SHORT-PAPER

# Catching up with Disorder: Dynamic Graphs with Out-of-Order Updates

**ANGELOS CHRISTOS G ANADIOTIS**, Oracle Corporation, Austin, TX, United States

**MUHAMMAD GHUFRAN KHAN**, Polytechnic Institute of Paris, Palaiseau, Ile-de-France, France

**IOANA MANOLESCU**, Polytechnic Institute of Paris, Palaiseau, Ile-de-France, France

**Open Access Support** provided by:

**Oracle Corporation**

**Polytechnic Institute of Paris**



**Published:** 22 June 2025

**Citation in BibTeX format**

SIGMOD/PODS '25: International Conference on Management of Data
*June 22 - 27, 2025*
*Berlin, Germany*

**Conference Sponsors:**
SIGMOD


https://doi.org/10.1145/3722212.3725085
ISBN: 9798400715648

# Catching up with Disorder: Dynamic Graphs with Out-of-Order Updates

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

Muhammad G. 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, potentially distributed. Providing both high ingestion rate and efficient analytics with transactional guarantees over such dynamic graphs is challenging, even more so when updates may be received *out-of-order* at the database. We propose to demonstrate HAL [2], a novel in-memory dynamic graph database system, which addresses 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.

**Demo video**: https://www.youtube.com/watch?v=S-Qi6Oj68oU

## CCS Concepts

• **Information systems** → **Graph-based database models**.

## Keywords

Dynamic graphs, Out-of-order updates, Graph databases, Multi-stream dynamic graph analytics

**ACM Reference Format:**
Angelos C. Anadiotis, Muhammad G. Khan, and Ioana Manolescu. 2025. Catching up with Disorder: Dynamic Graphs with Out-of-Order Updates. In *Companion of the 2025 International Conference on Management of Data (SIGMOD-Companion '25), June 22–27, 2025, Berlin, Germany*. ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3722212.3725085

## 1 Introduction

Dynamic graphs are omnipresent in real-time applications that generate massive amounts of data. In particular, timely analysis of high-velocity graph data streams is critical for applications such as monitoring of cyber attacks in system security applications [23], fraud detection in financial institutions [23], anomaly detection in IOT networks [14] 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, or database) 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** [7] (**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 ($W_T$).

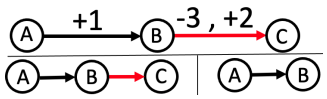


**Figure 1: Sample dynamic graph with out-of-order updates.**

Such scenarii are frequent in IoT applications [13], where changes in the network topology cause edges to appear and disappear rapidly. In IoT networks [9], issues such as multi-path routing [22], route fluctuations [3], link-layer retransmission [5], and router forwarding [17] 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. [14, 16]. 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 [20]. As UAVs move, *these links appear and disappear rapidly*. This dynamic behavior is crucial for surveillance and monitoring, agricultural field inspection [4], etc. 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 illustrates dynamic graphs with ooo update propagation on a graph (at the top) where for each edge, the database receives an insertion, and for one, 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 B and C, -3, +2 states that the deletion emitted at 3 is received before the insertion emitted at 2. In dynamic graph systems prior to HAL, e.g., [6, 12, 23], 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

[1] This can be achieved via well-known distributed systems techniques, e.g., [11].


*SIGMOD-Companion '25, Berlin, Germany*

ACM ISBN 979-8-4007-1564-8/2025/06
https://doi.org/10.1145/3722212.3725085

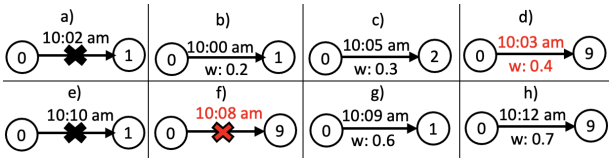


**Figure 2: Sample dynamic graph.**

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 shown 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.

Prior solutions for these challenges can be grouped in five categories: (*i*) *Buffer-based approaches* [19] temporarily 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* [15] 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. (*iv*) *Approximation-based systems* [1] provide bounded-error estimates of numerical queries, but cannot handle precise graph operations, e.g., shortest paths, which need an accurate snapshot. ) 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.

The recent **HAL** system [2] is a novel in-memory dynamic graph store with multi-version concurrency control protocol (MVCC), and the first to addres these challenges.

Below, we introduce some terminology in Section 2, outline HAL's novel storage structures in Section 3, then describe the scenario we propose to demonstrate in Section 4, highlighting key components and functionalities.

## 2 Dynamic Graphs with OOO Updates

At any point in time, a dynamic directed graph has a set of nodes and a set of edges. Each edge has a source and destination vertex. Each edge, and each node, may have some properties. Similar to other systems [6, 12, 23], we **store the graph topology separately** from node and/or edge properties, and focus on efficiently handling **updates and queries on the graph topology**. Data access paths based on node/edge properties can be optimized independently.

**Out-of-order update** An update is in-order or out-of order (ooo) depending on its arrival time at the database site. Specifically, update $u$ with stream time $S_T$, received at the database at database 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 $S'_T > S_T$.

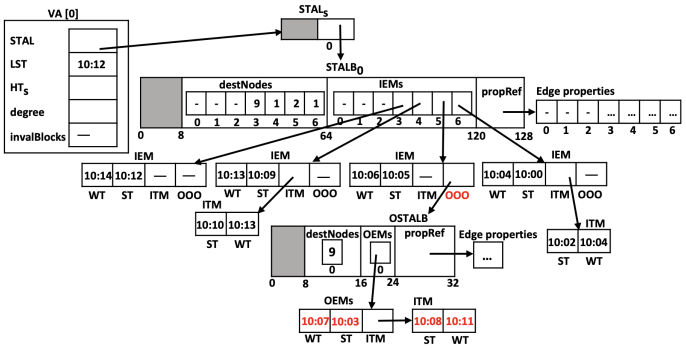


**Figure 3: Sample vertex entry VA[0].**

**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, HAL tries to **guess** 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*.

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 based on a few design principles. It stores and processes updates in an *adjacency list*, sorted by the source of the inserted/deleted edges. 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* (for space reasons, garbage collection details are delegated to [2]). 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 *support graph analytics concurrently with updates from multiple streams*.

The top data structure is the **History Adjacency List** (**HAL**), illustrated in Figure 3. HAL contains a **V**ertex **A**rray (VA), with an entry for each *source vertex*, e.g., VA[0] in the Figure. For a given source vertex $s$, VA[$s$] stores all updates (insertions, deletions) of edges having the source $s$, in a **S**tream **T**ime-ordered **A**djacency **L**ist STAL$_s$, which is an *append-only list of blocks*, e.g., STALB$_0$ in the figure is one block in STAL$_0$. VA[$s$] also stores the **l**atest **s**tream **t**ime (LST) 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. Still in VA[$s$], **degree** is the number of edges inserted but not

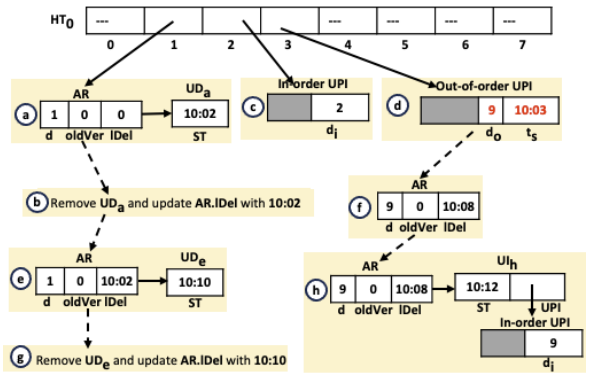


**Figure 4: $HT_0$ through insertions and deletions.**

deleted, currently in $STAL_s$, while **invalBlocks** references STALB blocks containing updates whose source is $s$, and which should be garbage-collected following deletions. Finally, VA[$s$] stores a **hash table** ($\mathbf{HT}_s$, 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 illustrates the evolution of $HT_0$. Each yellow area encloses the data structures created due to the respective entry in Figure 2; dashed arrows trace the data structures changes 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 $s \rightarrow d$, denoted UPI[$s$, $d$] encodes the position of the *latest (in-order or ooo)* $s \rightarrow d$ *insertion* in $STAL_s$. In Figure 2, the first insertion for the edge $0 \rightarrow 2$, 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 denote information *encoding the position of the insertion within* $STAL_s$. As the graph grows, the block holding an insertion may grow or move in the list (the latest updates are always first); our UPI encoding [2] 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 (deletions) of this edge, waiting for their corresponding deletions (insertions).* For instance, in Figure 4, when we receive a deletion of $0 \rightarrow 1$ at a), $HT_0$ does not hold a corresponding insertion; we create the staging area for $0 \rightarrow 1$, 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 $0 \rightarrow 9$ arrives at 10:08, we find the corresponding insertion in $HT_0$ at index 3, and replace the ooo UPI of $0 \rightarrow 9$ with a corresponding 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 [2] 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.

**Table 1: Complexity comparison for graph operations.**

| System | Edge insertion | Edge deletion | Edge look-up |
|---|---|---|---|
| Llama [18], Stinger [10] | $O(\|E\|)$ | $O(\|E\|)$ | $O(\|E\|)$ |
| GraphOne [10] | $O(1)$ | $O(\|E\|)$ | $O(\|E\|)$ |
| LiveGraph [23] | $O(1)$ | $O(\|E\|)$ | $O(1)$ |
| Teseo [12], Sortledton [6], **HAL out-of-order** | $O(log(\|E\|))$ | $O(log(\|E\|))$ | $O(log(\|E\|))$ |
| Spruce [21] | $O(\|E\|)$ | $O(log(\|E\|))$ | $O(log(\|E\|))$ |
| **HAL in-order** | $O(1)$ | $O(1)$ | $O(1)$ |

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

- *Only when the first update ever to be received by the database for* $s \rightarrow d$ *is an insertion*, we create a UPI, as we did above for the insertion $0 \rightarrow 2$ 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 $s \rightarrow d$ have been garbage collected.

Table 1 compares HAL's algorithmic complexity with that of prior systems (which do not handle ooo updates). HAL's very good complexity lead it to outperform the other systems by up to 73× in throughput, and 357× on graph analytics [2].

## 4 Demonstration Scenario

Lacking prior benchmarks with ooo updates, we plan to demonstrate HAL on dynamic graph datasets we built from known benchmarks, e.g., [8], as in [2]. We will make each of these datasets *interactive*, as follows. We start with a sequence of edge insertions and deletions as in [8], but allow users to *control when each update is received at the database*, in order for HAL to exercise its guessing of the most likely associate update, for each incoming one.

The interactive GUI we will use has three main parts (see Figure 5):

- **Ground Truth Graph State**: At the top (see Figure 5a), we show the so-called ground truth graph state. This is an artefact we build exclusively for the demo, to show what the graph should be *if all the updates were emitted from, and received to, the same site, thus without any delays.* Of course, in practice, in a dynamic, distributed graph setting, we do not know the ground truth.
- **Updates in Transit** (Figure 5b): In this area, we show updates emitted by streams, that are not yet received by the database server, and may be arbitrarily delayed. Users decide on the arrival order by *explicitly releasing updates from transit, in or out of order.*
- **Updates in Database** (Figure 5c): This area shows updates in the order they were received in the database. Each update has a database time $W_T$ and a stream time $S_T$. By default, the updates are ordered by database time, aligning with the approach used in traditional graph database systems. Alternatively, selecting the **order by stream time** option shows the storage order employed by HAL as shown in Figure 5d).

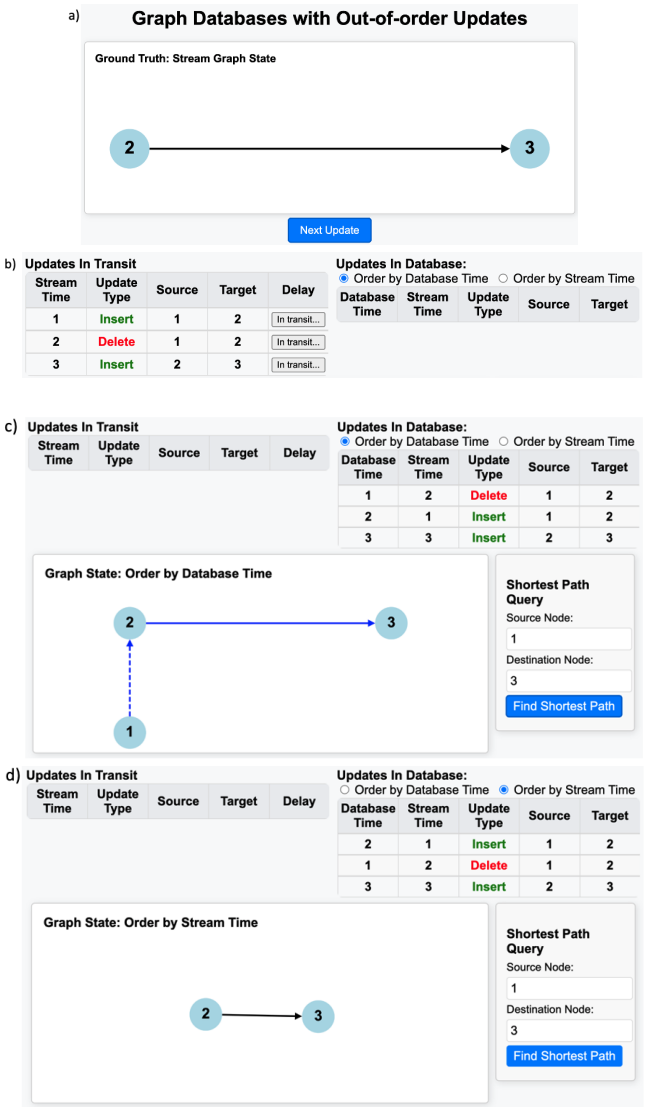


**Figure 5: Screen captures of HAL's interactive GUI.**

The state of the graph is shown for both configurations: ordered by database time (Figure 5c) and ordered by stream time (our system HAL, see Figure 5d). This allows users to visually compare the graph state across the two systems. To illustrate analytics, users can execute, e.g., shortest path queries on the current graph state, as shown in Figure 5c) and Figure 5d).

As an example, consider that three updates are emitted successively: the insertion of edge $1 \rightarrow 2$ at stream time 1, followed by its deletion at stream time 2, and the insertion of edge $2 \rightarrow 3$ at stream time 3. These updates are initially placed in the transit area, as shown in Figure 5b), whereas the correct graph state is shown in Figure 5a). Assume that the database receives these updates in an out-of-order manner: the deletion of $1 \rightarrow 2$, followed by the insertion of $1 \rightarrow 2$, and finally, the insertion of $2 \rightarrow 3$ at stream time 3. In our demonstration, the user can "make each update arrive" (Figure 5b) to the database by clicking the "In Transit" button next to that update.

Figure 5c) illustrates the updates stored in the database when ordered by database time. In this case, edge $1 \rightarrow 2$ is (incorrectly) part of the graph, as depicted by a dashed line. However, when the updates are ordered by stream time (our system HAL, Figure 5d), the correct graph state is maintained; here, the edge $1 \rightarrow 2$ does not exist. This ensures consistency with the ground truth graph shown in Figure 5a).

If a shortest path query is executed between nodes 1 and 3, traditional graph databases that order updates by database time (Figure 5c) incorrectly indicate that a path exists. However, comparing this with HAL (Figure 5d) and the ground truth graph (Figure 5a), demo attendees can notice that no such path exists.

## References


[1] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. B. Zdonik. Aurora: a new model and architecture for data stream management. *VLDB J.*, 12(2), 2003.

[2] A. C. Anadiotis, M. G. Khan, and I. Manolescu. Dynamic graph databases with out-of-order updates. *PVLDB*, 17(13), 2025.

[3] J. C. R. Bennett, C. Partridge, and N. Shectman. Packet reordering is not pathological network behavior. *IEEE/ACM Trans. Netw.*, 7(6), 1999.

[4] A. D. Boursianis, M. S. Papadopoulou, P. D. Diamantoulakis, A. Liopa-Tsakalidi, P. Barouchas, G. Salahas, G. K. Karagiannidis, S. Wan, and S. K. Goudos. Internet of things (IoT) and agricultural unmanned aerial vehicles (UAVs) in smart farming: A comprehensive review. *Internet of Things*, 18, 2022.

[5] A. Fayad, T. Cinkler, and J. Rak. Toward 6g optical fronthaul: A survey on enabling technologies and research perspectives. *CoRR*, abs/2406.00308, 2024.

[6] P. Fuchs, J. Giceva, and D. Margan. Sortledton: a universal, transactional graph data structure. *Proc. VLDB Endow.*, 15(6), 2022.

[7] H. Halawa and M. Ripeanu. Position paper: bitemporal dynamic graph analytics. In *GRADES-NDA*. ACM, 2021.

[8] A. Iosup, T. Hegeman, P. A. Boncz, et al. LDBC graphalytics: A benchmark for large-scale graph analysis on parallel and distributed platforms. *Proc. VLDB Endow.*, 9(13), 2016.

[9] V. K. Jain, A. P. Mazumdar, P. Faruki, and M. C. Govil. Congestion control in internet of things: Classification, challenges, and future directions. *Sustain. Comput. Informatics Syst.*, 35, 2022.

[10] P. Kumar and H. H. Huang. Graphone: A data store for real-time analytics on evolving graphs. *ACM Trans. Storage*, 15(4), 2020.

[11] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7), 1978.

[12] D. D. Leo and P. A. Boncz. Teseo and the analysis of structural dynamic graphs. *Proc. VLDB Endow.*, 14(6), 2021.

[13] A. P. Lepping, H. M. Pham, V. Markl, et al. Showcasing data management challenges for future IoT applications with NebulaStream. *Proc. VLDB Endow.*, 16(12), 2023.

[14] H. Li, Y. Zhao, Z. Mao, Y. Qin, Z. Xiao, J. Feng, Y. Gu, W. Ju, X. Luo, and M. Zhang. A survey on graph neural networks in intelligent transportation systems. *CoRR*, abs/2401.00713, 2024.

[15] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. Semantics and evaluation techniques for window aggregates in data streams. In *SIGMOD*. ACM, 2005.

[16] R. Li, F. Zhang, T. Li, N. Zhang, and T. Zhang. DMGAN: dynamic multi-hop graph attention network for traffic forecasting. *IEEE TKDE*, 35(9), 2023.

[17] G. Lu, Y. Chen, S. Birrer, F. E. Bustamante, and X. Li. POPI: a user-level tool for inferring router packet forwarding priority. *IEEE/ACM Trans. Netw.*, 18(1), 2010.

[18] P. Macko, V. J. Marathe, D. W. Margo, and M. I. Seltzer. LLAMA: efficient graph analytics using large multiversioned arrays. In *ICDE*, 2015.

[19] C. Mutschler and M. Philippsen. Distributed low-latency out-of-order event processing for high data rate sensor streams. In *IPDPS*, 2013.

[20] A. Rovira-Sugranes, A. Razi, F. Afghah, et al. AI-enabled routing protocols for UAV networks: Trends, challenges, and outlook. *Ad Hoc Networks*, 130, 2022.

[21] J. Shi, B. Wang, and Y. Xu. Spruce: a fast yet space-saving structure for dynamic graph storage. *Proc. ACM Manag. Data*, 2(1), 2024.

[22] W. Yang, L. Cai, S. Shu, J. Pan, and A. Sepahi. MAMS: mobility-aware multipath scheduler for MPQUIC. *IEEE/ACM Trans. Netw.*, 32(4), 2024.

[23] X. Zhu, M. Serafini, X. Ma, A. Aboulnaga, W. Chen, and G. Feng. Livegraph: A transactional graph storage system with purely sequential adjacency list scans. *Proc. VLDB Endow.*, 13(7), 2020.