Dynamic Graph Databases with Out-of-order Updates (extended version)

Angelos Christos Anadiotis, Muhammad Ghufran Khan, Ioana Manolescu

Dynamic Graph Databases with Out-of-order Updates (extended version)

Angelos Christos Anadiotis
Oracle
Zurich, Switzerland
angelos.anadiotis@oracle.com

Muhammad Ghufran Khan
Inria & Institut Polytechnique de Paris
Palaiseau, France
muhammad.khan@inria.fr

Ioana Manolescu
Inria & Institut Polytechnique de Paris
Palaiseau, France
ioana.manolescu@inria.fr

ABSTRACT

Several real-time applications rely on dynamic graphs to model and store data arriving from multiple streams. Providing both high ingestion rate and efficient analytics with transactional guarantees is challenging, even more so when updates may be received *out-of-order* at the database. In this work, we propose HAL, a novel in-memory dynamic graph database design, addressing these challenges. HAL outperforms comparable systems by a factor of up to 73× in terms of update processing throughput and up to 357× for analytics, while being the first to support out-of-order updates.

1 INTRODUCTION

Dynamic graphs are omnipresent in real-time applications that generate massive amounts of data [45]. In particular, timely analysis of high-velocity graph data streams [12] is critical for applications such as monitoring of cyber attacks in system security applications [53], fraud detection in financial institutions [53], anomaly detection in IOT networks [30, 52] and many more.

We consider dynamic graphs, where *edges are continuously added and deleted to a single graph, from multiple update streams*. The dynamic graphs are stored in a transactional graph database. Each edge update or deletion carries a *source (stream) time* S_T, assigned at the moment when it was emitted, and an *arrival (or transaction) time* W_T, assigned when the graph database receives it. We assume that *all the stream timelines are reconcilable*, that is: a global order can be established over any set of updates coming from multiple streams[1]. Furthermore, updates may be received at the database **out-of-order** [17] (**ooo**, in short): due to different latencies on the propagation paths between the data source and the database, there may be two edge operations (insertion and/or deletion) u_1, u_2, issued at stream times $S_T^1 < S_T^2$, concerning the same or different edges, such that u_2 is received at the database at a time W_T^2 when u_1 has not yet been received. Therefore, ooo updates are ordered differently based on source time (S_T) and on database time (W_T).

Such scenarii are frequent in IoT applications [10, 29], where changes in the network topology cause edges to appear and disappear rapidly. In IoT networks [20], issues such as multi-path routing [35, 51], route fluctuations [4], link-layer retransmission [13, 26], and router forwarding [37] may lead to ooo updates. For instance, in Intelligent Transportation Systems, there are two main variations: (*i*) Nodes (sensors) are fixed, e.g., intersections, while edges reflect route segments whose status may vary in real-time due to anomalies such as traffic jams, accidents, road closures, etc. [8, 16, 30, 34, 46, 52]. If a sensor sends an update indicating the opening of a route, followed by another update for its closure, receiving these ooo may lead to misrepresent the current traffic situation. (*ii*) Nodes move, e.g., Unmanned Aerial Vehicle (UAVs); edges denote communication links between them [43]. As UAVs move, these links appear and disappear rapidly. This dynamic behavior is crucial for security surveillance and monitoring, agricultural field inspection, [5, 21, 39, 50]. If an edge denoting the connection between two UAVs is created then removed, and the updates are received ooo, again path planning may be impacted.

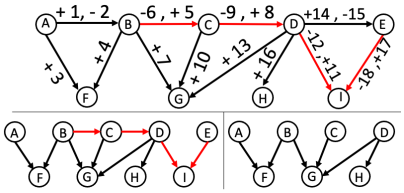

Figure 1: Sample dynamic graph with out-of-order updates.

Figure 1 illustrates dynamic graphs with ooo update propagation on a graph (at the top) where for each edge, the database receives an insertion, and for most, it also receives a deletion. On each edge, we show the one or two possible operations, insertions (+) and/or deletions (-) *in the order of their arrival* at the database, together with their stream time S_T. For instance, for the edge between A and B, +1, -2 states that the insertion with S_T=1 arrives before the deletion with S_T=2. On the contrary, for the edge between B and C, the deletion emitted at 6 is received before the insertion emitted at 5. In existing dynamic graph systems, e.g., [11, 14, 23, 28, 38, 53], unaware of possible ooo updates, updates are processed in the order in which they are received on the database site; these systems assume that no deletion arrives in the database before its associated insertion. **Such a deletion is ignored**; the (delayed) edge insertion is applied, leaving the edge in the system. Instead, the deletion should be recorded even before the matching insertion is received; when the insertion arrives, the respective edge should be understood as having been inserted *then* deleted. In our example, after the updates arriving as shown at the top of Figure 1, the graph stored by a system that does not handle ooo insertions is the one depicted at the bottom left. However, **the correct final graph** (at the bottom right) accurately depicts the sequence of events based on the order of their generation; the two differ in the edges shown in red at the top, and which were transmitted ooo.

Dynamic graph databases with ooo updates raise two challenges: maintaining consistent snapshot views of the dynamic graph, to be able to answer analytical queries over it; and, balancing high transactional write throughput and accurate analytical scans queries.

Existing solutions for these challenges can be grouped in five categories: (*i*) *Buffer-based approaches* [15, 22, 33, 40, 48] temporarily

[1]This can be achieved via well-known distributed systems techniques, e.g., [25].

store updates in a buffer, and sort them before insertion in the database; query accuracy cannot be guaranteed on updates at the buffer boundaries. Buffering also increases latency, which conflicts with real-time analytics. (*ii*) In *punctuation-based methods* [2, 31, 32] a special markers (or "punctuation") in a stream indicates that no ooo update follows. This also incurs delays as the system waits for the marker. (*iii*) *Speculation-based approaches* [6, 7, 41] optimistically assume all updates are in order, and processes queries and analytics on the resulting graph. However, when ooo updates are detected, rollbacks are needed. This invalidates previous query results, and complicates processing. (*iv*) *Approximation-based systems* [1, 9, 49] provide bounded-error estimates of numerical queries. Instead, we focus on precise graph operations, e.g., shortest paths, which need an accurate snapshot. (*v*) A naïve approach for handling ooo updates in dynamic graphs would *keep all edges in a stream-time ordered list*. However, this conflicts with many graph operations' data access patterns. For instance, graph traversals need random vertex access by their ID, then sequential scan over node neighborhoods; this incurs re-sorting nodes. Also, maintaining stream time order after ooo updates would significantly impact the throughput.

To address the above challenges, we introduce **HAL**, a novel in-memory dynamic graph store with multi-version concurrency control protocol (MVCC).

We make the following contributions:

- HAL is based on a novel data structure called Stream-time Sorted Adjacency List (STAL, in short), in which we efficiently store in-order and ooo updates, while also providing fast data access for graph analytics.
- We propose a novel data structure which we use to identify the most likely insertion-deletion operation pairs over the same edge, and integrate it fully within our store, even after garbage collection.
- Our experiments show that HAL strikes a good balance between high transactional write throughput and efficient analytical queries. Compared to prior systems (which do not handle ooo updates), HAL's throughput is up to 73× higher, while its advantage on analytics goes up to 357×.

Below, we outline related work, then clarify our assumptions in Section 3, before describing our proposed in-memory graph store, named HAL (Section 4), together with its associated algorithms (Section 5 to Section 8). 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 concluding.

2 RELATED WORK

Dynamic graph database systems [11, 14, 23, 28, 53] that enable efficient edge scans for analytics, and facilitate single-edge insertions, can be seen as two groups: those that provide transactional guarantees, such as LiveGraph [53], Teseo [28], Sortledton [14], Spruce [47], and those that do not, such as GraphOne [23], Stinger [11] and Llama [38].

Our work belongs to the former group, in which we further identify two main designs. **LiveGraph** stores graph edge entries (insertions and deletions) in *adjacency lists, one for each source vertex, stored in the order in which they are received*; it supports random vertex access. The edges in each adjacency list are stored contiguously, thus reading them does not cause random accesses. For each edge update, LiveGraph stores the transaction timestamp and also possibly the invalidation timestamp. In contrast, **Sortledton** and **Teseo** store edge updates in fixed-sized *blocks*; Teseo stores the blocks in a B+ tree, and Sortledton in a skiplist. Within each block, and across the B+ tree, respectively, skip list for each source vertex, *edge updates are sorted by destination IDs*. Versions are maintained using the Hyper [42] protocol: both systems store *the latest version* of each edge in a sequential block, while older versions are stored in overflow linked lists; scanning these leads to more random accesses, thus cache misses. Spruce maintains two blocks per source vertex: a buffer block and sorted block. The buffer block contains a fixed number of unsorted edges (64 edges); when it fills, all the edges it stored are merged into a single block sorted by destination IDs. Hence, fewer cache misses occur when reading the sorted adjacency list block for analytics. However, frequent merges needed to maintain one adjacency list per source vertex is costly. The Sortledton, Teseo, and Spruce stores do not preserve the update arrival order; extending them to support out-of-order updates would require a complete redesign.

System	Edge insertion	Edge deletion	Edge look-up
Llama, Stinger	$O(\|E\|)$	$O(\|E\|)$	$O(\|E\|)$
GraphOne	$O(1)$	$O(\|E\|)$	$O(\|E\|)$
LiveGraph	$O(1)$	$O(\|E\|)$	$O(1)$
Teseo, Sortledton, HAL out-of-order	$O(log(\|E\|))$	$O(log(\|E\|))$	$O(log(\|E\|))$
Spruce	$O(\|E\|)$	$O(log(\|E\|))$	$O(log(\|E\|))$
HAL in-order	$O(1)$	$O(1)$	$O(1)$

Table 1: Complexity comparison across systems

In non-transactional stores, **GraphOne** inserts edges in a *write store*, which is a circular buffer; when full, edge updates are sent to a separate *read store*, where they are sorted by vertex ID and stored together with multiple snapshots of the graph. **Stinger** uses an adjacency list design; edges are added to small blocks in the order of their arrival. Single writes and reads are thread-safe, but graph scans are not, if concurrent with updates [11].

Llama batches updates into a *write store* and periodically moves them to the *read store*, a multi-version compact CSR, supporting sequential access to vertices and their neighborhood [44]. Each version of CSR calls it a snapshot; the authors [38] suggest creating one every 10s. Table 1 shows the time complexity of 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 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 positives occur, the verification raises the cost to $O(|E|)$. Spruce takes $O(log(|E|))$ for existence checks and deletions; however, for insertion, it takes $O(|E|)$ due to the merging cost from buffer block to sorted block. GraphOne inserts an edge in $O(1)$ without checking for existence; deletion and lookup incur a linear search. Stinger and Llama incur $O(|E|)$ for all three operations because of searches in the adjacency list.

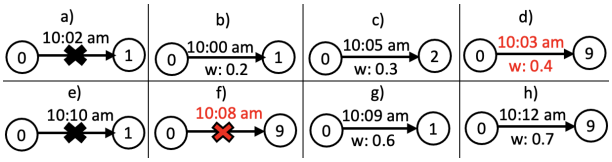

Figure 2: Sample dynamic graph.

Our system, HAL, performs in-order insertions in $O(1)$, better than the state-of-the-art; ooo insertions take $O(log|E|)$. For deletions and edge search: on in-order updates, HAL has better complexity than the existing systems; on ooo updates, its complexity matches that of Sortledton and Teseo, and avoids LiveGraph's worst case (false positives). Existing systems do not support ooo updates, which HAL is built to support, all the while improving performance on the operations also supported by other stores (Section 9).

3 PRELIMINARIES

Data model We consider a directed graph G, which at any point in time consists of a set of nodes N and edges E. Each edge has a source and destination vertex from N. Each edge, and each node, may have attributes (or properties). Similar to other systems [14, 28, 53], we **store the graph topology separately** from node and/or edge properties, and, in this work, we focus on efficiently handling **updates and queries on the graph topology**. For all such systems, data access paths based on node/edge properties can be added and optimized in a complementary way.

Out-of-order update An update can be judged in-order or out-of order (ooo) only at the database site. Specifically, update u with stream time S_T, received at the database at time W_T, is ooo if (i) u is a deletion and no insertion of the same edge has been received, and/or (ii) the database has already received at a time $W'_T<W_T$ an update with stream time $S'_T>S_T$.

Best-guess associated update At the database, insertions and deletions emitted from multiple streams are possibly received ooo. To build our current view of the graph, at any time, our system tries to **guesses** the associations between insertions and deletions of the same edge. Specifically, leveraging update stream times, to each insertion (respectively, deletion), we either:

- *associate the most likely deletion* (respectively, *insertion*); or
- *consider that none of the deletions* (respectively, *insertions*) *received so far is associated to this insertion* (respectively, *deletion*). In this case, the "missing" operation is either delayed (we will receive it later), or may never be emitted, e.g., some insertions are never followed by deletions.

When, for a given edge, more than one ooo update is received, *our best-guess associations between insertions and deletions may change.* We say then, that an ooo update may steal its associated update, from another. Sections 5 and 6 detail this.

4 STORAGE SYSTEM OVERVIEW

In this section, we present a high-level overview of our novel **History Adjacency List** (**HAL**, in short) data store. We opted for an *adjacency list design* for our graph storage because it offers a good balance between data access locality, which is important for queries, and high ingestion throughput, which is important for high transactional throughput [14].

The HAL design layout resembles the one of LiveGraph [53], which preserves the arrival order of updates (as detailed in Sec. 2). The HAL layout innovates on two areas:

Five layout optimizations: (i) the arrival order of the adjacency list is maintained based on the stream time; (ii) multiple fixed-size adjacency list blocks are proposed rather than a single adjacency list, thereby reducing costly resize requests; (iii) edge entry metadata is stored separately from the destination IDs; this allows faster access to destination ID scans, which are predominant in analytical queries, thus improving analytical scan performance (details of (i),(ii),(iii)) are in Sec. 4.2); (iv) we propose a novel strategy inspired by cracking [18] to minimize the data access needed to perform a per-edge deletion check during analytical scans concurrent with updates (we detail this in Sec. 7); (v) garbage collection is performed using writer threads instead of background threads. This approach reduces contention between the background threads and reader/writer threads, leading to better memory management (as we detail in Sec. 6.1).

Most likely insertion and deletion association strategy: We use a secondary index (hash table) that efficiently and accurately associates the most likely insertion and deletion entries in the adjacency list, leveraging their stream time. It can dynamically adjust these associations if multiple ooo updates are received for the same edge, thereby ensuring correct ordering of the adjacency list by stream time. We detail this in Sec. 4.1.

Throughout our discussion, an edge of the form $s \rightarrow d$ connects a *source* node with a *destination* node, called simply source and destination below. An *update* (or *entry*) is an insertion or deletion of an edge. To avoid confusions, we call *stream* an individual source of graph updates; *stream time* refers to the time local to that stream.

HAL stores and processes updates in an *adjacency list*, sorted by the source of the inserted/deleted edges; the list is the entry point for any read and write. HAL is *append-only*: insertions and deletions only add data to the store. To keep its size under control, however, HAL content can be *garbage collected* (as we will discuss in Sec. 6.1). HAL aims to *maximize efficiency for in-order updates*, which we expect to be more frequent than ooo ones; if and when needed, dedicated fields are created within HAL to store and exploit ooo updates efficiently. Finally, HAL aims to support graph analytics concurrently with updates from multiple streams.

HAL has two main structures: a *vertex array* (VA, in short), and a *stream time-sorted adjacency list* (STAL, in short), each holding an entry for each graph vertex, denoted, respectively, VA[s] and STAL$_s$. Figure 2 illustrates successive states, labeled a) to h), of a **sample graph**. Each edge depicts an update; if it is crossed, it is a deletion, otherwise, an insertion. On each edge we show its stream time, and an edge property, e.g., its weight w. On ooo edges, the stream time is shown in red; the cross is also red for ooo deletions. We describe the VA, respectively, STAL in Sec. 4.1 and 4.2.

4.1 The vertex array (VA)

Each **vertex array (VA)** entry VA[s] consists of several fields. Figure 3 illustrates it for the source vertex s having the ID 0. In the figure, a dash (–) designates an empty field (no known value), whereas "..." values or fields whose details we omit, for simplicity.

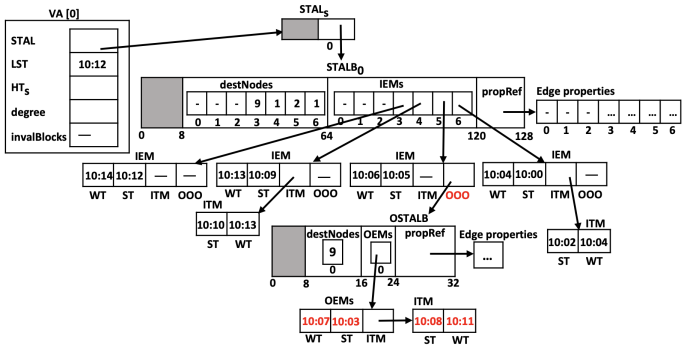

Figure 3: Sample vertex entry VA[0].

Finally, shaded areas denote metadata (of constant size), which we will introduce as needed, to explain our algorithms.

(1) A reference to the STAL entry corresponding to this vertex;
(2) The **latest stream time** (**LST**, in short) of an in-order update received so far for the source vertex s. For instance, in Figure 2, the latest in-order insertion with source 0 is 0⟶9 at 10:12 in h); hence, in Figure 3, VA[0] has 10:12 as LST.
(3) **degree**: the number of edges inserted but not deleted, currently in STAL_s;
(4) **invalBlocks** references STALBs (see below) with edges whose source is s, and which should be garbage-collected (cf. Sec. 6.1).
(5) A **hash table** (HT_s, in short) whose keys are the destination vertices d for which we have received some s⟶d updates, and whose values we detail below;

Figure 4 illustrates the evolution of HT_0. Each yellow area encloses the data structures created as a consequence of the respective entry in Figure 2; dashed arrows trace the evolutions data structures go through with each step, while solid arrows show references among data structures. For a given destination d, HT_s stores, as the graph evolves, *one among the following three data structures.*

Update position and indicator (**UPI**, in short): The UPI of s⟶d, denoted UPI[s, d] encodes the position of the *latest (in-order or ooo) s⟶d insertion* in STAL_s. In Figure 2, the first insertion for the edge 0⟶2, at c), is in-order, with stream time 10:05. Thus, we create an in-order UPI for destination 2 in HT_0 with this stream time, as shown in Figure 4. The dark grey UPI fields in this figure denote the information *encoding the position of the insertion within STAL_s*; this encoding is not immediate, because the block holding an insertion may grow or move, as the graph changes, yet the UPI always provides constant-time access to the insertion. UPIs and their decoding are detailed in Sec. 8.

Staging area (**AR**, in short): The AR for an edge s⟶d stores *insertions (deletions) of this edge, waiting for their corresponding deletions (insertions).* For instance, in Figure 4, when we receive a deletion of 0⟶1 at a), HT_0 does not hold a corresponding insertion; we create the staging area for 0⟶1, with an *update deletion block* UD_a, storing the stream time 10:02 of this deletion. Then, we store the AR at position 1 in HT_0. When the deletion f) of 0⟶9 arrives at 10:08, we find the corresponding insertion in HT_0 at index 3, and replace the ooo UPI of 0⟶9 with a corresponding AR. Details of AR processing are delegated to Sec. 5 and 6.

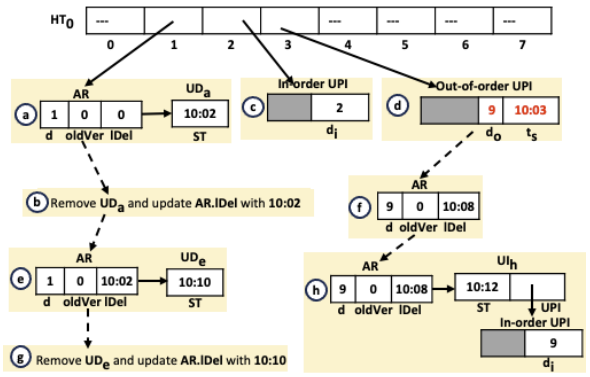

Figure 4: HT_0 through insertions and deletions.

Last garbage collected deletion (**LGCD**, in short) for vertex s is the stream time of the most recent deletion of s⟶d that has been garbage-collected (Sec. 6.1).

$\text{HT}_s(d)$ through the lifecycle of the edge The roles of these different data structures can be understood by examining the *lifecycle* of an edge s⟶d in our system.

- *Only when the first update ever to be received by the database for s⟶d is an insertion*, we create a UPI, as we did above for the insertion 0⟶2 in Figure 2 c).
- An AR exists as long as we have *one or more insertion(s) (respectively, deletion(s)) for which we did not yet receive corresponding deletion(s) (respectively, insertion(s)).* The operation we received *waits* for the one not yet received, in the AR. This concerns: in- or ooo insertions, for which we have not received a deletion; in the future, we may receive such a deletion, or not; and, every deletion for which the current STAL does not contain a possible associated insertion. This happens when all the insertions prior to our deletion, have already been marked as deleted (by other deletion entries).
- An LGCD is created when updates s⟶d have been garbage collected (Sec. 6.1).

VA ensures constant-time access to edges by their source vertex s, for one or multiple parallel threads.

4.2 The stream-time ordered adjacency list (STAL)

For each source vertex s, STAL_s is an append-only list of blocks, storing updates going from s to various destination vertices. STAL_s has some **metadata** (shaded area in Figure 3), notably: a **hasDeletes** flag which is true if deletions have been received for source vertex s, and **delNo** the number of slots freed by successive deletions. STAL_s also stores the **stream-time ordered adjacency list** (**STAL**, in short), which references one or more STAL blocks. Each **STAL block (STALB)** comprises:

(1) **metadata** (shaded in Figure 3), notably the boolean flags **hasDeletes** and **hasOOO**, indicating whether the STALB contains, respectively, deletions, or ooo updates, and the number of deleted entries in the STALB;

(2) **destNodes** holds $s \rightarrow d$ entries for the given s and various destinations d, sorted in descending stream time order.
(3) **IEMs** stores references to **in-order edge entry metadata** (**IEM** in short, see below) entries, one for each edge update in destNodes;
(4) **propRef** references a vector of the same size as destNodes and IEMs, storing, for each of the above edge updates, the properties that the edge may have, e.g., edge weight, etc. *Note that the edge properties vectors are not part of $STAL_s$, and actually not even part of our HAL structure*; we shown them in the figure to provide the reader with a global picture.

For instance, in Figure 2, we receive insertions of edges $0 \rightarrow 1$, $0 \rightarrow 2$, and $0 \rightarrow 9$ in steps b), c), d), g), h); in Figure 3, the destination ids of these updates are stored in destNodes, and the associated metadata in IEMs. For instance, Figure 3 illustrates the STALB labeled S_0, over 128 bytes: 8 bytes for metadata (at the left in gray color), 8 bytes for edge properties (at the right), 56 bytes (7 entries) for destNodes and 56 bytes (7 entries) for IEMs.

Importantly, **STALBs within $STAL_s$ are sorted by stream time in descending order**: let S_0, S_1 be two STALBs such that S_0 appears in $STAL_s$ *before* S_1. Then, any entry in S_0 is more recent (has a higher stream time) than any entry in S_1.

Each **IEM** **(in-order entry metadata)** contains:

(1) The **write (transaction) time** (**WT**), that is, the time when the entry is received at the database site;
(2) The **stream time** (**ST**) when the entry is emitted by its stream;
(3) The **invalidation time metadata (ITM)** of an insertion entry stores information about *the edge deletion most likely associated to this insertion* (recall Sec. 3), if one exists:
   - The stream time S_T of the deletion entry;
   - The transaction time W_T of the deletion entry;
(4) The **ooo updates (OOO)** field, initially empty, references a data structure storing ooo insertions *with the same source vertex, and the same or different destination* as the one for of the IEM, whose stream time is less than the IEM's ST, but higher than the STs of the previous IEM (if any). The data structure will be detailed in Sec. 5.1. We store ooo insertions in the same ST order as the in-order ones, but not at the same level. The reason is that in-order insertions are simply appended in the block holding the most recent insertions (thus, in constant time). To store an ooo insertion's destination and metadata in $STAL_s$ in the order dictated by its ST, some previously received in-order information may need being recopied, a cost we avoid.

For each arriving *in-order insertion*, we create an IEM; for each *ooo insertion*, we create an **ooo entry metadata (OEM)**, resembling an IEM, but without the OOO field. For each *deletion*, we create an ITM, and reference it from the IEM or OEM of the insertion so far received, that is most likely associated to this deletion. Recall from Figure 2, the insertions and deletions of edges $0 \rightarrow 1$, $0 \rightarrow 2$, and $0 \rightarrow 9$; Figure 3 shows the corresponding data structures created in $STAL_0$. The ITM with ST 10:10 referenced by the IEM with ST 10:09 shows that we pair the deletion of $0 \rightarrow 1$ at 10:10 in Figure 2 e) with the insertion at 10:09 in g). The ITM with ST 10:08 referenced by the OEM with ST 10:03 means that for the deletion of $0 \rightarrow 9$ at 10:08 in f), the most likely insertion is at 10:03 in Figure 2 d).

Remark. In STAL, *a deletion is only stored as an invalidation (ITM) of its most likely insertion*. If, upon receiving the deletion, we cannot find such an insertion, the deletion enters the staging area AR, but is not visible in STAL.

The STAL structure ensures the update entries for a given source vertex s are stored *in the descending order of their stream time*. As we will discuss, this ensures $O(1)$ complexity for the operations we expect to be most common: in-order insertions and deletions. Data structures within STAL, in particular the AR, IEMs, OEMs, and ITMs, enable us to provide efficient graph querying with snapshot isolation guarantees even in the presence of ooo updates.

In Sec. 5 and 6, we provide details of the HAL's internal operations (insertion and deletion), while Sec. 7 discusses the concurrent execution of analytical queries and updates.

5 INSERTION

A new edge (insertion) entry $s \rightarrow d$, called in this section **the new entry**, arrives at the database site at transaction time W_T, having the originating stream time S_T. We process it as follows:

(1) *Locate* in the VA the position corresponding to s, e.g., if the $0 \rightarrow 1$ arrives at 10:00 am (W_T), this is the VA[0].
(2) *Lock* this VA entry to prevent conflicts between several write-transactions having the source vertex s.
(3) *Compare* the new entry's S_T to the LST value in VA[s].
   (a) If LST<S_T, then $s \rightarrow d$ is an *in-order* update. Section 5.1 explains how we handle in-order insertions.
   (b) Otherwise, it is an *out-of-order* update; Section 5.2 describes how to handle these.
(4) *Unlock* VA[s].

At step (3) above, we consider the new entry in-order or out-of-order by comparing its stream time S_T "only" with the last insertion time in VA[s], while our definition of ooo updates (Section 3), more broad, would require comparing S_T with the highest stream time of any operation (insertion or deletion) received so far at the database, regardless of the vertices involved. The condition we test is sufficient to ensure that *at any point, for each source vertex, we can return all its outgoing edges, correctly reflecting all updates received so far, in- or out-of-order*. Indeed, given that our store is organized by the source vertex, our test suffices to place the new insertion at the right place (dictated by S_T) among all insertions with the same source. Further, any deletions of edges with source vertex s also accumulate in $STAL_s$ and thus will meet our new entry there, ensuring a correct view on our store at any point.

5.1 In-order insertion

Since $STAL_s$ stores most recent entries first, we add the new entry to its first STALB. If this STALB is full, its size doubles, up to reaching a system-wide upper bound **maxBlockSize**. When a STALB has reached this size, the STAL grows by *doubling* the number of blocks, from the initial block to 2, 4 etc. We detail the process since it is important for efficiently finding insertions in the STAL (Sec. 8.) In each doubled-up $STAL_s$, the full STALBs make up the second half of the list. Thus, if we denote full STALBs by gray-filled boxes and blocks with free space by white-filled ones, $STAL_s \rightarrow$ B_0 becomes $STAL_s \rightarrow$ B_1 $\rightarrow$ B_0 and then B_3 $\rightarrow$ B_2 $\rightarrow$ B_1 $\rightarrow$ B_0;

here, we store new entries in B_2, the new STALB closest (in ST order) to B_1; STAL_s references the block holding the most recent insertion, thus $\text{STAL}_s \rightarrow \boxed{B_2}$. When B_2 fills, this becomes $\text{STAL}_s \rightarrow \boxed{B_3} \rightarrow \boxed{B_2} \rightarrow \boxed{B_1} \rightarrow \boxed{B_0}$, etc.

We denote by $\boxed{\text{STALB}_e}$ the STALB where the new entry is stored, by $\boxed{d_e}$ the new entry's position within STALB_e's entries, and by $\boxed{\text{IEM}_e}$ the in-order edge metadata (IEM) created to reflect this insertion (recall Figure 3), storing S_T and W_T.

Next, to ensure constant-time access to the newly inserted entry, we create the **Update and Position Indicator (UPI)** for it, call it $\boxed{\text{UPI}_e}$, and insert it into HT_s. The UPI stores *information based on which e can be located in STAL_s in the future, even if STALB_e grows, is resized, or moved away, potentially many times, from the head of STAL_s*. Specifically, UPI_e stores: the position of the new entry in STALB_e; the position of STALB_e in STAL_s; and the sizes of STALB_e, and STAL_s, when the new entry was inserted.

When processing our insertion, four possibilities arise, depending on the previously received updates for $s \rightarrow d$:

- HT_s contains *no entry* for $s \rightarrow d$ (the new entry is the first arrived for this edge): the above steps suffice.
- HT_s contains an UPI for an older insertion $s \rightarrow d$, call it $\boxed{\text{UPI}_o}$. Between the insertions described by UPI_o and UPI_e, a deletion, not received yet, may or may not have been sent. To prepare the possible future processing of that ooo deletion, and to enable correct query answering before and after receiving it, UPI_o *must be marked as a previous version of UPI_e*; Sec. 5.1.1 details this.
- HT_s contains a *staging area (AR)* (recall Sec. 4) for $s \rightarrow d$; inserting the new entry is discussed in Sec. 5.1.2.
- HT_s contains a *latest garbage collected deletion (LGCD)*; we describe the associated processing in Sec. 5.1.3.

5.1.1 In-order insertion over existing UPI. Inserting over a UPI means an out-of-order deletion may arrive. We need to update UPI_o to indicate *it is the previous version of* UPI_e, and to mark its entry as deleted.

(1) From UPI_o, we locate the previous $s \rightarrow d$ insertion entry, call it e_o: block STALB_o, holding e_o at position p.
(2) At position p in STALB_o's metadata array IEMs, we find the entry metadata, call it EM_o, associated to e_o: EM_o is an IEM if e_o was in-order, and an OEM otherwise.
(3) To record e_o as deleted (by the update not yet received), we create an Invalidation Time Metadata (ITM) initialized with S_T and W_T, and the ITM to the EM_o metadata of UPI_o.
(4) In STALB_o's metadata, we set hasDeletes to true.

Receiving an insertion when the previous operation was also an insertion, means that both of them need to wait for their possible deletions, in a staging area. In details:

- We create a new AR, call it AR_e with: the destination id d, the latest deletion time (lDel) set to 0 (such a deletion has not arrived yet), oldVer set to 0 (no garbage collection has claimed a version of $s \rightarrow d$ so far).
- We create two update insertion blocks UI_o and UI_e, for the previous and new entry, with the ST (stream time) from the corresponding metadata: EM_o and IEM_e. We chain the blocks in AR_o in increasing ST order: $\text{UI}_o \xrightarrow{\text{next}} \text{UI}_e$.
- AR_e replaces UPI_o as the HT_s value for d.

Note that UI_o, UI_e reference UPI_o and UPI_e, which thus remain accessible. When a deletion arrives, first, we pair it with an insertion from AR, and then, use the UPI to locate the insertion entry in STAL to mark its deletion (Sec. 6.2).

5.1.2 In-order insertion over existing AR. Let $\boxed{\text{AR}_o}$ be the AR for $s \rightarrow d$, stored in STAL_s when the new entry arrives. We first check whether a deletion entry, emitted after e, has already been received at the database. For that, we compare the new entry stream time S_T with AR_o.lDel, the stream time of the last deleted $s \rightarrow d$ update (recall Sec. 4). For brevity, we just denote it $\boxed{\text{lDel}}$ below.

(1) If $S_T < \text{lDel}$, a deletion of $s \rightarrow d$ has already been received, following (in stream time order) the new (insertion) entry we are processing now.
(2) If $S_T \geq \text{lDel}$, then STAL_s has no record of a deletion following e. However, *deletion which we cannot pair with insertions when they arrive, are only recorded in the AR, and not in STAL_s* (as will be detailed in Sec. 6). Thus, the staging area may contain an update deletion block UD_o whose stream time is above S_T.
   (a) If such an UD_o does not exist, the new entry should be added to the store.
   (b) If UD_o exists, it deletes the new entry e.

In cases (1) and (2b), we **mark the new entry e as deleted** in STAL_s, much like in Sec. 5.1.1, except that the AR is already created; we also increase lDel to S_T, reflecting the most recent deletion. Further, in case (2b), we remove UD_o from the AR.

In case (2a), we **create a new update insertion block** UI_e, and insert it in the AR at the right place, based on its S_T. It may happen that UI_e is preceded, in the AR, by a prior insertion materialized by a UI_{prev} block. In that case, the UPI corresponding to UI_{prev} is marked in STAL_s as a previous version of UPI_e, just like we did for UPI_o in Sec. 5.1.1.

5.1.3 In-order insertion after GC. Garbage collection (GC) empties the staging area, replacing it with an LGCD. Thus, the first entry after GC needs to restart a staging area. We create a new update insertion block UI_e with S_T and UPI_e; we create the staging area AR_e whose oldVer is copied from the LGCD, and insert UI_e as the next block after AR_e. Finally, AR_e replaces the LGCD in HT_s for destination d. **The complexity** of handling an in-order insertion is $O(1)$, since each step takes only constant time.

5.2 Out-of-order insertion

To propagate the out-of-order insertion e **in STAL_s**, we create an out-of-order edge entry metadata (OEM) block $\boxed{\text{OEM}_e}$ with S_T as source time, W_T as transaction time. Then, based on S_T, we search in STAL_s for the out-of-order STAL block, call it $\boxed{\text{OSTALB}_e}$, where the new entry e fits; our search will first visit one or more STAL blocks (recall Figure 3). (*i*) If OSTALB_e does not exist, we create it with size 1, and connect it to the proper STALB, whose flag hasOOO is set to true. (*ii*) If OSTALB_e exists, if it is full, we double it up

	STAL	AR_o (before)	AR_o (after)
1	i_1	UD_2	No entry
2	i_1 i_3 i_5	$UI_1 \xrightarrow{next} UI_5$	$UI_1 \xrightarrow{next} UI_3 \xrightarrow{next} UI_5$
3	i_1 d_2 i_3 i_5	UI_5	$UI_3 \xrightarrow{next} UI_5$
4	i_1 i_3 i_5	$UI_3 \xrightarrow{next} UI_5$	$UI_1 \xrightarrow{next} UI_3 \xrightarrow{next} UI_5$
5	i_1 i_3 d_4	UD_6	$UI_1 \xrightarrow{next} UD_6$
6	i_1 i_3	$UD_2 \xrightarrow{next} UI_3$	UI_3
7	i_1 i_3 d_4	UD_6	$UI_1 \xrightarrow{next} UD_6$
8	i_1 i_3 d_4	No entry	UI_1
9	i_1 d_2 i_3 i_5 d_6	No entry	UI_3
10	i_1 i_3 i_5 d_6	UI_1	$UI_1 \xrightarrow{next} UI_3$

Table 2: Possible cases of OOO insertion over existing AR.

like we did for STAL blocks, except that upon reaching a maximum OSTALB size, $OSTALB_e$ *splits* in two smaller blocks, which become the two initial leaf nodes in an Adaptive Radix Tree [27] $\boxed{\text{ART}}$, ordered according to update stream time. From now on, out-of-order updates in $STAL_s$ will be stored in the ART. We add d in $OSTALB_e$'s destNodes, and OEMe in its OEMs.

Next, we reflect the insertion **in HT_s**, through a new out-of-order UPI, denoted $\boxed{UPI_e}$, initialized with S_T and d. As in Section 5.1, there are four possibilities, depending on what HT_s stores for d when our insertion arrives:

- *No entry* (e is the first $s \rightarrow d$ insertion ever arrived): we add UPI_e to the hash table HT_s and the processing of HT_s is finished.
- An *UPI* for a previous $s \rightarrow d$ insertion, call it $\boxed{UPI_o}$. An ooo deletion, not yet received, may or may not have been sent between the insertions described by UPI_o and UPI_e. We discuss this case in Section 5.2.1.
- An *AR* for $s \rightarrow d$; this means some insertions and/or deletions of $s \rightarrow d$ are waiting for their matching deletions (respectively, insertions). We detail our algorithm in Section 5.2.2.
- An *LGDV* (latest garbage-collected destination version); we describe the associated processing in Section 5.2.3.

5.2.1 Out-of-order insertion over existing UPI. If S_T is above UPI_o's stream time (e is emitted after the previously received insertion), we mark UPI_o as the previous version of UPI_e in $STAL_s$, as we did for UPI_o in Section 5.1. Otherwise, we mark UPI_e as a previous version of UPI_o. Then, we create a new staging area AR_e, where UPI_o and UPI_e wait for their possible, associated deletions. We create two update insertion blocks UI_o, UI_e and, depending on their stream times, we chain them as $UI_o \xrightarrow{next} UI_e$ or $UI_e \xrightarrow{next} UI_o$ in AR_e. Then, AR_e replaces UPI_e as the HT_s value for d.

5.2.2 Out-of-order insertion over existing AR. Let $\boxed{AR_o}$ be the AR for $s \rightarrow d$. Our new, ooo entry e may be older than the most recent garbage-collected $s \rightarrow d$ entry; we test this by comparing S_T with AR_o.oldVer. If S_T is smaller, e had already been *assumed* (even before it was received), *and garbage-collected* together with a subsequent deletion entry. In this case, we ignore e.

Our next steps depend on the entries (deletions and/or insertions of $s \rightarrow d$) in $STAL_s$ and AR_o. We illustrate our analysis with simple examples in Table 2, showing, for each case, the updates in STAL for $s \rightarrow d$, as well as AR_o before and after the insertion. We show each STAL insertion as i and deletion as d, with their stream time as an index, e.g., i_3 is an insertion with S_T=3. Our new (ooo) entry appears in red. Each AR block is also indexed with its stream time. We use small, consecutive times for illustration only.

We check whether a deletion entry, that would be most likely associated to e, has already been received.

If $S_T \geq AR_o$.lDel, the stream time of the last $s \rightarrow d$ deletion, such a deletion is not in $STAL_s$. However, it may be in AR_o (recall the **Remark** at the end of Section 4.2), under the form of a deletion update block, call it $\boxed{UD_o}$, with an ST above the new entry's S_T.

- If UD_o exists, we mark e in STAL as deleted by UD_o (through an ITM referenced from OEM_e). We remove UD_o from AR_o (case 1 Table 2).
- Otherwise, it is the new insertion who joins the staging area AR_o: we create a new update insertion block $\boxed{UI_e}$, and insert it in AR_o according to its stream time S_T. We look in STAL for two insertion entries whose ST are *the closest, before and after* S_T: call them i_{prev}, respectively, i_{next}.
  - If i_{prev} exists but has no deletion in STAL, we mark i_{prev} as the previous version of e (case 2 in Table 2).
  - If i_{prev} exists and it has a deletion in STAL, there is nothing more to do: e does not change our current most likely insertion-deletion pairs. This is case 3 in Table 2.
  - If e_{next} exists, we mark e as the previous version of e_{next}. This is case 4 in Table 2.

If $S_T < AR_o$.lDel, the deletion for e may be either in STAL or in AR_o. In STAL, we search for i_{prev} and i_{next} as above.

- If a deletion UD_o waits for the new entry in AR_o:
  - If i_{prev} exists, it has a deletion d_{prev} in STAL with a stream time above S_T, we mark e as deleted by d_{prev}, and we mark i_{prev} as the previous version of e. *Because d_{prev} is now associated to e, the previous entry i_{prev} "has lost its most likely deletion", and must wait for the possible future arrival of its deletion*: we create an update insertion block UI_{prev} with i_{prev}'s stream time, and insert it AR_o. This is illustrated in case 5 in Table 2.
  - If i_{next} does not exist, or it exists and its stream time is above that of UD_o, this indicates the waiting deletion UD_o should be paired with e. We remove UD_o from AR_o and mark e as deleted by UD_o. This is case 6 in Table 2.
  - If i_{next} exists and its stream time is below that of UD_o, the waiting deletion UD_o is closer to i_{next} than to the new entry. We mark e as a previous version of the e_{next}, and make it wait by creating a new UI_e with stream time S_T, and storing it in AR_o. Case 7 in Table 2 illustrates this.
- If UD_o does not exist (no deletion was waiting for e in AR_o):
  - If i_{prev} exists, and it has a deletion d_{prev} in STAL with a stream time above S_T, the new entry *steals* the d_{prev}, and we mark i_{prev} as the previous version of e. Now, i_{prev} must wait: we create UPI_{prev} and insert it in AR_o at the proper place. This is illustrated in case 8 in Table 2.
  - If i_{prev} exists, and it has a deletion with stream time below S_T, then e must wait for its possible own (later) deletion,

which may or may not have been emitted by some source: we store a new UI_e in the AR. This is case 9 in Table 2.
- If i_{prev} exists but it has no deletion in STAL, we mark i_{prev} as the previous version of e. We store a new UI_e for the new entry in the AR. Case 10 in Table 2 illustrates this.
- If i_{next} exists, we mark the new entry as a previous version of e_{next}. This is also shown in case 10 in Table 2.

5.2.3 Out-of-order insertion after GC. The process here is as for in-order insertions (Section 5.1.3): we reopen the staging area for the new entry to wait there.

The complexity of an ooo insertion is $O(log(|E|))$, in case we need to search in AR_o (sorted by ST).

6 DELETION

We now consider a deletion entry $\boxed{e}$ of the edge $s \rightarrow d$ with the stream time $\boxed{S_T}$ and received at transaction time W_T.

We lock VA[s] to prevent conflicts with other updates having the source vertex s. Next, garbage collection (described in Sec. 6.1) may trigger. Then, depending on what $STAL_s$ holds for d:

No entry : Start a staging area AR_e in which the deletion waits, under the form of an update deletion block UD_e with stream time S_T. Store AR_e in HT_s at position d.

LGCD : Reopen the staging area AR_e in which we make the new deletion wait, as above. Copy the LGCD into AR_e.oldVer, then store AR_e in HT_s.

AR : If S_T is smaller than AR_o.oldVer, e is in the past with respect to insertions and deletions already garbage-collected (e had already been assumed); there is nothing left to do.

If $S_T \geq AR_o$.lDel, we search AR_o for an insertion that might have been waiting for e. This insertion block, call it $\boxed{UI_{prev}}$, should have an ST lower than that of e, but higher than AR_o.lDel: an insertion with an ST behind lDel is followed first, by a deletion at lDel, and only later by e (thus, such an insertion cannot be paired with e).

- If UI_{prev} exists, e deletes its corresponding insertion. From UI_{prev}, we access its UPI, call it UPI_{prev}, which may denote an in-order or ooo insertion. We mark UPI_{prev} as deleted in the STAL, as shown in Sec. 6.2 if UPI_{prev} is in order, or as in Sec. 6.3 if it is ooo. Having paired UI_{prev} with a deletion, we remove it from AR_o. We set AR_o.lDel to S_T.
- Otherwise, we create an update deletion block UD_e, with stream time S_T, and insert it in AR_o.

If $S_T < AR_o$.lDel, we search $STAL_s$ for the closest insertion entry i_{prev}, whose stream time precedes S_T.

- If i_{prev} exists, and it has an associated deletion d_{prev}:
  - If $d_{prev} > S_T$, the current deletion e is closer to i_{prev} than its associated deletion was. Thus, e *steals* i_{prev}, and d_{prev} (re)joins the staging area, to wait for its insertion. We create an update deletion block UD_{prev} with the stream time of d_{prev}, insert it in AR_o, and mark the i_{prev} as deleted by the new entry e.
  - Otherwise, e is certainly not the deletion of i_{prev}, given that d_{prev} was emitted before e. Thus, e needs to wait for its insertion. We insert in AR_o a new update deletion block for e, with stream time S_T.
- If i_{prev} exists, but it has no deletion, we associate e as the deletion of i_{prev}: mark i_{prev} as deleted by e at S_T, and remove UI_{prev} from AR_o, since it has met its deletion.
- If i_{prev} does not exist, e needs to wait for its insertion; we create an update deletion block with stream time S_T, and insert it in AR_o.

UPI Let $\boxed{UPI_o}$ be the previous UPI. To apply the deletion in $STAL_s$, we follow the steps in Sec. 6.2 if UPI_o is in order, respectively, in Sec. 6.3 if it is ooo. In both cases, we then create a staging area AR_e, where e waits for its insertion, and replace UPI_e with AR_e in the HT_s entry at position d.

After the deletion is completed, we unlock VA[s].

6.1 Garbage collection

For each block B referenced in VA[s].invalBlocks, we compare the block's WT to a system-wide **least recent arrival time among the currently running queries** $\boxed{LQ_T}$ to prevent garbage-collecting entries on which the query with LQ_T may still be working. LQ_T is initialized to 0 when creating the store and is set to Q_T, the least query time among the currently running queries in the database whenever a query arrives.

If $B.WT < LQ_T$, we resize B down to its useful half:

- Create a new block B' (STALB or OSTALB, like B), whose size is half that of B.
- Copy the B entries that are not deleted into B'.
- In the staging area $AR_{s,d}$ for s and d, set oldVer to be highest ST among all deletions of $s \rightarrow d$ in B.
- If $AR_{s,d}$ is empty, and its oldVer is the stream time of the most recent entry for $s \rightarrow d$ in the STAL,this means no entry for $s \rightarrow d$ is still waiting for its corresponding operation, and our store is in a stable state wrt this edge. In this case, replace $AR_{s,d}$ with an LGCD storing oldVer.
- Replace B with B' in $STAL_s$.

6.2 Deletion of an in-order insertion

Recall from Section 5.1 that an UPI created for an in-order insertion o stores information based on which we can locate the STAL block B holding the insertion, and that insertion's metadata within B, call it IEM_o. To handle the deletion of the in-order insertion:

- We create an ITM block, with the stream time S_T of the deletion, and the current W_T, and reference it from IEM_o.
- In B's metadata (Section 4.2), we set hasDeletes to true, and increase delNo.
- If delNo reaches 50% of B'size, *we mark B for (future) garbage collection*, by adding a reference to it, together with the current transaction time W_T, in VA[s].invaBlocks.

The complexity of the above steps is $O(1)$.

6.3 Deletion of an ooo insertion

Based on UPI_o, we find the OSTALB block B in $STAL_s$ hosting the ooo insertion; B may be referenced from a STALB, as shown in

Figure 3, or we may find it by traversing an ART (recall Section 5.2). UPI_o also stores the position p of the ooo insertion within B.

- We find OEM_o, the metadata for the insertion, in B.OEMs at position p.
- Create an ITM block ITM_e with S_T and W_T, and reference it from OEM_e, to indicate that the insertion has been deleted.
- Update B's by setting hasDeletes to true, and increment its delNo counter.
- If number of deleted entries in B reaches 50% of its size, we add a reference to B, with the transaction time W_T, in VA[s].invalBlocks, for future garbage collection.

The complexity is $O(log(|E|))$, due to searching in the ART.

7 QUERIES

Dynamic graph systems support a large variety of analytics. Popular algorithms well-known implementations, such as Breadth-First Search, PageRank, Community Detection, can be plugged on top of any in-memory graph store giving access to its edges. In this section, we show how HAL *finds all edges whose source vertex is s, at the current database time*, since this is a crucial operations in graph computations. Other data access methods, e.g., finding nodes or edges by values of their properties, can be supported orthogonally to this work, focused on storing and querying the graph structure.

We set $\boxed{LQ_T}$, the least recent query time among the currently running queries, at the current database time. Then, for each STALB in $STAL_s$, starting with the most recent:

(1) If the hasDeletes field of the STALB is true, **deletion checks are needed:** when traversing the metadata entries associated with this STALB, we will check each IEM entry's ITM, to ensure we do not return an edge already deleted. If hasDeletes is false, we can skip this check for all entries in the STALB, thus read it sequentially with no need for random memory accesses, which is more cache-friendly. As our experiments show (Section 9), competitor systems which check each traversed edge suffer of their poor cache use.
(2) If the hasOOO field of the STALB is true, **out-of-order checks are needed**: when traversing the STALB, we will check each OOO field, and include any ooo updates there, in the result. If hasOOO is false, the tests can be skipped for this block's entries.
(3) Traverse the STALB's IEMs and in parrallel destNodes (recall from Section 4 that these are same-size arrays filled in parallel). For each IEM entry m, at position pm in IEMs:
    (a) Check whether the transaction time m.WT is less than LQ_T. If this holds: if no deletion or ooo checks are needed, return destNodes[pm], the destination node corresponding to the position of m; if deletion checks are needed, only if m is not deleted (it has no ITM), return destNodes[pm]. The check helps ensuring snapshot isolation: we only read the (non-deleted) edges received and stored before LQ_T.
    (b) If ooo checks are needed, and m has out-of-order updates, its OOO field references an OSTALB or ART (Section 5.2). For each OEM entry in that structure, call it o, we check whether the transaction time o.WT is less than LQ_T. If this holds, and no deletion checks are needed, return o's destination node. If deletion checks are needed, check o's ITM field, to see whether o is deleted. If it is not, return its destination.

Reducing edge deletion checks As an optimization, once a query has identified an edge in HAL as deleted, *this is recorded in a bit of the space devoted to storing d* in the insertion block's destNodes (recall Figure 4). Thus, a future query learns just by reading destNodes that the edge is deleted, and avoids a random memory access to read the OEM or IEM of the insertion. This optimization is in the spirit of database cracking [18]. This can very significantly speed up analytical queries, as we show in Section 9.6.
The complexity is $O(|R|)$, where R is the result set of edges.

8 HT_s AND UPI DETAILS

As shown in Sections 5 and 6, our entry insertion and deletion algorithms repeatedly rely on finding, in HT_s, the entry (UPI, AR, or LGCD) for destination vertex d. Also, these algorithms need to find, based on UPI_e, created for a past insertion entry e, the location in $STAL_s$ where information about this insertion is stored, e.g., the properties of that edge, whether the edge has been deleted, etc. In this section, we detail how the HT, and UPIs, support these operations in constant time.

The HT is created with 2 entries, and doubled up whenever 50% of its entries are taken. It holds exactly one entry in each bucket, and uses the *open addressing* technique [36], as follows.

- When an UPI is created for the *in-order* insertion of $s \rightarrow d$, the UPI is stored in HT_s in the bucket (d modulo current size of HT_s). If this position is already taken, the UPI is stored in the first free position following the initial one.
- An *ooo* insertion of $s \rightarrow d$ is first stored in $STAL_s$ (Section 5.1). It is stored in ST order, attached to a previous, in-order $s \rightarrow x$ insertion, as illustrated in Figure 3 (note that $s \rightarrow x$ is guaranteed to exist: if $s \rightarrow d$ was the first insertion ever with source s, $s \rightarrow d$ would not be out of order). Then, the UPI created for the *ooo* insertion of $s \rightarrow d$ is inserted in HT_s *just after the position (x modulo current size of HT_s).* Again, if this position is taken, the first free position following it is used.
- The AR and/or LGCD replacing an UPI keep the UPI's position in HT_s.

Because of the above, when looking for the UPI corresponding to a given $s \rightarrow d$, we need to check whether HT_s's bucket whose position corresponds to d is actually about $s \rightarrow d$. We detail this for in-order UPIs; for ooo ones, the process is similar.

(1) In HT_s, at the position (d modulo current size of HT_s), we find a candidate UPI, call it UPI_{cand}.
(2) If the 10-bit suffix of $UPI_{cand}.d_p$ equals the last 10 bits of d, UPI_{cand} may be about $s \rightarrow d$, but another check is needed; proceed to (3) below. Otherwise, read the next UPI in HT_s, then the next one, etc. and repeat (2) until we find either a sure match (see below), or an empty position, indicating d does not exist in HT_s.
(3) Access, in $STAL_s$, the insertion entry e_{cand} indicated by UPI_{cand}. If the destination node ID in e_{cand} is d, UPI_{cand} is surely about $s \rightarrow d$.

isUPI	i_u	d_s	b_s	b_i	s_s	s_i
63	62	61..52	51..48	47..37	36..32	31..0

Figure 5: UPI fields and their bit spans.

Last but not least, we detail how UPIs encode address of insertions within STAL. Let $\boxed{\text{UPI}_e}$ be the UPI created for an insertion entry e. Let B denote *the STALB closest to* e in STAL_s: if e is in order, B actually stores e; otherwise, B holds an IEM whose OOO field references a data structure holding ooo insertions, including e. The UPI for an in-order insertion of $s \rightarrow d$ has six fields (Figure 5):

- s_i stores B's position in STAL_s, *at the time of the insertion*;
- s_s stores log_2(the size of STAL_s *at the time of the insertion.* Recall (Section 5.1) that this size is always a power of 2;
- b_i stores the position of this entry in B's destNodes;
- b_s stores log_2(the size of B) *at the time of the insertion.*
- d_s stores an 10-bits suffix of d, the destination vertex ID;
- i_u is 1 in an UPI created for an in-order insertion, and 0 for an ooo insertion;
- isUPI is 1 in an UPI, and 0 in an AR or LGCD replacing it.

To find B and e based on UPI_e, e proceed as follows:

(1) From the metadata of STAL_s (dark gray area in Figure 3), we get its size.
(2) We compute the *current* position of B in STAL_s as:

$$\text{size of STAL}_s - (2^{s_s} - s_i)$$

In the above, 2^{s_s} was the size of STAL_s when e was inserted. When STAL_s doubles (Section 5), existing blocks B *remains at the same distance from the tail of* STAL_s; this distance is $2^{s_s} - s_i$. Substracting this difference from STAL_s leads exactly to the position of B in the current STAL_s.
(3) Compute the position of the insertion in B.IEMs as: size of $B - (2^{b_s} - b_i)$. The reasoning behind the calculation is similar to the one above: $2^{b_s} - b_i$ is the distance between the entry e and the end of B at the time of the insertion; this distance does not change as B gets resized.

9 EVALUATION

We evaluate our system on different workloads, also comparing it with the baselines Stinger [11], GraphOne [23], Llama [38], LiveGraph [53], Teseo [28], and Sortledton [14]. Below, we describe our experimental setup (Sec. 9.1), our benchmark dataset, and the graph analytics algorithms used (Sec. 9.2). Then, we study the performance of insertions (Sec. 9.3), updates (Sec. 9.4), analytic querying (Sec. 9.5), concurrent workloads (Sec. 9.6), workloads with ooo insertions (Sec. 9.7), and workloads with ooo updates (Sec. 9.8).

9.1 Hardware and software settings

Our experiments have been executed on a dual-socket Intel Xeon E5-2640 v4 server, with 40 hardware threads and 256 GB of DRAM. Our system is implemented in C++ and compiled with GCC v10.2, with the -O3 optimization flag. We set maximum size occupied by a STALB at 2^{15} bytes, and 2^{13} for an OSTALB. This allows us to store 512 entries per OSTALB, like Sortledton [14] does in its blocks. We report median times over five runs.

Dataset	Vertices $\|V\|$	Edges $\|E\|$	Average degree $\|D\|$
graph500-22, uni-22	2M	64M	26
graph500-24, uni-24	9M	260M	29
graph500-26	33M	1B	33
dota-league	61K	50M	836
edit-wiki	51M	255M	22
yahoo-songs	1.6M	256M	315

Figure 6: Graph datasets in our evaluation.

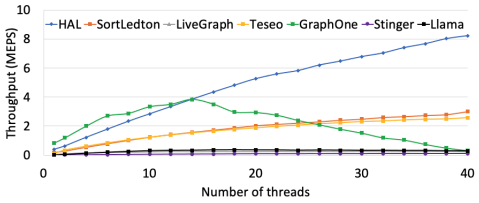

Figure 7: Graph500-24 scalability analysis.

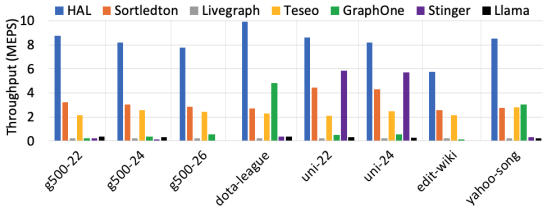

Figure 8: Random order insertion throughput.

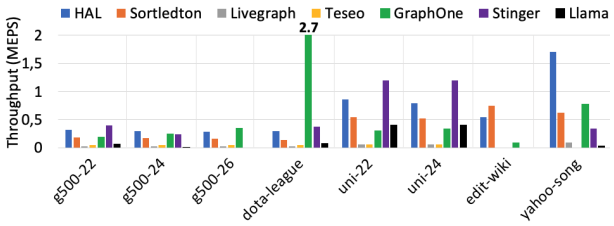

Figure 9: Ordered insertion throughput.

9.2 Workloads

We use the synthetic graph datasets graph-500 [19] with scale factors (SF) 22, 24, and 26; the node fan-out in these graphs follows a power-law degree distribution. To study the impact of the edge distribution across nodes, we used the driver of Teseo [28] to generate variants of graph-500 with scale factors 22 and 24, having *uniform node degree distributions*; we denote these graphs uni-22 and uni-24. We also use three real-world graphs, namely dota-league [19], edit-wiki [24] and yahoo-songs [24]. These datasets, used in previous comparable works, are undirected. As 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 Tab. 6. We use the LDBC graph analytics benchmark [19], 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, we use the algorithm implementations from the Graph Algorithm Platform [3], running on the driver implemented by Teseo [28].

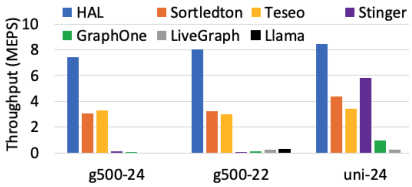

Figure 10: Throughput on updates workloads

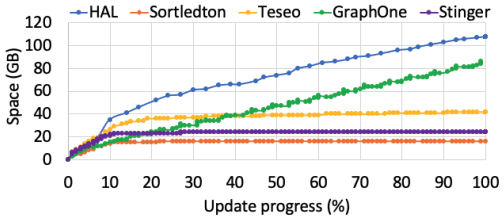

Figure 11: Memory usage on Graph500-24 update workload.

9.3 Edge insertions

To measure insertion performance, we insert successively all the edges of graph500-24, in a random order. All insertions are in order, since no baseline supports ooo updates. Figure 7 reports the insertion throughput, measured in Millions of Edges Per Second (**MEPS**, in short), as we increase the number of threads (on the x axis) from 1 to 2, 4, etc. until 40. **HAL scales up very well**, better than the other systems, as we increase the number of threads.

HAL outperforms Sortledton and Teseo because these systems pay sorting costs, where the neighborhood blocks are sorted by destination ids, thus leading to contention between writer threads. LiveGraph, Stinger, and Llama do not profit at all from parallelism, because of contention between multiple writer threads, simultaneously trying to *search linearly in the adjacency list (or multi-version CSR)* for the existence of edges. Up to 16 threads, GraphOne performs best, but it doesn't check for edge existence; if this check is added, as shown in [14, 28], the throughput decreases to 5 edges per second. Above 16 threads, contention between threads in the archiving phase strongly degrades its performance. *From now on, unless otherwise specified, we experiment with* 40 *threads.*

Figure 8 shows insertion throughput, again in MEPS, on different systems for our eight datasets in random order. **HAL performs better in all cases**. Sortledton and Teseo performance are close, due to their similar design. LiveGraph is slow, because it has to repeatedly resize the vectors storing edges, in order to provide fully sequential access to its adjacency lists. GraphOne throughput is generally low; on dota-league and yahoo-song, it performs better, because these datasets have comparatively fewer vertices. This lowers vertex ID translation costs, and reduces write buffer contention. On uni-22 and uni-24, Stinger ranks second in terms of throughput. This is because linear search for checking edge existence is quite fast when there are few edges per source.

Next, we repeat the experiment but with the edges *in the original order*. All the datasets but yahoo-song and wiki are *ordered on the source vertex ID*, while yahoo-song and wiki are also *mostly* sorted by source ID. This amounts to *bursty* workloads where concentrated updates arrive for successive source vertices. Figure 9 shows that all systems are affected by bursty insertions (lower throughputs than in Figure 8). This is due to higher writer contention, when all threads insert edges into the same source vertex. GraphOne outperformed HAL on two datasets (dota-league, g500-26) because of its unchecked, lock-free edge insertion, particularly effective on datasets with fewer vertices, such as dota-league. In contrast, HAL incurs a secondary index maintenance cost (HT[s]), and Teseo and Sortledton need sorting, leading to more writer contention.

On the wiki dataset, with a maximum of 5M edges adjacent to a node, Sortledton slightly outperforms HAL. This dataset translates into high bursts of edges adjacent to the same node.

Lesson learned: Storing each source vertex's neighborhood list sorted by destination IDs enables more efficient edge-existence checks, than the linear search cost of the append-only store. However, the sorting cost for an insertion is higher. To make edge existence checks efficient, HAL trades the space of the secondary index HT, for speed. On a uniform graph, linear search is cheaper than maintaining a sorted list. Bursty updates on the same source vertex significantly impact the transactional systems because of the per-source vertex locking.

9.4 Updates (insertions and deletions)

We generated update (insertion, deletion) workloads on g500-22 and g500-24, with power-degree edge distributions over nodes, and the uniform uni-24. The update workloads are generated as introduced by Teseo [28]: the first 10% of the workload operations load the graph, after which the remaining 90% (insertions and deletions) keep the (already large) graph at approximately the same size. The size of the update workload is $10 \times |E|$.

Throughput: Figure 10 shows the systems' throughput. HAL outperfom other systems by 2× to 73× in power-law graphs, and by 1.45× to 8.82× on uni-24 workloads because of HAL's insertion and deletion which run in constant time, vs. $O(log(|E|))$ for Sortledton and Teseo, $O(|E|)$ for GraphOne, Stinger, LiveGraph, and Llama. The linear search on the adjacency list penalizes less on a uniformly distributed graph (uni-24). We could not run LiveGraph and Llama on g500-24 and uni-24 workloads because of memory limits.

Memory consumption: We study the memory consumption of various systems in Figure 11. The x-axis is the update progress, while the y-axis shows the store memory size in GBs. In contrast with other systems, HAL's memory occupancy increases: HAL's support for ooo updates requires *maintaining records for each edge in the hash table without deleting them.* Also, support for ooo updates requires storing various *metadata fields*, notably IEMs (holding a stream time, transaction time, and the OOO data structures), which occupied roughly 15 GB by the end of the experiment.

Lesson learned: Compared with set-based systems, HAL uses more memory, in exchange for (i) support for out-of-order updates, (ii) more than 2× speed-up, and (iii) the possibility of supporting historical queries based on the stream time (in our future work).

9.5 Analytics

We ran the LDBC graph analytics benchmark [19] on all the systems. As in [14, 28], Fig. 12 shows the analytics results on the g500 datasets, and on dota-league. BFS and SSSP require random vertex access and sequential neighborhood access: on these, HAL is comparable to the baseline and outperforms LiveGraph, Teseo, GraphOne, Stinger. On PageRank, WCC, and CDLP, which need

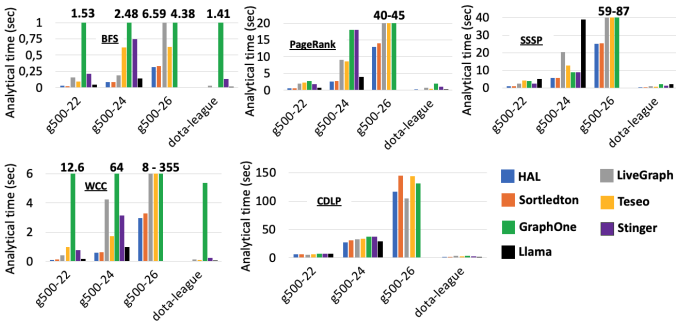

Figure 12: Performance evaluation on graph analytics.

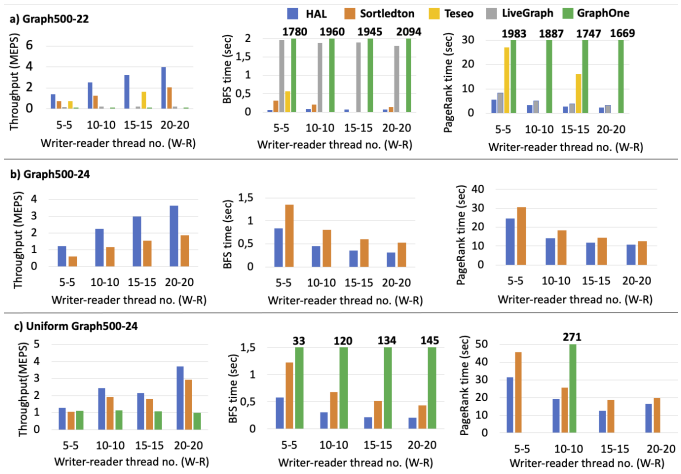

Figure 13: Throughput and query performance analysis using varying updates workloads on read/write workloads.

sequential vertex access and sequential neighborhood access, HAL outperforms all others, with Sortledton coming closest.

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. In contrast, HAL stores destination ids separately from the edge entry metadata, hence, fewer cache misses. Also, LiveGraph checks each entry's invalidation time to see if it was deleted. In contrast, HAL stores a boolean hasDeletes in each STAL block (Sec. 4.2), avoiding the check when the flag is false. **Teseo is slower than HAL** because: (*i*) Teseo needs a per-edge entry mapping (hash table) from sparse to dense vertex ids in the analytics algorithms, which is costly; (*ii*) a sorted neighborhood block in Teseo contains up to 512 edges, while a STALB holds 2047 edges, resulting in fewer random accesses. **GraphOne is slower than HAL** since it runs analytics on an ad-hoc merge of the write-store and the read-store; as noted in [14], it has a high overhead for accessing few edges, hence more cache misses. **Stinger is slower than HAL** because of more cache misses as it stores adjacent edges in a linked list of 14-edges blocks, incurring more random accesses to read the list. **Llama is slower than HAL** because it accesses multiple snapshots, hence more pointer chasing and cache misses, slowing traversals such as SSSP and BFS.

Sortledton is marginally slower than HAL on whole-graph algorithms, such as PageRank, WCC, and CDLP, due to the smaller blocks (512 edges for Sortledton, 2047 for HAL), leading to more cache misses in Sortledton. On algorithms accessing a subset of the graph, such as BFS and SSSP, on smaller datasets (graph500-22 and dota-league), **HAL is slightly slower**. This is because random vertex access are more frequent than sequential neighborhood scans, leading to slightly more cache misses for HAL; also, Sortledton reads 24 bytes to access a source vertex, and HAL reads 48.

Lessons learned Set-based systems (Sortledton and Teseo), which only read the latest version of the destination ids, outperform LiveGraph, which must traverse all their versions in the adjacency list, together with per-edge information. LiveGraph is also hampered by the need to check edge invalidation timestamps. HAL's separation of destination from edge entry metadata speeds up vertex ID accesses. HAL also avoids some edge validity checks (via the hasDeletes flag), and is faster by avoiding (like Sortledton) Teseo's runtime sparse-to-dense vertex ID mapping. GraphOne is slowed down by its merging of the read and write store, and high overhead when accessing few edges, as noted also in [14, 28].

9.6 Concurrent read-write workload

We run the analytics algorithms (BFS, PR) from Sec. 9.5 concurrently with the updates from Sec. 9.4. In Figure 13, the x-axis shows the combinations of Writer (W) and Reader (R) threads, e.g., 5-5 denotes 5 writer and 5 reader threads, and the y-axis shows the throughput (MEPS) and the analytical algorithms (BFS, PR) running time. We pick these values since in our setting, a total of 40 threads has previously shown best performance; of course, each application may have its own best read-write balance.

HAL outperforms the other systems, because of: (*i*) insertions and deletions in $O(1)$; (*ii*) optimized garbage collection (see Sec. 6.1) and database cracking strategy (see Sec. 7). GraphOne is penalized by its the expensive merging of read and write stores. On the uniform workload (uni-24), the insertions and deletions cost is less penalizing for Sortledton and GraphOne compared to HAL, due to the uniform nature of the workload. We could not run LiveGraph because of memory limits, and Teseo because of a runtime error.

We also measured the **impact of reducing edge deletion checks** (Sec. 7) by having each query *mark* the edges it finds deleted, so that future queries can avoid random accesses to learn the same thing from an IEM (OEM). In the above experiment, when using 15 writers and 15 reader threads, PageRank takes 11.90s with this optimization, and 50s without, the 4× speed-up.

Lesson learned: On a concurrent read-write workload, HAL outperforms other state-of-the-art systems owing to: (*i*) append-only storage, which often allows the writer to append data without affecting the reader; (*ii*) constant-time edge deletions; (*iii*) avoid some deletion checks in database cracking style, which improves cache locality and optimizes garbage collection. We also observe that GraphOne's edge copying concurrently with the updates is costly.

9.7 Out-of-order insertions

We now study the performance of our system when faced with a mixture of in-order and out-of-order insertions. Since prior systems do not support OOO updates, we built a **benchmark** based on graph500-22 and graph500-24, as follows. (*i*) Sort the (insertion) workload by the source vertex. (*ii*) Remove the edges whose source

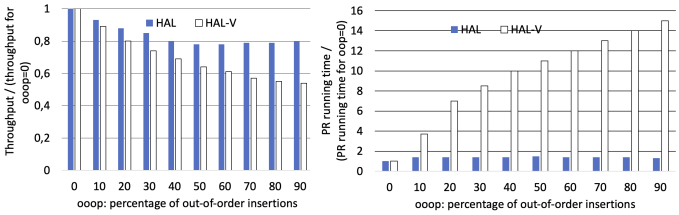

Figure 14: Performance on in- and out-of-order insertions.

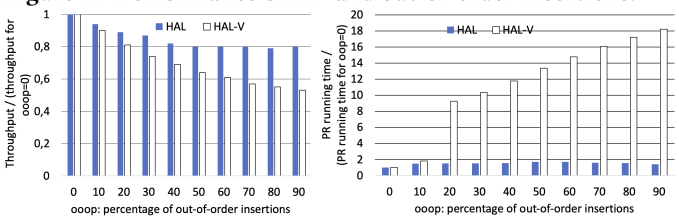

Figure 15: Performance on in- and out-of-order insertions.

vertex has less than 10 edges in the dataset (to facilitate our next steps, see below). We thus obtain an *O*OO *i*nsertion *l*ist called **OIL** (243M edges between 1.5M vertices in graph500-24, and 59M edges between 0.5M vertices in graph500-22). OIL edges are given consecutive STs: 1, 2, 3 etc. (*iii*) For a given **out-of-order percentage** *ooop*, we swap some edges in OIL, to ensure that exactly *ooop* among them are ooo. For instance, if *ooop* is 40%, among each 10 successive edges with the same source, we swap the 1st with the 5th, making the (now) first four updates ooo. Lacking control of how multiple threads intersperse their work, we only use *one thread*.

Recall that HAL stores out-of-order updates first in OSTALBs, and only lazily creates ART (Sec. 5.2). We investigate how this compares to creating an ART at the first out-of-order update received. Specifically, we design a **HAL variant**, called **HAL-V**, which does exactly this, and compare it with HAL.

Figure 14 and Figure 15 (left) show how the **throughput** of HAL and HAL-V is impacted by *ooop*, compared to *ooop*=0 (in-order insertions only). HAL is impacted by 20%, whereas HAL-V shows a 40% decline in throughput when 90% of the updates are ooo. This is due to no ART cost in HAL (as long as the OSTALB is not full). In contrast, HAL-V pays an ART insertion cost for each ooo update.

We observed that at 50% ooo updates, the throughput decreased by 22%, with no further decrease observed thereafter. This is because, at the 50% ooo threshold, the costs incurred for binary searches to locate ooo updates in the STAL[*s*], and the costs for maintaining ooo STALBs (including resizing and ordering by timestamp), are balanced. Beyond this point, there is no further decrease in throughput; the system incurs relatively lower binary search costs but higher maintenance costs for ooo STALBs, as the majority of operations are concentrated within ooo STALBs.

At right in Figure 14 and Figure 15, we study the impact of ooo insertions on **analytics running time** (specifically, PageRank). For each *ooop* value, we run the respective insertion workload, then run PageRank on the graph, and show its running time divided by the time on a graph without ooo updates. HAL-V is strongly impacted (PageRank takes 15× more for oop=90% than oop=0), while HAL is impacted by 1.3×. Both HAL and HAL-V incur a cost for cache misses after every ten entries when accessing the IEM.OOO field to scan the out-of-order insertions, likely the reason for HAL's performance decrease. HAL-V also incurs additional cache misses

Datasets	0%	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
graph500-22	1.00	1.11	1.10	1.09	1.08	1.10	1.11	1.12	1.14	1.16	1.2
graph500-24	1.00	1.04	1.04	1.04	1.04	1.05	1.06	1.07	1.08	1.1	1.11

Table 3: Throughput variations when varying the swap percentage *sw*, on graph500-24 and graph500-22 workloads.

for each edge entry read from the ART, whereas HAL incurs only one for every eight entries (in an OSTALB, each ooo destination ID occupies 8 bytes, thus a cache miss after every eight entries.)

Lesson learned: HAL performance is very robust as the proportion of ooo insertions increases. The extra algorithmic complexity incurred by "lazily" creating ARTs for out-of-order insertions pays off by keeping insertion and analytics very efficient.

9.8 Out-of-order updates

We built an **ooo update benchmark** based on graph500-22 and graph500-24 as follows. (*i*) For each edge in the datasets, record first its insertion, then its deletion, leading to an *O*OO *u*pdate *l*ist we call **OUL**, whose entries are timestamped 1, 2, etc. (*ii*) For a given **swap percentage** *sw*, we swap some updates in OUL. For instance, if *sw* is 10%, after each 10 entries in OUL, we select one edge and swap its insertion with its deletion. If *sw* is 100%, among each 10 edges in the list, we swap each insertion with its deletion. As in Sec. 9.7, we only use *one thread* in this experiment.

Table 3 shows no slowdown (even a little improvement) in HAL throughput as *sw* grows: ooo deletions are fast. We also tested the accuracy of HAL with respect to other systems that do not support ooo updates; we used LiveGraph (Sortledton throws an exception at the first ooo update, while Teseo seems to get "stuck"). For each *sw*, after loading, each system is asked for all the edges present in the graph. HAL returns an empty graph regardless of *sw*, which is correct (in OUL, for each insertion with a given ST, there exists a deletion with ST+1). LiveGraph wrongly returns more and more edges as *sw* grows.

Lesson learned: The ability to properly handle ooo updates is crucial to ensure correctness (transactional querying). HAL achieves this at the cost of more memory.

10 CONCLUSION AND PERSPECTIVES

We presented HAL, a scalable in-memory dynamic graph store, providing consistent graph analytics even in the presence of out-of-order updates. Our system is currently capable of processing approximately 7.5 million updates per second, which is 2.5× faster than the best-performing state-of-the-art system (Sortledton [14]). On analytic graph queries, HAL outperforms existing systems by up to 357×, depending on the task and the settings.

ACKNOWLEDGMENTS

This work was supported by ANR-20-CHIA-0015.

REFERENCES

[1] Daniel J. Abadi, Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stanley B. Zdonik. 2003. Aurora: a new model and architecture for data stream management. *VLDB J.* 12, 2 (2003), 120–139. https://doi.org/10.1007/S00778-003-0095-Z

[2] Arvind Arasu, Shivnath Babu, and Jennifer Widom. 2006. The CQL continuous query language: semantic foundations and query execution. *VLDB J.* 15, 2 (2006), 121–142. https://doi.org/10.1007/S00778-004-0147-Z

[3] Scott Beamer, Krste Asanovic, and David A. Patterson. 2015. The GAP Benchmark Suite. *CoRR* abs/1508.03619 (2015). arXiv:1508.03619 http://arxiv.org/abs/1508.03619

[4] Jon C. R. Bennett, Craig Partridge, and Nicholas Shectman. 1999. Packet reordering is not pathological network behavior. *IEEE/ACM Trans. Netw.* 7, 6 (1999), 789–798. https://doi.org/10.1109/90.811445

[5] Achilles D. Boursianis, Maria S. Papadopoulou, Panagiotis D. Diamantoulakis, Aglaia Liopa-Tsakalidi, Pantelis Barouchas, George Salahas, George K. Karagiannidis, Shaohua Wan, and Sotirios K. Goudos. 2022. Internet of Things (IoT) and Agricultural Unmanned Aerial Vehicles (UAVs) in smart farming: A comprehensive review. *Internet Things* 18 (2022), 100187. https://doi.org/10.1016/J.IOT.2020.100187

[6] Andrey Brito, Christof Fetzer, Heiko Sturzrehm, and Pascal Felber. 2008. Speculative out-of-order event processing with software transaction memory. In *Proceedings of the Second International Conference on Distributed Event-Based Systems, DEBS 2008, Rome, Italy, July 1-4, 2008 (ACM International Conference Proceeding Series, Vol. 332)*, Roberto Baldoni (Ed.). ACM, 265–275. https://doi.org/10.1145/1385989.1386023

[7] Badrish Chandramouli, Jonathan Goldstein, and David Maier. 2010. High-Performance Dynamic Pattern Matching over Disordered Streams. *Proc. VLDB Endow.* 3, 1 (2010), 220–231. https://doi.org/10.14778/1920841.1920873

[8] Mengmeng Chang, Zhiming Ding, Zilin Zhao, and Zhi Cai. 2024. Heterogeneous Modular Traffic Prediction Based on Multilayer Graph Convolutional Network. *IEEE Trans. Intell. Transp. Syst.* 25, 7 (2024), 7805–7817. https://doi.org/10.1109/TITS.2024.3358747

[9] Charles D. Cranor, Theodore Johnson, Oliver Spatscheck, and Vladislav Shkapenyuk. 2003. Gigascope: A Stream Database for Network Applications. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*, Alon Y. Halevy, Zachary G. Ives, and AnHai Doan (Eds.). ACM, 647–651. https://doi.org/10.1145/872757.872838

[10] Guimin Dong, Mingyue Tang, Zhiyuan Wang, Jiechao Gao, Sikun Guo, Lihua Cai, Robert J. Gutierrez, Bradford Campbell, Laura E. Barnes, and Mehdi Boukhechba. 2023. Graph Neural Networks in IoT: A Survey. *ACM Trans. Sens. Networks* 19, 2 (2023), 47:1–47:50. https://doi.org/10.1145/3565973

[11] David Ediger, Robert McColl, E. Jason Riedy, and David A. Bader. 2012. STINGER: High performance data structure for streaming graphs. In *IEEE Conference on High Performance Extreme Computing, HPEC 2012, Waltham, MA, USA, September 10-12, 2012*. IEEE, 1–5. https://doi.org/10.1109/HPEC.2012.6408680

[12] Thomas Eiter and Leonid Libkin (Eds.). 2005. *Database Theory - ICDT 2005, 10th International Conference, Edinburgh, UK, January 5-7, 2005, Proceedings*. Lecture Notes in Computer Science, Vol. 3363. Springer. https://doi.org/10.1007/B104421

[13] Abdulhalim Fayad, Tibor Cinkler, and Jacek Rak. 2024. Toward 6G Optical Fronthaul: A Survey on Enabling Technologies and Research Perspectives. *CoRR* abs/2406.00308 (2024). https://doi.org/10.48550/ARXIV.2406.00308 arXiv:2406.00308

[14] Per Fuchs, Jana Giceva, and Domagoj Margan. 2022. Sortledton: a universal, transactional graph data structure. *Proc. VLDB Endow.* 15, 6 (2022), 1173–1186. https://doi.org/10.14778/3514061.3514065

[15] Lukasz Golab and M. Tamer Özsu. 2003. Issues in data stream management. *SIGMOD Rec.* 32, 2 (2003), 5–14. https://doi.org/10.1145/776985.776986

[16] Shengnan Guo, Youfang Lin, Huaiyu Wan, Xiucheng Li, and Gao Cong. 2022. Learning Dynamics and Heterogeneity of Spatial-Temporal Graph Data for Traffic Forecasting. *IEEE Trans. Knowl. Data Eng.* 34, 11 (2022), 5415–5428. https://doi.org/10.1109/TKDE.2021.3056502

[17] Hassan Halawa and Matei Ripeanu. 2021. Position paper: bitemporal dynamic graph analytics. In *GRADES-NDA '21: Proceedings of the 4th ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA), Virtual Event, China, 20 June 2021*, Vasiliki Kalavri and Nikolay Yakovets (Eds.). ACM, 7:1–7:12. https://doi.org/10.1145/3461837.3464514

[18] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. 2007. Database Cracking. In *Third Biennial Conference on Innovative Data Systems Research, CIDR 2007, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings*. www.cidrdb.org, 68–78. http://cidrdb.org/cidr2007/papers/cidr07p07.pdf

[19] Alexandru Iosup, Tim Hegeman, Wing Lung Ngai, Stijn Heldens, Arnau Prat-Pérez, Thomas Manhardt, Hassan Chafi, Mihai Capota, Narayanan Sundaram, Michael J. Anderson, Ilie Gabriel Tanase, Yinglong Xia, Lifeng Nai, and Peter A. Boncz. 2016. LDBC Graphalytics: A Benchmark for Large-Scale Graph Analysis on Parallel and Distributed Platforms. *Proc. VLDB Endow.* 9, 13 (2016), 1317–1328. https://doi.org/10.14778/3007263.3007270

[20] Vinesh Kumar Jain, Arka Prokash Mazumdar, Parvez Faruki, and Mahesh Chandra Govil. 2022. Congestion control in Internet of Things: Classification, challenges, and future directions. *Sustain. Comput. Informatics Syst.* 35 (2022), 100678. https://doi.org/10.1016/J.SUSCOM.2022.100678

[21] Sadaf Javed, Ali Hassan, Rizwan Ahmad, Waqas Ahmed, Rehan Ahmed, Ahsan Saadat, and Mohsen Guizani. 2024. State-of-the-Art and Future Research Challenges in UAV Swarms. *IEEE Internet Things J.* 11, 11 (2024), 19023–19045. https://doi.org/10.1109/JIOT.2024.3364230

[22] Yuanzhen Ji, Hongjin Zhou, Zbigniew Jerzak, Anisoara Nica, Gregor Hackenbroich, and Christof Fetzer. 2015. Quality-driven processing of sliding window aggregates over out-of-order data streams. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems, DEBS '15, Oslo, Norway, June 29 - July 3, 2015*, Frank Eliassen and Roman Vitenberg (Eds.). ACM, 68–79. https://doi.org/10.1145/2675743.2771828

[23] Pradeep Kumar and H. Howie Huang. 2020. GraphOne: A Data Store for Real-time Analytics on Evolving Graphs. *ACM Trans. Storage* 15, 4 (2020), 29:1–29:40. https://doi.org/10.1145/3364180

[24] Jérôme Kunegis. 2013. KONECT: the Koblenz network collection. In *22nd International World Wide Web Conference, WWW '13, Rio de Janeiro, Brazil, May 13-17, 2013, Companion Volume*, Leslie Carr, Alberto H. F. Laender, Bernadette Farias Lóscio, Irwin King, Marcus Fontoura, Denny Vrandecic, Lora Aroyo, José Palazzo M. de Oliveira, Fernanda Lima, and Erik Wilde (Eds.). International World Wide Web Conferences Steering Committee / ACM, 1343–1350. https://doi.org/10.1145/2487788.2488173

[25] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (1978), 558–565. https://doi.org/10.1145/359545.359563

[26] Hoang D. Le and Anh T. Pham. 2022. Link-Layer Retransmission-Based Error-Control Protocols in FSO Communications: A Survey. *IEEE Commun. Surv. Tutorials* 24, 3 (2022), 1602–1633. https://doi.org/10.1109/COMST.2022.3175509

[27] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, Christian S. Jensen, Christopher M. Jermaine, and Xiaofang Zhou (Eds.). IEEE Computer Society, 38–49. https://doi.org/10.1109/ICDE.2013.6544812

[28] Dean De Leo and Peter A. Boncz. 2021. Teseo and the Analysis of Structural Dynamic Graphs. *Proc. VLDB Endow.* 14, 6 (2021), 1053–1066. https://doi.org/10.14778/3447689.3447708

[29] Aljoscha P. Lepping, Hoang Mi Pham, Laura Mons, Balint Rueb, Philipp M. Grulich, Ankit Chaudhary, Steffen Zeuch, and Volker Markl. 2023. Showcasing Data Management Challenges for Future IoT Applications with NebulaStream. *Proc. VLDB Endow.* 16, 12 (2023), 3930–3933. https://doi.org/10.14778/3611540.3611588

[30] Hourun Li, Yusheng Zhao, Zhengyang Mao, Yifang Qin, Zhiping Xiao, Jiaqi Feng, Yiyang Gu, Wei Ju, Xiao Luo, and Ming Zhang. 2024. A Survey on Graph Neural Networks in Intelligent Transportation Systems. *CoRR* abs/2401.00713 (2024). https://doi.org/10.48550/ARXIV.2401.00713 arXiv:2401.00713

[31] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. 2005. Semantics and Evaluation Techniques for Window Aggregates in Data Streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*, Fatma Özcan (Ed.). ACM, 311–322. https://doi.org/10.1145/1066157.1066193

[32] Jin Li, Kristin Tufte, Vladislav Shkapenyuk, Vassilis Papadimos, Theodore Johnson, and David Maier. 2008. Out-of-order processing: a new architecture for high-performance stream systems. *Proc. VLDB Endow.* 1, 1 (2008), 274–288. https://doi.org/10.14778/1453856.1453890

[33] Ming Li, Mo Liu, Luping Ding, Elke A. Rundensteiner, and Murali Mani. 2007. Event Stream Processing with Out-of-Order Data Arrival. In *27th International Conference on Distributed Computing Systems Workshops (ICDCS 2007 Workshops), June 25-29, 2007, Toronto, Ontario, Canada*. IEEE Computer Society, 67. https://doi.org/10.1109/ICDCSW.2007.35

[34] Rui Li, Fan Zhang, Tong Li, Ning Zhang, and Tingting Zhang. 2023. DMGAN: Dynamic Multi-Hop Graph Attention Network for Traffic Forecasting. *IEEE Trans. Knowl. Data Eng.* 35, 9 (2023), 9088–9101. https://doi.org/10.1109/TKDE.2022.3221316

[35] Zhaoyi Li, Jiawei Huang, Shiqi Wang, and Jianxin Wang. 2024. Achieving Low Latency for Multipath Transmission in RDMA Based Data Center Network. *IEEE Trans. Cloud Comput.* 12, 1 (2024), 337–346. https://doi.org/10.1109/TCC.2024.3365075

[36] 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, 1 (2015), 60–68. https://doi.org/10.2991/IJNDC.2015.3.1.7

[37] Guohan Lu, Yan Chen, Stefan Birrer, Fabián E. Bustamante, and Xing Li. 2010. POPI: a user-level tool for inferring router packet forwarding priority. *IEEE/ACM Trans. Netw.* 18, 1 (2010), 1–14. https://doi.org/10.1145/1816288.1816289

[38] Peter Macko, Virendra J. Marathe, Daniel W. Margo, and Margo I. Seltzer. 2015. LLAMA: Efficient graph analytics using Large Multiversioned Arrays. In *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea,*

April 13-17, 2015, Johannes Gehrke, Wolfgang Lehner, Kyuseok Shim, Sang Kyun Cha, and Guy M. Lohman (Eds.). IEEE Computer Society, 363–374. https://doi.org/10.1109/ICDE.2015.7113298

[39] Guido Maier, Antonino Albanese, Michele Ciavotta, Nicola Ciulli, Stefano Giordano, Elisa Giusti, Alfredo Salvatore, and Giovanni Schembra. 2024. WatchEDGE: Smart networking for distributed AI-based environmental control. *Comput. Networks* 243 (2024), 110248. https://doi.org/10.1016/J.COMNET.2024.110248

[40] Christopher Mutschler and Michael Philippsen. 2013. Distributed Low-Latency Out-of-Order Event Processing for High Data Rate Sensor Streams. In *27th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2013, Cambridge, MA, USA, May 20-24, 2013*. IEEE Computer Society, 1133–1144. https://doi.org/10.1109/IPDPS.2013.29

[41] Christopher Mutschler and Michael Philippsen. 2013. Reliable speculative processing of out-of-order event streams in generic publish/subscribe middlewares. In *The 7th ACM International Conference on Distributed Event-Based Systems, DEBS '13, Arlington, TX, USA - June 29 - July 03, 2013*, Sharma Chakravarthy, Susan Darling Urban, Peter R. Pietzuch, and Elke A. Rundensteiner (Eds.). ACM, 147–158. https://doi.org/10.1145/2488222.2488263

[42] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 2015. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives (Eds.). ACM, 677–689. https://doi.org/10.1145/2723372.2749436

[43] Arnau Rovira-Sugranes, Abolfazl Razi, Fatemeh Afghah, and Jacob Chakareski. 2022. A review of AI-enabled routing protocols for UAV networks: Trends, challenges, and future outlook. *Ad Hoc Networks* 130 (2022), 102790. https://doi.org/10.1016/J.ADHOC.2022.102790

[44] Yousef Saad. 2003. *Iterative methods for sparse linear systems*. SIAM. https://doi.org/10.1137/1.9780898718003

[45] Sherif Sakr, Angela Bonifati, Hannes Voigt, Alexandru Iosup, Khaled Ammar, Renzo Angles, Walid G. Aref, Marcelo Arenas, Maciej Besta, Peter A. Boncz, Khuzaima Daudjee, Emanuele Della Valle, Stefania Dumbrava, Olaf Hartig, Bernhard Haslhofer, Tim Hegeman, Jan Hidders, Katja Hose, Adriana Iamnitchi, Vasiliki Kalavri, Hugo Kapp, Wim Martens, M. Tamer Özsu, Eric Peukert, Stefan Plantikow, Mohamed Ragab, Matei Ripeanu, Semih Salihoglu, Christian Schulz, Petra Selmer, Juan F. Sequeda, Joshua Shinavier, Gábor Szárnyas, Riccardo Tommasini, Antonino Tumeo, Alexandru Uta, Ana Lucia Varbanescu, Hsiang-Yun Wu, Nikolay Yakovets, Da Yan, and Eiko Yoneki. 2021. The future is big graphs: a community view on graph processing systems. *Commun. ACM* 64, 9 (2021), 62–71. https://doi.org/10.1145/3434642

[46] Wei Sang, Huiliang Zhang, Xianchang Kang, Ping Nie, Xin Meng, Benoit Boulet, and Pei Sun. 2024. Dynamic multi-granularity spatial-temporal graph attention network for traffic forecasting. *Inf. Sci.* 662 (2024), 120230. https://doi.org/10.1016/J.INS.2024.120230

[47] Jifan Shi, Biao Wang, and Yun Xu. 2024. Spruce: a Fast yet Space-saving Structure for Dynamic Graph Storage. *Proc. ACM Manag. Data* 2, 1 (2024), 27:1–27:26. https://doi.org/10.1145/3639282

[48] Utkarsh Srivastava and Jennifer Widom. 2004. Flexible Time Management in Data Stream Systems. In *Proceedings of the Twenty-third ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 14-16, 2004, Paris, France*, Catriel Beeri and Alin Deutsch (Eds.). ACM, 263–274. https://doi.org/10.1145/1055558.1055596

[49] Srikanta Tirthapura and David P. Woodruff. 2012. A General Method for Estimating Correlated Aggregates over a Data Stream. In *IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012*, Anastasios Kementsietsidis and Marcos Antonio Vaz Salles (Eds.). IEEE Computer Society, 162–173. https://doi.org/10.1109/ICDE.2012.62

[50] Haijun Wang, Haitao Zhao, Jiao Zhang, Dongtang Ma, Jiaxun Li, and Jibo Wei. 2020. Survey on Unmanned Aerial Vehicle Networks: A Cyber Physical System Perspective. *IEEE Commun. Surv. Tutorials* 22, 2 (2020), 1027–1070. https://doi.org/10.1109/COMST.2019.2962207

[51] Wenjun Yang, Lin Cai, Shengjie Shu, Jianping Pan, and Amir Sepahi. 2024. MAMS: Mobility-Aware Multipath Scheduler for MPQUIC. *IEEE/ACM Trans. Netw.* 32, 4 (2024), 3237–3252. https://doi.org/10.1109/TNET.2024.3382269

[52] Mingyang Zhang, Tong Li, Yue Yu, Yong Li, Pan Hui, and Yu Zheng. 2022. Urban Anomaly Analytics: Description, Detection, and Prediction. *IEEE Trans. Big Data* 8, 3 (2022), 809–826. https://doi.org/10.1109/TBDATA.2020.2991008

[53] Xiaowei Zhu, Marco Serafini, Xiaosong Ma, Ashraf Aboulnaga, Wenguang Chen, and Guanyu Feng. 2020. LiveGraph: A Transactional Graph Storage System with Purely Sequential Adjacency List Scans. *Proc. VLDB Endow.* 13, 7 (2020), 1020–1034. https://doi.org/10.14778/3384345.3384351