TVA: A Version-aware Temporal Graph Storage System for Real-time Analytics

Wenhao Li
Renmin University of China
ruclwh@ruc.edu.cn

Zhanhao Zhao
Renmin University of China
zhanhaozhao@ruc.edu.cn

Jinhao Dong
Renmin University of China
imonedjh@gmail.com

Jiamin Hou
Zhejiang Univ. & RUC
jiaminhou@zju.edu.cn

Wei Lu
Renmin University of China
lu-wei@ruc.edu.cn

Yunhai Wang
Renmin University of China
cloudseawang@gmail.com

Xiaoyong Du
Renmin University of China
duyong@ruc.edu.cn

ABSTRACT

Analyzing temporal graphs can reveal valuable insights that are typically hidden in static graphs. Unfortunately, existing graph storage systems either lack native temporal support or suffer from high latency when querying temporal graphs. This paper presents TVA, a new temporal graph storage system designed for efficient temporal query processing. First, TVA introduces a specialized multi-version storage architecture that separates version metadata from actual data, i.e., the property values associated with different versions of vertices and edges. This architecture enables efficient version retrieval for a vertex or edge by quickly locating valid version metadata and directly dereferencing it to access the corresponding property values. Second, we design tailored data structures, namely the temporal table and enhanced hopscotch-based hashing, to compactly organize the version metadata of adjacent vertices and edges, thus reducing random I/O for metadata lookups during the neighborhood scan initiated from a vertex. Finally, to further accelerate neighborhood scans over multiple vertices, we propose a version-skipping strategy that reuses temporal information obtained from prior scans, thereby avoiding redundant metadata lookups across scans. Empirical evaluations demonstrate that TVA achieves up to 9.9× lower temporal query latency and 2.2× lower storage overhead compared to state-of-the-art temporal graph storage systems.

PVLDB Reference Format:
Wenhao Li, Zhanhao Zhao, Jinhao Dong, Jiamin Hou, Wei Lu, Yunhai Wang, and Xiaoyong Du. TVA: A Version-aware Temporal Graph Storage System for Real-time Analytics. PVLDB, 19(10): 2536 - 2548, 2026.
doi:10.14778/3828612.3828613

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/Sakuraaa0/TVA.git.

1 INTRODUCTION

A *temporal graph*, also known as a *dynamic or time-varying graph*, represents the evolution of relationships between entities, the entities themselves, or both over time [11, 30]. Unlike static graphs, temporal graphs retain rich temporal information, which is crucial for a wide range of applications, such as fraud detection [3, 4, 34] and social network analysis [18, 21, 28].

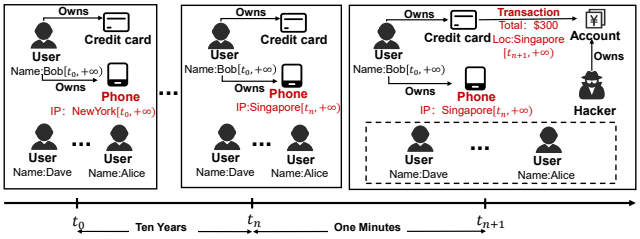

Figure 1: An Example of Financial Temporal Graph.

Example 1. Figure 1 shows a typical temporal graph in the financial domain, where vertices represent entities, such as users, credit cards, and bank accounts, and edges denote their relationships, such as ownership and transaction events. Each vertex and edge can have multiple versions over time. Suppose that user Bob's phone IP address, stable in New York throughout the time interval $[t_0, t_n)$, suddenly changes to Singapore at time t_n. This IP change creates a new version of the vertex representing Bob's phone. Shortly afterward, at time t_{n+1}, Bob initiates a transaction in Singapore. The pattern of a risky transaction following a sudden location change strongly suggests that Bob's account has been compromised. To detect such risks, one can leverage a fraud detection model with relevant graph data [14, 31, 32]. However, if only a static graph (e.g., at time t_{n+1}) is provided, the model may fail to identify the fraud due to the absence of recent geolocation change information. Therefore, it is essential to obtain a temporal graph within an appropriate time period, e.g., $[t_{n-1}, t_{n+1}]$, that includes the necessary temporal context for effective fraud detection. □

Continuing from Example 1, a typical temporal query retrieves transactions that occur between $[t_{n-1}, t_{n+1}]$, along with their corresponding user versions. However, some graph systems [7, 10, 16] retain only the current state of the graph, making them unable to handle such queries. In contrast, systems like T-GQL [6] store versions as separate objects, incurring significant storage overhead. Snapshot-based approaches, such as Clock-G [20], periodically store full snapshots of the graph and maintain logs of deltas between snapshots. It requires expensive reconstruction of snapshots from deltas. A recent system, AeonG [13], applies fine-grained versioning but incurs redundant traversal and extensive random I/O when scanning version chains across large subgraphs. Furthermore,

Wei Lu is the corresponding author.

Proceedings of the VLDB Endowment, Vol. 19, No. 10 ISSN 2150-8097.
doi:10.14778/3828612.3828613

general-purpose storage systems are also ill-suited for temporal graphs: RDBMSs suffer from computationally expensive join operations during multi-hop traversals, while Key-Value (KV) stores lack structural awareness, leading to inefficient temporal range scans.

Addressing these challenges requires a new storage architecture that can efficiently locate relevant versions of a given time condition. Our goal is to minimize the version traversal cost during temporal query processing, thereby improving the overall performance. However, achieving this requires addressing three key challenges: ❶ First, each vertex or edge may contain multiple properties, and updates typically affect only a subset of them. To reduce storage overhead, systems often adopt delta-based encoding [13, 20], where only the modified properties are stored in each version. However, this makes it infeasible to reconstruct a complete version without traversing its version chain. As a result, achieving fast version retrieval with minimal storage redundancy is not straightforward. ❷ Second, a vertex can be connected to multiple edges, each maintaining its own version chain. Existing data structures are primarily designed and optimized for accessing the latest version of data, while historical versions are often appended to each data item using a simple linked structure [6, 13, 20]. This design prevents historical data queries from benefiting from the core optimization mechanisms of the system. Designing a unified structure to co-manage these version chains efficiently, particularly for high-degree vertices, remains a significant challenge. ❸ Third, temporal queries typically involve scanning the neighborhoods of multiple vertices [19, 21]. However, existing systems process each of these neighborhood scans independently. Leveraging this cross-query temporal locality to avoid redundant version traversals and computation is key to enhancing overall query efficiency.

In this paper, we present TVA, a version-aware graph storage system designed for efficient temporal query processing. We introduce a multi-version storage architecture that separates version metadata from actual data. The actual property values of graph objects are stored in a compact structure, with pointers in the metadata, allowing retrieval by first locating the metadata and then dereferencing it. This minimizes random I/O and enables low-latency version retrieval (Challenge ❶). We also design specialized data structures: a temporal table for compact vertex metadata and a hopscotch-based hashing structure for topological relationships. We use a hopscotch-inspired algorithm to co-locate temporally related metadata in the same hash bucket, storing versions in temporal order with a bounded offset. This allows logarithmic lookup time for version metadata, while maintaining access locality by dynamically relocating hot and cold data based on update frequency. SIMD techniques further accelerate version evaluation (Challenge ❷). Finally, we propose a version-skipping approach that links versions within a snapshot, enabling subsequent scans to resume from previously retrieved metadata, avoiding redundant lookups (Challenge ❸).

In summary, we make the following contributions:

- We propose TVA, a new temporal graph storage system designed for efficient temporal query processing.
- We design a multi-version storage architecture tailored for temporal graphs, which reduces redundant version traversal cost without sacrificing storage efficiency.
- We introduce a hopscotch-based hash structure to enable bounded lookup time when performing neighborhood scans under given time conditions. In addition, we utilize SIMD technology to enable parallel query processing.
- We propose a version-skipping approach that reuses temporal information from previous scans to improve temporal query performance.
- We conduct extensive experimental evaluations on widely-used benchmarks, demonstrating that TVA achieves up to 2.2× lower storage overhead and up to 9.9× faster temporal query processing compared to state-of-the-art graph systems.

2 BACKGROUND

In this section, we review existing temporal graph storage systems, discuss the basic techniques used in the design of TVA, and formally define the problem that TVA aims to address.

2.1 Temporal Graph Model

Temporal Graph Definition. A temporal graph is defined as $\mathcal{G} = (\mathcal{X}, \mathcal{E})$:

- Each vertex $x \in \mathcal{X}$ is associated with a set of properties ρ and an active time interval $\tau = [t_{\text{start}}, t_{\text{end}})$.
- Each edge $e \in \mathcal{E}$ is a tuple (src, dst, τ, ρ), where $src, dst \in \mathcal{X}$ are the endpoints of the edge, $\tau = [t_{\text{start}}, t_{\text{end}})$ denotes the edge's active time interval, and ρ is a set of properties.

Each vertex and edge may possess multiple labels. To simplify the subsequent discussion, we assume that each vertex or edge has a single label. For example, we denote a user vertex as x^{user}, a phone vertex as x^{phone}, and a transaction edge as e^{txn}. A one-to-one pair (e.g., user-phone) is denoted as $(x^{\text{user}}, x^{\text{phone}})$. We denote the different versions of a vertex (edge) as $x_n.v_m$ ($e_n.v_m$), where x_n (or e_n) refers to the n-th vertex (or edge), and v_m indicates its m-th version. Throughout this paper, both vertices and edges are collectively referred to as graph objects. For versioning semantics, TVA supports both logical version numbers (e.g., transaction IDs) and physical timestamps. Without loss of generality, we use physical time intervals $[t_{start}, t_{end})$ in our examples.

Temporal Graph Operation. Temporal graph updates evolve the state of the graph while preserving historical information. The supported operations are as follows:

- **Create Vertex or Edge:** Add a graph object o to the temporal graph $\mathcal{G}$ and assign corresponding label and properties, with a time version assigned as $\tau = [t_1, +\infty)$.
- **Delete Vertex or Edge:** Find the graph object $o \in \mathcal{X} \cup \mathcal{E}$ of the corresponding version, update its $\tau = [t_1, +\infty)$ to $[t_1, t_2)$ to mark its deletion.
- **Update Vertex or Edge:** Find the graph object $o \in \mathcal{X} \cup \mathcal{E}$ of the corresponding version, update its $\tau = [t_1, +\infty)$ to $[t_1, t_2)$ and mark it as a historical version. Then, create a new version with $\tau = [t_2, +\infty)$ to represent the latest version and apply the necessary updates to the vertex or edge.

Temporal Graph Query. Temporal graph queries aim to retrieve graph objects that satisfy both a temporal condition and a predicate over their attributes. Given a time point or interval t_q, a temporal query returns all vertices and edges whose validity intervals intersect t_q. The result can be viewed as a static graph snapshot obtained by filtering out objects that are not valid during t_q. Formally, let a

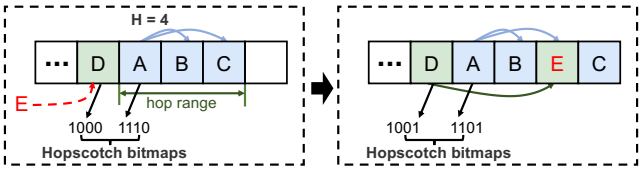

Figure 2: An Example of Hopscotch Hash Table.

graph object $o \in \mathcal{X} \cup \mathcal{E}$ have a unique identifier $o.id$ and a validity interval $o.\tau$. A temporal query is defined as:

$$O_{\text{query}} = \left\{ o \in \mathcal{X} \cup \mathcal{E} \;\middle|\; P(o) = \text{true} \wedge \; o.\tau \cap t_q \neq \emptyset \right\}, \qquad (1)$$

where $P(o)$ is a predicate over graph objects. This formulation generalizes common temporal query types. For example, retrieving a specific object corresponds to $P(o) : o.id = id_q$; and retrieving a set of objects corresponds to a general predicate, e.g., $o.\rho$[Location] = "New York".

2.2 Hopscotch Hash

Hopscotch Hash [12] is an open-addressing hash algorithm that constrains each key to a fixed-size neighborhood ($\mathcal{H}$) near its primary slot. Each slot maintains an $\mathcal{H}$-bit bitmap, indicating which of the next $\mathcal{H}$ slots hold its key. During insertion, if a key's primary slot is occupied, the algorithm identifies the first available empty slot. If this empty slot is outside the desired neighborhood, the algorithm searches the previous $\mathcal{H}$-1 items to find an item that can be swapped into the empty slot without violating its neighborhood constraint. The swap is repeated until an empty slot is moved into the neighborhood of the key's primary slot. Figure 2 shows an example of this process. In the left subfigure, keys A, B, and C share the same hash value and are inserted into slots within the hop range of A's primary slot (Bitmap=1110). Similarly, D is placed in its own primary slot (Bitmap=1000). When inserting E, which has the same hash value as D, its primary slot is full. Through the hopscotch insertion process, E is moved into a slot (Bitmap=1001) that still falls within the hop range of D's primary slot.

Although the original hopscotch insertion algorithm is effective in resolving hash collisions, it fails to preserve data ordering and is not designed for managing temporal data versions. Instead of a direct application, we innovatively extend and redesign the hopping mechanism to propose an enhanced hopscotch-hash algorithm, which is specially tailored for organizing multiple temporal versions of data objects within a unified hash table. Our design addresses the specific challenges of temporal locality and version lookup efficiency, as detailed in § 4.2.

3 SYSTEM OVERVIEW

TVA supports efficient temporal graph management through two key components: the storage engine and the query engine.

3.1 Storage Engine

As shown in Figure 3, TVA's storage engine adopts a hybrid architecture with two specialized components, designed to handle the different types of data in a temporal graph. An incoming update is routed by the `Relationship Classifier` to the Temporal Property Storage if it modifies vertices' intrinsic properties, or to the Dynamic Topology Storage if it creates, deletes, or modifies an edge between vertices. The central idea behind these two components is to separately manage version metadata and the actual data.

Temporal Property Storage: This component is optimized for efficient management of vertex properties (e.g., a user's name or a phone's IP address). At any moment, each vertex maintains at most one current value per property; therefore, a columnar storage layout is employed to facilitate efficient access to current data [1]. Historical values for each property are stored separately in a specialized structure called `Temporal Buffer`. Versioning metadata is maintained by `Temporal Table` structure. Upon property updates, the existing value is not overwritten; instead, it is transferred to the corresponding `Temporal Buffer`, and a new record is added to the `Temporal Table` to index this update. We will provide further details in § 4.1.

Dynamic Topology Storage: This component is responsible for managing the dynamic connections between vertices (e.g., transactions or social links), where a vertex can have multiple edges for the same relationship. Such dynamism makes columnar layouts for current data, where data location can be calculated by a predictable offset [23], unsuitable. We observe that for the edges of the same vertex, there is no strong order correlation among different edges, but for different versions of the same edge, it is best to store them in chronological order to speed up lookup. To achieve this, we construct a storage structure based on a hopscotch algorithm. This structure stores all edges of a vertex within a single hash bucket, and it organizes all versions of an edge chronologically within a "neighborhood". In addition, we optimize dynamic topology storage through a cold-hot data separation strategy. We will introduce the details in § 4.2.

3.2 Query Engine

The query engine processes user queries and retrieves relevant graph data from the hybrid storage. TVA supports all temporal graph operations described in § 2.1.

For Temporal Property Storage, all basic metadata for each vertex is stored in the `Header` of the `Temporal Table`. TVA first identifies the target vertex, then locates its `Header` in the `Temporal Table` based on the offset to check whether the data exists in the `Current Storage`. If present, the data can be read directly from the corresponding column; if not, TVA performs a binary search to locate the correct `Version` in the `Temporal Table`, using the offset recorded in that `Version` to determine the data position. Additionally, when a query retrieves all data from the same timestamp, `NextOffset` structure is introduced to use information from the previous row, thereby accelerating queries for subsequent rows. Further details are provided in § 5.1.

For Dynamic Topology Storage, we similarly start by identifying the relevant vertex. Subsequently, using the `HashFunction` and `LatestValuePtr` in the storage structure pointed to by that vertex, we determine the potential range for the edge we need to find. Finally, we locate the specific data within this range. Further details are provided in § 5.2.

These two structures enable efficient temporal queries on both properties and topology. In addition, these compact storage layouts enable us to efficiently leverage SIMD hardware acceleration,

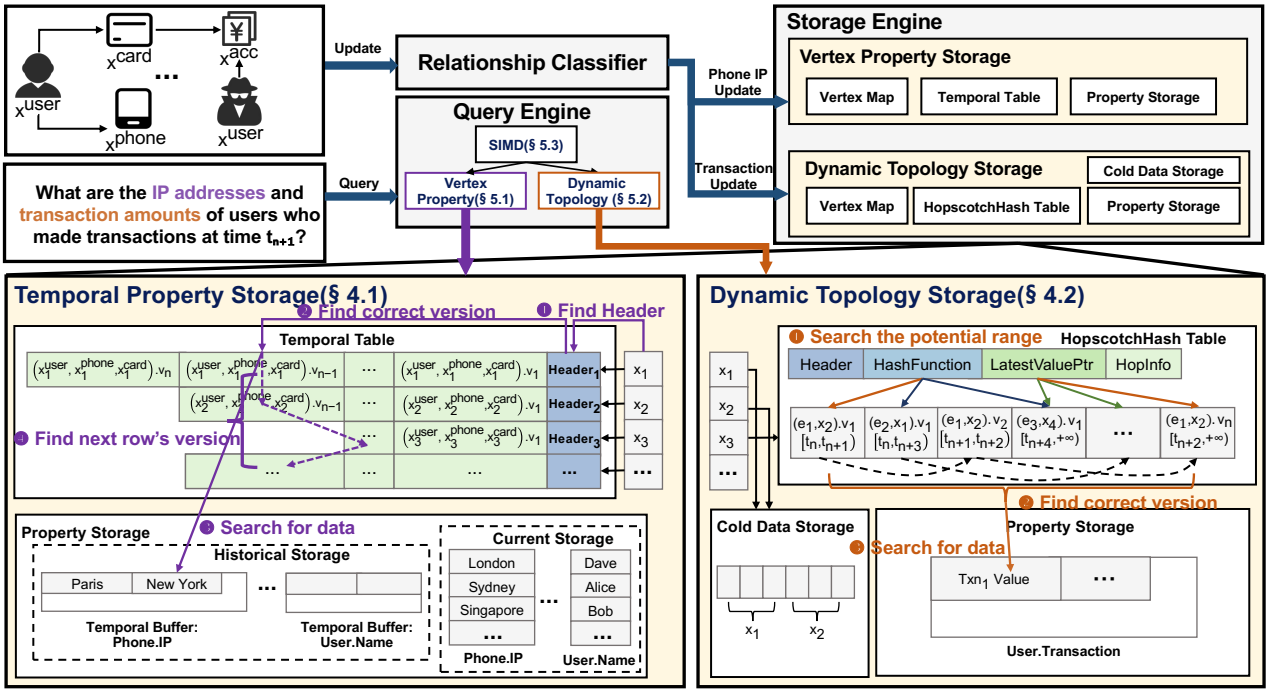

Figure 3: Overall Architecture of TVA.

thereby further speeding up temporal queries. More details are provided in § 5.3.

Example 2. Figure 3 presents example query statements that demonstrate operations on both storages. The query highlighted in purple corresponds to Temporal Property Storage, while the one in orange illustrates an operation on Dynamic Topology Storage.

The purple query corresponds to: "Snapshot of IP addresses at t_{n+1}." It locates the first vertex x_1 and performs a binary search on the Temporal Table to find the matching version. For subsequent queries, the previously identified `Version` can be leveraged to quickly locate potential neighboring position of the next row's `Version`, thereby narrowing the search. The orange query resolves the question, "What are the transaction amounts at time t_{n+1}?" We take Bob's transaction as an example. Let Bob's vertex be x_3, and the relevant edge be e_1. Using the `HashFunction`, we can locate the oldest version of e_1, while `LatestValuePtr` points to its latest version. Searching within this range then retrieves the desired data. If we need to search for the next edge of x_3, we can similarly utilize the information obtained in e_1. □

4 STORAGE DESIGN

In this section, we detail the design of the Temporal Property Storage and dynamic topology storage. We provide detailed pseudocode for Create, Read, Update, and Delete (CRUD) operations to explicitly demonstrate how TVA handles data manipulation.

4.1 Temporal Property Storage

Key Idea. We propose a multi-version storage architecture for vertex properties, based on the observation that each vertex maintains at most one current value for any given property. The current state is stored in a columnar layout, which allows efficient direct access for current queries. To address the inefficiency of traditional methods that require traversing version chains for historical queries, we introduce a dedicated historical storage design. The core of this design is the `Temporal Table`, which separates version metadata from property values and stores all historical data in a unified, compact region. Metadata entries are stored in a tightly packed, contiguous format to enable rapid direct lookups. As a result, retrieving any version requires only locating its metadata and dereferencing the corresponding value, thereby minimizing overhead. In addition, we introduce `NextOffset` structure within the `Temporal Table` that links temporally-related versions across different vertices, accelerating snapshot queries that require data from the same timestamp.

Storage Format. As illustrated in Figure 4, the fundamental layout of `Current Storage` in this component is columnar storage.

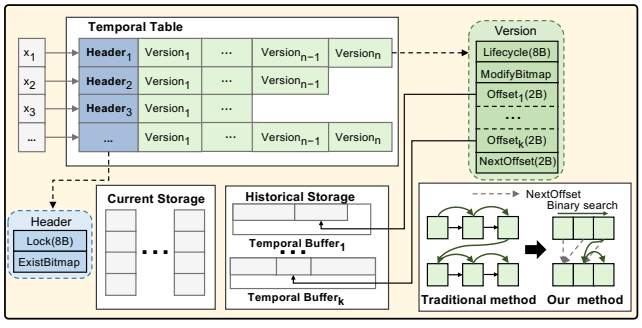

Figure 4: The Storage Format for Vertex Properties.

Algorithm 1: Create and Update Operations in Temporal Property Storage

Input: Vertex vid, PropKey p, Value v, Time t, Temporal Table $\mathcal{T}$, Current Store C, Historical Store $\mathcal{H}$

1 **Function** `Create`($vid,\ p,\ v,\ t$):
2 $\quad C[vid][p] \leftarrow v$
3 $\quad ver \leftarrow$ `NewVersion`$(t,\ \infty)$
4 $\quad ver.Offset[p] \leftarrow$ **null**
5 $\quad \mathcal{T}[vid]$.`Append`(ver)
6 **Function** `Update`($vid,\ p,\ v_{new},\ t$):
7 $\quad v_{\text{old}} \leftarrow C[vid][p]$
8 $\quad C[vid][p] \leftarrow v_{\text{new}}$
9 $\quad offset_{\text{old}} \leftarrow \mathcal{H}$.`Append`$(v_{old})$
10 $\quad ver_{\text{archived}} \leftarrow$ `NewVersion`(t)
11 $\quad ver_{\text{archived}}.Offset[p] \leftarrow offset_{\text{old}}$
12 $\quad \mathcal{T}[vid]$.`Append`$(ver_{archived})$

The vertex IDs allocated according to the chosen scheme serve as primary keys. Properties are organized into distinct columns and indexed by their labels and property keys. To manage temporal data and updates effectively, we introduce two critical extensions:

`Temporal Table`: For each vertex ID, `Temporal Table` manages version metadata and stores it separately from the corresponding property values. Each row consists of a `Header` and several `Version` entries. The `Header` contains a `LOCK` field for concurrency control during updates, as well as an `ExistBitmap`, which indicates which property columns currently have data for the corresponding vertex. Each `Version` entry represents an update to the vertex and records the historical versions of the relevant properties. Specifically, it includes a `Lifecycle` field that specifies the validity interval $[t_{\text{start}}, t_{\text{end}})$ of that historical state, and a `ModifyBitmap` that identifies which property columns are involved in this version. Additionally, an array of $\texttt{Offset}_1$ to $\texttt{Offset}_k$ is included, where k denotes the number of property columns. Each $\texttt{Offset}_j$ points to the storage location of the historical value for the j-th property within the `Temporal Buffer`; if it is set to `nullptr`, it instead refers to the value in the `Current Storage`. Finally, the `Version` structure contains a `NextOffset` pointer, which links to a specific `Version` in the next row. When a new `Version` is created, this pointer is set to the latest `Version` in the following row at that time. These `NextOffset` pointers collectively form multiple version chains, through which we can efficiently retrieve subsequent data points within the same temporal context.

`Historical Storage`: This is a large, append-only region where the actual historical values are stored. Each property has its corresponding `Temporal buffer`. When a property is updated, its previous value is written contiguously into its buffer.

Insert, Update or Delete. We design the system to maintain current data efficiently while archiving historical versions without costly reorganizations. The core idea is to physically separate current and historical states. In our system, a delete operation is treated as a special type of update operation and how read operations are performed will be elaborated in Section 5. To explicitly demonstrate how TVA handles data manipulation, we present the pseudocode for Graph operations in Algorithm 1.

For **Create** operations (lines 1-5), TVA directly writes the value to the `Current Storage` to ensure $O(1)$ access for latest-state queries, while initializing the version metadata in the `Temporal Table`.

The **Update** process (lines 6-12) is the core of our version-aware design. When vertex properties are updated at time t, we proceed as follows: First, to prioritize the performance of reading the latest graph state, the new property value (v_{new}) is written directly into the corresponding property column of the `Current Storage` (line 8), implicitly marking its validity as $[t, +\infty)$. Simultaneously, the previous property value (v_{old}) is preserved: it is appended to the append-only `Historical Storage` (*Temporal Buffer*), as shown in line 9. Finally, a new `Version` entry is created and appended to the `Temporal Table` to manage the metadata of this state change (lines 10-12). This new entry records the lifecycle end time (t) of the old version (representing the interval $[t_{\text{current}}, t)$) and stores the offset pointing to the archived v_{old}. Additionally, as detailed in Section 4.1, this new version entry maintains a `NextOffset` pointer, linking it to the latest version of the subsequent vertex to accelerate cross-vertex snapshot scanning.

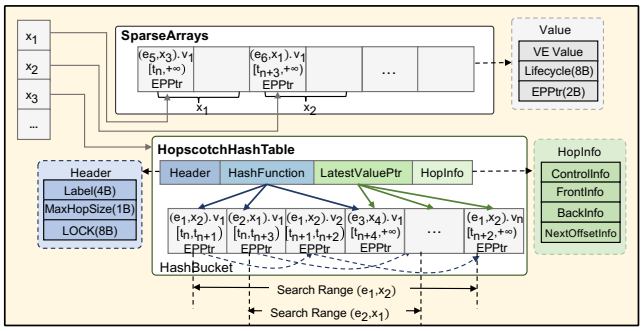

Figure 5: The Storage Format for Dynamic Topologies.

4.2 Dynamic Topology Storage

Key Idea. For dynamic topology and its associated edge properties, it is difficult to use a simple vertex-to-storage mapping because one vertex can be linked to many data items of the same relationship. Additionally, the uncertainty in the number of edges associated with each vertex makes it difficult to predict the storage requirements needed for compact storage. This issue is even more obvious in real graphs, where some vertices have many edges and are updated often [35, 37], leading to a large number of old versions. To solve this, we use a mixed storage method: infrequently updated ("cold") data is stored in a `SparseArray`, while frequently updated ("hot") data is managed by an improved `HopscotchHash Table` that is designed for handling time-related data. This structure stores all objects with the same relationship in the same hash bucket. Our proposed "Hopscotch" algorithm guarantees that all historical versions of a given data item are stored in temporal order with a bounded offset.

Storage Format. The overall architecture is shown in Figure 5. The system stores data based on vertex degrees using two structures: `SparseArray` for "cold" data storage and `HopscotchHash Table` for "hot" data storage. We define concrete criteria for this division: A vertex v is classified as "hot" and migrated to the HopscotchHash Table if its degree exceeds a threshold T_{deg}, or its total version count reaches T_{ver}, formally expressed as $\sum_{e \in E(v)} N_v(e) \geq T_{ver}$ (where $N_v(e)$ is the version count of edge e). This migration is triggered

asynchronously to minimize blocking. A background worker allocates a new hash bucket, re-organizes the existing data, and atomically updates the pointer, allowing the current write operation to complete without blocking. Conversely, migration from Hopscotch back to SparseArray is rare; it only occurs during a vacuum/purge operation where data prior to a specific timestamp is cleared, reducing the edge count below the threshold. A `Vertex Map` directs different vertices to their respective storage units. Both structures employ a unified minimal storage unit for edge information that encapsulates three components: `VE Value` (e_i, x_j), which captures the graph's topological information; `Lifecycle`, specifying the validity interval $[t_{\text{start}}, t_{\text{end}})$ of that historical state; and `EPPtr`, which points to the specific storage location of the edge's corresponding property. Next, we will introduce these two specific data structures.

`SparseArray`: For "cold" data storage, it allocates fixed-size data slots for each vertex. This array-based structure offers good spatial locality, facilitating efficient scanning operations. Edges are inserted in a temporally ordered manner within a vertex's allocated space, simplifying subsequent time-series queries for that vertex's edges. When the number of elements reaches its maximum limit, all data will be transferred to the `HopscotchHash Table` for storage.

`HopscotchHash Table`: For "hot" data storage, which are associated with numerous and frequently updated edges, HopscotchHash Table is employed. This structure is designed for efficient insertion and $O(1)$ retrieval of the latest edge data, with historical versions accessible with logarithmic time complexity. The architecture of the HopscotchHash Table integrates several components: `Header`, which maintains metadata such as the label, the `MaxHopDistance` parameter, and `Lock` for concurrency control; `HashFunction` that maps edges to 64-bit hash values, while additionally recording the upper seven bits in the `HopInfo` to enhance SIMD query operations, as detailed in § 5.3. The position indicated by `HashFunction` can be considered as the initial location of the corresponding item. To track the most recent version of each item, a dedicated hash table, `LatestValuePtr`, indexes data item identifiers to the storage location of their latest versions in the `HashBucket`.

Version control and efficient traversal are further enabled by four arrays collectively referred to as `HopInfo`. `ControlInfo` stores the upper seven bits of the hash value for each `HashBucket` slot. `FrontInfo` and `BackInfo` record bidirectional version offsets, respectively tracking links to the chronologically preceding and succeeding versions of a given entry. `MaxHopDistance` guarantees that all versions of a particular edge are stored within a bounded offset. In addition, the `NextOffset` array facilitates fast queries for subsequent data entries sharing the same timestamp.

Insert, Update, or Delete. These graph operations require finding the corresponding data insertion position in the Hopscotch Hash Table. Therefore, the key challenge is to resolve conflicts between temporal records. To address this issue, we propose a novel Hopscotch algorithm for temporal data storage.

Two Hopscotch Ways. When inserting a new version of a data item, we first obtain the most recent position p via `LatestValuePtr`. The new version must be inserted within the range $[p + 1, p +$ `MaxHopDistance`$]$. If no empty slot is found within this range, the system must create a free slot via a "Hopscotch" algorithm, which involves relocating existing data items.

Algorithm 2: Hop Update Process

Input: Free index idx_{free}, Destination index idx_{dst}, FrontInfo $fronts$, BackInfo $backs$, Max hop distance $d_{\text{max\_hop}}$, Judge distance d_{judge}, HashBucket $\mathcal{B}$

Output: An updated HopscotchHash Table with available free slot

```
1  Function UpdateHop(idx_free, idx_dst):
2      idx_now ← idx_free − 1
3      while idx_free ∉ [idx_dst, idx_dst + d_max_hop] do
4          if idx_free − idx_dst ≤ d_judge then
5              if d_max_hop − fronts[idx_now] ≥ idx_free − idx_now then
6                  Swap(idx_now , idx_free)
7                  idx_free ← idx_now
8                  idx_now ← idx_free − 1
9              else
10                 idx_now ← idx_now − 1
11         else
12             idx_initial ← idx_now
13             while fronts[idx_initial] ≠ 0 do
14                 idx_initial ← idx_initial − fronts[idx_initial]
15             if idx_initial ≥ idx_dst then
16                 idx_free ← HopChain(idx_now, idx_free)
17                 idx_now ← idx_free − 1
18             else
19                 idx_now ← idx_now − 1
20 Function Swap(i, j):
21     B[j] ← B[i]
22     backs[i − fronts[i]], fronts[j] ← fronts[i] + j − i
23     fronts[i + backs[i]], backs[j] ← backs[i] − j + i
24 Function HopChain(idx_end, idx_free):
25     idx_right ← idx_free
26     idx_left ← idx_end
27     while fronts[idx_left] ≠ 0 do
28         Swap(idx_left , idx_right)
29         idx_right ← idx_left
30         idx_left ← idx_left − fronts[idx_left]
31     return idx_left
```

In practice, to efficiently perform the operations illustrated in the example, we employ two distinct strategies based on the distance between the target location and the nearest empty slot.

As shown in Algorithm 2, we define a threshold d_{judge} (line 4). When the distance between the empty bucket and the target position is below this threshold, we employ the single-step hopping method (lines 5-10), where each item is checked to see if it can be relocated. However, when the empty bucket is too far from the target position, sequentially checking each item would incur high computational overhead. In this case, we adopt a second method, the chain hopping approach (lines 11-19). This method checks each data item to determine its oldest version location by the `FrontInfo` (lines 12-14), and if the position is after the target location, we can move the entire data item backward (lines 15-17). This action does not violate any hop distance constraints, so no further validation is necessary. The chain hopping operation involves moving each data item to the position of its next item, with the last item being moved to the empty bucket (lines 24-31). This approach quickly

brings the data item closer to the target position, thus reducing the number of comparisons. The hop-bounded layout guarantees that the search scope for a specific version is restricted within a localized neighborhood in the hash bucket, preventing full-chain traversal even when version chains become long. This design preserves logarithmic retrieval time with respect to the number of versions per edge.

Discussion. Although our design is inspired by the Hopscotch hashing algorithm, it differs fundamentally in its objective. The original algorithm focuses on resolving hash collisions by using a bitmap to keep keys within a fixed neighborhood of their home bucket. In contrast, our approach is designed to support ordered storage and efficient retrieval of multiple temporal versions of a single data item. To this end, we replace the bitmap with item-centric `FrontInfo` and `BackInfo`, which explicitly capture inter-version relationships. This design enables two novel mechanisms, *single-step hopping* and *chain hopping*, where the latter leverages these inter-version links and is unique to our structure.

5 TEMPORAL QUERY ENGINE

In this section, we present the query processes and time complexity analysis for Temporal Property Storage and Dynamic Topology Storage, and introduce the use of SIMD technology.

5.1 Query Engine for Vertex Properties

To access the latest property version, the system directly reads from the relevant column using the vertex ID offset, avoiding version management overhead. For temporal queries at time t, it consults the vertex's `Temporal Table` and uses each `Version`'s `ModifyBitmap` and `Lifecycle` to locate the appropriate version entry via binary search. When multiple properties are queried simultaneously, parallel lookups are employed after the corresponding versions are identified to further accelerate retrieval. The actual data value's location is then determined by the `Offset`: if null, the current value is used; otherwise, the value is retrieved from the `Temporal Buffer`. Further, to optimize the requirement of finding all objects at the same time point, the system finds the first object as described above. Then, for all subsequent objects, it uses the `NextOffset` pointer from the previously queried `Version` to quickly locate their relevant `Version`, reducing comparison overhead and avoiding repeated full searches.

Theorem 1. *Let the* `NextOffset` *of* $\texttt{Version}_i$ *point to the next row's* $\texttt{Version}_j$*, with the lifecycles of* $\texttt{Version}_i$ *and* $\texttt{Version}_j$ *given by* $[ti_{start}, ti_{end})$ *and* $[tj_{start}, tj_{end})$*, respectively. Then, the lifecycles satisfy:*

$$[ti_{start}, ti_{end}) \cap [tj_{start}, tj_{end}) \neq \emptyset \wedge ti_{start} > tj_{start} \quad (2)$$

Please see the proofs in our github [27]. This theorem allows us to start the version search for the next object from a temporally relevant position, effectively narrowing down the search space.

I/O Analysis. Here, we denote the number of vertices as N, with an average version count per vertex represented by N_v.

Read the latest version: Accessing the most current data is highly efficient in our system. It involves a direct lookup in the primary property columns, resulting in approximately one random access to fetch the data for the given object's property.

Table 1: Time Complexity of Different Operations in TVA.

Operation	HopscotchHash Table		Temporal Table
	Situation	Time Complexity	Time Complexity
Read the latest version	$N_v \cdot N_e \leq S$	$O(S)$	$O(1)$
	$N_v \cdot N_e > S$	$O(1)$	
Read a specific historical Version	$N_v \cdot N_e \leq S$	$O(S)$	$O(\log(N_v))$
	$N_v \cdot N_e > S$	$O(\log(N_v))$	
Read specific historical versions for all objects	$N_v \cdot N_e \leq S$	$O(S)$	$O(N)$
	$N_v \cdot N_e > S$	$O(N \cdot N_e)$	
Insert, Update or Delete	$N_v \cdot N_e \leq S$	$O(1)$	$O(1)$
	$N_v \cdot N_e > S$	$O(1)$ ... *average* $\Omega(N_v \cdot N_e)$ ... *worst*	

Read a specific historical version: To retrieve a historical version, TVA first performs a random access to locate the entries in the `Temporal Table`, then uses binary search to find the correct `Version`. It finally retrieves the actual data from the `Temporal Buffer` using the `Offset`. This approach achieves a time complexity of $O(\log N_v)$, compared to $O(N_v)$ for version chain methods. Moreover, since `Version` uses a contiguous array, TVA only incurs random I/O at the final step, whereas version chain approaches require random I/O at each search step.

Read specific historical versions for all objects: When querying a full graph snapshot at time t, locating the first vertex's version is similar to a single-point query. For subsequent vertices, TVA uses the `NextOffset` pointer to jump directly to the relevant region for each vertex, avoiding a full search. Finding the exact version within this region typically requires a small number k of probes, leveraging temporal locality. Thus, after the initial search, the complexity for the remaining $(N-1)$ vertices is approximately $O((N-1)k)$, where k is a small constant, resulting in an overall complexity of $O(N)$. In contrast, systems without this optimization may require $O(N \cdot N_v)$ time for N independent full searches. Therefore, `NextOffset` substantially reduces the complexity of snapshot queries.

5.2 Query Engine for Dynamic Topology

For cold data, since each vertex data is stored in a fixed, compact array, traversing the array can quickly locate the required version. For hot data in the `HopscotchHash Table`, the query process is as follows: to access the latest version of an edge, the system uses `LatestValuePtr` to directly locate the storage position by destination vertex ID, achieving $O(1)$ time complexity. For historical data at a specific time t, the system combines `HashFunction` and `LatestValuePtr` to determine the bounds of available versions, with storage distances capped by `MaxHopDistance`. Leveraging SIMD instructions, up to 16 slots can be compared simultaneously; when $\texttt{MaxHopDistance} \leq 16$, these versioned edges can be regarded as tightly stored, as validated in experiment 7.5.1. How we leverage SIMD operations to accelerate query processing will be detailed in § 5.3. Overall, this enables fast, binary-search-style queries in a compact region, reducing time complexity from linear to logarithmic compared to traditional chained traversals. Additionally, for querying all objects at the same time point, `NextOffsetInfo` can be used to accelerate subsequent lookups, following a process similar to that described in § 5.1.

I/O Analysis. Here, we denote the number of vertices as N. Each vertex in the SparseArray has a size of S. Each hash bucket can contain a maximum of M items, and we represent the vertex degree

as N_e, `MaxHopDistance` as H, and the average number of versions per edge as N_v.

Read the latest version: if $N_v \cdot N_e \leq S$, the SparseArray contains fewer data items. A bounded scan of this small array, with complexity $O(S)$, is extremely efficient due to its excellent data locality. When $N_v \cdot N_e > S$, the data is stored in HopscotchHash Table, and the position of the most recent data item can be directly found using the `LatestValuePtr`, so the time complexity is $O(1)$.

Read a specific historical version: SparseArray requires $O(S)$, while HopscotchHash Table uses binary search with $O(\log N_v)$ due to sorted data and pointers.

Read historical versions for all objects: After identifying the first edge, subsequent versions can be found using `NextOffsetInfo`, resulting in $O(N \cdot N_e)$ complexity.

Insert, Update or Delete: In SparseArray, insertion is $O(1)$ due to direct access. In HopscotchHash Table, insertion is $O(1)$ on average, but may require $O(N_v \cdot N_e)$ in the worst case if resizing occurs. However, the hop failure probability is low, approximately $\frac{1}{M \cdot H!}$, reducing this occurrence significantly.

Theorem 2. *The probability of hop failure in HopscotchHash Table is* $\frac{1}{M \cdot H!}$

Please see the proofs in our github [27]. Finally, Table 1 summarizes the I/O characteristics of TVA, focusing on its core components: the versioned ColumnStore with Temporal Table and the Hopscotch-optimized Hash Table.

5.3 SIMD Acceleration

To further improve query performance, TVA leverages Single Instruction, Multiple Data (SIMD) techniques. Next, we elaborate in detail how we utilize this technique to get an edge in our Hopscotch-Hash Table. As discussed in § 4.2, the control array `ControlInfo` records the upper seven bits of the hash value along with a control bit, forming an 8-bit control field. Specifically, values from 0x00 to 0x7F indicate a valid entry and encode the upper seven hash bits; 0x80 denotes an empty slot; and 0xFF marks a deleted entry. This design allows for an initial assessment of whether a candidate slot may contain the target data item by simply comparing the 8-bit control field during query operations. Since each control field occupies only 1 byte, its small size allows it to be efficiently loaded into the cache, which reduces memory access latency.

We implement a SIMD-optimized binary search process within the `GetTemporalEdge` function, as shown in Algorithm 3. Instead of directly comparing full data items, it first calls the `Match` function (line 6) to rapidly filter a batch of candidates by matching their Control Fields. The `Match` function is where SIMD is applied (lines 21-26). It first loads 16 Control Fields into a SIMD register (line 22). Then, another instruction `_mm_cmpeq_epi8` performs a parallel comparison of these 16 bytes against the target hash's upper bits (line 24). The `_mm_movemask_epi8` instruction efficiently converts the 16-byte comparison result into a 16-bit bitmap, identifying all matching candidates at once (line 25). After that, we compare the data items matched by the `Match` function (lines 8-17).

The application of SIMD technology in Temporal Property Storage lies in `Version` lookup, with the specific operational procedure being largely consistent with the above description. This approach allows TVA to perform an initial screening on multiple data items using just a few CPU instructions. It reduces the latency associated with traditional item-by-item comparisons and accelerates the location and retrieval of historical versions.

Algorithm 3: Get a Specific Temporal Edge

```
1  Input: Search range start index idx_start, End index idx_end, Target
       edge E, ControlInfo controls, HashBucket B
2  Output: The target edge's index
3  Function GetTemporalEdge(idx_start, idx_end, E):
4      while idx_start ≤ idx_end do
5          idx_mid ← ⌊(idx_start + idx_end)/2⌋
6          res_match ← Match(idx_mid, E.control)
7          upper_bound_updated ← false
8          while res_match ≠ 0 do
9              idx = __builtin_ctzll(res_match) + idx_mid
10             res_match &= res_match − 1
11             if B[idx].dst = E.dst then
12                 if B[idx].τ ∩ E.τ ≠ ∅ then
13                     return idx
14                 else if B[idx].τ.t_start > E.τ.t_end then
15                     idx_end ← idx − 1
16                     upper_bound_updated ← true
17                     break
18         if not upper_bound_updated then
19             idx_start ← idx_mid + 16
20     return IDX_NOT_FOUND
21 Function Match(i, c):
22     v ← _mm_loadu_si128(controls + i)
23     w ← _mm_set1_epi8(c)
24     x ← _mm_cmpeq_epi8(v, w)
25     mask_match ← _mm_movemask_epi8(x)
26     return mask_match
```

6 IMPLEMENTATION

TVA is implemented in C++ with ~7,000 lines of code. TVA realizes the hybrid architecture described above: it combines versioned columnar storage with an optimized Hopscotch hash table to enable efficient storage and querying of temporal graphs. TVA supports both property and non-property graph input formats. When the input is a non-property graph, TVA only utilizes Dynamic Topology Storage for the entire graph. TVA implements all temporal graph data management operations defined in § 2.1, providing users with rich interfaces to perform complex queries and updates directly on temporal graphs.

TVA supports Snapshot Isolation (SI). To ensure data consistency in concurrent environments, TVA uses fine-grained locking at the `Header` of each object's `Temporal Table` and `HopscotchHash Table`, enabling thread-safe parallel updates and queries. This locking granularity avoids the performance bottlenecks caused by global locks, improves data access efficiency under multi-threaded workloads, and guarantees both atomicity and consistency of operations.

Fault Tolerance and Persistence. To ensure data durability, TVA implements a Write-Ahead Logging (WAL) mechanism. Each WAL entry is structured as a compact binary sequence starting with a header that contains a 4-byte magic number, version identifier,

operation type and flags. This header is followed by an 8-byte timestamp, payload length, the variable-length payload, and finally a CRC32 checksum to guarantee data integrity during recovery. To address diverse storage constraints, TVA introduces a flexible persistence architecture managed via a `PersistManager` that supports two distinct modes. The *Pointer Mode*, designed for high-frequency access, retains TVA's core in-memory structures (i.e., `ColumnTable`) to preserve traversal speed, but replaces bulky property values with lightweight 64-bit `DiskOffset` pointers referencing an append-only disk file. For scenarios requiring full persistence, the *Full Mode* leverages RocksDB as the backend. We design a specialized time-encoded key schema, specifically encoding keys as `user:{uid}:ts:{timestamp}` for vertices and `friend:{src}:{dst}:ts:{timestamp}` for edges, to exploit the lexicographical ordering of the underlying LSM-tree. This ensures that historical versions are physically adjacent on disk, enabling efficient point-in-time queries via iterator seeking and temporal range scans without loading the entire history into memory.

7 EVALUATION

In this section, we evaluate TVA through comparisons with state-of-the-art graph systems. We conduct system-level and micro benchmarks to measure the query performance and storage consumption.

7.1 Experiment Setup

We conduct our experiments on a machine equipped with an Intel(R) Xeon(R) Gold 5220 CPU @ 2.20GHz and 128 GB of memory, running Ubuntu 20.04. All code is compiled using GCC 11.3.0 with the `O3` optimization flag. All systems utilize 5 threads by default.

Competitors. We compare TVA with two categories of graph storage systems. 1) For temporal data operations, we compare with the most advanced graph data structures that support temporal functionality: Clock-G [20], T-GQL [6], and AeonG [13]. We also included PostgreSQL [22] (representing temporal-extended RDBMS) and RocksDB [9] (representing Key-Value stores) in our experiments. Note that both Clock-G and T-GQL store all vertices and edges in memory. To ensure a fair comparison, we cache all data in memory for all the evaluations. 2) For recent data operations, we select state-of-the-art dynamic graph storage systems as baselines for comparison: GraphOne [16], Stinger [7], and Sortledton [10]. Since these structures do not support temporal operation. To ensure a fair comparison for MVCC graph stores, we standardize the experimental setup by maintaining complete version histories across all compared systems to rigorously evaluate their temporal query capabilities without the interference of garbage collection.

Graph Datasets. We employ a variety of graph datasets for evaluation, including: IMDB [24], an actor collaboration network from IMDB; DBLP [33], a collaboration network based on DBLP; YouTube [33], a social network from YouTube; Epinions [25], the Epinions "who-trust-whom" social network; Pokec [26], Slovakia's most popular social network; and LDBC SNB [8], the LDBC Social Network Benchmark designed to simulate real-world social networks.

Graph Benchmark. For temporal queries, we utilize three temporal workloads: T-mgBench, T-LDBC, and T-gMark, following [13]. T-mgBench is based on the Pokec dataset and follows the Memgraph mgBench workload. Specifically, the "`FOR TT AS OF` t" clause is appended to queries Q1 and Q3 to generate "time point" queries, while the "`FOR TT FROM` t_1 `TO` t_2" clause is included in Q2 and Q4 to produce temporal range queries. T-LDBC is adapted from the LDBC workload by integrating "`FOR TT AS OF` t" into the IS queries (IS1–IS7). T-gMark is based on gMark [2] and is designed to stress-test property management. It features 24 vertex properties and 82 edge properties, covering diverse query shapes and selectivities.

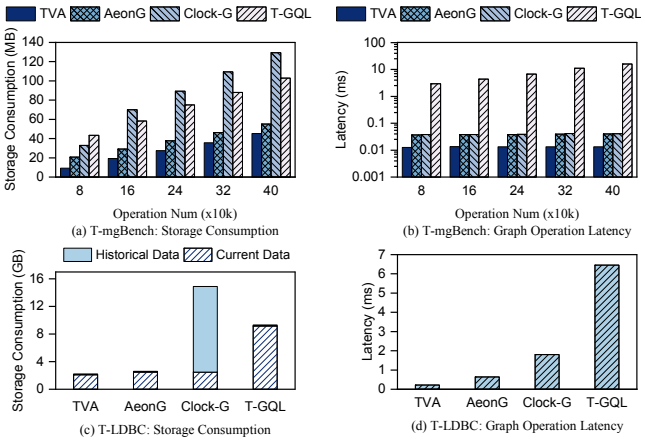

Figure 6: Temporal Graph Storage Consumption and Graph Operation Latency.

For recent data operations, we consider four representative graph analysis algorithms, including Breadth-First Search (BFS), Single-Source Shortest Path (SSSP), PageRank (PR), and Weakly Connected Components (WCC).

7.2 Experiment on Temporal Graph

7.2.1 Experiments on temporal graph storage consumption. We first use T-mgBench to analyze storage consumption under varying numbers of graph operations, as shown in Figure 6(a). The results show that TVA demonstrates the lowest storage overhead across different operation volumes. TVA's storage efficiency surpasses AeonG by up to 2.2×, Clock-G by up to 3.6×, and T-GQL by up to 4.7×. These storage savings primarily result from TVA 's compact storage structure. The system efficiently organizes historical data in the Temporal Buffer and employs a low-overhead structure to manage version metadata, thereby minimizing the overhead associated with maintaining historical data. In contrast, AeonG periodically stores full data within the version chain; T-GQL maintains complete information for all data; and Clock-G periodically creates full historical snapshots of the graph. All these approaches lead to higher storage overhead compared to TVA.

The results on the T-LDBC dataset, shown in Figure 6(c), further demonstrate this advantage and provide a clearer breakdown of storage composition. TVA achieves up to 1.2×, 6.7×, and 4.3× lower storage consumption than AeonG, Clock-G, and T-GQL, respectively. The total storage required by TVA remains minimal, with the "Historical Data" component being notably small in comparison to the "Current Data". In contrast, for Clock-G, the Historical Data constitutes a large proportion of the total storage. These results indicate that Clock-G's approach of periodically creating full historical snapshots substantially increases storage overhead.

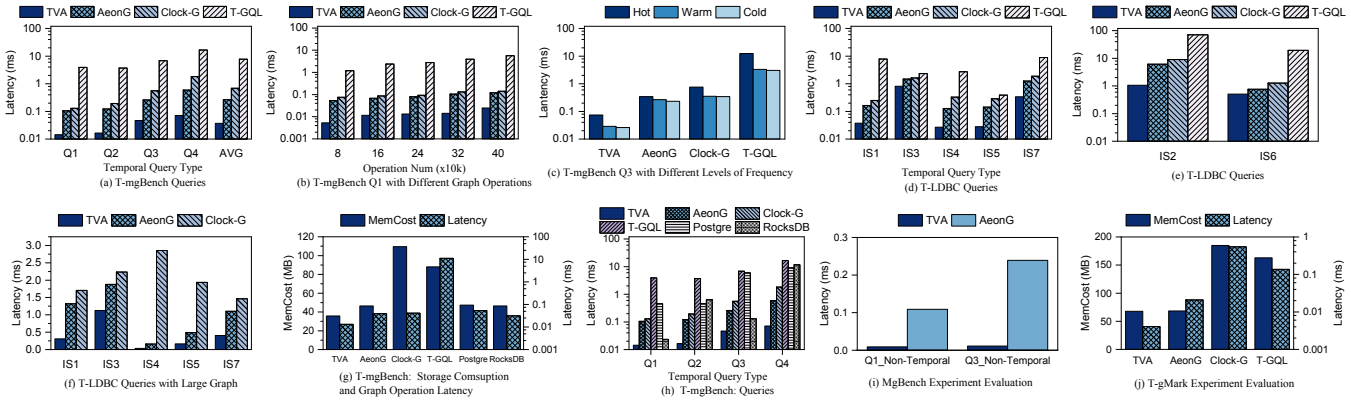

Figure 7: Comparisons on Temporal Query Latency.

7.2.2 Experiments on temporal graph operation latency. To evaluate the performance of temporal graph operations, we measured the average latency on the T-mgBench and T-LDBC, with the results presented in Figure 6(b) and Figure 6(d).

Figure 6(b) shows that on T-mgBench, TVA consistently maintains the lowest and most stable average graph operation latency as the number of operations increases from 80k to 400k. In contrast, other systems exhibit increased high latency. At 400k operations, TVA's average latency is 3×, 3.1×, and 1197.6× faster than AeonG, Clock-G, and T-GQL. This advantage is primarily due to TVA's efficient historical data management. Specifically, TVA avoids the overhead caused by AeonG's approach of archiving the latest data to a historical storage area and recording changes via a version chain, which introduces additional computational cost. Meanwhile, T-GQL suffers from inefficient traversal because it does not distinguish between current and historical data.

Subsequent experiments on the larger and more complex T-LDBC dataset, shown in Figure 6(d), further corroborate TVA's low-latency advantage. TVA records the lowest average operation latency, with its performance being 2.9× faster than AeonG, 8.1× faster than Clock-G, and 28.9× faster than T-GQL. Notably, on the large-scale T-LDBC graph, Clock-G's periodic creation of large historical snapshots consumes significant CPU and I/O resources. This resource contention adversely impacts its real-time graph operation performance, explaining its relatively higher latency.

7.2.3 Experiments on temporal graph analysis performance. We first evaluate temporal query performance using T-mgBench (Q1–Q4). As shown in Figure 7(a)-(c), TVA consistently outperforms AeonG, Clock-G, and T-GQL across all queries, achieving average latency reductions of up to 206.9×. This advantage stems primarily from our `Temporal Table` enabling efficient direct access to historical data, and our Hopscotch-based algorithm ensuring contiguous storage to minimize unnecessary scans. In contrast, AeonG suffers from traversing long version chains, Clock-G incurs overhead from aggregating multiple snapshots, and T-GQL performs expensive full-graph traversals. Furthermore, TVA maintains this superior performance under varying workloads, outperforming competitors even as the volume of graph operations scales up.

To validate the effectiveness of TVA in larger-scale and more complex scenarios, we conducted evaluations on T-LDBC. As shown in Figures 7(d) and (e), TVA continues to demonstrate superior performance across various query types, achieving latency reductions of up to 4.4×, 6.6×, and 207.1× compared to AeonG, Clock-G, and T-GQL, respectively. To further stress-test the systems, we evaluated an even larger graph containing 9.28 million vertices and 52.7 million edges. During this experiment, T-GQL failed to complete the test due to an Out-Of-Memory error. The results for the remaining systems are shown in Figure 7(f), where TVA achieves performance improvements of up to 4.7× and 83.9× over AeonG and Clock-G.

To justify the necessity of a specialized temporal graph store over general-purpose systems, we compared TVA against PostgreSQL (representing temporal-extended RDBMS) and RocksDB (representing Key-Value stores). The results in Figure 7(g) and (h) reveal fundamental architectural bottlenecks in these baselines. PostgreSQL relies on expensive join operations to traverse edges, resulting in a combinatorial explosion of intermediate results and a sharp decline in performance for multi-hop temporal queries. By utilizing a version-aware storage layout that replaces costly joins with direct pointer dereferencing, TVA achieves a 126.4× performance improvement over PostgreSQL. Similarly, although RocksDB is efficient for simple point queries, it lacks structural awareness, leading to excessive random I/O and inefficient scans during the multi-hop traversals and temporal range scans required by time-series graph workloads.

To address concerns about system overhead in non-temporal scenarios, we compared the performance of TVA and baseline systems on mgBench with all temporal data excluded. The results in Figure 7(i) indicate that while all systems naturally experience a performance boost due to the elimination of version retrieval steps, TVA maintains its superior architectural efficiency, achieving a 21.7× performance improvement over AeonG on static snapshots.

Furthermore, to evaluate property-intensive workloads on T-gMark, Figure 7(j) shows TVA's architecture decouples property values from version metadata, enabling parallel retrieval of multiple properties once the version metadata is located in the `Temporal Table`. This yields speedups of 5.1×, 132.9×, and 33.4× over AeonG, Clock-G, and T-GQL, respectively.

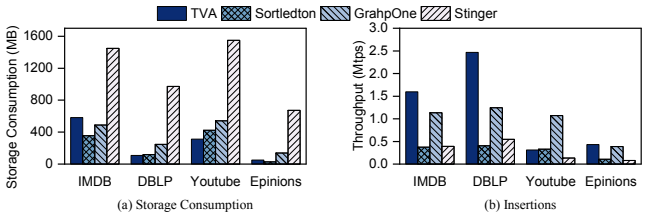

Figure 8: Current Graph Storage Consumption and Graph Operation Throughput.

7.3 Experiment on Current Graph

7.3.1 Experiments on current graph storage consumption. We first evaluate the current data storage overhead of TVA. The results are shown in Figure 8(a). The results demonstrate that TVA exhibits superior storage efficiency on the majority of datasets. Compared to the baseline systems, TVA 's space utilization is up to 1.4×, 1.8×, and 5.1× higher than Sortledton, GraphOne, and Stinger, respectively. An exception occurs on the IMDB and Epinions datasets, where TVA 's storage overhead is slightly higher than that of Sortledton. This is attributed to TVA 's storage management strategy, which pre-allocates fixed-size space for low-degree vertices. When the graph contains a large number of vertices with very low degrees, this strategy can lead to some wasted space. We consider this overhead acceptable because the reserved space is a key design choice that enables high insertion throughput for low-degree vertices.

7.3.2 Experiments on current graph operation latency. We then evaluate the insertion performance of different graph systems. Figure 8(b) shows the insertion throughput across multiple datasets. On the YouTube dataset, GraphOne marginally outperforms TVA, mainly because most edges connect to a few high-degree vertices, causing more hash collisions during position lookups. Notably, GraphOne skips edge existence checks in this experiment; enabling such checks would reduce its throughput to about 5 edges per second [17], due to the overhead of snapshot creation. On all other datasets, TVA consistently achieves up to 6.1×, 2×, and 4.5× higher insertion throughput than Sortledton, GraphOne, and Stinger. This can be attributed to TVA's hybrid insertion approach: it pre-allocates space for low-degree vertices to avoid frequent memory operations and utilizes a HopscotchHash Table for fast indexing of high-degree vertices, supporting efficient edge insertions. These features highlight TVA 's adaptability to diverse graph structures.

7.3.3 Experiments on current graph query latency. We further evaluate the performance of typical graph analysis tasks by measuring the execution times of four graph analysis algorithms, and compare TVA with its competitors. Figure 9 presents the normalized execution times of different systems across various datasets, where the execution time of TVA is set as the baseline and normalized to 1. TVA outperforms other methods in most cases, achieving speedups of up to 6.5×, 18.4×, and 5.6× compared to Sortledton, GraphOne, and Stinger. GraphOne suffers the largest performance decline during graph analysis due to its block-based adjacency list operations, which cause substantial random access. Sortledton outperforms others in BFS because its skiplist structure makes it convenient to access all neighbors of a vertex, while TVA incurs extra overhead

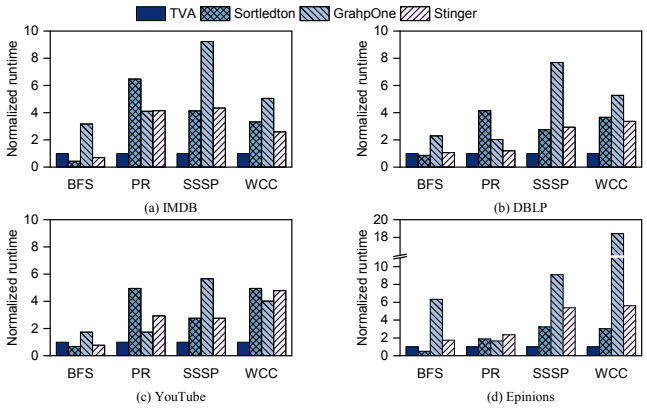

Figure 9: Current Graph Query Latency.

from hash table lookups during edge traversal. However, this approach is less effective for algorithms such as PR and SSSP that require frequent access to neighbors of different vertices.

7.4 Persistence Analysis

7.4.1 Effectiveness of TemporalChain in Long Version Chains. We evaluate TemporalChain for querying items at a specific timestamp. In Figure 10(a), TemporalChain achieves up to 6.2× speedup over independent item lookups by leveraging inter-item links to directly navigate to subsequent relevant versions. By confining the search scope within a bounded neighborhood and utilizing TemporalChain to bypass repeated full-chain traversals, TVA effectively circumvents the linear performance degradation ($O(N_v)$) typical of traditional linked-list version-chain approaches. Our design ensures that version retrieval time remains logarithmic ($O(\log N_v)$), making TVA's computational advantage increasingly prominent under heavy, history-intensive workloads.

7.4.2 Persistence Overhead. To rigorously quantify the cost of durability and I/O overhead when datasets exceed memory capacity, we evaluate TVA under two distinct memory-spillover persistence strategies managed by the PersistManager: TVA_1 (Pointer Mode) and TVA_2 (Full Mode). Figures 10(b) and (c) detail the storage footprint and operation latency. The Pointer Mode retains lightweight topology metadata in RAM while offloading bulky historical property values to an append-only disk file via 64-bit offsets, thus reducing the memory footprint. The Full Mode fully persists both topology and properties by leveraging RocksDB as a backend, utilizing a specialized time-encoded key schema (e.g., `user:uid:ts:timestamp`) to exploit LSM-tree lexicographical ordering so that historical versions remain physically adjacent on disk. Furthermore, we explicitly measured the overhead of our Write-Ahead Logging (WAL) mechanism. By serializing both version metadata and actual data into a highly compact binary log format—featuring a lightweight header and CRC32 checksum for data integrity—our persistence mechanism introduces only marginal latency penalties. The results confirm TVA maintains robust, competitive throughput even under strict durability guarantees.

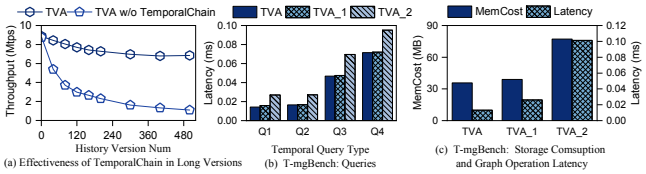

Figure 10: Persistence Analysis on TVA.

7.5 Ablation Study on TVA

7.5.1 Effectiveness of HopscotchTable. We evaluate the Hopscotch-Hash Table's hop distance and query efficiency. Figure 11(a) shows that under varying Zipf skewness (α), the hop distance for 99% of data insertions peaks at 8 and strictly remains below 16. This ensures versions are localized within a bounded, SIMD-friendly neighborhood. Compared to a standard MVCC version-chain baseline [15], Figure 11(b) demonstrates that HopscotchHash Table achieves up to 2.9× speedup as the history length grows, reducing lookup complexity to $O(\log N_v)$ via binary search.

7.5.2 Effectiveness of SIMD. As shown in Figure 11(c), leveraging SIMD instructions provides up to 1.4× query speedup by enabling parallel 16-byte candidate filtering. While the relative performance gap narrows with longer historical version chains and larger data volumes, SIMD remains a valuable optimization for rapid candidate screening when items have fewer historical versions.

7.5.3 Effectiveness of Migration Mechanism. We migrate a vertex to the HopscotchHashTable when either its degree exceeds T_{deg} or its update frequency exceeds T_{ver}; Figure 11(d) identifies the optimal boundary via normalized runtime across varying degrees. To avoid latency penalties, migration runs asynchronously: the triggering write flags the vertex and completes without blocking, while a background worker allocates a new bucket, re-organizes the edges, and atomically updates the pointer.

7.5.4 Scalability and Update Performance. To explicitly demonstrate TVA's concurrency and scalability under update-intensive conditions, we evaluated mixed read-write workloads. Figure 11(e) illustrates that TVA achieves near-linear throughput scaling as thread counts increase. Furthermore, Figure 11(f) utilizes varying write-read ratios to simulate realistic update-intensive workloads in concurrent operations. The results verify that our fine-grained Snapshot Isolation (SI) implementation, which locks only at the individual object's Temporal Table Header rather than globally, effectively minimizes lock contention. This allows TVA to maintain high query throughput without blocking readers, even under severe update pressure.

8 RELATED WORKS

Temporal Graph Management. Recently, many works focus on managing and querying temporal graph data. AeonG [13] extends graph databases with temporal support, but it cannot completely address the disadvantages posed by version chain scanning. Snapshot-based systems such as T-GQL [6] provide clear historical models but suffer from high storage costs and inefficient query performance due to full snapshot reconstruction. Kineograph [5] also explores

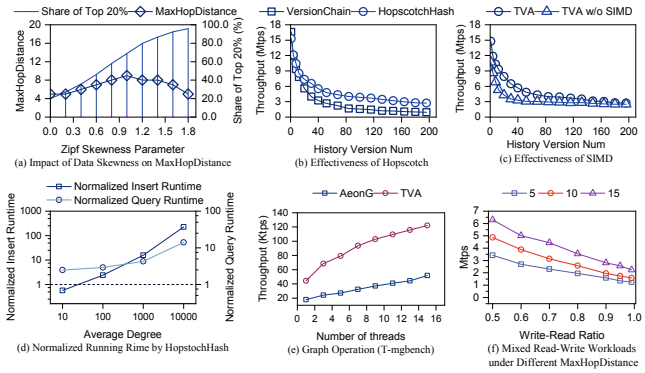

Figure 11: Performance Analysis on TVA.

fast-changing graphs, but with a focus on distributed environments and only capturing recent changes.

Multi-Version Database Systems. Classic multi-version database systems (MVCC) were primarily designed for high-concurrency transactional workloads, where historical versions are side effects of isolation rather than a persistent analytical asset [15]. Consequently, MVCC systems typically assume version chains are short and rely on aggressive garbage collection or vacuuming to remove obsolete versions and control storage growth and lookup overhead [29, 36]. In contrast, temporal graph databases must preserve long histories for time snapshot and temporal-range analytics, and their workloads further stress version management, making strong temporal locality across scans critical.

General-purpose Storage Systems. Temporal graphs can be mapped to relational tables such as vertex and edge tables and queried with relational operators, and modern RDBMSs also provide temporal features [9, 22]. However, even with temporal extensions, relational execution for multi-hop traversals typically requires repeated joins, which can cause a combinatorial blow-up as hop count increases. Likewise, KV stores excel at point lookups but lack structural awareness of graph topology. As a result, multi-hop traversals and temporal range scans suffer from excessive random accesses and poor locality.

9 CONCLUSION

This paper presented TVA, an efficient temporal graph storage system. TVA decoupled version metadata from actual data, and based on this principle, we designed tailored data structures, i.e., the temporal table and the enhanced hopscotch-based hash table, to enable fast version retrieval. Further, we introduced a version-skipping approach and leveraged SIMD parallelism to support high-performance temporal query processing. Experimental results demonstrated that TVA achieves superior performance in terms of temporal query latency, update efficiency, and storage compactness compared to state-of-the-art graph storage systems.

ACKNOWLEDGMENTS

This work is supported by the National Natural Science Foundation of China under Grant 62441230, 62461146205, 62072458 and 62472429.

REFERENCES

[1] Daniel J. Abadi, Samuel Madden, and Nabil Hachem. 2008. Column-stores vs. row-stores: how different are they really?. In *SIGMOD Conference*. ACM, 967–980.

[2] Guillaume Bagan, Angela Bonifati, Radu Ciucanu, George H. L. Fletcher, Aurélien Lemay, and Nicky Advokaat. 2017. gMark: Schema-Driven Generation of Graphs and Queries. *IEEE Trans. Knowl. Data Eng.* 29, 4 (2017), 856–869.

[3] Shaosheng Cao, Xinxing Yang, Cen Chen, Jun Zhou, Xiaolong Li, and Yuan Qi. 2019. TitAnt: Online Real-time Transaction Fraud Detection in Ant Financial. *Proc. VLDB Endow.* 12, 12 (2019), 2082–2093.

[4] Dawei Cheng, Xiaoyang Wang, Ying Zhang, and Liqing Zhang. 2022. Graph Neural Network for Fraud Detection via Spatial-Temporal Attention. *IEEE Trans. Knowl. Data Eng.* 34, 8 (2022), 3800–3813.

[5] 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 *Proceedings of the 7th ACM european conference on Computer Systems*. 85–98.

[6] Ariel Debrouvier, Eliseo Parodi, Matías Perazzo, Valeria Soliani, and Alejandro A. Vaisman. 2021. A model and query language for temporal graph databases. *VLDB J.* 30, 5 (2021), 825–858.

[7] David Ediger, Robert McColl, E. Jason Riedy, and David A. Bader. 2012. STINGER: High performance data structure for streaming graphs. In *HPEC*. IEEE, 1–5.

[8] Orri Erling, Alex Averbuch, Josep Lluís Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat-Pérez, Minh-Duc Pham, and Peter A. Boncz. 2015. The LDBC Social Network Benchmark: Interactive Workload. In *SIGMOD Conference*. ACM, 619–630.

[9] Facebook. 2016. RocksDB. https://github.com/facebook/rocksdb.

[10] Per Fuchs, Jana Giceva, and Domagoj Margan. 2022. Sortledton: a universal, transactional graph data structure. *Proc. VLDB Endow.* 15, 6 (2022), 1173–1186.

[11] 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*. ACM, 1:1–1:14.

[12] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. 2008. Hopscotch Hashing. In *DISC (Lecture Notes in Computer Science)*, Vol. 5218. Springer, 350–364.

[13] Jiamin Hou, Zhanhao Zhao, Zhouyu Wang, Wei Lu, Guodong Jin, Dong Wen, and Xiaoyong Du. 2024. AeonG: An Efficient Built-in Temporal Support in Graph Databases. *Proc. VLDB Endow.* 17, 6 (2024), 1515–1527.

[14] Jiaxin Jiang, Siyuan Yao, Yuchen Li, Qiange Wang, Bingsheng He, and Min Chen. 2025. Dupin: A Parallel Framework for Densest Subgraph Discovery in Fraud Detection on Massive Graphs. *Proceedings of the ACM on Management of Data* 3, 3 (2025), 1–26.

[15] Jong-Bin Kim, Kihwang Kim, Hyunsoo Cho, Jaeseon Yu, Sooyong Kang, and Hyungsoo Jung. 2021. Rethink the Scan in MVCC Databases. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 938–950. https://doi.org/10.1145/3448016.3452783

[16] Pradeep Kumar and H. Howie Huang. 2019. GraphOne: A Data Store for Real-time Analytics on Evolving Graphs. In *FAST*. USENIX Association, 249–263.

[17] Dean De Leo and Peter A. Boncz. 2021. Teseo and the Analysis of Structural Dynamic Graphs. *Proc. VLDB Endow.* 14, 6 (2021), 1053–1066.

[18] David B. Lomet, Roger S. Barga, Mohamed F. Mokbel, German Shegalov, Rui Wang, and Yunyue Zhu. 2005. Immortal DB: transaction time support for SQL server. In *SIGMOD Conference*. ACM, 939–941.

[19] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *SIGMOD Conference*. ACM, 135–146.

[20] Maria Massri, Zoltán Miklós, Philippe Raipin Parvédy, and Pierre Meye. 2022. Clock-G: A temporal graph management system with space-efficient storage technique. In *ICDE*. IEEE, 2263–2276.

[21] Youshan Miao, Wentao Han, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Enhong Chen, and Wenguang Chen. 2015. ImmortalGraph: A System for Storage and Analysis of Temporal Graphs. *ACM Trans. Storage* 11, 3 (2015), 14:1–14:34.

[22] PostgreSQL wiki. 2025. Temporal Extensions – PostgreSQL Wiki. https://wiki.postgresql.org/wiki/Temporal_Extensions. Accessed: 2026-01.

[23] Mark Raasveldt and Hannes Mühleisen. 2019. Duckdb: an embeddable analytical database. In *Proceedings of the 2019 international conference on management of data*. 1981–1984.

[24] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *AAAI*. https://networkrepository.com

[25] Owen Sacco, John G. Breslin, and Stefan Decker. 2013. Fine-Grained Trust Assertions for Privacy Management in the Social Semantic Web. In *TrustCom/ISPA/IUCC*. IEEE Computer Society, 218–225.

[26] Nedson Donato Soares, Regina Braga, José Maria N. David, Kennya Beatriz Siqueira, and Victor Ströele. 2022. Data Analysis in Social Networks for Agribusiness - A Systematic Mapping Study. *CoRR* abs/2208.14807 (2022).

[27] Wenhao Li. 2026. TVA. https://github.com/Sakuraaa0/TVA.

[28] Christo Wilson, Bryce Boe, Alessandra Sala, Krishna P. N. Puttaswamy, and Ben Y. Zhao. 2009. User interactions in social networks and their implications. In *EuroSys*. ACM, 205–218.

[29] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. 2017. An Empirical Evaluation of In-Memory Multi-Version Concurrency Control. *Proc. VLDB Endow.* 10, 7 (2017), 781–792. https://doi.org/10.14778/3067421.3067427

[30] Yuyang Xia, Yixiang Fang, and Wensheng Luo. 2025. Efficiently Counting Triangles in Large Temporal Graphs. *Proc. ACM Manag. Data* 3, 1 (2025), 38:1–38:27.

[31] Sheng Xiang, Guibin Zhang, Dawei Cheng, and Ying Zhang. 2025. Enhancing Attribute-Driven Fraud Detection With Risk-Aware Graph Representation. *IEEE Trans. Knowl. Data Eng.* 37, 5 (2025), 2501–2512.

[32] Fei Xiao, Yuncheng Wu, Meihui Zhang, Gang Chen, and Beng Chin Ooi. 2023. MINT: Detecting Fraudulent Behaviors from Time-series Relational Data. *Proc. VLDB Endow.* 16, 12 (2023), 3610–3623.

[33] Jaewon Yang and Jure Leskovec. 2012. Defining and Evaluating Network Communities Based on Ground-Truth. In *ICDM*. IEEE Computer Society, 745–754.

[34] Chang Ye, Yuchen Li, Bingsheng He, Zhao Li, and Jianling Sun. 2021. GPU-Accelerated Graph Label Propagation for Real-Time Fraud Detection. In *SIGMOD Conference*. ACM, 2348–2356.

[35] Song Yu, Shufeng Gong, Qian Tao, Sijie Shen, Yanfeng Zhang, Wenyuan Yu, Pengxi Liu, Zhixin Zhang, Hongfu Li, Xiaojian Luo, Ge Yu, and Jingren Zhou. 2024. LSMGraph: A High-Performance Dynamic Graph Storage System with Multi-Level CSR. *Proc. ACM Manag. Data* 2, 6 (2024), 243:1–243:28.

[36] Ming Zhang, Yu Hua, and Zhijun Yang. 2024. Motor: Enabling Multi-Versioning for Distributed Transactions on Disaggregated Memory. In *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024*, Ada Gavrilovska and Douglas B. Terry (Eds.). USENIX Association, 801–819. https://www.usenix.org/conference/osdi24/presentation/zhang-ming

[37] Wei Zhang, Cheng Chen, Qiange Wang, Wei Wang, Shijiao Yang, Bingyu Zhou, Huiming Zhu, Chao Chen, Yongjun Zhao, Yingqian Hu, Miaomiao Cheng, Meng Li, Hongfei Tan, Mengjin Liu, Hexiang Lin, Shuai Zhang, and Lei Zhang. 2024. BG3: A Cost Effective and I/O Efficient Graph Database in Bytedance. In *SIGMOD Conference Companion*. ACM, 360–372.