Aion: Efficient Temporal Graph Data Management

Georgios Theodorakis
Neo4j
george.theodorakis@neo4j.com

James Clarkson
Neo4j
james.clarkson@neo4j.com

Jim Webber
Neo4j
jim.webber@neo4j.com

ABSTRACT

Modern graph database management systems (DBMSs) can process highly dynamic labeled property graphs (LPGs) with many billions of relationships comfortably, but those systems often ignore the temporal dimension of data, how a graph evolved over time. Temporal analytics allow users to query and compute over the graph throughout its history so that valuable line-of-business data is always accessible and never lost. However, existing approaches tend to be ad-hoc and vary in performance depending on the size of the effective graph workload, such as local pattern matching or global graph algorithms.

In this work, we describe Aion, a transactional temporal graph DBMS that generalizes previous approaches for LPGs. Aion extends Neo4j, a modern graph DBMS, incurring minimal performance overhead by decoupling the graph's history from the latest graph version. To support efficient temporal analytics independently of workload characteristics, Aion adopts a hybrid temporal storage approach: (i) for fast full graph restoration at arbitrary time points, it uses *TimeStore* that indexes updates by time; (ii) for fine-grained graph history accesses, it uses *LineageStore* that indexes updates by entity identifiers. To enable incremental graph computations for improved latency, Aion introduces a compute-efficient in-memory LPG representation. Our experiments show that Aion achieves comparable or better performance versus existing non-transactional temporal systems and provides up to an order of magnitude speedup over classic Neo4j.

1 INTRODUCTION

Systems designers are increasingly turning to graph technology ($3.2 B estimated market size by 2025 [60]) to manage the volume and associative complexity of modern data. To accommodate such increasing demands, commercial graph database management systems (DBMSs) [5, 46, 50, 75] allow users to model real-world interactions as a set of nodes and relationships at many billions or trillion scale [52]. While these systems have made a significant impact, to date, they have generally lacked native support for analyzing the evolution of a graph over time.

Temporal analytics span a wide range of use cases, from data auditing [66] (e.g., HIPAA privacy compliance) and anomaly detection in IoT devices [45], to mining trends over time [7, 8, 67], and restoring data to a previous version (i.e., perform data repair). These are important classes of applications, and graph DBMSs must be able to support them regardless of the prevailing characteristics of their workloads.

Relational DBMSs have addressed the problem of temporal analytics over a single table [58, 63, 66], and temporal validity is even a standardized SQL:2011 [36] feature. However, maintaining the history of a graph without a predefined schema is challenging. In response, solutions involving commercial graph systems [13, 20, 64] enhance the labeled property graph (LPG) model [61] to store historical data as extra graph entities. While this may help solve the functional problem, it complicates application logic and incurs potential performance penalties for non-temporal workloads. Another approach is to capture the temporal behaviour using a sequence of snapshots by storing the graph state at specific time points along with the deltas between those snapshots [27, 30, 31, 34, 45]. Again, this theoretically solves the functional problem, but it is prohibitively expensive for point- or small-neighbourhood queries. Finally, systems such as Raphtory [67] or IBM SystemG [76] store the history of individual graph entities together, which allows fast local graph accesses but may result in an expensive all-history scan for full graph reconstruction.

The goal of this work is to design and implement a transactional graph DBMS that accounts for the following challenges of temporal applications: (i) handle the dynamically changing structure of LPGs (i.e., schemaless data); (ii) enable temporal capabilities without affecting non-temporal operations (i.e., querying the latest version of the graph); (iii) efficient storage and retrieval of graph history for different workloads (i.e., handle time and graph size dimensions); and (iv) efficient query execution over parts of data that remain the same between time points. Specifically, our contributions are as follows:

(i) Temporal graph data model. We formalize temporal graphs by enhancing LPGs with time capabilities that allow (bi-)temporal graph analytics. We extend Cypher (an SQL-like graph query language for graphs [22]) with temporal constructs and introduce idiomatic procedures for incremental processing from Cypher.

(ii) Temporal graph storage. To support efficient access to historical global graphs and subgraphs, we design a general-purpose temporal graph system called Aion[1], which exposes an intuitive API to retrieve various temporal graph access patterns. Based on the workload characteristics, Aion can choose between two temporal stores: (i) *TimeStore* for full graph snapshots and (ii) *LineageStore* for efficient fine-grained graph history access without user intervention.

(iii) Integration with commercial graph DBMS. We integrate Aion with Neo4j[2] to provide transactional guarantees for temporal queries. In addition, using Aion, we support efficient incremental graph computations by introducing a memory-friendly dynamic LPG data structure.

Our experimental evaluation highlights the benefits of our design: (i) for global queries, Aion outperforms Raphtory and Gradoop, two in-memory non-transactional temporal systems, by up to 7.3× and 52.2×, respectively; (ii) for point queries, it upholds comparable throughput to Raphtory and orders of magnitude higher performance than Gradoop, while providing support for out-of-core workloads. When integrated with Neo4j, Aion incurs only 28-41% storage increase and less than 15% ingestion performance overhead. At the same time, normal read transactions are unaffected by Aion, and temporal analysis is accelerated by up to an order of magnitude.

---

[1] Aion is a Hellenistic deity associated with cyclic time.

[2] The source code is available at https://github.com/Neo4jResearch/Aion.

Series ISSN: 2367-2005

10.48786/edbt.2024.43

The remainder of the paper is organized as follows: first, we explain the problem of temporal graph processing and survey state-of-the-art approaches (Sec. 2). We present our temporal graph model and Cypher extensions (Sec. 3); and then describe how Aion performs temporal graph storage (Sec. 4) before discussing how we integrate Aion with Neo4j (Sec. 5) to enable fast incremental computations. Finally, the paper presents our evaluation results (Sec. 6), related work (Sec. 7), and conclusions (Sec. 8).

2 BACKGROUND

This section introduces the salient concepts of graph DBMSs and temporal graph analytics. First, we categorize different traditional graph systems (Sec. 2.1). Then, we provide background on temporal data management (Sec. 2.2), and finally introduce the basics of incremental graph processing (Sec. 2.3).

2.1 Graph Databases

In this work, we focus on systems that support the LPG model [61], in which optionally labeled nodes in a graph represent entities that, in turn, are connected by named, directed relationships. Both nodes and relationships can host properties as key-value pairs. This model is common to many modern graph databases for online transactional processing and graph engines for online analytical processing. Prominent examples of graph databases include MemGraph [46], Neo4j [50], Neptune [5], and TigerGraph [75], which are designed to handle dynamically changing graphs at significant volume. Analytical engines enable high-performance (often parallel) graph computations on (static) graphs by adopting the vertex-centric model (see Pregel [43], GraphLab [41], Giraph [17], or GraphX [81]).

Neo4j transactional processing and analytics. Neo4j [50] is a graph DBMS for out-of-core workloads that maintains its own page cache. All operations are transactional with (at least) read-committed isolation. Users access the database using Java APIs or Cypher [22] queries locally or over the network using an efficient binary communication protocol called Bolt [55], whose behaviours are discussed in Sec. 6.7. For analytics on a static graph, Neo4j has the Graph Data Science (GDS) library and runtime [57]. Users construct a static Compressed Sparse Row (CSR) [42] projection using the current graph and execute algorithms over it by calling procedures from Cypher. As a popular graph database with a pre-existing graph compute engine, we have chosen to extend Neo4j for our experimental work in Aion. However, we believe the techniques are applicable to a broader range of graph database management systems, as discussed in Sec. 5.1.

2.2 Temporal Graph Data Management

In relational DBMSs, temporal analytics has been extensively studied [63], resulting in temporal validity becoming part of SQL:2011 standard [36]. Relational databases capture the history of a table using the temporal table construct [58, 66] or allow users to query directly changes atop tables [3]. Temporal SQL distinguishes two time dimensions: (i) system (or transaction) time, which represents when data was updated in the database; and (ii) application (or event) time, a timestamp from when an event occurred.

As the graph DBMSs from Sec. 2.1 do not have native support for temporal operations, they have historically used the model-based approach [28]. Graph entities are enriched with additional time properties (e.g., validity duration), and historical data is stored as extra nodes and relationships [13, 20, 64]. While this approach works with existing systems, it requires complicated application logic to perform graph operations [20] and incurs significant storage and runtime overheads [45, 76]. For example, Gradoop [62] is an analytical engine that supports distributed execution over the model-based approach at the significant cost of performing an all-history scan to retrieve valid graph parts.

Conversely, the snapshot-based approach [27, 30, 34, 35, 45] captures temporal behaviour using a sequence of snapshots (i.e., graph state materialization at a specific time) and logs deltas [24] between them. Graph updates and snapshots are stored in disjoint buckets, called time windows, enabling compact data storage [34, 45]. Tegra [31] implements a different approach for storing snapshots atop persistent adaptive radix trees (pART) [39] and is able to accelerate ad-hoc analytics on arbitrary time windows of the graph. Though snapshot storage is more efficient than model-based storage, it remains prohibitively expensive for pattern-matching queries that access small subgraphs (see Sec. 6.3), as it requires full snapshot materialization.

At the other end of the spectrum, systems such as Raphtory [68], IBM SystemG [76], and TGDB [28] use a fine-grained storage approach: graph updates are stored in a key-value store, where the key is either a node or a relationship ID and the corresponding value is a list of that element's history. For example, with Raphtory, a distributed analytics system that maintains the complete graph history in memory, its temporal graph model allows updates via data streams (without transactions). However, extracting the graph snapshot at an arbitrary time requires scanning all updates. In practice, this is similar to the model-based approach and negatively affects global query performance.

While temporal tables are the accepted solution for maintaining history with a predefined schema, preserving the history of a dynamically changing LPG is considerably more challenging. The existing solutions tackle the problem by choosing one of the following approaches: (i) model-based, (ii) snapshot-based, or (iii) fine-grained storage. By choosing one solution, the systems are biased towards its strengths and are sub-optimal in performance terms for other workloads. A general-purpose temporal graph storage engine should not fall foul of such biases and work well for a wide range of workloads.

2.3 Incremental Execution of Graph Algorithms

In contrast to the approaches described in Sec. 2.2, some systems enable incremental execution, whereby previously computed results can be used to shorten the execution time of a current calculation. For example, Kickstarter [78] enables incremental graph processing for monotonic algorithms like Breadth-First Search and Connected Components by capturing lightweight dependencies across results. Meanwhile, GraphBolt [44] tracks fine-grained dependencies across intermediate values, which allows the incremental computation of non-monotonic algorithms, such as PageRank or Triangle Counting. However, these systems do not allow querying historical data and are not designed for dynamic labeled property graphs. Chronos [27] is an offline snapshot-based temporal engine that supports both historical queries and incremental execution but requires an expensive preprocessing step to retrieve snapshots from disk. Finally, Tegra [31] introduces the Incremental Computation by entity Expansion (ICE) model, which allows sharing arbitrary computations across iterative graph queries. All these incremental approaches account for node and relationship deletions.

3 TEMPORAL GRAPH DATA MODEL

We shall now discuss how we extended the LPG model [61] and Cypher [22] to capture graph evolution over time.

Recall that **LPG** is defined as a pair $G = (V, E)$, where V is a set of nodes and E is a set of relationships. Each node $v \in V$ consists of a tuple $v = (nid, l, p)$, with nid being a unique identifier, l a set of labels that tag the node, and p a set of key-value properties. Each relationship $e \in E$ is represented as a tuple $e = (rid, src, tgt, l, p)$, where rid is a unique identifier, src and tgt denote the IDs of the nodes connected by the relationship, l is a single (or empty) label, and p a set of key-value properties. The relationships are directed from src to tgt. The properties' key is a string; the value can be a string, a primitive data type, or an array type, while labels are strings. A relationship e is considered valid only if its src and tgt nodes are present in G, including when src and tgt are the same. Clearly, when a node is deleted, we must first delete its relationships to transition to a new consistent graph state.

Graph updates. Let U be a universe of graph updates, each an operation of inserting, deleting, or updating a graph entity. We assume these updates construct an infinite sequence of tuples $S = \langle u_1, u_2, \ldots \rangle$. Each tuple is represented as $u = (\tau, id, op)$, where $u(\tau) \in \mathcal{T}$ is a timestamp that denotes when a transaction committed the update [33] (i.e., *system time*), id refers to the unique identifier of a graph entity (node or relationship), and op is the update operation performed. $\mathcal{T}$ is an ordered time domain of discrete positive integer values. We assume that all updates are ordered by their timestamps, which implies that no further changes are allowed on past updates.

A graph entity g can be added to a graph G only if $g \notin G$ at the time of insertion (relationships also require their src and tgt to exist). A graph entity g can be deleted if $g \in G$ during deletion. Updating graph entities refers to inserting, deleting, or updating properties and labels of existing graph entities. All graph updates yield a new valid LPG.

Temporal LPG is defined as a pair $G = (V_\tau, E_\tau)$, in which every node $v \in V_\tau$ consists of a tuple $v = (\tau_s, \tau_e, nid, l, p)$ and every relationship $e \in E$ consists of a tuple $e = (\tau_s, \tau_e, rid, src, tgt, l, p)$. The timestamps τ_s and τ_e represent the start time point (inclusive) and end time point (exclusive) for which the graph entity g is valid, where $\tau_s(g) < \tau_e(g)$. A graph entity insertion for g sets $\tau_e(g) = \infty$, a deletion updates its former $\tau_e(g)$ to a new value, and a property/label modification is considered as a deletion followed by an insertion. As a temporal LPG is created based on a valid sequence of updates S on a consistent graph, it follows the graph update and LPG constraints. As an entity can be removed and inserted at a later timestamp (once or multiple times), a temporal LPG can include entities with the same identifier and non-overlapping time intervals $[\tau_s(g), \tau_e(g))$.

Given the temporal data model definition, we observe that temporal graph analytic queries have two important dimensions: (i) system (transaction) time; and (ii) graph size. With respect to time, we must support queries over a single time point that return a regular LPG or range queries over a time interval that return a temporal LPG. For graph size, we distinguish three access patterns: (i) point queries of a single node or relationship; (ii) subgraph queries (e.g., n-hop neighbourhood); and (iii) global graph queries. An example of a point query over the time dimension is retrieving a node's state at an arbitrary timestamp or its history for a given time interval. Subgraph queries involve processing such as computing the local clustering coefficient or the community evolution over time. Finally, an example of a global query is calculating the PageRank of a social network on a specific day or daily over a month to extract temporal trends.

```
USE GDB FOR SYSTEM_TIME BETWEEN t1 AND t2
MATCH (n: Node)
WHERE id(n) = $id
RETURN n
```

(a) History lookup between t_1 and t_2 (exclusive)

```
USE GDB FOR SYSTEM_TIME AS OF t1
MATCH (n)-[*hops]->(m)
WHERE id(n) = $id
RETURN m
```

(b) Neighbourhood lookup at t_1

```
USE GDB FOR SYSTEM_TIME AS OF t1
MATCH (n: Node)
WHERE id(n) = $id
  AND APPLICATION_TIME CONTAINED IN (t2, t3)
RETURN n
```

(c) Bitemporal node lookup at t_1

Figure 1: Temporal Cypher extensions

Temporal Cypher. Queries in the time dimension are performed using the `USE` clause, an extension to the Cypher graph query language [22]. We enable filtering by transaction time with the keyword `FOR SYSTEM_TIME` – based on SQL:2011 [36] and commercial implementations of temporal tables [66] – alongside an interval specifier that defines the time interval for the query. Thereby allowing the interval to be specified in a variety of ways such as: (i) `AS OF` t_i that returns a valid graph at t_i; (ii) `FROM` t_i `TO` t_j that returns a temporal graph over the interval (t_i, t_j); (iii) `BETWEEN` t_i `AND` t_j that returns a temporal graph over the interval $[t_i, t_j)$; and (iv) `CONTAINED IN` (t_i, t_j) that returns a temporal graph over the interval $[t_i, t_j]$. Fig. 1a shows how a user can retrieve a node's history between $t1$ (inclusive) and $t2$ (exclusive), while Fig. 1b shows how to retrieve the n-hop neighbourhood of an arbitrary node at $t1$.

Bitemporal data model. To support bitemporal [26] LPGs, we have created two additional graph properties: (i) *application start* time to capture the creation of an event; and (ii) *application end* time to capture the deletion of an event. For this work, we assume the user manages the correctness of these properties, as arbitrary future and past dates can be assigned to data. A constraint is that the start time must be less than the end time for all graph entities. Filtering by *application time* in Cypher is expressed by extending the `WHERE` clause with a similar syntax as with *system time*. Fig. 1c shows an example where a user retrieves a node at *system time* $t1$ with *application time* between $[t2, t3]$.

4 TEMPORAL GRAPH STORAGE

Having formalized the temporal graph query design space, we next describe our storage system for persisting the history of dynamic graphs. We find that selecting the optimal strategy is highly workload- and graph-specific, leading to approaches whose performance is optimal only in parts of this space (e.g., fine-grained storage for subgraph history retrieval). Therefore, we have adopted a hybrid storage approach to support general-purpose temporal analytics.

Table 1: Temporal graph API

Access type	Definition in Java notation	Description
Point Queries	`List<Node> getNode(nodeId, start, end)`	Get node history between the given timestamps
	`List<Rel> getRelationship(relId, start, end)`	Get rel. history between the given timestamps
	`List<List<Rel>> getRelationships(nodeId, direction, start, end)`	Get a node's (in/out) relationship history
Subgraph Queries	`List<List<Node>> expand(nodeId, direction, hops, start, end, step)`	Get a node's n-hop history
Global Queries	`List<Entity> getDiff(start, end)`	Retrieve the difference between two time instances
	`List<Graph> getGraph(start, end, step)`	Get the history of a graph between two timestamps
	`Graph getWindow(start, end)`	Filter graph history by a time window
	`TGraph getTemporalGraph(start, end)`	Create a temporal graph

Our idea is to combine and enhance the state-of-the-art storage strategies: (i) *TimeStore* indexes graph updates by time to accelerate full graph reconstruction (i.e., snapshot-based approach); (ii) *LineageStore* indexes updates by the unique ID of their graph entity to enable fast subgraph history accesses (i.e., fine-grained storage). Using these stores, we design a temporal graph system called Aion that chooses between different strategies depending on the prevailing workload. Our design decouples temporal storage from the current working graph to simplify data access and ownership while retaining performance for OLTP use cases.

We shall now introduce the graph API exposed by Aion to enable querying the two dimensions of temporal graphs before discussing its hybrid storage design.

4.1 Temporal Graph API

The goal of the temporal API (summarized in Table 1) is to provide the end-user with a simple and intuitive interface to interact with the two dimensions of time-evolving graphs. We categorize the queries in terms of graph accesses (first column). *Point* queries return a node, a relationship, or the relationships of a node based on a given direction (i.e., incoming, outgoing, or both). *Subgraph* queries are supported using the `expand` method that returns the neighbourhood of a node for n-hops given a specified relationship direction. Finally, *global queries* return: (i) all graph updates between two timestamps (`getDiff`), enabling efficient incremental execution; (ii) LPG snapshots (`getGraph`); (iii) graph windows (`getWindow`); or (iv) temporal LPGs (`getTemporalGraph`).

The temporal dimension of queries is captured using the `start` and `end` transaction timestamp parameters in all methods [3]. If these two parameters are equal, the result of the method call is a single graph entity or the snapshot of a (sub)graph. If `end` is greater than `start`, the result of *point* queries is the history of graph entities. For *subgraph* and *global* queries, the result is a temporal graph. The latter queries, however, require the additional `step` parameter, which specifies how many updates to apply before materializing the subsequent result. For example, `getGraph(1993, 2023, 1-year)` returns thirty more granular snapshots (one per year) instead of creating a new snapshot for every single graph update applied in that time interval.

A user can extract temporal graphs with two distinct representations: a temporal graph (`TGraph`) or a set of regular snapshots (`List<Graph>`) [4]. There exist categories of algorithms that work with either representation.

For example, Fig. 2 shows a simplified version of a temporal graph representing an aviation network [79], where nodes (airports) and relationships (flights) are annotated with time intervals. For example, the interval $[0, 4)$ over node 4 denotes that the airport was open to incoming flights until $t = 3$, and the interval over a relationship depicts a flight departure and arrival time. This representation supports the efficient execution of temporal algorithms, such as describing temporal paths [79] as a topological-optimum problem [29] using a single scan approach instead of performing expensive joins across snapshots. The orange relationships comprise the earliest-arrival path between the airports 0 and 1, while the blue one is the latest-departure path.

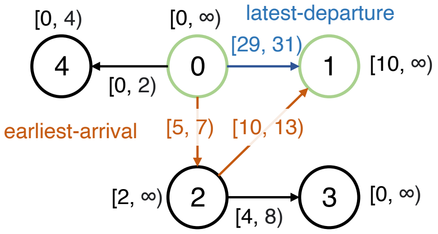

Figure 2: Shortest paths between node 0 and node 1 using temporal information

On the other hand, the incremental graph algorithms discussed in Sec. 2.3 operate on regular dynamic graphs. Therefore, we need support for both representations for efficient temporal analytics.

Regarding graph windows (`getWindow`), we allow users to retrieve a graph snapshot based on a timestamp filter, that is, to retrieve a consistent graph based on all present graph entities between `start` and `end`. This includes the connections of these present nodes that are valid at `start` (i.e., not deleted), even though they are not part of the updates that occurred within the interval. Graph windows enable the extraction of trends with time locality while pruning inactive entities (e.g., e-commerce [38] transactions of a specific week to capture Black Friday sales).

4.2 Modeling Updates Without Wasting Space

When indexing graph updates by time and entity IDs, we must manage both the additional storage requirements and the runtime overhead during data ingestion. In this section, we address the storage issue and leave the discussion for the transactional overhead for Sec. 5. Our target database for this work is Neo4j, which uses fixed-size records to store nodes and relationships [61]. Fixed-size records allow constant time lookups based on offsets into a file (by simply multiplying a record ID by its corresponding record size). However, this is prohibitively expensive to replicate in our design as it can introduce more than 2× storage overhead, maintaining a full copy of the data for both stores.

We have addressed this issue by decoupling Neo4j's general-purpose graph storage format from our temporal storage format. In the temporal graph case, we use variable-size records with two different record types: (i) fully materialized graph entities (e.g., a node with all its labels and properties); and (ii) deltas from the last update (e.g., a property deletion). Fig. 3 shows the three graph entities required to support the API from Table 1. A

---

[3] For brevity, we omit the *application time* parameters for bitemporal analytics.

[4] The snapshots can be computed eagerly or lazily depending on the application.

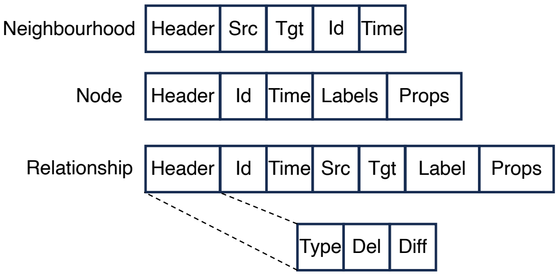

Figure 3: Graph entity storage layout

Table 2: Temporal storage using B+Trees

Store	Entity	Key	Value
Time-Store	graph update	ts	log offset
	graph snapshot	ts	file path
Lineage-Store	node	nodeId, ts	type, labels, props
	relationship	relId, ts	type, label, props
	out-neighbours	srcId, tgtId, ts	relId
	in-neighbours	tgtId, srcId, ts	relId

neighbourhood entity stores the original relationship ID along with the node IDs of its source and target, mapping it back to the source data. For outgoing relationships, the source is stored before the target, and for incoming relationships, the target is stored first.

When storing temporal graph entities to disk, we reserve the first byte (the header) for metadata regarding the entity type (i.e., node, relationship, or neighbourhood) and state (whether it is deleted, represents a delta from a previous update, or neither). To reduce space, all entities track only the transaction timestamp when the insertion, deletion [5], or update occurred (i.e., start time). The end time can be inferred by updates that follow. Instead of storing the strings directly in disk records, we replace them with a reference (4 bytes) to a string store, substantially lowering the size of labels and properties. When storing labels, we first store their number, followed by the actual label references. We reserve the most significant bit of a label's reference to denote whether it is present or deleted. For properties, we follow a similar approach and use the three most significant bits of a property's reference to store information about its state (e.g., deleted) and data type (e.g., int, long, float, string, or primitive data array) to know how to interpret the bytes that follow.

4.3 TimeStore: Indexing by Time

We now describe the design of *TimeStore*, an instance of snapshot-based storage [34] that implements the API from Sec. 4.1. The basic component of *TimeStore* is a log that contains all graph changes (similar to a DB write-ahead log with no retention policy). The changes are ordered by monotonically increasing transaction timestamps. Maintaining a single log for all updates simplifies the design compared to previous solutions [34, 45] at the expense of missed opportunities for storage reduction. The log may contain either fully materialized entries or deltas using the storage format discussed in Sec. 4.2.

To index the log entries and accelerate lookups based on time, *TimeStore* uses a B+Tree, resulting in $O(\log(n))$ accesses, where n is the number of entries. Table 2 shows the key-value B+Tree layout for each log entry: the timestamp of a graph update is the key, and the offset to the log is the value. Nonetheless, graph reconstruction from the beginning of time can be costly, and so *TimeStore* also eagerly creates snapshots based on a user-defined policy such that workload-specific expertise can be injected into the system. The policy can be time-based or operation-based (the number of updates), with the default being operation-based. These snapshots are stored on disk, and references to the files are maintained in a second B+Tree indexed by time. To avoid the I/O cost of reading graph snapshots from disk where possible, we introduce an in-memory Least Recently Used (LRU) cache for snapshots called *GraphStore*.

Algorithm 1: Expand method for *LineageStore* at t

Input: A node id, a rel. direction d, a number of $hops$, and a timestamp t

Output: The result R of node expansion

```
$R \leftarrow \emptyset$
$Q \leftarrow \{id\}$                                    ▷ queue for expansion
for $hop \leftarrow 1, \ldots, hops$ do
    $S \leftarrow \emptyset$                         ▷ set for visited nodes in current hop
    $qsize \leftarrow |Q|$
    for $i \leftarrow 1, \ldots, qsize$ do
        $cid \leftarrow Q.poll()$                                ▷ current id
        $rels \leftarrow getRelationships(cid, d, t, t)$
        for $r \in rels$ do
            if $r \notin S$ then
                $nId \leftarrow r.getNeighbourId()$   ▷ get neighbour id
                $R \leftarrow R \cup \{getNode(nId, t, t), hop\}$   ▷ set hop
                $S \leftarrow S \cup nId$
                $Q \leftarrow Q \cup nId$
```

To retrieve a graph on an arbitrary timestamp, *TimeStore* fetches the snapshot (from disk or *GraphStore*) with the closest timestamp and then applies the forward graph changes to reach the correct state. For multiple consecutive snapshots or a temporal graph creation, it uses the `getDiff` operation to perform a range scan over the log. *Point* or *subgraph* queries require the creation of a snapshot, followed by reading, filtering, and applying all valid updates from the log. This is an expensive operation with graph retrieval outweighing both subgraph access and traversal costs, as we show in Sec. 6.3.

4.4 LineageStore: Indexing by Entity History

With *TimeStore* supporting fast global analytics, we now focus on the second class of queries that access small parts of the graph, including those that comprise a small number of hops. *LineageStore* enables fast history access of graph entities at a node-, relationship-, and neighbourhood-level to node labels and properties using the four B+Tree indexes shown in Table 2. The entries stored in *LineageStore* use a similar layout as in Fig. 3, but their attributes are rearranged to enable temporal ordering when stored as key-value pairs. The keys are composite and are ordered first by entity identifiers, node IDs or source and target IDs for relationships, and then by timestamp.

Instead of storing logical pointers to the log entries of *TimeStore*, we chose to store graph updates in place either as deltas or fully materialized entities. This is encoded as `type` in the entry's value and has the effect of increasing locality for graph accesses. Specifically, using B+Tree range scans, entity history can be retrieved with $O(log(n))$ time complexity, as all updates are ordered by timestamp in the same or adjacent B+Tree pages.

---

[5] Deleted entities require space only for their ID and timestamp of deletion.

For example, to retrieve the complete history of a node starting from a timestamp t, we perform a `nodes.seek(low, high)` range scan, where `low = {nodeId, t}` and `high = {nodeId, ∞}`. In Sec. 6.5, we discuss how to select a materialization strategy that accounts for the reconstruction cost of a graph from deltas.

While updating node history is straightforward, relationship updates are more complicated: relationship creation or deletion also requires updating `in-` and `out-neighbours` indexes. As an alternative design, particularly for densely connected nodes, we experimented with a pointer-based representation, creating a double-linked list of relationships by storing logical pointers within a relationship [61]. However, that design significantly increased the storage requirements and the complexity of relationship updates compared to maintaining two separate neighbourhood indexes and was not chosen for our implementation.

Point and *subgraph* queries are translated directly to index lookups. Alg. 1 shows the implementation of the `expand` method for a single time point `t`, where we assume that `start` is equal to `end` for simplicity. In line 8, *LineageStore* retrieves the relationships of nodes with direction `d` at timestamp `t` by performing a range scan over the `in-` and `out-neighbours` indexes followed by a range scan over the `relationship` index to reconstruct the correct entity versions. Then, in line 12, if the neighbour (either source or target node ID depending on the direction `d`) has not been visited for that `hop`, it is added to the final result `R`. *Global* queries require an all-nodes scan with one-hop expansions. Therefore, their processing cost depends solely on the graph history size.

4.5 Handling Application Time

To accelerate lookups for the *application time* dimension, Aion can use the hybrid store as with the *system time*. However, this significantly penalizes ingestion cost as it requires updating multiple indexes, increases query complexity since it produces multiple execution plans, and adds storage overhead. Furthermore, as *application time* is user-defined, data may arrive out-of-order, requiring a watermark strategy [2] to guarantee when data is safe to read. Therefore, we decided to store *application start* and *end* time as graph properties. When querying with both time dimensions, a valid (sub)graph with respect to *system time* is retrieved first, and then a filter is applied for the *application time*. If the *application time* is not set as a property, we fall back to using the *system time*.

5 AION ARCHITECTURE

While hybrid temporal storage can provide efficient graph accesses, a temporal graph DBMS must: (i) adapt its execution strategies to be sympathetic to the workload characteristics; (ii) store temporal information with low overhead for write transactions; and (iii) limit redundancy-prone computations of temporal range queries. We describe the architecture of Aion, a graph system based on the hybrid storage from Sec. 4 that extends Neo4j to provide transactional guarantees for time-evolving queries. Backing Aion's storage with Neo4j's B+Tree implementation [53] offers sortedness, scalable accesses, out-of-core storage, and seamless integration with the page cache for increased performance.

In this section, we provide an overview of Aion's architecture (Sec. 5.1) and introduce an efficient in-memory LPG representation that enables incremental graph execution (Sec. 5.2). Lastly, we describe how Aion manages its own memory to avoid unnecessary managed language overheads (Sec. 5.3) from the underlying runtime.

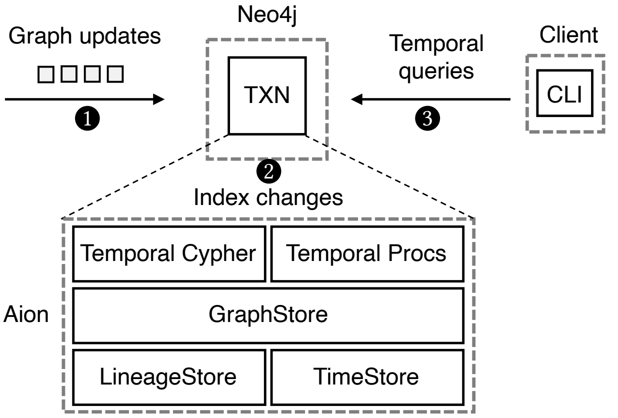

Figure 4: Aion architecture

5.1 Overview

Fig. 4 shows how Aion augments Neo4j with a hybrid store that consists of three separate indexable components: (i) the *GraphStore* that maintains a set of (temporal) graph snapshots based on LRU policy; (ii) the *TimeStore* that serves global queries; and (iii) the *LineageStore* that serves point and small subgraph queries. By storing temporal updates separately from Neo4j's data, Aion allows OLTP query execution over the latest graph version with no overhead in the common use case. To avoid adding the overheads for updating all three stores on the critical path of each write transaction, Aion only applies updates to the *TimeStore*, which, in turn, cascades them to both the *GraphStore* and *LineageStore* in the background.

Graph updates are passed to Aion from Neo4j via an event listener that is registered with the Neo4j database management service. The event listener is triggered in the after-commit phase of each write transaction at stage ❶. Each event provides Aion with access to all changes that are to be applied by the transaction and guarantees that: (i) updates are assigned a valid transaction time; and (ii) the constraints of Sec. 3 are satisfied, as committed transactions always result in a consistent labeled property graph. In the event that the transaction aborts, any changes can be rolled back and the transaction retried.

Stage ❷ is responsible for writing all changes to Aion's hybrid temporal store as part of running transactions. Since indexing graph updates by both time and entity IDs leads to performance overheads for write transactions (see Sec. 6.4), this stage is designed to provide low-latency transactional guarantees for temporal queries by employing a two-step process. First, only the *TimeStore* is updated synchronously; then, background workers asynchronously apply outstanding updates to the *LineageStore* and, if necessary, insert new snapshots into the *GraphStore*. Consequently, the *LineageStore* lags behind the *TimeStore*, and in the rare case that it cannot serve a temporal query, the *TimeStore* is used instead, which may incur a performance penalty. [6] Finally, recovering from failures is handled by replaying the transaction log from the last persisted transaction time to get a consistent state. As such, Aion maintains fault tolerance (in a single machine or cluster).

To access historical data, users submit their queries using temporal Cypher or procedures in stage ❸, executed as part of a transaction. Temporal Cypher is parsed using javaCC [32] and translated into an operator plan. Based on the cardinality

---

[6] We analyze the performance of the two stores in Sec. 6.2 and Sec. 6.3.

estimation of this generated plan, Aion adopts a simple heuristic to select between the two temporal stores: [7] (i) if less than 30% of the graph is accessed, Aion uses the *LineageStore*; (ii) otherwise, it constructs a full graph snapshot with the *TimeStore*.

Cardinality estimation. Aion uses histograms to track base statistics, including the number of: (i) nodes and relationships; (ii) nodes with a specific label; (ii) relationships with a specific type; (iv) relationships with a predefined pattern (e.g., `(:Label)-[:Type]->()`). Using these base statistics, it can derive the cardinality of more complex patterns, such as `#((:A)-[:R]->(:B)) = min(#((:A)-[:R]->()), #(()-[:R]->(:B)))`, and estimate the percentage of the graph history accessed.

Snapshot replication. One option for storing snapshots for *TimeStore* was to utilize Neo4j's functionality of persisting full graph snapshots or deltas since a past graph version [56]. Yet, this involves long-running read transactions to access graph entities and copies metadata (e.g., indexes) not required for temporal analytics, which is overbearing for small write transactions, incurring up to seconds of increased latency. To avoid the problem, we maintain the latest graph in-memory using the *GraphStore*, similar to an HTAP approach [65] by synchronously applying all committed graph updates. This allows faster snapshot replication to memory and disk storage without expensive read transactions.

Graph analytics using temporal procedures. Aion wraps the functionality exposed in Table 1 with temporal procedures (i.e., functions invoked from Cypher). It also allows the creation of static CSRs, known as graph projections, to be able to exploit the efficient parallel versions of the GDS library's algorithms [57]. Additionally, Aion supports the execution of algorithms directly on in-memory snapshots. This latter approach can be efficient for large graphs, as it does not involve the step of graph projection. Furthermore, executing algorithms on in-memory snapshots provides the basis for incremental computation, as discussed in the following section.

General applicability. We designed Aion as a standalone temporal management solution that can be integrated into other graph DBMSs by installing Aion's event handlers [8] in the DBMS, and exposing the temporal API to their users via the query language or UDFs. For any system that cannot integrate this way, Aion is implemented from a set of four standard Neo4j components: (i) Neo4j's B+Tree implementation for storage; (ii) the event listeners for the integration with the transaction layer; (iii) Cypher for its frontend; (iv) GDS projections for static graph analytics. Hence, an implementer can switch out each component for readily available versions in their system. For example, B+Trees can be replaced by any persistent key-value store that allows composite key ordering, out-of-core storage, and range scans, such as RocksDB or other Log-Structured-Merge stores.

Although our current implementation extends Cypher (see Sec. 3), other query languages are easily integrated by exposing the temporal API calls from Table 1. For instance, in an imperative language, such as Gremlin, we could provide extensions similar to previous work [10] and map the Gremlin steps to our API. Finally, by extracting the graph history into GDS projections, Aion creates static CSRs backed by byte arrays or Apache Arrow format that can be used by other libraries or many-core GPGPUs.

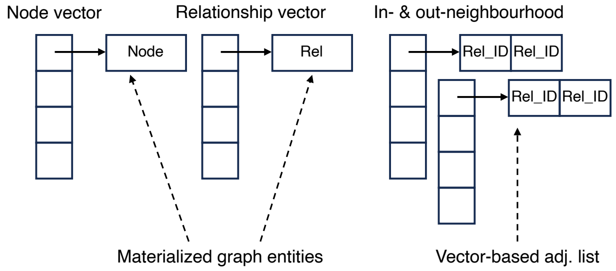

Figure 5: In-memory data structures

5.2 Incremental Graph Computations

We next discuss how Aion enables efficient incremental graph computations, starting with its in-memory dynamic graph representation. Static CSR representations can not handle dynamically changing LPGs since not all nodes have the same number of properties or property data types. In addition, while previous work [72] shows that on-the-fly static CSR creation is feasible for analytical engines, the overhead for OLTP systems is significant, especially so for large graphs, as it requires locking at node and relationship granularity. Therefore, we adopt an adjacency list-like design for our dynamic representation.

The in-memory graph consists of four data structures shown in Fig. 5: (i) a vector of materialized nodes (each containing all labels and properties); (ii) a vector of materialized relationships; (iii) a vector of the incoming relationship IDs for each node; and (iv) a vector of the outgoing relationship IDs for each node. Our design is based on the Sortledton [23] graph data structure but can handle an arbitrary number of labels and properties using the materialized graph entities' vectors. The four vectors provide a compact memory representation with fast operations: $O(1)$ time for entity insertion or update, and neighbourhood access. Only node and relationship deletions can be more expensive depending on the updated neighbourhood size, which we can amortize using gaps [19]. For parallelization, no read-write locks are required, as updates are performed using key partitioning (e.g., by node ID) and reads always precede writes for analytics. In addition, the data structures are resized according to the maximum node ID seen from the updates during key partitioning without locking to avoid heavy performance penalties.

To compact sparse graphs into a denser format, Aion uses a map to translate from a sparse domain of node IDs $[0, V_s)$, where only a subset of IDs refer to a valid node, to a dense domain $[0, V_d)$, where all IDs refer to valid nodes. This dense format enables efficient graph algorithms designed to store and retrieve data from vectors.

Despite this compact in-memory representation, storing multiple graph snapshots in *GraphStore* is still challenging from an implementation perspective. Neo4j maintains a page cache over persisted data (e.g., B+Tree pages, metadata, or entities participating in transactions), which limits the amount of available memory allocated for *GraphStore*. To further lower the storage requirements of in-memory graphs, we utilize the following optimizations: (i) *GraphStore*'s snapshots do not store their neighbourhoods in the respective vectors. Instead, they are computed on the fly with parallel construction when a snapshot is retrieved; (ii) when copying large graphs from the *GraphStore*, Aion uses Copy-on-Write (CoW) similar to Tegra [31] to avoid unnecessary data duplication; and (iii) in- and out-neighbourhood vectors do

---

[7] As future work, we want to develop an adaptive decision model for graph workloads with different characteristics.

[8] Event listeners are considered a commonplace pattern to intercept the lifecycle of transactions and adapt their behavior in application frameworks, such as Spring, or commercial databases, such as Oracle or Memgraph [47].

not store the source and target node IDs and an $O(1)$ lookup is required from the relationship vector to retrieve them.

While the dynamic representation above is sufficient for LPG storage, to store temporal graphs, we perform the following changes: (i) the node and relationship vectors store a list of entity versions instead of a single object; (ii) in- and out-neighbourhood vectors store all neighbourhood history for each entity. Every graph modification is modeled as a record append at the end of the respective adjacency lists. Thus, data is ordered by timestamp, allowing logarithmic-cost history access.

Incremental algorithms. Based on this graph representation, incremental algorithms are implemented as temporal procedures that materialize intermediate results and call the `getDiff` method between iterations. AION reuses the intermediate results to avoid redundant operations when analyzing consecutive snapshots. The intermediate and final results can be stored in *GraphStore* for efficient access by subsequent queries [31]. Similar to the *global* queries, incremental algorithms require the `step` parameter to produce results for a batch of data and not every update.

AION supports three categories of incremental algorithms: (i) non-holistic aggregations [70] such as the average value of a node or relationship property; (ii) monotonic path-based algorithms like Breadth-First Search (BFS) or Single-Source Shortest Path (SSSP); and (iii) non-monotonic algorithms that can converge to correct results independently of node initialization such as PageRank or Graph Coloring. For aggregations, we employ techniques from stream processing [71, 74] for efficient execution. Path-based and non-monotonic algorithms require more expensive dependency tracking, especially in the event of deletions. More specifically, for the monotonic path-based algorithms, we use the tag and reset technique [78], where deleted nodes are tagged, and their value is reset before propagating the tags to the remaining graph. For the last category of graph algorithms, we use the optimizations introduced in [77] and propagate changes based on dependencies between iterations.

5.3 Memory Management

Production-scale DBMSs introduce systemic overheads that increase the memory footprint and CPU cycles of database operations. In addition, systems implemented in managed languages (e.g., Java for Neo4j) can suffer from dynamic allocation and garbage collection penalties, which rapidly multiply with the systemic overheads (e.g., transactional or networking stack). To reduce the performance degradation of the memory-intensive incremental graph queries, AION minimizes memory allocation on the critical path. It utilizes statically allocated object pools, such as byte arrays for disk operations or roaring bitmaps [14] for algorithms. In addition, each worker thread maintains a separate object pool to avoid contention. To further lower the memory footprint, AION utilizes primitive collections [18] and replaces: (i) queues with circular buffers of pre-allocated objects, and (ii) maps with custom array implementations.

6 EVALUATION

In this section, we evaluate the performance and footprint of AION when processing temporal queries. We demonstrate that AION achieves comparable or better performance compared to state-of-the-art temporal analytics engines like Raphtory and Gradoop (Sec. 6.2) and the Enterprise Edition of Neo4j, a general-purpose non-temporal graph database. We then study the performance (Sec. 6.3) and overheads (Sec. 6.4) of hybrid storage before presenting a materialization strategy of deltas to reduce the cost for history reconstruction (Sec. 6.5). Finally, we show the performance of incremental algorithms (Sec. 6.6) with AION and provide an end-to-end system evaluation by submitting Cypher queries over Bolt (Sec. 6.7), as an end-user or client application would use AION.

6.1 Experimental Setup

All experiments are performed on an m5.8xlarge AWS EC2 instance with 32 physical cores, a 35.8 MiB LLC, 128 GiB of memory, and EBS [4] for storage (500 MB/s write bandwidth; 8k IOPS). We use Amazon Linux 2023 with kernel version 6.1, Corretto OpenJDK17, and rustc 1.72.0.

Datasets. For our evaluation, we use six real-world graph workloads to provide diverse scenarios for rich coverage: (i) DBLP [82] is an undirected co-authorship network, in which we replace relationships (s, t) with two directed ones, i.e., $s \rightarrow t$ and $t \rightarrow s$; (ii) WikiTalk [40] is a temporal network that captures the edits in Talk pages between Wikipedia users; (iii) Pokec [69] is an online social network; (iv) LiveJournal [7] represents an online community of users maintaining journals and blogs; (v) DBPedia [6] is the hyperlink network of Wikipedia with pages as nodes and hyperlinks as relationships; and (vi) Orkut [48] is another social network, in which we also replace undirected relationships with two directed ones as with DBLP. Apart from the WikiTalk graph, all other datasets are non-temporal, and so we have enriched their nodes and relationships with timestamps. To achieve this fairly, we load and shuffle all relationships, assign them monotonically increasing timestamps, and consume them in timestamp order to emulate relationship additions over time, where node creation always precedes the creation of any incident relationships.

Table 3 summarizes the graphs with their properties (including the number of nodes or relationships and the average degree) and their in-memory size in Neo4j and AION. For Neo4j, the size is measured as in [54] with additional bytes for JVM object headers and without accounting for indexes or other metadata stored on disk. For Aion, we use around 60 B and 68 B for nodes and relationships, respectively, and 4 B for each entry stored in the in- and out-neighbourhood vectors (see Fig. 5).

Graph database systems. We compare to (i) Raphtory (v0.5.6) [59], which supports efficient fine-grained accesses; (ii) Gradoop (v0.6.0) [62], a model-based temporal engine for distributed analytics that uses Flink [12]; and (iii) Neo4j Enterprise Edition (v5.7.0), a system without temporal capabilities. [9] For *global* queries, we use the TimeStore implementation, which is a throughput-optimized variation of the Copy-Log approach exhibiting the best performance from existing snapshot-based techniques at the expense of additional space overhead (as shown in recent work [45]). Regarding the memory configuration of AION, we reserve 32 GB for the JVM heap for object allocation, 40 GB for Neo4j's page cache, and 32 GB for the *GraphStore*. The remaining memory is reserved for OS operations and the client threads interacting with the database.

6.2 System Comparison for Graph Accesses

To study the efficiency of temporal storage and retrieval, we use the six graphs from Table 3 and measure the throughput of: (i) *point* queries that retrieve random relationships at arbitrary

---

[9] The code for other storage approaches [27, 31, 34, 45, 76] was not available at the time of writing.

Table 3: Evaluation Datasets

Dataset	Domain	|V|	|E|	|E|/|V|	Directed	Neo4j (in-memory)	Aion (in-memory)
DBLP [82]	citation	0.3 M	2.1 M	7	✗	180 MB	175 MB
WikiTalk [40]	communication	1 M	7.8 M	7.8	✓	667 MB	650 MB
Pokec [69]	social	1.6 M	30 M	18.8	✓	2436 MB	2338 MB
LiveJournal [7]	social	4.8 M	69 M	14.4	✓	5520 MB	5314 MB
DBPedia [6]	hyperlink	18 M	172 M	9.5	✓	14251 MB	13803 MB
Orkut [48]	social	3 M	234 M	78	✗	18068 MB	17209 MB

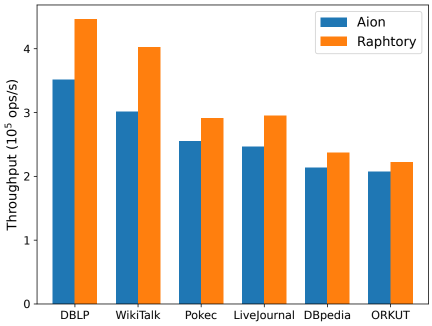

Figure 6: Fetching random relationships

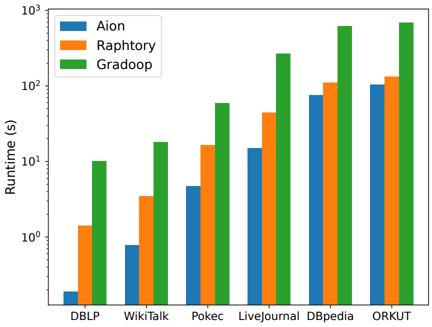

Figure 7: Fetching random snapshots

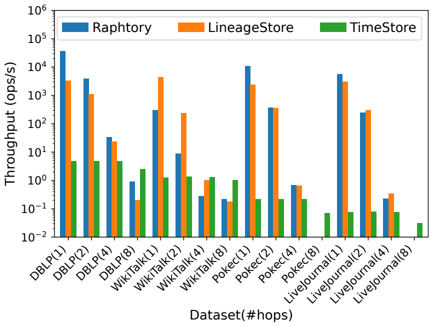

Figure 8: N-hop graph accesses

time points; [10] and (ii) *global* queries that fetch full graph snapshots at random timestamps. For the *point* query workload, we compare Aion against Raphtory and omit the results of Gradoop, as they are orders of magnitude worse for single-entity lookups. Given that Raphtory cannot support multigraphs (graphs that permit multiple relationships between the same source and target nodes, it loads only 42% and 79% of the relationships for WikiTalk and DBPedia datasets. This results in smaller overhead when performing an all-history scan for global queries and incorrect output for these graphs. Even though Raphtory supports a more restrictive graph model with a sole focus on in-memory analytics, it provides an upper-performance bound for Aion. For *global* queries, we compare against both Raphtory and Gradoop. We average the results obtained from all systems over 1 M runs for *point* queries and 100 runs for *global* queries.

***Point* queries.** Fig. 6 shows the throughput of *point* queries/ s, for which Raphtory is optimized. Specifically, Raphtory retrieves relationships by performing constant time lookups over in-memory arrays and filtering them by timestamp, while Aion retrieves graph entities from page-backed B+Trees with logarithmic complexity. We observe that for small graphs, such as DBLP and WikiTalk (only 3 M relationships loaded), Raphtory exhibits around 30% better throughput because large parts of the graph are retrieved from the last-level cache with fewer CPU cycles compared to using B+Tree reads over the page cache (data fits in memory).

For larger graphs, the performance difference drops below 7% because of the expensive checks that Raphtory performs to validate whether graph entities are visible at a specific timestamp. This includes scanning all incoming and outgoing relationships for each node, which is costly for nodes with a long history of updates. On the other hand, Aion scales well with the dataset size by accessing only the relevant graph history segments based on its efficient memory management and storage design. Given that typical graphs have more than 100 M entities from our experience with Neo4j deployments, we consider Aion's performance comparable to Raphtory for *point* queries.

***Global* queries.** The runtime measurements of Fig. 7 show that Aion outperforms Rapthory and Gradoop for *global* queries. In particular, for DBLP, WikiTalk, Pokec, and LiveJournal, Aion yields 7.3×, 4.5×, 3.5×, and 3× better throughput compared to Rapthory, respectively. For these datasets, Aion retrieves a snapshot from the *GraphStore* using CoW and loads only a small amount for graph updates from the log stored on disk. On the other hand, Raphtory performs an all-history scan followed by an expensive filter to construct a global snapshot (i.e., checking the history of relationship updates per node).

For the larger graphs (DBPedia and Orkut), Aion yields only 30-50% better performance as the *GraphStore* cannot cache multiple snapshots because of the dataset sizes shown in Table 3. By retrieving snapshots at random time points, Aion regularly evicts and loads new snapshots to the *GraphStore* from disk while replaying updates using the *TimeStore*'s log. In our current implementation, the data stored in the log is not buffered in memory and instead is read from disk. In addition, snapshots are reconstructed serially to avoid inconsistencies (e.g., a deletion happening before an addition).[11]

Compared to Gradoop, which parallelizes snapshot retrieval to all available cores, Aion achieves 6.6-52.2× lower runtime. To construct a snapshot, Gradoop performs a parallel scan and filter over the node and relationship tables (backed by CSV files), followed by two parallel join transformations required to remove dangling relationships from the produced subgraph. Gradoop spends nearly 80% of its time on this last verification step. Despite model-based approaches being embarrassingly parallel, performing an all-history scan along with complex logic to verify the result makes them prohibitively expensive, especially for low-latency (transactional) workloads.

Discussion. In Table 4, we summarize the space and time complexities of Aion, Raphtory, and Gradoop to provide a more

---

[10] As retrieving random nodes at arbitrary time points is a symmetrical operation and the number of nodes in the datasets from Table 3 is relatively smaller than the number of relationships, we omit the results for node retrieval.

[11] We want to investigate parallel log replay [80] to accelerate the process in the future.

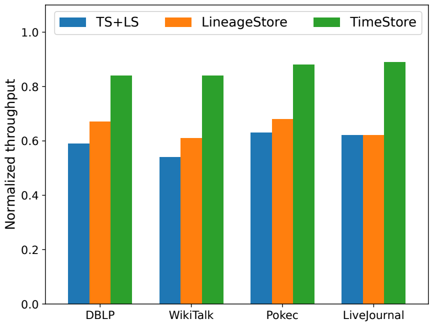

Figure 9: Ingestion overhead

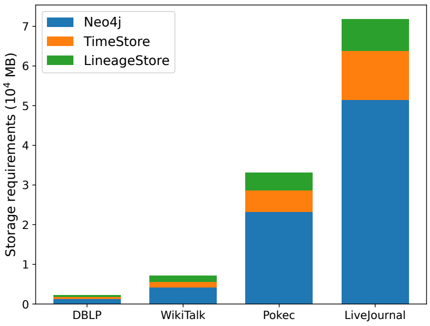

Figure 10: Temporal storage overhead

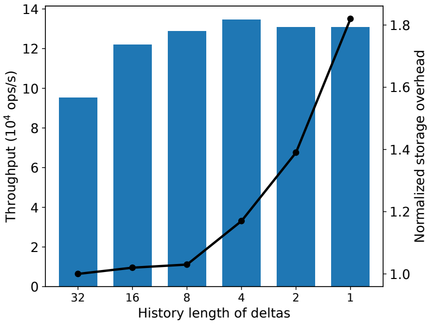

Figure 11: Materialization strategy

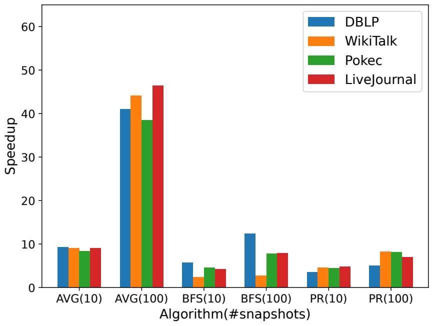

Figure 12: Incremental execution

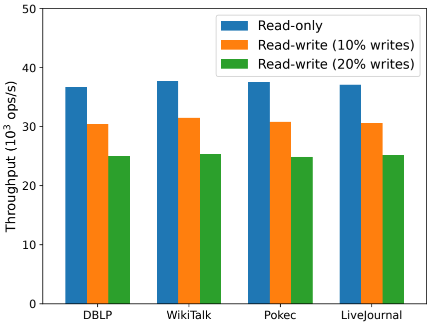

Figure 13: Transactions using Bolt

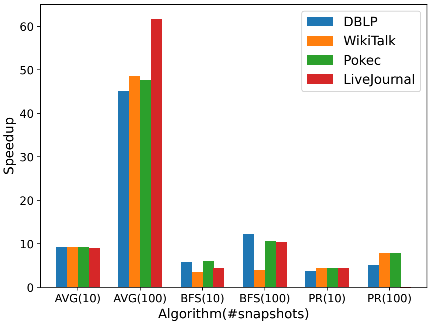

Figure 14: Speedup with procedures

Table 4: Storage and retrieval costs of graph database systems. $|U|$ = number of graph updates; $|U_R|$ = number of relationship updates; $|U_R^n|$ = number of relationship updates for node n; $|G|$ = size of a graph snapshot; k = number of snapshots; $\delta(|U|)$ = range scan over updates, with $log(|U|)$ lookup cost and $|L|$ updates replayed ($|L| <= |U|$).

System	Space	Rel. retrieval	Snapshot retrieval	Persistent
Aion	$2\lvert U\rvert + k\lvert G\rvert$	$log(\lvert U_R\rvert)$	$\lvert G\rvert + \delta(\lvert U\rvert)$	✓
Raphtory	$\lvert U\rvert$	$2\lvert U_R^n\rvert$	$\lvert U\rvert$	✗
Gradoop	$\lvert U\rvert$	$\lvert U_R\rvert$	$\lvert U\rvert$	✗

detailed analysis of the performance results presented above. In terms of space requirements, both Raphtory and Gradoop store the graph updates $|U|$ once in memory. Aion stores the graph updates twice (i.e., in *LineageStore* and *TimeStore*) along with an arbitrary number of k snapshots for *TimeStore*. Its data structures are persistent and enable the support of out-of-core use cases and failure recovery out-of-the-box.

For relationship *point* queries [12]: (i) Gradoop has to scan all the relationship updates ($|U_R|$); (ii) Raphtory checks whether the start and end nodes are visible at a given time by linearly scanning their relationship updates from vectors ($|U_R^n|$); and (iii) Aion performs a B+Tree lookup over all updates in $log(|U_R|)$ using the *LineageStore*. This explains why Gradoop cannot handle efficiently *point* queries, and the performance of Raphtory degrades with the size of the graph history. For *global* queries, both Raphtory and Gradoop must scan all the updates. Instead, with *TimeStore*, Aion has to copy the most relevant snapshot ($|G|$) from in-memory or disk, search the offsets of the remaining updates in $log(|U|)$ using a B+Tree, and then load and replay them from disk. Overall, we observe that Aion achieves good performance for both *point* lookups and *global* graph accesses at the expense of employing additional storage space.

6.3 Comparing Temporal Stores

Next, we explore the throughput of *TimeStore* and *LineageStore* in subgraph queries to identify a threshold for choosing between the two implementations when Aion generates a physical execution plan. We also compare against Raphtory to observe the limitations of another fine-grained storage approach, apart from *LineageStore*, for large subgraph queries. We average the results obtained from all solutions over 100 runs of n-hop queries that start from random nodes (see Alg. 1 for *LineageStore*). The number of hops ranges from one to eight.

Fig. 8 shows that for one- and two-hop queries, *LineageStore* and Raphtory achieve between two and three orders of magnitude better performance compared to *TimeStore*. In addition, Raphtory outperforms *LineageStore* for one-hop queries by 11×, 1.2×, and 1.8× for DBLP, Pokec, and Livejournal, while being 14× slower for WikiTalk. Still, with additional hops, its performance becomes comparable or worse than *LineageStore* because of accessing larger parts of the graphs and performing expensive checks for time validity, as discussed in Sec. 6.2.

For four hops, if the n-hop query accesses more than 30% of the graph, *TimeStore* yields comparable performance against *LineageStore* and Raphtory. Otherwise, it is one order of magnitude slower. This happens because *TimeStore* materializes a full graph snapshot while accessing only a subgraph, and the materialization cost outweighs the traversal cost.

When we double the hops from four to eight, every node is accessed on average 9× for DBLP, Pokec, and LiveJournal, and 2× for WikiTalk. For this workload, *LineageStore* and Raphtory are up to 12× and 5× slower or time out (i.e., requires more than five hours to complete for Pokec and LiveJournal). To allow Aion to handle subgraph queries efficiently and avoid such situations,

---

[12] The time complexity for node retrieval is symmetrical.

we adjusted the query planner to choose an execution strategy based on the estimated cardinality: Aion chooses *LineageStore* if the cardinality is less than 30% of the graph, and *TimeStore* otherwise.[13] As a result, Aion can adapt its execution to the workload characteristics of different temporal queries.

6.4 Data Ingestion and Storage Overhead

In this experiment, we evaluate the overhead of the hybrid store when the temporal infrastructure is integrated with the transaction processing flow of the host Neo4j database. To normalize the runtime overhead measurement, we compute the throughput of Neo4j without temporal storage and use it as a baseline for the cost introduced by the temporal stores. Following the best practices for write transactions [51], we batch updates together to increase performance in batches of 1000 transactions and perform inserts with 32 client threads.

In Fig. 9, we show the normalized throughput when synchronously updating the temporal store with each write transaction. As all approaches perform worse than the baseline, the solutions with lower overhead are closer to one. We compare the following approaches: (i) updating both stores (i.e., *TS + LS*); (ii) updating only *LineageStore*; and (iii) updating only *TimeStore*. When using both stores, we observe a 40% ingestion throughput decrease caused by the *LineageStore*. Its indexes (see Table 2) are more expensive to update because of the composite key comparisons, especially for B+Trees with multiple levels. In addition, materializing graph deltas (see Sec. 6.5) decreases the ingestion throughput, as Aion must lock the entire B+Tree to guarantee safe access to a key range. Instead, if Aion uses only *TimeStore*, the throughput is lowered by less than 15%, even though all writes are serialized to disk with the log. Therefore, to provide continued OLTP execution at the prevailing work rate, Aion updates synchronously the *TimeStore* and off the critical path the *LineageStore*, as discussed in Sec. 5.1. This allows Aion to use the appropriate store based on the workload characteristics at runtime (see Table 4).

Fig. 10 shows the storage overhead introduced by the hybrid store. Apart from the space required to store the graph data on disk (see the Neo4j column in Table 3), Neo4j requires additional space to store indexes (e.g., accessing entities by label), graph metadata, and transaction logs, retained for recovery purposes. In particular, Neo4j uses 6-9× more space than the original graph size, with the highest fragment of the storage cost attributed to the transaction logs. Compared to the total storage cost of Neo4j (reported with the blue bar), Aion increases the disk footprint between 29-41% for all datasets, with one-quarter of that overhead attributed to serialized graph snapshots. This experiment confirms the benefits of using variable-size records and deltas in Aion, which incur a modest storage overhead independent of the additional 2× data redundancy. In future, an approach to further reduce this overhead would be to compress data based on its characteristics (e.g., value distribution).

6.5 Materializing Graph Entities

In the previous experiments, we assumed a single version for each node and relationship as we performed only additions without updates or deletions. Next, we want to evaluate a strategy for choosing how many deltas we need to store before materializing entities. As discussed in Sec. 4.4, deltas significantly reduce the storage requirements of *LineageStore*. Meanwhile, they increase the cost of reconstructing a valid entity version (e.g., a node with its latest properties and labels) because *LineageStore* has to read and merge more B+Tree entries.

In Fig. 11, we use the DBLP graph and create history chains for its relationships by adding thirty-two new properties into the graph at different discrete times. We then materialize the relationships for every update (equivalent to a chain threshold of one), followed by every two, four, eight, and sixteen updates. No materialization (thirty-two implies we only maintain deltas without materialization) results in up to 40% lower throughput, which deteriorates with the increasing number of deltas. Conversely, materializing entities on every update increases storage up to 80% (see the black line in Fig. 11). However, the large key-value pairs created from the repeated materialization result in more page reads (i.e., fewer pairs fit in a single B+Tree page), affecting performance adversely. Finally, we observe that materializing deltas every four updates appears to strike the best balance for this workload with only 16% storage increase, and we, thus, adopt this materialization strategy for Aion.

6.6 Incremental Query Execution

In Fig. 12, we measure the speedup of incremental graph computations over regular execution for either ten or hundred consecutive snapshots. We use DBLP, WikiTalk, Pokec, and LiveJournal datasets to vary the number of graph entities (i.e., from 2.4 M to 73.8 M) and the average degree of the graph. More specifically, for each graph, we load half of the relationships to the first snapshot, divide the remaining ones into a hundred increments, and apply one batch at a time to create the subsequent snapshots. We evaluate three classes of algorithms: (i) running global average (*AVG*) over a relationship property; (ii) Breadth-First Search (*BFS*) using random nodes as a starting point; and (iii) PageRank (*PR*) that runs either for up to one hundred iterations or until a convergence threshold is reached, which we set as $\epsilon = 0.01$.

The computation of the running global average requires maintaining only a counter and a sum over all active relationship properties. No expensive dependency tracking is required for deletions, which results in up to 9× and 46.5× performance speedup for all graphs with 10 and 100 snapshots, respectively. BFS and PageRank exhibit lower speedups (between 2.3-12× and 3.5-8.3×) because changes must propagate through the graph to compute the subsequent result. Specifically, PageRank requires completing all iterations (or reaching convergence) for the affected nodes, which impacts performance. Nonetheless, this experiment shows that: (i) our dynamic graph data structures can efficiently handle updates while enabling incremental graph analytics for labeled property graphs; and (ii) using more snapshots increases the opportunities for exploiting past computations.

6.7 Temporal Cypher over Bolt

In the previous experiments (Secs. 6.2, 6.3, and 6.6), we accessed Aion using the embedded mode of Neo4j which binds directly to the running binary in-process. However, because of the additional worker threads dedicated to query compilation, transaction management, and networking, these execution layers increase ITLB, data, and instruction cache misses, thus potentially lowering performance. To explore how Aion's design copes with such systemic overheads, we evaluate it using temporal Cypher queries in a more typical client-server arrangement over Bolt (Neo4j's communication protocol).

---

[13] See Table 4 for the time complexity of retrieving individual relationships (*LineageStore*) and graph snapshots (*TimeStore*).

In Fig. 13, we emulate three OLTP workloads in which 32 client threads (each pinned to one available CPU core) perform read and write transactions using Cypher. The reads retrieve temporal graph entities at arbitrary time points, and the writes create or update nodes and relationships, thus updating AION. We use the following read-write ratios: (i) read-only workload, (ii) 10% writes, and (iii) 20% writes. For the read-only workload, we observe that AION exhibits similar performance for all workloads with close to 37 K queries/ s and saturates the throughput of read-only Neo4j transactions with Bolt. When introducing 10% writes, the throughput drops by 20%. With 20% writes, the drop is nearly 35% compared to the read-only scenario. Overall, AION can handle mixed transactional workloads efficiently.

Next, we evaluate the speedup of incremental graph computations over classic Neo4j. Compared to the previous experiment, these Cypher queries can be considered long-running read-write transactions [15], and we study their performance in isolation by running a single global query at a time. Similar to the GDS [57] implementation of graph algorithms, we use procedures with dedicated pools of worker threads and memory. For fairness, instead of constructing a static CSR for each snapshot, we execute analytics on top of our dynamic graph representation (Sec. 5.2) and store the results in *GraphStore* to avoid serialization overheads (i.e., users can later request parts or full results). Fig. 14 shows that incremental execution over procedures yields higher speedups for global average (9-61×) and BFS (3.5-12×) compared to Sec. 6.6 because it removes repetitive query compilation and task scheduling overheads. Finally, both experiments with Bolt demonstrate the importance of our memory reduction optimizations for efficient end-to-end temporal analytics, as multiple snapshots and results can be stored in memory along with Neo4j's page cache.

7 RELATED WORK

Graph analytics systems, such as Pregel [43], PowerGraph [25], GraphLab [41], Giraph [17], and GraphX [81], focus on high-performance static graph processing by scaling out to a cluster of nodes. Gradoop [62] is a temporal distributed graph engine atop Flink [12]. Raphtory [67] provides fine-grained in-memory temporal storage without transactional or multigraph support. However, both Gradoop and Raphtory require an all-history scan followed by a filter to retrieve valid (sub)graphs at arbitrary time points. Unlike this design, AION supports general-purpose temporal analytics with efficient local pattern-matching and global query execution.

Streaming graph systems, such as GraphInc [11], Kineograph [16], Kickstarter [78], and GraphBolt [44], enable efficient analytics on streaming graphs without the capability of querying the graph history. Compared to our tag and reset approach for incremental execution, Kickstarter [78] uses lightweight dependency tracking that enables pruning unnecessary computations. GraphBolt [44] performs analysis over non-monotonic algorithms, while AION can only handle algorithms that converge to correct results independently of node initialization. Both approaches could be integrated into AION to generalize incremental graph computations. Naiad [49] enables non-monotonic incremental execution by indexing the data differences in its computation model. However, Naiad is not specialized for graph operations [44], and it cannot handle efficiently historical queries [31].

Dynamic graph data structures. Llama [42] and Teseo [19] have a CSR-design, while Stinger [21], GraphOne [37], Livegraph [83], and Sortledton [23] use adjacency lists for dynamic graph storage. These graph representations offer different functional characteristics, such as read- [23] versus write-optimized [19, 42] storage or transactional guarantees [19, 23, 83]. Apart from Livegraph and Stinger, the remaining representations are not designed for dynamic LPGs and, thus, are unsuitable for AION. Still, Livegraph and Stinger introduce runtime (i.e., concurrent read and write accesses) and memory overheads (e.g., over 100 GB memory consumption for ingesting a graph similar to Orkut [19, 23]) that are prohibitive for maintaining multiple graph versions in memory. Instead, AION uses a more compact memory representation and handles concurrency at the execution level (i.e., updates precede reads).

Temporal graph analytics. While model-based approaches [13, 20, 64] allow graph DBMSs without native temporal support to store time-evolving graphs, they introduce significant storage and runtime overheads. Snapshot-based approaches [27, 30, 34, 35, 45] use snapshots and delta logs to accelerate global queries. Chronos [27] is an offline temporal framework [9] that stores graph entities from different snapshots together to increase cache locality but requires an expensive preparation step for snapshot retrieval [31]. DeltaGraph [34] proposes a hierarchical index for storing multiple snapshots efficiently, and Clock-G [45] reduces the storage footprint with the δ-Copy+Log technique. Tegra [31] stores graph history based on persistent ARTs [39] and allows sharing arbitrary computations across snapshots for fast ad-hoc window analytics. These snapshot-based systems, however, are not designed for local pattern-matching queries and require full graph reconstruction. Systems that use fine-grained storage [28, 68, 76] face similar challenges with model-based approaches for global query execution. AION provides a hybrid storage solution that works well for a broader range of workloads.

Other optimizations for evolving graphs include: (i) computation reordering to share results and communication across snapshots [73, 77]; and (ii) sharing results between queries [1]. These techniques are orthogonal to our work, which uses incremental execution for temporal analytics. TeGraph [29] describes temporal paths as a topological-optimum problem, which is solved using a single scan model and an efficient time-aware format.

8 CONCLUSION

In this work, we formalize time-evolving graphs based on the LPG model to enable (bi-)temporal analytics. Based on this formalization, we developed AION to achieve efficient analytics irrespective of workload characteristics. AION exposes a simple API to query the two dimensions of temporal graphs (time and graph size) and achieves efficient execution for both using a hybrid storage approach. In addition, it introduces efficient graph data structures that enable fast incremental graph computations. Consequently, AION achieves comparable or better performance to existing state-of-the-art approaches and improves temporal analytics computations by 3.5-61.5× over the Neo4j graph DBMS.

REFERENCES

[1] Mahbod Afarin, Chao Gao, Shafiur Rahman, Nael Abu-Ghazaleh, and Rajiv Gupta. 2023. CommonGraph: Graph Analytics on Evolving Data. In *ASPLOS*.

[2] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. 2013. Millwheel: Fault-tolerant stream processing at internet scale. *Proc. VLDB Endow.* (2013).

[3] Tyler Akidau, Paul Barbier, Istvan Cseri, Fabian Hueske, Tyler Jones, Sasha Lionheart, Daniel Mills, Dzmitry Pauliukevich, Lukas Probst, Niklas Semmler, et al. 2023. What's the Difference? Incremental Processing with Change Queries in Snowflake. *Proc. ACM Manag. Data* (2023).

[4] Amazon. 2023. Amazon Elastic Block Store. https://aws.amazon.com/ebs/. Last access: February 26, 2024.

[5] Amazon Neptune. 2023. https://aws.amazon.com/neptune/. Last access: February 26, 2024.

[6] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. 2007. Dbpedia: A nucleus for a web of open data. In *Proc. Int. Semant. Web Conf.*

[7] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. 2006. Group formation in large social networks: membership, growth, and evolution. In *SIGKDD*.

[8] Bahman Bahmani, Abdur Chowdhury, and Ashish Goel. 2010. Fast incremental and personalized pagerank. *arXiv* (2010).

[9] Maciej Besta, Marc Fischer, Vasiliki Kalavri, Michael Kapralov, and Torsten Hoefler. 2021. Practice of streaming processing of dynamic graphs: Concepts, models, and systems. *TPDS* (2021).

[10] Jaewook Byun, Sungpil Woo, and Daeyoung Kim. 2020. ChronoGraph: Enabling Temporal Graph Traversals for Efficient Information Diffusion Analysis over Time. *TKDE* (2020).

[11] Zhuhua Cai, Dionysios Logothetis, and Georgos Siganos. 2012. Facilitating real-time graph mining. In *CouldDB*.

[12] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink™: Stream and Batch Processing in a Single Engine. *Data Eng. Bull.* (2015).

[13] Ciro Cattuto, Marco Quaggiotto, André Panisson, and Alex Averbuch. 2013. Time-varying social networks in a graph database: a Neo4j use case. In *GRADES*.

[14] Samy Chambi, Daniel Lemire, Owen Kaser, and Robert Godin. 2016. Better bitmap performance with roaring bitmaps. *Software: practice and experience* (2016).

[15] Audrey Cheng, Jack Waudby, Hugo Firth, Natacha Crooks, and Ion Stoica. 2024. Mammoths Are Slow: The Overlooked Transactions of Graph Data. *Proc. VLDB Endow.* (2024).

[16] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. 2012. Kineograph: taking the pulse of a fast-changing and connected world. In *EuroSys*.

[17] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. 2015. One trillion edges: Graph processing at facebook-scale. *Proc. VLDB Endow.* (2015).

[18] Eclipse Collections. 2023. https://github.com/eclipse/eclipse-collections. Last access: February 26, 2024.

[19] Dean De Leo and Peter Boncz. 2021. Teseo and the analysis of structural dynamic graphs. *Proc. VLDB Endow.* (2021).

[20] Ariel Debrouvier, Eliseo Parodi, Matías Perazzo, Valeria Soliani, and Alejandro Vaisman. 2021. A model and query language for temporal graph databases. *VLDB Journal* (2021).

[21] David Ediger, Rob McColl, Jason Riedy, and David A Bader. 2012. Stinger: High performance data structure for streaming graphs. In *HPEC*.

[22] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An evolving query language for property graphs. In *SIGMOD*.

[23] Per Fuchs, Domagoj Margan, and Jana Giceva. 2022. Sortledton: a universal, transactional graph data structure. *Proc. VLDB Endow.* (2022).

[24] Betsy George and Shashi Shekhar. 2008. Time-aggregated graphs for modeling spatio-temporal networks. *Journal on Data Semantics XI* (2008).

[25] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. {PowerGraph}: Distributed {Graph-Parallel} computation on natural graphs. In *OSDI*.

[26] Hassan Halawa and Matei Ripeanu. 2021. Position paper: bitemporal dynamic graph analytics. In *GRADES*.

[27] Wentao Han, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Wenguang Chen, and Enhong Chen. 2014. Chronos: a graph engine for temporal graph analysis. In *EuroSys*.

[28] Jiamin Hou, Zhouyu Wang, Zhanhao Zhao, Wei Lu, and Xiaoyong Du. 2023. An Efficient Built-in Temporal Support in MVCC-based Graph Databases.

[29] Chengying Huan, Hang Liu, Mengxing Liu, Yongchao Liu, Changhua He, Kang Chen, Jinlei Jiang, Yongwei Wu, and Shuaiwen Leon Song. 2022. TeGraph: A Novel General-Purpose Temporal Graph Computing Engine. In *ICDE*.

[30] Anand Padmanabha Iyer, Li Erran Li, Tathagata Das, and Ion Stoica. 2016. Time-evolving graph processing at scale. In *GRADES*.

[31] Anand Padmanabha Iyer, Qifan Pu, Kishan Patel, Joseph E Gonzalez, and Ion Stoica. 2021. TEGRA: Efficient Ad-Hoc Analytics on Evolving Graphs. In *NSDI*.

[32] javaCC. 2023. https://github.com/javacc/javacc. Last access: February 26, 2024.

[33] Christian S Jensen, James Clifford, Ramez Elmasri, Shashi K Gadia, Pat Hayes, Sushil Jajodia, Curtis Dyreson, Fabio Grandi, Wolfgang Käfer, Nick Kline, et al. 1994. A consensus glossary of temporal database concepts. *Sigmod Record* (1994).

[34] Udayan Khurana and Amol Deshpande. 2013. Efficient snapshot retrieval over historical graph data. In *ICDE*.

[35] Georgia Koloniari, Dimitris Souravlias, and Evaggelia Pitoura. 2013. On graph deltas for historical queries. *arXiv* (2013).

[36] Krishna Kulkarni and Jan-Eike Michels. 2012. Temporal features in SQL: 2011. *Sigmod Record* (2012).

[37] Pradeep Kumar and H Howie Huang. 2020. Graphone: A data store for real-time analytics on evolving graphs. *TOS* (2020).

[38] Srijan Kumar, Xikun Zhang, and Jure Leskovec. 2019. Predicting dynamic embedding trajectory in temporal interaction networks. In *SIGKDD*.

[39] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *ICDE*.

[40] Jure Leskovec, Daniel Huttenlocher, and Jon Kleinberg. 2010. Governance in social media: A case study of the Wikipedia promotion process. In *ICWSM*.

[41] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. 2012. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *Proc. VLDB Endow.* (2012).

[42] Peter Macko, Virendra J Marathe, Daniel W Margo, and Margo I Seltzer. 2015. Llama: Efficient graph analytics using large multiversioned arrays. In *ICDE*.

[43] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *SIGMOD*.

[44] Mugilan Mariappan and Keval Vora. 2019. Graphbolt: Dependency-driven synchronous processing of streaming graphs. In *EuroSys*.

[45] Maria Massri, Zoltan Miklos, Philippe Raipin, and Pierre Meye. 2022. Clock-G: A temporal graph management system with space-efficient storage technique. In *ICDE*.

[46] MemGraph. 2023. https://memgraph.com/. Last access: February 26, 2024.

[47] Memgraph. 2023. Triggers. https://memgraph.com/docs/fundamentals/triggers. Last access: February 26, 2024.

[48] Alan Mislove, Massimiliano Marcon, Krishna P Gummadi, Peter Druschel, and Bobby Bhattacharjee. 2007. Measurement and analysis of online social networks. In *SIGCOMM*.

[49] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *SOSP*.

[50] Neo4j. 2010. Neo4j Graph Data Platform. https://neo4j.com/. Last access: February 26, 2024.

[51] Neo4j. 2020. Best Practices to Make (Very) Large Updates in Neo4j. https://neo4j.com/blog/nodes-2019-best-practices-to-make-large-updates-in-neo4j/. Last access: February 26, 2024.

[52] Neo4j. 2021. Neo4j Breaks Scale Barrier with Trillion+ Relationship Graph. https://neo4j.com/press-releases/neo4j-scales-trillion-plus-relationship-graph/. Last access: February 26, 2024.

[53] Neo4j. 2022. Indexes for search performance. https://neo4j.com/docs/cypher-manual/current/indexes-for-search-performance/. Last access: February 26, 2024.

[54] Neo4j. 2022. Understanding Neo4j's data on disk. https://neo4j.com/developer/kb/understanding-data-on-disk/. Last access: February 26, 2024.

[55] Neo4j. 2023. https://neo4j.com/docs/bolt/current/bolt/. Last access: February 26, 2024.

[56] Neo4j. 2023. Backup and restore. https://neo4j.com/docs/operations-manual/current/backup-restore/. Last access: February 26, 2024.

[57] Neo4j. 2023. Neo4j Graph Data Science. https://github.com/neo4j/graph-data-science. Last access: February 26, 2024.

[58] Oracle. 2023. Implementing Temporal Validity. https://www.oracle.com/webfolder/technetwork/tutorials/obe/db/12c/r1/ilm/temporal/temporal.html. Last access: February 26, 2024.

[59] Pometry. 2023. Raphtory. https://github.com/Pometry/Raphtory. Last access: February 26, 2024.

[60] Gartner Research. 2023. Market Guide for Graph Database Management Systems. https://www.gartner.com/en/documents/4018220. Last access: February 26, 2024.

[61] Ian Robinson, Jim Webber, and Emil Eifrem. 2015. *Graph databases: new opportunities for connected data.*

[62] Christopher Rost, Kevin Gomez, Matthias Täschner, Philip Fritzsche, Lucas Schons, Lukas Christ, Timo Adameit, Martin Junghanns, and Erhard Rahm. 2022. Distributed temporal graph analytics with GRADOOP. *VLDB journal* (2022).

[63] Betty Salzberg and Vassilis J Tsotras. 1999. Comparison of access methods for time-evolving data. *CSUR* (1999).

[64] Konstantinos Semertzidis and Evaggelia Pitoura. 2016. Time Traveling in Graphs using a Graph Database.. In *EDBT/ICDT Workshops*.

[65] Sijie Shen, Zihang Yao, Lin Shi, Lei Wang, Longbin Lai, Qian Tao, Li Su, Rong Chen, Wenyuan Yu, Haibo Chen, et al. 2023. Bridging the Gap between Relational {OLTP} and Graph-based {OLAP}. In *ATC*.

[66] SQL Server. 2023. Why temporal? https://learn.microsoft.com/en-us/sql/relational-databases/tables/temporal-tables?view=sql-server-ver16#why-temporal. Last access: February 26, 2024.

[67] Benjamin Steer, Naomi Arnold, Cheick Tidiane Ba, Renaud Lambiotte, Haaroon Yousaf, Lucas Jeub, Fabian Murariu, Shivam Kapoor, Pedro Rico, Rachel Chan, et al. 2023. Raphtory: The temporal graph engine for Rust and Python. *arXiv* (2023).

[68] Benjamin Steer, Felix Cuadrado, and Richard Clegg. 2020. Raphtory: Streaming analysis of distributed temporal graphs. *Future Generation Computer Systems* (2020).

[69] Lubos Takac and Michal Zabovsky. 2012. Data analysis in public social networks. In *International scientific conference and international workshop present day trends of innovations*.

[70] Kanat Tangwongsan, Martin Hirzel, and Scott Schneider. 2021. In-order sliding-window aggregation in worst-case constant time. *The VLDB Journal* (2021).

[71] Kanat Tangwongsan, Martin Hirzel, and Scott Schneider. 2023. Out-of-Order Sliding-Window Aggregation with Efficient Bulk Evictions and Insertions. *Proc. VLDB Endow.* (2023).

[72] Daniel ten Wolde, Tavneet Singh, Gábor Szárnyas, and Peter Boncz. 2023. DuckPGQ: Efficient property graph queries in an analytical RDBMS. In *CIDR*.

[73] Manuel Then, Timo Kersten, Stephan Günnemann, Alfons Kemper, and Thomas Neumann. 2017. Automatic algorithm transformation for efficient multi-snapshot analytics on temporal graphs. *Proc. VLDB Endow.* (2017).

[74] Georgios Theodorakis, Alexandros Koliousis, Peter Pietzuch, and Holger Pirk. 2020. Lightsaber: Efficient window aggregation on multi-core processors. In *SIGMOD*.

[75] TigerGraph. 2023. https://www.tigergraph.com/. Last access: February 26, 2024.

[76] Warut D Vijitbenjaronk, Jinho Lee, Toyotaro Suzumura, and Gabriel Tanase. 2017. Scalable time-versioning support for property graph databases. In *Big Data*.

[77] Keval Vora, Rajiv Gupta, and Guoqing Xu. 2016. Synergistic analysis of evolving graphs. *TACO* (2016).

[78] Keval Vora, Rajiv Gupta, and Guoqing Xu. 2017. Kickstarter: Fast and accurate computations on streaming graphs via trimmed approximations. In *ASPLOS*.

[79] Huanhuan Wu, James Cheng, Silu Huang, Yiping Ke, Yi Lu, and Yanyan Xu. 2014. Path problems in temporal graphs. *Proc. VLDB Endow.* (2014).

[80] Yingjun Wu, Wentian Guo, Chee-Yong Chan, and Kian-Lee Tan. 2017. Fast failure recovery for main-memory dbmss on multicores. In *SIGMOD*.

[81] Reynold S Xin, Joseph E Gonzalez, Michael J Franklin, and Ion Stoica. 2013. Graphx: A resilient distributed graph system on spark. In *GRADES*.

[82] Jaewon Yang and Jure Leskovec. 2012. Defining and evaluating network communities based on ground-truth. In *ICDM*.

[83] Xiaowei Zhu, Guanyu Feng, Marco Serafini, Xiaosong Ma, Jiping Yu, Lei Xie, Ashraf Aboulnaga, and Wenguang Chen. 2020. LiveGraph: A Transactional Graph Storage System with Purely Sequential Adjacency List Scans. *Proc. VLDB Endow.* (2020).