Growing Up HAL: Historic and Property Graph Queries

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

Numerous applications, from social media to the banking industry, traffic management, and science, model application data as directed graphs. Fast graph analytics systems are implemented in-memory. Among existing systems, History Adjacency List (HAL) [2, 3] is distinguished by the ability to correctly answer queries even when updates, received in the graph database from multiple sources, arrive out of order, that is, impacted by variable transmission delays that may jeopardize query answer correctness in a system unaware of such issues. In this paper, we describe two recent extensions we brought to HAL, and which increase its functionalities. First, we demonstrate its capabilities to support *historical queries*, that is, queries that carry over the state of the dynamic graph at a given point in time. Second, we describe extensions we implemented to make HAL support *arbitrary numbers of properties on nodes and edges*, making it compatible with the needs of the popular LDBC benchmark [9], and present performance results of HAL on this benchmark.

VLDB Workshop Reference Format:
Angelos Christos Anadiotis, Muhammad Ghufran Khan, and Ioana Manolescu. Growing Up HAL: Historic and Property Graph Queries. VLDB 2025 Workshop: LSGDA 2025.

VLDB Workshop Artifact Availability: The source code, data, and/or other artifacts have been made available at https://gitlab.inria.fr/cedar/hal-dynamic-graph.

1 Introduction

Dynamic graphs are ubiquitous in modern real-time applications that produce immense volumes of data. The ability to promptly analyze these high-speed graph streams is pivotal for use cases such as detecting cyber intrusions in security systems [29], identifying fraudulent activities in financial sectors [29], and uncovering anomalies in Internet of Things (IoT) environments [16]. We consider a model in which a single dynamic graph evolves through continuous edge insertions and deletions originating from multiple data streams. This dynamic graph is maintained within a transactional graph database. Each edge operation is associated with a source (stream) timestamp S_T, reflecting the time of generation, and a write timestamp W_T, marking the time it is recorded in the database. We assume that *the timelines of all streams can be reconciled*, meaning that a global ordering can be derived over updates originating from multiple sources[1]. Importantly, these updates can arrive out of order [8](**ooo**, in short): due to asynchronous network delays, it is possible for an update u_2 with a later S_T to be committed before an earlier update u_1, resulting in $W_T^2 < W_T^1$. This discrepancy necessitates careful management of both logical (stream) and physical (database) time. Such scenarios are common in IoT-based systems [15], where dynamic changes in network topology result in frequent insertion and deletion of edges. Various factors in IoT environments—such as multi-path routing [28], route instability [4], link-layer retransmissions [6], and variations in router-level forwarding [19]—can lead to *ooo* delivery [10].

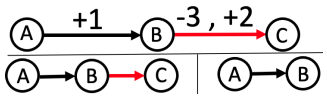

Figure 1: Sample dynamic graph with out-of-order updates.

In the context of Intelligent Transportation Systems (ITS), there are two main variations: (*i*) When *nodes* (e.g., sensors) are stationary—typically located at fixed points like intersections—edges represent *route segments* whose conditions change dynamically due to real-time events such as traffic congestion, accidents, or road closures [16, 18]. If a sensor emits an update about a route becoming available, followed by a closure notification, receiving these updates ooo can mislead the system about the actual road state. (*ii*) When *nodes move*, as in the case of Unmanned Aerial Vehicles (UAVs) [24], edges represent transient communication links between devices. These links fluctuate as UAVs move, leading to rapid changes in network connectivity. Such temporal volatility is critical in applications like surveillance, environmental monitoring, or precision agriculture [5]. If link insertion and deletion updates are received ooo, the resulting inconsistency can significantly affect downstream tasks such as path planning or mission coordination. Figure 1 shows how ooo update propagation affects dynamic graphs. The top portion shows a graph where each edge undergoes one or two operations—insertions (+) and deletions (-)—with updates arriving at the database in an order that may differ from their original emission. Each operation is annotated with its stream time S_T and listed according to its arrival order. Take the edge between nodes B and C as an example: the label -3, +2 indicates that a deletion emitted at stream time 3 is received before an insertion emitted at time 2. Traditional systems, such as those in [7, 14, 25, 29], which are unaware of possible ooo updates, process updates strictly based on their arrival time and assume that a deletion cannot precede its corresponding insertion. As a result, such deletions are ignored, and the subsequent insertion is applied, leaving the edge incorrectly retained.

Instead, *the deletion must be preserved even if the corresponding insertion has not yet been received*; upon arrival of the insertion,

Proceedings of the VLDB Endowment. ISSN 2150-8097.

This work was made possible by Software Heritage, the universal source code archive: https://www.softwareheritage.org.

[1]This is achievable using established techniques in distributed systems, e.g., [12].

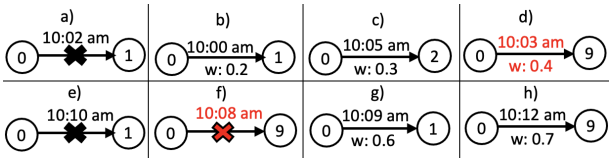

Figure 2: Sample dynamic graph.

the edge should be interpreted as having been first inserted and then deleted. In our example, a system that does not account for ooo updates would produce the incorrect graph state shown at the bottom left of Figure 1. The correct final graph—reflecting the actual order of generation—is shown at the bottom right, with the discrepancy corresponding to the red-highlighted edges affected by ooo delivery. Dynamic graph databases that support ooo updates pose two major 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 to address these challenges can be broadly categorized into five approaches: (*i*) *Buffer-based approaches* [22] temporarily store updates in a buffer and reorder them before database insertion. While effective for restoring stream-time order, this approach may compromise query correctness near buffer boundaries and increase latency—limiting their suitability for real-time analytics; (*ii*) *Punctuation-based methods* [17] use special markers, or "punctuation," within the stream to signify that no further ooo updates will arrive for a given window. Although this enables correctness guarantees, it incurs processing delays while the system waits for punctuation to proceed; (*iii*) *Approximation-based systems* [1] compute bounded-error estimates for numeric queries over streaming data. However, they fall short for precise graph operations such as shortest paths, which need an accurate snapshot. (*iv*) *Time-ordered maintenance* maintains a stream-time ordered list of all edge updates as a naïve strategy to preserve temporal correctness. However, this structure clashes with typical access patterns required for graph algorithms. For instance, traversals necessitate random vertex access followed by sequential neighborhood scans—leading to costly reordering. Also, maintaining stream time order after ooo updates would significantly impact the throughput.

The recent **HAL** system [2, 3] we proposed is a novel in-memory dynamic graph store with multi-version concurrency control protocol (MVCC), and the first to address these challenges. Below, we introduce some terminology in Section 2, before outlining the HAL storage structures in Section 3. Then, Section 4 describes the novel algorithms we use to answer historical queries over dynamic graphs managed by HAL. Then, in Section 5, we describe how we extend HAL to support properties on both nodes and edges. We then present experiments in Section 6, before concluding.

2 Dynamic Graphs with OOO Updates

At any given moment, a dynamic directed graph consists of a set of vertices and a set of directed edges, each defined by a source and destination node. Nodes and edges may also carry associated properties. As in prior systems [7, 14, 29], HAL **decouples the graph topology from node and edge properties**, prioritizing efficient processing of **graph topology updates and queries**[3].

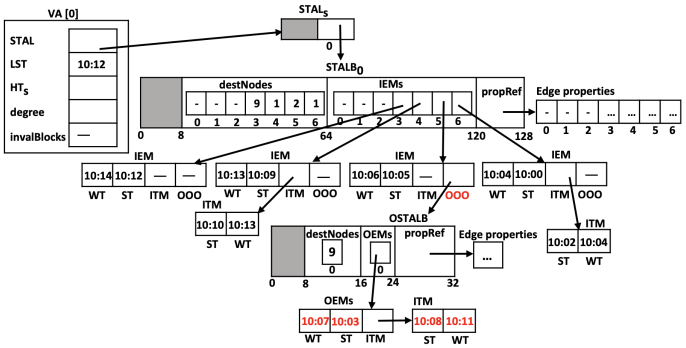

Figure 3: Sample vertex entry VA[0].

The efficient storage and access of node and edge properties is a contribution the present work brings, as we will detail in Section 5.

Out-of-order update. Updates can be received in-order or ooo, depending on when they arrive at the database. An update u with a given stream time S_T and write time W_T is ooo if either (*i*) it is a deletion that arrives before any prior insertion of the same edge, or (*ii*) another update with a later S'_T has already arrived at an earlier W'_T.

Best-guess associated update Insertions and deletions from multiple streams may arrive ooo at the database. To maintain the current view of the graph at any given time, HAL attempts to infer the associations between insertions and deletions corresponding to 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.*

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.

3 The HAL Store

HAL's dynamic graph store is built upon several guiding principles. It organizes edge insertions and deletions within an adjacency list, sorted by the source of the inserted/deleted edges. The system is fundamentally append-only, meaning that updates do not overwrite existing data but only add to it. To avoid unbounded growth, HAL includes garbage collection capabilities, with further technical details found in [3]. It is specifically optimized for in-order updates, which are assumed to be more common than ooo updates ones; when required, HAL introduces tailored fields to efficiently manage ooo updates. Finally, the system is designed to support concurrent processing of graph queries and updates arriving from multiple streams.

The primary data structure utilized is the **History Adjacency List** (**HAL**), depicted in Figure 3. HAL consists of a **V**ertex **A**rray (VA), where each entry corresponds to a distinct *source vertex*; for

instance, VA[0] in the figure represents one such entry. For a given source vertex s, VA[s] maintains a log of all edge updates (both insertions and deletions) originating from s, organized within a **S**tream **T**ime-ordered **A**djacency **L**ist, denoted STAL$_s$. This structure takes the form of an *append-only list of blocks*, maintaining the ordering of the updates by stream time from s to various destination vertices.

Each STAL$_s$ contains associated **metadata** (represented by the shaded region in Figure 3), including a **hasDeletes** flag—set to true if any deletions have been recorded for the source vertex s—and a counter, **delNo**, which tracks the number of slots vacated as a result of consecutive deletions. Furthermore, STAL$_s$ comprises one or more **stream-time ordered adjacency list blocks** (**STALBs**). Each such block, denoted **STALB$_s$**, encapsulates the following components:

(1) **metadata** (shaded in Figure 3), notably the boolean flags **hasDeletes** and **hasOOO**, indicating whether the STALB contains, respectively, deletions, or out-of-order 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 whose length matches that of the corresponding `destNodes` and `IEMs` arrays, and stores the frequently accessed edge property: the *weight*. Other properties associated with edges are maintained separately in external storage, as discussed in detail in Section 5.

For example, in Figure 2, edge insertions such as $0 \rightarrow 1$, $0 \rightarrow 2$, and $0 \rightarrow 9$ occur across steps b), c), d), g), and h). These updates are reflected in Figure 3, where the corresponding destination node identifiers are stored in the destNodes array, and their associated metadata are captured in IEMs.

Figure 3 further illustrates the structure of a STALB labeled S_0, which occupies 128 bytes in total: 8 bytes reserved for metadata (shown on the left in gray), 8 bytes for edge properties (on the right), and 56 bytes each for storing up to 7 entries in both the destNodes and IEMs arrays.

A key structural property is that **STALBs within STAL$_s$ are ordered in descending stream time**: given two STALBs, S_0 and S_1, where S_0 precedes S_1 within STAL$_s$, it follows that all entries in S_0 are more recent—that is, they have higher stream times—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 Section 2), if one exists:
   - The stream time S_T of the deletion entry;
   - The transaction time W_T of the deletion entry;
(4) The **out-of-order updates (OOO)** field, which is initially null, points to a data structure that stores ooo insertions sharing the same source vertex and either the same or a different destination as the one referenced by the current IEM. These insertions are characterized by stream times that are less than the IEM's stream time (ST), yet greater than the ST_s of the preceding IEM—if such an entry exists. The data structure referenced by the OOO field depends on the number of these ooo insertions: if there are at most 512 entries, they are stored in a single **out-of-order stream-time ordered adjacency list blocks** ($OSTALB_s$), which follows the same layout as a $STALB_s$, except that it omits the `hasOOO` field. When this threshold is exceeded, the entries are partitioned across multiple blocks, which are subsequently organized into an Adaptive Radix Tree (ART) [13], with each block indexed by the highest stream time among its entries.

As updates are received, the system handles them as follows: for each *in-order insertion*, an IEM is created; for each *ooo insertion*, an **out-of-order entry metadata (OEM)** is created. An OEM is structurally similar to an IEM but omits the OOO field. For every *deletion*, a corresponding ITM is created and linked to the IEM or OEM of the insertion that is currently deemed most likely to correspond to the deletion.

To illustrate, consider Figure 2, which shows insertions and deletions of edges $0 \rightarrow 1$, $0 \rightarrow 2$, and $0 \rightarrow 9$. The resulting data structures, as constructed in STAL$_0$, are depicted in Figure 3. Specifically, the ITM with stream time 10:10, referenced by the IEM with stream time 10:09, indicates that the deletion of $0 \rightarrow 1$ at 10:10 (Figure 2e) is associated with the insertion at 10:09 (Figure 2g). Similarly, the ITM with stream time 10:08, linked to the OEM at 10:03, shows that the deletion of $0 \rightarrow 9$ at 10:08 (Figure 2f) is best matched with the insertion at 10:03 (Figure 2d). The STAL structure ensures the update entries for a given source vertex s are stored *in the descending order of their stream time*. This ensures $O(1)$ complexity for the operations we expect to be most common: in-order insertions and deletions [3].

Remark. In STAL, *a deletion is only stored as an 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.

VA[s] also maintains the **l**atest **s**tream **t**ime (LST) corresponding to the most recent in-order update received for the source vertex s. For example, in Figure 2, the most recent in-order insertion from source 0 is the edge $0 \rightarrow 9$ at 10:12, as shown in step h); accordingly, Figure 3 reflects 10:12 as the LST stored in VA[0]. In addition to LST, VA[s] tracks the **degree**, representing the number of active edges—i.e., those inserted but not yet deleted—currently maintained within STAL$_s$. It also includes a reference to **invalBlocks**, which identifies the set of STALB blocks containing updates from source s that are potentially ready for garbage collection due to subsequent deletions. Lastly, VA[s] stores a **hash table** (**HT$_s$**, in short) whose keys are the destination vertices d for which we have received some $s \rightarrow d$ updates, and whose values we detail below.

Figure 4 depicts the evolution of HT$_0$. Each yellow-shaded region highlights the data structures created in response to the corresponding entry from Figure 2; dashed arrows denote data structure changes on each step, while solid arrows represent references among data structures. As the graph evolves, HT$_s$ maintains, for

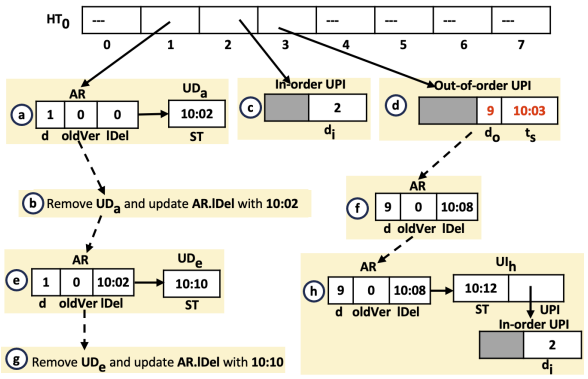

Figure 4: HT_0 through insertions and deletions.

each destination d, *one of the following three data structures*, selected based on the current update state.

- **Update position and indicator** (**UPI**, in short): The UPI associated with the edge $s \rightarrow d$, denoted as UPI$[s, d]$, encodes the position of the *most recent* $s \rightarrow d$ *insertion*, whether it is in-order or ooo, within $STAL_s$. For example, in Figure 2, the initial insertion of edge $0 \rightarrow 2$ occurs in step c), with a stream time of 10:05, and is in-order. Consequently, an in-order UPI entry for destination vertex 2 is created in HT_0, as illustrated in Figure 4. The dark grey UPI fields represent metadata that *encode the location of the insertion entry within* $STAL_s$. As the database grows, the block holding an insertion may grow or move in the list (the latest updates are always first); our UPI encoding [3] guarantees constant-time access to the insertion, throughout the graph evolution.
- **Staging area** (**AR**, in short): The AR for an edge $s \rightarrow d$ stores insertions (or deletions) that are awaiting their corresponding deletions (or insertions). For example, in Figure 4, when a deletion of $0 \rightarrow 1$ is received at step a), HT_0 contains no prior insertion for this edge. Consequently, a staging area is initialized for $0 \rightarrow 1$, and an update deletion block UD_a is created to store the stream time 10:02 associated with this deletion. The corresponding AR is then stored at index 1 in HT_0. Later, when the deletion of $0 \rightarrow 9$ arrives at step f) with stream time 10:08, a matching insertion is found at index 3 in HT_0. At this point, the previously maintained out-of-order UPI for $0 \rightarrow 9$ is replaced with an AR. *The staging area the component of HAL's storage where incoming operations "match" those previously received, which are staged while awaiting their possible "pair"*. Its operating details [3] follow the possible cases: ooo deletion (resp., insertion) received before (resp. after) the corresponding insertion (resp. deletion). Our algorithms also handle the possibility that some inserted edges are never deleted.
- **Last garbage collected deletion** (**LGCD**, in short) for vertex s is the stream time of the most recent deletion of $s \rightarrow d$ that has been garbage-collected.

$HT_s(d)$ through the lifecycle of the edge The purpose of these distinct data structures can be understood by analyzing the *lifecycle* of an edge $s \rightarrow d$ in our system.

Table 1: Complexity comparison for graph operations.

System	Edge insertion	Edge deletion	Edge look-up
Llama [20], Stinger [11]	$O(\|E\|)$	$O(\|E\|)$	$O(\|E\|)$
GraphOne [11]	$O(1)$	$O(\|E\|)$	$O(\|E\|)$
LiveGraph [29]	$O(1)$	$O(\|E\|)$	$O(1)$
Teseo [14], Sortledton [7], **HAL out-of-order**	$O(log(\|E\|))$	$O(log(\|E\|))$	$O(log(\|E\|))$
Spruce [25]	$O(\|E\|)$	$O(log(\|E\|))$	$O(log(\|E\|))$
HAL in-order	$O(1)$	$O(1)$	$O(1)$

- *A UPI is created only when the database receives its first update for* $s \rightarrow d$ *as an insertion*, similar to what was done for the insertion $0 \rightarrow 2$ in Figure 2 c).
- An AR persists as long as there are *one or more insertion(s) (or deletion(s)) for which the corresponding deletion(s) (or insertion(s)) have not yet been received*. The operation we received *waits* for the one not yet received, in the AR. This applies to: in- or ooo insertions that do not have a corresponding deletion; in the future, such a deletion might be received, or it might not; and, every deletion for which the current STAL does not contain a potential 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 $s \rightarrow d$ have been garbage collected.

Table 1 presents a comparison of HAL's algorithmic complexity with that of existing systems, which do not accommodate ooo updates. The superior complexity of HAL enables it to surpass the performance of other systems by up to 73× in throughput and 357× in graph analytics [3].

4 Historical queries

In this section, we address the challenge of evaluating *historical queries* over a dynamic graph at a given moment, specified as a *stream time* τ. Recall that the stream time globally orders the moments where updates are emitted in the system, whereas the transaction time is impacted by the order in which updates reach the database. Thus, the stream time is the natural time to use for historical queries. For ease of presentation, below, we consider the query requesting *all the graph edges at stream time* τ; based on this, more complex queries can also be handled.

The problem, then, is to decide, at a given moment (transaction time) τ_h when the query is issued, whether each edge $s \rightarrow d$ in the store, was logically present at time τ, with $\tau \leq \tau_h$.

We must guarantee that only updates that both *occurred (were issued) before the* τ, and *were committed before* τ_h are reflected. This is in order to ensure transactional query processing, as updates that were not committed before τ_h are considered not received yet at the database.

For each edge $s \rightarrow d$ in the store, one of the following holds:

- The edge was inserted at or before τ, was never deleted, and the insertion committed before τ_h: this edge is still present in the graph at τ_h, and should be included in the answer.
- The edge was inserted at or before τ, was deleted after τ, and the insertion committed before τ_h: the edge still existed at τ and should be included.

- The edge was inserted and then deleted before or at τ, and the deletion committed before τ_h: the edge no longer existed at τ and must not be in the answer.

For instance, in Figure 3, a query is issued at τ_h = 10:15, asking for the edges in **VA[0]** as they were at τ = 10:05.

To make this reasoning, for every source vertex s, HAL uses STAL_s, the stream-time sorted adjacency list storing updates whose source node is s, as described in Section 3. We traverse STAL_s to determine which edges were present at τ, considering only edges that had been committed before the query's arrival time τ_h. We proceed for each source vertex s as follows:

1. Locate the starting STALBs using stream time τ. We first perform a binary search on STAL_s using the stream timestamp τ as the search key. This identifies the most recent STALBs, S_p, whose entries could be present at time τ. All STALBs including S_p and older than S_p are traversed in reverse stream-time order. This avoids unnecessary work, bounding the scan to only relevant updates (the others are too recent).

2. Classify the STALB using metadata flags. Before accessing a STALB's entries, we read its hasDeletes and hasOOO flags (Section 3), which state whether the block contains deletions, respectively, ooo updates. We distinguish four cases:

(1) **Both `hasDeletes` and `has000` are true:** The STALB contains deletions, and and ooo updates. For each IEM entry, we check both the ITM field (to detect deletions) and the OOO field (to include any present OOO updates).
(2) **`hasDeletes` is true, `has000` is false:** The STALB contains deletions, but no OOO updates. We check each IEM entry's ITM field, to confirm that it was not deleted before τ.
(3) **`has000` is true, `hasDeletes` is false:** The STALB contains ooo updates but no deletion. We follow the OOO field for each entry (if present) to locate relevant updates in OSTALBs or ARTs.
(4) **Neither flag is set:** The STALB contains neither deletions nor out-of-order updates. This enables a *fast traversal* (just scan entries sequentially), without the need to check per-entry metadata. This improves cache locality and avoids costly random memory accesses.

In our example, for τ = 10:05 and τ_h = 10:15, the binary search identifies STALB_0 as the relevant block. At this point, it contains both deletions and ooo entries. Thus, STALB_0 is in **Case (1)**.

3. Locate the first relevant entry within the STALB. On the most recent STALB S_p, perform a binary search on its IEMs array to find the first index e_i of an update whose stream time is $\leq \tau$. This is the point from where we start reading. For older STALBs, which are always full, we scan from index 0 (newest) to the end. HAL also inspects the IEM at index $e_i - 1$ (if present). Although its stream time exceeds τ, it may contain an OOO pointer referencing OEM entries with stream time $\leq \tau$. If such a pointer exists, we process it as described in step (1c), ensuring that all relevant ooo updates are considered. In Figure 3, a binary search on the IEMs in STALB_0 using τ = 10:05 locates index 5, corresponding to the insertion entry for the edge $0 \rightarrow 2$. This is the most recent insertion not exceeding τ and serves as the entry point for the scan. We also verify that the IEM entry at index 4 does not contain an OOO field, confirming that the entry at index 5 is the correct starting point.

(1) **Scan IEM entries from the position e_i.** We scan IEM entries from e_i toward the oldest in S_p, and do the same for older STALBs (which, by the way HAL works, are guaranteed to be full). For each IEM entry at position *curIndex* within its block, we evaluate its eligibility based on transaction visibility and metadata as follows:
   (a) **Transaction time check:** We first check whether the IEM entry at the *curIndex* in the STALB was committed before the query arrived, that is, before τ_h, or after. If the entry was committed after τ_h, it was not part of the database at that time, therefore it is skipped.
   If the entry was committed before τ_h, and the STALB contains neither deletions nor ooo updates (case (4)), or only ooo updates (case (3)), the entry is considered present, and we include $s \rightarrow d$ in the query result.
   (b) **Deletion check (if applicable):** When the current STALBs contains deletions (in cases (2) or (1)) and the IEM entry at the *curIndex* in the current STALBs has an associated deletion record, we access its ITM block, which holds the metadata for that deletion. We examine the ITM to determine whether the deletion is relevant to the current query. Specifically, we check that the deletion's stream time is later than τ (meaning that the edge was still logically present at τ), and that the insertion (i.e., IEM entry) was committed before the query time τ_h. If both conditions are satisfied, the edge is considered present at τ, and we include $s \rightarrow d$ in the result.
   (c) **Out-of-order (OOO) update check (if applicable):** When the current STALBs includes ooo updates (cases (3) or (1)), and the IEM entry at *curIndex* has an OOO field, we follow that field to retrieve potential past updates that arrived ooo, were issued before τ, and committed before the query time τ_h. Depending on the number of such entries, as explained in Section 3, the OOO field may reference either a single OSTALB (Out-of-order STALB) if there are 512 or fewer entries, or an Adaptive Radix Tree (ART) which is an index over several OSTALBs, if there are more entries. In either case, we need to identify the metadata (OEM) attached to these entries; we collect them in a set denoted OEM_o. We proceed as follows:
      - When the OOO field leads to a single OSTALB, we perform a binary search within this OSTALB using τ to locate OEM entries whose stream time is earlier than τ. We gather these entries in OEM_o.
      - When the OOO field leads to an ART, we search the ART for all OSTALBs whose keys are less than or equal to τ. In each retrieved OSTALB, we use binary search to extract the OEM entries whose stream time is earlier than τ, and add them to OEM_o.

   For each OEM_o entry, we apply a deletion check as follows:

- If the entry is not marked as deleted (i.e., with `hasDeletes` flag set to false) and committed before the query arrival time τ_h, we include $s \longrightarrow d$ in the query result.
- Otherwise, we check if the entry has ever been deleted: if that is the case, it has an associated ITM. Upon finding an ITM, we examine it to determine when the deletion occurred. If the deletion's stream time is later than τ, and if the entry was committed before the query arrival time τ_h, we conclude that the edge $s \longrightarrow d$ was still present at τ. Thus, it should be included in the result. If the entry has no ITM, it has ever been present since its insertion, and it is included in the result.

In our working example, continuing the scan in Figure 3, we begin at index 5 of STALB_0, which contains the IEM_5 entry for edge $0 \longrightarrow 2$ with write timestamp (WT) 10:06. Since WT $< \tau_h =$ 10:15 and no deletion is recorded, this edge is included in the query result. This IEM_5 has an OOO field, which HAL follows; it leads to an OSTALB marked with `hasDeletes`, thus it contains deletions. Within this OSTALB, we locate an OEM_0 for edge $0 \longrightarrow 9$ with WT 10:07 $< \tau_h$, and a deletion committed at 10:08, also before τ_h. Since the deletion's stream time is $\geq \tau =$ 10:05, the edge was still present at τ and is included in the result. We then proceed to index 6 of STALB_0, which contains the IEM_6 entry for edge $0 \longrightarrow 1$ inserted at 10:04, with a deletion committed at 10:04, both before τ_h. Because the deletion's stream time is earlier than τ, edge $0 \longrightarrow 1$ is excluded from the query result.

The worst-case computational complexity of evaluating a historical query is $O(\log(|E|) + |R|)$, where $|E|$ is the number of edge entries and $|R|$ is the size of the query result.

5 Property-rich nodes and edges

As outlined in Section 3, *edge properties* are not stored directly within the graph topology but are instead *referenced* from it. Specifically, for a given source node s, the STAL_s entry, as shown in Figure 3, includes a reference to a vector storing the *weight* property of each edge. This property can be viewed as generic, as it is frequently used in graph analytical algorithms such as Single-Source Shortest Path (SSSP).

To enable attaching arbitrary properties on edges, as well as vertex properties, we store them independently within RocksDB [21], *a high-performance, embeddable key-value store optimized for fast storage*. This separation ensures that the graph topology remains compact and thus HAL continues to provide efficient analytical operations on the graph topology, operations that are often central for analytics.

RocksDB organizes key-value data within *column families*, each of which can be seen as an independently managed database within a single RocksDB instance. Each column family consists of one key-value collection (called *memtable*), implemented by a skip list-based structure (SkipListMemTable) [26]. Memtables provide logarithmic-time access and are periodically flushed to disk as Sorted String Tables (SSTables) under RocksDB's Log-Structured Merge-tree (LSM-tree) architecture. Each column family maintains its own Log-Structured Merge-tree (LSM-tree) [23]. This design allows each column family to be tuned and accessed independently for best performance.

To support flexible and efficient access to properties, we store vertex and edge properties in separate RocksDB column families within RocksDB. The decision reflects the fact that these two categories of data are semantically orthogonal and follow different access and update patterns. By storing them separately, our system stands to benefit from improved data locality, faster point lookups, and higher write throughput.

To encode vertex and edge properties within RocksDB, our system employs *composite keys* that enable direct and efficient access to each property. This departs from existing systems [7, 14, 25, 29], which co-locate all properties of a vertex or edge and require deserialization of an entire property block to retrieve a single property. Our solution avoids unnecessary data access, reduces cache misses, and improves both query latency and memory efficiency, particularly in analytical workloads that target selective subsets of properties. The details of the vertex and edge property formats are as follows:

- **Vertex properties.** The key used to store a vertex property is of the form `<updateStreamTime>_<vertexId>_<propertyName>_<op>`. For example, the key `1_1_name_insert` identifies an insert operation for the `name` property of vertex 1 at stream time 1; a possible value is, e.g., `"Khan"`. This design also preserves the update chronological order.
- **Edge properties.** Similarly, an edge property is indexed using the format: `<updateStreamTime>_<src>-<dst>_<propertyName>_<op>`, where `<op>` indicates the type of operation (e.g., `insert`, `delete`). A key like `1_1-2_createdDate_insert` records the insertion of a `createdDate` property on the edge from vertex 1 to vertex 2 at stream time 1, with a value such as `"01/10/2025"`.

This structure encodes not only the source and destination of the edge, but also the stream timestamp of the update and the property name, enabling a log-style representation of updates. This logging approach is consistent with the design principles of HAL, which stores the graph as a history of updates. Insertions and deletions of properties are allowed both on nodes and on edges. Key-value pairs thus formed are inserted into the memtable of their corresponding column family.

Transactional coordination and consistency. Given that we now store graph information in two distinct stores (the core HAL structure for the topology, the core HAL structure, plus RocksDB for properties), we need to ensure *transactional guarantees* for updates relying on these two stores. For that, we adopt a decoupled but coordinated transactional protocol, with HAL serving as the transaction coordinator. Specifically, property updates are first committed within the (transactional) RocksDB context, using its native optimistic concurrency control mechanism [27]. If that transaction fails, we do not attempt the corresponding topological update in HAL; otherwise, we go forward with the HAL transaction (structure update). Since HAL only appends updates (that it may later match with other pending updates), HAL update transactions do not fail.

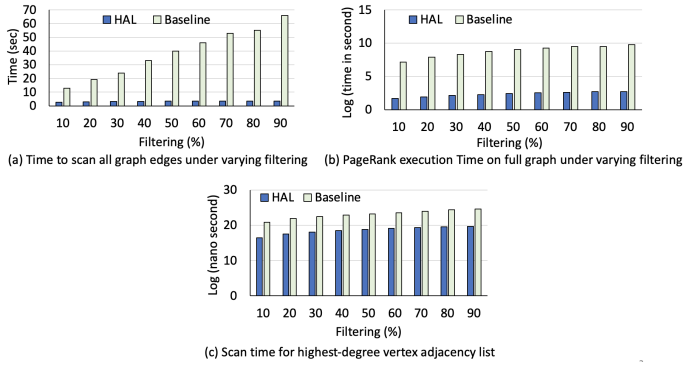

Figure 5: Performance of historical queries at different moments in the update history.

This ordering of operations guarantees that no edge or vertex becomes visible in the topology, unless its associated properties are durably persisted.

Historical queries over property graphs As explained above, we have extended (a) the HAL topology store with historical queries, (b) separately (and orthogonally), the storage model with node and edge properties, on which snapshot queries are already supported. The ingredients are in place to combine the two extensions, i.e., answer historical queries over property graphs. This extension is part of our future work.

6 Experiments

We have implemented the new algorithms proposed in this work within the HAL system [2, 3] and experimentally assess their performance. We describe our experimental setting in Section 6.1, before discussing experimental results in Section 6.2 and 6.3. Our system is available at https://gitlab.inria.fr/cedar/hal-dynamic-graph.

6.1 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 [7] does in its blocks. We report median times over five runs.

6.2 Historical queries

To evaluate the performance of historical queries under out-of-order (OOO) insertions, we based our experiments on the Graph 500 scale-24 dataset [9], which contains approximately 9 million vertices and 260 million edges. Lacking a benchmark with OOO updates, we reuse our update workload from [3], based on the Graph 500-24 dataset, but featuring 50% OOO insertions. We construct our workload as follows: (*i*) Sort the graph500-24 workload (edge insertions) by the source vertex. (*ii*) Remove the edges whose source vertex has fewer than 10 edges in the dataset (this facilitates our next steps; see below). We thus obtain an *O*OO *i*nsertion *l*ist called **OIL** (243 million edges, between 1.5 million vertices). Edges in OIL are assigned consecutive stream times: 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 out-of-order. In our case, we use $ooop = 50\%$, resulting in a realistic OIL where approximately half of the edges are disordered.

Since existing systems [7, 14, 25, 29] do not support historical queries in the presence of out-of-order (OOO) insertions, we implemented a **baseline** system to compare our approach with This baseline employs a conventional adjacency list structure, where updates are stored in the order in which they are received (not necessarily the order in which they are emitted), unlike HAL. To support historical queries, we augment the baseline with a **secondary index** based on an Adaptive Radix Tree (ART). In this index, each key is a stream time, and the corresponding value indicates the position, in the adjacency list, of an update emitted at that stream time (since we assume a global order on updates, there is at most one update per stream time). This secondary ART enables access to data by their stream time, an access pattern not supported by the simple baseline.

To illustrate a varied range of historical queries, we have computed 9 query stream times $\tau_1, \ldots, \tau_9$ as follows. For each $1 \leq i \leq 9$, τ_i is the highest stream time among the first $i \times 10\%$ of the edge updates in the above workload. Figure 5 presents the performance of historical queries at stream time τ_i, for $1 \leq i \leq 9$; the value plotted on the x axis is $i \times 10\%$. For instance, 10% filtering corresponds to querying 10% of the 243 million total edges. Figure 5(a) shows the time required to scan all edges under different filtering levels. Figure 5(b) presents the execution time of the PageRank algorithm computed over the filtered edge sets, while Figure 5(c) reports the scan time within the adjacency list of the highest-degree vertex, chosen to avoid bias from low-degree vertices.

In all three cases, **HAL** consistently outperforms the baseline system. This is due to HAL's data layout, which stores edge updates sorted by stream time. As explained in Section 4, HAL leverages this ordering to perform a binary search to quickly identify the position corresponding to the target stream time, followed by a cache-friendly linear scan. In contrast, the baseline system relies on the secondary ART tree index. Although an ART enables efficient lookup of the target stream time in $O(\log |E|)$ time, subsequent scanning suffers due to random-access reads, as the adjacency list is not sorted by stream time—leading to frequent cache misses. This performance gap is clearly visible across all query types in Figure 5, demonstrating the efficiency of HAL in supporting historical queries, particularly as filtering levels increase.

Lesson learned: HAL delivers significantly better performance for historical queries under out-of-order insertions due to its stream-time-sorted layout. In contrast, the baseline system suffers from the overhead of random-access patterns introduced by its reliance on secondary indexing.

6.3 Property graph queries

To evaluate the performance of property-aware vertex and edge insertions, we rely on the popular LDBC Social Network Benchmark (SNB) interactive workload [9], at scale factors 1, 3, 10 and 30; the dataset characteristics are shown in Table 2. We present a comparison with LiveGraph [29], one of the most recent transactional graph systems for dynamic graphs. Systems such as [7, 14, 25] provide only partial property graph support, typically limited to a single edge property (e.g., weight) and no support for vertex properties,

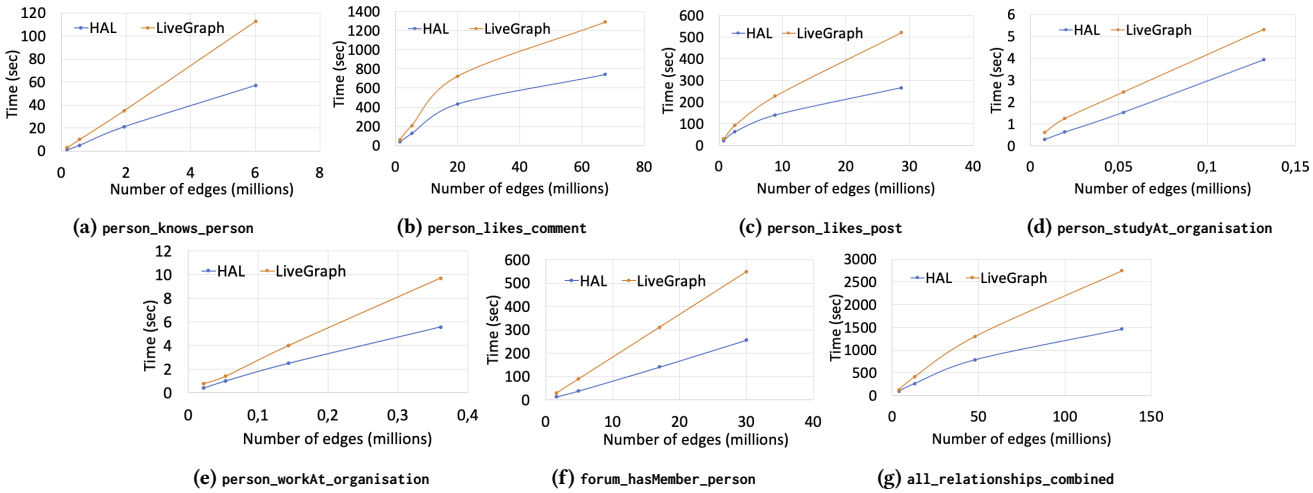

Figure 6: Edge insertion time across SNB relationships.

as they are primarily designed for structural graph updates. In contrast, LiveGraph supports the full LDBC SNB update benchmark, making it a suitable and practical baseline for our evaluation. We evaluate insertion performance in this PG context. We focus on six benchmark relationships, namely `person_knows_person`, `person_likes_comment`, `person_likes_post`, `person_studyAt_organisation`, `person_workAt_organisation`, `forum_hasMember_person`. These are the only ones in the SNB dataset that include edge properties.

Table 2: Estimated number of nodes and edges in LDBC SNB Interactive v1 datasets at different scale factors.

Scale factor (SF)	Number of vertices	Number of edges
1	~3M	~17M
3	~9M	~51M
10	~30M	~170M
30	~90M	~510M

To ensure compatibility with our system and LiveGraph, which both support only integer vertex IDs, we address a subtle yet important issue in the LDBC SNB dataset: overlapping node IDs across different entity types. For example, both `person` and `forum` nodes may have IDs such as 1, 2, etc., leading to collisions when ingesting multiple entity types into a shared graph context. This lack of global uniqueness arises because the SNB benchmark organizes node and edge data across multiple CSV files, where IDs are guaranteed to be unique only within the scope of each node type, not across the entire graph. To resolve this, we apply a normalization strategy that ensures globally unique vertex identifiers while preserving the integrity of the original dataset. Concretely, we first count the number of nodes for each entity type (e.g., 1000 `person` nodes, 2000 `forum` nodes) and then define a translation offset n_T for each node type T. This offset is added to the original ID of each node of type T, such that the resulting IDs become globally unique. For instance, we may assign $n_T = 0$ for `person` nodes and $n_T = 1000$ for `forum` nodes, ensuring that the adjusted `forum` node IDs begin where the `person` IDs end.

This ID translation is applied consistently across the dataset: first, to each CSV file containing node data, and second, to each CSV file encoding edge data, by shifting both the source and destination vertex IDs according to their respective types. This preprocessing step allows us to merge all node and edge types into a unified graph structure without ambiguity, thereby enabling fair and collision-free insertion experiments across all supported relationships.

For each relation type, we measure the time required to insert both vertices and edges along with their associated properties. The resulting performance plots are shown in Figure 6, where the x-axis represents the number of edges (in millions) and the y-axis indicates the insertion time (in seconds). Across all evaluated relationships, HAL achieves higher insertion throughput than LiveGraph. This is primarily due to two architectural differences between the systems. First, LiveGraph relies on Bloom filters for edge existence checks, which may return false positives and trigger linear scans of its Transactional Edge Log (TEL). In contrast, HAL maintains a secondary index implemented as a hash table (HT in Section 3), allowing constant-time edge existence checks. Second, LiveGraph assigns a single TEL block per source vertex, which leads to frequent resizing as adjacency lists grow. HAL avoids this overhead by distributing edges across multiple fixed-size STALB blocks, reducing the need for costly reallocations.

Lesson learned: HAL's support for constant-time edge existence checks and multiple sorted adjacency list blocks enable efficient insertions. By separating property storage from the topology and indexing each property individually, HAL maintains a lightweight structure optimized for analytical scans. Despite relying on external storage for vertex and edge properties, HAL consistently outperforms LiveGraph across all evaluated SNB relationships.

7 Conclusion and Perspectives

We presented two key extensions to the HAL graph store [3]: support for historical queries and property-rich graphs. HAL now efficiently evaluates temporal queries and manages arbitrary properties on nodes and edges, aligning with LDBC SNB requirements. Experiments show that HAL outperforms baseline systems on historical queries and achieves higher throughput than LiveGraph on property graph workloads, demonstrating its effectiveness for advanced dynamic graph use cases.

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). doi:10.1007/S00778-003-0095-Z

[2] Angelos Christos Anadiotis, Muhammad Ghufran Khan, and Ioana Manolescu. 2025. Catching up with Disorder: Dynamic Graphs with Out-of-Order Updates (demonstration). In *SIGMOD*.

[3] Angelos Christos Anadiotis, Muhammad Ghufran Khan, and Ioana Manolescu. 2025. Dynamic Graph Databases with Out-of-order Updates. *PVLDB* 17, 13 (2025).

[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). doi: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 of Things* 18 (2022). doi:10.1016/J.IOT.2020.100187

[6] 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). doi:10.48550/ARXIV.2406.00308 arXiv:2406.00308

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

[8] Hassan Halawa and Matei Ripeanu. 2021. Position paper: bitemporal dynamic graph analytics. In *GRADES-NDA*. ACM. doi:10.1145/3461837.3464514

[9] Alexandru Iosup, Tim Hegeman, Peter A. Boncz, et al. 2016. LDBC Graphalytics: A Benchmark for Large-Scale Graph Analysis on Parallel and Distributed Platforms. *Proc. VLDB Endow.* 9, 13 (2016). doi:10.14778/3007263.3007270

[10] 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). doi:10.1016/J.SUSCOM.2022.100678

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

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

[13] 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. doi:10.1109/ICDE.2013.6544812

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

[15] Aljoscha P. Lepping, Hoang Mi Pham, Volker Markl, et al. 2023. Showcasing Data Management Challenges for Future IoT Applications with NebulaStream. *Proc. VLDB Endow.* 16, 12 (2023). doi:10.14778/3611540.3611588

[16] 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). doi:10.48550/ARXIV.2401.00713 arXiv:2401.00713

[17] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. 2005. Semantics and Evaluation Techniques for Window Aggregates in Data Streams. In *SIGMOD*. ACM. doi:10.1145/1066157.1066193

[18] Rui Li, Fan Zhang, Tong Li, Ning Zhang, and Tingting Zhang. 2023. DMGAN: Dynamic Multi-Hop Graph Attention Network for Traffic Forecasting. *IEEE TKDE* 35, 9 (2023). doi:10.1109/TKDE.2022.3221316

[19] 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). doi:10.1145/1816288.1816289

[20] Peter Macko, Virendra J. Marathe, Daniel W. Margo, and Margo I. Seltzer. 2015. LLAMA: Efficient graph analytics using Large Multiversioned Arrays. In *ICDE*. doi:10.1109/ICDE.2015.7113298

[21] Meta Open Source. [n. d.]. RocksDB: A Persistent Key-Value Store for Fast Storage. https://rocksdb.org/.

[22] Christopher Mutschler and Michael Philippsen. 2013. Distributed Low-Latency Out-of-Order Event Processing for High Data Rate Sensor Streams. In *IPDPS*. doi:10.1109/IPDPS.2013.29

[23] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Inf.* 33, 4 (June 1996), 351–385. doi:10.1007/s002360050048

[24] Arnau Rovira-Sugranes, Abolfazl Razi, Fatemeh Afghah, et al. 2022. AI-enabled routing protocols for UAV networks: Trends, challenges, and outlook. *Ad Hoc Networks* 130 (2022). doi:10.1016/J.ADHOC.2022.102790

[25] 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). doi:10.1145/3639282

[26] Meta Open Source. 2021. RocksDB MemTable: In-Memory Data Structures (Skip List). https://github.com/facebook/rocksdb/wiki/MemTable. Accessed: 2025-05-29.

[27] Meta Open Source. 2023. RocksDB Optimistic Concurrency Control Protocol. https://github.com/facebook/rocksdb/wiki/Transactions. Accessed: 2025-05-29.

[28] 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). doi:10.1109/TNET.2024.3382269

[29] 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). doi:10.14778/3384345.3384351