

Scalable Analytics on Multi-Streams Dynamic Graphs

Angelos Anadiotis
Oracle
Lausanne, Switzerland

Muhammad Ghufuran Khan
Inria & Institut Polytechnique de Paris
Palaiseau, France

Ioana Manolescu
Inria & Institut Polytechnique de Paris
Palaiseau, France

ABSTRACT

Several real-time applications rely on dynamic graphs to model and store data arriving from multiple streams. In addition to the high ingestion rate, the storage and query execution challenges are amplified in contexts where consistency should be considered when storing and querying the data. Our work addresses the challenges associated with multi-stream dynamic graph analytics. We propose a database design that can provide scalable storage and indexing, to support consistent read-only analytical queries (present and historical), in the presence of real-time dynamic graph updates that arrive continuously from multiple streams.

KEYWORDS

Dynamic graph, read-only present and historical queries, multi-stream graph processing

1 INTRODUCTION

Dynamic graphs are omnipresent in the context of real-time applications that generate massive amounts of events. These events can be seen as high-velocity data streams, whose timely analysis is critical for applications such as monitoring of cyber attacks in system security applications [16], fraud detection in financial institutions [16], anomaly detection in computer networks [8] and many more.

Our work considers the above-mentioned real-time use cases. We consider a database modeled as a labeled property graph (LPG), that is continuously updated. Updates may arrive from multiple streams, creating a need for appropriate transaction support, in order to avoid inconsistencies; on such a graph, we must be able to answer analytical queries about the current graph state, as well as historical queries.

In this paper, we present the challenges (Section 2), outline the approach we investigate (Section 3), describe a new data structure we propose for storing dynamic graph data, named HAL (Section 4), together with its associated algorithms (Section 5 to Section 7). We present an experimental evaluation showing that our system's specific optimizations allow it to cut a good compromise between memory and speed, and outperform comparable systems, in Section 9, before discussing more related work and concluding.

2 CHALLENGES

In our problem setting, we focus on systems that can *continuously ingest temporal labeled property graph (LPG) updates from multiple streams* while at the same time supporting a diverse set of graph analytic queries with *snapshot isolation consistency guarantees*. Further, we assume that the clocks at the sites where streams originate are synchronized; this can be achieved by various techniques well-known in the distributed systems area, e.g., [9].

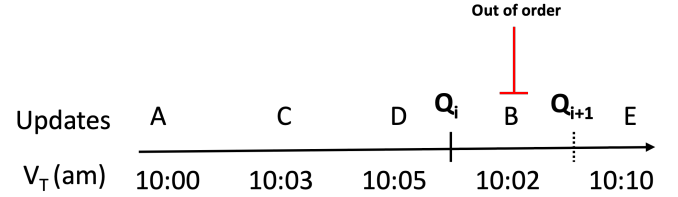


Figure 1: Example scenario for out-of-order update

Systems providing snapshot isolation guarantees typically order transactions based on their timestamps. Ideally, these timestamps would reflect the generation time of each ingested event, especially given that the clocks on the sources are synchronized. However, updates from a given stream may arrive later with respect to other streams due to any kind of transmission delay, leading to *out-of-order* [6] updates. Therefore, for every pair of events, the order of their ingestion time does not necessarily correspond to the order of creation time, and hence, any causal order between events may not be reflected by the database at all times.

Figure 1 exemplifies a sequence of updates labeled A to E, considering the update generation time (the time when the update is generated by a given stream) V_T . Note that update B arrives out of order (after D, which was emitted later than B). After the first three *received* updates have been ingested, that is, reflected in the database state, query Q arrives and requires two iterations on the updates ingested up to that time. In the first iteration, the query Q_i computes on the updates (A, C, and D), and before starting the second iteration $Q_{(i+1)}$, the delayed update B arrives. At this point, if B became visible to the query, then the database state would be consistent with respect to the data sources, but it would also break snapshot the isolation guarantees, which is not acceptable.

Existing systems, e.g., [8, 11, 13, 16], support temporal queries in a multi-stream setting, such as the one that the paper studies, by storing the event generation time as a property and then considering it during querying. However, in such a case, each query would need to access the whole database in order to filter out the events that do not match the time frame that the query considers. Therefore, the event generation time needs to be treated as a first-class citizen in dynamic graph databases.

A straight-forward approach to the above problem is to use an index on the event generation time. Nevertheless, graph queries already require indices to efficiently navigate through the graph with varying access patterns per query/graph traversal type. Accordingly, there is a need for a system design that addresses the challenges that are present both in traditional and in temporal graph queries.

In summary, this paper identifies the following challenges in building a scalable temporal graph database management system:

- **C1:** Provide scalable and consistent storage for multi-stream graph data ingestion.

- **C2:** Enable scalable time-based graph analytics for point queries.
- **C3:** Design storage organization and layout for dynamic property graphs.

Prior work addressing some of the challenges described above will be discussed in more detail in Section 10.

3 APPROACH

We aim at a scalable system that can answer analytical queries (present, historical) over dynamic graphs, updated multiple streams, whose updates may arrive out-of-order. To address the challenges listed in Section 2:

- We maintain a data store that follows the **multi-version concurrency control (MVCC)**, in short [15] storage model. In this store, the temporal graph updates from multiple streams are considered *write-only* transactions, while analytical queries (present and historical) are executed as *read-only* transactions. This ensures consistent graph analytics even in the presence of out-of-order updates, addressing challenges **C1**. Furthermore, to ensure correct edge semantics and consistency, we assume that **each sender (stream) is coherent**, which means that:
 - No sender ever sends a redundant edge insertion; an insertion is redundant if the stream previously sent such an insertion with no deletion of the same data item in between;
 - Similarly, there are no redundant edge deletions; a delete is redundant if the stream has already sent such a deletion, and the stream has not sent an insertion of an identical same data in between;
 - No sender will send a deletion of an entry for which it has not sent an insertion previously.
- To efficiently process point queries based on source time, we ensure that *the per-source neighbors lists are sorted by source time* (the time when an event originates on its source machine) *in descending order*, i.e., from the latest to the oldest. We use an Adaptive Radix Tree (ART) [10] index to preserve this order in the presence of an out-of-order update; this addresses the challenge **C2**. Details of the use of the ART appear in Section 4.
- To address challenge **C3**, we design the system in such a way that *the graph topology and graph properties are stored separately*, thus updates to the topology are processed separately from graph property updates. This allows us to optimize data access paths separately for these two largely orthogonal components of the property graph.

Core Design Decisions We consider an in-memory dynamic graph database management system. The graph is stored in an adjacency list, which provides a good trade-off between data access locality, required in queries, and high ingestion throughput, required in insertions. In the following, the data structures and algorithms that we propose achieve low computational complexity while maintaining high data access locality to optimize the use of modern hardware in scale-up servers.

4 DATA STORE: HISTORY ADJACENCY LIST (HAL)

In this section, we describe the main data structure we use to address challenges **C1** and **C2**.

We introduce the append-only **History Adjacency List (HAL)**, in short) data store, which ensures scalable data ingestion in the multi-stream scenarios as well as graph analytics in the presence of present and historical read-only queries.

As a general rule, each update is an edge addition or deletion, and we organize such updates primarily according to the source vertex of the added/deleted edge (not to be confused with an *update source*, one stream or site from which updates originate).

Within the HAL, we use a **Vertex Array (VA)**, in short), and **Source Time sorted Adjacency List (STAL)**, in short) as shown in Figure 2. Both the VA and the STAL have an entry for each graph vertex, denoted, respectively, $VA[s]$ and $STAL[s]$.

Each $VA[s]$ entry stores five fields:

- (1) A reference to the STAL entry corresponding to this vertex;
- (2) The **Latest Source Time (LST)**, in short), i.e., the latest source time of an in-order update received so far for this vertex;
- (3) A Hash table (**HT**[s], in short) keyed by the destination vertices d connected to s by some an edge $s \rightarrow d$. The HT value associated to a given d is the **Update Position and Indicator (UPI)**, in short) of $s \rightarrow d$, denoted $UPI[s, d]$. It compactly encodes the position of the latest (in-order or out-of-order) $s \rightarrow d$ entry in the STAL, together with a few more fields. Details of the HT and UP will be provided in Section 7;
- (4) **Degree**: the number of edge entries in the STAL;
- (5) A **lock**, used to prevent conflicts between transactions updating s .

For each source vertex s , its **Source-Time ordered Adjacency List (STAL)**, in short), denoted $STAL[s]$, contains:

- References to **STAL blocks (STALBs)**, see below);
- **STAL metadata**, specifically:
 - **curPos**, which points to the STALB most recently inserted in $STAL[s]$;
 - **size**, the length of $STAL[s]$ vector;
 - **isDeletion**, stating whether updates in $STAL[s]$ comprise one or more deletions;
 - **emptySpace**, the number of slots that have been freed by successive deletions in $STAL[s]$.

$STAL[s]$ is *ordered by source update time*, in the following sense: if the address of a block S_0 appears in $STAL[s]$ *before* the address of another S_1 , all the source update times appearing in S_1 are *after* those appearing in S_0 (as illustrated in Figure 2).

Each STALB stores information about a (fixed) number of edges going from s to various destination nodes. Specifically, the STALB comprises:

- (1) **Metadata** in the first 8 bytes (items (5) to (9) below);
- (2) **DestEntries** stores edge entries whose source is s and having various destination ids. These are sorted in the descending order of the source time of the update $s \rightarrow d$;

State-of-the-art	Present query	Historical queries	Out-of-order update
LiveGraph	YES	YES	NO
Teseo	YES	NO	NO
Sortledton	YES	NO	NO
Our system (Hal)	YES	YES	YES

Table 1: Comparison of state-of-the-art systems in the context of our dynamic graph challenges.

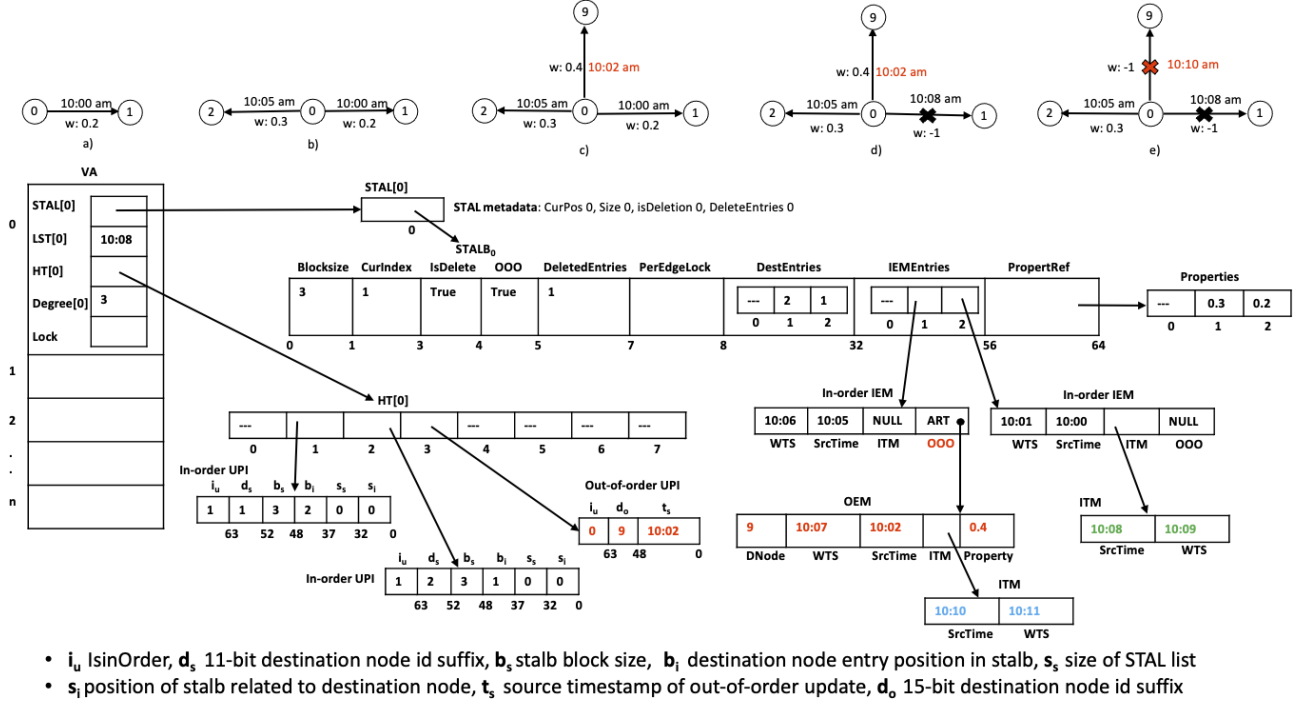


Figure 2: History Adjacency List (HAL)

- (3) **IEMEntries**, storing addresses of **In-order Edge Entry Metadata (IEM)**, in short (see below) entries for each of the above edge updates.
- (4) **PropertyRef**, a reference to a vector with the same number of entries storing, for each of the above edge updates, the properties that the edge may have, e.g., edge weight, etc. PropertyRef is stored in the last 8 bytes of STALB;
- (5) **Blocksize**, the size of the STALB;
- (6) **CurIndex**, the position of the most recently inserted entry in the STALB.
- (7) Two boolean flags (**IsDelete** and **OOO**) indicating whether there are deletions, or out-of-order updates in the STALB;
- (8) **DeletedEntries** stores the number of deleted entries in the STALB.
- (9) The **PerEdgeLock** field is used to lock the STALB if needed.

DestEntries and IEMEntries each occupy half of the STALB space not taken by the metadata or by PropertyRef. For instance, Figure 2 illustrates the STALB labeled S_0 , over 64 bytes: 8 bytes for metadata

(at the left), 8 bytes for edge properties (at the right), 24 bytes (3 entries) for DestEntries and 24 bytes (3 entries) for IEMEntries.

Each **IEM** (In-order Edge entry Metadata) block consists of:

- (1) The **write time-stamp (WTS)** is the transaction time of the edge, that is, the time when it is received at the database site;
- (2) The **SrcTime** is the time when an edge is emitted from the source machine;
- (3) The **Invalidation Time Metadata (ITM)** stores information about edge deletion (if applicable):
 - The source time of the deletion request, SrcTime;
 - The transaction time of the deletion request, WTS.
- (4) The out-of-order update (**OOO**) field stores the root of an ART [10] in which out-of-order edge entry metadata (**OEM**, in short) will be inserted. The order among entries in the ART is that of the update source time (recall that we consider the stream sources “reasonably” synchronized among themselves, thus a single timeline can be constructed from their source times).

Overall, the STAL structure ensures the update entries for a given source vertex s are stored *in the descending order of their source time*.

This is achieved on one hand through the ARTs, and on the other hand by keeping other data structures time-sorted during insertions, updates, and when querying, while also ensuring low complexity (in $O(1)$) for the operations we expect to be most common: in-order insertions and deletions. We discuss these algorithms next.

5 INSERTION

We detail the steps necessary in our platform for recording the arrival of a new edge, denoted $s \rightarrow d$. As a recall, we need to ensure scalable and consistent storage (challenge C1) whether updates arrive in order (the database site receives them in the order of their emission at the source sites), or out of order. Our algorithms leverage our proposed storage organization and layout for dynamic graphs (challenge C3). Further, meeting this challenges will enable to also provide efficient querying of the dynamic graphs (challenge C2), as we will show in Section 8. A new edge (insertion) entry $s \rightarrow d$ received at the database site is processed as follows:

- (1) Locate in the VA the position corresponding to s , e.g., if the new edge is $0 \rightarrow 1$ arriving at 10:00 am (V_T), this is the VA[0]. Lock this position to prevent conflicts between several write-transactions having the source vertex s .
- (2) Compare the new edge entry update source time (S_T) to the LST value in VA[s].
 - (a) If $LST < S_T$, then $s \rightarrow d$ is an *in-order* update; follow the steps in Section 5.1 below.
 - (b) Otherwise, it is an *out-of-order* update; follow the steps in Section 5.2 below.
- (3) Unlock STAL[s].

We next show how each of these insertions is handled.

5.1 In-order insertion

We handle these as follows.

- (1) Create an IEM block with: the edge's source time in SrcTime, NULL in the ITM, NULL in the OOO, WTS as the edge's transaction time.
- (2) Locate the STALB, say S_{io} , which is currently receiving updates. If S_{io} is full (that is, it contains L2Size entries), proceed to (3) below, otherwise, to (4).
- (3) If the L2Size of S_{io} is still smaller than an upper bound **MaxBlockSize**, then double the size of S_{io} , update the meta-data accordingly, and follow step (4). The aggressive resizing (by doubling up), on top of a small initial size, is adopted since it has been shown [5] to yield good performance in the (frequent) case when the number of edges adjacent to a node follows a power-law distribution. Otherwise (S_{io} had already been extended up to size MaxBlockSize), create a new STALB, call it S_{new} , of size 1, with the appropriate meta-data, and insert S_{new} in STAL[s]. The role of S_{io} below (4) will be played by S_{new} .
- (4) In S_{io} , at position $S_{io}.CurIndex-1$, store d in the DestEntries, the IEM block in the IEMEntries, the $s \rightarrow d$ edge properties in the Property vector. Then, update $S_{io}.CurIndex$, decrementing it by 1.
- (5) Create a new UPI, denoted UPI_e , storing the position (in the STAL) of the newly inserted edge entry. We say UPI_e is an *in-order UPI*, since it is due to an in-order insertion.

Concretely, UPI_e is an 8-bytes bit vector, structured in six fields. Figure 3 details their relative positions and lengths in the UPI, starting with the 0-th bit at the right:

i_u	d_s	b_s	b_i	s_s	s_i
63	(52, 62)	(48, 51)	(37, 47)	(32, 36)	(0, 31)

Figure 3: In-order Update position indicator (UPI) Bit Vector

- s_i , starting at position 0, stores on 32 bits $S_{io}.CurrPos$, the current position where $s \rightarrow d$ was inserted;
 - s_s stores on 5 bits as required to store the current size of STAL[s]. Recall that STAL[s] is created with an initial size of 1 and then its size is always a power of 2; for example, in Figure 2, the size of the STAL[s] is 1, we store $\log_2(1) = 0$. With 5 bits we can store values up to 32, leading to $MaxBlockSize=2^{32}$.
 - b_i stores on 11 bits $S_{io}.CurIndex$.
 - b_s stores the size of S_{io} , the STALB where the edge entry is inserted.
 - d_s stores the 11-bit suffix of d 's node ID;
 - The i_u flag (bit at position 63) is set to 1.
- Then, UPI_e is added to the hash table HT[s]. More details on the UPI and the hash table will be provided in Section 7.

For instance, assume we receive the following successive insertions: $0 \rightarrow 1$ and $0 \rightarrow 2$ at timestamps 10:00 am, and 10:05 am respectively. The resulting entries in our data structures are shown in black in Figure 2, while the newly created UPI (at the bottom left in Figure 2) has the values

1	1	3	2	0	0
---	---	---	---	---	---

 in its six fields. Note that we store in d_s only an 11-bit *suffix* of d 's identifier (not the full ID). While we rely on suffices for compactness, extra measures are taken when reading the data, to avoid confusing nodes whose IDs have the same suffix. We detail how this is achieved in Section 7. As shown above, the **computational complexity of handling an in-order insertion** is $O(1)$, since each step (including doubling up a STALB) takes constant time.

5.2 Out-of-order insertion

To handle the out-of-order insertion $s \rightarrow d$:

- (1) Create an out-of-order edge entry metadata (OEM) block. In this block, DNode stores d , WTS refers to the transaction time, SrcTime is the source time of the out-of-order update, and the ITM block is set to null.
- (2) Through binary search on STAL[s], using the update source timestamp, find the STALB, say S_{oo} , where the out-of-order update should be placed.
- (3) Through binary search on $S_{oo}.IEMEntries$, using the update source timestamp, locate the IEM block where the out-of-order update should be placed.
 - (a) If the OOO field of that IEM block is null, **create an ART tree** ordered by update source time, and holding the newly created OEM block as its first leaf. Otherwise (the ART exists), insert the OEM block to the ART.
 - (b) Set the OOO field of S_{oo} field to true.
- (4) Create a new UPI denoted UPI_e , to record the position of the newly inserted edge entry. UPI_e is itself called *out-of-order*

UPI, with a structure simpler than that of in-order UPIs (see Figure 4). It is also a 8-byte array, where:

- the i_u bit (position 63) is 0;
- d_o stores the 15-bits suffix of the destination node ID d ;
- the entry source time is stored in SrcTimestamp.

Then, UPI_e is added in the hash table $\text{HT}[s]$.

i_u	d_o	t_s
63	(48, 62)	(0, 47)

Figure 4: Out-of-order UPI for sample insertion.

For instance, assume that after the insertion illustrated in black text Figure 2, we receive the out-of-order insertion $0 \rightarrow 9$ with timestamp 10:02 am. The resulting entries are shown in red font in Figure 2, while the new UPI_e is

0	9	10:02
---	---	-------

 in Figure 2. Unlike in-order insertions, out-of-order ones accumulate in ART trees, and only the ART roots are stored in the in-order IEM blocks. Thus, out-of-order updates never lead to STAL blocks becoming full (or splitting); only in-order updates do that. Out-of-order insertions require binary search (in $\text{STAL}[s]$ based on SrcTimestamp, then in the STALB to find the IEM block where the ART root is/should be stored), and also incurs the cost of searching the ART, leading to a worst-case **complexity** of $O(\log(|E|))$. While this is not constant-time, we can reasonably expect out-of-order updates to be less frequent than in-order ones; ours is the first dynamic graph management system capable of correctly handling a mix of in-order and out-of-order updates.

6 DELETION

We now consider the deletion of an edge e of the form $s \rightarrow d$. Recall that we assume that an edge deletion only occurs after the respective edge has been inserted. We proceed as follows:

- (1) Locate the position of the source vertex s in the VA.
- (2) Lock it to prevent conflicts with other write transactions (insertions or deletions) where s is the source node, and get its hash table, say $\text{HT}[s]$.
- (3) Look up d in $\text{HT}[s]$, to get the UPI (Section 4) associated to the insertion entry for e , let's call this UPI_e . UPI_e encodes whether the deletion request is related to in-order update, respectively, an out-of-order update. We handle them as discussed below.
- (4) After handling the deletion, unlock $\text{HT}[s]$.

6.1 In-order deletion

- (1) From UPI_e (recall Figure 3), access s_i to get the position, in $\text{STAL}[s]$, of the STAL block, call it S_i , where the insertion $s \rightarrow d$ had been stored, at the time of that insertion. Note that this position may no longer be correct, in case the $\text{STAL}[s]$ has been resized at some point in time after e was inserted! We discuss how to handle such situations shortly below.
- (2) To account for the possible resize that may have changed the position of S_i , from the STAL metadata, we get the current size of $\text{STAL}[s]$ (Size in Figure 2). We then compute the *current* position of S_i in $\text{STAL}[s]$, of as

$$\text{STAL}[e].\text{Size} - (\text{UPI}_e.s_s - \text{UPI}_e.s_i)$$

In the above, $\text{UPI}_e.s_s - \text{UPI}_e.s_i$ computes how far the S_i position is, from the last index of $\text{STAL}[e]$ when the edge entry was inserted. Call this difference Δ_{si} . Then, we subtract Δ_{si} from $\text{STAL}[e].\text{Size}$ to obtain the current entry position in the $\text{STAL}[e]$.

For instance, in Figure 2, $\text{UPI}_e.s_i$ is 0, $\text{UPI}_e.s_s$ is 1, and the current $\text{STAL}[e].\text{Size}$ is 1, thus $(1 - (1 - 0))$ is 0, indicating the current position of S_i .

- (3) Locate S_i in $\text{STAL}[s]$ as the block at the position computed as above.
- (4) $\text{UPI}_e.b_i$ is the position of the e insertion entry in $S_i.\text{IEMEntries}$, when e was inserted.
- (5) $\text{UPI}_e.b_s$ is the size of S_i when e was inserted say.
- (6) Compute the exact position of the insertion in $S_i.\text{IEMEntries}$ as:

$$S_i.\text{BlockSize} - (\text{UPI}_e.s_s - \text{UPI}_e.s_i)$$

The reasoning behind the calculation is the same as above (leveraging the same Δ_{si}).

Then, we take the following steps:

- (a) Access the IEM block in $S_i.\text{IEMEntries}$, call it IEM_i , corresponding to the insertion of e in S_i .
- (b) Create an ITM block, call it ITM_d , with SrcTime as the timestamp when the deletion request originated from the source machine. Store ITM_d in the ITM field of IEM_e .
- (c) Set the S_i IsDelete flag to true.
- (d) Commit the deletion request by storing the current (transaction) time in the WTS of ITM_d .

For instance, assume that we receive the in-order deletion $0 \rightarrow 1$ with timestamp 10:08 am. The resulting ITM entry is shown in green font in Figure 2:

10:08	10:09
-------	-------

.

As shown above, the **complexity of handling in-order deletions** is $O(1)$.

6.2 Out-of-order deletion

This process is slightly different, given that for OOO updates we need to store the source timestamp in UPI_e (see Figure 4). To keep the data structures compact, UPI_e does not store the STAL position of the insertion entry. Instead, we have to retrieve it by binary search on the ordered data structures, as follows.

- (1) Get the source timestamp of the insertion of e , from UPI_e .
- (2) Based on this, in $\text{STAL}[s]$, find the STALB, say S_i , where the out-of-order insertion of e was placed.
- (3) Locate the IEM block, say IEM_i , corresponding to the insertion of e , by binary search on $S_i.\text{IEMEntries}$.
- (4) $\text{IEM}_i.\text{OOO}$ field is the root of an ART, in which we look up the OEM block, call it OEM_i , corresponding to the insertion.
- (5) Create an ITM block, ITM_d , with the deletion request source time as SrcTime.
- (6) Store ITM_d in the ITM field of OEM_i .
- (7) Set the IsDelete flag of S_i to true.
- (8) Commit the deletion request by storing the current (transaction) time in the WTS of ITM_d .

For instance, assume that we receive the out-of-order deletion $0 \rightarrow 9$ with timestamp 10:10 am. The resulting ITM entry is shown in blue font in Figure 2:

10:10	10:11
-------	-------

.

Similarly to out-of-order insertions, **the complexity of handling out-of-order deletions** is $O(\log(|E|))$.

7 PER-SOURCE VERTEX TABLE (HT)

Recall that we store for each destination node d present in the $\text{STAL}[s]$, the most recent position of the each destination node's entry, called **Update Position Indicator** of s and d (or $\text{UPI}[s, d]$, in short). Each edge insertion leads to creating an UPI; each edge look-up needs to read the UPI, and similarly, each edge deletion must locate, then delete an UPI. In this section, we describe these operations: UPI creation on one hand, UPI look-up or deletion (their processing is quite similar) on the other hand. We store UPIs in a dedicated optimized hash table, based on the open-addressing technique [12]; we call this structure **Per-Source Vertex Hash Table (HT)[s, d]**, in short).

Below, Sections 7.1, 7.2, respectively, describe how we insert, look up, and delete content from the HT.

7.1 UPI Insertion

For inserting the UPI of a newly inserted edge entry $s \rightarrow d$, call this edge e , in the HT. Insertions of in-order UPIs are described under (1) below, while out-of-order UPI insertions follow (2).

- (1) **In-order UPI**
 - Create an in-order UPI, call it UPI_i , then store the e position of $\text{STAL}[s]$ as described in Section 5.1.
 - Compute the position, say i , in the HT where UPI_i should be stored, as $(d \text{ modulo HT size})$, and store UPI_i there if $\text{HT}[i]$ is empty; otherwise, increase i until we find a free position $\text{HT}[i]$ and store UPI_i there.
- (2) **Out-of-order UPI**
 - Create an out-of-order UPI, call it UPI_o , then store the e source timestamp in $\text{UPI}_o.t_s$ as described in Section 5.2.
 - Compute the position, say i , in the HT where UPI_o should be stored, as $(d \text{ modulo HT size})$ and store UPI_o there if $\text{HT}[i]$ is free; otherwise, increase i until we find a free position and store UPI_o there.

As shown above, collision handling is potentially expensive. To keep its cost under control in practice, the HT size is twice as large as its number of entries, making it likely to find an empty space very fast.

More generally, collisions could be further avoided by using the double hashing techniques proposed in Robin Hood hashing [3] and GraphTango [1] (not published yet). An important observation is that the former causes cache misses in case of collision; the more recent GraphTango [1] presents a cache-friendly double hashing technique; we could also experiment with it in the future.

7.2 UPI Deletion or Lookup

Now assume we need to find, in the HT, the UPI of an edge entry e corresponding to $s \rightarrow d$, in order to delete the UPI. We do that as follows:

- (1) Compute the position, say l , where the UPI related to e may exist in the HT, as $(d \text{ modulo HT size})$. Call that UPI_l .

- (2) Check the field $\text{UPI}_l.i_u$ field to see whether UPI_l is an in-order or out-of-order UPI. In the former case, follow with step (2a) below; in the latter, follow with (2b).

(a) In-order UPI

- (i) Access the field $\text{UPI}_l.d_s$ (suffix of the destination node for which UPI_l was created). Compare the 11-bit suffix of d , the destination node in e , with $\text{UPI}_l.d_s$: if they match, then follow step (2(a)ii); otherwise, read the UPI at the next HT position $l + 1$ into UPI_l , and return to step (2). This search stops when we find the precise UPI created when e was inserted, or we find an empty position in the HT.
- (ii) Go to the UPI_l indicated position in the STAL, read the destination id, and compare it with the ID of destination node d .
 - If they are equal, we have found the UPI related to e . If we are handling a deletion, delete the UPI at $\text{HT}[l]$.
 - Otherwise, read the UPI at the next HT position $l + 1$ into UPI_l , and return to (2).

Search stops when we find the UPI created when inserting e , or we find an empty position in the STAL, signifying that the desired UPI does not exist.

(b) Out-of-order UPI

- (i) Access the field $\text{UPI}_l.d_s$ and compare it with the 15-bit suffix of the destination node ID d . If they match, then follow with step (2(b)ii); otherwise, increment the HT index by 1 to get the next UPI_l and return to step (2). Search stops when we find a UPI related to e_i or we find an empty space in the HT which means e_i does not exist in the STAL.
- (ii) Access the source timestamp field $\text{UPI}_l.t_s$, and follow these steps:
 - Apply a binary search on $\text{STAL}[s]$ using $\text{UPI}_l.t_s$ to get the STALB to which t_s belongs, again apply a binary search on that STALB to get the IEM block where the out-of-order update is placed, further access the OOO field of that IEM block to get the ART root, and search the ART with t_s as a key, to get the OEM block of UPI_l .
 - Access the DNode field of the OEM block, and compare it with the destination node id d .
 - If they match, UPI_l indeed corresponds to the insertion of e . If we are handling a deletion, delete UPI_l from $\text{HT}[l]$.
 - Otherwise, increment the HT index to $l + 1$, read into UPI_l the entry at that index, and repeat from (2).

The search stops when we find a UPI related to e or an empty space in the HT.

8 QUERIES

Dynamic graph systems, including ours, can be used for a large variety of computations. Popular benchmarks compare them on algorithms including such as Breadth-First Search, PageRank, Community Detection, etc. The computation steps necessary for implementing any of these algorithms are well-known, and remain the

same regardless of the graph data management system. Thus, existing evaluation frameworks *implement a set of graph algorithms, on top of a uniform graph data access API*, whereas dynamic graph systems implement simple `getEdge(s, d, time)` and `getVertex(id, time)`-style operations. Since the graph algorithm costs are the same, the remaining performance differences can directly be attributed to the efficiency of the data store.

The main operations dynamic graph systems have to support are simple *edge requests*, of several forms. *Present* queries require the last committed state of the edge at the database site, i.e., the edge's current state. *Historical* queries explicitly specify time information. They can be either *point* (or *snapshot*) queries request an edge such as it was at a certain point in the past, or *interval* queries focused on a specific past time interval. Interval queries can be implemented as a sequence of point queries, one for each timestamp in the interval. Thus, below, we discuss the processing of **present** and **point** edge queries. Without loss of generality, below, we show how to evaluate a query requesting all the edges whose source node is s (the query node). The modifications needed to handle variants of this elementary query, e.g., asking for the edges' properties, or filtering based on them, etc., are quite straightforward. Each query also specifies either the present time, or a specific requested timestamp.

8.1 Present queries

We denote the present query by p_q , and the time when the query is asked, by τ . For each STALB in $\text{STAL}[s]$, starting from the currently inserted STALB towards the oldest STALB:

- (1) Check the `IsDelete` and `OOO` fields of each STALB. Four cases can occur:
 - (a) If both are true, the STALB contains both deletion entries and out-of-order updates. In this case, when traversing the metadata entries associated with this STALB, we will check each IEM entry's ITM and OOO fields to ensure that *we do not read a deleted IEM entry, and do not miss out-of-order updates*.
 - (b) If `IsDelete` is true and `OOO` is false, the STALB contains deleted entries but no out-of-order one. We will check the ITM field only.
 - (c) If `OOO` is true and `IsDelete` is false, there are out-of-order updates in the STALB, but no deletion. We will check the OOO field only.
 - (d) Both are false: we will not check ITM nor OOO fields in the IEM entries, thus speeding up the STALB traversal. This case analysis is done *once per STALB*; it may avoid accessing and testing fields in *all the IEM entries for this STALB*. Avoiding to access and read these boolean fields makes our system more cache-friendly. As our experiments show (Section 9), competitor systems such as LiveGraph, which makes some checks for each traversed edge, suffer, among others, from their poor usage of the cache.
- (2) Traverse IEMEntries and in parallel DestEntries, from the position `IEMEntries.CurIndex`. For each IEM entry m , at position pm in IEMEntries:
 - (a) Check whether $m.WTS < \tau$: recall that WTS is the transaction time of m . If this holds, and the current STALB is

in case (1d), return `DestEntries[pm]`, the destination node corresponding to the position of m (recall from Section 4 that IEMEntries and DestEntries are parallel, same-size arrays). Otherwise, ignore m and move to the next IEM entry. The check helps ensuring snapshot isolation: we only read the (non-deleted) edge entries that existed in the STALB before τ .

- (b) Check m 's OOO and/or ITM fields, or both, when the STALB is in case (1a), (1b) or (1c) above. If m has out-of-order updates, its OOO field points to an ART. For each OEM entry in that ART, call it o , proceed as follows:
 - (i) Check that $o.WTS < \tau$; WTS is the transaction time of the out-of-order entry o . If this holds, and the current STALB meets only case (1c), return `o.DNode`; otherwise move to the next ART entry.
 - (ii) If the current STALB includes deletions (`IsDelete` flag), also check the ITM field of o , to see whether o is deleted or not. If it is not deleted, return `o.DNode`; otherwise, the OEM entry is ignored.
- If m has ITM, then ignore the entry; otherwise, return `DestEntries[pm]`.

For present queries, the worst-case complexity is $O(|R|)$, where $R \subseteq Q$ is the set of edges that belong to the query result.

8.2 Point queries

Let h_q be a point query, issued at the query transaction time τ_h , and containing a user-specified source timestamp τ . This query asks for all the nodes d such that an edge $s \rightarrow d$ had been inserted in the graph, and had not been deleted, by τ_h . We proceed as follows:

- (1) Through binary search on $\text{STAL}[s]$, with τ as the search key, find the STALB from where we start reading, call it S_p . We will read up to the oldest STALB.
 - (2) Through binary search on S_p with τ as search key, get the entry location to start reading, say, e_i .
 - (3) Access the IEMEntries, and DestEntries fields of each STALB. For S_p , scanning starts from the `IEMEntries[e_i]`; for older STALBs, it starts from the `IEMEntries[CurIndex]`. For each IEM entry in the STALB, call it m , we check the below conditions:
 - (a) $m.WTS < \tau_h$, where WTS is the transaction time of the IEM entry. If it is true and the current STALB is in the case (1d) introduced in the Section 8.1, return `DestEntries[current index]`; otherwise, ignore it and move to the next. This ensures that h_q only reads edge entries that existed before the τ_h timestamp, contributing to consistency (snapshot isolation).
 - (b) Check the m 's OOO and/or ITM fields, or both, when the current STALB is in one among the cases (1a), (1c), or (1b, introduced in Section 8.1. If $m.ITM$ exists, proceed as follows:
 - (i) Check if $m.SrcTime > \tau$. If yes, m belongs to the result, because at timestamp τ , m was valid (not yet deleted). Hence, return `DestEntries[DestEntries.CurIndex]`.
- If $m.OOO$ exists, access the root of the ART, and for each OEM entry in that ART, call it o , do the following:

- (i) Check if $o.WTS < \tau_h$; WTS refers to the transaction time of out-of-order entry. If this holds, and if the current STALB is in the case (1c) introduced in Section 8.1, return $o.DNode$; otherwise move to the next ART entry.
- (ii) If the STALB currently read contains some deletions, check $o.ITM$ to see whether the o is deleted or not. If it is not deleted, return $o.DNode$. If o is deleted, follow the below steps:
 - (A) Access the $o.ITM$ block, say ITM_o .
 - (B) Check if $ITM_o.SrcTime < \tau$. If yes, then o should contribute to the result, because at the τ timestamp, o was valid; hence, return the $o.DNode$.

The worst-case computational complexity is $O(\log(|E|) + |R|)$ where the edge set R is the query result.

9 EVALUATION

We implemented the data structures and algorithms previously described, and describe experiments which confirm its performance advantages with respect to the state of the art. Below, we describe our hardware and software experimental setup (Section 9.1); the benchmark dataset and the algorithm used for analytics (Section 9.2); the qualitative analysis of related systems (Section 9.3). We conclude by performance studies on insertions (Section 9.4), updates (Section 9.5), and analytic querying (Section 9.6).

9.1 Hardware and software settings

We run our experiments on a dual-socket machine with intel Xeon E5-2640 v4, which has 40 hardware threads and 256 GB of RAM. All system source code is written in C++ and compiled on GCC v10.2, with the optimization flag -O3. In our system, the maximum number of entries allowed per block is 2047; for the competitor Sortledton [5], it is set to 512 entries per block. All reported times are medians over five runs.

9.2 Workloads

We use the synthetic graph datasets graph-500 [7] with scale factors (SF) 22, 24, and 26; the node fan-out in these graphs follows a power-law degree distribution. We also use one real-world graph, namely dota-league from [7]. These datasets, used in previous comparable works, are undirected, and do not contain multiple edges between two vertices. Translating them in our framework designed for directed graphs, like in prior work, we replace each undirected edge (s, d) by two directed edges, $s \rightarrow d$ and $d \rightarrow s$. Each edge has just one property, namely weight, that is double precision real number; we generate these weights at random between 0 and 1 with a uniform distribution. The main dataset metrics appear in Figure 5.

We compare our system with existing competitors using the LDBC graph analytics benchmark [7], from which we use five graph algorithms: Breadth-First Search (BFS), PageRank (PR), Single-Source Shortest Path (SSSP), Community Detection Via Label Propagation (CDLP), and the Weakly Connected Components (WCC). For fair comparison, the implementation of the graph algorithms is taken from the Graph Algorithm Platform Benchmark Suite (GAP BS) [2], and runs on the driver implemented by Teseo [11].

Dataset	Vertices $ V $	Edges $ E $	Average degree $ D $
Graph500-22	2,396,657	64,155,735	26
Graph500-24	8,870,942	260,379,520	29
Graph500-26	32,804,978	1,051,922,853	33
dota-league	61,170	50,870,313	836

Figure 5: Dataset description

9.3 Competitors and complexity comparison

We compared our system with three other cache-friendly hybrid analytical/transactional processing (HTAP) systems for graphs, namely: LiveGraph [16], Teseo [11], and Sortledton [5]. LiveGraph stores graph edges in adjacency list, one for each source node; it supports random vertex access, and sequential neighborhood access. The edges in each adjacency list are stored contiguously, thus reading them does not cause random accesses. To handle graph updates, LiveGraph manages *versions* of edge entries in the vector. This is costly in terms of memory, as we need to store, for each edge update, the transaction timestamp and the possible invalidation timestamp. Its advantage is to efficiently support historical queries, by appending new edges to their respective adjacency lists as they arrive. Thus, edges are naturally sorted by transaction time, allowing historical queries to run in $O(\log(|E|))$.

In contrast, Sortledton and Teseo follow a *set-based neighborhood* design, where the blocks of edges are sorted by destination id. The maintenance of edge entry versions is done by the Hyper Multi-Version Concurrency Control [14] protocol: both systems store the latest version of the edges in a sequential block, and older versions are stored in a linked list. In Sortledton, blocks of edges are sorted and connected through a skiplist. Teseo follows a Compressed Sparse Row (CSR) design, where the vertices and edges are stored in a B+ tree with 2MB-size leaves, called a FAT tree, which is a packed memory array supporting sequential vertex access and sequential neighborhood access. However, the set-based neighborhood design (sorted by destination ids) does not maintain the arrival order of the edges; hence, for historical queries, its complexity is worse, $O(|E|)$, than the one of LiveGraph, $O(\log(|E|))$.

System	Edge insertion	Edge deletion	Find edge
Sortledton	$O(\log(E))$	$O(\log(E))$	$O(\log(E))$
Teseo	$O(\log(E))$	$O(\log(E))$	$O(\log(E))$
LiveGraph	$O(1)$	$O(E)$	$O(1)$
HAL in-order	$O(1)$	$O(1)$	$O(1)$
HAL out-of-order	$O(\log(E))$	$O(\log(E))$	$O(\log(E))$

Figure 6: Complexity comparison for elementary graph operations

Figure 6 shows the time complexity of for the main operations (edge insertion, edge deletion, and edge look-up) on related systems. In practice, edge insertion requires two steps: (i) check if the edge exists already, (ii) insert it if not already there. Sortledton and Teseo perform (i,ii) in $O(\log(|E|))$. LiveGraph does not sort but simply appends the edges in the neighborhood list; it uses Bloom filters to check the edge's existence, which takes $O(1)$. However, if false

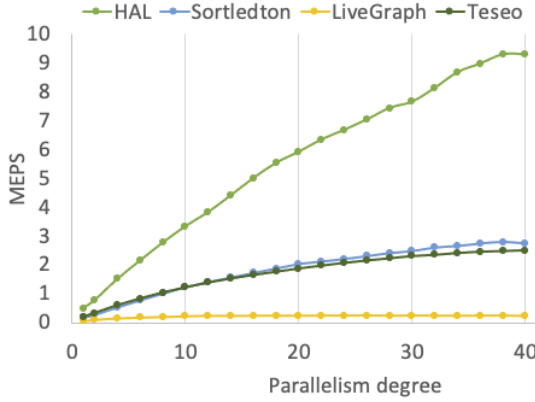


Figure 7: Graph500-24 scalability analysis.

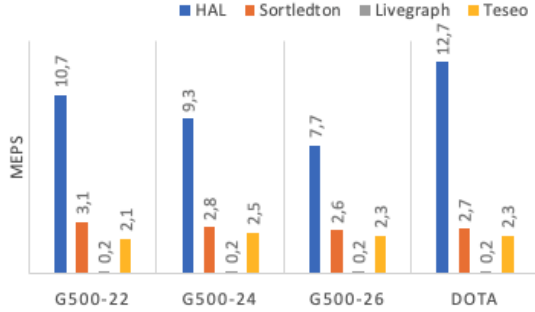


Figure 8: Insertion throughput on all systems and datasets.

positives occur, the verification raises the cost to $O(|E|)$. For in-order insertion, our system HAL does not need (i), as we treat each insertion (from a distinct site) as a separate update; we perform (ii) in $O(1)$ as we simply append edges to the neighborhood list. At the same time, we are able to check for the existence of an edge in $O(1)$ thanks to our HT. Out-of-order insertions, supported only in HAL, take $O(\log|E|)$ for step (ii). HAL’s complexity is better than competitors’ for in-order insertion. For out-of-order insertion, the complexity is similar to those of Sortledton and Teseo, but worse than LiveGraph’s best-case scenario (no false positive in the Bloom filter).

To find and delete edges, Sortledton and Teseo take $O(\log(|E|))$ because of the set-based neighborhood design, while HAL needs $O(1)$ for in-order updates and $O(\log(|E|))$ for out-of-order updates. LiveGraph takes $O(1)$ (without false positive) for deletions, and up to $O(|E|)$ (with false positives) to find an edge. For both deletions and edge search, for in-order updates, HAL has better complexity than the existing systems; for out-of-order updates, its complexity is the same as that of Sortledton and Teseo, and better than LiveGraph’s.

9.4 Edge insertions

To measure insertion performance, we insert successively all the edges of the graph500-24 dataset, in a random order. Since the

competitor systems do not support out-of-order insertions, we do not have them in our workload.

Figure 7 reports the insertion throughput, measured in Million (of inserted) Edges Per Second (MEPS, in short), as we increase the number of threads (on the x axis) from 1 to 2, 4, 6, ... until 40. **HAL scales up very well, better than the other systems, as we increase the number of threads.** While Sortledton and Teseo also gain from parallelism, they do so much less than HAL; LiveGraph does not benefit at all from it. This is due to contention between multiple writer threads, simultaneously trying to (i) search linearly in the adjacency list for many edge existence checks, and (ii) resize requests. At maximum parallelism, HAL outperforms Sortledton and Teseo by 3 \times , and LiveGraph by 30 \times . From now on, we report on experiments with 40 threads.

Figure 8 shows insertion throughput, again in MEPS, on different systems for our four datasets. **HAL performs better in all cases.** Its running time shows some variability when increasing the size of the datasets; this reflects the cost we pay for maintaining the HT and resizing the STAL vector. HAL performs better since we simply *append* the edges in adjacency lists, without sorting by destination id (insertions mostly in $O(1)$).

Sortledton and Teseo perform very similarly on all the datasets, because of their similar set-based neighborhood design; for edge existence check, this requires takes $O(\log_2|E|)$. Even though LiveGraph append newly arrived edges (without sorting by destination ids) in the adjacency list, it still performs significantly slower than the other systems. This is because LiveGraph provides completely sequential access to the adjacency list, which in turn leads to more resize requests as the edge vector, initially of size 1, is resized. The cost to check for the existence of the new edge also plays a role (see the discussion of false positives in Bloom filters above).

As previously explained, in our setting, edge existence checks are not needed upon insertions. But just to check whether HAL could efficiently also perform these checks, we ran an extra experiment, inserting the graph500-24 dataset with 40 threads and existence checks. The throughput decreased modestly, from 9.3 MEPS to 9.1 MEPS (2%), which is not significant, still leaves HAL the best-performing system.

Lesson learned: In systems that follow the set-based neighborhood design, with edge blocks that are connected through B+ tree (Teseo) or skip lists (Sortledton), the necessary sorting steps limit the throughput due to contention between writer threads; their advantage is that they do not need an extra index when checking for edge existence. In contrast, HAL simply appends edges in the adjacency list without sorting by destination ids, hence, its throughput is significantly better, because of lower contention between writer threads. However, it needs extra space for maintaining the secondary index (our hash table HT).

9.5 Updates (insertions and deletions)

Next, we evaluate our system on a mixed workload made of insertions and deletions, introduced in Teseo [11]. Unlike the previous insertion experiments, 10% of the operations in the update workload load the graph, after which, 90% of the operations are insertions and deletions keeping the (already large) graph of more or less the same size. The size of the update workload is $10 \times |E|$.

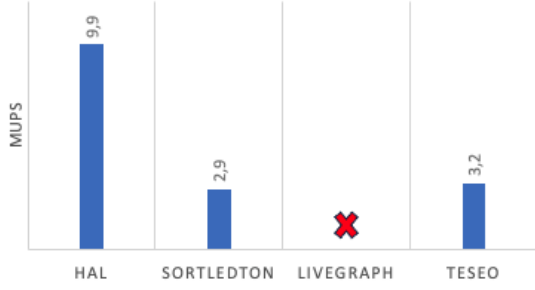


Figure 9: Graph500-24 update workload result

Figure 9 shows different systems’ throughput except for LiveGraph, which ran out of memory, on the graph500-24 datasets. The y axis counts Millions of Updates Per Second (MUPS). HAL performs more than 3× faster than Sortledton and Teseo, due to: (i) the append-only operations needed to record edge updates, whereas competitors need to use sorted list and/or linked lists (Hyper protocol), which incur on one hand sorting costs, and on the other hand more random memory accesses, thus lower cache usage; (ii) deletions in constant time compared to competitors that need binary search, and incur random accesses to the blocks accessed via skip lists in Sortledton and B+ trees in Teseo.

Lesson learned: Set-based systems have a lower update throughput as compared to HAL due to the reasons described above. However, these competitors need less space to manage different versions of the edges, compared to HAL which needs to store with each edge a transaction time (WTS), and invalidation time (deletion time of an edge), as well as the HT. Thus, HAL’s performance is obtained trading some space for more throughput.

9.6 Analytics

We run the LDBC graph analytics benchmark [7] on top of Teseo, Sortledton, LiveGraph and HAL, for our four datasets. We choose Sortledton (the latest system in the literature) as baseline, and measure the slowdown of other systems with respect to it.

Figure 10 shows the benchmark analytics algorithm results on these datasets and systems: the algorithms on the x axis, and the slowdowns on the y axis, the lower, the better. BFS and SSSP require random vertex access and sequential neighborhood access, where HAL performs three times better than the baseline system. In contrast, PageRank, WCC, and CDLP require sequential vertex access and sequential neighborhood access, in which HAL performs significantly better than LiveGraph and Teseo, and slightly better than Sortledton.

LiveGraph is slower than HAL due to storing edge entry metadata (transaction timestamp, invalidation timestamp, property size) with destination ids, which causes more cache misses during the access. For instance, a single edge entry takes 32 bytes, which means a cache miss occurs every two edge entries. On the other hand, in HAL, destination ids (DestEntries in STALBs) are stored separately from the edge entry metadata (IEMs), hence, cache misses occur more rarely, after eight edge entries. Additionally, LiveGraph checks each entry’s invalidation time to see whether the edge entry is deleted or safe to read. In contrast, in HAL system, the IsDelete

flag in each STALB helps to read the edge entry directly, if there is no deletion in the STALB.

Teseo performs significantly worse than HAL mainly because: (i) Teseo needs a per-edge entry mapping from sparse to dense vertex ids in the analytics part of the graph algorithm using a hash table, which is costly. (ii) In Teseo, sorted neighborhood block contains up to 512 edges as compared to HAL, where the maximum number of edges in a STALB is 2047, resulting in fewer random accesses. Sortledton’s performance is slightly slower than HAL’s because of the number of edges allowed in the block (512 for Sortledton, there are 512 edges per block, 2047 for HAL), leading to more random accesses for Sortledton.

Lessons learned Set-based systems (Sortledton and Teseo) incur fewer cache misses than LiveGraph, since the former read the latest version of the destination ids, while LiveGraph must traverse all versions of the destination ids in the adjacency list, together with per-edge information. LiveGraph is also hampered by the need to check invalidation timestamps, to see if an edge entry is still valid. Thus, set-based systems outperform LiveGraph. However, set-based systems do not preserve edge entry order, slowing down historical queries.

To provide scalable support for both present and historical queries, HAL adopts an approach in-between the two above. Specifically, HAL stores the destination ids separately from the edge entry metadata in the STALB, given that destination IDs are accessed more often. HAL spares per-edge deletion checks by checking each STALB’s isDelete flag, which may signal that a block contains no deletion, thus no check is needed on its edges.

Finally, converting sparse vertex identifiers into dense ones during analytics computation (like Teseo does) is costly. In contrast, HAL (like Sortledton) makes this conversion at edge insertion time.

10 RELATED WORK

Existing systems supporting multi-stream dynamic graph analytics can be classified into two main categories: those which provide transactional guarantees, such as LiveGraph [16], Teseo [11], and Sortledton [5], and those that do not, such as Llama [13], GraphOne [8], and STINGER [4].

Our work belongs to the former group, and Table 1 summarizes the challenges associated with these systems. Further, there are two main different data storage designs in transactional systems: (i) set-based neighborhood blocks, where the edges are sorted by destination ids and edge entry version maintenance is done by the Hyper protocol [14]; sample systems are Sortledton [5] and Teseo [11], and (ii) edges stored in the adjacency blocks without sorting by destination ids, with per-entry version management within the adjacency block, represented by LiveGraph [16]. We discussed these systems’ details in Section 9.3.

The advantage of the set-based design is that we do not need any extra index on the adjacency list to delete or lookup any specific destination entry. Also, it takes less space, as we do not manage the version in the adjacency list. However, it does not preserve the historical arrival order; hence, historical queries run on top of set-based systems are not efficient. On the other hand, in LiveGraph [16], the order of edge arrival is maintained.

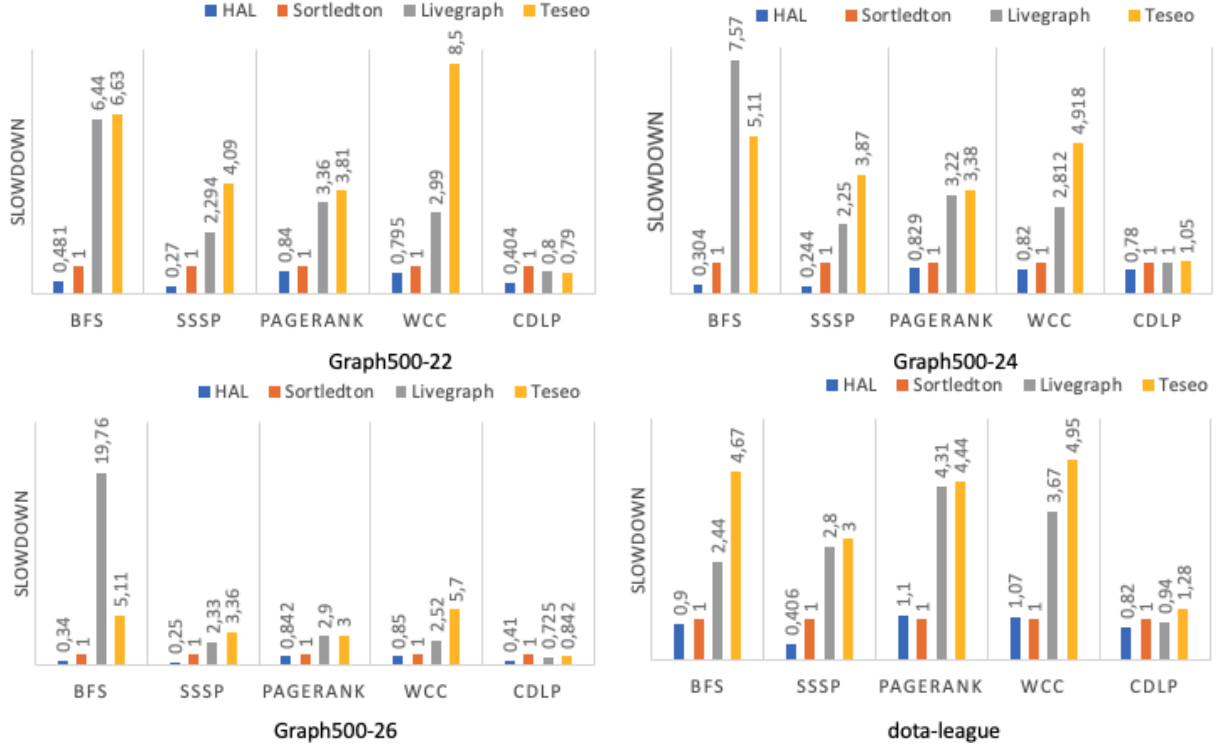


Figure 10: Performance evaluation on graph analytics.

All the systems mentioned in Table 1, as well as the one we aim for, support queries based on the current state of the dynamic graph. Except for LiveGraph [16] and this paper, no other system supports historical state queries. In our work, we follow the LiveGraph [16] MVCC protocol with the optimization described in Section 4 to improve update throughput and performance on analytics. LiveGraph [16] supports historical queries based on transaction time; on the other hand, we are supporting historical queries based on source time (time when the update was emitted from the source machine). Existing systems do not provide consistency guarantees in the presence of out-of-order updates, while our system attains this goal, as explained in Section 4 to 7 and validated through our experiments (Section 9).

11 CONCLUSION AND PERSPECTIVES

This paper aims to provide a scalable storage and indexing solution that can ingest real-time graph updates from multiple streams and, on top of that, provide consistent graph analytics (read-only present and historical queries) even in the presence of out-of-order updates. Our system performs approximately 9 million updates per second, which is three times faster than the Sortledton. We are still doing further experimentation to proof our system. In the future, we will provide mixed-workload (updates and analytics in parallel) and historical query results.

Acknowledgments This work received support from the ANR AI Chair SourcesSay project.

REFERENCES

- [1] Alif Ahmed, Farzana Ahmed Siddique, and Kevin Skadron. 2022. GraphTango: A Hybrid Representation Format for Efficient Streaming Graph Updates and Analysis. arXiv:2212.11935 [cs.DS]
- [2] Scott Beamer, Krste Asanović, and David Patterson. 2017. The GAP Benchmark Suite. arXiv:1508.03619 [cs.DC]
- [3] Pedro Celis, Per-Ake Larson, and J. Ian Munro. 1985. Robin hood hashing. *26th Annual Symposium on Foundations of Computer Science (sfcs 1985)* (1985), 281–288.
- [4] David Ediger, Rob McColl, Jason Riedy, and David A. Bader. 2012. STINGER: High performance data structure for streaming graphs. In *2012 IEEE Conference on High Performance Extreme Computing*. 1–5. <https://doi.org/10.1109/HPEC.2012.6408680>
- [5] Per Fuchs, Domagoj Margan, and Jana Giceva. 2022. Sortledton: A Universal, Transactional Graph Data Structure. *Proc. VLDB Endow.* 15, 6 (feb 2022), 1173–1186. <https://doi.org/10.14778/3514061.3514065>
- [6] Hassan Halawa and Matei Ripeanu. 2021. Position Paper: Bitemporal Dynamic Graph Analytics (GRADES-NDA '21).
- [7] Alexandru Iosup, Tim Hegeman, Wing Lung Ngai, Stijn Heldens, Arnau Prat-Pérez, Thomas Manhardt, Hassan Chafio, Mihai Capotă, Narayanan Sundaram, Michael Anderson, Ilie Gabriel Tănase, Yinglong Xia, Lifeng Nai, and Peter Boncz. 2016. LDBC Graphalytics: A Benchmark for Large-scale Graph Analysis on Parallel and Distributed Platforms. *Proc. VLDB Endow.* 9, 13 (Sept. 2016), 1317–1328.
- [8] Pradeep Kumar and H. Howie Huang. 2019. GraphOne: A Data Store for Real-time Analytics on Evolving Graphs. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. USENIX Association, Boston, MA, 249–263.
- [9] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (jul 1978), 558–565. <https://doi.org/10.1145/359545.359563>
- [10] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *2013 ICDE*. 38–49.
- [11] Dean De Leo and Peter A. Boncz. 2021. Teseo and the Analysis of Structural Dynamic Graphs. *Proc. VLDB Endow.* 14 (2021), 1053–1066.
- [12] Dapeng Liu and Shaochun Xu. 2015. Comparison of Hash Table Performance with Open Addressing and Closed Addressing: An Empirical Study. *Int. J. Networked Distributed Comput.* 3 (2015), 60–68.

- [13] Peter Macko, Virendra J. Marathe, Daniel W. Margo, and Margo I. Seltzer. 2015. LLAMA: Efficient graph analytics using Large Multiversioned Arrays. In *2015 IEEE 31st International Conference on Data Engineering*. 363–374.
- [14] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 2015. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (2015).
- [15] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. 2017. An empirical evaluation of in-memory multi-version concurrency control. *Proceedings of the VLDB Endowment* 10 (03 2017), 781–792.
- [16] Xiaowei Zhu, Guanyu Feng, Marco Serafini, Xiaosong Ma, Jiping Yu, Lei Xie, Ashraf Aboulmaga, and Wenguang Chen. 2020. LiveGraph: a transactional graph storage system with purely sequential adjacency list scans. *Proceedings of the VLDB Endowment* 13 (03 2020), 1020–1034.