ImmortalGraph: A System for Storage and Analysis of Temporal Graphs

YOUSHAN MIAO, University of Science and Technology of China
WENTAO HAN and KAIWEI LI, Tsinghua University
MING WU, FAN YANG, and LIDONG ZHOU, Microsoft Research Asia
VIJAYAN PRABHAKARAN, Microsoft Research Silicon Valley
ENHONG CHEN, University of Science and Technology of China
WENGUANG CHEN, Tsinghua University

Temporal graphs that capture graph changes over time are attracting increasing interest from research communities, for functions such as understanding temporal characteristics of social interactions on a time-evolving social graph. ImmortalGraph is a storage and execution engine designed and optimized specifically for temporal graphs. Locality is at the center of ImmortalGraph's design: temporal graphs are carefully laid out in both persistent storage and memory, taking into account data locality in both time and graph-structure dimensions. ImmortalGraph introduces the notion of *locality-aware batch scheduling* in computation, so that common "bulk" operations on temporal graphs are scheduled to maximize the benefit of in-memory data locality. The design of ImmortalGraph explores an interesting interplay among locality, parallelism, and incremental computation in supporting common mining tasks on temporal graphs. The result is a high-performance temporal-graph system that is up to 5 times more efficient than existing database solutions for graph queries. The locality optimizations in ImmortalGraph offer up to an order of magnitude speedup for temporal iterative graph mining compared to a straightforward application of existing graph engines on a series of snapshots.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming

General Terms: Design, Algorithms

Additional Key Words and Phrases: Concurrent computing, temporal graph, graph algorithms

ACM Reference Format:
Youshan Miao, Wentao Han, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Enhong Chen, and Wenguang Chen. 2015. ImmortalGraph: A system for storage and analysis of temporal graphs. ACM Trans. Storage 11, 3, Article 14 (July 2015), 34 pages.
DOI: http://dx.doi.org/10.1145/2700302

1. INTRODUCTION

Graphs can naturally capture connections and relationships. Real-world graphs often evolve over time as connections and relationships change [Leskovec et al. 2005]. There is a growing interest in analyzing not only the static structure of graphs, but also their time-evolving properties. For example, consider studying diameter changes of

This work was done while the first three authors were with Microsoft Research Asia as research interns.
A preliminary version of this work was presented at the 9th European Conference on Computer Systems (EuroSys), Amsterdam, Netherlands, Apr. 2014.
Authors' address: Y. Miao, University of Science and Technology of China; email: mooncake@mail.ustc.edu.cn; W. Han and K. Li; emails: {hwt04, likw14}@mails.tsinghua.edu.cn; M. Wu, F. Yang, L. Zhou, and V. Prabhakaran; emails: {miw, fanyang, lidongz, vijayanp}@microsoft.com; E. Chen; email: cheneh@ustc.edu.cn; W. Chen; email: cwg@tsinghua.edu.cn.

DOI: http://dx.doi.org/10.1145/2700302

an evolving social network [Leskovec et al. 2005], characterizing social relationships according to changing user activities in online social networks [Wilson et al. 2009], and observing how Web-page ranks change over time [Yang et al. 2007]. New algorithms are also proposed specifically for evolving graphs to extract new insights [Wilson et al. 2009; Lerman et al. 2010]. These emerging applications demand a storage and computation system designed and optimized specifically for *temporal graphs* that maintains all historical information of evolving graphs for query and computation.

ImmortalGraph is a storage and computation engine designed specifically for temporal graphs. ImmortalGraph keeps all historical information of a graph, over which it enables efficient temporal graph query and iterative temporal graph computation. It introduces *snapshot group* as a basic unit of storage and processing for temporal graphs that captures temporal graph information within a specified time interval. Snapshot groups allow temporal graph information to be expanded incrementally as new snapshot groups are generated continuously and added as the graph evolves. The resulting series of snapshot groups can be distributed and replicated in a cluster of machines.

ImmortalGraph achieves high efficiency through careful optimizations for data locality. For a temporal graph, the data layout can exhibit either *time(-dimension) locality*, in which the states of a vertex (or an edge) at two consecutive time points are laid out consecutively, or *structure(-dimension) locality*, in which the states of two neighboring vertices at the same time point (i.e., in the same graph snapshot) are laid out close to each other. ImmortalGraph identifies two storage layouts—representing locality on the two dimensions, respectively—with certain index structures that are optimized for different access patterns. Instead of attempting to strike an appropriate compromise between the two, ImmortalGraph leverages replication where different replicas store different layouts.

ImmortalGraph pays special attention to *temporal graph mining* that understands graph evolution over time. This involves running a graph-mining algorithm, designed for a static graph, on a series of *snapshots*. Temporal graph mining is more expensive than graph mining on a static graph. Nevertheless, the addition of the time dimension in ImmortalGraph gives rise to interesting and rich new opportunities beyond existing graph engines that work on static graphs [Malewicz et al. 2010; Low et al. 2012; Gonzalez et al. 2012; Prabhakaran et al. 2012; Roy et al. 2013]. The parallel in-memory graph engine of ImmortalGraph is designed to enable efficient temporal graph mining both on multicore machines and in distributed settings. Fundamentally, the graph engine design of ImmortalGraph centers on two issues: temporal graph in-memory layout and graph computation scheduling. Rather than taking the straightforward approach of applying an existing graph engine on each temporal graph snapshot, ImmortalGraph proposes *locality-aware batch scheduling* (LABS) with two key observations.

First, time-locality and structure-locality are not created equal. Time-locality can be arranged "perfectly" because time is linear, while structure-locality can only be approximated because it is challenging to project graph structure into a linear space. The design of ImmortalGraph therefore favors time-locality for temporal graph mining when it lays out multiple graph snapshots in memory. Second, ImmortalGraph schedules graph computation to leverage time-locality in its in-memory temporal graph layout. Such co-design of scheduling and layout is for maximizing the benefits of locality. Traditional graph engines arrange computation around each vertex (or each edge) in a graph; temporal graph engines, in addition, calculate results across multiple snapshots. ImmortalGraph *batches* operations associated with each vertex (or each edge) across multiple snapshots instead of batching operations for vertices/edges within a certain snapshot. These seemingly simple observations have proven effective on different graph computing implementations whether they are stream-based, push-based, or pull-based, as described in Section 4.7.

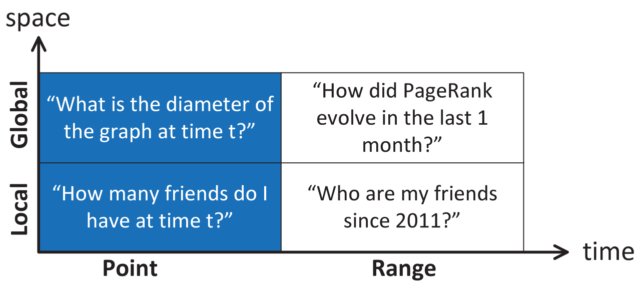

space
Global
Local
"What is the diameter of the graph at time t?"
"How did PageRank evolve in the last 1 month?"
"How many friends do I have at time t?"
"Who are my friends since 2011?"
Point
Range
time

Fig. 1. Classification of temporal graph applications.

To execute efficiently on a multicore server, the ImmortalGraph graph engine design further considers issues related to parallelism, such as the impact of locking to avoid conflicts. ImmortalGraph also examines interplay with incremental graph computation that might help speed up iterative graph mining on multiple snapshots [Cheng et al. 2012].

The major contributions of our work are as follows: (1) We carefully design two data layouts for temporal graph storage and leverage replication to provide good data locality for specific applications; (2) we propose a novel locality-aware scheduling for iterative computations on temporal graphs to achieve higher performance and thoroughly investigate its interplay with parallelism and incremental computation; and (3) we implement this temporal graph system—ImmortalGraph—to demonstrate the effectiveness of the data layouts and locality-aware scheduling design through extensive evaluations on real-world temporal graphs with billions of updates.

The rest of this article is organized as follows. We describe motivating examples and design challenges in Section 2. We explain the ImmortalGraph graph layout, locality-aware scheduling, and programming interface in Sections 3, 4, and 5, respectively. We present detailed evaluation results in Section 6. Finally, we discuss related work in Section 7 and conclusions in Section 8.

2. IMMORTALGRAPH OVERVIEW

We begin this section by presenting workloads on temporal graphs, then discuss challenges in building a system to support them.

2.1. Motivating Examples

A temporal graph tracks *all* the information relevant to the evolution of a graph, including every graph edit activity, such as the addition of a vertex or deletion of an edge, along with its timestamp. Such a temporal graph store enables new applications that can be used to analyze the past, build detailed metrics of a graph at a specific instance, and even predict future trends.

Workloads on a temporal graph can be broadly classified into four quadrants, as shown in Figure 1. Queries can be point-in-time and access a subset or the whole graph. For example, a query to compute the diameter of a graph at time t would construct and traverse a graph snapshot at t to find the longest shortest path. Current graph management systems can handle point-in-time queries only at "now." Graph queries can also span a time range. For example, a query to find all friends acquired since a certain year scans the graph to find new edges added to a vertex in the given time range.

Some temporal graph queries are simple ones that mostly access a portion of a temporal graph *once*. Since it is inconceivable to store the entire graph history in memory, such simple temporal queries result in random-storage I/O. Therefore, it is important to minimize I/O numbers and improve access locality of I/O bound queries.

In addition to simple temporal graph queries, a temporal graph store should also support iterative computations that traverse a graph *repeatedly* for a range of discrete time points. For example, PageRank can be computed on a series of Web graph snapshots to capture the trend of how a page's rank changes. Another example of such temporal mining is to measure change in centrality—such as celebrities' importance in a social network. Unlike temporal queries, iterative computations perform intensive processing on their in-memory working sets and can have high concurrency with multicores. Therefore, to improve the performance of temporal iterative computations, we must focus on improving in-memory accesses and minimizing concurrency control overheads.

2.2. Challenges

While temporal graphs enable richer analysis, we face three challenges when building a system for them.

2.2.1. Storing Updates. First, there is an inherent tradeoff between storing a precomputed graph state (which enables fast queries) and the space it occupies in memory and disk (or SSD). Consider a straw man approach that stores every graph update activity in a log. While such a format is compact and simple to implement, running a temporal graph query requires expensive reconstruction of all graph snapshots within the queried time range. Contrast this with an alternate straw man approach in which we checkpoint a full graph snapshot whenever it is modified. This approach produces graphs that are easier to query, but introduces too much redundant graph information.

To create a compact layout without sacrificing performance, ImmortalGraph uses *snapshot groups*. A snapshot group, G_{t_1,t_2}, consists of a graph state in the time range $[t_1, t_2]$. Particularly, it contains a checkpoint of the entire graph at the start time t_1 and all graph updates made until t_2. Therefore, a temporal graph consists of a series of snapshot groups for successive time ranges. A snapshot group G_{t_1,t_2} contains enough information to access a graph snapshot at any point in the time range. However, accessing a snapshot at a time t after t_1 is more expensive because ImmortalGraph must read all updates made from t_1 to t and merge them into t_1's checkpoint. ImmortalGraph further allows users to specify a redundancy ratio threshold (representing the allowed maximum percentage of redundant data) to control snapshot group size.

2.2.2. Optimizing Layout. The second challenge arises from conflicting requirements for temporal and spatial locality in temporal graph queries. We observe that, depending on the type of query, that is, where it fits in the four quadrants of Figure 1, different layouts can improve performance. To illustrate this, consider the graph shown in Figure 2 that is created at time t_0 and then modified at times t_1, t_2, and t_3. Conceptually, the changes can be viewed across two dimensions of time and space, as shown in the bottom half of Figure 2, in which a point represents a change to a vertex at a specific time.

Temporal graph updates can either be stored in structure-locality or time-locality order within each snapshot group. The structure-locality format is simply a concatenation of vertical segments on the space–time axis, whereas the time-locality format is a concatenation of horizontal segments.

Given a point-in-time or range query on a vertex or a subset of neighboring vertices (i.e., queries in the bottom two quadrants in Figure 1), we can observe that time-locality format usually works better. For example, to execute a query "find all neighbors of v_1 at t_3," first we read the checkpoint at the snapshot group beginning and then change the checkpoint by sequentially reading the updates until t_3. In this example, this results in two consecutive reads in time-locality order (assuming one graph edit activity costs

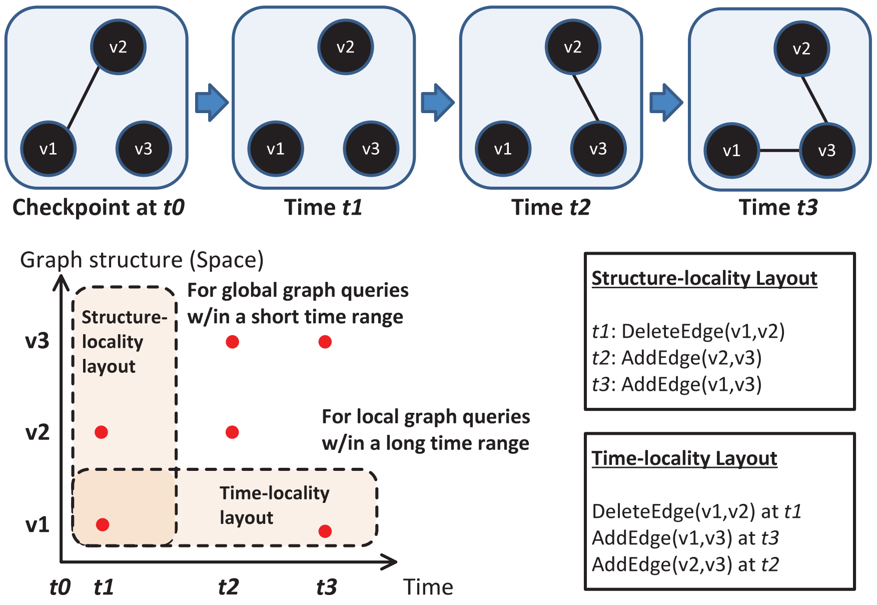


Fig. 2. Organizing Graph Updates.

one read). In contrast, structure-locality format would cause several seeks skipping vertices over different updates, such as with the three reads in this example. On the other hand, when the query spans a larger subset of vertices or the entire graph, we find structure-locality layout to work better, especially when the time range specified in the query is small.

Structure-locality or time-locality order is not optimal for all queries. Therefore, ImmortalGraph leverages replication that is necessary for both high availability and performance within a data center, keeping replicas optimized for different access patterns. As we keep data on replicas in different orders based on the query type within the four quadrants of Figure 1, ImmortalGraph automatically chooses appropriate replicas to improve access patterns.

2.2.3. Scheduling Temporal Iterative Workloads. Existing graph engines support static graph mining with a *scatter-gather* iterative computation model [Malewicz et al. 2010; Low et al. 2012; Gonzalez et al. 2012; Prabhakaran et al. 2012; Roy et al. 2013]. Such a model performs computation in multiple iterations. Each iteration includes a *scatter* phase that propagates a local value (e.g., a rank) associated with a vertex to its neighbors, followed by a *gather* phase that accumulates updates from neighbors to compute the local vertex's new value.

In order to understand graph evolutions over time (e.g., to understand how rank values of Web pages change over a period of time), users need to run scatter-gather iterative computation on a series of graph snapshots representing the states of a temporal graph at different points in time. Efficiently supporting such *temporal iterative graph mining* is one of the key design goals of ImmortalGraph.

At first glance, it might seem that the straightforward way of applying an existing scatter-gather graph engine on each snapshot, which we use as the baseline for our evaluations, would suffice. Our observation suggests that this straightforward solution is far from optimal and leaves significant room for improvement in performance.

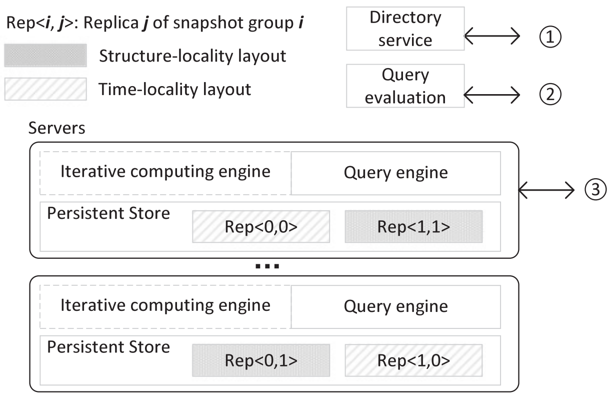


Fig. 3. Overview of ImmortalGraph architecture.

Unlike simple graph queries that usually access graph data once, iterative graph mining accesses a graph repeatedly during its computation. To support iterative graph computation over a series of snapshots, ImmortalGraph must address the following design issues.

First, it must decide on the in-memory layout of snapshot series. Our experiences with real-world large temporal graphs have shown that different in-memory data structures lead to great differences in computation performance (see Section 6.3). ImmortalGraph favors the time-locality layout over the structure-locality layout when performing iterative computation.

Second, ImmortalGraph must decide how to schedule computations that span multiple snapshots and, for each snapshot, involves propagation among neighbors within that snapshot. There are different ways of scheduling. One obvious choice is to schedule an iterative computation for each snapshot, while an alternative is to run iterative computation on multiple snapshots together. That is, propagations from one vertex to its neighbors on multiple snapshots can be scheduled together. ImmortalGraph adopts LABS(locality aware batch scheduling), which exploits the benefit of temporal graph layout by aligning scheduling mechanisms (Section 4).

Third, ImmortalGraph must enable parallelism of computations on multicore or distributed machines. One obvious option is to assign different snapshots to different cores; an alternative is to assign different graph partitions (aligned across multiple snapshots) to different cores. While the former strategy exhibits embarrassing parallelism since no synchronization is required in the computation, our results show that the latter, when carefully designed, can produce better performance.

Finally, ImmortalGraph should also consider the effect of incremental computation by continuously computing from the previous snapshot's result. Incremental computation is especially effective when the computation cost is proportional to the numbers of changes between two snapshots and significantly lower than that of computing from scratch.

These design choices have intrinsic dependencies among them, thus have to be considered together.

2.3. Architecture

Figure 3 shows the overview of the ImmortalGraph architecture. The round rectangles in the figure represent servers that store replicas of snapshot groups with structure-locality or time-locality layout. ImmortalGraph stores multiple snapshot groups consecutively in persistent storage that spans multiple servers. In Figure 3, a server contains replicas with different graph layouts serving different kinds of workloads.

ImmortalGraph serves simple queries that mostly access temporal graph data once. To execute a query, a client first consults a *directory service* to find the corresponding replicas of the snapshot group containing a specified time range (Step ①). The client evaluates the query through a *query evaluation service* to choose the replica with the proper graph layout (Step ②). The query engine on the desired replica finally serves the query (Step ③).

ImmortalGraph also supports more complicated iterative graph-mining workloads. A client runs such graph-mining algorithms using an *iterative computing engine* on the identified replica. ImmortalGraph supports all applications that are also targets of existing graph engines [Malewicz et al. 2010; Low et al. 2012; Gonzalez et al. 2012; Prabhakaran et al. 2012; Roy et al. 2013] with the addition of graph-mining capability in the time dimension. This includes graph mining at a certain point in time or within a time range.

3. TEMPORAL GRAPH DATA LAYOUT ON PERSISTENT STORAGE

A graph in ImmortalGraph consists of vertices and edges, with application-dependent data associated with each vertex or edge. For example, each vertex might be associated with a `rank` value or each edge associated with `type` or `weight` properties. ImmortalGraph models the "evolution" of a graph and treats a temporal graph as a series of *activities*; an activity involves the addition, deletion, and modification of vertices, edges, or their associated data at a particular point in time. For example, $\langle \mathsf{delV}, v_6, t_1\rangle$ is an activity that removes a vertex v_6 at time t_1, $\langle \mathsf{addE}, (v_6, v_1, w), t_2\rangle$ adds a new edge from v_6 to v_1 with a weight w at time t_2, while $\langle \mathsf{modE}, (v_6, v_1, w'), t_3\rangle$ modifies the weight associated with edge (v_6, v_1) to w' at time t_3.

ImmortalGraph organizes a temporal graph into snapshot groups, each responsible for a time interval. A snapshot group stores the snapshot of the graph at the start time t_s as a *checkpoint*, as well as all the activities that fall into this time interval. Depending on applications, a snapshot group is stored as edge files (for edge-related states and activities) and vertex files (for vertex-related ones). For example, there can be one vertex file for the rank values and others for other vertex-associated properties. In the rest of this section, we focus on the description of an edge file as edge/vertex files are treated the same way.

ImmortalGraph is designed for different classes of temporal graph applications, as described in Section 2. Some applications might require only the activities associated with a particular vertex v in a time interval, while others might be interested in retrieving an entire snapshot of a graph at a particular time t. Time-locality layout supports the former, while structure-locality layout supports the latter. These two layouts represent two different choices in ImmortalGraph along the two competing dimensions of a temporal graph, namely, the time dimension and the graph structure dimension. We elaborate on those two layouts next.

3.1. Time-Locality Graph Layout

Consider a temporal graph query to retrieve all activities associated with a vertex v that falls into the time interval of a snapshot group. Time-locality layout is designed for such a query as it groups together all activities associated with a vertex. Figure 4 illustrates the time-locality layout of an edge file. The file starts with an index to each vertex in this snapshot group, followed by a sequence of segments corresponding to vertices. The index allows ImmortalGraph to locate the starting point of a segment corresponding to a specific vertex without a sequential scan. A segment for a vertex v_0 consists of a checkpoint sector C_0 that includes edges associated with v_0 and

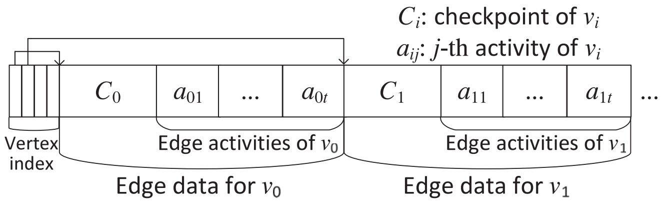


Fig. 4. The time-locality format.

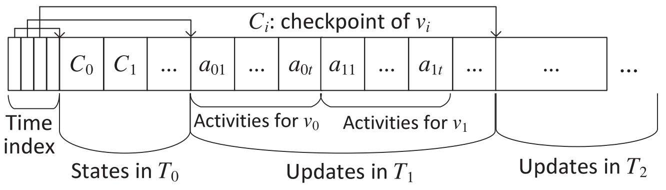


Fig. 5. The structure-locality format.

their properties at the start time of this snapshot group, which is followed by edge activities associated with v_0. For example, C_0 might contain information in the form of $(v_0, v_1, w_1), (v_0, v_5, w_2), \ldots, (v_0, v_n, w_m)$ that indicates that the graph snapshot at the start time of snapshot group contains edges (v_0, v_1) with weight w_1, (v_0, v_5) with weight w_2, (v_0, v_n) with weight w_m, and so on. The sequence $(a_{01}, a_{02}, \ldots, a_{0t})$ refers to edge activities related to v_0, sorted by the timestamps where $a_{01} = \langle \mathsf{addE}, (v_0, v_6, w), t_1\rangle$ adds a new edge (v_0, v_6) with weight w at time t_1, $a_{02} = \langle \mathsf{modE}, (v_0, v_1, w'), t_2\rangle$ changes the weight of edge (v_0, v_1) to w' at time t_2, and $a_{0t} = \langle \mathsf{delE}, (v_0, v_5), t\rangle$ removes edge (v_0, v_5) at time t.

3.2. Structure-Locality Graph Layout

Now consider a temporal graph query to retrieve an entire graph snapshot at a particular time. Executing such a query on time-locality layout is expensive because the information needed to construct an entire snapshot is scattered across the snapshot group. Even for reconstructing the snapshot at the start time of a snapshot group, all pieces of the checkpoint must be fetched, leading to a large number of random I/O operations or loading of unnecessary information. For such a query, structure-locality layout is more appropriate because it places the checkpoint corresponding to a graph snapshot together and organizes all activities of a temporal graph in each small time interval continuously.

Figure 5 shows the structure-locality graph layout in a snapshot group. Instead of having a vertex index, as in a time-locality layout, the structure-locality layout has a time index as the header, followed by a checkpoint that stores the graph snapshot at the start time of the snapshot group. The format of each checkpoint C_i is identical to that in the time-locality format. Structure-locality layout further divides the time range of the snapshot group into multiple smaller time intervals $T_i (1 \le i \le n)$. Within each T_i, edge activities for each vertex are stored continuously in timestamp order.

In the structure-locality layout, a query for the graph snapshot at the start time of the snapshot group, that is, the checkpoint, or for activities happening during a time interval like T_i, will involve many fewer I/O requests compared to the time-locality layout. A query for the graph snapshot at the time point close to the start time of the snapshot group can also benefit from the structure-locality layout that only fetches needed information, whereas in the time-locality layout, needed information is interleaved with unneeded information.

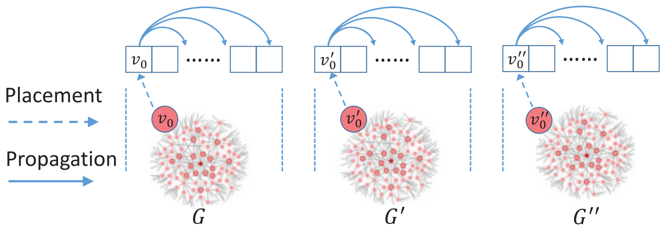


Fig. 6. Propagation pattern when data is grouped by snapshot.

3.3. Auxiliary Index and Link Structures

The time-locality and structure-locality layouts have two fundamental dimensions: while time-locality layout is organized as a series of vertex segments with a vertex index, structure-locality layout is organized as a series of time segments with a time index. The main index can be an array whose size equals the number of segments in the snapshot group, thus accessing a segment requires only $O(1)$ time complexity. A secondary index can also be built within each segment if needed, for example, when the segment is large. Our experiences indicate that it often makes sense to have a secondary vertex index for each time segment in structure-locality layout, choosing a reasonable representation depending on the number of different vertices with activities in a time segment. In practice, ImmortalGraph measures *sparsity ratio*, that is, the number of updated vertices over the total number of vertices. If the sparsity ratio in a time range T_i is less than $2/3$, the index array in T_i excludes all elements for the corresponding vertices that are not updated. In this case, index access to a certain vertex relies on binary search.

Queries on a temporal graph often involve identifying the state associated with a vertex at a particular time t. ImmortalGraph further introduces a link structure for each activity to link to the next activity associated with the same edge/vertex. In practice, ImmortalGraph adds a new field t_u in an activity; the value of this field is set to infinity if the activity is the last one in the snapshot group for that edge or vertex. To find the state at time t for a vertex v, ImmortalGraph scans activities in time order until it hits an activity at t_1 with its t_u field set to t_2, such that $t_1 \leq t < t_u = t_2$ holds, because this is the last activity for this vertex before or at time t.

4. ITERATIVE GRAPH MINING ON TEMPORAL GRAPHS

ImmortalGraph proposes *Locality-Aware Batch Scheduling* (LABS), which makes two fundamental design choices: one is to favor time locality over structure locality when laying out a temporal graph; the other is to match scheduled access pattern with data locality.

4.1. LABS Illustrated

Figures 6 and 7 illustrate the opportunity presented by LABS to achieve better data locality. The figures show a temporal graph with three graph snapshots, G, G', G'', representing graph states at three different time points. v_0, v_0', and v_0'' are three versions of the same vertex in the 3 snapshots, respectively (for simplicity, we use the same notations to denote the associated data of the vertex). The vertex has three neighbors that it needs to propagate its value to during graph computation.

Figure 6 shows a straightforward data organization for a temporal graph using snapshot by snapshot. This arrangement, at the expense of scattering different versions of data of the same vertex (say, v_0) away, attempts to place all vertices within the same snapshot close to each other. However, in a graph structure, it is difficult to ensure that

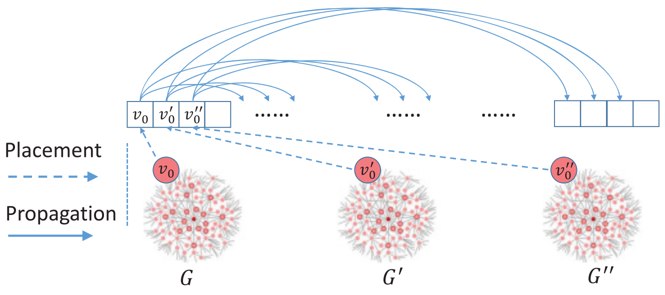


Fig. 7. Propagation pattern when data is grouped by vertex.

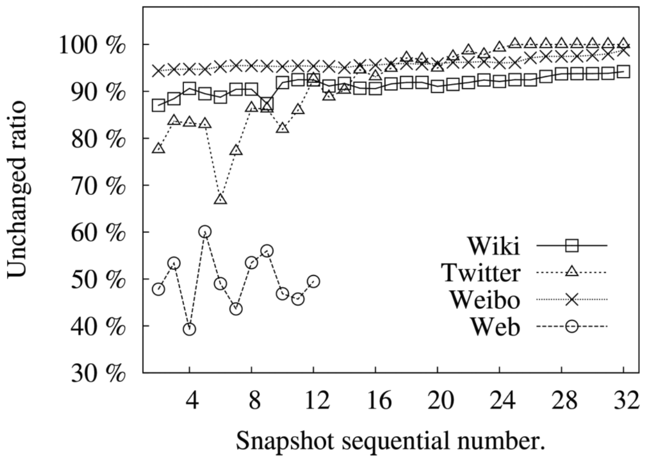


Fig. 8. Graph unchanged ratio over time.

all neighbors are placed close to every vertex. If all three neighbors of v_0 are scattered far from each other, running graph mining snapshot by snapshot would incur 9 cache misses for the propagations from v_0 to the 3 neighbors in the 3 snapshots within each iteration.

On the other hand, Figure 7 shows another data layout of the temporal graph that groups graph data by vertex. LABS adopts this layout to place different versions of the same vertex data together. Through batching the propagations to multiple versions of v_0's same neighbor, LABS can decrease the number of cache misses to 3 if the vertex data size is small enough to fit in the cache line.

In real-world graphs, it is very common to have a vertex's same neighbor in multiple snapshots, making it possible for benefit from LABS. Figure 8 shows the evolution statistics of 4 real-world graphs (more details about these graphs can be found in Section 6). The x-axis denotes the sequential number of snapshots (12 snapshots for Web as it is monthly data for one year, 32 snapshots for others) that are uniformly selected from each graph's entire time span. The y-axis is an *unchanged ratio*—the ratio of edges that each snapshot (except the first) "inherits" from its immediate predecessor snapshot. In Figure 8, the unchanged ratio of these graphs is above 50%. Moreover, the unchanged ratio can exceed 80% in a majority of cases. This indicates that the structures of real-world graphs are quite stable over time.

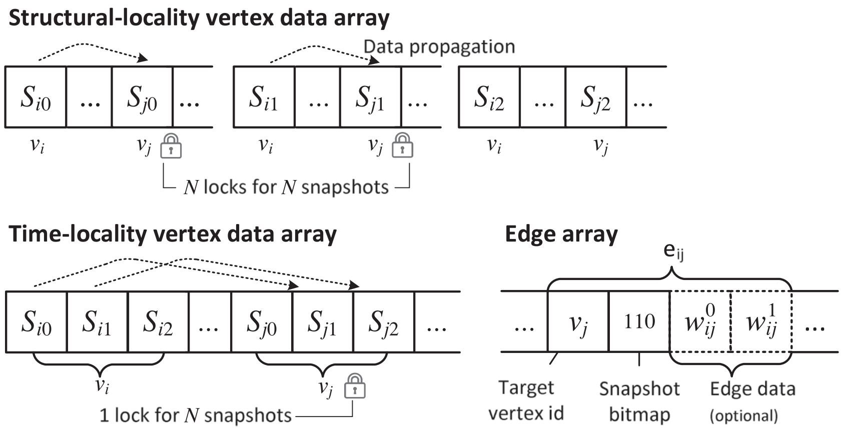

Structural-locality vertex data array
Data propagation
S_{i0} ... S_{j0} ...
S_{i1} ... S_{j1} ...
S_{i2} ... S_{j2} ...
v_i v_j
N locks for N snapshots
Time-locality vertex data array
S_{i0} S_{i1} S_{i2} ... S_{j0} S_{j1} S_{j2} ...
1 lock for N snapshots
Edge array
e_{ij}
... v_j 110 w_{ij}^0 w_{ij}^1 ...
Target vertex id
Snapshot bitmap
Edge data (optional)

Fig. 9. Illustration of time-locality and structure-locality in-memory layouts for vertices and edges.

In addition to reduced cache misses, LABS also decreases data access volumes and offers interesting design choices for parallel graph computation. The following sections describe details of ImmortalGraph's LABS design.

4.2. In-Memory Data Structure

At the beginning of computation, ImmortalGraph loads on-disk data that contains graph snapshots of interest into main memory for repeated access. The information is divided into an edge array and a vertex data array (for application data, such as ranks, associated with each vertex).

An in-memory data structure maintains the reconstructed states of the specified snapshots and discards any unnecessary fields (e.g., timestamps) stored in the on-disk layout. Our experience shows that in-memory computation often dominates the end-to-end cost of graph mining. We therefore focus on optimizing the in-memory data structure.

To lay out a temporal graph in memory, we have the choice of *structure-locality* to favor locality in *graph structure*, or *time-locality* layout to favor locality in *time dimension*.

As shown at the top of Figure 9, structure-locality layout groups graph information for the same snapshot together in a way that maximizes structure locality [Prabhakaran et al. 2012]. It places the data one snapshot after another to favor locality on the graph structure. Suppose a computation propagates S_{ik}, the state of vertex v_i in snapshot *k*, to S_{jk}, vertex v_j in the same snapshot (designated by the dashed arrow). This structure-locality layout tries to place S_{ik} close to S_{jk}.

Time-locality layout, instead, groups information for the same vertex across multiple snapshots and places the data one vertex after another. As shown in Figure 9, vertex data in the same snapshot is scattered compared with structure-locality layout. Yet, time-locality layout exhibits good data locality in the time dimension. In Figure 9, S_{i1}, the data of vertex v_i in snapshot 1, will *always* be placed next to S_{i0}, the corresponding data in snapshot 0.

In the edge array of time-locality layout, an edge is uniquely identified by the two vertices it connects and all edges are grouped by their source (or destination) vertices. Each edge array entry contains a vertex ID to index corresponding vertex data in the vertex array. It is also associated with a snapshot bitmap that specifies snapshots containing the edge. For example, the bottom-right of Figure 9 shows an element in the edge array that represents an edge e_{ij} from vertex v_i to v_j. The value of the snapshot bitmap is 110, indicating that the edge exists in snapshots 0 and 1, but not 2. The snapshot bitmap saves memory and provides an efficient way to check whether or not a snapshot contains an edge. Edge e_{ij} may have associated data (e.g., edge weight) in the corresponding snapshots designated by the snapshot bitmap. For example, w_{ij}^0 and w_{ij}^1 in Figure 9 denote associated data of e_{ij} in snapshot 0 and 1.

Although data locality on the graph structure can be captured by carefully placing vertices in a static graph snapshot [Prabhakaran et al. 2012], the real performance gain of this scheme relies heavily on the actual graph structure. After all, the graph structure is not linear so it is hard to place in a linear address space with good locality. In contrast, we have found it easier to exploit data locality in the time dimension because data access patterns in this dimension are often linear. ImmortalGraph therefore favors time-locality layout, coupled with locality-aware batching scheduling, which we describe in the next section.

4.3. Locality-Aware Batch Scheduling

ImmortalGraph argues for a scheduling mechanism that combines the considerations in both the execution of operations and the underlying data layout strategy. The scheduling should align data access patterns with the underlying data layout to achieve better locality. For example, if in-memory layout places states associated with a vertex across multiple snapshots together in a time-locality layout, it would be ideal to schedule computation that operates on those states together. If an in-memory layout places a vertex close to its neighbors in the same snapshot of a structure-locality layout, it would be ideal to schedule computation that operates on those vertices together.

Instead of doing computation snapshot by snapshot on a structure-locality layout, LABS aligns data access patterns in computation with a time-locality layout. To do this, LABS *batches* processing on each vertex across all snapshots. Similarly, for each edge of a vertex, LABS performs propagations to a neighboring vertex for all snapshots in a batch. Because vertex data for all the snapshots are placed contiguously in the time-locality layout, LABS can therefore exploit excellent data locality in the time dimension.

Besides data locality, batched scheduling can also save edge array enumerating time. LABS enumerates the edge array only *once* for processing all snapshots; otherwise, snapshot-by-snapshot scheduling would involve one enumerating the edge array for each snapshot. This reduces memory accesses for LABS.

4.4. Preparing the In-Memory Layout for ImmortalGraph

ImmortalGraph provides an interface to load a series of snapshots from one or more snapshot groups into memory. The implementation of this interface loads the time-locality on-disk layout to reconstruct the states of the specified snapshots through a sequential scan on the snapshot groups. The on-disk time-locality layout is convenient as it matches the time-locality in-memory layout for ImmortalGraph. It is worth noting that the on-disk image contains all the information related to a temporal graph and records the activities faithfully, while the in-memory layout is optimized particularly for the graph mining in which it stores the reconstructed *states* rather than update activities (delta), and that it is in a compact format with only the necessary information.

In our experience, when loading the on-disk temporal graph and reconstructing the snapshots, ImmortalGraph can always saturate the bandwidth of the disk (even the SSD hard drive). In our experiments, the cost of loading on-disk data is often a small fraction of the end-to-end graph computation time.

4.5. Parallel Processing with LABS

LABS can use multiple cores to parallelize graph mining on a series of snapshots. There are two design choices for parallelization. We can either assign each snapshot to a CPU core, which we call *snapshot-parallelism*, or partition each snapshot by vertices and assign each partition to a core, which we call *partition-parallelism*.

For example, assume that we have two CPU cores c_0 and c_1, and two snapshots S_0 and S_1. The snapshots can be partitioned into two parts, P_{00} and P_{01} for S_0, and P_{10} and P_{11} for S_1. The snapshots are partitioned in a consistent way such that a vertex that exists on both snapshots is assigned to the partition with the same ID. In snapshot-parallelism, we assign S_0 and S_1 to c_0 and c_1, respectively, and let the two cores run concurrently. In partition-parallelism, we assign $\{P_{00}, P_{10}\}$ to c_0 and $\{P_{01}, P_{11}\}$ to c_1.

There are interesting trade-offs between different types of parallelization strategies. Snapshot-parallelism does not involve synchronization among cores because computations on different snapshots are independent. However, the strategy is fundamentally incompatible with LABS. In contrast, partition-parallelism incurs the overhead of intercore communication. It requires locks to protect concurrent vertex data propagation due to the cross-partition edges. Nevertheless, partition-parallelism can be enhanced with LABS for better locality. Meanwhile, with LABS, the intercore synchronization cost of partition-parallelism can be significantly reduced because the lock on a vertex and the propagation through an edge can be performed in a batch for multiple snapshots. Specifically, assume that (v_i, v_j) shown in Figure 9 is a cross-partition edge and it exists in N snapshots (2 in this case). Without LABS, the propagations along the edge for N snapshots might involve N rounds of intercore communications and N times of locking on the destination vertex (i.e., N locks for N snapshots), while with LABS this introduces only one intercore communication and one locking for all N snapshots (i.e., 1 lock for N snapshots). Although the 1-lock-for-N-snapshot looks to introduce larger critical sections and decrease concurrency, the fact that the batched propagation along an edge for multiple snapshots incurs a similar amount of intercore communication to the propagation for one snapshot makes them similarly fast. Our evaluation results demonstrate that, when integrated with LABS, partition-parallelism can be significantly more efficient than snapshot-parallelism.

4.6. Incremental Computation with LABS

Incremental computation is another effective approach to optimizing graph computation on a series of snapshots [Cheng et al. 2012]. For example, we compute the single-source shortest path (SSSP) on a graph snapshot S_0 with vertex v_0 as the source vertex. After the computation, each vertex has a computed associated data representing the distance between the vertex and v_0. When we perform the same computation on snapshot S_1, we use the computed results on S_0 as the initial value for the current computation on S_1. The convergence of the computation can be much faster when the distances between v_0 and most of the vertices do not change in S_1 compared to those in S_0.

However, incremental computation has its limitations. For example, some incremental SSSP algorithms are designed to handle edge insertion only (or edge removal only) [Roditty and Zwick 2004]. Moreover, reusing results in the previous snapshot does not always reduce computation time. In an extreme case, one edge removal near v_0 might make it disconnected from the main part of the graph. In this case, most vertices in the graph have to recompute and update their distances to v_0.

ImmortalGraph enhances incremental computation in two significant ways. First, to compute N snapshots from S_0 to S_{N-1}, ImmortalGraph first computes the results for snapshot S_0. It then computes the rest of $N-1$ snapshots (S_1 to S_{N-1}) in a batch using

LABS. In the batch processing, ImmortalGraph uses the results computed in snapshot S_0 as the initial value for the $N-1$ subsequent snapshots, thereby enabling incremental computation. While the total amount of computation in this way might be higher than a pure incremental computation approach (which computes on the snapshots in serial order), our approach does benefit from better locality as well as the reduced number of accesses to the edge array.

Second, because the snapshots are known in advance, for a group of N snapshots, ImmortalGraph can precompute the *intersection* (or the *union*) of these N snapshots, so that each true snapshot simply adds (or removes) edges/vertices to that intersection (or union) graph to allow incremental computation even when the algorithm only supports edge/vertex insertion (or removal). This enlarges the scope in which incremental computation is applicable. For example, consider an initial graph snapshot $S_0 = (V_0, E_0)$. The next snapshot S_1 might have removed edge e_1, while adding many other edges. The initial graph G_0 for incremental computation can be the intersection of S_0 and S_1, which would be $(V_0, E_0 - \{e_1\})$. Thus S_0 and S_1 can be constructed from G_0 by adding edges only.

4.7. Modes of Iterative Graph Computation Model

As mentioned in Section 2.2.3, most existing graph engines support the *scatter-gather* iterative graph computation model. The model can be implemented in different modes with subtle differences that could have deep implications for performance. In this section, we introduce different ways to implement the scatter-gather model and note that ImmortalGraph—including the key design elements, time-locality layout and LABS—is fully compatible with these implementations.

The scatter-gather model can be implemented in a *vertex-centric* way [Malewicz et al. 2010; Low et al. 2012; Gonzalez et al. 2012; Prabhakaran et al. 2012] or *edge-centric* [Roy et al. 2013] way. A vertex-centric graph engine iterates over vertices; it requires that users provide a `scatter` function and a `gather` function for each vertex. The `scatter` function specifies the behavior of each vertex where it computes and propagates (scatters) the local value to its neighbors; the `gather` function dictates the vertex behavior when it gathers updates from its neighbors. A vertex-centric engine uses two versions of the vertex data array (see Figure 9 in Section 4.2) during the computation: one stores the value computed in the previous iteration, and the other keeps the most updated value computed in the current iteration. The two vertex data arrays switch roles after each iteration.

An edge-centric engine like X-Stream [Roy et al. 2013] iterates over edges rather than vertices. It requires not a vertex-associated function but, for each edge, an `edge_scatter` function and an `edge_gather` function that describes what value needs to be propagated through the edge and where the value associated with the edge needs to be applied, respectively. In the scatter phase, the edge-centric engine scans the edge array and writes the data computed from source vertices to an update array *sequentially*. In the gather phase, it *sequentially* reads the computed data stored in the update array and applies it to the destination vertices. In order to make scatter and gather operations as sequential as possible, the edge-centric engine introduces an additional *shuffle* phase between the scatter and gather phases to partition the update array by the destination vertex. The similar shuffle operation is also done on the edge array by the source vertex, before the computation starts. This way, the engine can mitigate the random data accesses by streaming the updates with edges into and out of sequential buffers (i.e., update edge array). This graph-computation mode maximizes the chances of sequential data processing and is referred to as *stream* mode.

A vertex-centric engine can operate in *push* mode or *pull* mode. In push mode [Malewicz et al. 2010; Prabhakaran et al. 2012], the engine checks the change

```
// Query vertex data and the changes
Vertex QueryVertex(Timestamp, VertexId);
VertexChangeIterator QueryVertexChanges(TimeRange, VertexId);

// Query edge and edge changes
EdgeIterator QueryEdge(Timestamp, VertexId);
EdgeChangeIterator QueryEdgeChanges(TimeRange, VertexId);
```

Fig. 10. Query interface.

of the vertex state from the previous iteration. It sends the value to all neighbors for updating only when there is a change that is significant enough. In pull mode [Low et al. 2012; Cheng et al. 2012; Shun and Blelloch 2013], the engine collects the states of the neighbors of a vertex by pulling, rather than pushing the changes. Although seemingly similar, push and pull modes have important differences that have significant performance implications. For example, consider the execution on a single multicore machine, where the computation on each vertex can be executed concurrently. In push mode, a vertex needs to use a lock to protect the process of updating the value of a neighbor because multiple vertices (with an outgoing edge to the same vertex) might be updating the state concurrently. In contrast, in pull mode, no locks are needed because, in each iteration, a vertex needs to read only the value in the previous iteration from its neighbors. This value is stored in the aforementioned two-version vertex data array and is immutable during the current iteration. Moreover, in pull mode, a vertex is the only entity reading and updating its own state in the current iteration. Although no lock is required, pull mode needs to pay the cost of checking the significance of the value changes of the neighbors. This checking on a neighboring vertex needs to be performed multiple times if different vertices share a neighbor, which is a significant overhead that sometimes overweights the saving of locks (see details in Section 6).

Regardless of the different implementations, being vertex-centric or edge-centric model, push, pull, or stream mode, ImmortalGraph together with LABS is beneficial to various graph-engine implementations. We have implemented ImmortalGraph with all the previously described implementations. We demonstrate the effectiveness of all the implementations and explain the reasons in detail in Section 6.

4.8. ImmortalGraph Graph Engine in a Distributed Setting

Although our experiences with real-world large temporal graphs have shown that multiple snapshots can usually fit in memory on a powerful multicore machine, we have extended ImmortalGraph to a distributed environment, in which a series of snapshots are partitioned and assigned to different machines, much like the way they are partitioned and assigned to different cores on a multicore machine. This allows ImmortalGraph to handle even larger temporal graphs when needed.

5. PROGRAMMING WITH IMMORTALGRAPH

ImmortalGraph supports two types of programming interfaces. The first is a set of low-level *query* interfaces (see Figure 10), which is used in simple time-aware graph traversals. For example, `QueryEdge` and `QueryVertex` can be used in multihop neighborhood query, random walk, and breadth-first search (BFS) given a time instance. `QueryEdgeChanges` or `QueryVertexChanges` can be used to query the activities of a certain vertex or its associated edges within a time range.

The second type is a set of higher-level interfaces used for time-aware iterative graph computation (see Figure 11). These interfaces are user-defined functions invoked by the underlying iterative computing engine at certain execution steps. They are essentially

```
void PreIteration(Iteration&, Vertex&);
void VertexFunc(Iteration&, Vertex&);
void EdgeFunc(Iteration&, Edge&);
void EdgeChangeFunc(Iteration&, Edge&);
void SetTimeRange(TimeRange);
void VoteToHalt(VertexId, SnapshotId = ALL);
```

Fig. 11. Iterative computing interface.

the temporal extension on the vertex-centric programming model used by existing graph engines [Malewicz et al. 2010; Low et al. 2012; Cheng et al. 2012; Gonzalez et al. 2012]. Iterative graph computations consist of multiple iterations. `PreIteration` is a vertex-based interface invoked at the beginning of each iteration. It is usually used for data preparation before each iteration or testing the termination condition. In an iteration, `VertexFunc` performs any vertex-associated operation. `EdgeFunc` performs edge-associated actions, which usually involve propagating values calculated in `VertexFunc` to neighbors. The executions of `PreIteration` are in parallel across vertices, as are the executions of `VertexFunc`. An optional global barrier is introduced between the execution of the two interfaces to satisfy the need of both synchronous and asynchronous graph algorithms. `EdgeFunc` is called right after the corresponding `VertexFunc` without a barrier. `EdgeFunc` across vertices are executed in parallel, hence the destination vertex is protected by a lock in the data propagation. We also introduce `EdgeChangeFunc`, which enumerates edge changes within a time range given by `SetTimeRange`. Finally, a vertex calls `VoteToHalt` when it finishes the computation in a certain snapshot. The entire computation ends when all vertices have called `VoteToHalt` in all snapshots involved in the computation.

Next, we explain how these interfaces can be used to implement various temporal graph applications.

Figure 12 illustrates a simplified pseudo-code that computes PageRank on a series of given graph snapshots. Here, we use push mode as an example; other modes of iterative graph computation model could be written similarly. The implementation uses LABS mechanism. In the code, `prevRanks`, `currRanks`, `subRanks`, `degrees` are global vertex data arrays with the in-memory layout described in Section 4. `currRanks` stores the initial rank value for each vertex (code omitted).

In `PreIteration`, each vertex checks the results in the previous iteration to see whether to `VoteToHalt` in a certain snapshot. To prepare for the next iteration, it resets `currRanks` after swapping values in `currRanks` to `prevRanks`. `VertexFunc` calculates the `subRank` that each vertex will propagate to its neighbors. `EdgeFunc` propagates the value to each neighbor.

Within each iteration, the rank values of a vertex across all snapshots are calculated and propagated in a batch using a `foreach(snapshot)` loop. This exploits locality in the involved vertex data arrays, that is, LABS.

To reduce the burden of the users to implement such a program, ImmortalGraph offers another set of interface (with the suffix "LABS"). The user can implement with it just like for an algorithm on a static graph snapshot, as shown in Figure 13. The LABS mechanism is applied by the underlying system automatically, while the interfaces listed in Figure 11 enable users to access the data of multiple snapshots within one interface, making it possible to implement temporal graph-specific applications such as interaction graph and centrality of dynamic networks, as introduced later in this section.

Interaction graphs. Wilson et al. [2009] argue that *interaction graphs* characterize social relationships more accurately than the traditional "snapshot" of online social

```
void PreIteration(Iteration& itr, Vertex& tv) {
  for (snapshot_t sId = 0; sId < tv.numSnapshots; sId++) {
  // Test termination condition
    if (|currRanks[tv.id, sId] - prevRanks[tv.id, sId]| < EPSILON)
        VoteToHalt(tv.id, sId); //snapshot-aware VoteToHalt()

    prevRanks[tv.id, sId] = currRanks[tv.id, sId];
    currRanks[tv.id, sId] = 0;
  }
}

void VertexFunc(Iteration& itr, Vertex& tv) {
  for(snapshot_t sId = 0; sId < tv.numSnapshots; sId++)
    subRanks[tv.id, sId] = DAMPING_FACTOR * \
        prevRanks[tv.id, sId] / degrees[tv.id, sId];
}

void EdgeFunc(Iteration& itr, Edge& te) {
  for (snapshot_t sId = 0; sId < te.numSnapshots; sId++) {
    // Propagate if te exists in the snapshot sId
    if (te.edgeExist(sId))
      currRanks[te.dst, sId] += subRanks[te.src, sId];
  }
}
```

Fig. 12. Explicit LABS-enabled PageRank pseudo-code.

```
void PreIteration_LABS(Iteration& itr, Vertex& tv) {
  // Test termination condition
  if (|currRanks_LABS[tv.id] - prevRanks_LABS[tv.id]| < EPSILON)
      VoteToHalt_LABS(tv.id);

  prevRanks_LABS[tv.id] = currRanks_LABS[tv.id];
  currRanks_LABS[tv.id] = 0;
}

void VertexFunc_LABS(Iteration& itr, Vertex& tv) {
  subRanks_LABS[tv.id] = DAMPING_FACTOR * \
      prevRanks_LABS[tv.id] / degrees_LABS[tv.id];
}

void EdgeFunc_LABS(Iteration& itr, Edge& te) {
  // Propagate if te exists in the this snapshot
  if (te.edgeExist_LABS())
    currRanks_LABS[te.dst] += subRanks_LABS[te.src];
}
```

Fig. 13. Implicit LABS-enabled PageRank pseudo-code.

```
void PreIteration(Iteration& itr, Vertex& tv) {
  // Terminate after a certain number of iteration
  if (itr.num >= tv.numSnapshots) {VoteToHalt(tv.id);}
  if (tv.id == 0) {
    snapshot_t sId = tv.numSnapshots - itr.num;
    SetTimeRange(TimeRange(times[sId-1], times[sId]));
  }
  if (itr.num > 0) {
    centralities[tv.id] += localInfls[tv.id];
    // Temporal propagation
    globalInfls[tv.id] = localInfls[tv.id] + GAMMA * globalInfls[tv.id];
  }
  localInfls[tv.id] = 0;
}
void EdgeChangeFunc(Iteration& itr, Edge& te) {
  //Reverse spatial propagation. D_INFL: direct influence
  localInfls[te.src] += D_INFL + ALPHA * globalInfls[te.dst];
}
```

Fig. 14. Pseudo-code for computing centrality for dynamic networks.

networks, such as Facebook. An interaction graph is a temporal graph specified by two parameters n and t, where n is the number of interactions between two users and t denotes time range during which the interaction happens. Given n and t, the computation of an interaction graph accesses the global graph with a localized time range once (i.e., a global-range query). We do not show the code of the interaction graph due to its simplicity.

Centrality metric for dynamic networks. Lerman et al. [2010] propose a centrality metric that evaluates the importance of a vertex in a temporal graph. According to this method, an edge activity $\langle (v_i, v_j), t\rangle$ from vertex v_i to v_j happening at time t has the probability of further influencing the *downstream* vertices of v_j at time $t + \delta$ with a spatial damping factor α and temporal damping factor γ. A vertex v_k is a downstream vertex of v_i if it has at least one *time-dependent* path $TPath_{(i,k)}[t, t + \delta]$, where $TPath$ refers to the path that exists from v_i to v_k in the time period $[t, t + \delta]$. At time $t + \delta$, downstream vertex v_k of v_i receives the amount of influence $\alpha^{(n-1)} \times \gamma^{\delta}$ if v_k is n-hop away in $TPath_{(i,k)}[t, t + \delta]$. The centrality of v_i is defined as the total amount of influence v_i imposes on all its downstream vertices over a period of time. Similar to the interaction graph, the application accesses the whole graph within a given time range once (i.e., also a global-range query).

Figure 14 illustrates how ImmortalGraph calculates the centrality in a given time interval. The computation uses the interface `EdgeChangeFunc` since it focuses on the change (activity) instead of the snapshot of an edge.

The computation iteratively aggregates (dampened) influences through multiple subtime intervals given by array `times`. Each iteration is responsible for the influence aggregation in a subinterval. `EdgeChangeFunc` gathers local influences of edge activities to the array `localInfls` (the term *local influence* refers to the gathering within the neighborhood). It essentially performs a (reverse) spatial propagation with a damping factor α. The retrieved edge activities are within the time range instructed by `SetTimeRange` called in `PreIteration`. In the beginning of each iteration, each vertex first adds the local influence gathered in the previous iteration to array `centralities` and performs a temporal propagation, that is, the result in the previous subtime interval has to be dampened by a factor of γ for the current time interval and stores in array `globalInfls`

Table I. Temporal Graph Statistics

Graph	# of vertices	# of edge activities	Time span
Wiki	1.871 M	39.953 M	6y
Twitter	7.512 M	61.633 M	3mo
Weibo	27.707 M	4.900 B	3y
Web	133.633 M	5.508 B	12mo

Note: M: million, B: billion, y: year, mo: month.

that is about to be further propagated in the next iteration. The final results are in array `centralities`.

With the interfaces described in this section, ImmortalGraph supports various temporal graph computations, ranging from simple graph queries with temporal extension to sophisticated algorithms that only work on a temporal graph.

6. EVALUATION

We evaluate ImmortalGraph both on a commodity multicore machine and on a small distributed testbed. The multicore machine is equipped with dual 2.4 GHz Intel Xeon E5-2665 processors (16 cores in total), 128 GB of memory. The distributed testbed consists of 4 multicore servers with the same configuration and interconnects with InfiniBand (double data rate, 40 Gbps). We use the following 4 real-world temporal graphs in the experiments.

Wikipedia reference graph (Wiki): This is a temporal graph of English Wikipedia Web pages. Each Wiki page is a vertex. An edge activity $\langle(m, n), t\rangle$ represents a hyperlink from page m to n created at time t [Mislove 2009]. The Wiki graph consists of 1.87 million vertices and 40 million hyperlinks. It shows how the English Wiki network evolved over a period of 6 years.

Time-evolving Web graph (Web): The graph is similar to the Wiki graph except that each vertex now is a Web page in the .uk domain, and an edge denotes a link between pages at a certain time instance. The temporal Web graph includes 12 monthly snapshots in the .uk domain [Boldi et al. 2008]. Each edge activity is associated with the creation or removal time observed in the snapshot. In total, the Web graph contains more than 133 million vertices and 5 billion edges.

Twitter mention graph (Twitter): In Twitter, a tweet containing a string like "@tom" means that the publisher mentions user "tom." In a Twitter mention graph, a user is a vertex. An edge activity $\langle(m, n), t\rangle$ in the mention graph means that user m mentioned n in a tweet at time t. The mention graph indicates the amount of attention each user pays to others and how the attention changes over time [Romero et al. 2011]. We have collected more than 102 million tweets over 3mo, from which we derived a temporal graph with around 7 million vertices and 61 million edge updates (events).

Weibo mention graph (Weibo): Weibo [Sina 2013] is the Chinese counterpart of Twitter. The meaning of the Weibo mention graph is the same as that of Twitter. We have collected more than 7 billion Weibo microblogs over 3y. The derived temporal graph contains around 28 million vertices and 4.9 billion edge updates.

The size and the time span of each temporal graph are summarized in Table I.

For parallel/distributed graph computation, we partition the graphs using Metis [Karypis and Kumar 1995], a public implementation of a multilevel k-way graph partitioning algorithm. Metis is believed to be an effective partitioning method for a general graph to minimize cross-partition edges and balance among partitions. Within each partition, we use spectral placement to order the vertices in the graph layout for better locality on the graph-structure dimension [Prabhakaran et al. 2012]. Note that it is believed that to partition the graph by *edge*, which may cut a single vertex into multiple replicas across partitions, is an effective way to partition graphs exhibiting the power law [Gonzalez et al. 2012; Chen et al. 2013]. All these partitioning techniques are compatible and complementary to ImmortalGraph. A particular graph partitioning technique alone does not affect the performance advantage of LABS if under the same configuration.

6.1. Evaluation Methodology

Our experiment goals include: (1) to validate the effectiveness of time-locality and structure-locality layouts for various temporal graph queries; (2) to demonstrate the benefit of the LABS mechanism for iterative computations in multicore environments, including incremental computing on temporal graphs; and (3) to verify that ImmortalGraph outperforms existing database systems for temporal graph queries.

For the first goal, we select temporal graph queries that cover all 4 quadrants in Figure 1. We name the 4 quadrants global-point, global-range, local-point, and local-range, respectively. The queries usually access the on-disk graph data once. We clear the buffer cache before each experiment.

For the second goal, we conduct the experiments using 5 selected graph applications including PageRank [Brin and Page 1998], weakly connected component (WCC), single-source shortest path (SSSP), maximal independent set (MIS), and a general abstraction of vertex-centric graph computations: sparse matrix-vector multiplication (SpMV) [Gonzalez et al. 2012]. Each is computed on a series of snapshots of different temporal graphs.

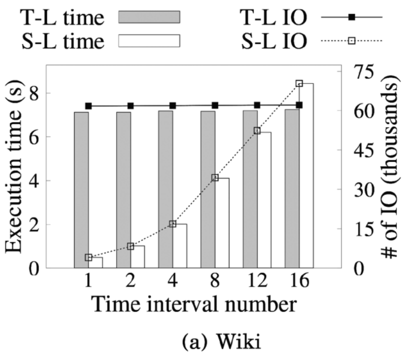
For the last goal, we compare ImmortalGraph with PostgreSQL [PostgreSQL 2013], an open-source database that supports SQL temporal extension. We store a temporal graph as an edge table. Each row in the table represents an edge activity with the schema as $\langle s, d, t_1, t_2 \rangle$, where s and d refer to the source and destination vertex ids, t_1 is the timestamp of the activity, and t_2 is the time when the immediate next edge activity between the vertices happened.

PostgreSQL supports temporal queries mainly through a built-in B-tree index on timestamps. A Time-Split B-tree (TSB tree) would provide superior performance for temporal data access [Lomet and Salzberg 1989]. Since no implementation of the TSB tree–based databases such as ImmortalDB is openly available [Lomet et al. 2005], we therefore implement our own TSB-tree structure to store the temporal graph for a fair comparison. In the TSB-tree structure, we also store each edge activity in a record with $\langle s, d \rangle$ as the key. s and d have the same meaning as in the schema used in PostgreSQL.

6.2. Effectiveness of Graph Layouts

Global-point query. A global point query retrieves the complete snapshot of a graph at a given time point. It gets the data of all involved vertices and edges by scanning the index structures in the edge file to locate and access edges of each vertex. Figure 15 shows the performance of a global-point query under different data layouts for Wiki and Weibo graphs. The results for the remaining temporal graph are similar, thus are omitted. In Figure 15, the primary (left) y-axis is the total execution time, and the secondary (right) y-axis shows the I/O number involved in the query. The x-axis denotes the time point offset relative to the begin time of the snapshot group. Since different temporal graphs have different time scales, the value in the x-axis is the number of time intervals from the begin time. This time interval is the same as the one used in the structure-locality layout.

Figure 15 shows that, when the given time point is closer to the begin time, structure-locality outperforms time-locality. This is because, in the time-locality format, all temporal data for a vertex is placed together. Hence a data block may contain too much temporal information beyond the given time point and is therefore useless to the query.



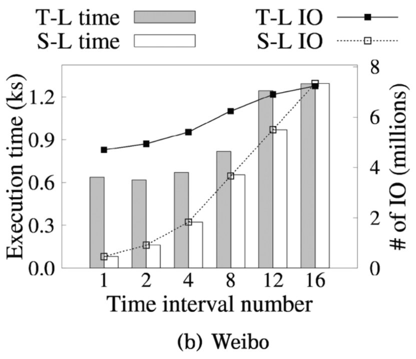
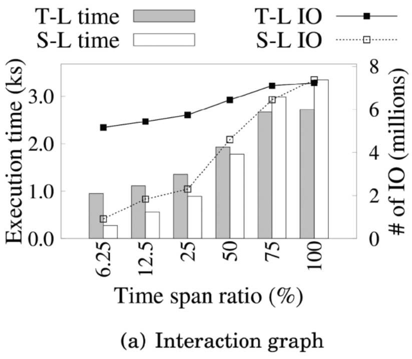


Fig. 15. Performance of a global-point query at different time points. T-L refers to time-locality and S-L refers to structure-locality. The same notation applies to Figures 16, 17, and 18.



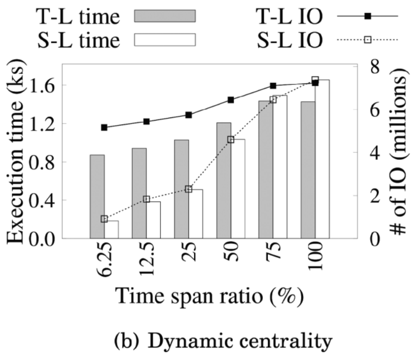


Fig. 16. Performance of global-range queries on Weibo graph.

The closer the time point is to the begin time, the more likely the data block contains useless temporal data in the time-locality layout.

When given at a later time point, the performance difference between the structure-locality and time-locality layouts decreases since the temporal data in a block for the time-locality layout becomes more useful for the queries. When the given time point is closer to the end time of the snapshot group, the time-locality format can even outperform structure-locality. This is because the structure-locality layout has a secondary index structure for each time interval, and access to these indices introduces extra I/O overhead. The amount of I/O shown in the secondary y-axis in Figure 15 is consistent with the execution time.

Therefore, for a global-point query, ImmortalGraph can choose appropriate data layouts according to the distance between the given time point and the begin time of the snapshot group: when the given time point is very close to the end time, choose the time-locality layout; otherwise, it is best to choose the structure-locality layout.

Global-range query. We use interaction graph [Wilson et al. 2009] and dynamic centrality [Lerman et al. 2010] as examples of global-range queries. Although the two applications are implemented using the iterative computing interface, their access patterns are more similar to simple queries, which access graph data once or very few times rather than repeatedly. We show only the results on the Weibo graph as the performance trends on the other graphs are similar.

Figure 16 shows the results. In the figure, the y-axes are the same as the ones in Figure 15. The x-axis shows different time intervals, which range from 1/16 (6.25%) of

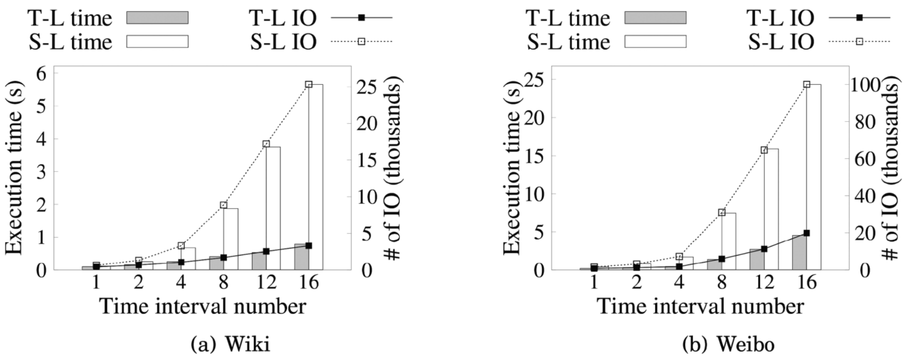


Fig. 17. Performance of 2-hop query with given time point for 100 random vertices.

the entire snapshot time range to 100% of the time range with a step size setting to 1/16. The center point in each time range locates in the middle of the entire time range of the snapshot group.

The trend of the results is similar to the one in global-point queries. When the specified time range is small, the structure-locality layout is better since it only involves I/O for the data that are mostly useful due to good data locality (already divided by smaller time intervals). When the time range becomes larger and close to the entire time range of the snapshot group, the structure-locality layout suffers from the extra I/O cost from the index structure accesses, hence the time-locality layout outperforms.

For global-range queries, ImmortalGraph chooses the proper layout according to the length of the specified time range. If the time range is close to the entire time range of the snapshot group, it chooses the time-locality layout; otherwise, it chooses the structure-locality layout.

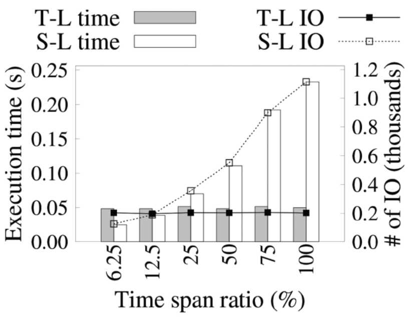
Local-point query. The chosen example for a local-point query is a two-hop neighborhood query for a given vertex in the graph snapshot at a given time point. Figure 17 shows the average performance of local point queries for 100 randomly selected vertices with varying time points. The meaning of x-axis and the y-axes is the same as in Figure 15.

As the figure shows, when the given time point is close to the begin time of the snapshot group, the performance of the structure-locality and time-locality layouts are comparable. When given at later time points, time-locality outperforms structure-locality more and more significantly. This demonstrates the benefit of the good data locality provided by the time-locality layout for local-point queries, especially for cases when the given time point is far from the begin time of the snapshot group.

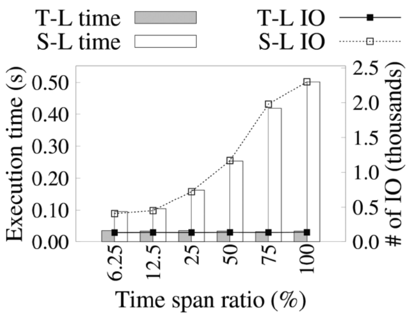
Local-range query. The example of local-range query that we use is to query for the edge activities of a given vertex within a given time range. The selection of the time range for each query is the same as in previous global-range query experiments.

Figure 18 shows the average running time of local-range queries for 100 randomly chosen vertices. Because the system behavior under local-range query is very similar to the local-point query, the two data layouts show a similar performance trend, as in the local-point query case. Therefore, ImmortalGraph would also use the time-locality format for local-range query.

Note that, when the specified time range is very small (e.g., smaller than one subtime interval in the structure-locality layout), the time-locality layout can perform worse than structure-locality (see Figure 18(a)). This is because, in this case, a data block in the time-locality layout could contain temporal information (of a vertex) beyond the given smaller time range, while for the structure-locality layout, it is possible that the



(a) Wiki



(b) Weibo

Fig. 18. Performance of edge activity queries for 100 random vertices.

Table II. CPU Cache Miss Counts in Three Modes for MIS on Wiki Graph (in millions)

	Push mode			Pull mode			Stream mode		
Batch size	L1d	LLC	dTLB	L1d	LLC	dTLB	L1d	LLC	dTLB
1	8,759	649	3,462	6,470	859	3,419	4,091	1,090	79
4	3,865	584	1,003	2,638	753	839	1,290	274	23
16	1,107	265	287	926	365	230	493	95	10
32	687	196	160	635	272	126	386	62	9

Note: L1d = level 1 data cache; LLC = last level cache; dTLB = data translation lookaside buffer.

information of multiple vertices in a data block can be touched and could all be useful for the given time range.

6.3. Effectiveness of LABS

To demonstrate the advantage of the proposed temporal graph layout and LABS, we first study the performance of ImmortalGraph in the single-thread case. We use the straightforward approach of running graph computation on each snapshot one by one as the baseline for our comparisons. To generate N snapshots for our experiments, we equally divide the second half of the entire time range by N to have snapshots covering nonoverlapping time ranges. The first snapshot is chosen in the middle of the entire time range to generate a graph large enough that is meaningful for large-scale graph computation. An important parameter for LABS is its *batch size*, which is the number of snapshots that are batched together for iterative computation in LABS. Note that our baseline is essentially the execution with the batch size set to 1.

Figure 19 shows the performance of ImmortalGraph compared to our baseline, for different applications on the Wiki, Twitter, and Weibo graphs. We compare the performance in all three processing modes: push, pull, and stream. As the figures show, ImmortalGraph outperforms our baseline consistently across all applications and in all three modes. The gain becomes more significant when the batch size increases. When the batch size is 32, ImmortalGraph runs more than 20 times faster for SSSP on the Weibo graph in pull mode.

Our further investigations on cache misses and edge accesses show that locality and batching across snapshots are the underlying reasons for ImmortalGraph's superior performance and also help explain some of the differences observed in different modes. We report those numbers next.

Reduced cache-miss counts. Table II shows the numbers of CPU cache misses and TLB misses for MIS (one iteration) on the Wiki graph; we have observed similar effects for other applications on other graphs (omitted). The numbers are measured through

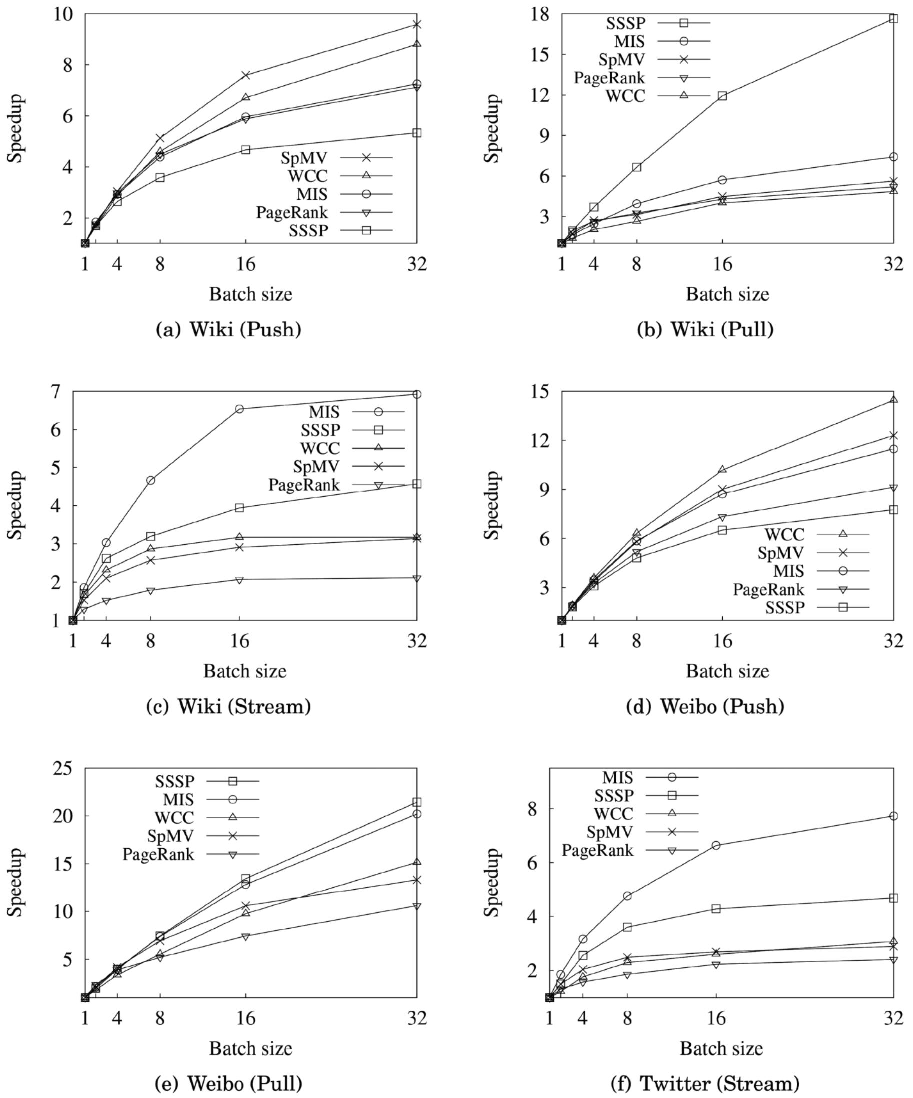


Fig. 19. ImmortalGraph single-thread speedup in computation time.

hardware CPU performance counters. As shown, the miss count decreases with the increase of the batch size. This suggests that the speedup is due to better data locality and explains why a larger batch size brings more gains.

In push mode, ImmortalGraph enables consecutive writes for multiple snapshots, which reduces the number of cache misses. Likewise, the consecutive reads in pull mode bring similar benefits to ImmortalGraph. In stream mode, ImmortalGraph is particularly beneficial in the shuffle stage [Roy et al. 2013]. The shuffling of the edge-associated values in multiple snapshots is performed consecutively in a batch, thereby reducing the number of cache misses.

Note that, in stream mode, the TLB-miss count is small compared to other modes due to its streaming behavior. Also, even when the batch size is 1, the stream mode

Table III. Number of Edge Array Access for PageRank in the First Iteration

Graph	BS: 1	BS: 4	BS: 16	BS: 32
Wiki	757 M	200 M	62 M	40 M
Twitter	1193 M	323 M	104 M	62 M

Note: BS = batch size.

reduces the chance of random access, which in turn reduces the cache miss count. This explains why we observe the least gain in stream mode.

Reduced access to edge array. Another factor contributing to ImmortalGraph's better performance is the batching effect across snapshots, which reduces the number of edge-array accesses. For each propagation, a vertex needs to access its edges in the edge array to discern its neighbors before it can propagate the value in push and stream mode, or pull the value in pull mode. In ImmortalGraph, the edge access is done for all batched snapshots once, rather than once for each snapshot.

A larger N brings larger benefits due to more saved accesses to edge array. Table III shows the numbers of accesses to edge array for PageRank in the first iteration on the Wiki and Twitter graphs. In the first iteration, the number of edge accesses in push, pull, and stream modes is the same. As expected, the larger the batch size, the fewer the number of edge accesses in the edge array.

ImmortalGraph with incremental computation. We then show the benefit of LABS-enhanced incremental computation in ImmortalGraph compared to the standard incremental computation approach. In the experiment, we compute 128 snapshots that are evenly spread over the last 10 months of the Wiki graph (from June 2006 to March 2007). Two adjacent snapshots are separated more than 2 days apart, which account for more than 130k edge activities on average.

The standard incremental computation approach runs on each snapshot in sequence, using the results of the previous snapshot. The LABS-enhanced incremental computation with a batch size of n first computes the first snapshot S_0 and incrementally computes the next n snapshots ($S_1 ... S_n$) using LABS by reusing the results of S_0. It further computes the following n snapshots ($S_{n+1} ... S_{2n}$) incrementally by reusing the results of snapshot S_n. The computation moves forward until all 128 snapshots are calculated.

Figure 20 shows the comparisons for WCC and SSSP on the Wiki graph in the single-thread case. We choose to use push mode because in this mode only updates are propagated, making incremental computation more effective. The x-axis represents different batch size in LABS, and the y-axis shows the performance improvement of the proposal in percentage. Note that the case in which batch size equals 1 is the standard incremental computation.

Figure 20 shows that our proposal can outperform the naïve incremental method by more than 60%. Initially, the increase of the batch size brings more benefits due to a larger batching effect, as previously explained. When batch size becomes even larger, the difference between later snapshots (e.g., $S_1 ... S_n$) and the initial snapshot (e.g., S_0) is also larger. This introduces more *duplicated* computation for later snapshots, assuming a simple approximate model in which the amount of incremental computation between two snapshots is proportional to the number of changes. For example, to calculate S_n from S_0 incurs more duplicated computation than to compute S_n from S_{n-2} (if $n > 2$). Hence, when the batch size is large enough, such unnecessary computation results in a reduced performance gain, as shown in Figure 20. A system should strike a balance between batching effects and incremental computation.

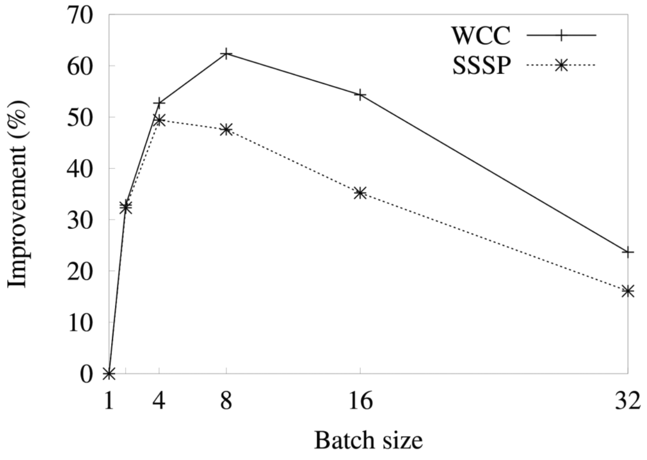


Fig. 20. The performance gain of incremental LABS against standard incremental computation with varying batch size on a Wiki graph.

Note that, in the multithread case, the benefit of a LABS-enhanced incremental computation will be amplified due to other advantages in multicore settings, as will be discussed in Section 6.4.

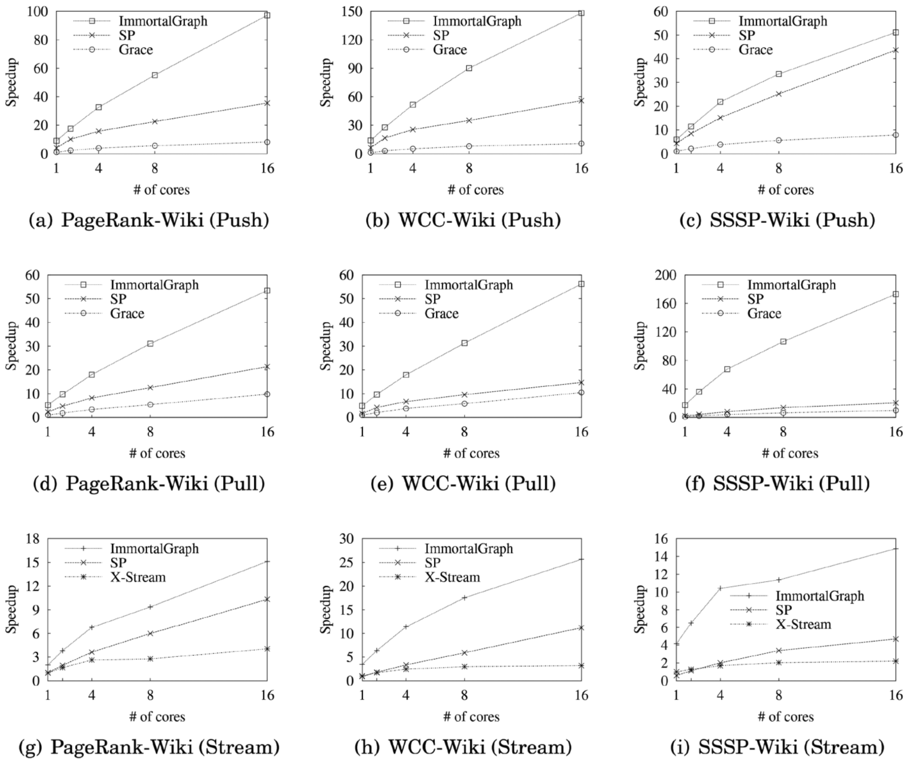
6.4. ImmortalGraph Performance on Multicore Machines

The performance of temporal iterative graph mining on a multicore server is heavily influenced by the subtle interplay among data locality, intercore communication, and other factors such as lock contentions. Our evaluation has shown that ImmortalGraph continues to outperform alternative designs and its advantage is sometimes even amplified.

We compare ImmortalGraph to two recent in-memory graph engines, Grace [Prabhakaran et al. 2012] and X-Stream [Roy et al. 2013], which are both optimized for multicore machines. Grace is a vertex-centric graph engine. The original implementation of Grace supports only push mode. We further extend Grace to support pull mode. X-Stream is an edge-centric graph engine that supports stream mode.

We modify X-Stream to support snapshot-parallelism. It is worth pointing out that the key design of ImmortalGraph can be integrated with different existing graph engines such as Grace and X-Stream, making it widely applicable and useful in enhancing existing graph engines.

In the experiment, we compute 32 snapshots with an equal time interval across the second half of the entire time range. (We select the first snapshot at the middle of the time range to have a snapshot large enough for a meaningful parallel computation.) All experiments use a batch size of 32. Figure 21 shows the results on the Wiki graph as the computation uses different numbers of cores. We are using the same baseline as in the single-threaded case. Partition-parallelism and snapshot-parallelism are used in this set of experiments. The y-axis denotes the speedup compared to our baseline. Note that, even on a single core, ImmortalGraph with batch size 32 has already achieved significant speedups, as shown in the previous experiment (Figure 19). We see more than 10 times of additional speedup on 16 cores (without hyperthreading). As shown in Figure 21, ImmortalGraph scales better than Grace and X-Stream in all three modes and for all tapplications. We observe similar effects in other graphs (e.g., the Weibo/Twitter graph in Figure 22) and other applications. We discuss the differences in the three modes that lead to different performance behaviors at the end of this section. Next, we



(a) PageRank-Wiki (Push) (b) WCC-Wiki (Push) (c) SSSP-Wiki (Push)
(d) PageRank-Wiki (Pull) (e) WCC-Wiki (Pull) (f) SSSP-Wiki (Pull)
(g) PageRank-Wiki (Stream) (h) WCC-Wiki (Stream) (i) SSSP-Wiki (Stream)

Fig. 21. Performance comparisons on multicore with the Wiki graph. SP = snapshot-parallelism.

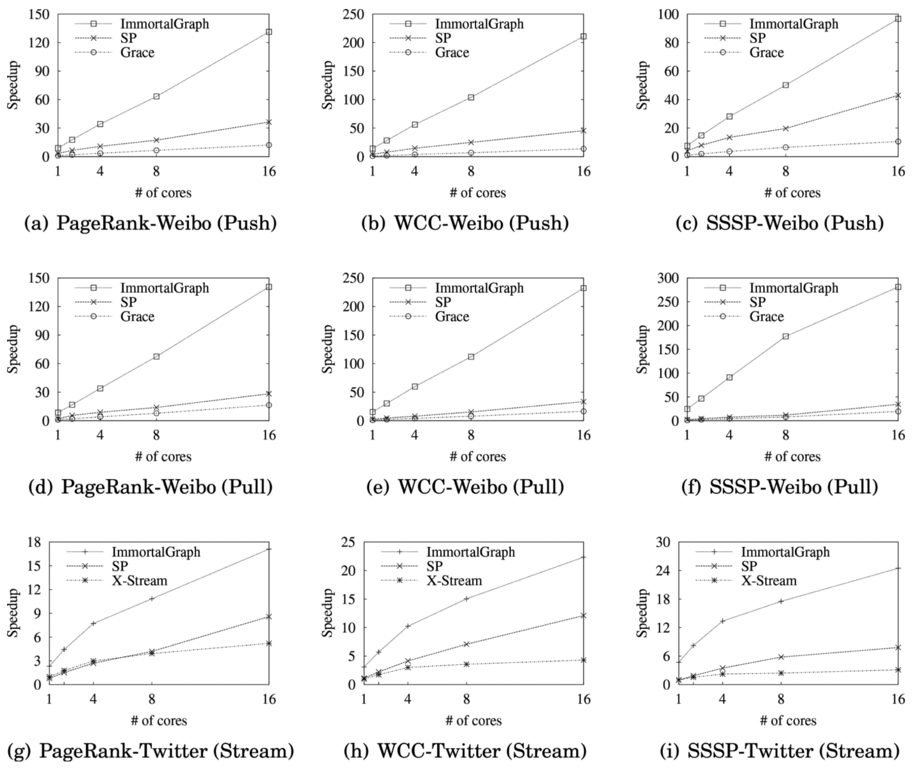
Table IV. Number of Intercore Communications for PageRank on the Wiki Graph (in millions)

	Push			Pull		
# of cores	2	4	8	2	4	8
ImmortalGraph	23.1	58.6	105.2	31.0	55.8	71.5
Grace	977.6	2471.6	4244.2	1740.4	3047.9	3923.8

report the results of in-depth analyses that help explain the performance advantages of ImmortalGraph. We further discuss the results of snapshot-parallelism later.

*Reduced intercore communications.*Our further investigation reveals that one key reason that ImmortalGraph outperforms Grace is the reduced intercore communication cost. In pull mode, ImmortalGraph pulls updates of a vertex from a remote core. ImmortalGraph performs such remote reads in a batch (across multiple snapshots). Because the vertex values in consecutive snapshots are placed together, ImmortalGraph reduces the number of remote reads: values of multiple snapshots are likely stored within a cache line. The push mode in ImmortalGraph has the same benefit for a similar reason, except that a remote read becomes a remote write (i.e., push). In stream mode, the intercore communication is not a dominant factor because each CPU core mainly communicates with the memory. The data exchange between cores are done using memory indirectly.

Table IV shows the intercore communication overheads in push and pull mode for PageRank (one iteration) on the Wiki graph. As the table shows, the number of intercore communications (measured through hardware performance counter) of ImmortalGraph is significantly smaller than that of Grace in various multicore settings.



(a) PageRank-Weibo (Push) (b) WCC-Weibo (Push) (c) SSSP-Weibo (Push)

(d) PageRank-Weibo (Pull) (e) WCC-Weibo (Pull) (f) SSSP-Weibo (Pull)

(g) PageRank-Twitter (Stream) (h) WCC-Twitter (Stream) (i) SSSP-Twitter (Stream)

Fig. 22. Performance comparisons on multicore with the Weibo and Twitter graphs. SP = snapshot-parallelism.

Table V. Level of Lock Contention Comparison for PageRank on the Wiki Graph

# of cores	2	4	8	16
ImmortalGraph	1.32s	1.34s	1.85s	4.02s
Grace	28.85s	34.25s	47.54s	96.73s

Note: s = second.

Reduced lock contentions in push mode. In push mode, when a vertex performs a write operation to another vertex, it needs to acquire a lock to the destination vertex because multiple vertices may write to the same destination concurrently. This is another source of overhead in the multicore setting. LABS manages to reduce such lock contentions in push mode for ImmortalGraph as it acquires locks in a batch across snapshots.

Table V shows the total spinlock running time, an indication of the level of lock contention, in push mode for PageRank on the Wiki graph (1 iteration). It shows that ImmortalGraph incurs one order of magnitude fewer contentions than Grace. Similar trends can be observed for other applications in push mode. Note that lock contention is not a critical issue in pull and stream modes.

Snapshot-parallelism. Temporal graphs provide more choices for parallel computation. Snapshot-parallelism assigns each snapshot for one CPU core to compute. There is no lock contention or intercore communication. However, the computation within each CPU core cannot exploit locality to reduce cache misses as the LABS mechanism in ImmortalGraph does. Even for snapshot-parallelism, we use the same in-memory layout as described in Section 4.2. In particular, there is a single read-only edge array shared by all snapshots; the edge array uses the snapshot bitmap for compression. All cores will access the same edge array during computation, but no locking is needed as the array is read-only. This format reduces the in-memory footprint and can potentially even reduce cache misses. However, snapshot-parallelism cannot benefit from the reduced access to the edge array, as LABS does.

Figures 21 and 22 show the performance of various applications in different modes on different temporal graphs for snapshot-parallelism and ImmortalGraph. It shows that the performance of snapshot-parallelism is worse than that of ImmortalGraph. We observe similar trends for other applications.

Note that, in stream mode, snapshot-parallelism is sometimes slower than X-Stream when the degree of parallelism is low. This is because, in order to support snapshot-parallelism, we use some auxiliary data structure to extend the implementation of X-Stream, which increases the randomness of memory access.

Snapshot-parallelism in all three modes is able to consistently outperform Grace or X-Stream with the increase of core due to better parallelism (e.g., fewer intercore communications).

Other observations. Finally, we briefly comment on the difference of the push, pull, and stream modes.

As explained, push mode requires heavy locks for data propagation. Pull mode, on the other hand, reads data from other vertices concurrently and does not require locks. The stream mode is nearly lock-free: it requires only a few lightweight atomic operations in the scatter phase [Roy et al. 2013].

In pull mode, in order to detect whether the value of the neighbors has been updated, a vertex has to check the "dirty" bit of neighbors. Each vertex has to scan the edge array to discern its neighbors, thus requiring $O(|E|)$ access, where E is the set of edges in the graph. In contrast, in push mode each vertex needs to check the dirty bit of its own only, resulting in $O(|V|)$ access, where V is the set of vertices in a graph. Because $|E|$ is typically significantly larger than $|V|$, this cost is higher in pull mode than that in push mode. In addition, the overhead of a read operation to check the dirty bit of remote vertices is more significant in multicore environments, especially when neighbors are located in another CPU core, leading to even higher overhead. Experiments show that this can result in worse performance in pull mode for some applications, such as SSSP, than that in push mode.

In our experiments, we have observed that the memory footprint in stream mode is significantly larger than in the other two modes due to its edge-centric nature. In fact, X-Stream cannot accommodate the Weibo graph on a single machine with 128GB memory (note that X-Stream is capable of leveraging external memory, which is out of the scope of this article). Moreover, unlike in push and pull modes, in which updates go directly to the destination vertex, stream mode achieves this indirectly through edges and has an extra shuffling stage. This incurs additional read/write operations.

Our experiences with the three modes indicate that no single mode is the best for all applications on all graphs due to the different trade-offs in each mode. However, despite the different performance characteristics in different modes for different applications, the benefits of ImmortalGraph have shown up consistently in all cases.

6.5. ImmortalGraph Distributed Performance

We have set up a small distributed testbed to test whether the benefits of ImmortalGraph extend to a distributed setting. In particular, our distributed testbed consists of 4 servers that are connected through InfiniBand. To fully exploit the capability of InfiniBand, ImmortalGraph uses MPI for the intermachine communications.

Table VI shows the running time for different applications (5 iterations) on the Web and Weibo graphs in the push mode. The web graph has 12 snapshots, so we

Table VI. ImmortalGraph Performance in Distributed Environments

	Web graph			Weibo graph		
Applications	PageRank	WCC	SSSP	PageRank	WCC	SSSP
ImmortalGraph	472s	332s	124s	2002s	1250s	48s
Baseline	781s	670s	136s	7318s	6405s	518s

Note: Baseline = to compute snapshot by snapshot.

Table VII. Breadth-First-Search Performance

	ImmortalGraph	TSB-tree
Wiki	16.5 s	52.1 s
Twitter	28.9 s	48.5 s
Weibo	2027 s	6306 s

Table VIII. Two-Hop Query Performance

	ImmortalGraph	TSB-tree	PostgreSQL
Wiki	3.9 s	7.0 s	66.6 s
Twitter	2.0 s	3.2 s	118.4 s
Weibo	23.6 s	33.4 s	1680.0 s

set the batch size to 12. The Weibo graph runs on 32 snapshots. To focus more on the distributed environment, we use a single thread on each server. The results show that ImmortalGraph can run more than 3 times faster than the naïve implementation that computes snapshot by snapshot (PageRank on Weibo). Note that applications run slower in the Weibo graph because the number of cross-partition edges is much larger than that of the Web graph: the ratio between interpartition and intrapartition edge number is 3:1 in the Weibo graph and 1:2 in the Web graph.

Because network communication incurs high overhead, the gains from better locality in ImmortalGraph are smaller in the end-to-end performance, compared to a single-machine setting. We expect the benefit to be less visible in a more network-constrained environment, in which the network communication cost dominates, even though our solution does enable batching across snapshots to make communication more effective.

6.6. Comparison to Relational Database

We use breadth-first-search (BFS) at a time point in the middle of the time range of the snapshot group to investigate the performance difference between TSB-tree and ImmortalGraph. In the experiment, ImmortalGraph uses the time-locality format. It is shown in Table VII that ImmortalGraph can be up to 3 times faster than TSB-tree.

The major overhead of TSB-tree comes from the inefficient index support for graph-specific operations. TSB-tree needs O(log N) time when traversing from one vertex to its neighbor (where N is the total number of vertices in the graph), while ImmortalGraph needs just O(1) time. Moreover, the compact size of ImmortalGraph's array-based index also offers a great advantage compared to the tree-based index, which requires a lot of pointers.

Table VIII shows the comparison results of two-hop query on 1000 randomly chosen vertices at a random time point. Because two-hop query involves less index access, the performance gap between TSB-tree and ImmortalGraph is smaller. We also show the performance of PostgreSQL. Since PostgreSQL does not optimize for the temporal data query, it performs the worst in the three solutions.

7. RELATED WORK

Temporal graph research is active and has uncovered important properties in real-world temporal graphs [Leskovec et al. 2005; Wilson et al. 2009; Aggarwal and Wang 2010].

There exist systems that consider a subset of temporal operations supported by ImmortalGraph. Khurana and Deshpande [2013] develop a system to retrieve all snapshots of a temporal graph given a time range. The system does not consider the access to a subarea in the temporal graph and algorithms designed specifically for temporal graphs. Ren et al. [2011] study the computation for a shortest path-like graph algorithm on a series of snapshots in a temporal graph. The system cannot support other categories of graph algorithm (like PageRank). More important, no system has studied the trade-off between different graph layouts for different temporal graph access patterns, and no system has exploited the benefit of data locality in multicore systems as well as the interplay with incremental computation when processing a series of graph snapshots iteratively.

Graph database systems such as Neo4j [Neo4j 2013] support graph traverse efficiently. A comprehensive list of existing graph databases is available in Wikipedia [2013]. None of them, however, stores historical graph data, not to mention analysis of it.

On the other hand, temporal data access has been studied extensively in relational data models. Salzberg and Tsotras [1999] survey access methods for time-evolving data. In a relational data model, historical data access can be characterized as a key/time-based point query (i.e., given a specific key and time) or range query (i.e., both the key and time can be a range). A variety of tree-based indices such R-Tree, TSB tree [Lomet and Salzberg 1989] and HV-Tree [Zhang and Stradling 2010] have been proposed for the key/time-based queries [Salzberg and Tsotras 1999]. Several database systems such as the TSB-tree-based ImmortalDB [Lomet et al. 2005] and PostgreSQL [PostgreSQL 2013] support such key/time-based data lookup.

In a temporal graph, a key/time-based query is still useful. For example, queries for a vertex/edge at a given time instance can leverage the techniques for key/time-based lookup. Complementing the key/time-based historical data access techniques in the relational model, ImmortalGraph is optimized for the graph traversal and iterative computation in a temporal graph.

ImmortalGraph enables efficient temporal iterative graph mining. The key technique that differentiates ImmortalGraph from existing graph engines is the joint design of the temporal graph layout and the scheduling mechanism (LABS) to fully exploit data locality of temporal graphs and the batching effect.

The importance of data locality is well known [Frigo et al. 1999] and has been studied in the context of multicore graph engines in Grace [Prabhakaran et al. 2012] and X-Stream [Roy et al. 2013]. ImmortalGraph further advocates exploiting the data locality along the time dimension, even at the expense of trading data locality in graph structure.

ImmortalGraph is complementary to the recent research on graph engines in that it is applicable not only to the vertex-centric graph engines, whether it is push based [Malewicz et al. 2010; Prabhakaran et al. 2012] or pull based [Low et al. 2012; Cheng et al. 2012; Shun and Blelloch 2013], but also to those with edge-centric, stream-based graph engines [Kyrola et al. 2012; Gonzalez et al. 2012; Roy et al. 2013]. ImmortalGraph further explores interactions with techniques such as incremental computation [Cheng et al. 2012; Murray et al. 2013] to understand the trade-offs between incremental computation and locality. Comparing with existing graph engines, ImmortalGraph's interfaces enable operations on historical graph data such as accessing multiple snapshots and graph changes in a time range. This enables the analysis of temporal graph–specific applications such as interaction graphs or centrality of dynamic networks. Unique to temporal graphs, ImmortalGraph proposes LABS to further exploit the benefit of data locality in time dimension.

While previous work has used different file system structures either in a single system [Muller and Pasquale 1991] or in a data center [Thereska et al. 2012] to improve performance, ImmortalGraph leverages its knowledge of query types and replication to dynamically choose appropriate copies.

The proposed temporal graph data layout and LABS are also effective in distributed environments. It explores an orthogonal dimension in the design space and is largely complementary to techniques such as dynamic load balancing, priority scheduling, automatic pull/push mode switching, and fine-grained synchronization [Khayyat et al. 2013; Venkataraman et al. 2013; Shun and Blelloch 2013; Nguyen et al. 2013].

There exist other types of iterative in-memory data processing engines such as Piccolo, Spark, and Naiad [Power and Li 2010; Zaharia et al. 2012; Murray et al. 2013]. These engines are not specifically designed for graph mining, thus do not consider graph-aware optimizations.

8. CONCLUSION

Temporal graphs represent an emerging class of applications that impose a unique set of challenges that are not being sufficiently addressed by current systems. A temporal graph has both a spatial dimension and a temporal dimension, which is the source of many design challenges, but also enlarges the design space to offer interesting opportunities beyond what is possible for a static graph. ImmortalGraph's LABS demonstrates one such opportunity. We believe that temporal graphs will become even more important over time and hope that ImmortalGraph can inspire further system research in this new area.

ACKNOWLEDGMENTS

We sincerely thank the anonymous reviewers and *Transactions on Storage* editors for their valuable comments and suggestions. We also want to thank Thomas Moscibroda, who provided invaluable suggestions about applications that helped start this project, as well as Sean McDirmid and Nathan Holdstein for their suggestions about editing. This work has been partially supported by the National High-Tech Research and Development Plan (863 Project) 2012AA010903, as well as the National Science Foundation for Distinguished Young Scholars of China (Grant No. 61325010).

REFERENCES

Charu C. Aggarwal and Haixun Wang (Eds.). 2010. *Managing and Mining Graph Data. Advances in Database Systems*, Vol. 40. Springer. 40–43 pages.

Paolo Boldi, Massimo Santini, and Sebastiano Vigna. 2008. A large time-aware web graph. *SIGIR Forum* 42, 2, 33–38.

Sergey Brin and Lawrence Page. 1998. The anatomy of a large-scale hypertextual web search engine. *Computer Networks* 30, 1–7, 107–117.

Rong Chen, Jiaxin Shi, Yanzhe Chen, Haibing Guan, Binyu Zang, and Haibo Chen. 2013. *PowerLyra: Differentiated Graph Computation and Partitioning on Skewed Graphs*. Technical Report IPADSTR-2013-001. Shanghai Jiao Tong University, Shanghai, China.

Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. 2012. Kineograph: Taking the pulse of a fast-changing and connected world. In *Proceedings of the 7th European Conference on Computer Systems (EuroSys'12)*. 85–98.

M. Frigo, C. Leiserson, H. Prokop, and S. Ramachandran. 1999. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science (FOCS'99)*. 285–298.

Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI'12)*. 17–30.

George Karypis and Vipin Kumar. 1995. *METIS—Unstructured Graph Partitioning and Sparse Matrix Ordering System, Ver 2.0*. Technical Report. University of Minnesota, Minneapolis, MN.

Zuhair Khayyat, Karim Awara, Amani Alonazi, Hani Jamjoom, Dan Williams, and Panos Kalnis. 2013. Mizan: A system for dynamic load balancing in large-scale graph processing. In *Proceedings of the 8th European Conference on Computer Systems (EuroSys'13)*. 169–182.

Udayan Khurana and Amol Deshpande. 2013. Efficient snapshot retrieval over historical graph data. In *Proceedings of the 29th International Conference on Data Engineering (ICDE'13)*. 997–1008.

Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-scale graph computation on just a PC. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*, Vol. 8. 31–46.

Kristina Lerman, Rumi Ghosh, and Jeon Hyung Kang. 2010. Centrality metric for dynamic networks. In *Proceedings of the 8th Workshop on Mining and Learning with Graphs (MLG'10)*. 70–77.

Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. 2005. Graphs over time: Densification laws, shrinking diameters and possible explanations. In *Proceedings of the 11th ACM SIGKDD International Conference on Knowledge Discovery in Data Mining (KDD'05)*. 177–187.

David Lomet, Roger Barga, Mohamed F. Mokbel, German Shegalov, Rui Wang, and Yunyue Zhu. 2005. Immortal DB: Transaction time support for SQL server. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data (SIGMOD'05)*. 939–941.

David Lomet and Betty Salzberg. 1989. Access methods for multiversion data. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data (SIGMOD'89)*. 315–324.

Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. 2012. Distributed GraphLab: A framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment (PVLDB'12)* 5, 8, 716–727.

Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD'10)*. 135–146.

Alan Mislove. 2009. *Online social networks: measurement, analysis, and applications to distributed information systems*. Ph.D. Dissertation. Rice University, Houston, TX.

Keith Muller and Joseph Pasquale. 1991. A high performance multi-structured file system design. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP'91)*. 56–67.

Derek Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martin Abadi. 2013. Naiad: A timely dataflow system. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*. 439–455.

Neo4j. 2013. Neo4j: The graph database. Retrieved July 5, 2015 from http://neo4j.org.

Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A lightweight infrastructure for graph analytics. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*. 456–471.

PostgreSQL. 2013. PostgreSQL. Retrieved July 5, 2015 from http://postgresql.org.

Russell Power and Jinyang Li. 2010. Piccolo: Building fast, distributed programs with partitioned tables. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI'10)*. 1–14.

Vijayan Prabhakaran, Ming Wu, Xuetian Weng, Frank McSherry, Lidong Zhou, and Maya Haridasan. 2012. Managing large graphs on multi-cores with graph awareness. In *Proceedings of the 2012 USENIX Annual Technical Conference (ATC'12)*, Vol. 12.

Chenghui Ren, Eric Lo, Ben Kao, Xinjie Zhu, and Reynold Cheng. 2011. On querying historical evolving graph sequences. *Proceedings of the VLDB Endowment (PVLDB)* 4, 11, 726–737.

Liam Roditty and Uri Zwick. 2004. On dynamic shortest paths problems. In *Proceedings of the 12th Annual European Symposium on Algorithms (ESA'04)*. 580–591.

Daniel M. Romero, Brendan Meeder, and Jon Kleinberg. 2011. Differences in the mechanics of information diffusion across topics: idioms, political hashtags, and complex contagion on Twitter. In *Proceedings of the 20th International Conference on World Wide Web (WWW'11)*. 695–704.

Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-Stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*. 472–488.

Betty Salzberg and Vassilis J. Tsotras. 1999. Comparison of access methods for time-evolving data. *Computing Surveys* 31, 2, 158–221.

Julian Shun and Guy E. Blelloch. 2013. Ligra: A lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'13)*. 135–146.

Sina. 2013. Weibo. Retrieved July 5, 2015 from http://weibo.com.

Eno Thereska, Phil Gosset, and Richard Harper. 2012. Multi-structured redundancy. *Proceedings of the 4th USENIX Conference on Hot Topics in Storage and File Systems (HotStorage'12)*.

Shivaram Venkataraman, Erik Bodzsar, Indrajit Roy, Alvin AuYoung, and Robert Schreiber. 2013. Presto: Distributed machine learning and graph processing with sparse matrices. In *Proceedings of the 8th European Conference on Computer Systems (EuroSys'13)*. 197–210.

Wikipedia. 2013. Graph database. Retrieved July 5, 2015 from http://en.wikipedia.org/wiki/Graph_database.

Christo Wilson, Bryce Boe, Ra Sala, Krishna P. N. Puttaswamy, and Ben Y. Zhao. 2009. User interactions in social networks and their implications. In *Proceedings of the 4th European Conference on Computer Systems (EuroSys'09)*. 205–218.

Lei Yang, Lei Qi, Yan-Ping Zhao, Bin Gao, and Tie-Yan Liu. 2007. Link analysis using time series of web graphs. In *Proceedings of the 16th ACM Conference on Information and Knowledge Management (CIKM'07)*. 1011–1014.

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th Symposium on Networked Systems Design and Implementation (NSDI'12)*. 2–2.

Rui Zhang and Martin Stradling. 2010. The HV-tree: A memory hierarchy aware version index. *Proceedings of the VLDB Endowment (PVLDB)* 3, 1–2 397–408.

Received May 2014; accepted November 2014