Disk-Based Management of Interaction Graphs

Buğra Gedik and Rajesh Bordawekar

Abstract—In our increasingly connected and instrumented world, live data recording the interactions between people, systems, and the environment is available in various domains, such as telecommunciations and social media. This data often takes the form of a temporally evolving graph, where entities are the vertices and the interactions between them are the edges. An important feature of this graph is that the number of edges it has grows continuously, as new interactions take place. We call such graphs *interaction graphs*. In this paper we study the problem of storing interaction graphs such that temporal queries on them can be answered efficiently. Since interaction graphs are append-only and edges are added continuously, traditional graph layout and storage algorithms that are batch based cannot be applied directly. We present the design and implementation of a system that caches recent interactions in memory, while quickly placing the expired interactions to disk blocks such that those edges that are likely to be accessed together are placed together. We develop live *block formation* algorithms that are fast, yet can take advantage of temporal and spatial locality among the edges to optimize the storage layout with the goal of improving query performance. We evaluate the system on synthetic as well as real-world interaction graphs, and show that our block formation algorithms are effective for answering temporal neighborhood queries on the graph. Such queries form a foundation for building more complex online and offline temporal analytics on interaction graphs.

Index Terms—Interaction graphs, storage and querying, disk layout

1 Introduction

Graphs are popular data structures used to represent relationships between people, systems, and the environment, where the vertices represent entities and the edges represent the interactions among them. In many application domains capturing, storing, and analyzing this graph structure is a key enabling capability. For instance, the graph structure may represent the relationships in a social network, where finding communities in the graph [8] can facilitate targeted advertising. In the telecommunications (telco) domain, call details reports (CDRs) can be used to capture the call relationships between people [26], and locating closely connected groups of people can be used for generating promotions.

An important characteristic of many real-world graphs is their temporal nature. For instance, the mention graph of Twitter [35] gets a new edge every time a user mentions another user in a tweet. As another example, the call graph generated from CDRs in a telco application also evolves as people make calls to each other. These kinds of graphs grow forever, as users continuously interact with each other. In other words, these are append-only graphs, where new edges, and less often new vertices, are added as time progresses, but no deletions are made. We call such graphs *interaction graphs*.

There are two major kinds of temporal analysis that can be performed on interaction graphs. The first is *online analysis* that requires access to the recent interactions, which can often be captured by defining a sliding window [9] over the stream of graph insertions. The second is *offline analysis*, which requires analyzing the relationships captured by the graph over a desired *time range*. Unlike online analysis, offline analysis requires storing the ever growing graph on persistent storage. In this paper, we study graph storage and querying techniques that facilitate efficient temporal analysis over historical interaction graphs, as well as in-memory caching to enable online analysis over their most recent views.

Many graph algorithms rely on the fundamental operation of *graph traversal* and exhibit high access locality [33]. Given that a vertex is visited during a traversal, it is quite likely that the neighbors of this vertex will be visited shortly after. For instance, an n-hop breadth first search around a vertex exhibits high locality. This observation has motivated block-based disk layouts where the neighborlists of vertices that are highly connected (e.g., form a community) are placed into the same disk block [13]. This minimizes the number of blocks read, which reduces I/O. It also avoids the costly disk seeks, since chasing blocks often requires seeking to different areas of the disk. However, the focus of such work has been on static or slowly changing graphs. Temporal nature of interaction graphs bring several new challenges.

First, the ever growing nature of the graph requires continuously creating new data blocks. To capture the potentially changing locality across vertices, we need to continuously revise the grouping of vertices and create the most suitable layout for different time frames. Second, the evolving nature of the graph requires block formation decisions to be made in an online manner. Offline graph partitioning techniques, such as METIS [16], are not suitable for handling online updates. As such, we need heuristic techniques to make accurate placement decisions (placing edges and vertices to blocks) in short time. Third, the system should facilitate time range based access, in order to enable

- *B. Gedik is with the Computer Engineering Department, Bilkent University, Bilkent, Ankara 06800, Turkey. E-mail: bgedik@cs.bilkent.edu.tr.*
- *R. Bordawekar is with the IBM T.J. Research Center, IBM Research, Yorktown Heights, NY 10598. E-mail: bordaw@us.ibm.com.*

Manuscript received 27 June 2013; revised 12 Nov. 2013; accepted 14 Dec. 2013. Date of publication 8 Jan. 2014; date of current version 26 Sept. 2014. Recommended for acceptance by A. Singh.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TKDE.2013.2297930

1041-4347 © 2013 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

historic analysis performed at different time scales. Last but not the least, the system should cache the recent window of updates in memory to support online analysis.

To address these challenges we develop a system for storing and querying interaction graphs. The system stores the recent edges in-memory and continuously offloads expired edges to the disk, removing them from the in-memory store. Expired edges are temporally buffered and written to the disk in blocks. The creation of these blocks uses an online algorithm that quickly and effectively determines which edges should the placed together, taking into account temporal as well as spatial locality. The key idea is to organize a block as one or more *temporal neighborlists* and use a *locality* metric to heuristically place edges to disk blocks. Additionally, we keep two index structures, in order to effectively run temporal neighborhood queries on the stored interaction graph. Using these indices and a buffer manager placed over the edge data stored on disk, the system makes it possible to reuse cached disk blocks, especially given that the block formation is done with temporal and spatial locality in mind.

This paper makes the following contributions:

- We develop a system for disk-based storage and temporal querying of interaction graphs, supporting both online (over a recent window) and offline (over a historical view) graph analytics.
- We develop fast block formation algorithms that maximize spatial and temporal locality for the stored edges, improving the query I/O performance.
- We provide an evaluation of the system using synthetic and real-world data, as well as a study of the impact of various system parameters on its performance.

The rest of the paper is organized as follows. Section 2 covers related work. Section 3 gives an overview of the data model. Section 4 describes the system architecture. The block formation algorithms are described in Section 5. Their efficient implementations are described in Section 6. Section 7 presets experimental evaluation and Section 8 concludes the paper.

2 Related Work

With the availability of large amounts of relationship and interaction data, the topic of graph data management and mining has enjoyed significant research interest. An overview is given by Aggarwal and Wang [1].

In many domains, such as social media and telecommunications, the graphs used are dynamic and evolve over time. Many mining algorithms rely on this temporal nature to gain new insights, such as understanding how the structure and properties of the network change [19], [22], how communities found in the graphs evolve over time [3], [34], or how influence spreads through the social network [17].

From the perspective of data management systems, we can divide the related work into to broad categories, i) in-memory graph processing systems, and ii) graph storage and querying systems. As an example of the former, Trinity [31] is a distributed graph query engine that relies on a memory cloud for storing the graph data and supports online and offline queries over this data. Another example is the in-memory, distributed graph data management system from Mondal and Deshpande [25], which is optimized to manage large-scale dynamically changing graphs and supporting low-latency query processing over them. Different than these systems, our work focuses on storing graphs on disk. Due to continuous growth and historical storage requirements of interaction graphs, keeping the entire graph in memory could be costly. Furthermore, since not all of the interactions are needed at any given time, storing all of them in memory could also be wasteful.

In terms of disk based systems, GBASE [15] is relevant. It is a large-scale graph management system based on Map/Reduce [7]. The system employs a graph storage method that relies on block compression to efficiently store homogeneous regions of graphs, and on a grid based technique to efficiently place blocks into files. However, the system is not designed for temporal graphs, and thus the notion of time ranges do not exist.

Another relevant research effort is the work of Khurana and Deshpande [18], which proposed a distributed system for storing the history of dynamic graphs and answering temporal *snapshot* queries over it. The system relies on a distributed hierarchical index structure that uses delta computations to compactly record historical information, and support efficient retrieval of historical graph snapshots. However, the data and query model is very different than the ones being considered in this work. Given a time point the system can return a snapshot of the graph for that time point; and similarly, given multiple time points or a time range, the system can return a series of snapshots representing the queried time points. This kind of interface is suitable for analyzing dynamic relationship graphs, where each edge has a valid time interval, and edges can be added or removed (e.g., the following and followee relationship in Twitter). This is similar to the *valid time* model from temporal databases literature [28]. In contrast, our work is focused on interaction graphs (e.g., the mention graph in Twitter, formed by mention tags found in Tweets connecting users to each other), where each edge has a time stamp and not a time interval associated with it. This is similar to the *transaction time* model from temporal databases. The time range queries supported by our system do not return graph snapshots, instead they return edges involved in interactions within the given time range for the vertices of interest (more like a neighborhood query).

A key contribution of our work is the online block formation algorithms used to group together temporally and spatially close edges. A relevant work in this area is the disk layout techniques proposed by Hoque and Gupta [13]. The main idea is to use the community structure to optimize the disk layout, so that graph traversals can be performed using less I/O. There are two major differences from our work. First, there is no temporal dimension, so time range based querying is not a concern in their work. Second, their system is batch based and does not support dynamic graphs.

There are also various systems for performing large-scale graph analysis, with different programming models. These include synchronous vertex programming pioneered by Pregel [24], such as Apache Giraph [10]; asychronous vertex programming pioneered by GraphLab [11], [20] and

generalized iterated matrix-vector multiplication pioneered by PEGASUS [14]. These works focus on analytical processing and do not provide data management capabilities. Also the focus is on large-scale graphs without a temporal dimension.

Another line of related work is streaming graph processing [2], [5], [37]. Rather than storing the entire history of the graph, these works maintain an in-memory sketch to answer queries approximately, with certain bounds.

3 Model

The interaction graph $G(V, E)$ consists of a set of vertices (V) and edges (E). The vertex set is mostly static, whereas the edge set grows continuously. While new vertices can be added, the rate of vertex (entitiy) addition is negligible compared to the rate of edge (interactions) addition. Vertices can be deleted, which means that no future edges incident upon the deleted vertices are to be admitted to the graph. Vertices can have vertex data associated with them as well, denoted by $d(v)$ for $v \in V$.

The edges of an interaction graph are *temporal*. An edge is denoted as $e = (u, v, t) \in E$, where $u \neq v$, $u, v \in V$ and t is a time stamp representing the occurrence time of the interaction. The graph is bidirectional, which means that both outgoing and incoming edges are accessible from a vertex. The edges cannot be deleted, as an interaction that has happened in the past cannot be undone. The edges can have data associated with them as well, denoted by $d(e)$ for $e \in E$.

The system supports two kinds of fundamental query operations on the graph, on top of which more advanced graph analytics can be built. These are:

- *Time range vertex query*: Given a time interval $[t_s, t_e)$, the goal is to find the list of vertices that have incident edges within time the interval.
- • *Time range neighborhood query*: Given a time interval $[t_s, t_e)$, and a vertex v, the goal is to find all incident edges of the vertex within the time interval.

Neighborhood queries also have a *selective* version, which supports specifying a filter on the edge data. Edges that are not satisfying this filter are not returned in the query results.

These are fundamental operations, because the first one enables us to find the vertices that are active during a given time interval, whereas the second one enables us to perform traversals within a given time interval, starting from a known vertex. For instance, in a CDR graph, using a time range vertex query we can find all numbers that were involved in a call during a given time interval. For a given number in this set, we can issue a series of time range neighborhood queries to trace the call sequences that originally initiated from this number, but disseminated to others.

4 System Architecture

We describe the basic architecture of our system.

4.1 Live and Historical Graphs

The system is divided into two main components, namely the *live graph* component that is stored in memory, and the *historical graph* component that is stored on the disk. The live graph component contains the recent window of edges added to the graph, whereas the historical graph contains all the edges that are not in the recent window anymore. The historical graph grows continuously, as expired edges flow into it from the live graph. A FIFO buffer stores all the edges in the window in time stamp order and facilitates expiring edges from the live graph. The size of the live graph is bounded by the number of edges kept in the FIFO. Fig. 1 shows this FIFO and the live graph on the left of the figure.

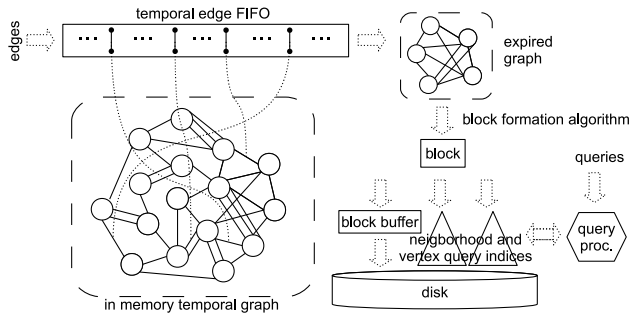


Fig. 1. Architecture of the temporal storage and querying system for live interaction graphs.

4.2 Expired Graph and Blocks

To facilitate the flow between the live graph and the historical graph, the expiring edges are buffered in memory and written to the disk in *blocks*. This buffer in-between is named the *expired graph*, shown on the top right of Fig. 1. When the size of the expired graph exceeds a threshold, a block is created and written to the disk, removing the edges contained within the block from the expired graph.

The live graph, the expired graph, as well as the blocks extracted from the expired graph are organized as a set of *temporal neighborlists*. A temporal neighborlist contains a *head vertex* and its neighbors restricted to a time interval. Formally, $l = \langle h = v \in V, e = \{(u, t) \,|\, (v, u, t) \in E\}\rangle$ represents a temporal neighborlist. Here, the head vertex is $l.h = v$, and there exists several edges from v to other vertices, kept in the list $l.e$. We keep the edges in $l.e$ in time stamp order, from oldest to the newest. A given edge (u, v, t) appears in neighborlists of both vertices it is incident upon, i.e., $(v, t) \in l_0$ s.t. $l_0.h = u$ and $(u, t) \in l_1$ s.t. $l_1.h = v$. We refer to the entries stored in the neighborlists as *half edges*. In summary, for each edge, there will always be two half edges.

Another important property of temporal neighborlists is that, they are *contiguous in time*. Let us denote the smallest time stamp in $l.e$ as $l.t_s$ and the largest one as $l.t_e$. We denote the neighborlist's *temporal interval* as $l.\delta_t = [l.t_s, l.t_e]$. We require that all edges (v, u, t) that have a time stamp in the neighborlist's temporal interval, that is $t \in l.\delta_t$, are in l. That is, l is a temporal slice over the entire time sorted neighborlist of the head vertex $l.h$.

4.3 Block Formation

A block, denoted by $B \in \mathcal{B}$, contains one or more temporal neighborlists identified by their head vertices, that is $B = \{v \mapsto l \,|\, l.h = v\}$. Importantly, the goal of block

formation is to capture *locality* as much as possible. There are two kinds of locality to be captured. The first one is *temporal locality*: neighborlists that are from the same time interval are grouped together into the same blocks. The second one is *spatial locality*, that is neighborlists that are close to each other with respect to having common graph edges are co-located in the same blocks. Typically, the size of the block is limited, that is $s(B) \leq b$, where b is the *block size*.

Improved locality for blocks has two main advantages. First, performing traversals via time range neighborhood queries require accessing edges within a time interval (temporal locality) and following the connections between the vertices requires visiting vertices that are close to each other in the graph space (spatial locality). Thus, successful block formation will reduce disk I/O. Second, since we use a bidirectional graph, co-locating the two half edges corresponding to an edge e in the same block also avoids storing the same edge data $d(e)$ twice. This also reduces the size of the graph on disk.

4.4 Indexing over Historical Graphs

The blocks of the historical graph are stored in a backend data store. The data store has a primary index that can locate the file offset of a block on the disk using the block id (denoted by $i(B)$ for a block B). We use a *block buffer* to store the frequently accessed blocks in memory. The buffer relies on a LRU replacement policy.

We keep two secondary indexes to support time range vertex and time range neighborhood queries. The first one is the *neighborhood query index*. It maps, for each head vertex in each block, the pair that contains the head vertex and the end time stamp of its neighborlist list to the pair that contains the block id for the head vertex and the start time stamp of its edge list. For instance, if there is a temporal neighborlist l in block B, with an edge list time range of $[l.t_s, l.t_e]$, then the index will contain a mapping from $(l.h, l.t_e)$ to $(i(B), l.t_s)$. Formally, the neighborhood query index is a mapping:

$$I^n = \{(v, t_e) \mapsto (i(B), t_s) \mid B[v].\delta_t = [t_s, t_e) \wedge \ v \in B \in \mathcal{B}\}.$$

We implement I^n using an LSM-tree [27] for achieving good update performance, but a traditional B-tree based index suffices as well. To answer a time range neighborhood query $(v, [t_s, t_e))$, we do a search on the neighborhood query index to find the entry with the vertex value of v and the smallest end time stamp that is $\geq t_s$. From there, we scan all entries in order using a cursor, until we reach an entry whose start time stamp is $\geq t_e$. For each entry scanned (except the last), we load the block and include the edges for the head vertex into the query result, as long as their time stamp is in the query time range $[t_s, t_e)$.

The second index we maintain is the *vertex query index*. It maps, for each head vertex in each block, the temporal extent of its neighborlist to a pair containing the block id and the head vertex itself. For instance, if there is a temporal neighborlist l in block B, with an edge list time range of $[l.t_s, l.t_e]$, then the index will contain a mapping from $(l.t_s, l.t_e)$ to $(i(B), l.h)$. Formally, the vertex query index is a mapping:

$$I^m = \{(t_s, t_e) \mapsto (i(B), v) \mid B[v].\delta_t = [t_s, t_e] \wedge v \in B \in \mathcal{B}\}.$$

We implement I^m using an index capable of answering time range queries over the intervals being indexed. In particular, we use an R+-tree with a single effective dimension. An alternative approach is to use a segment tree [6]. To answer a time range vertex query for the time range $[t_s, t_e)$, we do a search on the vertex query index to find all entries whose time ranges intersect with that of the query time range. For each entry returned, we add the head vertex to the query result, with one exception: When the query time range is strictly contained within the time range of the entry, it is possible that the neighborlist represented by the entry may not contain any edges within the given time range. For entries like that, we use the block id to retrieve the block from the disk and explicitly check if there is an edge in the neighborlist that falls within the query interval. If not, the head vertex is not added to the query result.

It is important to note that the neighborhood query index is used to perform a search on non-overlapping intervals, whereas the vertex query index is used to perform a search on possibly overlapping intervals. That is why a B-tree-like index that can answer range queries over single dimensional values is sufficient to implement the former, whereas the latter requires a spatial index. Also, our indexes cover the entire history of the graph. If query time ranges are known to have an upper limit, separate indexes could be built for different time ranges, which will reduce the insertion cost. We do not investigate this direction in our work.

The primary block index, the neighborhood query index, and the vertex query index, together form the historical graph. We also keep the vertex data. However, since vertex data is mostly static, it is kept in a table indexed by the vertex id, with an in memory LRU block cache placed on top of it in order to accelerate the accesses to vertex data.

5 Block Formation

In this section, we describe our block formation algorithm. Recall that the goal of this algorithm is to form blocks that exhibit high temporal and spatial locality.

5.1 Locality of Blocks

We define the locality of the block making use of two concepts: *conductance* and *cohesiveness*.

Conductance is a metric commonly used for graph partitioning [21]. In our context, it is defined as the ratio of the number of *dangling half edges* to the total number of half edges in the block. Dangling half edges are destined to vertices whose temporal neighborlists are either not in the block or do not contain the edge in question. Denoted by C^d, conductance is formally defined as follows:

$$C^d(B) = \frac{|\{(v, u, t) \mid I(v, u, t, B) \wedge \neg I(u, v, t, B)\}|}{\sum_{v \in B} |B[v].e|}, \quad (1)$$

$$I(v, u, t, B) \equiv v \mapsto l \in B \wedge (u, t) \in l.e.$$

Here, $I(v, u, t, B)$ is a Boolean predicate that evaluates to true if v is a head vertex in block B and contains the half edge

(u,t) in its neighborlist. $I(v,u,t,B) \wedge \neg I(u,v,t,B)$ is the set of all dangling half edges. If all edges are internal to the block B, then the conductance yields a value of 0. If all edges are external to the block, then it yields a value of 1.

One might be tempted to use $1 - C^d(B)$ as a locality metric. However, this does not work for interaction graphs, due to their temporal nature. To see this consider the following block that contains two edges and four half edges: $B_0 = \{v_0 \rightarrow \{(v_1,t_0)\}, v_1 \rightarrow \{(v_0,t_0)\}, v_3 \rightarrow \{(v_4,t_1)\}, v_4 \rightarrow \{(v_3,t_1)\}\}$. The two edges (v_0,v_1,t_0) and (v_3,v_4,t_1) are not connected to each other in any way. Yet, this block has a conductance value of $C^d(B_0) = 0$ and thus maximum locality of 1.

What is missing is the cohesiveness of the block. This is a metric commonly used for finding highly connected regions, or communities in graphs. In our context, we define cohesiveness as the number of head vertex pairs that are connected to each other via edges in the block, divided by total number of head vertex pairs. Denoted by C^h, cohesiveness is formally defined as follows:

$$C^h(B) = \frac{|\{(v,u) \mid \exists t \text{ s.t. } I(v,u,t,B) \wedge I(u,v,t,B)\}|}{|B| \cdot (|B|-1)}. \quad (2)$$

In the running example, we have $C^h(d) = \frac{4}{4*3} = 1/3$. It should be clear that for interaction graphs, cohesiveness is also not enough by itself as there could be many dangling edges whose existence does not factor into the cohesiveness metric. The impact of such dangling edges are captured by the conductance metric. Cohesiveness and conductance are complementary to each other.

As a result, we define the locality of a block, denoted by L, as the geometric mean of cohesiveness and one minus the conductance. That is:

$$L(B) = \sqrt{C^h(B) \cdot (1 - C^d(B))}. \quad (3)$$

5.2 The Block Formation Algorithm

To facilitate block formation, the expired graph is maintained as a hash table, denoted by $\mathcal{H}$, where the keys are the head vertices and the values are the temporal neighborlists. We use $\mathcal{H}[v]$ to denote the temporal neighborlist of vertex v in the expired graph. When the size $s(\mathcal{H})$ reaches a threshold of $m \cdot W$, we remove a block's worth of edges from H and write the resulting block B to the disk. Here, W is the size of the temporal edge FIFO and $m \in [0,1]$ is the relative size of the expired graph.

For this purpose, we use a greedy algorithm. The algorithm starts with an initial set of *partial* candidate blocks. These initial candidates contain only a single temporal neighborlist, with a single edge in it. At each greedy step, the algorithm grows these partial candidate blocks by considering several alternative *expansions*, and picking the ones that provide the best improvement. For this purpose, a *utility* metric based on locality is used. Once a partial candidate block reaches the maximum block size, it is inserted into a *final* candidate block list and the procedure continues until no partial candidates remain. Finally, the block among the final candidates with the best utility metric is chosen.

Algorithm 1 shows the procedure used to construct the next block. Once constructed, the block is removed from the expired graph and is stored on the disk. Additionally, the indexes I^n and I^m are updated with new entries.

Algorithm 1: GETNEXTBLOCK$(\mathcal{H}, C, b)$

Param : $\mathcal{H} = \{v \mapsto l\}$, vertex to temporal neiglist mapping
Param : C, initial candidate selection policy
Param : k, number of initial candidates desired
Param : b, block size of the algorithm
$V \leftarrow$ INITCANDIDATES$(\mathcal{H}, C, k)$ ▷ Potential head vertices
$\mathcal{S} \leftarrow \{\{v \mapsto_{\mathcal{H}} [v].t_s\} \mid v \in V\}$ ▷ Candidate partial blocks
$\mathcal{F} \leftarrow \{\}$ ▷ Candidate finalized blocks
while $|\mathcal{S}| > 0$ **do** ▷ There are partial candidates
  $\mathcal{S}' \leftarrow \{\}$ ▷ Init. expanded partial candidates
  for $P \in \mathcal{S}$ **do** ▷ For existing partial candidates
    $P' \leftarrow$ EXTENDBLOCK$(\mathcal{H}, P, b)$ ▷ Expand the candidate
    if $s(\mathcal{P}') = b$ **then** ▷ The expanded block is full
      $\mathcal{F} \leftarrow \mathcal{F} \cup P'$ ▷ Add it to the finalized blocks
    else ▷ The expanded block is still partial
      $\mathcal{S}' \leftarrow \mathcal{S}' \cup P'$ ▷ Add it to the new candidate set
  $\mathcal{S} \leftarrow \mathcal{S}'$ ▷ Replace candidate partial block set
$P \leftarrow \text{argmin}_{P \in \mathcal{F}} L(P, \mathcal{H})$ ▷ Find the best candidate
▷ Form a block from the best candidate
$B \leftarrow \cup_{v \mapsto t \in P} \{v \mapsto \langle h{=}v, e{=}\{(u,t') \mid (u,t') \in \mathcal{H}[v].e \wedge t' \leq t\}\rangle\}$
$\forall_{v \mapsto t \in P}\, \mathcal{H}[v].e \leftarrow \mathcal{H}[v].e \setminus B[v].e$ ▷ Remove block from $\mathcal{H}$
return B ▷ Return the block

In Algorithm 1, as a first step the INITCANDIDATES() method is called to come up with an initial set of vertices that will be used to form the partial block candidates. The parameter k is used to adjust the number of initial candidates selected, and the parameter C is used to specify the policy. The Algorithm 2 shows the five different policies used. *Random* picks k random head vertices from $\mathcal{H}$. *Old* uses the k head vertices that have the oldest first edges in their temporal neighborlists. *New* picks the ones with the newest first edges. The *Max* policy picks the head vertices with the longest temporal neighborlists and *Min* the ones with the shortest.

Algorithm 2: INITCANDIDATES$(\mathcal{H}, C, k)$

Param : $\mathcal{H} = \{v \mapsto l\}$, vertex to temporal neiglist mapping
Param : C, initial candidate selection policy
Param : k, number of initial candidates desired
if $C = \mathit{Random}$ **then return** argmax-$k_{\{l.h \mid (l.h \mapsto l) \in \mathcal{H}\}}$ RAND()
else if $C = \mathit{Old}$ **then return** argmin-$k_{\{l.h \mid (l.h \mapsto l) \in \mathcal{H}\}}\, l.t_s$
else if $C = \mathit{New}$ **then return** argmax-$k_{\{l.h \mid (l.h \mapsto l) \in \mathcal{H}\}}\, l.t_s$
else if $C = \mathit{Max}$ **then return** argmax-$k_{\{l.h \mid (l.h \mapsto l) \in \mathcal{H}\}}\, |l.e|$
else if $C = \mathit{Min}$ **then return** argmin-$k_{\{l.h \mid (l.h \mapsto l) \in \mathcal{H}\}}\, |l.e|$

Once the initial set of head vertices are determined, the set of candidate partial blocks ($\mathcal{S}$ in the algorithm) are formed by taking the first edge from the temporal neighborlists of each head vertex. This results in k candidate partial blocks. Rather than keeping the individual edges, each partial block is organized as a mapping $\{v \mapsto t\}$ from head vertices to time stamps. We assume that a mapping $v \mapsto t$ implies that all edges in $\mathcal{H}[v].e$ with time stamp $\leq t$ are part of the candidate partial block. The set $\mathcal{S}$ of candidate partial blocks are then expanded until they all reach the maximum block size, b, at which point they are added to the candidate finalized blocks list ($\mathcal{F}$ in the algorithm). The EXTENDBLOCK()

procedure given in Algorithm 3 defines how a given candidate partial block is extended. Once the expansion is complete, the best candidate block in $\mathcal{F}$ is chosen using the locality metric L from Equation (3), by simply adapting it for candidate blocks (where only the head vertex to time stamp mappings are held). Finally, Algorithm 1 constructs an explicit block from the best candidate block, and removes edges that are contained in this block from $\mathcal{H}$.

Algorithm 3: EXTENDBLOCK$(\mathcal{H}, P, b)$

Param : $\mathcal{H} = \{v \mapsto l\}$, vertex to temporal neighlist mapping
Param : P, partial block candidate to be extended
Param : b, block size of the algorithm
▷ Find possible expansions that remove dangling edge(s)
$T \leftarrow \{(v,t) \mid \exists u \text{ s.t. } (u,t) \in \mathcal{H}[v].l \wedge (v,t) \in \mathcal{H}[u].l \wedge P[u] \geq t \wedge (v \notin P \vee t > P[v])\}$
▷ Remove exapnsions that violate block size constraints
$R \leftarrow \{(v,t) \mid (v,t) \in T \wedge s(P \cup \{v \mapsto t\}) \leq b\}$
if $|R| = 0$ **then** ▷ No expansions available
  ▷ Consider one-edge expansions, for all head vertices in $\mathcal{H}$
  $T \leftarrow \{(v,t) \mid v \mapsto l \in \mathcal{H} \wedge t = \min(\{t' \mid (u,t') \in l.e \wedge t' > P[v]\})$
  ▷ Size filter again, as new head vertices in P cause more size growth than a single edge's worth
  $R \leftarrow \{(v,t) \mid (v,t) \in T \wedge s(P \cup \{v \mapsto t\}) \leq b\}$
$(v,t) \leftarrow \text{argmax}_{(v,t) \in R} \frac{L(P \cup \{v \mapsto t\}) - L(P)}{s(P \cup \{v \mapsto t\}) - s(P)}$ ▷ Best expansion
return $P \cup \{v \mapsto t\}$ ▷ Return the extended candidate

The EXTENDBLOCK() procedure aims at extending a candidate block P, while preserving the temporal and spatial localities.

First, it finds all vertex-time stamp pairs (v,t) for which there exists a vertex u such that the two half edges corresponding to the edge (v,u,t) are both contained in the expired graph ($(u,t) \in \mathcal{H}[v].l \wedge (v,t) \in \mathcal{H}[u].l$), yet only the half edge (v,t) is in the candidate block ($P[u] \geq t \wedge (v \notin P \vee t > P[v])$). In other words, the half edge (v,t) is currently dangling in the candidate block, as it is only contained within the temporal neighborlist of u, but its half edge pair (u,t) is not within the temporal neighborlist of v. This means that extending v's temporal neighborlist up to time stamp t in P is one potential expansion opportunity that will remove a dangling edge. Removing a dangling edge this way is a *necessary* condition for increasing locality of the candidate block. However, it is not *sufficient*, since extending the temporal neighborlist of v in P up to time stamp t may bring in several *new* dangling edges into the candidate block.

In the algorithm, the set of candidate expansions are represented by T. If an expansion opportunity in T results in growing the block such that it exceeds the block size ($|P \cup \{v \mapsto t\}| \leq b$), then it is not a viable expansion. As a result, we remove such expansions from T, resulting in a filtered set R.

If R is empty, then we consider all single edge expansions possible (these may include expansions that do not remove any of the dangling edges). Finally, we pick the expansion that maximizes our utility metric U. Formally:

$$U(P,(v,t)) = \frac{L(P \cup \{v \mapsto t\}) - L(P)}{s(P \cup \{v \mapsto t\}) - s(P)}. \quad (4)$$

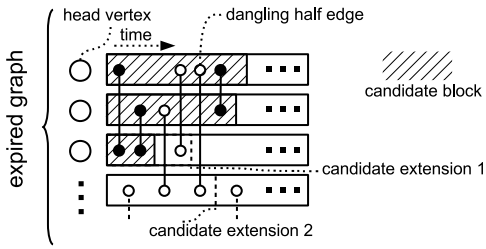


Fig. 2. Illustration of the block formation algorithm.

In other words, the utility metric computes the improvement in the locality metric per increase in the block size. For a given expansion (v,t), the number of new edges added can be computed by looking at all the edges in $H[v].e$ that have a time stamp less than or equal to t but larger than the current highest time stamp in v's neighborlist within P, that is $P[v]$. For computing the increase in size, we also consider the size cost of adding a new head vertex to the block in case $v \notin P$.

5.3 Example Illustration

Fig. 2 illustrates an excerpt from the operation of the algorithm. In the figure, we see the set of temporal neighborlists from the expired graph. A partial candidate block is marked on the expired graph using diagonal stripes. Recall that a given edge (u,v,t) appears as half edges in two different temporal neighborlists in the expired graph. Specifically, the ones for which u and v are the head vertices. As such, every edge in the figure connects two neighborlists. The half edges that are dangling are marked with empty circles in the figure.

Recall that the core of the algorithm is to grow the candidate blocks. We consider expansions that add at least one non-dangling half edge to the block. In the figure, we show two such expansions. The first one increases the temporal extent of one of the existing neighborlists in the candidate block, whereas the second one adds a new neighborlist to the candidate. In order to compute which one is a better expansion, we have to first compute the current locality of the candidate block.

Before the expansion, the candidate block has nine half edges in total. Out of these nine half edges only three of them are dangling and six are not dangling. As a result, the conductance is given by $C_0^d = 3/9 = 1/3$. Since all pairs of neighborlists are connected by a non-dangling edge, the cohesiveness is $C_0^c = 3/3 = 1$. As a result, we have $L_0 = \sqrt{C^h \cdot (1 - C^d)} = 0.817$.

The first candidate expansion brings in only a single edge, increasing the edge count to 10, and reduce the number of dangling edges to 2. This increases the locality value to $L_0 = \sqrt{1 - 2/10} = 0.894$ (no change to cohesiveness). Since only one edge is added, the cost is increased by only 1. The utility metric is then given by $U_0 = (0.894 - 0.817)/1 = 0.077$.

The second candidate expansion brings in three half edges. Among these one is a new dangling edge, whereas the other two match the previously dangling edges in the candidate block. As a result, after the expansion, there are 12 edges, two of which are dangling. The conductance is

then given by $C_1^c = 2/12 = 1/6$. Among the four head vertices, five of the six pairs are connected, and as such the cohesiveness reduces to $C_1^h = 5/6$. The locality for this expansion is given by $L_1 = \sqrt{(1-1/6)\cdot(5/6)} = 0.833$. Accordingly, the utility metric for this expansion is $U_1 = (0.833 - 0.817)/3 = 0.005$. As a result, the first expansion is selected as the better alternative.

6 Implementation Details

In this section, we describe the efficient implementation of the algorithms described in Section 5. The block formation algorithm needs to run fast, as the edges are continuously flowing from the live graph to the historical graph, where the block formation algorithm runs in-between, extracting blocks from the expired graph.

So far we have described the block formation in general terms, without paying attention to the computational costs. To make the algorithm efficient, we need to avoid operations that require scanning the entire expired graph. There are three steps in the algorithm that require attention with respect to this. These are: i) the selection of the initial candidates, ii) the selection of the head vertices for expansion, and iii) the selection of the temporal extents for each head vertex during expansion.

6.1 Observations

We make a few observations that help with efficient implementation of these steps.

Observation 1. For the *Old* candidate selection policy, the creation of the initial candidate blocks can be implemented efficiently by keeping a min heap on the time stamps of the oldest edges in the neighborlists associated with the head vertices in the expired graph. When a new edge moves from the temporal edge FIFO to the expired graph, no updates are needed to this heap unless a new neighborlist has to be created. When the block formation algorithm runs, as part of the initial candidate formation (Algorithm 2) the min heap can be used to locate the top k head vertices in time logarithmic to the number of neighborlists in the expired graph. When the block formation is complete, for each neighborlist included in the block, their entry in the min heap needs updating.

Observation 2. The selection of new head vertices for expansion can be naïvely performed by considering all head vertices in the expired graph. Note that this step is performed during expansion of the candidate blocks (Algorithm 3), which is called many times during the creation of the block (Algorithm 1), until all the candidate blocks are all full. As a result, it is costly to iterate over the head vertices of the expired graph. This is also unnecessary, as we are only interested in expansions that add at least one non-dangling edge. This means that we are looking for expansions that involve head vertices that currently appear in the temporal neighborlists of the candidate block. As such, we can iterate over these vertices while considering expansions, rather than the entire list of head vertices in the expired graph.

Observation 3. In Algorithm 3, the most critical step is to find all expansions of a given neighborlist that would add at least one non-dangling edge to the candidate block. A naïve implementation may consider all expansions to determine the ones that satisfy this condition. Given the expired graph $\mathcal{H}$, the partial candidate block B, and a head vertex v, the number of all possible expansions is $|\mathcal{H}[v].e| - |B[v].e|$. However, it wasteful to consider all edges in this list. In particular, if t is the maximum of the time stamps in the candidate block B, that is $t = \max_{u \mapsto l \in B} l.t_e$, then all to be added edges with a time stamp greater than t are guaranteed to be dangling. This means that we can maintain the maximum time stamp as part of the candidate block and limit the temporal extents of the expansions to this maximum.

Observation 4. In Algorithm 3, there are several possible expansions and for each, the utility metric U is computed. For two potential expansions (v, t_1) and (v, t_2) that share their head vertices, most of the utility metric computation is repeated. Assuming $t_1 < t_2$, the utility metric for the second expansion can be computed by only considering the additional edges it brings. To achieve this, the expansions are considered in head vertex and time extent order, and the utility metric is computed incrementally, avoiding repeated computation.

6.2 Structure of a Candidate Block

To support efficient implementation, we extend the candidate blocks with additional state. In particular, we maintain the following:

The size of the block, the total edge count, the dangling edge count, and the number of connected pairs of head vertices are maintained, so that the utility function can be computed incrementally (Observation 4).

A hash set containing the vertices present in the neighborlists of the candidate block is maintained, so that the head vertices to be used for new expansions can be quickly determined (Observation 2).

Finally, a priority queue on the largest time stamps of the candidate block's neighborlists is maintained, so that temporal extents of the potential expansions can be upper bounded (Observation 3).

6.3 Operation of the Algorithm

With the new candidate block structure at hand, the algorithm operates as follows, assuming the k initial partial candidate blocks are determined (see Observation 1). First, we iterate over the hash set containing the present vertices in the block. For each vertex in the set, we consider extending its temporal neighborlist in the candidate block. If the vertex is not present in the candidate block, we add a new temporal neighborlist for it. To find potential expansions for a head vertex, we start from the oldest edge in its neighborlist within the expired graph that is not already contained in the candidate block, and consider extending the temporal extent up to that edge's time stamp. If the added edge is not a dangling one, then we have a valid expansion. We continue this search process for immediately following edges with increasing time stamps to find additional expansions. Checking whether an edge considered for expansion is a

dangling edge or not requires checking the temporal extent of the neighborlist associated with the other vertex incident upon the edge (the one that is not the head vertex of the neighborlist being expanded) within the candidate block. We continue testing edges until we reach an edge whose time stamp is higher than the highest time stamp contained in the priority queue maintained over the time stamps of the neighborlists of the candidate block.

Given an edge (v, u, t) that satisfies these conditions, we perform the expansion by adding all the edges that have a time stamp smaller or equal to t from the expired graph's neighborlist ($\mathcal{H}[v].e$) to that of the candidate block's ($B[v].e$). While the edges are added, we update the block size, edge count, and the dangling edge count. This makes it possible to compute the locality of the extended candidate block incrementally. We then update the present vertices, as the newly inserted edges can bring new vertices that were not present in the neighborlists of the candidate block before. Finally, the priority queue of time stamps is updated.

6.4 Time Complexity

The running time of the efficient implementation of the block formation algorithm can be easily upper bounded. The initial candidate list can be determined in time $\mathcal{O}(k \cdot \log N_r)$, where N_r is the number of neighborlists in the expired graph. The k initial candidate blocks are expanded at each step. In the worst case, each expansion can bring only a single edge. If we denote the number of edges a block can hold as M_b, then the block expansion is performed $\mathcal{O}(k \cdot M_b)$ times. A given expansion cannot test more than M_b edges and each test requires $\log N_b$ operations (due to the time stamp updates), where N_b is the number of neighborlists in the block. As such the computational complexity of the algorithm is given by $\mathcal{O}(k \cdot (\log N_r + {M_b}^2 \cdot \log N_b))$.

In practice, the algorithm achieves a running time complexity closer to $\mathcal{O}(k \cdot (\log N_r + M_b \cdot \log N_b))$, as the number of potential expansions considered during the candidate block expansion is much lower than the number of edges M_b, since the close temporal alignments of the neighborlists will effectively cut the viable expansions based on Observation 3.

7 Experimental Evaluation

In this section we evaluate our system, with a special focus on the impact of different blocking algorithms on the performance.

7.1 Experimental Setup

We first provide details on our implementation, evaluation environment, the data and query sets used, and the metrics employed in our evaluation.

7.1.1 Implementation

Our implementation was done in C++ using LLVM 3.2 compiler. For the neighborhood query index, we use the LevelDB library [23] and for the vertex query index, we use the libspatialindex library [12] with a custom implementation of an LRU buffer management policy. For workload generation, we use Boost Graph Library [32].

7.1.2 Environment

For the experiments, we used a machine with a 2.2GHz Intel i7 processor that has 32 KB L1 data, 32 KB L1 instruction, 256 KB L2 (per core), 6 MB L3 (shared) cache, and 8 GB of main memory. The processor has four cores, but our implementation only uses a single core (except for workload generation). The disk used is a 500 GB Toshiba SATA 5400RPM disk. The operating system used was CentOS GNU/Linux with the 2.6 kernel and ext4 filesystem. It is worth noting that our evaluation heavily focuses on I/O impact of algorithmic techniques, and not on absolute performance.

7.1.3 Data Sets

We use synthetically generated data sets as well as a real data set from Twitter.

Synthetic data sets: We use an R-MAT-generated [4] power-law graph with 100K nodes and 1M edges to generate 100M interactions. An exponentially distributed inter-arrival time with a mean of 10 milliseconds is used for this purpose. To generate interactions, we pick a random vertex group using a Zipf distribution over $10K$ groups of vertices. The higher the rank of a vertex group, the more likely it is for the vertices in the group to initiate an interaction. Once the vertex group is determined, we then pick a vertex uniformly at random from the group. This vertex becomes the source vertex of the interaction. We pick the destination vertex of the interaction uniformly at random from the neighbors of the source vertex. Once a destination vertex is chosen, it is temporarily moved to a vertex group with a rank higher or equal to that of the vertex group of the source vertex. The rank is selected uniformly at random. If and when the destination vertex is selected as a source vertex in the future, it is inserted back into its original vertex group. The default value of the Zipf skew parameter is set to 1.5. Setting it higher increases the temporal and spatial locality of interactions.

Real data set: As the real data set, we use Twitter messages from the time interval 14 May-05 June 2013. The data set contains messages from 500K most prolific Twitter users from Turkey. Interestingly, the time interval during which the data is collected coincides with the 2,013 protests in Turkey [30] that generated massive amount of discussion and interaction in the social media. We convert the twitter data into an interaction graph, as follows: If a tweet from user x mentions another user y, then an interaction between x and y is established. Additionally, a message sent by a user x always generates an interaction between x and its virtual pair x'. We add the tweet text as data to the interactions.

7.1.4 Queries

We use several kinds of queries to evaluate the performance of block formation algorithms and our interaction graph implementation. The first two are the fundamental queries we introduced in Section 4: time range vertex queries and time range neighborhood queries. These are directly supported by the APIs provided by our interaction graph library.

In addition to these, we support basic graph kernels over the historical interactions. These are:

TABLE 1
Defaults Values for the Parameters Used

description	*default*	*range*
C, the initial candidate selection policy	*Old*	[*Old*,*New*,*Min*,*Max*]
k, the number of initial candidates	10	$[1, 32]$
m, relative size of the expired heap	0.1	$[0.00125, 0.16]$
b, block size (in bytes)	1024	$[512, 16384]$
z, skew in vertex popularity	1.5	$[0.5, 3.0]$

- n-hop neighborhood: given a vertex and a time range, find all interactions that are within n edges away from the vertex.
- clustering coefficient [36]: given a time range, for all vertices involved in interactions during that time range, compute the clustering coefficient.
- n-step random walk: given a vertex and a time range, perform a random walk of n steps over the vertices and edges during that time range.
- page rank [29]: given a time range, for the sub-graph consisting of vertices and edges within that time range, compute the page rank of all vertices.

7.1.5 Metrics

In our evaluation, we rely on four metrics. The most fundamental ones are the amount of time taken and disk I/O performed to answer queries. Another metric we use is the throughput of edge insertion. Note that this depends not only on the I/O cost, but also on the running time cost of the block formation algorithm used. Finally, we use the storage overhead as a metric.

7.1.6 Defaults

Table 1 gives the descriptions, default settings, and ranges for the parameters we study in our experiments. The in-memory live graph is large enough to store one million interactions.

7.2 Micro Benchmarks

We use synthetic data sets to study the impact of various workload, algorithm, and system parameters on the performance of the system. In terms of the block formation algorithms, we consider several alternatives. We name variations of the algorithm described in Section 5, which differ in terms of their initial candidate selection policies, as *GE-Old*, *GE-New*, *GE-Min*, *GE-Max*, and *GE-Rand*.

In addition to these, we also include three additional algorithms, which do not rely on candidate expansion. These algorithms are called *G-Old*, *G-Max*, and *G-Rand*. They iteratively pick one of the temporal neighborlists in the expired graph and add the first half edge from that neighborlist to the block. This continues until the block is full. The *G-Old* approach picks the temporal neighborlist whose first edge is the oldest among all, *G-Max* picks the one that has the largest size, and *G-Rand* picks one uniformly at random.

7.2.1 Effectiveness of the Query Indexes

We first investigate the effectiveness of the vertex and neighborhood query indexes. Fig. 3 plots the query execution time (using the left y-axis) and the result size (using the right y-axis) as a function of the size of the query's time range. The results are plotted for both vertex queries and edge queries (neighborhood queries).

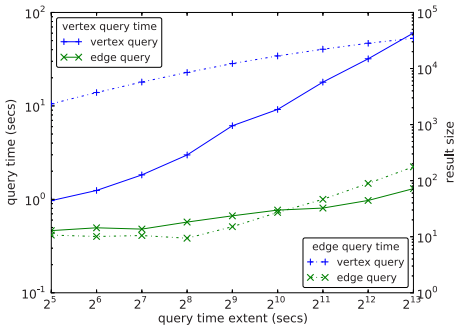


Fig. 3. Effectiveness of the vertex and neighborhood query indexes.

We make two observations from the figure. First, the increase in the query execution time is proportional to the increase in the result size for edge queries, as we enlarge query time ranges. This is expected as the number of edges retrieved and thus the amount of I/O performed is proportional to the query time extent.

Second, and more interestingly, the increase in the query evaluation time is dis-proportionally high compared to the increase in the result size for vertex queries, as we enlarge the query time ranges. This is due to the fact that vertex query index is designed to support arbitrarily small time ranges in queries and as such the number of entries it visits is proportional to the size of the query time range. However, the number of vertices that have interactions during a given time range eventually converges to the total number of vertices. In other words, when a large time range is used, the vertex query index retrieves various entries that do not contribute to the end result, since the vertices retrieved are already in the query result.

To support large query time ranges more efficiently, we could easily create a smaller version of the vertex query index that uses per day granularity for its time stamps. However, for most graph analytics, a time range vertex query is run once, whereas the time range neighborhood queries are run successively to traverse the edges of the vertices of interest. As such, we do not further optimize vertex queries.

7.2.2 Locality and Block Formation Algorithms

Fig. 4 plots the locality metric (Section 5, Equation (3)) as a function of the number of initial candidates for different block formation algorithms. Note that algorithms that are not based on block expansion do not rely on initial candidates. As such, their results do not show variation as the number of candidates is altered.

We make three main observations. First, the *GE-Old* approach achieves the best locality values. The highest locality achieved is 0.43, which is around 50 percent higher than the closest one (*GE-Rand* and *GE-Min*). Second, we see that algorithms that rely on expansion perform better in terms of providing higher locality. Last, we see that increasing the number of initial candidates provides improved locality for the expansion based algorithms. This increase is more pronounced for the *GE-Old* approach. Its locality

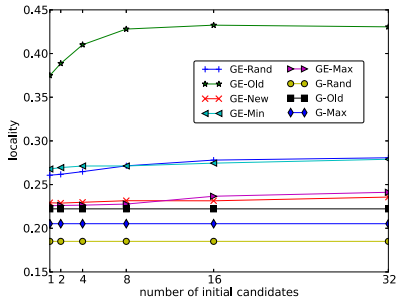


Fig. 4. Locality of different block formation algorithms.

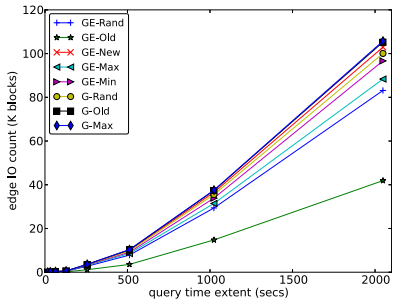


Fig. 6. Query I/O for different blocking algorithms.

shows a 15 percent increase going from one candidates to 16 candidates. However, additional increase does not bring further improvements. In fact, the locality is flat after eight candidates, justifying our default setting of 10.

7.2.3 Query Performance of Blocking Algorithms

Figs. 5 and 6 plot the query execution time and the amount of I/O performed during query execution, respectively, as a function of the query time extent, for different block formation algorithms. The query used for these experiments and others that follow are 3-hop neighborhood queries.

We observe that the *GE-Old* algorithm provides the lowest query running time compared to other block formation algorithms. In particular, the closest approach (*GE-Rand*) has 70 percent higher running time compared to *GE-Old*, and the worst approach (*G-Rand*) has 158 percent higher running time. Results for I/O are even more pronounced, closest approach having 98 percent higher I/O cost compared to *GE-Old*.

7.2.4 Impact of the Expired Graph Size

Fig. 7 plots the I/O count (using the left y-axis) and locality (using the right y-axis) as a function of the expired graph size, for different query time extents. The expired graph size is altered by changing the value of the parameter m, which sets the size relative to that of the temporal edge FIFO.

We observe that higher expired graph sizes provide lower I/O. However, there is diminishing returns after $m = 0.05$. The impact of the expired graph size is very pronounced for lower values and especially for the large query extents. For instance, for query time range of around half an hour, the I/O cost can be as much as 150 percent higher if the expired graph size is not set properly. We see that the default value of 0.1 provides good performance across different query time range values.

The figure also shows the reason why increasing expired graph sizes improve the I/O performance. Clearly, the locality is increasing as the expired graph size increases. And similar to the I/O cost, the gains in locality diminish after a certain size. The increased expired graph size provides additional flexibility to the block formation algorithm in terms of selecting the spatially close edges to place together, which improves locality, and in turn the query I/O performance.

7.2.5 Impact of Block Size on Query Performance

Fig. 8 shows query time as a function of the block size for different query time ranges. We see that for large query time ranges, smaller blocks provide better performance. For small query time ranges, we observe that there is a optimal block size at which the best performance is achieved. Note that higher block sizes are expected to reduce the locality (which we study shortly), and as such the increase in query time with increasing block sizes is expected. On the other hand, small block sizes have higher overhead with respect to seek time and one-time overheads. This impact is more pronounced for smaller query ranges because a large portion of the retrieved edges in the block are not used (they are not in the query time range).

Fig. 9 shows the per byte overhead (on the left y-axis) and locality (on the right y-axis) as a function of the block size. The per byte overhead is computed by assuming that the entire graph is just an in memory interaction graph with no overheads for blocking, no index for accelerating vertex queries, and no index for neighborhood queries.

We make two observations from the figure. First, we verify our intuition that larger blocks have lower locality.

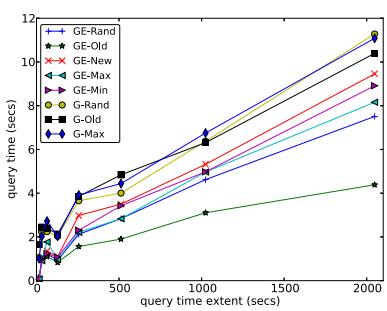


Fig. 5. Query time for different blocking algorithms.

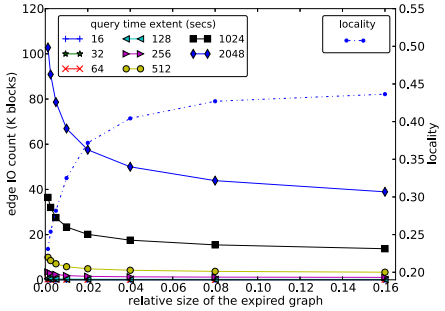


Fig. 7. Impact of expired graph size on query I/O.

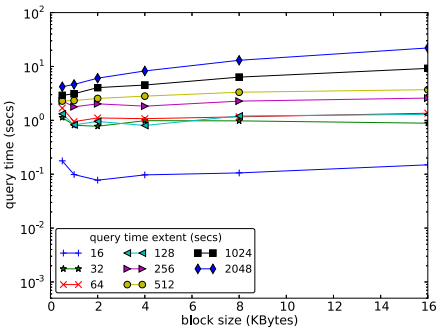


Fig. 8. Query time as a function of the block size.

Second, we see that small blocks have higher per byte overhead. In particular, a 0.5K block can have up to 25 percent higher space overhead on this example. The positive impact of smaller block size on the locality and its negative impact on space overhead together explain the tradeoff observed earlier in Fig. 8.

7.2.6 Impact of Group Popularity Skew on Query Performance

Fig. 10 plots the I/O cost of answering queries when using an interaction graph that employs *GE-Old* blocking algorithm relative to that of *G-Rand* algorithm. This value is plotted on the left y-axis as a function of the skew in group popularities, and for different query time ranges. A higher skew implies that there are a small group of more frequently interacting vertices in the graph. The graph also plots the locality for the *GE-Old* and *G-Rand* algorithms, using the right y-axis. Note that the locality is a property of the historical graph stored on the disk and does not depend on the query time ranges.

Looking at the localities, we observe that the *GE-Old* block formation algorithm, which relies on candidate expansion, is able to take advantage of the additional skew. Its locality increases with increasing skew, even though the rate is decreasing and eventually the locality flattens (after $z > 2.0$). This is in contrast to *G-Rand*, which is unable to take advantange of the skew, as it does not perform expansion. The expansion step of *GE-** algorithms is particularly effective in increasing the spatial locality of the blocks.

Looking at the relative I/O costs, we observe that the *GE-Old* algorithm provides between 25 to 70 percent lower cost compared to *G-Rand*. We also observe that the improvement is more pronounced for smaller query time ranges. But even for the half an hour query time range, we observe between

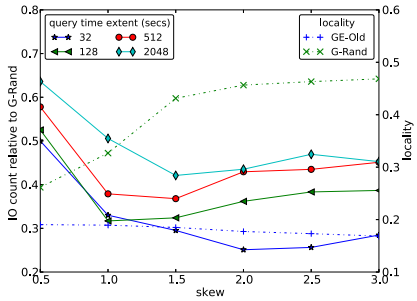


Fig. 10. Impact of group popularity skew on performance.

25 to 60 percent reduction in I/O cost over the range of skew values considered. In general, higher skew brings better improvement. But when the skew increases too much (beyond $z > 1.5$ for most query time ranges), the relative improvement lessens slightly.

7.2.7 Impact of Query Hops on I/O Performance

Fig. 11 plots the query I/O cost of the *GE-Old* blocking algorithm relative to that of the *G-Rand* algorithm, as a function of the number of hops in the query, for different query time ranges.

We observe that the *GE-Old* algorithm provides between 35 to 78 percent improvement in I/O cost compared to *G-Rand*. Interestingly, the improvement is highest when the number of hops in the neighborhood query is 2 (around 78 percent for all query time ranges), which is a significant improvement compared to what is observed for one hop only (between 35 to 45 percent for different time ranges). As the number of hops is further increased, the relative improvement decreases. For the query time range of 2,048 seconds, it stabilizes at around 50 percent improvement, and for 32 seconds, it reaches 67 percent improvement at six hops. In general, the initial increase in the improvement can be attributed to the significant increase in the query result size. However, as the number of hops increase, the number of additional interactions added to the result set is diminishing. This means that additional blocks are read, but just a few of the interactions contained within contribute to the result. As a result, there is less opportunity for *GE-Old* to best *G-Rand*.

7.2.8 Impact of Block Formation Complexity on Throughput

Fig. 12 plots the throughput of edge insertion (on the left y-axis) and locality (on the right y-axis), as a function of the

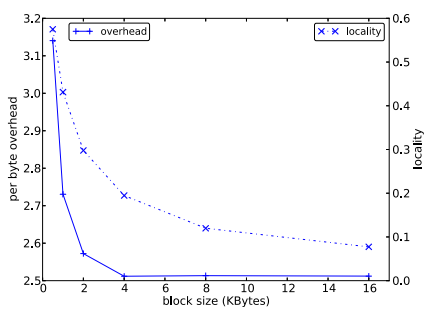


Fig. 9. Impact of block size on storage overhead.

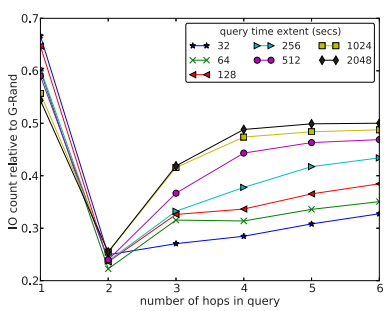


Fig. 11. Impact of query hops on I/O performance.

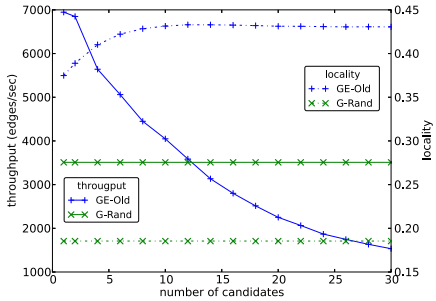


Fig. 12. Impact of block formation on throughput.

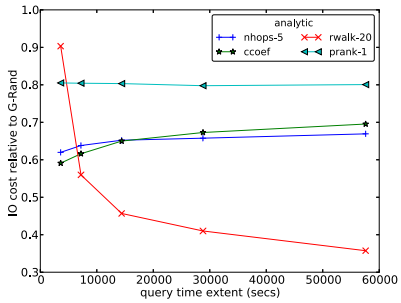


Fig. 13. IO cost for different graph analytic kernels.

number of initial candidates used for the *GE-Old* algorithm. It also plots the same measures for *G-Rand*, but since this algorithm does not perform expansion, the number of initial candidates does not impact performance.

Note that the throughput depends on two factors. The first is the amount of I/O performed. If a block formation algorithm leaves too many dangling edges, then the amount of I/O will increase, since the edge data would be written twice to the disk. Second, the throughput depends on the complexity of the block formation algorithm. For instance, *GE-Old* has a higher running time complexity compared to *G-Rand*, since it performs the expansion step.

When using a single initial candidate, the *GE-Old* algorithm achieves around $2\times$ the throughput of *G-Rand*. With increasing number of candidates, the *GE-Old* starts to provide improved locality, which improves the query performance as we have seen in the earlier experiments. Going from one initial candidates to two brings down the throughput only slightly, as the I/O is still the dominating factor. As the number of initial candidates further increase, the running time complexity starts dominating, resulting in reduced throughput. However, it is important to note that when the locality flattens (around 10 initial candidates), *GE-Old* still has higher throughput compared to *G-Rand*. In summary, *GE-Old* is able to provide better throughput when the number of candidates is set to the smallest value that maximizes the locality. Since there is no point in setting the number of initial candidates any higher than 10, *GE-Old* can provide its best query efficiency, while still providing higher throughput than non-expansion based approaches.

For experiments using faster disks (e.g., SSDs), the running time performance of block formation may become a concern, requiring a parallel implementation of block formation. Since each initial candidate is expanded independently, the block formation is trivial to parallelize. We leave this study as a future work.

7.3 Analytic Kernels

We now look at the query performance using the Twitter data set. Fig. 13 plots the I/O cost of *GE-Old* relative to that of *G-Rand*, as a function of the query time range and for different analytic kernels. The nhops-5 is a neighborhood query with five hops, rwalk-20 is a random walk with 20 steps, ccoef is clustering coefficient, and prank-1 is a single iteration of the page rank. More details can be found in Section 7.1.

We observe that expansion based block formation provides effective reduction in query I/O. For page rank, the improvement is 20 percent and not dependent on the query time range. The clustering coefficient and n-hop neighborhood queries show 40 to 30 percent reduction in I/O, with a slight decrease in improvement as the query time extent increases. Finally, random walk shows the lowest as well as the highest improvements, ranging between 10 to 65 percent. As the query time range increases, the improvement in I/O increases for random walk, but with a decreasing rate.

In general, the ability of *GE-Old* to capture both spatial and temporal localities enables it to reduce the query I/O further, compared to *G-Rand*, which only captures temporal locality.

7.4 Comparison to an RDBMS

To measure the relative performance of our solution compared to a baseline approach, we used Postgres 9.2—an off-the-shelve RDBMS that supports recursive SQL queries. We defined a table clustered on time to keep the Tweets, over which indexes are defined on from user, to user, and time fields. The indexes are crucial in achieving good performance, as a time range limited traversal using recursive SQL relies on joins performed between the temporary result table and the main table, where the join condition involves a equality on from and to user fields and a range condition on the time field.

We used the Twitter data set for this experiment. Compared to our approach, Postgres was $3.2\times$ slower in inserting the data. This is mostly because RDBMSs are not optimized for streaming inserts. Multiple secondary indexes we have used to speedup querying further exacerbates the problem, due to the index update cost. On the other hand, our approach has the I/O advantage of performing batch inserts as blocks. Yet, our approach also has the CPU disadvantage of having to compute a good blocking before it can write the block to disk.

Fig. 14 plots the relative improvement in query time for n-hop temporal neighborhood queries, as a function of different time interval sizes, and for different values of n. For small query time ranges, we see up to $60\times$ performance with our approach, compared to Postgres. For small ranges, the query result sizes as well as the I/O performed is small, so the constant overheads of the RDBMS dominates. For larger time ranges, the performance improvement with our approach is around $10\times$, that is an order of magnitude.

Interestingly, the slowness of the RDBMS approach is not due to poor index use or query planning. For instance, the speedup is not too different for different number of hops (more hops mean more joins). The key disadvantage of the

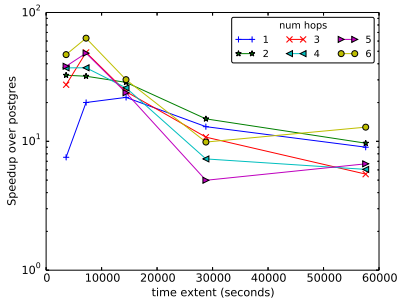


Fig. 14. Speed-up of n-hop queries relative to RDBMS implementation.

RDBMS is that it is unable to store the interactions in an order that considers both temporal and spatial locality. Our approach achieves this by storing a set of temporal neighborlists as a block, defining a metric that captures both kinds of localities, and forming blocks using this metric to minimize random accesses.

7.5 Summary

Our results show that the *GE-Old* algorithm provides the best performance with respect to minimizing query I/O. With the initial candidate size set to $k = 10$, it can still outperform the *G-Rand* algorithm in terms of the update throughput it supports, while providing significantly better query I/O performance. Further increasing k does not bring much improvement in query I/O, but reduces the update throughput. Thus, the default setting of $k = 10$ is a good one, and the performance is not sensitive to minor changes around this ideal value. The relative size of the expired graph in terms of the number of edges it holds relative to the temporal edge FIFO, that is m, is best set to a value close to 0.1, not going below 0.05. Larger values do not bring any additional advantage in terms of query I/O, unnecessarily increasing the memory requirements. Too small values hurt the query I/O performance. In general, the expired graph uses less than 5 percent extra memory on top of the live graph and the temporal edge FIFO. Finally, we advocate relatively small block sizes of around 1-2 KB. This is because larger blocks hurt locality. Block sizes smaller than 1K increase the storage overhead too much and are best avoided.

Our system architecture is designed to handle streaming interactions, as it uses an in-memory buffer to keep the recent window of interactions. It also provides efficient querying, since it employs smart blocking techniques to write temporally and spatially local interactions to disk in batches. The *GE-Old* algorithm we use for this purpose is effective compared to other alternatives. The effectiveness of *GE-Old* is more pronounced when the interactions exhibit spatial locality (a community of users interact with each other) and temporal locality (interactions happen close in time). Our experiments show that Twitter mention graph exhibits these characteristics.

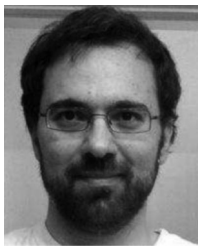
8 Conclusion

We have developed a system that supports temporal storage and querying of evolving interaction graphs. The system maintains a buffer consisting of recent interactions in-memory and continuously expires older interactions by writing them to the historical graph stored on the disk. Indexes are maintained over the historical graph to support time range vertex and neighborhood queries, which can be used to build higher level temporal graph analytics. We introduced block formation algorithms that are used to optimize the grouping of expired edges into blocks, such that the temporal and spatial locality is increased, and as a result query performance is improved. We have shown the effectiveness of these algorithms using synthetic as well as real-world interaction graphs. In particular, we have shown that the block formation algorithms are able to reduce the I/O required to answer queries as well as for writing the live edges to disk, yet they are fast enough to avoid any diverse effects on the edge insertion throughput.

References

[1] C. Aggarwal and H. Wang, "Graph Data Management and Mining," *A Survey of Algorithms and Applications*, C. Aggarwal, ed., Springer, 2010.
[2] C.C. Aggarwal, Y. Li, P.S. Yu, and R. Jin, "On Dense Pattern Mining in Graph Streams," *Proc. Very Large Databases Conf. (PVLDB)*, pp. 975-984, 2010.
[3] T. Berger-Wolf and J. Saia, "A Framework for Analysis of Dynamic Social Networks," *Proc. ACM Int'l Conf. Knowledge Discovery and Data mining (SIGKDD)*, pp. 523-528, 2006.
[4] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A Recursive Model for Graph Mining." *Proc. Fourth SIAM Int'l Conf. Data Mining*, 2004.
[5] G. Cormode and S. Muthukrishnan, "Space Efficient Mining of Multi-Graph Streams," *Proc. ACM Int'l Symp. Principles of Database Systems (PODS)*, pp. 271-282, 2005.
[6] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf., *Computational Geometry: Algorithms and Applications*. Springer Verlag, 1997.
[7] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Proc. USENIX Symp. Operating System Design and Implementation (OSDI)*, pp. 137-150, 2004.
[8] S. Fortunato, "Community Detection in Graphs," *Physics Reports*, vol. 483, nos. 3-5, pp. 75-174, 2009.
[9] B. Gedik, "Generic Windowing Support for Extensible Stream Processing Systems," *Software: Practice & Experience*, Mar. 2013, DOI: 10.1002/spe.2194.
[10] Apache Giraph, giraph.apache.org/, Retrieved June, 2013..
[11] J.E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. "PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs," *Proc. USENIX Symp. Operating System Design and Implementation (OSDI)*, pp. 17-30, 2012.
[12] M. Hadjieleftheriou, E.G. Hoel, and V.J. Tsotras, "SaIL: A Spatial Index Library for Efficient Application Integration," *GeoInformatica*, vol. 9, no. 4, pp. 367-389, 2005.
[13] I. Hoque and I. Gupta, "Disk Layout Techniques for Online Social Network Data," *IEEE Internet Computing*, vol. 16, no. 3, pp. 24-36, May/June 2012.
[14] U. Kang, C.E, Tsourakakis, and C. Faloutsos., "PEGASUS: A Peta-Scale Graph Mining System - Implementation and Observations," *Proc. IEEE Int'l Conf. Data Mining (ICDM)*, pp. 229-238, 2019.
[15] U. Kang, H. Tong, J. Sun, C.-Y. Lin, and C. Faloutsos, "Gbase: A Scalable and General Graph Management System," *Proc. ACM Int'l Conf. Knowledge Discovery and Data mining (SIGKDD)*, pp. 1091-1099, 2011.
[16] G. Karypis and V. Kumar., "Multilevel Graph Partitioning Schemes," *Proc. Int'l Conf. Parallel Processing (ICPP)*, pp. 113-122, 1995.
[17] D. Kempe, J. Kleinberg, and E. Tardos, "Maximizing the Spread of Influence through a Social Network," *Proc. ACM Int'l Conf. Knowledge Discovery and Data mining (SIGKDD)*, pp. 137-146, 2003.
[18] U. Khurana and A. Deshpande., "Efficient Snapshot Retrieval over Historical Graph Data," *Proc. IEEE Int'l Conf. Data Eng. (ICDE)*, 2013.

[19] R. Kumar, J. Novak, and A. Tomkins, "Structure and Evolution of Online Social Networks," *Proc. ACM Int'l Conf. Knowledge Discovery and Data Mining (SIGKDD)*, pp. 611-617, 2006.
[20] A. Kyrola, G. Blelloch, and C. Guestrin, "GraphChi: Large-Scale Graph Computation on Just a PC," *Proc. USENIX Symp. Operating System Design and Implementation (OSDI)*, pp. 31-46, 2012.
[21] F.T. Leighton and S. Rao, "Multicommodity Max-Flow Min-Cut Theorems and Their Use in Designing Approximation Algorithms," *J. ACM*, vol. 46, no. 6, pp. 787-832, 1999.
[22] J. Leskovec, J. Kleinberg, and C. Faloutsos, "Graph Evolution: Densification and Shrinking Diameters," *ACM Trans. Knowledge Discovery from Data*, vol. 1, no. 1, pp. 2:1-2:41, 2007.
[23] leveldb - A Fast and Lightweight Key/Value Database Library by Google, http://code.google.com/p/leveldb/, Retrieved Mar. 2012.
[24] G. Malewicz, M.H. Austern, A.J. Bik, J.C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A System for Large-Scale Graph Processing," *Proc. ACM Int'l Conf. Management of Data (SIGMOD)*, pp. 135-146, 2010.
[25] J. Mondal and A. Deshpande, "Managing Large Dynamic Graphs Efficiently," *Proc. ACM Int'l Conf. Management of Data (SIGMOD)*, pp. 145-156, 2012.
[26] A.A. Nanavati, G. Siva, G. Das, D. Chakraborty, K. Dasgupta, S. Mukherjea, and A. Joshi, "On the Structural Properties of Massive Telecom Call Graphs: Findings and Implications," *Proc. ACM Int'l Conf. Information and Knowledge Management (CIKM)*, pp. 435-444, 2006.
[27] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The Log-Structured Merge-Tree (LSM-Tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351-385, 1996.
[28] G. Ozsoyoglu and R. Snodgrass, "Temporal and Real-Time Databases: A Survey," *IEEE Trans. Knowledge and Data Eng.*, vol. 7, no. 4, pp. 513-532, Aug. 1995.
[29] L. Page, S. Brin, R. Motwani, and T. Winograd, "The Pagerank Citation Ranking: Bringing Order to the Web," Technical Report SIDL-WP-1999-0120 Stanford InfoLab, 1999.
[30] "Turkey Protests Spread from Istanbul to Ankara, Euronews", http://www.euronews.com/2013/05/31/turkey-protests-spread-from-istanbul-to-ankara/,Retrieved June, 2013.
[31] B. Shao, H. Wang, and Y. Li, "Trinity: A Distributed Graph Engine on a Memory Cloud," *Proc. ACM Int'l Conf. Management of Data (SIGMOD)*, 2013.
[32] J.G. Siek, L.-Q. Lee, and A. Lumsdaine, *Boost Graph Library, The: User Guide and Reference Manual*. Addison-Wesley, 2002.
[33] R. Steinhaus, "G-Store: A Storage Manager for Graph Data," master's thesis, Univ. of Oxford, 2011.
[34] L. Tang, H. Liu, J. Zhang, and Z. Nazeri, "Community Evolution in Dynamic Multi-Mode Networks," *Proc. ACM Int'l Conf. Knowledge Discovery and Data Mining (SIGKDD)*, pp. 677-685, 2008.
[35] Twitter, http://www.twitter.com/, Retrieved Mar., 2012.
[36] D.J. Watts and S.H. Strogatz, "Collective Dynamics of 'Smallworld' Networks," *Nature*, vol. 393, pp. 440-442, 1998.
[37] P. Zhao, C.C. Aggarwal, and M. Wang, "gSketch: On Query Estimation in Graph Streams," *Proc. Very Large Databases Conf. (PVLDB)*, pp. 193-204, 2011.



Buğra Gedik received the PhD degree in computer science from Georgia Tech. He is currently on the faculty of the Computer Engineering Department, İhsan Doğramacı Bilkent University, Turkey. His research interests are in distributed data-intensive systems, with a particular focus on stream computing and big data technologies.

Rajesh Bordawekar received the MS and PhD degrees in computer engineering from Syracuse University. He is currently a research staff member at the IBM T.J. Watson Research Center. His research is on understanding the interactions between applications, programming languages/runtime systems, and computer architectures.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.