Managing Large Dynamic Graphs Efficiently

Jayanta Mondal
University of Maryland, College Park
jayanta@cs.umd.edu

Amol Deshpande
University of Maryland, College Park
amol@cs.umd.edu

ABSTRACT

There is an increasing need to ingest, manage, and query large volumes of graph-structured data arising in applications like social networks, communication networks, biological networks, and so on. Graph databases that can explicitly reason about the graphical nature of the data, that can support flexible schemas and node-centric or edge-centric analysis and querying, are ideal for storing such data. However, although there is much work on single-site graph databases and on efficiently executing different types of queries over large graphs, to date there is little work on understanding the challenges in distributed graph databases, needed to handle the large scale of such data. In this paper, we propose the design of an in-memory, distributed graph data management system aimed at managing a large-scale dynamically changing graph, and supporting low-latency query processing over it. The key challenge in a distributed graph database is that, partitioning a graph across a set of machines inherently results in a large number of distributed traversals across partitions to answer even simple queries. We propose aggressive replication of the nodes in the graph for supporting low-latency querying, and investigate three novel techniques to minimize the communication bandwidth and the storage requirements. First, we develop a hybrid replication policy that monitors node read-write frequencies to dynamically decide what data to replicate, and whether to do *eager* or *lazy* replication. Second, we propose a clustering-based approach to amortize the costs of making these replication decisions. Finally, we propose using a *fairness* criterion to dictate how replication decisions should be made. We provide both theoretical analysis and efficient algorithms for the optimization problems that arise. We have implemented our framework as a middleware on top of the open-source CouchDB key-value store. We evaluate our system on a social graph, and show that our system is able to handle very large graphs efficiently, and that it reduces the network bandwidth consumption significantly.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Parallel Databases*; H.2.4 [**Database Management**]: Systems—*Query Processing*

SIGMOD '12, May 20–24, 2012, Scottsdale, Arizona, USA.

General Terms

Algorithm, Design, Performance, Experimentation

Keywords

Graph databases, Social networks, Replication, Feed delivery

1. INTRODUCTION

In today's world, networks are everywhere. There are social networks, communication networks, financial transaction networks, citation networks, gene regulatory networks, disease transmission networks, ecological food networks, sensor networks, and more. Network data arises even in mundane applications like phone call data, IP traffic data, or parcel shipment data. There is a growing need for data management systems that can support real-time ingest, storage, and querying over such network data. Network data is most naturally represented as a graph, with nodes representing the entities and edges denoting the interactions between them. However, there is a lack of established data management systems that can manage such graph-structured data, and support complex querying or analysis over them. Further, the sizes of these networks, and the number of operations that need to be supported, are growing at an unprecedented pace, necessitating use of parallel and distributed solutions. However, graph operations are not easily parallelizable, and even simple queries over a distributed graph may result in a large number of traversals across the network. The MapReduce framework has emerged as a framework for parallelizing many large-scale analysis tasks. However, MapReduce framework is aimed toward batch processing of largely static data, and cannot support either real-time data ingest or real-time querying.

There is much work on single-site graph database systems [1, 16, 17, 42, 19], and on executing specific types of queries efficiently over them through strategic traversal of the underlying graph, e.g., reachability [25, 48], keyword search queries [14, 44, 5, 26], subgraph pattern matching [19, 11], etc. However, distributed management of dynamic graph data is not as well-studied. There is some work on executing specific types of queries or performing specific types of analysis, e.g., subgraph pattern matching [7, 23], data mining [28] etc.But those works either have limited focus or, in the case of Pegasus [28], are meant for batch processing.

Our goal in this work is to build a system that can support scalable and distributed management of very large, dynamically changing graphs. Keeping with the hardware trends and to support low-latency operations, our system is intended to be fully *in-memory*, and uses disks only as a backing store for historical, archival data. The key challenge with building such a system is that effectively partitioning graphs is notoriously challenging, especially in a dynamic environment. Standard hash-based partitioning schemes do not work well because they end up *cutting* too many edges, i.e.,

placing the endpoints in different partitions. This is a problem because most graph queries or analysis tasks require traversing the edges to fetch the neighbors' information. This not only increases query latencies but also increases the total network communication, thus limiting the scalability of the system.

This has led many researchers to consider more sophisticated partitioning schemes. Although the problem of optimally partitioning a graph into equal-sized partitions while minimizing the edges cut is NP-Hard, there is much work on practical solutions to this problem, and several software packages are available that can generate very good graph partitionings [29, 9, 20]. These techniques however cannot handle highly dynamic graphs where the node access patterns and the graph structure itself may change very rapidly [33]. More importantly, in most practical applications, the highly interconnected nature of graph data means that clean disjoint partitions that minimize the edge-cut do not typically exist [18, 40]. Social networks in particular are very hard to partition because of overlapping community structure, and existence of highly-connected dense components (cores) [34, 38, 39, 6].

We instead investigate an aggressive replication-based approach in this work to scalably manage a large, dynamic graph, where the key idea is to replicate the nodes in the graph to minimize the number of distributed traversals. This approach has been extensively studied in distributed systems and distributed databases (see, e.g., [47, 27]), however, to our knowledge, there is little work on understanding how to use it for distributed graph data management. In a recent work, Pujol et al. [40, 41] considered one extreme version of it for scaling online social networks: they aim to replicate the graph sufficiently so that, for every node in the graph, all of its neighbors are present locally (called *local semantics*). They also use *active replication* (i.e., a *push-on-change* model) where all the replicated data is kept up-to-date. Such an approach however suffers from very high, unnecessary communication to keep the replicas up-to-date; Facebook reportedly uses *pull-on-demand* model instead [21]. Further, the replication overhead to guarantee local semantics, i.e., the number of average copies of each graph node, may be too high in most cases (for a sample Facebook dataset, they needed approximately 2 copies of each node with just 8 partitions [41]).

In this paper, we propose a hybrid, adaptive replication policy that uses a novel *fairness requirement* to guide the replication decisions, and utilizes predictive models about node-level read/write access patterns to choose whether to maintain the replicas actively or passively. The fairness requirement is characterized by a threshold $\tau \leq 1$, and can be stated simply: for each graph node, we require that at least a τ fraction of its neighbors be present locally at the same site. The local semantics [41] becomes a special case of this with $\tau = 1$. A key concern with a policy that makes fine-grained push-pull decisions is that, the overhead of maintaining these decisions (i.e., for a node, deciding which of the neighbor replicas are up-to-date) is very high. We design and evaluate novel clustering-based schemes for this purpose, where we group together nodes with similar access patterns to reduce the overhead without compromising quality. We analyze the problems of deciding what to replicate, and choosing when to push vs pull, and provide both theoretical analysis and efficient practical algorithms (optimal for the latter problem). Our algorithms are *decentralized* by nature, and enable us to make the decisions locally at each node. This also naturally enables us to change the decisions during periods of low load, and/or stagger the times when they are made to avoid significant slowdowns.

We have implemented our distributed graph data management system on top of the Apache CouchDB open-source key-value store. CouchDB supports a sophisticated replication mechanism, and we leverage it by building on top of that. We present a comprehensive experimental evaluation which shows that our algorithms are practical, support low-latency operations, and decrease the total amount of communication by a significant fraction over other policies.

Outline: We begin with a brief overview of our proposed system and discuss various design decisions that we have made (Section 2). We then discuss the key component of our system, the *replication manager* (Section 3), and present algorithms for making the replication decisions (Section 4). We then present a comprehensive experimental evaluation (Section 5), and discuss some of the most related work (Section 6).

2. SYSTEM OVERVIEW

We start with the brief description of the underlying data model and the high-level system architecture, and then briefly discuss some of the key trade-offs in such a system and define a fairness criterion.

2.1 Data and Query Model

The data is represented as a graph $G(V, E)$ where V is the set of all nodes and E represents the set of all edges. To avoid confusion, we refer to the vertices of the graph as nodes, whereas we refer to the sites (machines) across which the graph is partitioned as either sites or *partitions*. The graph is distributed across multiple partitions, and each node has information about its cross-partition neighbors. The basic operations one could perform on the nodes are $read$ and $write$. A $write$ on a node is simply updating or appending node information, whereas a $read$ on a node is reading the information stored in that node. In a traversal we are performing reads on all the nodes that are part of the traversal.

A typical query in a social network context could be: *For a person x, find all of his friends who have attended Stanford Business School and who have a friend from South Africa.* For a query like this, we have to start from node x, visit all its neighbors to check which of them had attended Stanford business school, and then for all those friends, visit their neighbors till we find a neighbor matching the predicate.

Another type of query we may want to support is a subgraph pattern matching query. An example of such a query could be: *Given a citation network, find all the papers that discuss graph pattern matching, and are a result of collaboration between researchers from Stanford and MIT*. To execute such a complex query, we would typically need to build an index on the relevant attributes (e.g., paper abstracts) to quickly find the candidate nodes that may be of interest and then traverse the neighborhoods of those candidate nodes to find the matches.

Although our system is aimed at supporting different types of queries flexibly, for ease of illustration, we primarily focus on a special type of query prevalent in the social network domain, namely, the *"fetch updates from all my neighbors"* query. Given a specific node x, this query requires us to traverse the neighborhood of x and find the latest writes that have been made in that neighborhood. In today's social networks with a large fraction of nodes having a non-trivial number of neighbors, these queries are very hard to scale [21]. This problem is also called the "feed delivery" problem, and also arises commonly in publish-subscribe networks [43]. We revisit the issue of generality below when we describe the system architecture and formulate the optimization problems.

2.2 Architecture

Figure 1 shows the high-level architecture of our system comprising of $replication\ manager$ and other supporting modules. The key components of the system are as follows:

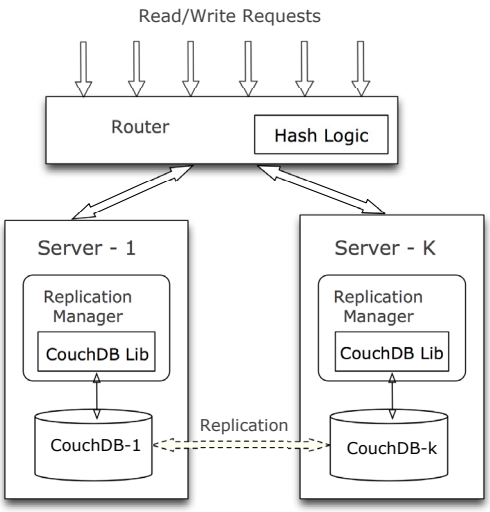

Figure 1: System Architecture

2.2.1 Router

The router is responsible for routing the incoming read and write requests to the appropriate sites. We assume that for all the requests, we are given a *start* node *id*, i.e., the identifier of the node from which the traversal should begin. For complex queries (e.g., subgraph pattern match queries), we assume that an external query processor is responsible for generating the set of start node candidates, perhaps through use of an index. We do not discuss that component further in the paper.

The router uses a *hash partitioning-based* scheme to partition the nodes across the sites. As we discussed earlier, this may result in very large edge-cuts, but is still the preferred method of partitioning in practice for several reasons: (1) hash-based partitioning typically results in balanced workload across the sites and is much more robust to *flash traffic*, (2) the routing process is not only highly efficient, but it can be infinitely parallelized by adding more routers, (3) there is no complicated logic involved in assigning new nodes to partitions, and (4) for a given node, we only need to list its neighbors' ids and not their locations. A scheme that tries to optimize the edge-cuts or other metrics requires large routing tables to be maintained, which increase the routing cost and are hard to keep consistent if the router is replicated. Further, in dynamic networks like social networks, the partitioning may become suboptimal very quickly.

However, we note that the algorithms and the techniques that we develop in the rest of the paper are completely independent of the choice of the partitioning logic. In future work, we plan to investigate building an incremental and flexible partitioning scheme to further reduce the number of distributed traversals.

2.2.2 Storage and Replication

We use the Apache CouchDB key-value store as our backend storage to store all the information related to a node. CouchDB is a schema-free document-oriented data management system, which stores *documents* that can contain any number of fields and attachments. The schema-free nature of CouchDB makes it ideal for storing heterogeneous graph data, where different nodes may have different attributes, and the amount of information stored about a node may have a very wide range (we typically would wish to store historical information as well). There are several reasons we chose CouchDB over other key-value stores. Many of the other key-value stores (e.g., HBase) do not give us sufficient control over where the data is placed, whereas CouchDB is intended primarily as a single-server product.

More importantly, CouchDB has excellent replication support, optimized for minimizing network bandwidth, for selectively replicating data across multiple sites. The documents or databases to be replicated can be specified between any pair of CouchDB servers, and CouchDB will keep the replicas up-to-date by sending only the changes that have occurred. Further, for each database that is replicated, we can specify whether the replication should be "continuous" (i.e., push-based) or not (i.e., pull-based), and these decisions can be changed easily.

As above, the techniques we develop in the rest of the paper are largely independent of this choice, and we can replace CouchDB with another key-value store. However, in that case, depending on the features supported by the key-value store, we may have to write a layer on top of the key-value store to support adaptive pull-based or push-based replication.

2.2.3 Replication Manager

The replication manager is the most important component of our system, and is in charge of making the replication decisions to minimize the network bandwidth and query latencies, and enforce the fairness requirement (discussed below). The replication manager monitors the node read-write frequencies, which are themselves stored along with the nodes in the CouchDB server. It periodically reconsiders the selective replication decisions (i.e., what is replicated, and whether it is active or passive) in a decentralized fashion – each replication manager can make the decisions for the graph node in its partition autonomously. It implements those decisions by appropriately instructing the CouchDB servers. We discuss the specifics of the replication manager in more detail in the next section.

2.3 Trade-offs and Requirements

We briefly discuss some of the key trade-offs and desired properties of a dynamic graph data management system. We also define the fairness criterion and discuss its implications.

Network Bandwidth: It is desirable from a distributed system that the communication overhead be minimized. As discussed earlier, there are two factors at play here: query latencies and replica maintenance. In most of the real-time applications today, read/write operations are latency-critical and failing to keep those under acceptable limits may lead to demise of such applications [22]. To ensure low-latency query execution, we need to minimize the number of cross-partition traversals, and if there is no natural partitioning of the data, then we must use active replication for that purpose. However, in a dynamically evolving graph, the cost of keeping the replicas up-to-date may exceed the benefits of replication. Furthermore, both the write and read access patterns may change dynamically, and different policies may be best at different times. Hence, we must not only choose replicas carefully to ensure low-latency operations, but we should also try to adapt the replication decisions in response to changing access patterns.

Balanced Load: Balanced load across the sites is another very important metric. Balanced load ensures that no resource is under or over-utilized, thereby bringing down the overall system cost and increasing the efficiency of the system. Apart from minimizing network bandwidth, it is expected that the network load will also be balanced for maximum utilization of the system bandwidth. Since our data graph is hash partitioned across sites, it is fair to assume that the network load will be evenly balanced. But, even then we have to make sure that the replication algorithm doesn't interfere with the balance. Secondly, balancing system load, i.e., the

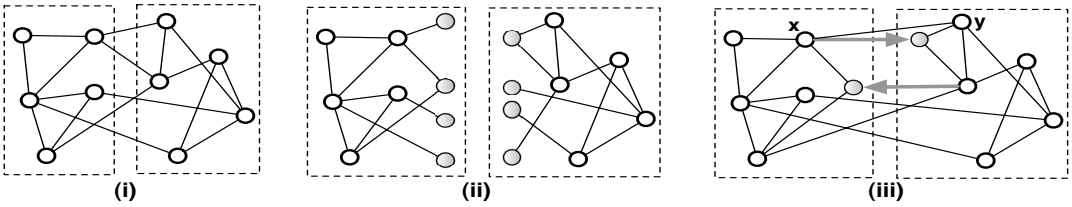

Figure 2: (i) An example graph partitioned across two partitions; (ii) Maintaining *local semantics* [41] requires replicating 80% of the nodes; (iii) We can guarantee fairness with $\tau = \frac{2}{3}$ by replicating just two nodes

load of a site is equally important. Key resources that can get hit hard in such scenario are the CPU and the main memory. Once again, hash partitioning naturally helps us with guaranteeing balanced load, however skewed replication decisions may lead to load imbalance.

Fairness Criterion: Ideally we would like that all queries are executed with very low latencies, which in our context, translates to minimizing the number of pulls that are needed to gather information needed to answer a query. For "fetch neighbors' updates" queries, this translates into minimizing the number of neighbors that are not present locally. In a recent work, Pujol et al. [41] presented a solution to this problem where they guarantee that all the neighbors of a node are replicated locally, and the replicas are kept up-to-date (they called this *local semantics*). This guarantees that no pulls are required to execute the query. However, the number of replicas needed to do this in a densely connected graph can be very high. Figure 2 shows an instance of this where we need to replicate 8 out of 10 nodes to guarantee local semantics for all the partitions. The cost of maintaining such replicas is likely to overwhelm the system. This may be okay in a highly over-provisioned system (we would expect Facebook to be able to do this), but in most cases, the cost of additional resources required may be prohibitive.

Instead, we advocate a more conservative approach here where we attempt to ensure that all queries can make some progress locally, and the query latencies are largely uniform across the nodes of the graph. Such uniformity is especially critical when we are using read/write frequencies to make replication decisions, because the nodes with low read frequencies tend to have their neighbors not replicated, and queries that start at such nodes suffer from high latencies. We encapsulate this desired property using what we call a *fairness criterion*. Given a $\tau \leq 1$, we require that for all nodes in the graph, at least a τ fraction of its neighbors are present or replicated locally. In case of "fetch neighbors' updates" queries, this allows us to return some answers to the query while waiting for the information from the neighbors that are not present locally. For other queries, the fairness requirement helps in making progress on the queries, but the effect is harder to quantify precisely, and we plan to analyze it further in future work. As we can see in Figure 2(c), we need to replicate 2 nodes to guarantee a fairness of 0.8 for the example graph.

Provide Cushion for Flash Traffic: Flash traffic is simply a flood of unexpected read/write requests issued to the system within a small period of time. For example, events like earthquake could cause a deluge of tweets to be posted and consumed on Twitter within seconds. In such situation, any system that does aggressive active replication (e.g., if we were maintaining local semantics) could suffer significantly, as the bandwidth requirement will increase suddenly. We do not optimize for flash traffic directly in this work. However, conservative replication and hash-based partitioning helps in alleviating these problems in our system.

3. REPLICATION MANAGER

In this section, we describe the design of our replication manager in detail. We begin with a brief overview and describe the key operating steps. We then discuss each of the steps in detail.

3.1 Overview

We define some notation that we use in the rest of the paper. Let $G(V, E)$ denote the data graph, let $\Pi = \{P_1, \cdots, P_l\}$ denote the disjoint partitions created by hash partitioning, i.e., $\forall i : P_i \subset V$ and $\cap_i P_i = \phi$. Each of the partitions P_i itself is divided into a number of *clusters*, $C_{i1}, \cdots, C_{ik}$ (we assume the same number of clusters across the partitions for clarity). All replication decisions are made at the granularity of a cluster, i.e., the replication decisions for all nodes within a cluster are identical (this does not however mean that the nodes are replicated as a group – if a node has no edges to any node in another partition, we will never replicate it to that partition). We discuss both the rationale for the clustering, and our approach to doing it below.

Notation	Description
$\Pi = \{P_1, \cdots, P_l\}$	Set of all partitions
R_{ijk}	Replication table corresponding to the cluster C_{ij} and partition P_k
C_{ij}	j^{th} cluster of P_i
$\langle C_{ij}, P_k \rangle$	a cluster-partition pair, $i \neq k$
H	Cost of a push message
L	Cost of a pull message
$\omega(n_i, t)$	Write frequency of n_i at time interval t
$\omega(C_{ij}, t)$	Cumulative write frequency of C_{ij}
$\rho(n_i, t)$	Read frequencies for n_i
$\rho(P_k, C_{ij})$	Cumulative read frequency for P_k w.r.t. C_{ij}

Table 1: Notation

Implementing the Replication Decisions: As we have discussed before, we use CouchDB as our backend store and to implement the basic replication logic itself. In CouchDB, we can specify a *table* (called *database* in CouchDB) to be replicated between two CouchDB servers. Our replication logic is implemented on top of this as follows. For every cluster $C_{ij} \in P_i$, for every other partition P_k with which it has at least one edge, we create a table, R_{ijk}, and ask it to be replicated to the CouchDB server corresponding to P_k. We then copy the relevant contents from C_{ij} to be replicated to that table R_{ijk}. Note that, we usually do not copy the entire information associated with a graph node, but only the information that would be of interest in answering the query (e.g., the latest updates, rather than the history of all updates).

If the decision for the cluster-partition pair $\langle C_{ij}, P_k \rangle$ is a "push" decision, then we ask the CouchDB server to keep this table *continuously* replicated (by setting an appropriate flag). Otherwise, the table has to be manually *sync*-ed. We discuss the impact of this design decision on the overall performance of the system in detail in Section 5. We periodically delete old entries from R_{ijk} to keep its size manageable.

We also need to maintain metadata in partition P_k recording which clusters are pushed, and which clusters are not (consulting R_{ijk} alone is not sufficient since partial contents of a node may exist in R_{ijk} even if it is not actively replicated). There are two pieces of information that we maintain: first, we globally replicate the information about which clusters are replicated to which partitions. Since the number of clusters is typically small, the size of this metadata is not significant. Further, the replication decisions are not changed very frequently, and so keeping this information up-to-date does not impose a significant cost. Secondly, for each node, we maintain the cluster membership for all its cross-partition neighbors. This coupled with the cluster replication information enables us to deduce whether a cross-partition neighbor is actively replicated (pushed) or not. Note that, the cluster membership information is largely static, and is not expected to change frequently. If we were to instead explicitly maintain the information about whether a cross-partition neighbor is replicated with each node, the cost of changing the replication decisions would be prohibitive.

How and When to Make the Replication Decisions: We present our algorithms for making the replication decisions in the next section. Here we present a brief overview.

- The key information that we use in making the replication decisions are the read/write access patterns for different nodes. We maintain this information with the nodes at a fine granularity, by maintaining two histograms for each node. As an example, for a social network, we would wish to maintain histograms spanning a day, and we may capture information at 5-minute granularities (giving us a total of 120 entries). We use the histogram as a predictive model for future node access patterns. However, more sophisticated predictive models could be plugged in instead. We discuss this further in Section 3.2.
- For every cluster-partition pair $\langle C_{ij}, P_j \rangle$, we analyze the aggregate read/write histograms of C_{ij} and P_k to choose the *switch points*, i.e., the times at which we should change the decision for replicating C_{ij} to P_k. As we discuss in the next section, this is actually not optimal since it overestimates the number of pull messages required. However, not only can we do this very efficiently (we present a linear-time optimal algorithm), but we can also make the decisions independently for each cluster-partition pair affording us significant more flexibility.
- When the replication decision for a cluster-partition pair $\langle C_{ij}, P_k \rangle$ is changed from push to pull, we need to ensure that the fairness criterion for the nodes in P_k is not violated. We could attempt to do a joint optimization of all the decisions involving P_k to ensure that it does not happen. However, the cost of doing that would be prohibitive, and further the decisions can no longer be made in a decentralized fashion. Instead we reactively address this problem by heuristically adjusting some of the decisions for P_k to guarantee fairness.

In the rest of section, we elaborate on the motivation behind monitoring access patterns and our clustering technique.

3.2 Monitoring Access Patterns

Many approaches have been proposed in the past for making replication decisions based on the node read/write frequencies to minimize the network communication while decreasing query latencies. Here we present an approach to exploit *periodic patterns* in the read/write accesses, often seen in applications like social networks [4, 13], to further reduce the communication costs. We illustrate this through a simple example shown in Figure 3. Here for two nodes w and v that are connected to each other but are in different

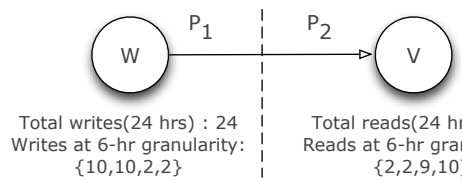

Figure 3: Illustrating benefits of fine-grained decision making: Making decisions at 6-hr granularity will result in a total cost of 8 instead of 23.

partitions, we have that over the course of the day, w is predicted to be updated 24 times, and whereas v is predicted to be read (causing a read on w) 23 times. Assuming the push and pull costs are identical, we would expect the decision of whether to push the updates to w to the partition containing v or not to be largely immaterial. However, when we look at fine granularity access patterns, we can see that the two nodes are active at different times of the day, and we can exploit that to significantly reduce the total communication cost, by having v pull the updates from w during the first half of the day, and having w push the updates to v in the second half of the day. In the context of human-activity centered networks like social networks, we expect such patterns to be ubiquitous in practice.

To fully exploit such patterns, we collect fine granularity information about the node access patterns. Specifically, for each node we maintain two equi-width histograms, one that captures the update activity, and one that captures the read activity. Both of these histograms are maintained along with the node information in the CouchDB server. We will assume that the histogram spans 24 hours in our discussion; in general, we can either learn an appropriate period, or set it based on the application. We use these histograms as a predictive model for the node activity in future.

For a node n_i, we denote by $\omega(n_i, t)$ the predicted update frequency for that node during the time interval starting at t (recall that the width of the histogram buckets is fixed and hence we omit it from the notation). We denote cumulative write frequency for all nodes in a cluster C_{ij} for that time interval by $\omega(C_{ij}, t)$. We similarly define $\rho(n_i, t)$ to denote the read frequency for n_i. Finally, we denote by $\rho(P_k, C_{ij}, t)$ the cumulative read frequency for P_k with respect to the cluster C_{ij} (i.e., the number of reads in P_k that require access to a node in C_{ij}).

3.3 Clustering

As we discussed above, we cluster all the nodes in a partition into multiple clusters, and make replication decisions for the cluster as a unit. However, we note that this does not mean that all the nodes in the cluster are replicated as a unit. For a given node n, if it does not have a neighbor in a partition P_j, then it will never be replicated at that partition. Clustering is a critical component of our overall framework for several reasons.

First, since we would like to be able to switch the replication decisions frequently to exploit the fine-grained read/write frequencies, the cost of changing these decisions must be sufficiently low. The major part of this cost is changing the appropriate metadata information as discussed above. By having a small number of clusters, we can reduce the number of required entries that need to be updated after a decision is changed. Second, clustering also helps us in reducing the cost of making the replication decisions itself, both because the number of decisions to be made is smaller, and also because the inputs to the optimization algorithm are smaller. Third, clustering helps us avoid *overfitting*. Fourth, clustering makes node addition/deletion easier to handle as we can change node's association to cluster transparently w.r.t. other system operations. By making decisions for clusters of nodes together, we are in essence

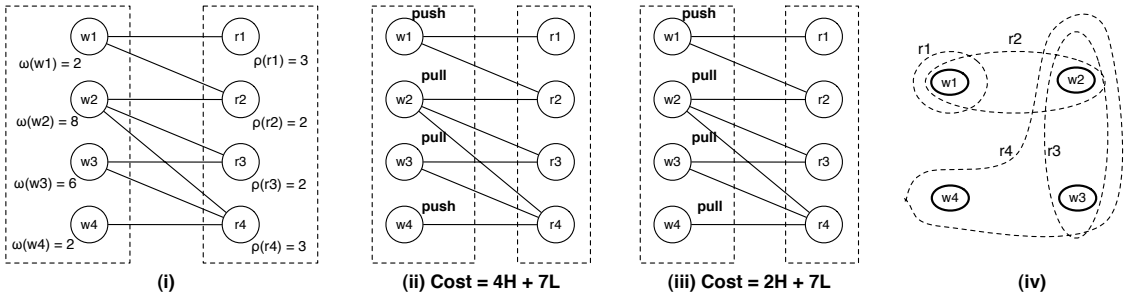

Figure 4: (i) An example instance where we consider whether to replicate the single-node clusters from the left partition to the right partition; (ii) Making decisions for each cluster-partition pair independently; (iii) Optimal decisions; (iv) Modeling the problem instance as a weighted hypergraph.

averaging their frequency histograms, and that can help us in better handling the day-to-day variations in the read/write frequencies.

To ensure that clustering does not reduce the benefits of fine-grained monitoring, we create the clusters by grouping together the nodes that have similar write frequency histograms. More specifically, we treat the write frequency histogram as a vector, and use the standard *k-means* algorithm to the clustering. We discuss the impact of different choices of k in our experimental evaluation.

We note that clustering is done offline, and we could use sampling techniques to do it more efficiently. When a new node is added to the system, we assign it to a random cluster first, and reconsider the decision for it after sufficient information has been collected for it.

4. MAKING REPLICATION DECISIONS

In this section, we present our algorithms for making replication decisions. We assume that the clustering decisions are already made (using the *k-means* algorithm), and design techniques to make the cluster-level replication decisions. We begin with a formal problem definition, and analyze the complexity of the problem. We then present an optimal linear-time algorithm for making the replication decisions for a given cluster-partition pair in isolation ignoring the fairness requirement (as we discuss below, this is not an overall optimal since the decisions for the clusters on a single partition are coupled and cannot be made independently). We then present an algorithm for modifying the resulting solution to guarantee fairness.

4.1 Problem Definition

As before let $G(V, E)$ denote the data graph, $P_1, \cdots, P_l$ denote the hash partitioning of the graph, and let C_{ij} denote the clusters. We assume that fine-grained read/write frequency histograms are provided as input. For the bucket that starts at t, we let $\omega(n_i, t), \omega(C_{ij}, t)$ denote write frequencies for n_i and C_{ij}; $\rho(n_i, t)$ denote the read frequency for n_i; and , $\rho(P_k, C_{ij}, t)$ denote the cumulative read frequency for P_k with respect to the cluster C_{ij}.

Next we elaborate on our cost model. We note that the total amount of information that needs to be transmitted across the network is independent of the replication decisions made, and depends only on the partitioning of the graph (which is itself fixed a priori). This is because: (1) the node updates are assumed to be append-only so waiting to send an update does not eliminate the need to send it, and (2) we cache all the information that is transmitted from one partition to the other partition. Further, even if these assumptions were not true, for small messages, the size of the payload usually does not impact the overall cost of sending the message significantly. Hence, our goal reduces to minimizing the number of messages that are needed. Let H denote the cost of one push message sent because of a node update, and let L denote the cost of a single pull message sent from one partition to the other. We allow H and L to be different from each other.

Given this, our optimization problem is to make the replication decisions for each cluster-partition pair for each time interval, so that the total communication cost is minimized and the fairness criterion is not violated for any node.

It is easy to capture the read/write frequencies at very fine granularities (e.g., at 5-minute granularity), however it would not be advisable to reconsider the replication decisions that frequently. We can choose when to make the replication decisions in a cost-based fashion (by somehow quantifying the cost of making the replication decisions into the problem formulation). However, the two costs are not directly comparable. Hence, for now, we assume that we have already chosen a coarser granularity at which to make these decisions (we evaluate the effect of this choice in our experimental evaluation).

4.2 Analysis

Figure 4(i) shows an example data graph partitioned across two partitions that we use to illustrate the challenges with solving this problem. We assume that the cluster size is set to 1 (i.e., each node is a cluster by itself). We omit the intra-partition edges, and also the time interval annotation for clarity. We consider the question of whether to replicate the clusters from P_1 to P_2, and use the write frequencies for the nodes in P_1, and the read frequencies for the nodes in P_2. We call a node in P_1 a writer node, and a node in P_2 a reader node.

Following prior work [43], one option is to make the replication decision for each pair of nodes, one writer and one reader, independently. Clearly that would be significantly suboptimal, since we ignore that there may be multiple readers connected to the same writer. Instead, we can make the decision for each writer node in P_1 independently from the other writer nodes, by considering all reader nodes from P_2. In other words, we can make the decisions for each cluster-partition pair. Figure 4(ii) shows the resulting decisions. For example, we choose to push w_1 since the total read frequency of r_1 and r_2 exceeds its write frequency (here we assume that $H = L$).

These decisions are however suboptimal. This is because it is useless to replicate w_4 in the above instance without replicating w_2 and w_3, because of the node r_4. Since neither of w_2 and w_3 is replicated, when doing a query at node r_4, we will have to pull some information from P_1. We can collect the information from w_4 at the same time (recall that we only count the number of messages in our cost model – the total amount of data transmitted across the network is constant). Figure 4(iii) shows the optimal decisions.

As it turns out, it is possible to make these decisions optimally in polynomial time (note that we ignore the fairness criteria here). Figure 4(iv) shows another way to model this problem, where we turn the problem instance into a weighted hypergraph. The nodes of the hypergraph are the nodes in P_1, with the write frequencies used as weights. For each reader node, we add a hyperedge to this graph over the nodes that it is connected, and weight of the hyperedge is the read frequency of the node. Now, say a subset S_1 of the nodes in P_1 are replicated. Let S_2 denote the hyperedges that are completely covered by S_1, i.e., hyperedges that only contain nodes from S_1. Then, the total cost for these two partitions is:

$$\sum_{v \in S_1} \omega(v) + \sum_{u \notin S_2} \rho(u) = C + \sum_{v \in S_1} \omega(v) - \sum_{u \in S_2} \rho(u)$$

where $C = \sum \rho(u)$ is a constant. In other words, we pay the cost of one push message per node in S_1 and one pull message per node not in S_2. This problem is similar to the well-studied problem of finding the sub-hypergraph of a hypergraph with the maximum density (the standard density metric is $\sum_{u \in S_2} \rho(u) / \sum_{v \in S_1} \omega(v)$). We can use similar max-flow based techniques to solve our problem (in fact the above optimization goal is simpler), however we omit the details because we do not use such an algorithm in our system for several reasons. First, even though the problem can be solved in polynomial time [30, 12], the complexity of the algorithm is still quite high. This coupled with the fact that the size of the input is large (the number of hyperedges is equal to the number of nodes in P_2), that approach would be infeasible. We instead use a heuristic that we discuss below that greedily makes a local decision for each cluster-partition pair, significantly reducing both the input size and hence the overall complexity.

So far we have ignored the fairness criterion. For the two partitions P_1 and P_2 as above, the fairness criterion requires that, for every reader node in P_2, at least a τ fraction of its neighbors be replicated. The problem of finding the optimal replication decisions given a fairness requirement is unfortunately NP-Hard. Note that, when $\tau = 1$, this problem does not reduce to the problem considered by Pujol et al. [41] (who prove their partitioning problem to be NP-Hard). This is because they are trying solve the graph partitioning problem itself, to come up with a good partitioning of the graph. In our case, the solution for $\tau = 1$ is trivial – we must replicate every node into every partition that it is connected to (we call this the *all-push* solution in our experimental evaluation).

Theorem 1 *The problem of optimally replicating nodes to guarantee fairness is NP-Hard.*

PROOF. We show a reduction from the *set cover* problem. In a set cover instance, we are given a collection of sets $S_1, \cdots, S_n$ over a universe $U = \{e_1, \cdots, e_m\}$ (i.e., $S_i \subseteq U$, and $\cup S_i = U$), and the goal is to find the smallest collection of sets such that every element in U is contained in at least one of those sets. Given a set cover instance, we create an instance of our problem with two partitions as follows.

Following the above terminology, let P_1 be the partition that contains the writer nodes, and let P_2 be the partition that contains the reader nodes. For each set S_i, we add a writer node w_i in P_1. For each element in the universe e_j, we create a reader node r_j in P_2. We connect w_i to r_j if $e_j \in S_i$. Let τ be the fairness threshold. We connect each of r_j to sufficient nodes in P_2 such that we are exactly one neighbor short of achieving fairness for r_j. For instance, if $\tau = 0.5$ and if r_j is connected to 5 nodes in P_1, then we connect r_j to 4 nodes in P_2 (adding dummy nodes if needed). In other words, for every node r_j, we need to replicate exactly one of its neighbors from P_1 to guarantee fairness.

Finally, we set the read frequencies for the nodes in P_2 to be very low, and write frequencies for the nodes in P_1 to be sufficiently high so that by default none of the nodes in P_1 will be replicated to P_2.

Given this setup, it is easy to see that choosing the minimum number of nodes from P_1 to push to guarantee fairness for all nodes in P_2 is identical to the set cover problem. □

4.3 Proposed Algorithm

In this section, we present our overall algorithm for making and changing replication decisions. The algorithm is decentralized by nature, and does not require global coordination (however, replication managers do need to communicate statistics and the replication decisions to other replication managers). The algorithm operates in two phases. In the first phase, at each partition P_i and for each cluster C_{ij} in it, we decide whether to replicate the cluster C_{ij} at each of the other partitions, based purely on the read/write frequency histograms, and ignoring the fairness criterion. For efficiency, we do not make global decisions even within a site, and instead we make independent decisions for each cluster-partition pair $\langle C_{ij}, P_k \rangle, i \neq k$. Given the cumulative read/write frequency histograms for the cluster and the partition, we present a linear-time optimal algorithm to decide the switch points, i.e., the points at which the replication decisions should be switched.

In the second phase, run at each partition independently, we enforce fairness criterion for all the nodes at that partition by switching some replication decisions for clusters at other partitions from push to pull. As discussed above, this problem is NP-Hard in general, and we use a greedy heuristic based on the standard greedy heuristic for solving the set cover problem.

4.3.1 Optimal Decisions for a Cluster-Partition Pair

Next we present an optimal linear-time algorithm for making decisions of when to switch replication decisions for a given cluster-partition pair $\langle C_{ij}, P_k \rangle, i \neq k$. Let $\omega(C_{ij}, t)$ denote the write frequencies for C_{ij} and $\rho(P_k, C_{ij}, t)$ denote the read frequencies for P_k w.r.t. C_{ij}. We assume that we are given a constraint on the maximum number of times we are allowed to switch the replication decision, C (without any such constraint, we would make a different replication decision for each time interval). We can instead assign a cost to making a replication decision, and optimize for the lower total cost – the algorithm below can be easily adapted to that effect.

Let there be n buckets in the frequency histogram. For each bucket (i.e., each time interval), we compute the benefit of replicating C_{ij} over doing a pull from P_k. For time interval t, this is computed as:

$$b_t = \rho(P_k, C_{ij}, t) \times L - \omega(C_{ij}, t) \times H$$

Thus we have n numbers, denoted $b_1, \cdots, b_n$, that represent the benefit of a push over a pull for the corresponding intervals. Note that some of these numbers may be negative – if all of the numbers are positive, then we would always push C_{ij} to P_k.

We first compress this sequence of numbers by coalescing the entries with the same sign together. In other words, if we have a contiguous sequence of positive numbers, we will replace it with a single number that is the sum of those numbers. Similarly, we would coalesce any sequences of negative numbers. The rationale behind this is that, we would never want to switch replication decisions in the middle of such a sequence.

Let $s_1, \cdots, s_m$ denote the resulting sequence of alternating positive and negative numbers. Let $opt(C', push, i)$ denote the optimal cost for the subproblem $s_i, \cdots, s_m$ using at most C' switches and assuming that the decision for the time period corresponding

to s_i is a PUSH. We similarly define $opt(C', pull, i)$. Then we can see that:

$$opt(C', push, i) = s_i + max\{ \quad opt(C', push, i+1), \\ opt(C'-1, pull, i+1)\}$$

In essence, we check both possibilities for s_{i+1}, PUSH or PULL, and choose the best of the two. Similarly,

$$opt(C', pull, i) = -s_i + max\{ \quad opt(C', pull, i+1), \\ opt(C'-1, push, i+1)\}$$

Here we have to use $-s_i$ since s_i is benefit of doing push and we are doing a pull in the time period corresponding to s_i. The base case of the recursion is when $C = 0$ at which point we simply return the sum of the remaining items, possibly negated. The computational complexity of the algorithm can be seen to $O(nC)$.

4.3.2 Guaranteeing Fairness

Finally, we discuss how we ensure that the fairness requirement is satisfied for all nodes. The replication manager at each partition runs this algorithm independently of the other partitions, and may change some of the replication decisions for clusters at other partitions with respect to that partition.

Since the problem is NP-Hard, we develop a heuristic based on the standard greedy heuristic for set cover. For a partition P_k, let Γ_k denote the nodes for which fairness guarantee is not satisfied. Let C_{ij} be a cluster at another partition P_i which is *not* replicated at P_k, i.e., the decision for $\langle C_{ij}, P_k \rangle$ is a pull. Then let $benefit_{ijk}$ denote the total benefit of changing the decision for that cluster-partition pair. This is computed as:

$$benefit_{ijk} = \sum_{v \in \Gamma_k} |nei(v) \cap C_{ij}| - remaining(v)$$

where $nei(v)$ denote the set of neighbors of v, and $remaining(v)$ denote the number of neighbors after replicating which the fairness criterion would be satisfied for v. Further, let $cost_{ijk}$ be the cost of switching the decision for C_{ij} from a pull to a push. We greedily choose the cluster to be replicated that has the highest $benefit_{ijk}/cost_{ijk}$ ratio, and continue until the fairness criterion is met for all nodes.

5. EVALUATION

In this section, we present a comprehensive experimental evaluation using our prototype system. Lacking real datasets with sufficient detail, we constructed a social network graph based on a commonly used social network model, and also constructed a trace of user activity on a social network by gathering user activity data on Twitter and extrapolating. We focus on the "fetch updates from all my neighbors" queries which are the most common class of queries in such networks. As discussed in Section 2, our system is built on top of CouchDB, and we used Amazon EC2 to run our experiments. Our key findings can be summarized as follows:

- Our hybrid replication approach results in significant savings in the network communication cost over the baseline approaches.
- The granularity at which we make push/pull decisions plays an important role in determining how much savings we can obtain.
- The hash-based partitioning scheme results in balanced network and CPU load in our system.
- Our fairness guarantee reduces the average number of pulls required to answer read queries.

We begin with describing the dataset, and the experimental setup.

5.1 Dataset

We constructed our data set to match the workload of a social network. We have used a *preferential attachment* model to generate the data graph which has been shown to model a social network very well [8, 3, 2, 32]. The network is generated by adding one node at a time, and the new nodes are preferentially connected to the existing nodes with higher degrees (i.e., the probability of connecting to an existing node depends on the degree of that node). Most of our experiments were run on a social network containing 1.8 million nodes, and approximately 18 million edges (generated with the preferential attachment factor of 10).

The second key component of our simulated dataset is the user activity patterns. We chose 100 Twitter users with sufficient number of tweets and downloaded their tweets to get their access trace. This trace only gives us the write frequencies of the nodes. In our experiment, we have assumed that the read frequency of a node is linearly related to its write frequency. In reality this linear factor might be different for different users; however, we assume a constant read to write ratio for all nodes. From the access traces, we created write frequency histograms and linearly scaled those to get the read frequency histograms. Once we had the pool of histograms, we assigned them to the nodes in the network. Motivated by recent work on modeling user activity on Twitter [15], we used the following assignment process. We assigned histograms to the graph nodes one at a time. When considering which histogram to assign to a node, we check the histograms already assigned to the other nodes in the same partition, and find the histogram that has been assigned to the largest number of nodes in that partition. We assign the same histogram to the node under consideration with 50% probability, otherwise we choose any one of the remaining histograms with equal probability.

However, we do not use these assigned histograms directly. For each user, we instead randomized the assigned histogram by generating a trace by treating the histogram as a probability distribution, and then building a histogram on the generated trace. This ensures sufficient diversity in the user histograms across the network.

5.2 Experimental Setup

We ran our experiments on Amazon EC2 infrastructure using 7 EC2 instances (1 instance is equivalent to a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor, 1 core, and 1.7G of memory). We used 1 instance to play back the write and read traces (i.e., to send updates and queries to the system), and rest of the instances to host the graph. Each server had a CouchDB server and a copy of replication manager running. As we noted earlier, our network has 1.8 million nodes, which translated into about 300,000 nodes per partition. We used a trace containing a total of approximately 25 million events (reads and writes), corresponding to a single day. The default write to read ratio in the query workload was set to 0.2 (i.e., there are 5 read queries for every update). For each of the experiments, we ran the trace against the system (after selecting the appropriate replication approach), and computed the total network messages. For most of our plots, we plot the *average number of network messages per site*.

We compared three approaches: (1) *all-pull*, where we do not do any replication, (2) *all-push*, where the nodes are replicated sufficient to guarantee no pulls would be needed (i.e., local semantics), and (3) *hybrid*, our hybrid approach. Unless otherwise specified, the number of clusters at each partition was set to 6.

5.3 Evaluation Metric

Our main evaluation metric is the amount of network communication in terms of messages [46] exchanged across all the servers

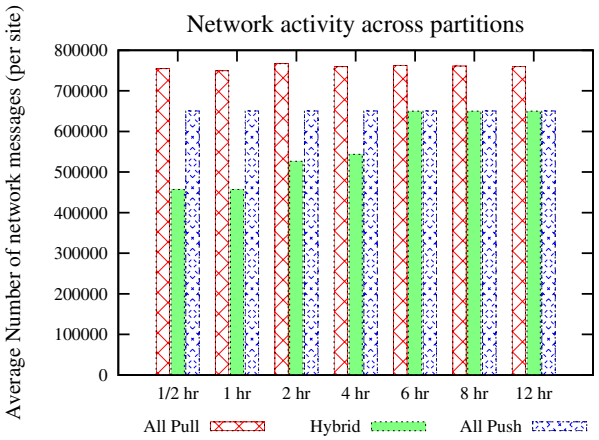

Figure 5: Making fine-grained decisions can result in almost 33% savings over coarse-grained decisions

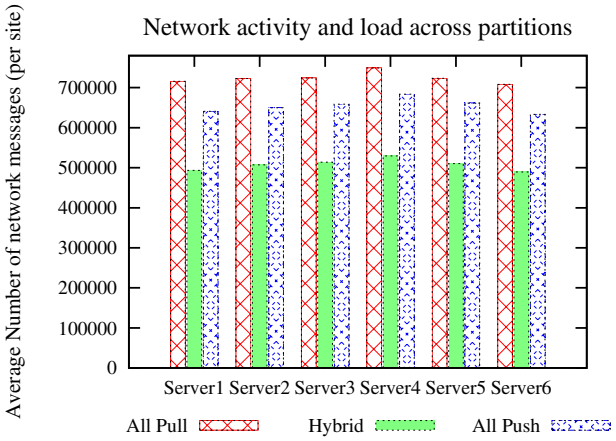

Figure 6: Hash partitioning results in almost uniform load across the partitions

as logged by our replication middleware. As we discussed earlier, once a data item is replicated to a partition, it is cached and will not be transferred again. Because of this, the amount of data transfer across the servers is independent of the replication decisions that are made (we can easily modify the cost functions in our algorithms to account for this if desired). Hence for most of the results, we report the total number of push and pull messages (i.e., we assume $H = L = 1$).

For a *push* decision, we use continuous replication of CouchDB and there is a message involved every time the corresponding graph node is updated. However, the way we count the number of *pull* messages is slightly different, and reflects the constraints imposed by CouchDB and our setup. In fact, this results in a significant underestimation in the number of pull messages as some of our experiments also illustrate.

The way a *pull* works in our system is that, the replication manager asks CouchDB to *sync* the appropriate replication table (see Section 3.1). However since the replication tables correspond to clusters, all updates to that cluster are pulled from the cluster's home partition. To amortize the cost of this, we enforce a minimum gap between two pulls corresponding to the same cluster by using a *timeout*. In other words, if a cluster has been recently pulled, we do not pull it again until the timeout expires. In our experimental evaluation, the timeout is set to 800ms, so the data can be at most 800ms stale (which is reasonable in a social network application). We further discuss the rationale in Section 5.4.7.

5.4 Results

5.4.1 Impact of Histogram Granularity

We start with a set of experiments to verify our hypothesis that by making decisions in a fine-grained manner can result in significant savings. Figure 5 shows the results for this experiment. Here we varied the histogram granularity from 1/2 hour to 12 hours, and counted the total number of messages that were needed. The *all-pull* and *all-push* approaches are unaffected by this, however, we can see that by making decisions at the finest granularity, i.e., every 1/2 hour, resulted in almost 33% savings over coarse-grained decisions. This validates our hypothesis that we can exploit the user activity patterns to reduce the network communication costs.

We also note that overall our default workload is read-heavy, and hence all-push solution is usually better than all-pull solution (although it results in higher memory consumption). As we move toward coarse-grained histograms, we observed that most of the replication decisions became push. But our algorithm is able to exploit the diversity in the access patterns when making decisions at finer granularities to achieve significant savings.

5.4.2 Bandwidth Consumption and Network Load

Figure 6 shows the total network communication across the servers. For each server, we aggregated the network communication resulting from writes happening to the corresponding partition, and the reads directed to the partition. As we can see, all the approaches resulted in fairly balanced load across the partitions, with *hybrid* achieving almost 20% savings over *all-push* in all cases. This can be attributed to the hash partitioning scheme that we use, which guarantees that the overall read and write distributions across the partitions are largely uniform.

5.4.3 Varying the Number of Clusters

Next we study the effect of k, the number of clusters in each partition. We varied the number of clusters from 4 to 9, and we show the results in Figure 7. Along with the network communication costs (plotted on the *left* y-axis), we also plot the size of the cluster mapping table, the metadata that is needed to decide which of a node's neighbors are replicated (on the *right* y-axis). As we can see, as the number of clusters increases the size of the cluster mapping table increases as expected. What is somewhat counter-intuitive is that the total communication cost also increases beyond 6. We expect that with large numbers of clusters, we can make more fine-grained decisions which should aid in reducing the total network communication cost.

The reason for this is somewhat subtle, and has to do with the way *pulls* are handled in our system. Recall that a single pull actually syncs an entire cluster, i.e., it propagates all updates for a single cluster from the home partition to the partition making the pull. Thus increasing the cluster sizes results in an decrease in the number of pulls that are required. We expect that if we were counting the number of pulls explicitly, that would result in the behavior as expected (however, in a read-heavy workload, that would imply that the all-push solution would always be better by a margin).

5.4.4 Varying Write-Read Ratio

We examine how the replication techniques perform for workloads that have different mixes of reads and writes. We simply varied the read/write ratio of the workload and calculated the average cost in terms of total number of communications, incurred by the three approaches. For hybrid, we also plot the costs when the fairness threshold τ is set to 0.5. Figure 9 shows the results of

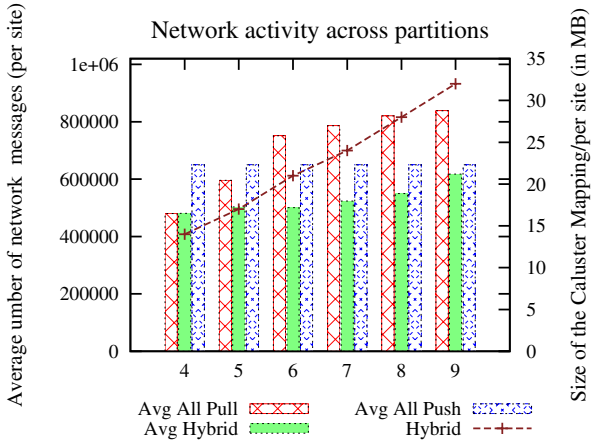

Figure 7: Varying the number of clusters

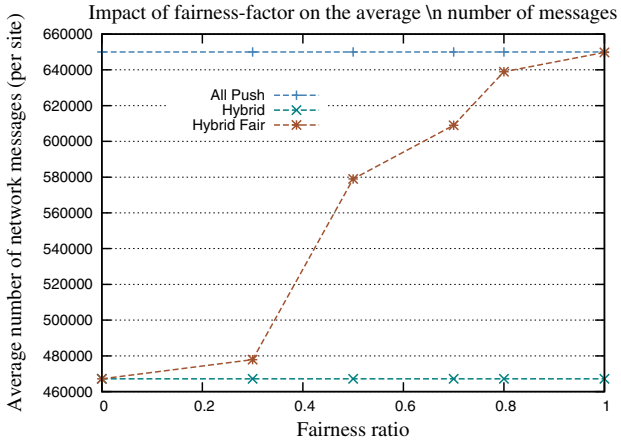

Figure 8: Impact of fairness factor on network communication

this experiment. We kept the number of reads more or less constant, and varied the number of writes. As we can see, our approach did consistently better than the other two. Since the number of reads were almost constant, the performance of the *all-pull* approach does not change significantly. However, the costs of the other three approaches increase almost linearly, with base *hybrid* showing the best performance. In fact, with low write/read ratio, the hybrid approach is almost equivalent to *all-push*, but as the write frequency increases, pull decisions are favored, and hybrid starts performing much better than *all-push*. With fairness threshold set to 0.5, the *hybrid* approach does worse than the basic *hybrid* approach, because in order to guarantee fairness, it is forced to do more active replication than optimal.

5.4.5 *Varying the Fairness Threshold*

Next we investigate the impact of fairness threshold (Section 2.3). We vary the threshold from 0 (in which case, the approach is the basic hybrid approach) to 1 (equivalent to *all-push*). In general, increasing fairness threshold will result in more push decisions than is optimal. Figure 8 illustrates this point where we plot the average network communication cost as before. As we can see, increasing the fairness threshold results in a move toward all-push solution. We do not plot the cost of *all-pull* in this case for clarity.

Figure 5.4.5 shows the latencies of read queries for the different approaches. As we expect, the latency is lowest for the *all-push* solution, with an absolute value of about *2ms*. The cost for the *all-pull* approach is relatively quite high, almost *22ms*. The hybrid approach is somewhere in-between, with somewhat higher latencies than the *all-push* approach but with, as we saw earlier, significantly lower network activity. As expected, when the fairness factor is varied, the average latency drops reaching *2ms* with $\tau = 1$.

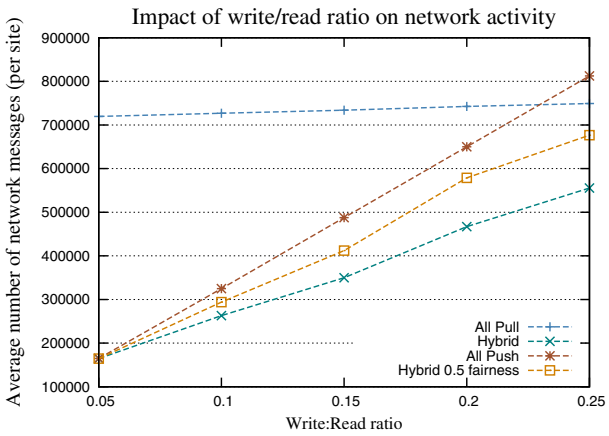

Figure 9: Increasing the write:read ratio results in favoring pulls over pushes

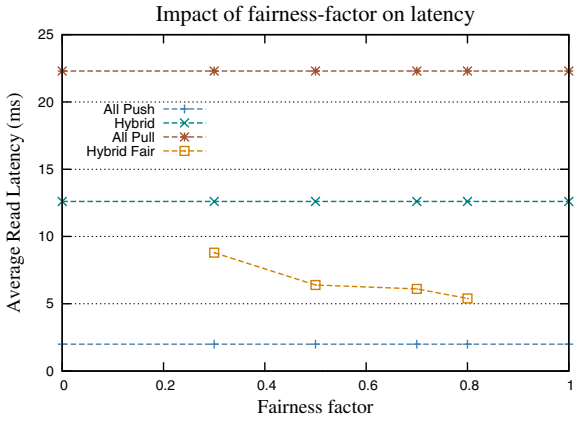

Figure 10: Impact of the fairness factor on read query latencies

This set of experiments not only validates our assertion that fairness threshold is an important novel consideration for highly dynamic graph databases, but also shows the efficiency of our system at processing queries with very low absolute latencies.

5.4.6 *Varying the Graph Density*

We also investigated how our system performs as we increase the density of the graph. We changed the preferential attachment factor (PA) in the graph generator to create graph with same number of nodes but with different densities. Here by density we mean the average number of neighbors of a node. We changed the attachment factor from 5 to 20 and analyzed the performance of different replication strategies. This results in varying the average degree of the graph from about 5 to about 20.

Figure 11 shows that our hybrid replication techniques continues to perform better than *all-pull* and *all-push* approaches. One point to note here is, as we increase the density of the graph the performance of our hybrid techniques degrades, and moves towards *all-push*. The reason for this is, though the write frequency of the nodes remains the same, the cumulative read frequency increases on an average. Thus our already read-heavy workload becomes even more skewed towards reads, resulting a preference for the *all-push* approach. We note that the cost of the *all-pull* approach increases as expected with the increase in the graph density.

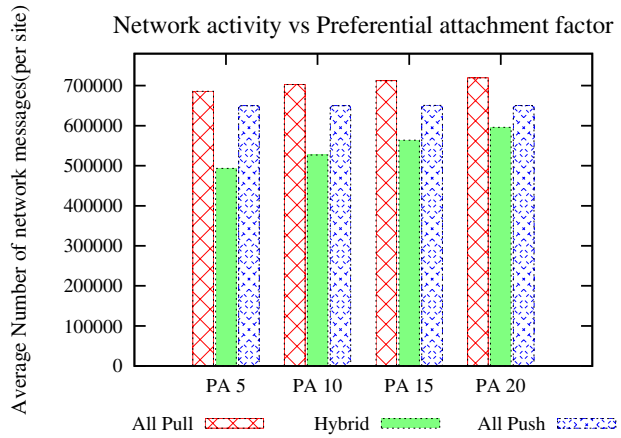

Figure 11: Increasing the density of the graph increases the number of reads, and decreases the opportunities to exploit the read/write skew

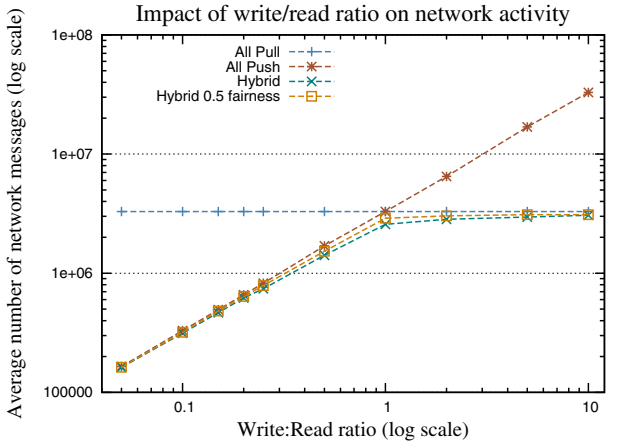

Figure 12: Comparing the techniques without push/pull timeout (note that both axes are in log-scale)

5.4.7 *Effect of the Pull Timeout*

Finally we discuss the $timeout$ used for $pull$ in our system. We note that we don't have such a timeout for $push$. It might seem to be an unfair comparison, but given the read-heavy workload it is a natural system design decision. Intuitively the $timeout$ captures the difference in cost between a $push$ and a $pull$ (we could have also captured this by using appropriate values for H and L). However we also performed a set of experiments with no $timeout$ for $pull$ to further evaluate the uneven use of $timeout$ for $push$ and $pull$. Figure 12 shows how different schemes perform with no $timeout$ parameter for either $push$ or $pull$ and with the read:write ratio varying from 20:1 to 1:10. We can see that $push$ is favored when the workload is read-heavy, and as the number of writes increases, $pull$ is preferred. Our proposed hybrid approach performs better than the best of those two, with the maximum benefit observed when the workload is balanced in terms of reads and writes – the maximum benefit we see is almost 25% (note that both the axes are in log-scale).

6. PREVIOUS WORK

There is an increasing interest in large-scale, distributed graph data management, with several new commercial and open-source systems being developed for that purpose. Some of the key systems include Neo4j [37], HyperGraphDB [31], InfiniteGraph [24], FlockDB [45], GraphBase [10], Trinity [36], Pegasus [28], and Pregel [35]. Neo4j is disk-based transactional graph database which can handle very large graphs, but does not do horizontal partitioning. HypergraphDB, Trinity, InfiniteGraph all support horizontal partitioning and parallel query processing, with the former two supporting richer hypergraph data model. Our techniques can be applied for minimizing the network communication and for reducing query latencies in these systems. On the other hand, Pegasus and Pregel are both batch-processing systems based on the MapReduce framework, and are not aimed at online query processing or dynamic graphs.

There is much work on monitoring read/write frequencies and making replication decisions based on them. For instance, Wolfson et al. propose the *adaptive data replication* approach [46], where they adapt the replica placement based on the read/write patterns. Their algorithm is primarily designed for a tree communication network, but can also handle general graph topologies. There is much subsequent work on this topic (see Kadambi et al. [27] for a recent work on using a similar approach for geographically distributed data management system). However, the problems that we encounter in dynamic graph data management are significantly different and require us to develop new approaches. The primary reason for this is that the data items are not read individually, but are always accessed together in some structured manner, and exploiting that structure is essential in achieving good performance.

Similar decisions about pushing vs pulling also arise in content distribution networks (CDNs), or publish-subscribe systems. Each node in the graph can be seen as both a producer of information and a consumer of information, and we can use techniques like the one proposed by Silberstein et al. [43] for deciding whether to push or pull. However, that work has primarily considered a situation where the producers and consumers are distinct from each other, and usually far apart in the communication network. The reciprocal relationships observed in graph data change the optimization problems quite significantly. Secondly, that work has typically focused purely on the information delivery problem, and the techniques cannot be directly used for executing other types of queries.

7. CONCLUSIONS

In this paper, we presented the design of a distributed system to manage and query large graphs efficiently. Despite the increasing need for graph data management in a variety of applications, there has been a surprising lack of research on general purpose, online graph data management systems. To alleviate the performance concerns stemming from the partitioning of a graph across a large number of machines, we proposed a hybrid replication mechanism that monitors the node read/write frequencies to make fine-grained decisions about when to use active replication vs passive replication. We proposed a clustering technique to reduce the overhead of maintaining the replication decisions, and introduced the novel notion of a fairness guarantee that enables us to trade increased communication for lower latencies. Our prototype system can not only handle large graphs efficiently, but can answer queries with very low latencies. Our experimental results validate the effectiveness of our approach. We are continuing to extend our work in many different directions. In this paper, we focused on a simple type of graph query that requires accessing all the neighbors of a node, and we are working on generalizing this to support other types of queries efficiently by maintaining statistics about which nodes are accessed together. We are also working on designing graph partitioning algorithms that can efficiently handle the highly dynamic and evolving nature of many real-world networks, especially rapid changes to the graph structure itself.

Acknowledgments: This work was supported by Air Force Research Lab (AFRL) under contract FA8750-10-C-0191, by NSF under grant IIS-0916736, and an Amazon AWS in Education Research grant.

8. REFERENCES

[1] B. Amann and M. Scholl. Gram: a graph data model and query languages. In *ACM Hypertext*, 1992.

[2] L. A. N. Amaral, A. Scala, M. Barthelemy, and H. E. Stanley. Classes of small-world networks. *Proceedings of The National Academy of Sciences*, 2000.

[3] A. Barabasi and R. Albert. Emergence of scaling in random networks. *Science*, 1999.

[4] F. Benevenuto, T. Rodrigues, M. Cha, and V. A. F. Almeida. Characterizing user behavior in online social networks. In *Internet Measurement Conference*, 2009.

[5] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using banks. In *ICDE*, 2002.

[6] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics-theory and Experiment*, 2008.

[7] M. Brocheler, A. Pugliese, and V. S. Subrahmanian. COSI: Cloud oriented subgraph identification in massive social networks. In *ASONAM*, 2010.

[8] A. Capocci, V. D. P. Servedio, F. Colaiori, L. S. Buriol, D. Donato, S. Leonardi, and G. Caldarelli. Preferential attachment in the growth of social networks: The internet encyclopedia wikipedia. *Phys. Rev. E*, 2006.

[9] U. V. Catalyurek and C. Aykanat. Patoh: Partitioning tool for hypergraphs. *Bilkent University, Tech. Rep*, 1999.

[10] FactNexus Pty Ltd. Graphbase. `http://www.graphbase.net/`, 2011.

[11] W. Fan, J. Li, S. Ma, N. Tang, Y. Wu, and Y. Wu. Graph pattern matching: from intractable to polynomial time. In *VLDB*, 2010.

[12] G. F. Georgakopoulos and K. Politopoulos. Max-density revisited: a generalization and a more efficient algorithm. *The Computer Journal*, 2007.

[13] S. A. Golder, D. M. Wilkinson, and B. A. Huberman. Rhythms of social interaction: messaging within a massive online network. *CoRR*, abs/cs/0611137, 2006.

[14] K. Golenberg, B. Kimelfeld, and Y. Sagiv. Keyword proximity search in complex data graphs. In *SIGMOD*, 2008.

[15] R. Gonzalez, R. C. Rumin, A. Cuevas, and C. Guerrero. Where are my followers? understanding the locality effect in twitter. *CoRR*, abs/1105.3682, 2011.

[16] R. Guting. GraphDB: Modeling and querying graphs in databases. In *VLDB*, 1994.

[17] M. Gyssens, J. Paredaens, and D. van Gucht. A graph-oriented object database model. In *PODS*, 1990.

[18] J. Hamilton. Scaling linkedin. `http://perspectives.mvdirona.com/2008/06/08/ScalingLinkedIn.aspx`, 2008.

[19] H. He and A. K. Singh. Graphs-at-a-time: query language and access methods for graph databases. In *SIGMOD*, 2008.

[20] B. Hendrickson and R. Leland. An improved spectral graph partitioning algorithm for mapping parallel computations. *SIAM Journal on Scientific Computing*, 1995.

[21] High Scalability Blog. Why are facebook, digg, and twitter so hard to scale? `http://highscalability.com/blog/2009/10/13/why-are-facebook-digg-and-twitter-so-hard-to-scale.html`, 2009.

[22] High Scalibility Blog. Friendster lost lead because of a failure to scale. `http://highscalability.com/blog/2007/11/13/friendster-lost-lead-because-of-a-failure-to-scale.html`, 2007.

[23] J. Huang, D. Abadi, and K. Ren. Scalable SPARQL querying of large RDF graphs. In *VLDB*, 2011.

[24] InfiniteGraph. `http://www.infinitegraph.com/`, 2011.

[25] R. Jin, Y. Xiang, N. Ruan, and H. Wang. Efficiently answering reachability queries on very large directed graphs. In *SIGMOD*, 2008.

[26] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar. Bidirectional expansion for keyword search on graph databases. In *VLDB*, 2005.

[27] S. Kadambi, J. Chen, B. Cooper, D. Lomax, R. Ramakrishnan, A. Silberstein, E. Tam, and H. G. Molina. Where in the world is my data? In *VLDB*, 2011.

[28] U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system. In *ICDM*, 2009.

[29] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. In *SIAM*, 1999.

[30] S. Khuller and B. Saha. On finding dense subgraphs. In *ICALP*, 2009.

[31] Kobrix Software. A general purpose distributed data store, 2011. `http://www.kobrix.com/hgdb.jsp`.

[32] R. Kumar, J. Novak, and A. Tomkins. Structure and evolution of online social networks. In *KDD*, 2006.

[33] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney. Statistical properties of community structure in large social and information networks. In *WWW*, 2008.

[34] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Journal of Internet Mathematics*, 2009.

[35] 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. In *PODC*, 2009.

[36] Microsoft Research. Trinity. `http://research.microsoft.com/en-us/projects/trinity/`, 2011.

[37] Neo4j. Neo4j open source nosql graph database. `http://neo4j.org/`, 2011.

[38] M. Newman. Why social networks are different from other types of networks. *Physical Review E*, 2003.

[39] M. Newman. Modularity and community structure in networks. In *Proc. of The Natl. Academy of Sciences*, 2006.

[40] J. M. Pujol, V. Erramilli, and P. Rodriguez. Divide and conquer: Partitioning online social networks. *CoRR*, abs/0905.4918, 2009.

[41] 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. In *SIGCOMM*, 2010.

[42] L. Sheng and Z. Ozsoyoglu. A graph query language and its query processing. In *ICDE*, 1999.

[43] A. Silberstein, J. Terrace, B. F. Cooper, and R. Ramakrishnan. Feeding Frenzy: Selectively materializing users' event feeds. In *SIGMOD*, 2010.

[44] T. Tran, H. Wang, S. Rudolph, and P. Cimiano. Top-k exploration of query candidates for efficient keyword search on graph-shaped (RDF) data. In *ICDE*, 2009.

[45] FlockDB. `https://github.com/twitter/flockdb`.

[46] O. Wolfson, S. Jajodia, and Y. Huang. An adaptive data replication algorithm. In *TODS*, 1997.

[47] O. Wolfson and A. Milo. The multicast policy and its relationship to replicated data placement. In *TODS*, 1991.

[48] H. Yildirim, V. Chaoji, and M. J. Zaki. GRAIL: Scalable reachability index for large graphs. In *VLDB*, 2010.