Graph Aware Caching Policy For Distributed Graph Stores

Hidayet Aksu
Department of Computer Engineering
Bilkent University, Ankara, Turkey
haksu@cs.bilkent.edu.tr

Mustafa Canim, Yuan-Chi Chang
IBM T.J. Watson Research Center
Yorktown Heights, NY, USA
{mustafa, yuanchi}@us.ibm.com

Ibrahim Korpeoglu, Özgür Ulusoy
Department of Computer Engineering
Bilkent University, Ankara, Turkey
{korpe,oulusoy}@cs.bilkent.edu.tr

***Abstract*—Graph stores are becoming increasingly popular among NOSQL applications seeking flexibility and heterogeneity in managing linked data. Conceptually and in practice, applications ranging from social networks, knowledge representations to Internet of things benefit from graph data stores built on a combination of relational and non-relational technologies aimed at desired performance characteristics. The most common data access pattern in querying graph stores is to traverse from a node to its neighboring nodes. This paper studies the impact of such traversal pattern to common data caching policies in a partitioned data environment where a big graph is distributed across servers in a cluster. We propose and evaluate a new graph aware caching policy designed to keep and evict nodes, edges and their metadata optimized for query traversal pattern. The algorithm distinguishes the topology of the graph as well as the latency of access to the graph nodes and neighbors. We implemented graph aware caching on a distributed data store Apache HBase in the Hadoop family. Performance evaluations showed up to 15x speedup on the benchmark datasets preferring our new graph aware policy over non-aware policies. We also show how to improve the performance of existing caching algorithms for distributed graphs by exploiting the topology information.**

***Index Terms*—Graph Aware; Cache; Big Data Analytics; Distributed Computing; Apache HBase**

I. Introduction

The technique of data caching is well known and widely applied across tiers of computing and storage systems. With the emergence of a new generation of social and mobile applications built on graph data stores or graph data model implemented on legacy database technology, the knowledge about graph traversal based queries can be exploited to devise efficient caching policies that are graph topology aware. Simultaneously, the policy must address metadata properties that come with nodes[1] and edges in the graph, since query predicates are often imposed on those properties to select next steps in the traversal.

Among the use cases of graph data store such as social networks, knowledge representations, and Internet of things, while their respective graph topology may be small and fit on a single server, adding all the metadata properties easily drives up computing and storage requirements beyond the capacity of one server. The context of our investigation thus is anchored on scale out, big data clusters in which the graph and its data is partitioned horizontally across servers in the cluster. We assume topology and metadata about a node or edge are co-located since they are most often accessed together. In addition, as reflected in real-world workload, updates to change graph topology are allowed, which makes one-time static graph clustering less beneficial.

Figure 1 illustrates the context in which our cache solutions fit. The graph data is partitioned and distributed over a cluster of servers with low communication latency. Each distributed node hosts its own data cache and manages the data with the knowledge of local vs. remote graph data. The local data could be found either in the memory cache or in the persistent medium such as disks. A client submits a query to the server hosting the queried root node and the server communicates with its peers to process the client's query.

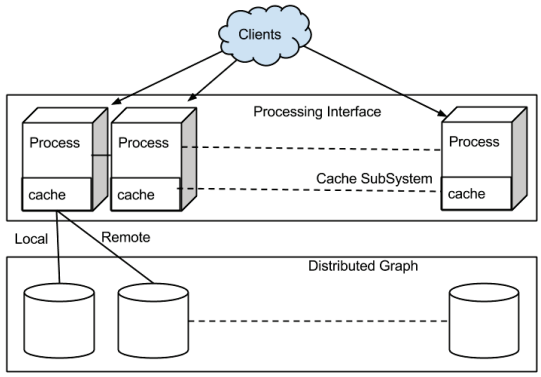


Fig. 1: Cache layer is located between graph storage and distributed processing node. Cache layer knows if a graph file is local or remote and designed to fetch and evict items with graph aware optimization.

Our main contributions in this paper can be summarized as follows:

- We point out that distributed graph stores encounter performance bottlenecks due to slow disk and network accesses. Since optimal graph partitioning problem is

[1]In the rest of this paper we use the terms "node" and "vertex" interchangeably.

978-1-4799-8218-9/15 $31.00 © 2015 IEEE
DOI 10.1109/IC2E.2015.39

known to be NP-Hard we propose a practical solution to cope with this problem.
- We discuss inefficiencies of popular cache policies in processing graph queries through qualitative and quantitative measures.
- We propose a novel cache design which is both graph access and data partition aware.
- We demonstrate how to improve the hit ratio of existing caching policies by exploiting the topology information.
- We present a robust implementation of our algorithms on top of Apache HBase, a horizontally scaling distributed storage platform through its Coprocessor computing framework [1].
- We run our experiments on 10 different real datasets and present detailed experiment results.

The rest of the paper is organized as follows. We first discuss the related work on cache and graph literature in Section II. Then we discuss the graph implementation on big data platform in Section III. Our proposed graph aware cache is presented in Section IV and we evaluate its performance on real social network datasets in Section V. Finally, Section VI concludes the paper.

II. Related Work

Many major large scale applications rely on distributed key-value stores [2], [3], [4], [5]. Meanwhile, distributed graphs are used by many web-scale applications. An effective way to improve the system performance is to deploy a cache layer. Facebook utilizes memcached [6] as a cache layer over its distributed social graph. Memcached is a general-purpose distributed memory cache which employs LRU (see Section IV-B for further details) eviction policy [7] where it groups data into multiple slabs with different sizes. Neo4j [8] is a popular open-source graph database with the ability to shard data across several machines. It provides two levels of caching [9]. The file buffer cache caches the Neo4j durable storage media data to improve both read and write performance. The object cache caches individual vertices and edges and metadata in a traversal optimized format. The object cache is not aware of graph topology and facilitates LRU as for the eviction policy.

On the other hand, Facebook's distributed data store [10], called *TAO*, is designed to serve as a cache layer for Facebook's social graph. It implements its own graph data model and uses a database for persistent storage. *TAO* is the closest work in the literature to our study. *TAO* keeps many copies of sharded graph regions in servers called *Followers* and provides consistency by using single *Leader server* per graph shard to coordinate write operations. *TAO* employs LRU eviction policy similar to memcached.

Pregel [11] provides a system for large-scale graph processing, however, it does not provide a caching layer. It touches on poor locality in graph operations while we study on how to obtain high locality and achieve it through prefetching using graph topology information. Neither *TAO* nor other studies exploit graph characteristics but they handle graph data as ordinary objects. Thus, our study is novel in the sense that it exploits graph specific attributes.

In order to reduce the latency of access in distributed graphs replication based solutions are also proposed in the literature. Mondal et al. in [12] propose an aggressive replication method for low latency querying. The proposed solution however assumes that the graph data resides in the memories and does not distinguish the latency difference between disk and network. Also the replication decision is made for a single node, whereas in our caching policy more than one hop away neighbors could be cached. The third problem is that the proposed algorithm requires keeping track of a histogram of read/write requests for each graph node which could be expensive to maintain.

III. Distributed Graph Handling with Apache HBase

We model interactions between pairs of *objects*, including structured metadata and rich, unstructured textual content, in a graph representation materialized as an adjacency list known as edge table. An edge table is stored and managed as an ordered collection of row records in an *HTable* by Apache HBase [1]. Since Apache HBase is relatively new to the research community, we first describe its architectural foundation briefly to lay the context of its latest feature known as *Coprocessor*, which our algorithms make use of for graph query processing.

A. HBase and Coprocessors

Apache HBase is a non-relational, distributed data management system modeled after Google's BigTable [13]. HBase is developed as a part of the Apache Hadoop project and runs on top of Hadoop Distributed File System (HDFS). Unlike conventional Hadoop whose saved data becomes read-only, HBase supports random, fast insert, update and delete (IUD) access.

Fig. 2 depicts a simplified diagram of HBase with several key components relevant to this chapter. An HBase cluster consists of master servers, which maintain HBase metadata, and region servers, which perform data operations. An HBase table, or HTable, may grow large and get split into multiple HRegions to be distributed across region servers. HTable split operations are managed by HBase by default and can be controlled via API also. In the example of Fig. 2, HTable 1 has four regions managed by region servers 1, 2 and 10 respectively, while HTable 2 has three regions stored in region servers 1 and 2. An HBase client can directly communicate with region servers to read and write data. An HRegion is a single logical block of record data, in which row records are stored starting with a row key, followed by column families and their column values.

HBase's Coprocessor feature was introduced to selectively push computation to the server where user deployed code can

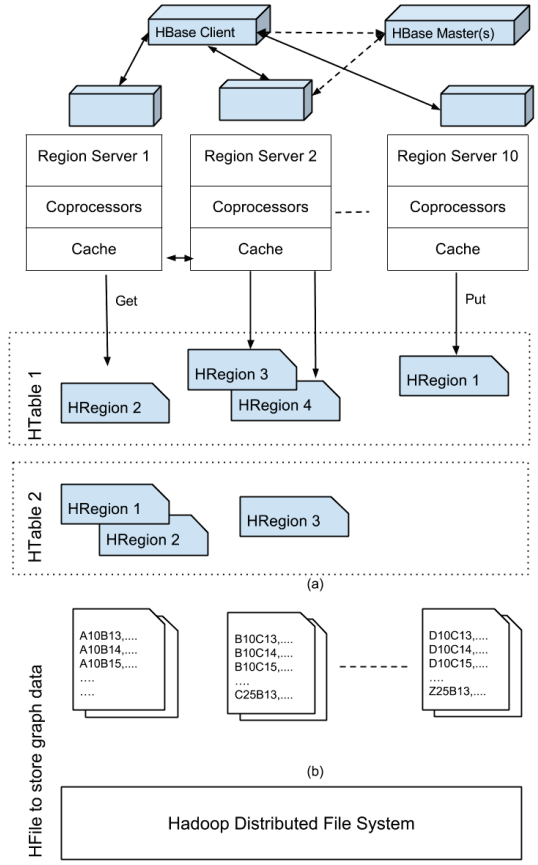


Fig. 2: An HBase cluster consists of one or multiple master servers and region servers, each of which manages range partitioned regions of HBase tables. Coprocessors are user-deployed programs running in the region servers. Cache is used by coprocessors and distributed with graph regions. Cache is located between Coprocessor and HRegions and HRegion accesses are first handled by cache layer.

operate on the data directly without communication overheads for performance benefit. The Endpoint Coprocessor (CP) is a user-deployed program, resembling database stored procedures, that runs natively in region servers. It can be invoked by an HBase client to execute at one or multiple target regions in parallel. Results from the remote executions can be returned directly to the client, or inserted into other HTables in HBase, as exemplified in our algorithms.

Fig. 2 depicts common deployment scenarios for Endpoint CP to access data. A CP may scan every row from the start to the end keys in the HRegion or it may impose filters to retrieve a subset in selected rows and/or selected columns. Note that the row keys are sorted alphanumerically in ascending order in the HRegion and the scan results preserve the order of sorted keys. In addition to reading local data, a CP may be implemented to behave like an HBase client. Through the Scan, Get, Put and Delete methods and their bulk processing variants, a CP can access other HTables hosted in the HBase cluster.

B. Graph Processing on HBase

We map the rich graph representation $G = \{V, E, M, C\}$ to an HTable. We first format the vertex identifier $v \in V$ into a fixed length string $pad(v)$. Extra bytes are padded to make up for identifiers whose length is shorter than the fixed length format. The row key of a vertex v is its padded id $pad(v)$. The row key of an edge $e = \{s, t\} \in E$ is encoded as the concatenation of the fixed length formatted strings of the source vertex $pad(s)$, and the target vertex $pad(t)$. The encoded row key thus will also be a fixed length string $pad(s) + pad(t)$. This encoding convention guarantees that a vertex's row immediately precedes the rows of its outbound edges in an HTable. Fig. 2, includes a simple example of encoded graph table, whose partitioned HRegions are shown across three servers. In this table, a vertex is encoded as a string of three characters such as 'A10', 'B13', 'B25', 'A21', etc. A row key encoded like 'A10B13' represents a graph edge from vertex 'A10' to 'B13'.

k-hop neighbors queries in Section V are implemented in several HBase Coprocessors to achieve maximal parallelism. When non-local vertex neighbors are to be read, a Coprocessor instance issues a *neighbors read message* to the remote HBase region server, which reads and returns the neighbors.

IV. Cache Policies

We implemented a graph library on top of Apache HBase and used commodity servers for experimentation. The implementation details of these system and some use cases are described in [14], [15], [16]. While running our graph algorithms on top of this platform, we experienced that a substantial amount of memory on our servers is available for use. Execution of graph algorithms typically require passing the state information between arbitrary graph nodes which can be located on different physical machines. Even if a bulk message passing protocol is executed between the machines as in Pregel, co-locating the neighboring graph nodes in same machines can bring significant performance advantages. Hence we concluded that available memory space can be exploited for caching purposes. One can argue that existing distributed file system caches can be leveraged for reducing the cost of back and forth communication cost between physical servers. However, as we show in our experiments a graph aware caching policy would perform much better than traditional caching policies. The main reasoning behind this claim is that a graph aware cache will know the graph access patterns while executing the queries. For instance accessing the neighbors of a node is a strong indication that the neighbors of neighbors of this particular node will also get accessed soon. Therefore a topology aware caching scheme can exploit this opportunity. Another problem with distributed caches is that the cache policy may not know whether the graph nodes and edges residing

in the cache are retrieved from the local machine's disk or from a remote machine located in the cluster. Considering the latency of access to either of these locations can benefit the caching algorithm to reduce the latency penalty while executing the graph queries. Before explaining the details of our algorithm we first provide an overview of existing caching schemes.

Cache systems in general aim to predict future data access requests and optimize its resources accordingly. In order to accomplish that goal two data access patterns are considered:

1) **spatial locality**, which indicates that future data accesses will target spatially close data to current accesses.
2) **temporal locality**, which means that future data accesses will target the same data currently accessed.

Spatial locality pattern is used to prefetch data into cache before it is not even accessed for the very first time. Temporal locality pattern on the other hand aims to keep already accessed data in cache to cater for possible future requests received under cache size limitation. Below we discuss how these two techniques can be extended to improve performance of graph processing in distributed graph infrastructures.

A. Exploitation of Spatial Locality for Graph Processing

Generic cache algorithms assume that iteration over logical data order is correlated to physical data order in lower layer in cache hierarchy. For instance, let's consider an iteration over the elements of an array a with an index variable i. While iterating over a, access to $a[i]$ proceeds with an access to $a[i+1]$ where $a[i]$ and $a[i+1]$ are physically co-located in the lower level storage medium. Thus, prefetching $a[i+1]$ upon fetching $a[i]$ would prevent a cache miss due to right prediction of future access.

In contrast to accessing array elements in a sequential fashion, graph traversals do not follow a physical data order in lower cache layers or storage medium. Graph traversal is correlated with graph topology rather than its storage pattern. Hence, prefetching the next element in storage layout would be a poor prediction method while processing the graphs. Majority of the graph algorithms such as k-neighborhood, k-egonet, shortest path require an iterative processing through the neighbor nodes. Therefore, prefetching either one hop or multiple hop neighbors of a particular graph node that is being accessed would increase the hit ratio dramatically. As we discuss below, we designed our caching algorithm with this observation in mind.

B. Eviction Algorithms

According to the temporal locality concept, currently requested data will be requested again in the near future. Based on this assumption, keeping every requested item in the cache would maximize the temporal locality benefit. However, that would not be a feasible approach because of the limited cache space of conventional servers. After inserting the requested items, the cache area gets full and an eviction mechanism is executed to claim empty space. If an evicted item is requested in the future, a cache miss occurs. Therefore, eviction algorithms are designed to minimize the cache misses in order to reduce the penalty of accessing the requested item from a slower medium. Below we briefly explain existing caching policies and later we describe our suggested caching policy for distributed graph platforms.

Least Recently Used (LRU) algorithm [8], [10], [7] is the most popular eviction policy in the literature. LRU keeps track of access order and selects the least recently used item for eviction.

Largest Item First (LIF) algorithm is item size sensitive where it evicts largest item in the cache. Evicting largest item allows cache to store several small items. This algorithm does not assume any correlation with item size and its access frequency.

Smallest Item First (SIF) algorithm is also item size sensitive and it evicts the smallest item in the cache. Thus, the algorithm tries to minimize miss penalty where small items are fetched faster than large items.

LRU algorithm is designed to keep track of recency of access to improve the hit ratio. Largest item first and smallest item first algorithms on the other hand considers the size of the items in the cache for making the decision of which items should be evicted. None of these algorithms however takes into account the topology of graph as well as the latency of access to the items. In a distributed graph architecture the requested items can be found either in a slow medium in the local machine or in a remote server. One can consider the latency of access to the location of the graph nodes while making the decision of eviction. Below we propose a novel caching algorithm suited for distributed graph platforms called "Clock Based Graph Aware algorithm" which aims to consider multiple factors while evicting items from the cache. As we discuss further details the goal of the algorithm is to minimize the access latency for multiple graph nodes rather than maximizing the hit ratio.

C. Clock Based Graph Aware Cache (CBGA)

Distributed graph processing has its own unique challenges when it comes to designing a caching algorithm. One has to take into account many factors such as locality and size of graph components, access patterns and topology of the graph. Below we discuss these factors in more detail and explain how these parameters can be considered in a caching policy.

1) *Local/remote placement*: On a single server, the graph nodes and edges can be located either in disk or in memory. However in a distributed platform the graph components can be found in different locations. These are local memory, local disk, remote memory or remote disk. If we know that bringing a data item from a remote server will be costlier than a local cache miss then the item brought from a remote server might have a higher

Algorithm 1: $AdmissionPolicy$

upon get(Vertex v)
$put\_cache(v, v.neighbors())$
for vertex u in $v.neighbors()$ **do**
  $put\_cache(u, u.neighbors())$
end for

Iteration over a graph makes access to spatially next element in graph topology.

priority to stay in the cache compared to the local data items.

2) *Metadata size*: Graph vertices typically have different number of neighbors and variable metadata sizes. For instance in a social networking graph a popular pop star can have millions of followers whereas many of the individuals might have less than a hundred followers. Also a user might have many metadata information compared to others. Therefore the size of a graph node together with its metadata can be quite different from other nodes in the graph.
3) *Uneven access probabilities*: Graph vertices have different centrality/popularity in networks. Thus some central items are requested more frequently than others. Similar to our previous example a popular user in a social networking site could get much more hit than some ordinary users. The caching policy should take this into account as well.
4) *Iteration on topology*: Many of the graph traversal algorithms require passing the state information along the neighbors of vertices. Accessing a node will imply that its neighbors and also their multi-hop neighbors will also get accessed.

We introduce a new caching policy called, clock-based graph aware caching (CBGA), that takes into account the aforementioned factors. The algorithm aims to exploit spatial locality of the graphs to handle topological closeness instead of storage level co-location. Whenever a graph node is accessed its one hop away neighbors as well as neighbors of neighbors are retrieved and put into the cache. Also all edges connecting these nodes and metadata associated with these edges are stored along this graph node. Prefetching neighbors of neighbors helps popular items to be cached earlier. The details of this admission policy is also described in Algorithm 1. Note that this method brings the multi-hop neighbors of the requested node regardless of their physical location in the system.

Once a graph node is put into the cache a time-to-live value (TTL) is assigned (See Algorithm 2 for the assignment of TTL). The TTL value is used to determine when to evict an item from the cache. Once the cache is full the eviction mechanism is executed to claim empty space. A clock pointer iterates through the items in the cache in a circular fashion.

Algorithm 2: $TTLAssignment$

upon put_cache(v,...) call
if $v \in local\_partition$ **then**
  $latency \leftarrow LOCAL\_ACCESS\_LATENCY$
else
  $latency \leftarrow REMOTE\_ACCESS\_LATENCY$
end if
$size \leftarrow get\_size(v)$
$distance \leftarrow get\_distance(v, s)$
$TTL_v \leftarrow \frac{latency}{(size * distance)}$
$TTL_v \leftarrow normalize(TTL_v)$
return

The TTL value for each cached item is computed at item cache time and normalized to fall into [1-250] range.

If the cache is full the eviction process starts and removes the items from the cache until enough empty space is claimed [2]. Once the eviction process kicks off the TTL value of the cache entries the clock is pointing is reduced by one. If the TTL value of an item reaches zero it is evicted from the cache. The eviction process is described in Algorithm 3. The TTL value is computed using the following formula:

$$TTL = \frac{l}{s * d} \tag{1}$$

where l is the average duration (latency) to fetch an item into the cache from either local server or from a remote server, d is the hop distance between this particular graph node and the graph node that is being queried, and s is the size of the cached item. As the latency parameter, l, increases, a bigger TTL value is assigned which makes it harder to remove the graph node from the cache. For instance if the access latency for a remote graph node is costlier than bringing it from a local disk then a higher priority can be given to remote graph nodes. In our graph platform implemented on HBase it is easy to distinguish local and remote graph nodes as they are partitioned into ordered key regions. Note that a remote item could be either in the cache of the remote server or the remote disk. In order to know if a remote item is in the remote cache or not servers can periodically broadcast a Bloom filter of their cache content and other servers can check if the requested item is in the remote cache or disk. Based on that the latency of access can be determined. A second option to estimate the cost of bringing a remote item would be to calculate the expected cost. If the average remote cache look up hit ratio is known, then the latency could be estimated by adding the network traversal cost on top of the disk IO cost multiplied by the hit ratio. For instance if the network traversal cost is 20ms and the disk IO cost is 10ms and the hit ratio is about 60%, then

[2]In our experiments the eviction process was executed until 20% of the cache was claimed to be empty.

the expected remote look up cost would be calculated as 20 + (10 * 40/100) = 24ms.

On the other hand as the distance parameter, d, gets bigger the probability of accessing the graph node will be reduced. For instance if the information of a person in a social graph is retrieved it is expected that the immediate neighbors of the person have a higher chance of being accessed than second order friends, which is also called friends-of-friends. Note that if the total distance of a node to all others is relatively small then a higher TTL value will be assigned to this node, thus higher closeness centrality provides a higher cache duration. In our experiments we noticed that considering this aspect increases the hit ratio dramatically.

The third parameter s is inversely proportional to TTL. It helps assigning higher priority to the graph nodes with smaller sizes. As the metadata size of a particular graph node increases it will occupy more space in the cache which will reduce the number of graph nodes cached. Thus, the larger vertices in the cache are more likely to be evicted based on this algorithm. In our experiments we used milliseconds and bytes as for the unit of the parameters l and s respectively. The $normalize$ procedure scales up the computed TTL value to make sure it is bigger than or equal to 1.

Algorithm 3: $EvictionPolicy$

```
1: upon CBGA_evict() call
2: while TRUE do
3:    for item u in cache.items() starting from last index
      do
4:       TTL_u ← TTL_u − 1
5:       if TTL_u ≤ 0 then
6:          evict(u)
7:          return
8:       end if
9:    end for
10: end while
```

When the cache requires the eviction policy to evict items, for each item in the cache *TTL* is decreased by one and the item is evicted if *TTL* is less than 1. Eviction iterations continue until an item is evicted.

CBGA uses eventual consistency model for cache coherency, a relaxed consistency model that is described by Terry et al. [17] and discussed by Werner [18]. Any item in the cache is associated with a TTL value which eventually decreases to zero and causes the item to be evicted. Essentially, any change on items is reflected to the cache after a sufficient period of time which is acceptable for many social network applications, e.g., Facebook [10]. Thus, all copies of an item in the cache will be consistent and reflect all updates to the item.

V. Performance Evaluation

In order to assess the performance of the proposed caching algorithm we installed Apache HBase on top of a cluster and loaded 10 different real graph data crawled from different social networking sites. Our experiments show that CBGA outperforms all other caching policies in terms of both hit ratio and overall execution time. We also modified existing caching schemes to make them topology aware. Our experiments prove that the hit ratio of existing caching algorithms can be improved significantly by considering the structure of graphs as well.

TABLE I: Key characteristics of the datasets used in the experiments

Name	Vertex Count	Bidirectional Edge Count	Ref
Twitter	1.1 M	170 M	[19]
Orkut	3.1 M	234 M	[20]
LiveJournal	5.2 M	144 M	[20]
Flickr	1.8 M	44 M	[20]
Patents	3.8 M	33 M	[21]
Skitter	1.7 M	22.2 M	[21]
BerkStan	685 K	13.2 M	[21]
YouTube	1.1 M	9.8 M	[20]
WikiTalk	2.4 M	9.3 M	[21]
Dblp	317 K	2.10 M	[21]

A. System Setup and Datasets

We stored our graphs on top of Apache HBase platform and used the data representation model described in Section III-B. We implemented the graph algorithms using HBase Coprocessors in order to take advantage of distributed parallelism. HBase Coprocessors can access to local and remote cache areas.

Our cluster consists of 1 master server and 5 slave servers, each of which is a c3.large instance running Linux on Amazon EC2. C3 instances are typically used for high performance computing applications such as distributed analytics, web-servers, front-end fleets etc.. Each c3.large instance comes with 2 Intel Xeon E5-2680 processors, 3.75GB memory space and two flash based SSDs with 16GB space in each. We use vanilla HBase environment running Hadoop 1.0.3 and HBase 0.94 with data nodes and region servers co-located on the slave servers. We used Ganglia distributed monitoring system to generate reports of CPU, memory, network and disk usage. We have not observed any interference from other processes on the cluster that can affect our performance results.

In our experiments we used 10 different real datasets crawled from different web sources. Some of the datasets are crawled from popular social networking cites such as Twitter, Orkut, LiveJournal and YouTube. The datasets were made available by Milove et al. [20], Social Computing Data Repository at ASU [19], and the Stanford Network Analysis Project [21]. We briefly recap the key characteristics of the datasets in Table I. More details about the datasets can be found in the references included in Table I. To emulate real world content rich graph edges, the datasets were prepared with a random text string attached to each edge. The size of the random text string varies between 100 bytes and 1KB.

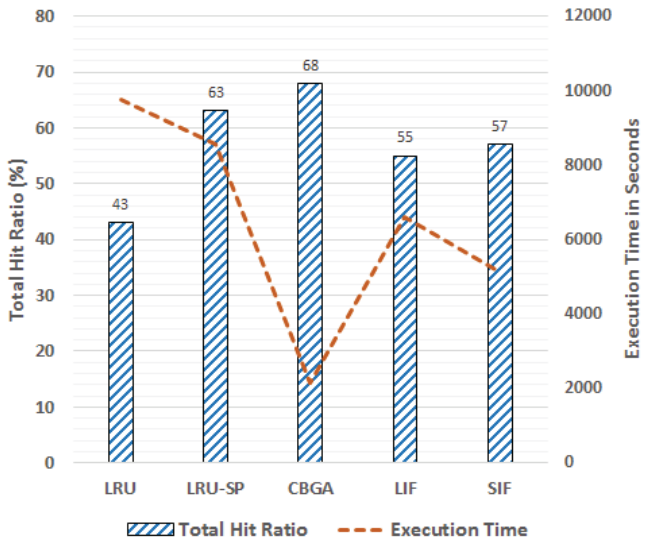


Fig. 3: Overall Hit Ratio and Workload Execution Time for Twitter dataset

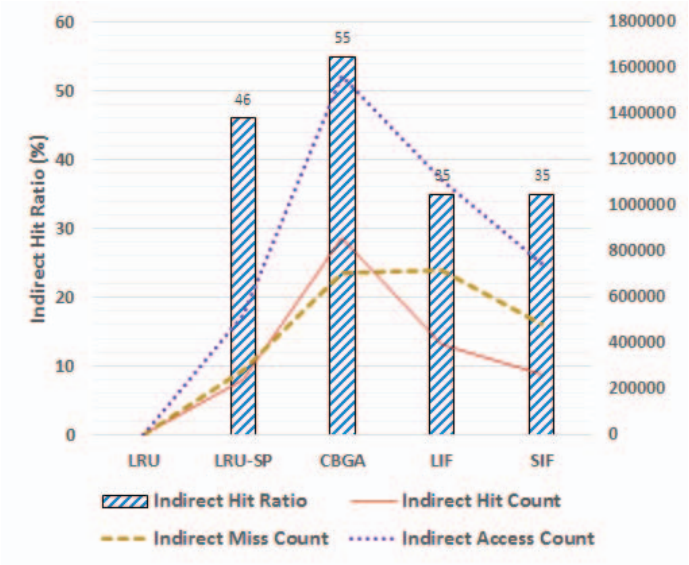


Fig. 5: Indirect Hit Ratio and other statistics for Twitter dataset

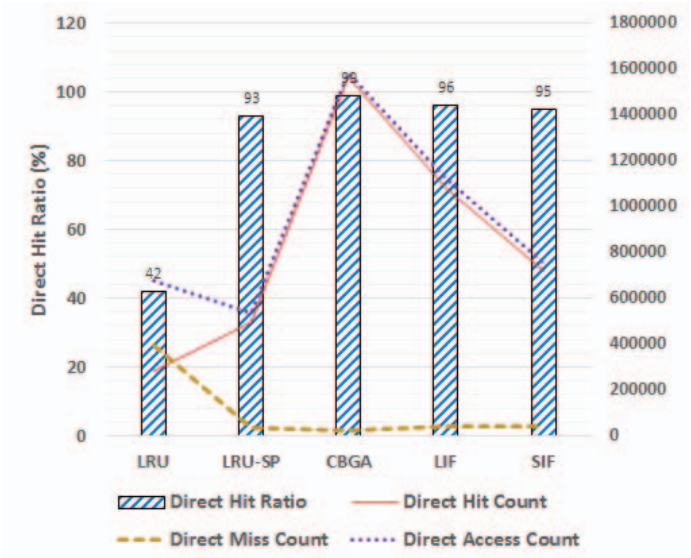


Fig. 4: Direct Hit Ratio and other statistics for Twitter dataset

Fig. 6: Remote Hit Ratio and other statistics for Twitter dataset

B. *Experiments*

We implemented the cache policies discussed in Section IV in HBase Coprocessors. Whenever a k-hop neighbor search query is received by HBase, the request is forwarded to the region server where the originating vertex is located. First hop away neighbors of this vertex are looked up in the cache. If any of them is not found then the vertex is requested either from the local region server or from other region servers running on other physical servers. The traversal of the subsequent searches continue likewise. For CBGA algorithm each cache miss is handled according to Algorithm 3. When the cache is full, the eviction policy is executed. The prefetching algorithm described in Algorithm 1 can be applied to the other caching policies we mentioned in Section IV. For instance, when a vertex is accessed from the cache, its one hop and two hop away neighbors can be brought and inserted into the cache even if an LRU, LIF or SIF algorithm is used as for the caching policy. In our experiments we observed that this prefetching techniques yield about 2X improvement in hit ratio. For the sake of fairness we implemented LRU, LIF or SIF with the prefetching technique and compared them with CBGA algorithm. For LRU algorithm we also implemented a version without this improvement to distinguish the benefit of

prefetching. In our experiments we show the results for both of these LRU implementations.

In order to prepare the query workload we examined the social network benchmarking tool LinkBench developed by Facebook [22]. The benchmark basically generates random k-hop queries and creates a query workload. Similar to this benchmark we generated 10000 random k-hop neighbor queries on each social network dataset where the originating vertex for the queries are selected randomly from the graph. We set an upper limit of 10000 vertices for the query result in order to prevent a single query to stall a region server. This could be the case when a very popular vertex in the graph (also called as supernova) is queried. In each Coprocessor we allocated a 10MB of memory heap space as the cache area.

In the first experiment we ran the query workload on Twitter dataset with different caching policy implementations and measured both hit ratio and execution time. The experiment results are shown in Figure 3. The left vertical axis shows the hit ratio and the right vertical axis shows the execution time. The difference between LRU and LRU-SP is that the former one does not have the prefetching feature whereas the latter one does have prefetching. The experiment results prove that CBGA policy achieves the highest hit ratio among all other caching policies. Another interesting observation is that the prefetching technique described in Algorithm 1 yields significant improvement in terms of hit ratio not just for CBGA but also for all other caching policies. Nevertheless the execution time of the workload is significantly faster than all other caching policies because of the distinction of the latency difference between local and remote accesses. For instance in LRU-SP, LIF and SIF the hit ratio is almost as high as CBGA but the execution time is much slower. This is an indication that these algorithms does not distinguish the graph nodes brought to the cache from local or remote servers. Another interesting observation is that despite the hit ratio improvement in LRU-SP the execution time does not change much compared to regular LRU. This also attributes to the previous reasoning which is about distinguishing local and remote resources. As a summary of this experiment we conclude that considering the topology as well as latency provides significant speed up in terms of performance.

When a particular vertex is looked up in the cache, this request could be originated from three different request types. First, the vertex could be the starting vertex for the graph traversal which is directly requested by the user. We call this look up as “direct look up”. The second option is that when a vertex is requested from the cache, the prefetching mechanism described in Algorithm 1 kicks in and brings its one hop and two hop neighbors into the cache. While executing the prefetching, the vertices are first looked up in the cache. This type of cache look up is called “indirect look up”. When a requested item is not found in the cache, the request is forwarded into another region server. Once a region server receives a loop up request from another region server, it first

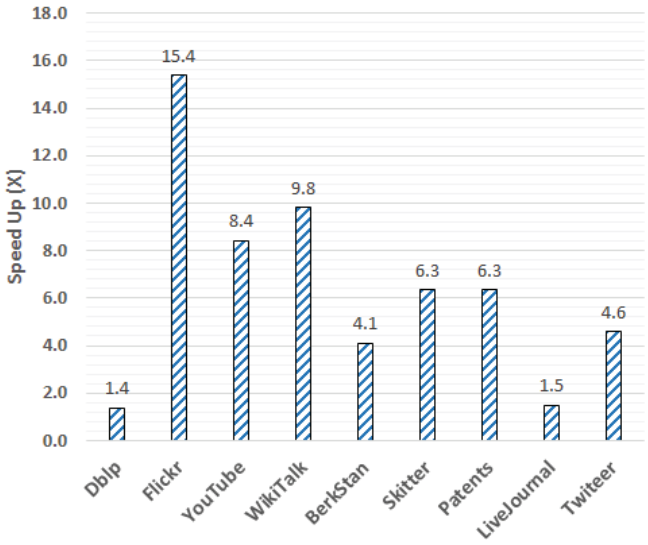


Fig. 7: Speedup achieved for each dataset when CBGA and LRU policies are compared.

checks its cache to see if it is found in the cache. This type of request is called “remote look up”. For each of these request types we had two types of counters for keeping track of accesses. For direct look up requests we monitor direct hit counts and direct miss counts. If a directly requested item is found in the cache direct hit count is incremented. Otherwise direct miss count is incremented. These statistics are provided in Figure 4. Direct access count shows the number of total direct access requests submitted to the cache. Direct hit ratio is computed by dividing the number of hits into the total number of direct access requests. Similarly indirect access requests as well as remote access requests are shown in Figure 5 and Figure 6 respectively. The hit ratio shown in Figure 3 includes aggregate hit ratio for all three types of requests. Note that the indirect access count of LRU is 0 in Figure 5. This is because the basic implementation of LRU does not have the prefetching improvement. LRU-SP shows the indirect access counts when LRU is used with the prefetching feature. One interesting observation is that even if the remote hit ratio for CBGA algorithm is less than others the overall execution time of CBGA is significantly smaller due to the distinction of latency of accesses.

We repeated the same experiment with other datasets but did not observe a notable hit ratio difference between different datasets. As for the execution time we compared the overall execution time of CBGA versus LRU and provided the speedup for all datasets in Figure 7. The speedup is computed by dividing the execution time of LRU by the execution time of CBGA. For each dataset we observed substantial speed up (about 15X in the best case). It is worth noting that in each of these experiments we observed a different speed up for different datasets. The datasets used in the experiments are

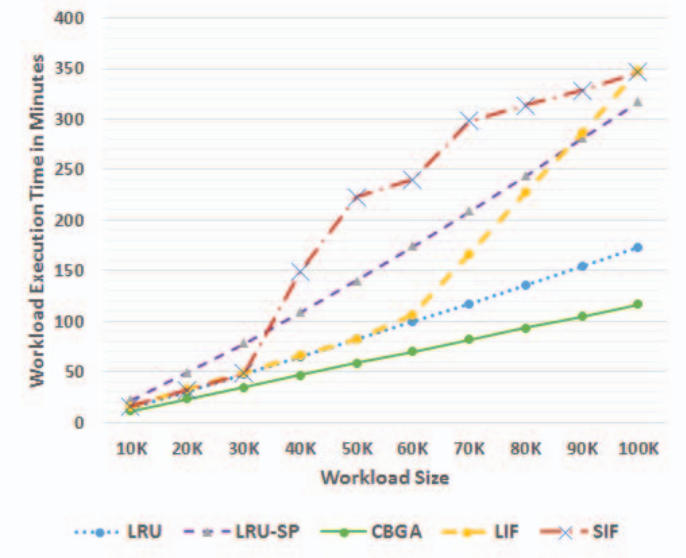


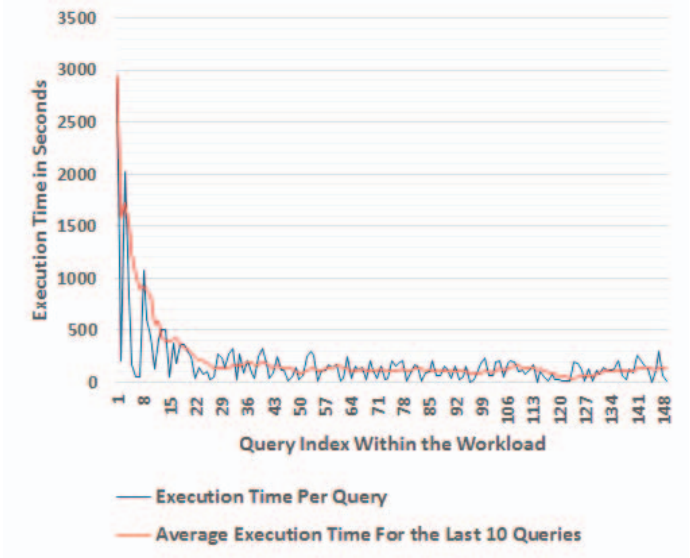
Fig. 8: Workload execution times under long runs for Flickr dataset.

real graphs from different domains. For instance Twitter is a social network, BerkStan is a web graph while Patents is a citation network among US Patents. Each of these graphs has different characteristics such as degree distribution, topology etc., which result in different performance results. Nonetheless, we observed that CBGA caching outperforms LRU based caching for each of these datasets.

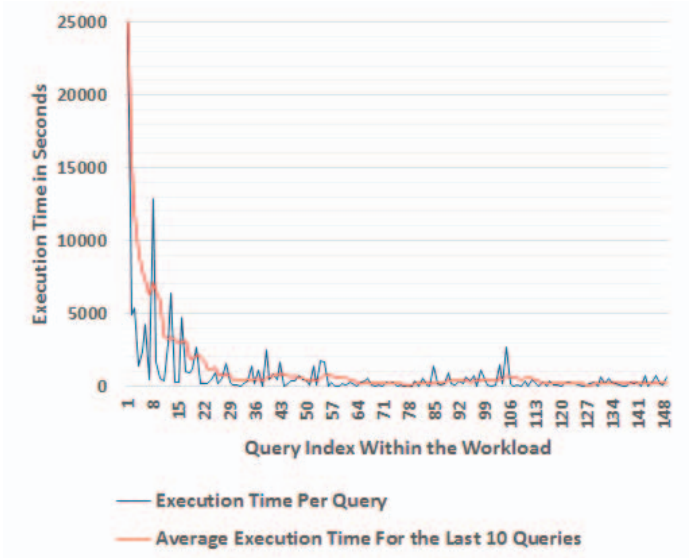
We also repeated our experiments with longer workloads to see the impact of change in the cache content. Figure 8 presents the performance of the policies under long runs for Flickr dataset. For increasing number of queries from 10K to 100K, we observe that CBGA policy provides stable lowest execution time.

In another experiment setting we ran the workload similar to the previous experiments but this time we measured the execution time of each individual query submitted to the system. Experiment results for Flickr and Twitter datasets are shown in Figure 9. The results prove that during the warm up period the execution time decreases dramatically. Once the warm up is over the execution time stabilizes. Since queries are randomly selected and their overhead is not equal (e.g., a vertex might have 10 neighbors in two hops while another vertex have 10000 neighbors in two hops) we observe some fluctuation in individual query times. Thus, we also computed the average execution time for the last 10 queries. Experiments for other datasets are not shown here since we observed similar warm up pattern.

We also computed the average number of queries executed per minute which is shown in Figure 10. After the completion of the warm up period, a significant improvement in throughput is observed.



(a) Flickr

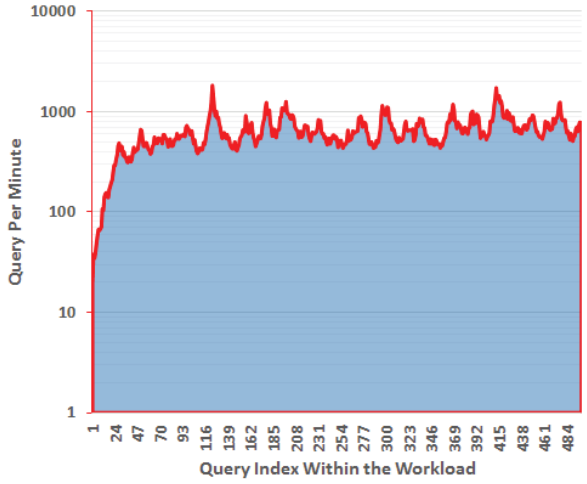


(b) Twitter

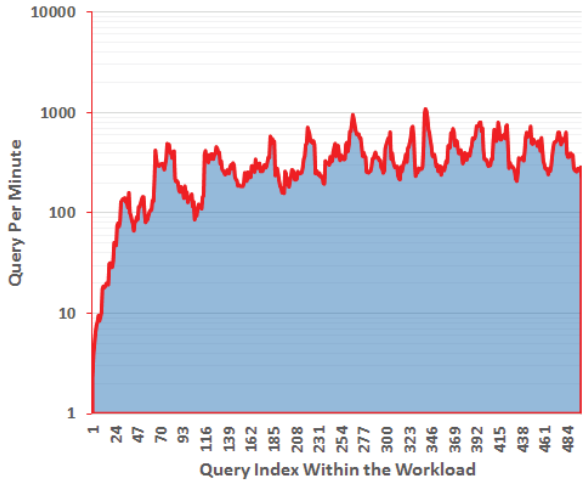
Fig. 9: Average query time is decreased while cache warms up for (a) Flickr and (b) Twitter datasets. The average execution time is calculated by using the last 10 queries instead of individual queries.

VI. Conclusion

To the best of our knowledge, this study is the first to propose a graph aware caching scheme for efficient graph processing in horizontally scaling solutions on big data platforms. We proposed a clock based graph aware cache (CBGA) system with cache and eviction algorithms designed with distributed graph processing context in mind. We ran experiments on our HBASE cluster, which demonstrate up to 15x speedup compared to traditional LRU based cache systems.



(a) Flickr



(b) Twitter

Fig. 10: The number of queries processed per minute increases while the cache warms up for Flickr and Twitter datasets.

We provided a distributed implementation of the caching algorithms on top of Apache HBase, leveraging its horizontal scaling, range-based data partitioning, and the newly introduced Coprocessor framework. Our implementation fully took advantage of distributed, parallel processing of the HBase Coprocessors. Building the graph data store and processing on HBase also benefits from the robustness of the platform and its future improvements.

Acknowledgments

We thank TUBITAK (The Scientific and Technological Research Council of Turkey) for partially supporting this work with project 113E274.

References

[1] `hbase.apache.org`.
[2] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," in *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6. ACM, 2007, pp. 205–220.
[3] "Project Voldemort: A distributed database," Online, Mar. 2012. [Online]. Available: http://project-voldemort.com/
[4] "Redis," Online. [Online]. Available: http://redis.io/
[5] "Apache Cassandra," Online. [Online]. Available: http://cassandra.apache.org/
[6] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab *et al.*, "Scaling memcache at facebook." in *nsdi*, 2013, pp. 385–398.
[7] C. Xu, X. Huang, N. Wu, P. Xu, and G. Yang, "Using memcached to promote read throughput in massive small-file storage system," in *Grid and Cooperative Computing (GCC), 2010 9th International Conference on*, Nov 2010, pp. 24–29.
[8] Neo4j, *Neo4j - The World's Leading Graph Database*, Std., 2012. [Online]. Available: http://neo4j.org/
[9] "Caches in Neo4j," Online. [Online]. Available: http://docs.neo4j.org/chunked/milestone/configuration-caches.html
[10] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. C. Li *et al.*, "Tao: Facebook's distributed data store for the social graph." in *USENIX Annual Technical Conference*, 2013, pp. 49–60.
[11] 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 *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 135–146.
[12] J. Mondal and A. Deshpande, "Managing large dynamic graphs efficiently," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '12. New York, NY, USA: ACM, 2012, pp. 145–156. [Online]. Available: http://doi.acm.org/10.1145/2213836.2213854
[13] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, pp. 4:1–4:26, Jun. 2008. [Online]. Available: http://doi.acm.org/10.1145/1365815.1365816
[14] M. Canim and Y.-C. Chang, "System g data store: Big, rich graph data analytics in the cloud," in *IC2E*, 2013, pp. 328–337.
[15] H. Aksu, M. Canim, Y.-C. Chang, I. Korpeoglu, and Ö. Ulusoy, "Multi-resolution social network community identification and maintenance on big data platform," in *BigData Congress*, 2013, pp. 102–109.
[16] H. Aksu, M. Canim, Y.-C. Chang, I. Korpeoglu, and O. Ulusoy, "Distributed k-core view materialization and maintenance for large dynamic graphs," *Knowledge and Data Engineering, IEEE Transactions on*, vol. PP, no. 99, pp. 1–1, Jan. 2014.
[17] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser, "Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System," in *Proceedings of the 15th Symposium on Operating Systems Principles (15th SOSP'95), Operating Systems Review*. Copper Mountain, CO: ACM SIGOPS, Dec. 1995, pp. 172–183.
[18] W. Vogels, "Eventually Consistent," *Commun. ACM*, vol. 52, no. 1, pp. 40–44, Jan. 2009. [Online]. Available: http://dx.doi.org/10.1145/1435417.1435432
[19] R. Zafarani and H. Liu, "Social computing data repository at ASU," 2009. [Online]. Available: http://socialcomputing.asu.edu
[20] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee, "Measurement and Analysis of Online Social Networks," in *Proceedings of the 5th ACM/Usenix Internet Measurement Conference (IMC'07)*, San Diego, CA, October 2007.
[21] `snap.stanford.edu/`.
[22] T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan, "Linkbench: A database benchmark based on the facebook social graph," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13. New York, NY, USA: ACM, 2013, pp. 1185–1196. [Online]. Available: http://doi.acm.org/10.1145/2463676.2465296