client-side metrics include (i) average latency and (ii) average throughput for read and write requests. In the following figures and tables, a bold value in a table indicates the best performance result obtained among the five schemes for the respective experiment instance.

6.4.1 Server-Side Performance Evaluation

In Table 3, we compare the performance of GP/L, GP/T, HASH, SCH, and RHP for the server-side metrics. We do not present the number of read operations in Table 3, since it is the same for all schemes. When we compare balancing performance of the five schemes (here, imbalance is computed as: $100 \times (\mathcal{W}_{max} - \mathcal{W}_{avg})/\mathcal{W}_{avg}$), we observe that GP-based approaches (GP/L, GP/T, and SCH) have the worst balancing performance for both read and write requests. This is because the one-hop replication scheme used by GP/L and GP/T does not take the balancing constraint into account during replication and the replication method in SCH does not consider the balance after query scheduling during partitioning. 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.

In Table 3, we also compare the total number of write operations (I/O load). Apart from the very good performance of SCH on $S = 8$ servers, GP-based approaches generally have the worst performance and they scale poorly with increasing number of servers, whereas HASH and RHP lead to similar I/O loads. The poor performance of GP/L and GP/T approaches is due to their aggressive replication methods that replicate the most active users, e.g., users that tweet more and have many friends and followers, leading to excessive I/O loads.

We also present query span results in Table 3. Since GP/L and GP/T use the one-hop replication scheme, they can respond to all read and write requests from a single node performing the best in terms of read and write span. SCH performs better than RHP in terms of read and write span and RHP performs better than HASH. Furthermore, when the number of servers increases, the rate of increase in the read and write query span is lower for SCH and RHP compared to HASH.

We should note that, as an indicator of query performance, presented span figures should be taken with a pinch of salt. Recall that we adopted a write consistency level of one in our Twissandra implementation and this enables acknowledging a write as soon as a single server acknowledges it. In this respect, write span can be thought as the minimum number of servers that cover all the data items in the multi-write query. However, for eventual consistency, a write to a data item is propagated to all replicas of that data item in the background. For a multi-write query, this means sending messages to all servers containing a replica of the data items in the query. In terms of average number of messages sent for a write, SCH performs the best and RHP performs the second best for $K = 8$ and 16 servers but for $K = 32$, RHP performs much better than all other schemes.

The results in Table 3 can be summarized as follows: GP-based approaches optimize only query span while disregarding balancing and I/O load minimization. HASH is generally used for its good load balancing properties, but as seen in the table, in fact it performs poorly in terms of balancing under skewed query workloads as is observed in real life query workloads. It achieves reasonable I/O load but has the worst locality performance. RHP strikes a balance on these three metrics by trading locality with load balancing and I/O load minimization, which leads to its superior query processing performance, as will be seen in the following section.

TABLE 4
Performance Comparison in Terms of Latency (msec) and Throughput (Queries/Sec)

Number of Servers			$S = 8$			$S = 16$			$S = 32$		
Number of Submitters ($U =$)			$S\times2$	$S\times3$	$S\times4$	$S\times2$	$S\times3$	$S\times4$	$S\times2$	$S\times3$	$S\times4$
Type	Metric	Scheme									
Read	Latency	GP/L	124	184	224	367	509	704	662	879	1165
		GP/T	120	171	226	297	465	594	676	1002	1376
		HASH	75	145	160	123	202	290	173	276	269
		SCH	81	125	175	224	346	470	675	1001	1520
		RHP	**52**	**61**	**76**	**70**	**90**	**108**	**73**	**118**	**150**
Read	Throughput	GP/L	119	120	131	83	90	87	91	102	102
		GP/T	121	128	130	101	98	102	89	90	87
		HASH	162	119	142	187	169	153	256	227	332
		SCH	161	152	141	125	127	117	86	86	77
		RHP	**259**	**324**	**338**	**381**	**434**	**475**	**726**	**674**	**700**
Write	Latency	GP/L	8	12	15	14	20	27	31	47	66
		GP/T	8	11	14	13	19	25	31	51	71
		HASH	18	44	49	36	63	97	59	111	89
		SCH	13	24	38	24	38	56	52	82	101
		RHP	**7**	**10**	**14**	**11**	**16**	**20**	**12**	**19**	**25**
Write	Throughput	GP/L	156	159	172	109	118	114	120	134	135
		GP/T	160	169	172	134	129	134	117	118	115
		HASH	214	156	188	247	223	202	337	299	438
		SCH	212	200	186	164	158	155	113	114	102
		RHP	**341**	**427**	**445**	**502**	**570**	**626**	**955**	**887**	**922**
Overall	Latency	GP/L	58	86	106	167	231	319	303	407	541
		GP/T	56	80	105	136	211	270	310	461	634
		HASH	43	87	97	74	123	180	108	183	166
		SCH	42	97	97	110	172	235	321	479	714
		RHP	**27**	**32**	**41**	**36**	**48**	**58**	**38**	**62**	**79**
Overall	Throughput	GP/L	275	279	303	192	208	201	211	236	237
		GP/T	281	297	302	235	227	236	206	208	202
		HASH	376	275	330	434	392	355	593	526	770
		SCH	373	352	327	289	278	272	199	200	179
		RHP	**600**	**751**	**783**	**883**	**1004**	**1101**	**1681**	**1561**	**1622**

6.4.2 *Client-Side Performance Evaluation*

Table 4 displays the average latency and throughput values observed by submitter nodes for read and write requests under the presence of both types of requests. Table 4 also shows the overall average latency per query (including both read and write queries) and the aggregate throughput.

Among all schemes, RHP achieves the best latency and throughput values in all experiment instances since it provides the best balance among the server-side metrics. GP-based approaches perform the worst in read latency and throughput, whereas HASH performs the worst in write latency and throughput. Relatively better write performance of GP-based approaches is due to their better locality compared to HASH.

As seen in Table 4, for all five schemes, the write latencies are far lower than the read latencies. This is due to the facts that, on average, write requests contain less operations then read requests (3.9 versus 39.8), and Cassandra is optimized for write latency (see Section 2.1.2).

An interesting pattern noticable in Table 4 is that, in several experiment instances, GP-based schemes perform worse than HASH. In fact, among GP-based schemes, only SCH performs better than HASH, and only for low server counts. This is due to the excessive I/O load and poor load balancing performance of GP-based schemes. Some servers utilizing the GP-based schemes are simply overwhelmed with the amount of writes they need to perform and this has a significant negative impact both on their read and write performance. Since our query log contains an interlaced mixture of reads and writes (as is the case for any real system), performance of reads are highly dependent on the performance of writes and vice versa.

Fig. 7 presents a visual comparison of the latency and throughput performance of GP/L, GP/T, HASH, SCH, and RHP under increasing workload. We fix the number of servers to $S = 32$ and vary the number of submitters $U = S \times 2$, $S \times 3$, $S \times 4$. As the number of submitters increases, the read and write latencies of all schemes increase. This is due to the load increase on servers. Since the increase in latency is not as high as the increase in load in most cases, the throughput values of all schemes improve slightly with increasing load. The performance of HASH can vary unpredictably across different runs. This is mainly due to its random token generation at the beginning of each experiment for keyspace partitioning.

Fig. 8 presents the weak scalability comparison of GP/L, GP/T, HASH, SCH, and RHP under increasing number of servers. In each experiment instance, number of submitters is set to $U = S\times4$ and the number of servers is varied $S = 8, 16, 32$. The read and write latency figures show that, among the five schemes, GP-based approaches display the worst scalability characteristics, whereas RHP displays the best. Furthermore, both HASH and RHP scale less

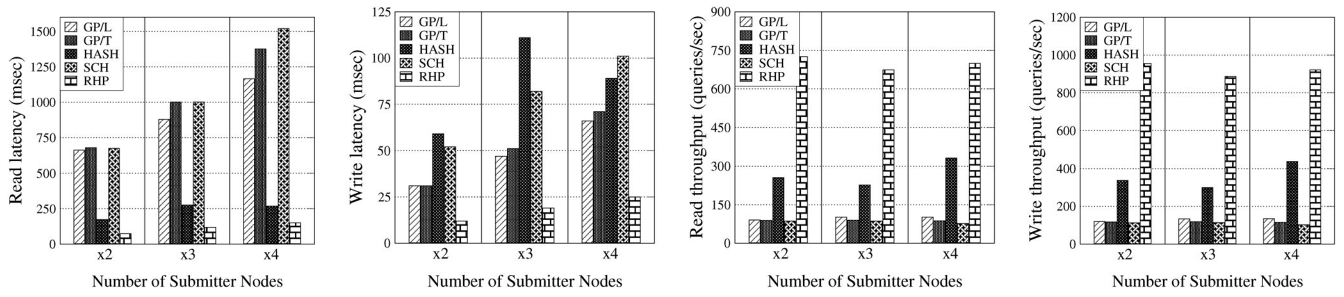


Fig. 7. Effect of increased load on read/write latency/throughput for $S = 32$ servers and $U = S \times 2$, $S \times 3$, $S \times 4$ submitters.

than ideally, which is probably due to the increase in communication overhead with increasing number of servers, as also seen in Table 3.

In Fig. 9, we compare the latency histograms of HASH and RHP. As seen in the presented CDF curves, for RHP, 90 percent of the read queries and 98.5 percent of the write queries can be answered below 200 ms, whereas in HASH, only 74 percent of read and 88 percent of write queries can be answered below 200 ms. RHP can perform 99th percentile of the queries below 700 ms for reads and below 300 ms for writes, whereas HASH can perform the same feat around 1 sec for reads and 800 ms for writes.

7 Related Work

There are recent studies indicating the deficiencies of the partitioning and replication methodologies used in social network data storage systems. [7] proposes a GP-based database partitioning scheme called SCHISM for OLTP-type Web applications that utilize distributed databases. Data items are represented via nodes, transactions are modeled via edges, and the partitioning objective is to minimize the number of distributed transactions. The partitioning/replication scheme in [7] requires generation of a much larger graph from the transaction graph. Replication is handled by "exploding" each node to a star shaped configuration of $n + 1$ nodes, where n indicates the number of transactions accessing the data represented by that node. After partitioning of this larger graph, replicas that fall into the same part are collapsed to a single replica. Another disadvantage of the replication mechanism in [7] is it is not possible to set the amount of replication that will be performed.

Pujol et al. [2] proposes social network partitioning schemes based on graph-partitioning, modular-optimization and random partitioning. Partition qualities are measured via metrics such as the number of internal messages or dialogs. Tests are performed over data sets collected from Twitter and Orkut. For small partition counts, graph-based approaches are shown to perform superior, whereas for large partition counts, modular optimization algorithms perform slightly better.

Pujol et al. [3] extends the work in [2] so that replication is also considered. The proposed replication scheme (one-hop replication) replicates all data items that are in partition boundaries. That is, data items that might be required in multiple servers are replicated to all of those servers. Unfortunately, this replication scheme enforces too much replication and can lead to high I/O loads due to excessive replication of frequently updated data.

Pujol et al. [6] is an extension of the above two studies with an alternative partitioning scheme. However, still the one-hop replication scheme is used for replication. Furthermore, all schemes in [2], [3], [6] use the social network structure for partitioning whereas in this study we make use of both the social network structure and interactions between users (query logs).

The work in [27] focuses on generating personal *feed* pages, pages containing recent activities of followed/tracked users/news-sources. In certain respects, issues addressed in [27] coincide with the problems we tackle. The news-sources broadcast their activities to many users and personal *feed* pages contain activities collected from many news-sources. However, the main problem in [27] is efficient construction of these personal Web pages. To this end, they compare the benefits of pre-materializing these pages with dynamic generation of them upon receipt of queries. We believe that our work and the studies in [27] are complimentary since the pre-materialized pages are generated via multi-user queries as well, so our optimizations are easily applicable to a system running the algorithms proposed in [27].

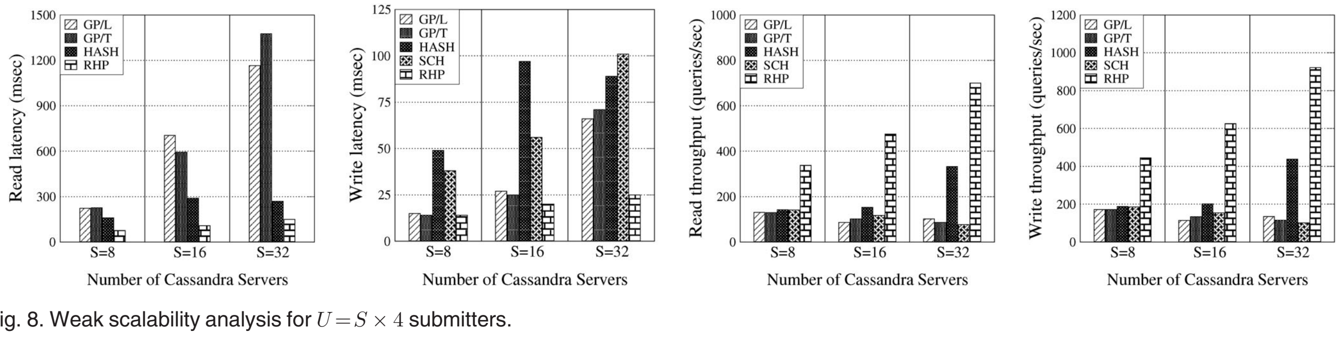


Fig. 8. Weak scalability analysis for $U = S \times 4$ submitters.

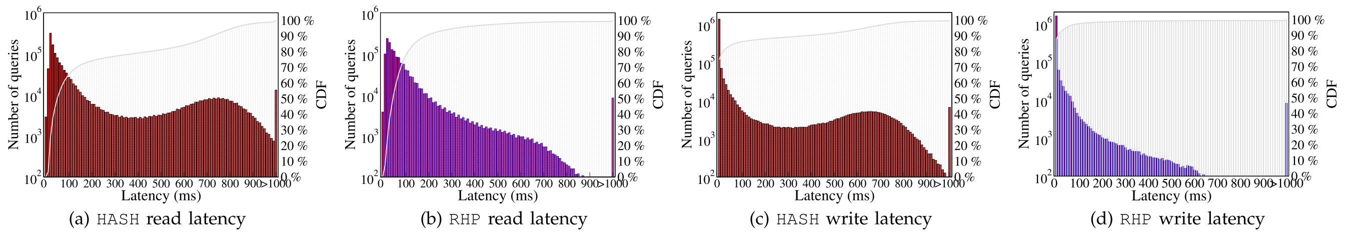


Fig. 9. Read and write latency histograms for HASH and RHP, $S = 16$ servers, $U = 64$ submitters.

In [28], the distributed partition management environment Sedge is proposed for processing large graphs. Sedge is based on Pregel [29], uses graph and complementary partitioning for static primary partitions and workload-aware dynamic secondary partitions. Sedge partition management involves identification and replication of hotspot partitions. Sedge is designed for distributed graph processing applications.

Yuan et al. [4] proposes GP-based models for efficient query processing in time-dependent social network queries. The activity prediction graph model in [4] enables handling of power-law relations observed in social network data via producing lighter tailed interactions. Unfortunately, this study does not address the replication problem.

In [5] and [30], dynamic data placement and replication algorithms for social networks are proposed. The authors of [30] propose the WEPAR dynamic partitioning and replication system. WEPAR differentiates the replicas of a record as either master or slave copies. The main idea in WEPAR is based on placing the master copies of related records in the same node and to generate slave copies for records that receive more read queries.

Authors of [5] extend PNUTS to support selective replication and their algorithm generates placements that respect given replication policy constraints. Their dynamic data placement scheme tries to make use of the temporal locality on data item accesses by adding new replicas when a read miss occurs, removing replicas when a local read is not performed for a while, and a write occurs. In our study, unlike in [5] where reactions to misses and unexpected hits are performed after the fact that these undesired operations are observed, we make use of previous logs to make a temporal prediction of future requests to avoid such operations.

8 Conclusion

In this work, we proposed a temporal activity hypergraph model whose replicated partitioning can be used for data partitioning and replication in social networks. The proposed model naturally encodes multi-way interactions incurred by the most common social network operations. The performance of the proposed model was tested over a popular social network application Twitter. Experimental results using the Cassandra NoSQL system running over Amazon EC2 cluster indicate that the proposed model achieves significant improvements over state-of-the-art hash- and graph-partitioning-based counterparts in terms of important metrics such as latency, throughput, and scalability.

Our results provide insights on parameters affecting the performance of social network storage systems in a cloud setting. 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 possibly perturbing load balance. All-in-all, optimizing solely one of these conflicting metrics does not yield satisfactory results. Our approach performs partitioning and replication simultaneously to reduce the number of servers processing queries while respecting load balancing and I/O load constraints under replication, thus striking a balance between conflicting metrics to achieve the best performance.

Future research avenues of this work include investigation of repartitioning mechanisms that avoid migration of data items in subsequent partitioning iterations and addition of mechanisms that can provide certain performance guarantees for certain percentile of queries.

Acknowledgments

This work was supported by The Scientific and Technological Research Council of Turkey under Grant EEEAG-112E120 and Turkish Academy of Sciences. This work was done when A. Turk was at Bilkent. The authors would like to thank Gunduz Vehbi Demirci and Mehmet Basaran for their valuable help in setting up of the experimental system.

References

[1] 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.

[2] J.M. Pujol, V. Erramilli, and P. Rodriguez, "Divide and Conquer: Partitioning Online Social Networks," *arXiv*, vol. cs.NI, http://arxiv.org/abs/0905.4918v1, Jan. 2009.

[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 (ICDE)*, pp. 205-212, 2012.

[5] S. Kadambi, J. Chen, B.F. Cooper, D. Lomax, R. Ramakrishnan, A. Silberstein, E. Tam, and H. Garcia-Molina, "Where in the World is My Data?" *Proc. VLDB Endowment*, vol. 4, no. 11, pp. 1040-1050, 2011.

[6] 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.

[7] 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, http://dl.acm.org/citation.cfm?id=1920841.1920853, Sept. 2010.

[8] A. Lakshman and P. Malik, "Cassandra: A Decentralized Structured Storage System," *SIGOPS Operating System Rev.*, vol. 44, no. 2, pp. 35-40, http://doi.acm.org/10.1145/1773912.1773922, Apr. 2010.

[9] E. Florenzano, "Twitter Example for Cassandra," https://github.com/twissandra/twissandra, 2014.

[10] 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.

[11] "Project Voldemort," http://project-voldemort.com, 2012.

[12] "Riak," http://basho.com/products/riak-overview/, 2012.

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

[14] C. Berge, *Graphs and Hypergraphs*. North-Holland, 1973.

[15] T. Lengauer, *Combinatorial Algorithms for Integrated Circuit Layout*. John Wiley & Sons, 1990.

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

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

[18] 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, http://dx.doi.org/10.1109/71.780863, July 1999.

[19] 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, http://dx.doi.org/10.1016/j.jpdc.2012.01.004, Apr. 2012.

[20] D.S. Johnson, "Approximation Algorithms for Combinatorial Problems," *Proc. ACM Fifth Ann.Symp. Theory of Computing (STOC '73)*, pp. 38-49, http://doi.acm.org/10.1145/800125.804034, 1973.

[21] 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, http://dx.doi.org/10.1109/MMIT.2010.107, 2010.

[22] 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.

[23] "Amazon ec2 Instance Types," http://aws.amazon.com/ec2/instance-types/, 2014.

[24] 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.

[25] 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.

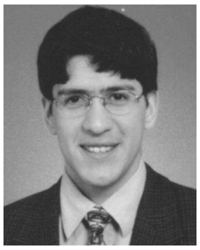
[26] J.N. Alertbox, "Participation Inequality: Encouraging More Users to Contribute," http://www.nngroup.com/articles/participation-inequality/, 2006.

[27] 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, http://doi.acm.org/10.1145/1807167.1807257, 2010.

[28] S. Yang, X. Yan, B. Zong, and A. Khan, "Towards Effective Partition Management for Large Graphs," *Proc. ACM SIGMOD Int'l Conf. Management of Data*, pp. 517-528, http://doi.acm.org/10.1145/2213836.2213895, 2012.

[29] G. Malewicz, M.H. Austern, A.J. Bik, J.C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A System for Large-Scale Graph Processing," *Proc. ACM SIGMOD Int'l Conf. Management of Data*, pp. 135-146, http://doi.acm.org/10.1145/1807167.1807184, 2010.

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



Ata Turk received the BSc and the PhD degrees from the Computer Engineering Department at Bilkent University, Turkey. His research interests include parallel information retrieval and algorithms. He is currently a postdoc at Yahoo Labs.

R. Oguz Selvitopi received the MSc degree in computer engineering in 2010 from Bilkent University, Turkey, where he is currently working toward the PhD degree. His research interests include parallel and distributed systems, scientific computing, and algorithms.

Hakan Ferhatosmanoglu is currently an associate professor at Bilkent University, Turkey. He was with The Ohio State University before joining Bilkent. His current research interests include scalable management and mining of multi-dimensional data. He received Career awards from the US Department of Energy, US National Science Foundation, and Turkish Academy of Sciences. He received the PhD degree from University of California, Santa Barbara.

Cevdet Aykanat received the PhD degree from The Ohio State University, in electrical and computer engineering. Since 1989, he has been with Bilkent University, where he is currently a professor. His research interests include parallel scientific computing and its combinatorial aspects, graph and hypergraph theoretic models, information retrieval, distributed databases, and grid computing. He was an associate editor on the editorial board of *IEEE TPDS* between 2007 and 2012.