The VLDB Journal (2025) 34:53
https://doi.org/10.1007/s00778-025-00932-w

REGULAR PAPER

An efficient and scalable graph database with built-in temporal support

Jiamin Hou[1,2] · Zhanhao Zhao[1,2] · Wei Lu[1,2] · Shiming Yang[1,2] · Shuang Liu[1,2] · Quanqing Xu[3] · Chuanhui Yang[3] · Xiaoyong Du[1,2]

Received: 30 September 2024 / Revised: 26 April 2025 / Accepted: 16 June 2025

Abstract

Real-world graphs are often dynamic and evolve over time. It is crucial for storing and querying a graph's evolution in graph databases. However, existing works either suffer from high storage overhead or lack efficient temporal query support, or both. In this paper, we propose AeonG, a new graph database with built-in temporal support. Based on a novel temporal graph model, we build AeonG with a hybrid storage engine and an efficient temporal query engine. The storage engine consists of current storage to manage the most recent versions of graph objects, and historical storage to manage previous versions. This separation minimizes the performance degradation when querying the most recent graph object versions. To reduce the historical storage overhead, we propose an *anchor+delta* strategy, in which we periodically create a complete version (namely anchor) of a graph object, and maintain every change (namely delta) between two adjacent anchors of the same object. In the query engine, we propose an anchor-based version retrieval technique to skip unnecessary historical version traversals to boost temporal query processing. Further, we extend AeonG into a cloud-native database with disaggregated compute and storage layers, thus enabling elastic and scalable management of temporal graph data. Extensive experiments are conducted on both real and synthetic datasets. The results show that AeonG achieves up to 5.73× lower storage consumption and 2.57× lower temporal query latency against state-of-the-art approaches, while introducing only 9.74% performance degradation for supporting temporal features.

Keywords Temporal graph · Graph database · Cloud-native database · Transaction time

1 Introduction

Graphs are prevalent in modeling relationships between real-world entities. To effectively manage graph data, many graph databases are developed, such as Neo4j [44], ArangoDB [7], Dgraph [15], and Memgraph [40]. Despite the fact that real-world graphs are often dynamic and evolve over time, these databases are typically designed to manage up-to-date graph data: when a graph changes, the database only stores the current (latest) state of the graph, i.e., the most recent values of vertices and edges, while discarding any previous (historical) state. However, time-evolving (temporal) graph data, which contains both the latest and historical states of a graph, is vital in many applications, such as financial fraud detection [1], traffic prediction [35], etc.

✉ Zhanhao Zhao
zhanhaozhao@ruc.edu.cn

Jiamin Hou
jiaminhou@ruc.edu.cn

Wei Lu
lu-wei@ruc.edu.cn

Shiming Yang
yang_shiming@ruc.edu.cn

Shuang Liu
shuang.liu@ruc.edu.cn

Quanqing Xu
xuquanqing.xqq@oceanbase.com

Chuanhui Yang
rizhao.ych@oceanbase.com

Xiaoyong Du
duyong@ruc.edu.cn

[1] Key Laboratory of Data Engineering and Knowledge Engineering, Ministry of Education, Beijing, China

[2] School of Information, Renmin University of China, Beijing, China

[3] OceanBase, Ant Group, Beijing, China

Published online: 26 June 2025

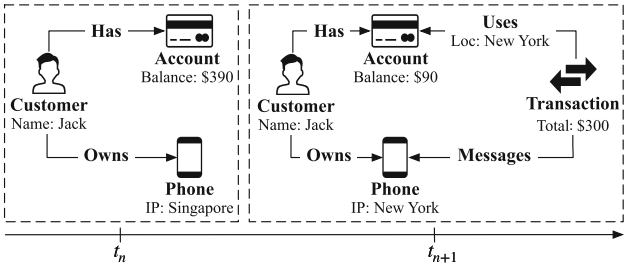


Fig. 1 The Evolution of a Customer Purchase Graph

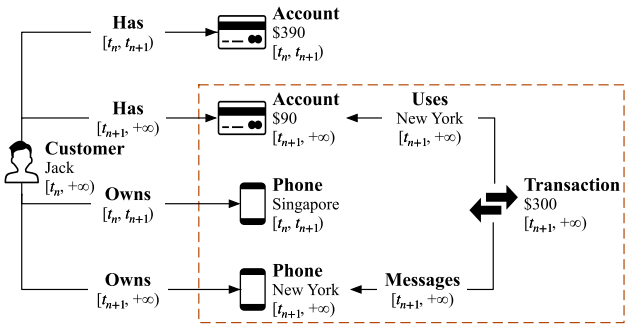


Fig. 2 Customer Purchase Graph with a Time Dimension

Example 1 Figure 1 shows the evolution of a customer purchase graph, where entities such as customers, bank accounts, phones, and transactions are modeled as vertices, and their relationships as edges. The phone logs its location during activities like receiving messages or web browsing. Each customer purchase invokes a transaction to update the account balance and record the transaction location. At time t_n, a customer named Jack has an account balance of $390, and his phone's location falls in Singapore. We store a graph reflecting this state, as shown in the left portion of Figure 1. At time t_{n+1}(one minute later), Jack initiates a $300 purchase in New York, updating the graph to the state shown on the right. The transaction location is identical to Jack's phone location at t_{n+1}, thus it appears to be legitimate. However, comparing the states at t_{n+1} and t_n reveals that Jack's phone moved from Singapore to New York within one minute—an implausible scenario indicating potential fraud. We would like to emphasize that changes in phone location alone are not inherently suspicious. However, when such a location shift is associated with a transaction, it warrants vigilance. As discussed above, this potentially fraudulent activity can only be identified by tracking the evolution of the graph structure over time. Therefore, traditional graph databases, which only maintain the latest state at t_{n+1}, would fail to detect such fraudulent transactions.

Thus far, various works have been proposed to manage temporal graph data. Several proposals [13, 14, 16, 50] assign timestamp properties to vertices and edges to represent their lifespans. As illustrated in Figure 2, instead of discarding previous states when the graph changes, these approaches maintain both current and historical states within a single graph. For instance, at time t_{n+1}, two vertices of Jack's phone coexist: one for the previous state with a time interval $[t_n, t_{n+1})$ and another for the current state with $[t_{n+1}, +\infty)$. This supports fraud detection, as in Example 1, by identifying the irregular sub-graph structure (highlighted in red) that indicates a transaction proceeded with location inconsistency. However, querying temporal data requires traversing the entire graph, leading to performance decline as historical states accumulate. Another line of research [11, 22, 23, 28, 29, 33, 38, 39, 42, 54] manages time-evolving graph data by periodically materializing the snapshots of the entire graph and logging the deltas between two successive snapshots. Querying a historical state requires first locating the nearest snapshot based on the timestamp, and then reconstructing the complete graph state using the snapshot and associated deltas. Therefore, these methods incur substantial storage overhead due to the maintenance of complete snapshots, and can lead to sub-optimal query performance because of the historical state reconstruction.

Our goal is to design a graph database for efficient temporal graph data management. This requires addressing four key challenges: First, as the volume of historical graph states grows continuously, minimizing storage overhead is non-trivial❶. Second, given the substantial amount of historical data, efficiently processing temporal queries while maintaining data consistency poses significant challenges❷. Third, native temporal support is required to allow users to conveniently query temporal graph data❸. Lastly, unlike traditional graph databases that only manage current data, a temporal graph database must be scalable to efficiently store and query ever-growing historical states❹.

We propose AeonG, a graph database with efficient built-in temporal support. By integrating the popular static property graph model with a time dimension, we define a *temporal property graph model* to formalize the representation and manipulation of temporal graph data. Guided by this model, we extend the common graph database architecture to design AeonG, enhancing the storage engine, query language, and query engine for efficient temporal processing. We build a *hybrid storage engine*❶, constituting current and historical storage to manage temporal graph data with minimal overhead. This engine maintains multiple versions for each vertex and edge, with the most recent versions retained in the current storage and previous versions in the historical storage. We integrate time dimensions into the data layout and develop the current storage based on the multi-version storage engine used in various existing graph databases. We propose a novel "anchor+delta" strategy to compactly organize historical data in the historical storage. In particular, we periodically create a complete version (namely anchor)

of a graph object and maintain every change (namely delta) between two consecutive anchors of the same object to reduce the historical storage overhead. Moreover, we introduce an asynchronous migration mechanism, deferring the transfer of outdated versions to historical storage until a periodic garbage collection is triggered [60], minimizing performance degradation.

We then introduce a *temporal-enhanced query language*❸, extending Cypher [19] to support temporal graph data access. Built upon our hybrid storage engine, we develop a built-in *temporal query engine*❷. We inherit two fundamental operations from existing graph databases, namely scan and expand, and extend them to enable consistent and efficient temporal query processing. For consistency, we leverage a snapshot visibility check for current data and a legal check mechanism for historical data. For efficiency, we propose a dual-index mechanism for fast temporal lookups and an anchor-based retrieval strategy to avoid unnecessary historical version traversals.

To further enhance system scalability and elasticity, we extend AeonG into a cloud-native database, namely AeonG-C❹. Aligning with cloud-native disaggregated architecture, we enhance the replay mechanism to manage both current and historical data. Furthermore, we propose an asynchronous prefetching approach to improve query efficiency and implement a two-phase fetching strategy to ensure consistency between the historical data caches and remote storage.

In summary, we make the following contributions:

- We present AeonG, a new graph database providing efficient built-in temporal support. Built with a temporal-enhanced query language, query engine, and storage engine, AeonG regards temporal features as the first citizen, making it simple and intuitive to manipulate temporal graph data.
- We propose a hybrid storage engine that employs an "anchor+delta" strategy to minimize storage overhead for historical data. In addition, we introduce an asynchronous migration strategy to minimize the overhead caused by the maintenance of temporal data.
- We design a temporal query engine, featuring a dual-index approach and an anchor-based version retrieval technique, to provide consistent and efficient temporal query processing with minimal historical version traversal overhead.
- We propose the cloud-native version of AeonG, called AeonG-C. By disaggregating the compute layer from the storage layer, AeonG-C enables elastic and scalable temporal graph data storage and retrieval.
- We implement AeonG and AeonG-C based on Memgraph [40], a real-world graph database. We conduct extensive experiments on both real and synthetic datasets, and compare AeonG against two state-of-the-art temporal graph databases [14, 39]. The results demonstrate that AeonG achieves up to 5.73× lower storage consumption and 2.57× lower latency for temporal queries, while only introducing 9.74% performance degradation for supporting temporal features. We also demonstrate the good performance and scalability of AeonG-C.

2 Modeling and query language

In this section, we formulate the temporal property graph model and present the temporal query language.

2.1 Temporal Property Graph Model

We define the temporal graph model by extending the static property graph model [3–5, 19] with a time dimension. In the property graph model, real-world entities are represented as vertices, and their relationships are modeled as edges. Each vertex or edge has a unique identifier (id for short), possibly several labels (e.g., customer, phone), and properties (e.g., Name: Jack).

Definition 1 (Property Graph) Let $\mathcal{N}$ and $\mathcal{E}$ denote sets of vertex ids and edge ids, respectively. Assume countable sets $\mathcal{L}$, $\mathcal{K}$, and $\mathcal{V}$ of *labels, property names*, and *property values*. A property graph is a tuple $G = \langle N, E, \rho, \lambda, \pi \rangle$ where:

- N is a finite subset of $\mathcal{N}$, whose elements are referred to as the *vertices* of G;
- E is a finite subset of $\mathcal{E}$, whose elements are referred to as the *edges* of G and $N \cap E = \emptyset$;
- ρ: $E \rightarrow (N \times N)$ is a total function mapping each edge to its source and destination vertices;
- $\lambda : (N \cup E) \rightarrow 2^{\mathcal{L}}$ is a total function mapping vertices and edges to finite sets of labels (including the empty set);
- $\pi : (N \cup E) \times \mathcal{K} \rightarrow \mathcal{V}$ is a finite partial function, mapping a vertex/edge and a property key to a value.

The property graph model, originally designed for static graphs, lacks the inherent ability to capture temporal evolution. In relational databases, the concept of "Transaction Time" [32] introduces a time dimension to track the lifespan of each data item within the system. Inspired by the transaction time, we integrate the time dimension into the property graph model to formally define the temporal property graph model.

Definition 2 (Temporal Property Graph) A temporal property graph is a tuple $G = \langle \Omega, N, E, \rho, \lambda, \pi, \sigma, \tau \rangle$ where:

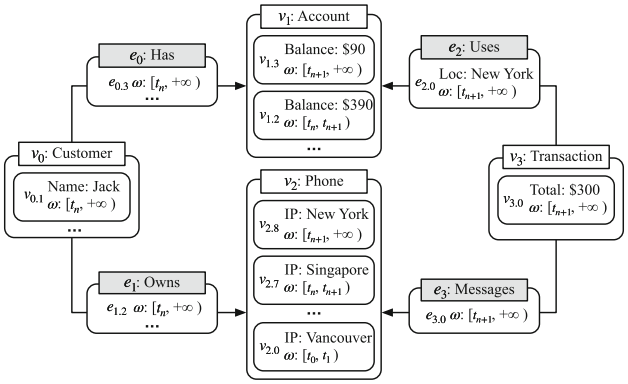


Fig. 3 A Running Example of Temporal Property Graphs

- Ω is a temporal domain, which is a finite set of consecutive timestamps. Formally, $\exists a, b \in T,\ a \leq b$, such that $\Omega = \{i \in T | a \leq i \leq b\}$, where T denotes the universe of time points;
- N, E, ρ, λ, and π inherit their definitions from Definition 1;
- $\sigma : (N \cup E) \times \Omega \rightarrow \{true, false\}$ is a total function that maps a vertex or an edge and a time period $\omega \in \Omega$ to a Boolean variable, indicating whether this vertex or edge exists during the period ω;
- $\tau : (N \cup E) \times \mathcal{K} \times \Omega \rightarrow \mathcal{V}$ is a partial function that maps a vertex or an edge, a property key, and a time period $\omega \in \Omega$ to a value, indicating the property value of a vertex or an edge during the period ω.

Constraints. To enforce the validity of the temporal property graph at any time point t, we impose two constraints: (1) An edge exists only if both its source and destination vertices exist at t. Formally, if $e \in E$, $\sigma(e, t) = true$, and $\rho(e) = (v_1, v_2)$, then $\sigma(v_1, t) = true$ and $\sigma(v_2, t) = true$; (2) A property value is valid only during the existence of its associated vertex or edge. Formally, $\forall o \in (N \cup E)$, $\forall k \in \mathcal{K}$, $\forall val \in \mathcal{V}$, and $\forall t \in \Omega$, $\tau(o, k, t) = val \rightarrow \sigma(o, t) = true$.

In our model, each graph object consists of multiple versions—one current and potentially several historical. Unlike existing works such as T-GQL [14], which assigns a time period to each graph object, our model assigns the time period to each version of a graph object (vertex or edge). For example, as depicted in Figure 2, when updating the entity "Phone", existing models retain two complete "Phone" vertices within the same graph, resulting in two redundant, unchanged "Owns" edges. In contrast, we create a new version of the "Phone" vertex and re-link the "Owns" edge to this version, with changed attributes stored in the historical version. Consequently, our model is less complex but more efficient by avoiding the creation of redundant vertices and edges. At any given time t, a graph object version with $\omega = [st, ed)$ is said to be **legal** if $st \leq t < ed$. We classify a version as current if it is legal at the current time and as historical otherwise.

Graph operations. Our temporal model supports graph operations as follows. Assume these graph operations are issued by a transaction committed at t_1.

- *Creating* a vertex or an edge: This involves adding a vertex or edge with a current version having a time period $\omega = [t_1, +\infty)$.
- *Deleting* a vertex or an edge whose current version is with $\omega = [st, +\infty)$: This entails updating ω to $[st, t_1)$.
- *Updating* a vertex or an edge whose current version is with $\omega = [st, +\infty)$: This marks the current version as a historical version by updating ω to $[st, t_1)$ and generates a new current version with $\omega = [t_1, +\infty)$ representing the up-to-date semantics.

Example 2 We present the temporal property graph of Example 1 in Figure 3. Here, $\Omega = [t_0, +\infty]$, $N = [v_0, v_1, v_2, v_3]$, $E = [e_0, e_1, e_2, e_3]$. Each vertex and edge owns a current version and several historical versions from t_0 to t_n. For brevity, we omit graph states before t_n. At t_n, there exist three vertices (v_0, v_1 and v_2) and two edges ($e_0 : (v_0, v_1)$) and $e_1 : (v_0, v_2)$). For instance, v_2 owns a current version $v_{2.7}$, which has a unique id 2, a label "Phone", a property with the key-value pair (IP, Singapore), and a lifespan $\omega = [t_n, \infty)$. Subsequently, at t_{n+1}, consider a customer purchase transaction. It updates the properties of v_1 and v_2, resulting in new versions for each of them. Take v_2 as an example: it marks $v_{2.7}$ as a historical version and generates a new current version $v_{2.8}$. τ maps $v_{2.8} \times IP \times [t_{n+1}, +\infty)$ to *New York* and maps $v_{2.7} \times IP \times [t_n, t_{n+1})$ to *Singapore*. We regard $v_{2.7}$ as legal at t_n, but not legal at t_{n+1}. Moreover, this transaction also creates v_3, e_2 and e_3, which have only the current version with $\omega = [t_{n+1}, +\infty)$. All the aforementioned graph operations adhere to the defined constraints. For instance, e_2 can be successfully created at t_{n+1} only after verifying linked vertices v_3 and v_1 exist at t_{n+1} (Constraint 1).

2.2 Temporal Graph Query Language

AeonG incorporates Cypher [19] as its query language and extends the standard syntax to support temporal queries. Specifically, we inherit the existing syntax for graph operations to create, update, or delete graphs, while enhancing the read query syntax to conveniently retrieve temporal graphs. As illustrated in Listing 1, AeonG introduces two temporal syntax extensions in the `MATCH` clause (lines 3-4): (1) FOR TT AS OF t, which retrieves all graph objects legal at time t, and (2) FOR TT FROM t_1 TO t_2, which locates all graph objects consistently legal within the time range from t_1 to t_2. The former is referred to as "time-point" queries, while the latter is known as "time-slice" queries. Users can apply any

time conditions to temporal queries, spanning a wide time range from the oldest historical states up to the most recent updates.

Listing 1 Syntax of Temporal-enhanced Cypher

```
[OPTIONAL] MATCH pattern_tuple
           [WHERE expr]
           [FOR TT AS OF expr|
            FOR TT FROM expr TO expr]
```

Example 3 Consider the query "What was Jack's phone IP at t_n". This query can be answered by issuing the following statement, where the temporal syntax is underlined: "MATCH (:Customer {name: 'Jack'})-[r]-(p:Phone) FOR TT AS OF t_n return p.IP. "

3 System architecture

In this section, we introduce the system architecture as shown in Figure 4. AeonG includes a transaction manager that enables handling a sequence of graph operations with ACID properties. We process transactions by employing the Multi-Version Concurrency Control (MVCC) [45]. MVCC ensures that transactions only see a consistent snapshot of the data that is **visible** to them, thus enabling multiple transactions to work concurrently without interfering with one another [30, 45]. Given our primary focus on temporal data management, we now describe how we utilize the MVCC mechanism to manage temporal data effectively. AeonG supports built-in temporal features through two major components: the storage and the query engine.

3.1 Storage Engine

The storage engine of AeonG has two physically isolated storages: current storage and historical storage. The current storage typically maintains the current versions of graph objects. In contrast, the historical storage manages historical versions of graph objects, which are asynchronously migrated from the current storage.

Current storage. As outlined in Section 2.1, graphs evolve under various graph operations. To efficiently record these changes, AeonG builds its current storage as a multi-version storage, maintaining multiple versions for each graph object. Each graph object has one current version retaining the up-to-date state and is linked to a list of historical versions preserving the previous states. When a graph object is updated, instead of directly overwriting the data, we create a new current version and move the previous one to the list of historical versions. We integrate the time dimension into the data layout and modification paradigm to trace accurate graph evolution, as detailed in Section 4.1.

Historical storage. AeonG does not store historical versions in the current storage permanently. Instead, we migrate them to the historical storage for long-term maintenance. To handle the potentially large volume of historical data, we properly compress the historical storage. We organize migrated historical versions in a key-value format. The key contains the metadata of a historical version, while the value holds detailed properties of this version. Instead of retaining all properties for every version, we organize versions in an "anchor+delta" manner. We utilize deltas to record relative differences between subsequent versions, minimizing the storage cost of ever-growing historical data. In addition, after a series of deltas, we maintain an anchor to store the complete state of a graph object, facilitating the reconstruction process when executing temporal queries. We will introduce the details in Section 4.2.

Asynchronous migration. AeonG utilizes an asynchronous migration approach to transfer historical data from the current storage to the historical storage. Rather than triggering a migration immediately following an update, this migration is postponed until the MVCC garbage collection process. This design ensures that transferring ever-growing historical datais lightweight, thus minimizing overhead on the current storage. We will present our asynchronous migration in Section 4.2.

Example 4 In the right part of Figure 4, we demonstrate how AeonG stores the customer purchase graph as presented in Example 2. In the current storage, component Ⓒ records the current versions at t_{n+1}. Besides, the historical versions at t_n ($v_{1.2}$ and $v_{2.7}$ in this case), are stored in component Ⓓ. Take v_2 as an example. To capture the change in v_2's IP from Singapore to New York at t_{n+1}, AeonG performs two steps. First, it updates v_2 in place to create a new current version $v_{2.8}$. Second, to maintain the previous state, AeonG generates a historical version $v_{2.7}$, which is linked to $v_{2.8}$ in a chain and managed by MVCC. We migrate historical data in component Ⓓ to the historical storage (component Ⓔ) asynchronously. In the historical storage, the historical versions, $v_{1.2}$ and $v_{2.7}$, are organized as an anchor (represented as a long rectangle) and a delta (represented as a short rectangle), respectively.

3.2 Query Engine

The query engine is responsible for handling user-issued queries, retrieving relevant graph data from the hybrid storage engine. Adhering to the "textbook" separation of components, AeonG consists of a parser, an optimizer, and an executor. While inheriting those components from existing graph databases, AeonG further extends them to support temporal queries.

Parser and optimizer. The parser translates queries and generates corresponding syntax trees as input for the query

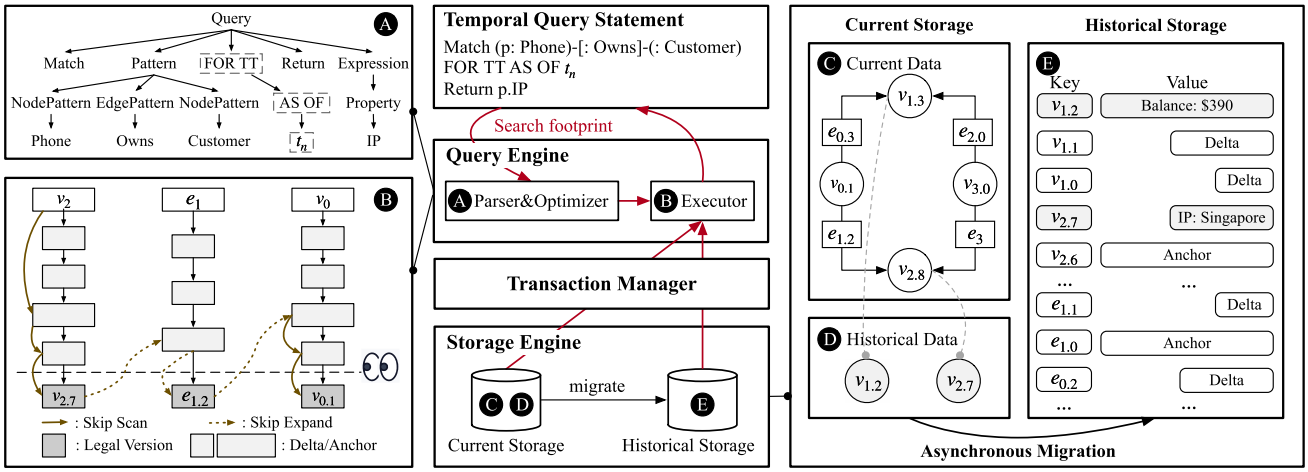


Fig. 4 An Overview of AeonG - AeonG consists of a temporal query engine, a hybrid storage engine, and an MVCC-based transaction manager. We employ an "anchor+delta" strategy to reduce the historical storage overhead, while using an anchor-based version retrieval technique to ensure efficient temporal query processing

optimizer. To support temporal query syntax, AeonG extends its lexical, syntactic, and semantic analyses to recognize time qualifiers as defined in Section 2.2. Using the resulting syntax tree, the optimizer generates execution plans for the executor.

Executor. AeonG extends two core fundamental operations from conventional graph databases, scan and expand. The scan operator retrieves the required vertex versions for each query, while the expand operator fetches relevant edge and adjacent vertex versions. We enhance these operators to enable consistent and efficient processing of temporal queries. First, to ensure consistency, we obtain current and historical data separately from two storage engines and then merge the results. For current data, we follow the conventional query mechanism, leveraging MVCC's snapshot visibility check [45] to capture visible graph object versions. For historical data, we address incomplete results caused by asynchronous migration (where some data remains in the current storage) through a legal check mechanism. We verify that the requested version(s) originate from consistent snapshot(s), thereby guaranteeing consistency. Second, we enhance existing indexing mechanisms to enable efficient index lookup for temporal queries. We propose a dual-index approach, including the vertex locating index to locate starting vertices for graph traversal, and the version index optimized for retrieving historical versions of vertices, edges, and topological structures. Finally, to efficiently traverse historical versions from substantial historical data, we propose an anchor-based version retrieval technique to minimize unnecessary traversals. To reconstruct a desired version, we directly locate the nearest anchor with a lifespan ω aligning with the query time constraint and apply subsequent deltas.

Example 5 Figure 4 (A) depicts a simplified syntax tree for a given temporal query statement. Based on it, AeonG then utilizes the executor to fetch query results from the hybrid storage engine. Figure 4 (B) illustrates the search footprint for the query "What were the phone IPs of all customers at t_n". We only reconstruct four relevant vertices/edges. Starting with the vertex v_2, we skip to seek its nearest anchor and collect all relevant deltas to reconstruct the desired legal version $v_{2.7}$. We then expand $v_{2.7}$ to get its linked edge $e_{1.2}$ and adjacency vertex $v_{0.1}$ without traversing the entire version chain of e_1 and v_0.

4 Hybrid Storage Engine

4.1 Current Storage

Inheriting existing native graph databases [21], AeonG organizes graph data into three storage components: (i) vertex properties (VP), (ii) edge properties (EP), and (iii) graph topology, i.e., vertices' incoming and outgoing edges (VE). Like most native graph databases [27, 40, 44], we embed the topology within vertices to enable efficient neighborhood traversal. However, tracking graph evolution under this design is non-trivial, as changes may occur in both semantics (e.g., properties) and structure (e.g., topology). We have to identify different types of operations applied on the graph. For example, we should prevent the creation of a new vertex version when the vertex's relevant graph topology changes but its properties remain unchanged. To address this, we associate the time dimension to each independent storage component to separately record semantic changes and structural changes.

Data layout. As shown in Figure 5, the data store comprises two components: the Vertex Store, which maintains a list of vertex objects, and the Edge Store, which stores a list of edge objects. Every vertex object v has a unique graph identifier

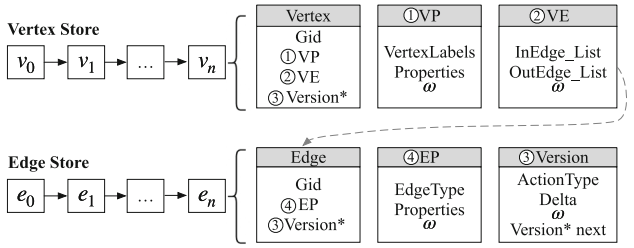


Fig. 5 Data Layout of Current Storage

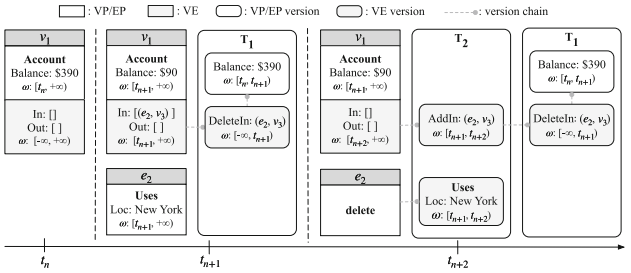


Fig. 6 An Example of Current Storage Layout

Gid, a VP part, a VE part, and a pointer to a linked list of historical versions (version chain). While every edge object e has a unique identifier Gid, an EP part, and a pointer to a linked list of historical versions. Specifically,

- The VP part stores a set of vertex labels and property value pairs associated with the current version of v, along with a time period ω indicating v's current semantic lifespan.
- The VE part tracks the current version of v's incoming and outgoing edges, represented as a list of (edge Gid, neighbor vertex Gid) pairs. It also includes ω to record v's current structural lifespan.
- The EP part stores an edge type and property value pairs of the current version of e, along with a time period ω indicating e's current semantic lifespan.
- Each historical version contains: an action type indicating the changes made to a VP, VE, or EP part; a delta recording steps to revert the changes to restore the previous version; a time period ω capturing the historical version's lifespan; and a pointer to the next historical version. All historical versions generated by the same transaction are clustered in an undo buffer following the MVCC mechanism.

Modification paradigm. We now elaborate on how AeonG evolves the graph to handle various graph operations. Suppose a graph operation is invoked by a transaction T_i with a commit time t_i. The modification paradigm for the graph data layout is as follows.

1. When a vertex is created, we create a vertex object, set its VP part's time as $\omega = [t_i, +\infty)$, set its VE part's time as $\omega = [-\infty, +\infty)$, and link it in the vertex object lists.
2. When an edge is created, we create an edge object, set its EP part's time as $\omega = [t_i, +\infty)$, and link it in the edge object lists. We also create connections to relevant vertex objects by setting their VE part's time to $\omega = [t_i, +\infty)$ if their previous VE part's time is $[-\infty, +\infty)$. Finally, we create VE versions to undo newly added edges with $\omega = [-\infty, t_i)$ and link them to the corresponding vertices' version chain.
3. When updating/creating/deleting a property value of a vertex object with $\omega = [t_j, +\infty)$, we first update relevant property values in the VP part and set its time as $\omega = [t_i, +\infty)$. We then create a historical VP version to capture the state of the vertex prior to the modification and set its ω as $[t_j, t_i)$. This VP version is linked to the vertex's version chain. Updating a property value of an edge follows the same logic.
4. When a vertex is deleted, we first delete all property values of the relevant vertex object, following the paradigm (3), and then delete all connected edges, following the paradigm (5).
5. When an edge is deleted, we decompose it into the deletion of all property values and the deletion of connections with relevant vertices. The former acts on the edge object, following the paradigm (3). The latter acts on the source and destination vertex object. Take the source vertex object with $\omega = [t_j, +\infty)$ as an example. We first update its VE part to delete this edge from outgoing edge lists and set its VE part's time to $\omega = [t_i, +\infty)$. We then create a VE version to record this deleted edge with $\omega = [t_j, t_i)$ and link it to the vertex's version chain.

Example 6 We illustrate the data layout of the current storage. As shown in Figure 6, we reconsider Example 2. To further represent the structural change, we suppose an event deleting e_2 at t_{n+2}. We showcase the graph evolution of v_1 and e_2. At t_{n+1}, the transaction T_1, representing a customer purchase, is committed. It first updates the VP part of v_1 and generates a VP version linked to it. Then it updates the VE part of v_1 and generates a linked VE version. Finally, it creates the graph object e_2. At t_{n+2}, T_2 is committed to delete e_2, which affects v_1 and e_2 objects. It first acts on e_2's EP part to clear all semantic information and generates an EP version to record the previous edge state. Then, it acts on the VE part of v_1 and generates a VE version.

4.2 Historical Storage

In MVCC, historical versions are not retained in the current storage permanently. Instead, once these versions are

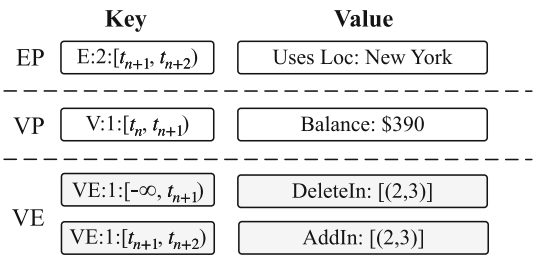


Fig. 7 Key-value Format in Historical Storage

Algorithm 1: Data migration

```
1 Function Migrate(CT):
    Input: CT, committed transaction no longer active;
2   kvs ← ∅; //reclaimed versions;
3   foreach undo ∈ CT do
4     kvs ←encode2KV(undo);
5   KV_store::putMultiples(kvs);
6   physically delete CT.undos;
7 End Function
```

no longer needed by any active transaction, they are safely removed through garbage collection (GC) to optimize the performance of the current storage. AeonG utilizes this mechanism to transfer those inaccessible versions to the historical storage for long-term maintenance. For the sake of communication, historical versions in the current storage are referred to as "unreclaimed", while those in the historical storage are referred to as "reclaimed". In this subsection, we first present the optimized key-value format for storing historical versions and then outline the process of migrating unreclaimed versions to the historical storage.

KV format. Reclaimed versions are organized in a key-value format, where the key represents the version's metadata, and the value stores the corresponding detailed information. AeonG groups three types of historical versions (VP, EP, and VE versions) into their respective segments. In each segment, the key consists of three elements: the $Type$ prefix, the graph identifier Gid, and ω of the version. The $Type$ prefix indicates the version type: 'V' for VP, 'E' for EP, and 'VE' for VE. The Gid is a unique identifier of the graph object linked to the version, while ω represents the version's lifespan. As for the value field, it contains the remaining semantic information, i.e., the delta of the version, which records only data changes compared to the previous version. This organization ensures that data sharing the same prefix in the key are physically clustered together in a SkipList [18], which ensures different versions of the same entity are automatically sorted based on their lifespan ω. As a result, it becomes efficient to retrieve in chronological order. Figure 7 illustrates the KV format of reclaimed historical versions, derived from the unreclaimed versions shown in Figure 6.

Anchor+delta. We utilize deltas to reduce the storage overhead. However, retrieving a reclaimed graph object requires assembling the latest version with all previous deltas, incurring significant reconstruction costs for long version chains. To mitigate this, we introduce anchors at intervals in the delta data, where an anchor represents the complete state of a graph object. To differentiate anchors from deltas in the KV store, we append a one-character suffix to the key's $Type$, where 'A' denotes anchors and 'D' represents deltas. Specifically, to reconstruct a certain reclaimed version o_1, we seek its most recent anchor o_2, collect all deltas from o_2 to o_1, and combine them to reconstruct o_1.

We propose an adaptive anchoring approach, which assigns a proper anchor interval u_o for each graph object based on Eq. (1):

$$u_o = \begin{cases} c & f(o) \le \tau_1 \quad \text{low} \\ \rho \cdot c & \tau_1 < f(o) \le \tau_2 \quad \text{medium} \\ \rho^2 \cdot c & \tau_2 \le f(o) \quad \text{high} \end{cases} \tag{1}$$

Intuitively, graph objects with higher update frequencies require larger anchor intervals to seek a balance between query efficiency and storage overhead. Therefore, we heuristically separate graph objects into three categories (low, medium, and high) according to their update frequency $f(o)$, and assign proportionally larger anchor intervals for graph objects with higher update frequency. To precisely capture this proportionality, we introduce a parameter $\rho = \tau_2/\tau_1$ with $\rho > 1$ to define anchor intervals in a geometric progression, such that $u_{high} = \rho \cdot u_{medium} = \rho^2 \cdot u_{low}$, where u_{high}, u_{medium}, and u_{low} indicate the anchor interval of graph objects with high, medium, and low update frequencies, respectively. The parameters τ_1, τ_2, and c can be flexibly configured by users based on the specific update frequency of the application and the available storage and CPU resources. We acknowledge that deep learning techniques offer potential for automatic parameter tuning. However, this is orthogonal to our current focus and limited by space, we leave it as future work.

Data migration. In MVCC, unreclaimed historical versions will be physically removed from the current storage through an asynchronous GC phase when their relevant committed transactions (CTs) are no longer active. AeonG collects those versions and migrates them to the historical storage for long-term maintenance, as detailed in Algorithm 1. We use kvs to store reclaimed data in a key-value format (line 2). Each unreclaimed version $undo$ in the CT is encoded into a key-value pair kv and added to the collection kvs (line 4). Subsequently, we store those encoded reclaimed versions kvs in the historical key-value store in batch (line 5). Finally, we physically delete those successfully migrated versions from the current storage (line 6).

5 Temporal Query Engine

AeonG adopts a vertex-centric query model, which is typically employed in graph databases such as Neo4j [44], Memgraph [40], and Kùzu [52]. In this model, subgraphs are retrieved by scanning vertices using the scan operator and then expanding to adjacent vertices and edges through the expand operator. AeonG enhances these two operators to enable consistent and efficient temporal query processing. In this section, we present the indexing mechanism applied to these operators and then describe how these operators work.

5.1 Indexing Mechanism

In a vertex-centric database, indexes are typically constructed on current vertices within the current storage to efficiently locate starting vertices for graph traversal. Conventional indexing structures, such as SkipList, B^+-trees, and LSM trees, are employed to enable rapid retrieval. However, since these indexes only cover current vertices, they cannot directly support temporal queries, which require locating historical versions of starting vertices, as well as efficiently retrieving historical versions of edges and neighboring vertices. To address these limitations, AeonG proposes a dual-index approach that consists of two separate indexes: (1) the vertex locating index, which adheres to standard indexing approaches for locating starting vertices and (2) the version index, designed to optimize lookups for historical versions of vertices, topological structures, and edges.

The vertex locating index is designed for efficient lookup of starting vertices, which includes two categories: current indexing and historical indexing. The current index maps vertex labels, property values, or vertex IDs to vertices, following the standard practice in graph databases [40]. The historical index captures outdated index entries resulting from updates to the current index. Upon each index update, outdated index entries are transferred to the historical index. We implement the vertex locating index based on the standard index structure provided by graph databases, e.g., SkipList. Given a temporal query, we check both the current and historical indexes to identify all candidate seed vertices. In particular, we acquire shard/exclusive locks on related indexes before read/write operations, and only release these locks after the operation completes, to ensure consistency.

The version index includes a hash index and a key-value index. The hash index uses a composite key of $Type$ and Gid (the same elements discussed in Section 4.2), with the value representing the time interval from the start time $\omega.st$ of the oldest version to the end time $\omega.ed$ of the latest version of a graph object. The key-value index leverages the existing indexing mechanism in the key-value store, such as the SkipList in RocksDB. During a temporal query, given a starting vertex obtained from the vertex locating index and a temporal constraint, we first query the hash index to check if historical versions exist, and then search the key-value index to retrieve specific versions.

Example 7 Consider the temporal query illustrated in Example 3. To optimize query performance, we construct a vertex locating index on the “name" property of Customer entities, mapping “name=Jack" to v_0, and build a version index for all historical versions like $v_{2.7}$. We first use the scan operator to fetch the seed vertex version. The vertex locating index identifies v_0 via its current index, as no updates have occurred to the “name" property of v_0. The version index is then used to retrieve relevant versions based on Gid=0 and the temporal constraint t_n. In this case, the historical version lookup terminates at the hash index since no historical versions exist for v_0. Details of the scan operator are provided in Section 5.2. Next, we use the expand operator for graph traversal. We rely on in-graph structures like CSR to fetch $Gids$ of adjacent vertices and edges and then employ the version index to fetch relevant historical versions, such as $v_{2.7}$. Further details about the expand operator are provided in Section 5.3.

5.2 Scan Operator

AeonG uses the scan operator to efficiently fetch vertex versions while ensuring data consistency for both current data and historical data. We elaborate on it from the aspect of fetching data from each storage component. When fetching data from the current storage, it is essential to ensure consistent data capture in the presence of concurrent transactions. To achieve this, we start by locating relevant vertex object(s) of interest. For each vertex object, we first employ the snapshot visibility check [45] to find a visible version of the given transaction. All versions preceding this visible version in the version chain are candidate legal versions we may want. Then we utilize a legal check mechanism, which verifies whether each candidate version v' is legal to the given query time condition, as per the Eq. (2).

$$\omega.st \leq C.t_2 \wedge \omega.ed > C.t_1 \tag{2}$$

Here, $\omega.st$ and $\omega.ed$ represent the start and end time of v''s lifespan, respectively. C represents the time condition of the given query with begin time t_1 and end time t_2. For a time-point query, $t_1 = t_2$.

When fetching data from the historical storage, there is no need to handle transaction conflicts as the historical storage serves read-only queries that users cannot change the data. Therefore, we directly employ the legal check mechanism to get desired versions. To optimize historical version retrieval, we employ an anchor-based skip retrieval strategy to reconstruct desired versions. To restore a specific legal version v', we directly seek the most recent anchor v in the KV store by

the probe prefix "AV:Gid:C", where 'AV' represents anchors in the VP segment, Gid is the vertex identifier, and C is the query time constraint. We then assemble v' by combining v with all subsequent deltas until v'. Due to the specialized key-value format design in historical storage, we can efficiently locate the nearest anchor through the key-value index.

Algorithm 2 shows the pseudo-code of fetching vertices from the hybrid storage engine. We start by scanning from the vertex object v, which is either the first vertex in the graph or the vertex pointed by the vertex locating index. Next, we fetch desired versions matching the query condition C. Initially, we retrieve data from the current storage (lines 4-7). We check whether v and its historical unreclaimed versions are visible to the current transaction (line 5). The function `TemporalCheck()` is employed to assess their visibility based on Eq. (2) (line 6). Next, we fetch historical versions from the historical store using the function `FetchFromKV()` (line 9). This process begins by querying the hash index of the version index to determine if the vertex has any historical versions that satisfy the temporal constraint by checking the value index against the temporal condition C (line 13). We then utilize the Gid and temporal conditions specified in the query as probe keys to fetch specific versions. Specifically, we find the most recent anchor kv_a based on the probe prefix (line 14), seek all previous deltas kv_d that satisfy the temporal check (lines 15-18), and assemble kv_a with them to get desired versions (line 17). Notably, the keys in the key-value store are prefixed with Gid, with the same graph objects clustered together in a SkipList index. This allows for efficient retrieval of historical versions.

Complexity analysis. The scan operator queries versions of a vertex v in a dataset with a total of n vertices. This process consists of two parts: (1) locating the current version of v and (2) querying the historical versions of v. The complexity of locating the current version depends on the specific retrieval mechanisms selected in the current storage, such as $log(n)$ for B$^+$-tree index look-up and n for non-index lookup, denoted as $O(\iota(n))$. The complexity of querying the historical versions depends on the chosen approach for introducing temporal features. In AeonG, we first locate the nearest anchor. Since we organize historical versions in the key-value store using SkipList, the complexity of this process is $O(log(A_v))$, where A_v is the average number of anchors for vertices. Then, we sequentially scan deltas from the anchor until satisfying the query time condition, with a time complexity of $O(u)$, where u represents the anchor interval length defined in Eq. (1). In conclusion, the scan operator has a complexity of $O(\iota(n) + log(A_v) + u)$.

Algorithm 2: Retrieving vertices

```
1  Function VertexRead(Gid, C):
       Input: v ← the vertex of Gid which we start to scan; C,
              temporal condition;
       Output: Σ, the result set;
2      while v do
3          // fetch from the current storage;
4          foreach v' ∈ (v ∪ v.versions) do
5              if !SnapshotCheck(v') then continue;
6              if TemporalCheck(v'.ω, C) then
7                  Σ ← Σ ∪ Reconstruct(v');
8          // fetch from the historical storage;
9          FetchFromKV(Gid, C, Σ);
10         v ← v.next();
11     return Σ;
12 Function FetchFromKV(Gid, C, Σ):
13     if !HasVersions(Gid, C) then return;
14     kv_a ← KV_store::seeknext(Gid, C);
15     kv_d ← KV_store::seeknext(Gid, kv_a.key);
16     while kv_d ∧ kv_d.ω.st ≤ C.t_2 do
17         kv_a ← combine (kv_a, kv_d);
18         if TemporalCheck(kv_d.ω, C) then
19             Σ ← Σ ∪ kv_a;
20         kv_d ← kv_d.next();
```

5.3 Expand Operator

Similar to the scan operator, the expand operator employs different retrieval strategies in two separate storage engines. Additionally, the expand operator considers the retrieval of graph structures. We now detail how the expand operator fetches edge and neighboring vertex versions for a given vertex.

As shown in Algorithm 3, given a vertex v, we first use the function `VERead()` to access the v's adjacency list version(s) from the VE segment (line 2), which contains a list of (edge id e_{Gid}, neighbor vertex id nv_{Gid}) pairs. The function `VERead()` shares a similar logic as the function `VertexRead()` in Algorithm 2, which combines current and historical data to get desired versions. We then fetch legal versions of specific semantic information of edges and adjacency vertices based on their unique $Gids$ (lines 3-8). We first obtain linked edge versions using the `EdgeRead()` function, which follows a similar logic to `VertexRead()` (line 4). To expedite the search for the linked edge version, we optimize the skip look-up strategy for anchor locating to skip more unnecessary versions. As defined by Constraint 1 in Section 2.1, the edge must be legal for its connected vertices. This implies that the lifespan of the edge version must intersect with the lifespan of its connected vertex version. Since we already hold a scanned vertex version v, we use v's lifespan ω to narrow the probe time scope C based on the following equation.

$$f(C, \omega) = [max(C.t_1, \omega.st), min(C.t_2, \omega.ed)] \quad (3)$$

By implementing this approach, we efficiently bypass more unnecessary versions, thereby enhancing the query per-

Algorithm 3: Expanding Vertices

```
1 Function ExpandVertices (v, C):
    Input: v, the graph vertex need to expand; C, temporal
           condition;
    Output: Σ, the result set;
2   Σ_ve ← VERead(v_Gid, C) //get adjacency lists;
3   foreach (e_Gid, nv_Gid) ∈ Σ_ve do
4     Σ_e ← EdgeRead(e_Gid, f(C, v.ω)) ;
5     foreach e ∈ Σ_e do
6       Σ_nv ← VertexRead(nv_Gid, f(C, e.ω));
7       foreach nv ∈ Σ_nv do
8         Σ ← Σ ∪ (e, nv) ;
9   return Σ;
```

formance. Subsequently, we retrieve the neighbor vertex versions of each holding edge version based on nv_{Gid} (lines 5-6) with a similar logical process. Finally, we obtain the final results (lines 7-8).

Complexity analysis. The expand operator retrieves linked edges and adjacent vertices of a specific vertex version in the following three steps. First, we fetch the adjacency list version. Similar to the complexity of the scan operator, the associated complexity is $O(log(A_{ve}) + u)$, where A_{ve} is the total number of anchors for adjacency lists, and u is the average length of u_o. Next, for each pair (e_{id}, nv_{id}) in the adjacency list, we fetch the corresponding edge version. Finally, we locate the neighbor vertex version. Similar to the first step, the complexities of these two steps are $O(log(A_e)+u)$ and $O(log(A_v)+u)$, where A_e and A_v are the total number of anchors for edges and vertices, respectively. In conclusion, the overall complexity of the expand operator is $O(log(A_{ve}) + u + D \times (log(A_e) + u + log(A_v) + u))$, where D is the average number of vertex degrees. We would like to highlight that the complexity of both Algorithm 2 and Algorithm 3 is at the $O(log(n))$ level, which is reasonable and generally acceptable in graph query processing [34, 36, 61].

6 Cloud-native AeonG-C

To further enhance the system's scalability and elasticity for storing and querying continuously growing historical data, we extend AeonG within a cloud-native architecture, referred to as AeonG-C.

Architecture. As shown in Figure 8, AeonG-C is designed as a disaggregated database with two distinct layers: a compute layer and a storage layer, allowing each layer to scale independently. The compute layer comprises multiple nodes, each hosting compute-intensive components of AeonG. The storage layer, inherited from AeonG's hybrid storage, is a shared storage system, which is accessible by every compute node through the network. This disaggregated design adheres to the architecture of modern cloud-native databases [12, 58]. Thus, we can offer virtually unlimited resources to users, with the ability to auto-scale up or down based on varying workloads and scales of temporal data.

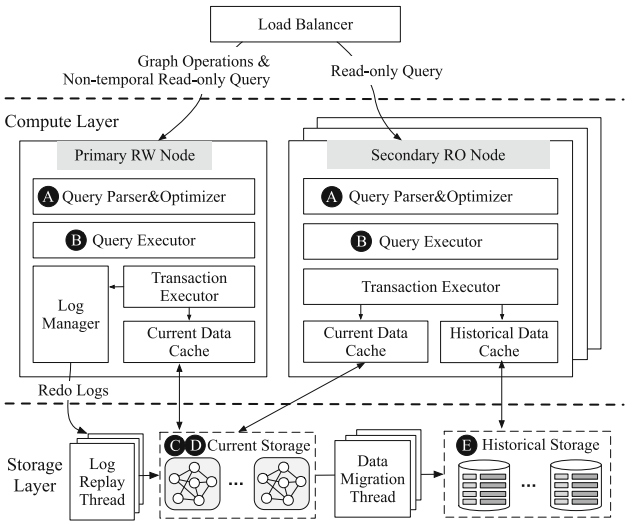


Fig. 8 AeonG-C architecture

In the compute layer, there is a single read-write (RW) node and multiple read-only (RO) nodes. Queries sent by clients are first forwarded through a load balancer to the appropriate nodes. The RW node processes graph operations and non-temporal read-only queries, whereas RO nodes manage all read-only queries, encompassing both non-temporal and temporal queries, as elaborated in Section 2.2. Note that temporal queries are exclusively assigned to RO nodes to alleviate the workload of the RW node, which needs to handle complex graph operations. Both RW and RO nodes are equipped with a **query parser**, **optimizer**, **executor**, and **transaction executor** derived from AeonG. Additionally, to better accommodate the disaggregate architecture, AeonG-C introduces data caches and a log manager. **Data caches**, present in both RW and RO nodes, interact with the storage layer to temporarily store hot data, thereby reducing the network overhead to offer fast data access. The **log manager**, however, is exclusive to the RW node. It is responsible for handling redo processing caused by graph operations, which are then offloaded to the storage tier for persistent storage.

The storage layer of AeonG-C adheres to the design principles of AeonG, incorporating two physically isolated shared storage systems: current storage and historical storage. To further align with the disaggregated architecture, AeonG-C introduces additional **log replay threads** and **data migration threads**, which are responsible for replaying the temporal graph data.

Discussion. AeonG-C follows established cloud-native principles, such as offloading redo logging in the storage layer

for data replay and caching hot data in the compute layer for fast access. However, due to the complexity of storing and accessing historical data, integrating these techniques for temporal graph data management introduces challenges. First, the replay mechanism must handle not only current data, but also additional historical data. To solve this, we first design redo logs to record graph evolution in the RW node and then propose two asynchronous replay threads for replaying temporal data in the storage layer. Redo log generation and replay are detailed in Section 6.1 and 6.2.1, respectively. Second, given the additional interactions between the compute and storage layers, efficiently processing temporal queries while ensuring data consistency is essential. Therefore, we propose an asynchronous prefetching approach to preload the most recent current data and their corresponding historical versions into the data caches of RO nodes, as discussed in Section 6.2.2. Further, we propose a two-phase fetching strategy to ensure consistency between historical data caches and remote storage, as detailed in Section 6.3.

6.1 Graph Operation Processing

We now detail how AeonG-C processes graph operations to generate redo logs that capture graph evolution within the RW node. Graph operation processing follows the MVCC mechanism to ensure ACID properties. Rather than decomposing the relevant transaction into subtransactions for manipulating data from remote storage nodes, AeonG-C directly accesses data from the data cache. By doing so, we eliminate the costly concurrency control required by distributed transactions, thereby enhancing transaction throughput [24, 37, 57]. If the required data does not reside in the data cache, we then load it from the remote cloud's current storage through the network. Note that only the current storage is accessed, as manipulating graph data affects only the most recent data. In particular, a transaction T generates redo logs to record the modified graph data during graph operations, following three phases:

- Start phase: We start a transaction T for graph operations and assign it a unique transaction ID (TID).
- Read/Write phase: We perform read and write operations on graph data by accessing the data cache. Here, we generate redo deltas, which adhere to the modification paradigm discussed in Section 4.1.
- Commit phase: We first obtain a commit timestamp (cts) of T from a centralized timestamp sequence, which allocates a monotonically increasing timestamp. We then generate the **redo log** to record the data modifications made by the transaction T. Specifically, the redo log includes TID, metadata graph identifiers (Gid), cts, and detailed redo deltas modified by T. Lastly, we transfer the redo log to the storage layer via the log manager.

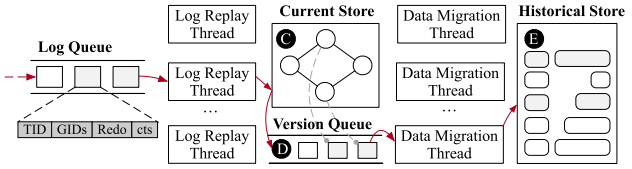


Fig. 9 AeonG-C Storage Layer

6.2 Disaggregated Storage

In this section, we first elaborate on the replay mechanism for replaying temporal graph data within the storage layer. Next, we present the data cache mechanism in the compute layer, highlighting an asynchronous prefetching approach to improve the efficiency of temporal queries.

6.2.1 Redo Log Replay

AeonG-C introduces two asynchronous replay threads within the storage layer, i.e., log replay threads and data migration threads, to conduct redo log replay transmitted from the RW node. As shown in Figure 9, upon receiving a redo log in the storage layer, it is first stored in a log queue. The **log replay threads** continuously fetch logs from the queue asynchronously, replay current data, and write them to the shared current storage. During this process, updating or deleting a data item may generate a historical version, as discussed in Section 4.1. AeonG-C collects these historical versions in a version queue. The **data migration threads** then asynchronously fetch historical versions from the queue and migrate them to the shared historical storage.

Algorithm 4 shows the pseudo-code for functions related to replaying temporal graph data, including the `LogReplay()` function called by log replay threads and the `DataMigration()` function called by data migration threads. In the `LogReplay()` function, we continuously fetch a redo log of a transaction T from the log queue (line 2). For each graph data item identified by its Gid, we replay the log information and update the relevant graph data item (line 5). Similar to the undo chain discussed in Section 4.1, AeonG-C records the redo log via redo chains in a log file. During replay, actions are redone by traversing the redo chain from past to future. In addition to directly updating the new version, we generate a historical version to record the steps required to revert the changes, and we store it in the version queue (line 7). In the `DataMigration()` function, we continuously fetch batches of k versions from the version queue (lines 10-11). Each version is encoded into a key-value pair and added to the collection kvs (line 12). We then batch-write these encoded historical versions into the historical KV store (line 13) and delete them from the version queue (line 14).

6.2.2 Data Cache

AeonG-C adopts a cloud-native data cache mechanism, where the cache interacts with the storage layer to temporarily store frequently accessed data. Given the distinct query workloads of the RW and RO nodes, our caching strategy is tailored accordingly: the RW node caches only current data, while the RO nodes cache both current and historical data. Both caches utilize the Least Recently Used (LRU) replacement policy.

We propose an asynchronous prefetching approach to enhance temporal query efficiency on RO nodes. Specifically, we preload the most recent current data and their corresponding historical versions into the data cache. To achieve this, we first combine the metadata of redo logs updated by multiple transactions into a batch within the RW node and consolidate them into a transaction status table TST, which includes TID, cts, and Gid. Subsequently, we broadcast the TST to all RO nodes. Upon receiving the TST, the RO nodes asynchronously update graph data items in the data cache based on the information provided. We cache the most recently updated current data into the current data cache and its historical versions within a recent k-minute time window into the historical data cache. For the current data cache, we employ classic indexing mechanisms. For the historical data cache, we design an in-memory hash index, where the key is Gid and the value is a list of tuples, each containing a disjoint, continuous time interval and the associated historical version information. Our proposed strategy enhances temporal query efficiency, as detailed in the following subsection. This approach is motivated by two key observations. First, temporal queries in practical scenarios often focus on relatively recent data, such as data modified within the last five minutes or half an hour [31]. Asynchronous updates balance data freshness and performance, as temporal queries prioritize consistency over accessing the most latest data. Second, caching recent versions accelerates visibility checks for transactions accessing the appropriate data version.

6.3 Read-only Queries

AeonG-C utilizes RO nodes to handle read-only queries, including both non-temporal and temporal queries. Similar to AeonG, AeonG-C retrieves current and historical data (for temporal queries) separately and combines the results as needed. Data is first fetched from the data cache; if unavailable, it is retrieved from remote storage. However, unlike AeonG, AeonG-C requires address version consistency between the data caches on RO nodes and the remote storage.

In a cloud-native architecture, it is essential to ensure that the retrieved versions of graph objects are consistent between the disaggregated data caches and the remote storage layer.

Algorithm 4: Replay Temporal Graph

```
1  Function LogReplay():
2      fetch a redo log of T from the log queue;
3      foreach Gid ∈ T.Gids do
4          //update new version;
5          UpdateVersion(Gid, T.redo, T.cts);
6          //copy historical version to version queue;
7          CopyVersion(Gid, T.cts);
8  Function DataMigration():
9      kvs ← ∅;
10     while k do
11         copy a version from the version queue;
12         kvs ←encode2KV(version);
13     KV_store::putMultiple(kvs);
14     delete k versions from the version queue;
15 End Function
```

For instance, the data cache may store multiple versions of a graph object v_3, such as $v_{3.5}$ in the current cache and $v_{3.3}$ in the historical cache. This is feasible because the current and historical data caches are managed independently. When processing a temporal query seeking versions $v_{3.5}$, $v_{3.4}$, and $v_{3.3}$, even though the data caches hold $v_{3.5}$ and $v_{3.3}$ that correspond to the relevant time conditions, the result would be incorrect because the retrieved versions are not contiguous as it lacks $v_{3.4}$. This inconsistency arises because the retrieved versions are not contiguous. In disaggregated architecture, data consistency requires the continuity of the version timeline in the data cache.

To address this, we propose a two-phase fetching strategy. As previously outlined, our data cache design stores the current version of a graph object and its historical versions within a k-minute window. This ensures that the current version and its corresponding historical versions are contiguous, while the continuity of other historical versions requires additional verification. Accordingly, we divide the fetching strategy into two phases based on time conditions: one retrieves continuous versions in the data cache, and the other accesses versions from the historical cache, where continuity is not guaranteed. For the former, we can proceed according to Algorithm 2 and 3. For the latter, we utilize the historical hash index, as discussed in Section 6.2.2, to verify whether the historical data cache contains all versions that satisfy the specified time conditions. If so, the data is returned; otherwise, missing versions are fetched directly from remote storage.

Algorithm 5 outlines the steps for fetching versions from the disaggregated storage. For a transaction T fetching a graph object g identified by Gid, we hold a start timestamp st of T. Initially, we get the start lifespan $\omega.st$ of the most recent version of g from the data cache (line 1). If $\omega.st$ is obtained, it indicates that g exists in the data cache, prompting us to set the flag f_c to true and attempt data retrieval from the data cache (lines 2-6). Recall we employ a two-

Algorithm 5: Processing read-only queries

Input: Gid, the objective graph identifier; st, start timestamp; T, read-write transaction
Output: Σ, the result set;

```
1 ω.st ⟵ GetFromCurrentCache(Gid, f_c);
2 if f_c then
3     //read continuous versions;
4     Σ ← Σ ∪ ItemReadFromCache(Gid, T.b);
5     //read cached historical versions;
6     Σ ← Σ ∪ FindHistoricalCache(Gid, f_h, T.a);
7 //read from remote storage;
8 if !f_c or !f_h then
9     Σ ← Σ ∪ FetchRemoteStorage(Gid, T) ;
```

phase fetching strategy based on two distinct time windows. We divide the temporal condition C into two time windows: a lower non-continuous time window a and an upper continuous time window b, as defined in Eq. (4).

$$T(C', \varphi) = \begin{cases} a : C', b : \emptyset & \varphi > C'.t_2 \\ a : \emptyset, b : C' & \varphi < C'.t_1 \\ a : [C'.t_1, \varphi], b : [\varphi, C'.t_2] & C'.t_1 \le \varphi \le C'.t_2 \end{cases} \tag{4}$$

Here, C' represents the rewritten temporal condition C, defined as $C' = [C.t_1, max(C.t_1, min(C.t_2, st))]$. To simplify the description, we unify the upper boundary to ensure that the retrieved versions comply with the snapshot visibility check, meaning their timeline must precede the start time st of T. We define φ as $\omega.st - k$, marking the boundary for reliable continuous versions in the data cache. We retrieve the versions from the data cache that satisfy the continuous time window $T.b$ (line 4). This processing follows the retrieval mechanisms outlined in Algorithm 2 and 3, as the timeline of versions within this time window is continuous. Next, we seek the versions from the historical data cache that satisfy $T.a$ (line 6). Given that the timeline of versions in this time window is non-continuous, we must verify whether the historical data cache contains all versions required by $T.a$. To solve it, we utilize the in-memory hash index maintained in the historical data cache (as discussed in Section 6.2.2) to check if $T.a$ is fully covered by all cached versions. If it is, we return the results directly; otherwise, we set the flag f_h to false, indicating the necessity to retrieve versions from remote storage. Lastly, if either f_c or f_h is false, we must fetch the desired data from remote storage (line 9).

7 Implementation

In this section, we present the implementation of the standard AeonG, the distributed AeonG called AeonG-D, and the cloud-native AeonG called AeonG-C. Our system is written in C++, and is publicly available at [2].

7.1 Implementation of AeonG

AeonG is built on Memgraph [40] and RocksDB [18]. Memgraph is a commercial native graph database that supports the property graph model, Cypher, and MVCC. We extend Memgraph to serve as the primary database engine, providing the basic query engine and current storage engine for AeonG. We then integrate RocksDB, a popular KV store, as the historical storage to manage historical data. Our approach is generally applicable to native graph databases that support MVCC.

Query engine. AeonG extends the *parser* to recognize temporal queries defined in Section 2.2, incorporating the temporal qualifier into `Cypher.g4` and enhancing `CypherMainVisitor()` to recognize temporal qualifier. Furthermore, AeonG enhances the *executor* by modifying two fundamental operators: scan and expand operators. In the `ScanAllCursor.Pull()` function, we introduce a function `AddHistoricalVertices()` to capture both unreclaimed and reclaimed historical versions (Algorithm 2). Similarly, in the `ExpandCursor.Pull()` function, we add `AddHistoricalEdges()` to retrieve historical edges and neighboring vertices (Algorithm 3).

Storage engine. The storage engine is hybrid, consisting of a *current* storage and a *historical* storage. The *current* storage is derived from Memgraph's storage, where the `Vertex` structure maps to the vertex object, the `EdgeRef` structure maps to the edge object, and the `Delta` structure represents historical unreclaimed versions. We then associate the time dimension with those structures to introduce temporal support, as discussed in Section 4.1. Timestamps are assigned by a global clock when relevant transactions are committed, with a time granularity of milliseconds. Moreover, AeonG supports all indexes provided by Memgraph, such as vertex label indexes and property value indexes. We integrate RocksDB into Memgraph as the *historical* store by starting a RocksDB process when the Memgraph instance starts. We introduce a function, `Migrate()`, within the `CollectGarbage()` function of Memgraph, designed to transfer unreclaimed data into key-value pairs and then migrate them to RocksDB (Algorithm 1). We introduce a system parameter, `retention_period`, to set the historical data retention period. For example, setting `retention_period` to one month enables the periodic removal of historical versions whose end lifespan $\omega.ed$ is earlier than one month ago. Given that historical versions have non-overlapping and continuous lifespans, we can safely truncate all earlier versions, and the remaining versions still maintain continuous lifespans.

We also implement a distributed version of AeonG, denoted as AeonG-D. In AeonG-D, we develop the historical storage based on TiKV [25], an efficient distributed key-value store. We extend the historical storage to a distributed backend by replacing the RocksDB interfaces with TiKV, which enables AeonG-D to handle larger volumes of historical data. AeonG-D inherits TiKV's partitioning strategy, which partitions data based on continuous key ranges. Data is horizontally partitioned into multiple regions, each representing a range of consecutive keys. This strategy aligns with the "key" design in our historical storage, ensuring that different versions of the same entity are grouped within the same region. We implement the `FetchFromKV` function asynchronously, allowing the scan and expand operators to issue concurrent fetch requests to TiKV for different entries, thereby enabling parallel data retrieval. By the way, this alternative implementation demonstrates the broad applicability of our approach.

7.2 Implementation of AeonG-C

We now present the implementation of AeonG-C, which extends AeonG by adopting a disaggregated architecture that features distinct compute and storage layers.

Compute Layer. Each compute node runs multiple threads of two types: worker and logger. Worker threads, managed by the Memgraph instance, are responsible for executing queries. Logger threads, newly introduced to accommodate the disaggregated architecture, manage additional tasks. In the RW node, a logger thread transfers redo logs to the storage layer. In the RO node, a logger thread updates the data cache with temporal graph data retrieved from the storage layer.

Given that Memgraph is an in-memory database, we utilize its storage engine to serve as the current data cache. For the historical data cache, we introduce the `HistoricalVertex` and `HistoricalEdge` structures to maintain historical versions. We employ the LRU policy for cache replacement. When requested data is not found in the cache, the `LoadData()` function is invoked to retrieve the data from the storage layer. Furthermore, the RW node uses the `AsyncTST()` function to broadcast the transaction status table TST to all RO nodes. RO nodes asynchronously update relevant graph data items based on the received TST.

Storage Layer. The storage node is equipped with two RocksDB instances to preserve temporal graph data and provide data access services. The first instance, `RocksDB@current`, manages the current data while the second, `RocksDB@historical`, stores the historical data. We use RocksDB to persist current graph data in a key-value format, similar to the layout of historical data, with the key distinction that all anchor versions are preserved. We implement an individual log replay component that replays redo logs produced by the RW node into the `RocksDB@current` instance and a data migration component to transfer historical versions to the `RocksDB@historical` instance as outlined in Section 6.2.1. We use two interfaces, `Get()` and `Put()`, to make the storage services transparent to compute nodes, log replay threads, and data migration threads. This abstraction ensures that AeonG-C can operate transparently across various cloud storage services, such as Microsoft Azure [55], Amazon S3 [51], and others. For example, users can deploy AeonG-C on the cloud by substituting the RocksDB with other cloud-native key-value stores, such as RocksDB-Cloud [49] or Slatedb [53] by simply replacing the `Get()` and `Put()` interfaces. Communication between the compute and storage layers is efficiently managed using a bRPC [10] proxy.

8 Correctness and Fault Tolerance

In this section, we first present the proof of correctness for data consistency and then analyze consistency under fault-tolerant conditions.

8.1 Proof of Correctness

We first provide the formal definition of consistency, followed by a detailed analysis of general consistency in AeonG. Finally, we discuss the consistency mechanisms in the extended systems, AeonG-C and AeonG-D.

8.1.1 Consistency Definition

Consistency ensures that the database remains in a valid and correct state at any point in time. In AeonG, we enhance the classic MVCC mechanism to maintain consistency for temporal graph data management. Formally, consistency is defined as:

$$\forall T, \forall g \in G, \exists S \text{ such that } T(g,t) \equiv S \tag{5}$$

Here, T is a transaction, g is a graph object, G represents the graph, and S is a valid database snapshot. This definition ensures that any transaction T operating g at any time t, whether a read or write, produces results consistent with a valid snapshot S, maintaining database correctness.

8.1.2 General Consistency of AeonG

We now detail the write and read consistency of AeonG. For write consistency, we need to address the integrity of generating graph object versions under the modification paradigm outlined in Section 4.1. For read consistency, we address the

retrieval of consistent graph object versions from the hybrid storage system.

Theorem 1 *(Write Consistency) For every time point during the lifespan of a graph object (from its creation to its deletion), a transaction T generates exactly one unique version.*

Proof In the MVCC protocol, each transaction T is assigned a start timestamp $T.st$ and a commit timestamp $T.ct$. A transaction T commits successfully only if no other transaction with a commit timestamp in $[T.st, T.ct]$ writes to data that T also writes. Otherwise, T aborts. Upon successful commit, T generates a new version $g.o$ with a tentative lifespan $g.o.\omega = [T.ct, +\infty]$. Suppose two transactions T_i and T_{i+1} concurrently update g and commit sequentially, such that $T_{i+1}.ct > T_i.ct$ (or vice versa, $T_i.ct > T_{i+1}.ct$). When T_i commits, it creates $g.o_i$ with $g.o_i.\omega = [T_i.ct, +\infty]$. When T_{i+1} subsequently commits, it updates $g.o_i.\omega$ to $[T_i.ct, T_{i+1}.ct]$ and then creates $g.o_{i+1}$ with $g.o_{i+1}.\omega = [T_{i+1}.ct, +\infty]$. Since MVCC enforces the commit order of transactions, the lifespans of $g.o_i$ and $g.o_{i+1}$ are chronological and non-overlapping, ensuring $g.o_i.\omega.st < g.o_i.\omega.ed = g.o_{i+1}.\omega.st < g.o_{i+1}.\omega.ed$. For a sequence of committed transactions $\{T_1, T_2, ..., T_i, T_{i+1}\}$ generating versions $\{g.o_1, g.o_2, ..., g.o_i, g.o_{i+1}\}$, the commit order $T_1.ct < T_2.ct < ...T_i.ct < T_{i+1}.ct$ guarantees that version lifespans are consecutive and non-overlapping, i.e., $g.o_1.\omega.st < g.o_1.\omega.ed = ... = g.o_{i+1}.\omega.st < g.o_{i+1}.\omega.ed$. Thus, at any time point t, T generates exactly one unique version of a graph object. □

Theorem 2 *(Read Consistency) Given a read-only transaction T fetching temporal graph data with a condition time C, T can be verified to have retrieved the data from the hybrid storage with a consistent state.*

Proof For a read-only transaction T, graph object versions are retrieved from a unified global snapshot based on reliable lifespan $g.o.\omega$. For a version $g.o$ in the current storage, we draw support for the MVCC's snapshot visibility check. Specifically, when a read-write transaction T_w commits, its changes become visible to all transactions whose start timestamps are larger than T_w's commit timestamp. Given that $T_w.ct = g.o_w.\omega.st$, we can deduce that versions of a graph object $\{g.o_1, g.o_2, ..., g.o_i, g.o_{i+1}\}$ become visible to T if $g.o.\omega.st < T.st$ (line 5, Algorithm 2). After identifying all visible versions $g.o$, we utilize their lifespans ω to align C to determine the required versions (line 6, Algorithm 2). For a version $g.o$ in the historical storage, no concurrency control is required because these versions are immutable. Thus, we simply use a legal check mechanism that exclusively aligns $g.o.\omega$ to C (line 18, Algorithm 2). By doing so, we ensure data consistency within each storage layer, thereby maintaining the correctness of hybrid storage. □

8.1.3 Consistency in Extended Architectures.

The extended systems, AeonG and AeonG-C, inherit the core consistency guarantees of AeonG while incorporating additional distributed consistency mechanisms tailored to their respective architectures.

In AeonG-C, we adopt a disaggregated architecture comprising a shared hybrid storage layer and a compute layer with one RW node and multiple RO nodes. To ensure write consistency, we maintain the guarantees by centralizing all write operations in a single RW node, thus eliminating the need for distributed transaction management across multiple compute nodes. For read consistency, it is essential to verify the consistency between the disaggregated data caches and the remote storage layer. To address this, we propose a two-phase fetching verification strategy, as detailed in Section 6.3.

In AeonG-D, we replace the local historical storage of AeonG with the distributed key-value store TiKV. In this context, we must consider both write and read consistency for historical versions in a distributed environment. To do so, we leverage TiKV's Raft consensus protocol [47], which provides strong consistency across the distributed storage cluster.

8.2 Fault Tolerance

We adopt standard fault tolerance mechanisms [43, 59] in AeonG, AeonG-D and AeonG-C. A key distinction in our temporal systems is the inclusion of an additional migration process (Algorithms 1 and 4) to transfer unreclaimed versions from current to historical storage. We categorize fault scenarios into three types: failures in current storage, historical storage, or both. Table 1 outlines the fault-tolerance mechanisms for these scenarios. Below, we analyze these scenarios in detail and provide a proof sketch of the system's fault tolerance.

- The historical storage fails before/after writing historical versions: We recover the system by restarting the historical storage. Whether using RocksDB or TiKV, the historical storage ensures fault tolerance via mechanisms like write-ahead logging (WAL) [58], enabling the migration process to resume.
- The historical storage fails during version writing: In this case, the hybrid storage may enter an inconsistent state, with only partial versions written. The system is recovered by restarting the historical storage and rewriting the incomplete batch of versions. Given each version's unique key, overwriting ensures consistency. It is possible for a version to be duplicated in both current and

Table 1 Fault-tolerance Mechanisms

Category	Failure Time Point	Strategy
Failures in the historical storage	Before writing historical versions	Restart historical storage
	After writing historical versions	
	During writing historical versions	Restart historical storage & Overwrite & deduplication
Failures in the current storage	During migration process	Restart current and historical storage & Redo migration process & deduplication
Failures in the current and historical storage		

historical storage. To solve it, we incorporate a deduplication step retaining the versions from the current storage.
– The current storage or both storages fail during the migration process: We recover the system by restarting both current and historical storages and redoing the entire migration process. It is difficult to pinpoint the exact stage of the migration process when the current storage fails. Therefore, we opt to redo the entire migration process. Deduplication is also applied to handle potential duplicates.

9 Evaluation

In this section, we first outline the experimental setup. We then compare AeonG against two state-of-the-art temporal systems, Clock-G [39] and T-GQL [14]. Additionally, we present a comprehensive performance analysis of AeonG. Finally, we discuss the performance of AeonG-D and AeonG-C.

9.1 Experimental Setup

AeonG and its extended versions, AeonG-D and AeonG-C, are built on: Memgraph $v2.2.0$, RocksDB $v6.14.6$, and TiKV $v7.1.2$ for evaluation. We run the experiments in a cluster of up to 5 nodes. Each node is equipped with 32 Intel(R) Xeon(R) Gold 5220 CPU@2.20GHz, 128GB memory, running CentOS 7.9.

9.1.1 Baseline Systems

Given that existing systems are primarily designed as single-node architectures, we compare AeonG with two baseline single-node systems with temporal support:
T-GQL [14]: A state-of-the-art graph database that assigns a time period to each graph object (vertex or edge) for temporal support. Since we cannot obtain the source code of T-GQL, for fair comparisons, we implement T-GQL based on Memgraph. Note that T-GQL stores all the vertices and edges in memory.
Clock-G [39]: A state-of-the-art graph storage engine that manages temporal data by periodically creating snapshots of the entire database. Since Clock-G is not open-sourced, we implement its temporal data management approach into our codebase for fair comparisons. We record both the snapshots and the logs between successive snapshots in RocksDB. We further introduce the query engine like that used in AeonG to support temporal queries. By default, Clock-G creates a snapshot after executing 80K graph operations.

9.1.2 Workloads.

As detailed in Table 2, we conduct our experiments with three temporal-enhanced workloads:
T-mgBench is based on the real-world Pokec [56] dataset, which is used in Memgraph's mgBench [41] workload. As outlined in Table 3, T-mgBench includes four temporal queries by extending the non-temporal queries in mgBench with a temporal dimension. Specifically, we add "FOR TT AS OF t" to Q1 and Q3, forming "time-point" queries, and add "FOR TT FROM t_1 to t_2" to Q2 and Q4, forming "time-slice" queries. By default, we use a query mix with a ratio of 1:1:1:1 for Q1–Q4.
T-LDBC derives from LDBC [26], a well-known synthetic graph workload, by incorporating temporal syntax into the LDBC IS and IC queries.
T-gMark is based on gMark [8], a well-known synthetic graph workload. It consists of four datasets as shown in Table 2. We use gMark's query generation tool to create non-temporal queries, and then transform them into temporal queries by adding the time condition "FOR TT AS OF t". The query generation follows gMark's default configuration.

To effectively evaluate the efficiency of temporal features, for each workload, we generate additional historical data before evaluations. Unless otherwise specified, we first use the data generation tools from the original workload to create the initial dataset, and then execute graph operations

Table 2 Workload Characteristics

	T-mgBench	T-LDBC	T-gMark			
			Bib	WD	LSN	SP
# of Vertices	10K	3,181K	100K	103K	100K	100K
# of Edges	122K	17,256K	121K	93K	200K	385K
Density	12.17	5.42	1.2	0.90	2	3.85
# of Vertex Labels	1	8	5	24	15	7
# of Edge Labels	1	25	4	82	27	7
# of Graph Operations for Data Generation	320K	1M	320K	320K	320K	320K

with a mix of 80% updates, 10% creates, and 10% deletes to generate historical data. The access distribution of update operations and queries follows the Zipf [62] distribution to simulate real-world graph manipulation scenarios.

9.1.3 Default Configuration

By default, the Zipf distribution factor is set to 1.1. The parameters of the adaptive anchoring approach defined in Eq. (1) are configured as $\tau_1 = 1k$, $\tau_2 = 10k$, and $c = 1\%$. The `retention_period` setting is disabled, ensuring that all historical data is retained permanently.

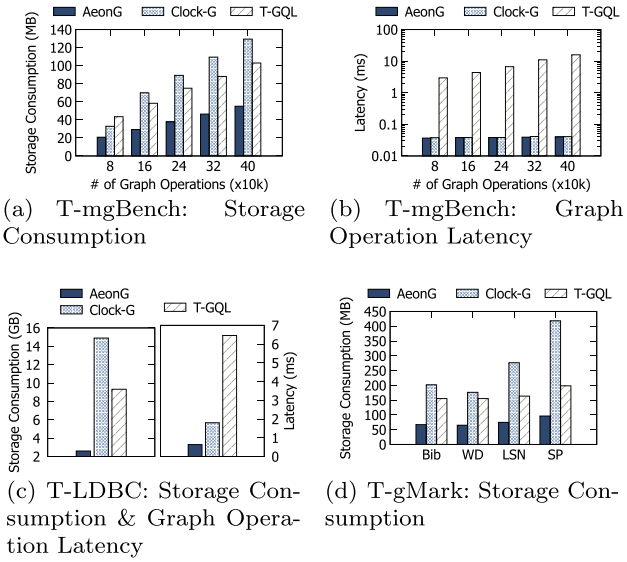
9.2 AeonG vs Baseline Systems

We now compare AeonG with two baseline systems, Clock-G and T-GQL. Since T-GQL is an in-memory database, to ensure fair comparisons, we make sure all data is cached in memory for AeonG and Clock-G by configuring RocksDB's MemTable size to 640 MB.

9.2.1 Experiments on the storage consumption

We first evaluate storage consumption using T-mgBench, plotting the results under varying numbers of graph operations in Figure 10(a). As observed, AeonG reduces the storage consumption by up to 2.4× compared to Clock-G and 2.09× compared to T-GQL as the number of graph operations increases. This efficiency stems from AeonG's "anchor+delta" strategy, which compactly stores most historical graph data as deltas, significantly minimizing the storage overhead for temporal data. In contrast, T-GQL's graph model lacks compact storage for historical data, and Clock-G periodically creates full historical snapshots of the graph. Both systems incur higher storage overheads than AeonG.

We also run experiments using T-LDBC and T-gMark to evaluate the storage overhead. As shown in Figure 10(c), AeonG exhibits up to 5.73× and 3.59× lower storage consumption compared to Clock-G and T-GQL, under T-LDBC. Further, as observed in Figure 10(d), the storage consumption of AeonG is lower than that of Clock-G and T-GQL by



(a) T-mgBench: Storage Consumption

(b) T-mgBench: Graph Operation Latency

(c) T-LDBC: Storage Consumption & Graph Operation Latency

(d) T-gMark: Storage Consumption

Fig. 10 Comparisons on Storage Consumption and Graph Operation Latency

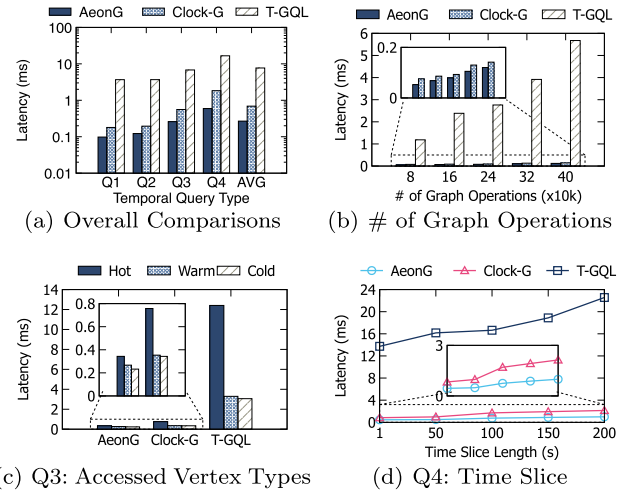
up to 4.34× and 2.39×, under T-gMark. This trend is consistent with that observed in Figure 10(a), demonstrating that AeonG still achieves lower storage overhead when handling large and complex graph workloads.

9.2.2 Experiments on graph operation latency

We then evaluate the graph operation latency with varying numbers of graph operations using T-mgBench. As shown in Figure 10(b), AeonG performs similarly to Clock-G, but significantly outperforms T-GQL by up to 397.06×. As the number of graph operations grows, both AeonG and Clock-G exhibit a performance degradation of 5.4× from 80k to 400k, whereas T-GQL shows a much larger degradation of 34.95×. The performance difference is mainly because T-GQL does not separate the storage of current and historical data. Therefore, graph operations, such as updates, require traversing through a larger number of graph objects (both current and historical data) to reach the specific graph object for updating. In contrast, AeonG separates current and his-

Table 3 Temporal Queries of T-mgBench

Query	Statement
Q1	Match (n: User {id: *$id*}) FOR TT AS OF t RETURN n
Q2	Match (n: User {id: *$id*}) FOR TT From t_1 to t_2 RETURN n
Q3	Match (n: User {id: *$id*})-[e]->(m) FOR TT AS OF t RETURN n,e,m
Q4	Match (n: User {id: *$id*})-[e]->(m) FOR TT From t_1 to t_2 RETURN n,e,m



(a) Overall Comparisons (b) # of Graph Operations (c) Q3: Accessed Vertex Types (d) Q4: Time Slice

Fig. 11 Temporal Query Analysis on T-mgBench

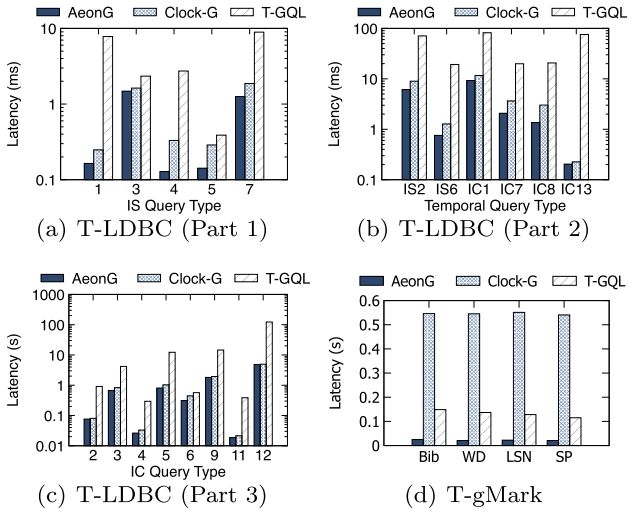
torical data, leading to much smaller latency overheads. The similar performance of AeonG and Clock-G in T-mgBench can be explained as the overhead from snapshot creation is not significant when handling a relatively small size graph. Therefore, to further evaluate storage operation latency with larger graphs, we conducted additional experiments using the T-LDBC workload. As observed in Figure10(c), the graph operation latency of AeonG is lower than Clock-G and T-GQL by up to 2.82× and 10.11×, respectively. As T-LDBC is more substantial, Clock-G requires extra CPU and IO resources to periodically create large historical snapshots, which can negatively affect graph operation performance due to resource contention.

9.2.3 Experiments on temporal query latency

We now analyze the performance of temporal queries under various configurations. We first conduct performance evaluations across various temporal queries in T-mgBench, and plot the query latency on different temporal queries in Figure 11(a). By default, we set the time slice length for Q2 and Q4 to 100s. We can observe that AeonG reduces the query latency by 2.57× compared to Clock-G and 37.57× compared to T-GQL. The superior performance of AeonG can be attributed to our built-in query engine, which employs an efficient anchor-based version retrieval technique to avoid unnecessary version traversal. In contrast, to access desired historical elements, T-GQL necessitates traversing the entire graph, while Clock-G requires fetching the corresponding historical snapshot and appending logs on it, thereby resulting in slower performance.

We then study the latency of temporal query Q1 with the varying number of graph operations. Figure 11(b) shows that AeonG outperforms Clock-G by up to 1.43× and achieves an up to 47.18× improvement compared to T-GQL. This performance gap becomes increasingly pronounced with a growing number of graph operations. As discussed, AeonG exhibits better performance due to the proposed optimized temporal query engine. We further evaluate the query latency of Q3 by classifying queries based on the update frequency of the vertices they access. Specifically, we classify all vertices into "cold", "warm", and "hot" categories based on their access probability under a Zipf distribution (33%, 66%, and 100% accordingly). Queries are then labeled based on the category of the vertices they access. As shown in Figure 11(c), AeonG outperforms the next-best system, Clock-G, in all query categories by up to 2.21×. We can also observe that "hot" queries, which require accessing more historical data, generally have lower performance than "warm" and "cold" queries. However, in AeonG, "hot" queries are only 1.29× slower than "warm" queries, while in T-GQL and Clock-G, "hot" queries underperform "warm" queries by up to 2.14× and 3.77×, respectively. As discussed, the smaller performance gap of AeonG can be attributed to our anchor-based version retrieval technique, which avoids unnecessary version traversal. We also study the latency of temporal query Q4 with varying time slice length from 1s to 200s. As observed in Figure 11(d), AeonG outperforms Clock-G and T-GQL by up to 2.27× and 33.23×, respectively, showing a consistently superior performance under different time slice lengths.

We additionally evaluate the temporal query performance on T-LDBC. As depicted in Figure 12(a), 12(b) and 12(c), AeonG outperforms among all temporal query types, achieving lower latency. It achieves lower latency by up to 1.68× and 25.46× compared to Clock-G and T-GQL for IS queries, and up to 2.3× and 336× for IC queries. These results align with the trends observed in Figure 11(a). T-GQL exhibits poorer performance on IC queries due to their increased complexity, which necessitates additional time to traverse and process subgraphs within large temporal graphs.



(a) T-LDBC (Part 1) (b) T-LDBC (Part 2)

(c) T-LDBC (Part 3) (d) T-gMark

Fig. 12 Temporal Query Analysis on T-LDBC and T-gMark

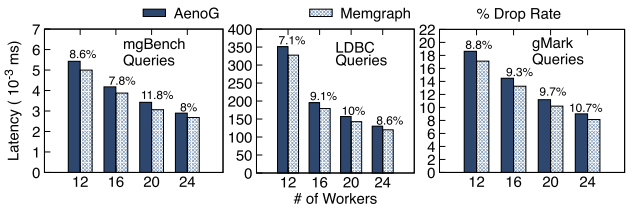


Fig. 13 AeonG vs Memgraph on Non-temporal Queries

We also conduct experiments using T-gMark. As shown in Figure 12(d), AeonG consistently outperforms all datasets, achieving up to 26.16× speedup over Clock-G and 6.56× over T-GQL. T-gMark involves extensive structural traversals over large volumes of temporal data, which imposes substantial overhead on Clock-G due to its need to retrieve and reconstruct multiple snapshots. In contrast, AeonG accesses only relevant data from the current snapshot. T-GQL maintains both current and historical states within a single graph, avoiding costly reconstruction. Conversely, under T-mgBench and T-LDBC workloads, where the graph is large but temporal access is limited, Clock-G outperforms T-GQL. T-GQL incurs higher traversal overhead when the graph size increases, whereas Clock-G can retrieve required snapshots without extensive graph traversal, resulting in superior performance.

9.3 Performance Analysis on AeonG

We now provide an in-depth analysis of the performance of AeonG under diverse configurations. In the following experiments, we fix RocksDB's MemTable size to the default value of 64 MB.

9.3.1 The performance of non-temporal queries

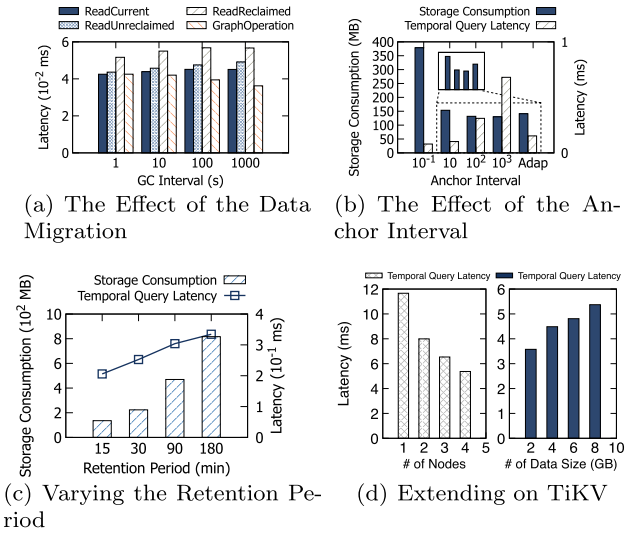
We first analyze the performance of AeonG on non-temporal queries to study the impact of introducing temporal features in its fundamental system, Memgraph. We utilize various non-temporal queries defined within three original unextended workloads: mgBench, LDBC, and gMark. We run corresponding queries based on the datasets generated by T-mgBench, T-LDBC, and T-gMark, and plot the average query latency of each workload in Figure 13. The results indicate that AeonG experiences an acceptable performance drop of up to 9.74% compared with Memgraph, a trade-off for its support of temporal queries. AeonG adopts a design that separates the current database and asynchronously transfers historical data, ensuring minimal impact on dominant non-temporal queries.

9.3.2 The impact of historical data migration

We next use the T-mgBench workload to study the query performance with varying GC intervals to control the frequency of historical data migration. We plot the latency of queries across different data types: current, reclaimed, and unreclaimed data, and graph operation in Figure 14(a). As observed, the query performance for current and unreclaimed data is relatively similar, both outperforming reclaimed data queries by up to 25.8%. This difference is attributed to the fact that querying current and unreclaimed data both need to traverse the version chain in the current storage, while querying reclaimed data requires the additional step of reconstructing a historical version using anchors and deltas in the historical storage, as detailed in Section 4. Further, increasing the GC interval from 1s to 1000s leads to a 17.3% decrease in the graph operation latency and a 9.5% increase in the query latency. This is expected as less frequent migrations can reduce contention with graph operations, thereby enhancing graph operation performance. In contrast, less frequent migrations result in longer version-chain traversal in the current storage, negatively impacting query performance.

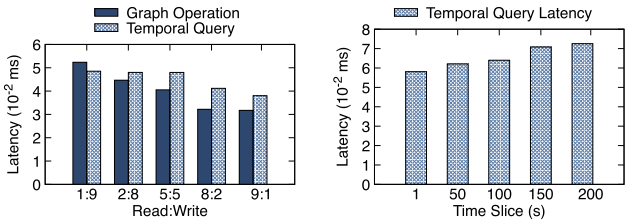
9.3.3 Analysis on the anchor interval

We evaluate the effectiveness of our adaptive anchoring approach using the T-LDBC workload. As shown in Figure 14(b), when we assign a fixed anchor interval u to each graph object and vary u from 1 to 1000, we observe that storage consumption of the historical storage decreases by 2.9× and the temporal query latency increases by 2.15×. In contrast, our adaptive anchoring approach consistently achieves near-optimal query performance and storage efficiency against all fixed anchor interval settings, because of its ability to properly balance query latency and storage overhead efficiency.



(a) The Effect of the Data Migration (b) The Effect of the Anchor Interval (c) Varying the Retention Period (d) Extending on TiKV

Fig. 14 Performance Breakdown Analysis on AeonG

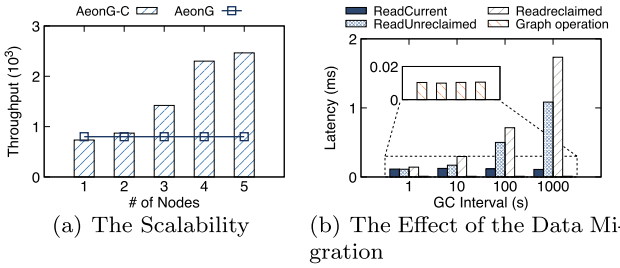


(a) Overall Performance (b) Varying Time Slice

Fig. 15 Concurrent Read-Write Performance of AeonG

9.3.4 The impact of the historical retention period

We utilize the T-LDBC workload to evaluate the historical storage overhead and temporal query latency with varying historical data retention periods. We simulate one day's amount of graph evolution in just one minute, which is done by assuming a daily operation count of 100K and executing these operations within one minute. Consequently, we set historical data retention periods at 15, 30, 90, and 180 minutes, simulating real-world scenarios of half a month, one month, one quarter, and half a year, respectively. The results, shown in Figure 14(c), indicate that the storage consumption increases by 6.02× and query performance decreases by 1.62× as the retention period extends from 15 to 180 minutes. It is expected since a longer retention period results in more historical data being maintained, leading to decreased query performance. Based on this observation, we consider enabling users to set a proper `retention_period` to achieve a balance among storage overhead, historical data duration, and query performance.



(a) The Scalability (b) The Effect of the Data Migration

Fig. 16 Performance Analysis on AeonG-C

9.3.5 Analysis of Concurrent Read-Write Workloads

We evaluate the impact of concurrent read-write workloads on T-LDBC. First, we vary the read-write ratio by adjusting the proportion of temporal queries (reads) to graph operations (writes). As shown in Figure 15(a), the latency for graph operations decreases by 1.65× and for temporal queries by 1.27×, as the ratio shifts from 1:9 to 9:1. Higher write volumes increase transaction conflicts and abort rates, degrading graph operation performance. Simultaneously, increased writes generate more historical versions, which require longer traversal during temporal queries. Next, with the read-write ratio fixed at 5:5, we vary the time slice length of temporal queries. Figure 15(b) shows that latency increases by 1.25× as the time slice length grows, consistent with the trends observed in Figure 11(d).

9.4 Performance Analysis on extended AeonG

We evaluate the performance of the extended versions of AeonG: AeonG-D and AeonG-C. We deploy AeonG-D within TiKV across 5 nodes by default, with historical data horizontally partitioned among these nodes. For AeonG-C, the compute layer is configured with up to 5 RO nodes, each equipped with a 40GB data cache, while the storage layer utilizes RocksDB instances. Experiments are conducted using the T-LDBC workload, with query types selected randomly, and a total of 100 queries executed.

9.4.1 The scalability of AeonG-D

We first evaluate the scalability of AeonG-D by increasing the server count with 10 GB of historical data. As shown in the left part of Figure 14(d), temporal query latency decreases by up to 2.8× when scaling from 1 to 5 servers, due to increased parallelism—each server independently processes version lookups. Next, we assess performance as the historical data volume grows from 2 GB to 10 GB. The right part of Figure 14(d) shows that latency increases by up to 1.6×, as larger datasets incur higher data retrieval costs. Similar trends have been observed in prior work [29, 34, 39].

9.4.2 The scalability of AeonG-C

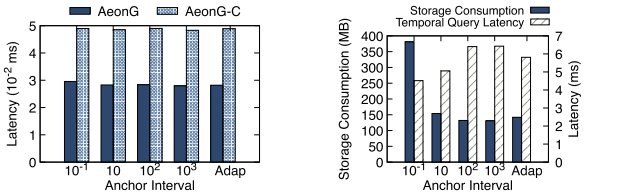
We evaluate the scalability of AeonG-C on temporal queries. Given that queries can be distributed across multiple compute nodes, we use transaction throughput as the primary metric in this experiment. As depicted in Figure 16(a), throughput scales consistently with additional RO nodes, achieving a 3.08× increase with five nodes compared to a single-node setup. In contrast, AeonG shows no improvement due to its single-node design. These results validate the scalability of AeonG-C: as the number of nodes increases, AeonG-C effectively handles a greater volume of temporal queries. Notably, with one RO node, AeonG-C delivers performance comparable to AeonG, with only a 9% drop—thanks to its effective caching mechanism that reduces remote data access overhead.

9.4.3 The impact of historical data migration

We next evaluate the impact of historical data migration in AeonG-C, with results presented in Figure 16(b). The performance of graph operations remains stable, consistent with AeonG, as AeonG-C offloads garbage collection to the storage layer. In contrast, reading unreclaimed data exhibits a 9.68× increase in latency, while reading reclaimed data shows a 12.2× increase, a more significant impact compared to AeonG. These access patterns require retrieving versions from the version queue. As the GC interval grows, the number of unreclaimed versions in the version queue increases, leading to longer times due to lock acquisition for concurrent access. Reading reclaimed data incurs additional overhead from reconstructing historical versions from the historical storage, further degrading performance.

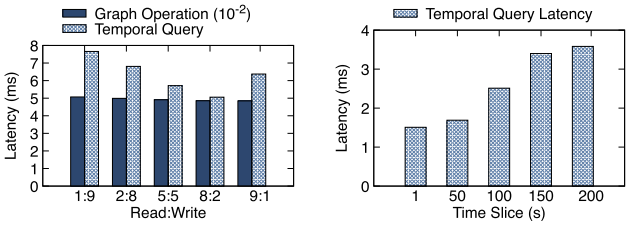
9.4.4 Analysis on the anchor interval

We evaluate the impact of the anchor interval on graph operation performance, storage overhead, and temporal query performance in AeonG-C. For graph operation performance, shown in Figure 17(a), both AeonG-C and AeonG exhibit minimal sensitivity to the anchor interval, with stable latency. However, AeonG-C incurs a 1.72× performance drop due to bRPC-induced network overhead between the RW node and storage layer. Regarding storage overhead and temporal query performance, as depicted in Figure 17(b)y, AeonG-C follows the same trend as AeonG. Storage consumption decreases by 2.9×, while temporal query latency increases by 1.4× as the anchor interval increases from 1 to 1000. The latency in AeonG-C grows more slowly than in AeonG, as the overall delay is primarily influenced by network costs, which reduces the impact of the anchor interval. Furthermore, our adaptive anchoring approach remains effective in AeonG-C,



(a) Graph Operation Performance (b) Storage Overhead and Temporal Query Performance

Fig. 17 Analysis On the Anchor Interval



(a) Overall Performance (b) Varying Time Slice

Fig. 18 Concurrent Read-Write Performance of AeonG-C

ensuring near-optimal query performance and storage efficiency.

9.4.5 Analysis of Concurrent Read-Write Workloads

We evaluate the impact of concurrent read-write workloads. As shown in Figure 18(a), graph operation latency in AeonG-C decreases slightly by 1.05× as the read ratio increases, indicating modest sensitivity compared to AeonG. This is because temporal queries are offloaded to RO nodes, alleviating load on RW nodes. Temporal query latency drops by 1.51× when the read-write ratio shifts from 2:8 to 9:1, a more significant reduction than in AeonG. This is attributed to the additional time required to traverse the redo log and version queues, which must be locked for thread safety. At high read ratios (e.g., 9:1), performance degrades as frequent historical reads increase lock contention. Additionally, as shown in Figure 18(b), temporal query latency in AeonG-C increases by 2.37× with longer time slices, consistent with Figure 15(b).

9.4.6 Analysis on the data cache size

We evaluate the performance of AeonG-C using the T-LDBC workload with a scale factor of 10 and varying data cache sizes ranging from 1 MB to 100 GB [48]. The original dataset occupies approximately 82 GB of memory when fully cached. To assess the performance of the data cache, we first warm up the database by loading sufficient graphs to populate the data cache, followed by executing our workloads. We examine the performance of read/write workloads under two access node distributions: Zipf (default setting) and uniform.

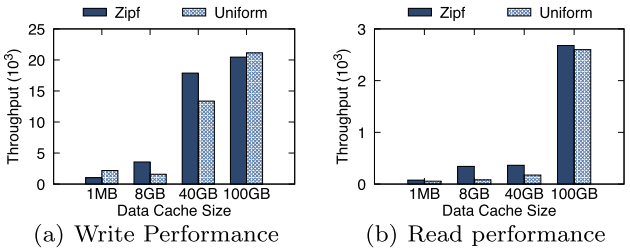


Fig. 19 Performance Analysis on Data Cache Size

For write performance, as shown in Figure 19(a), AeonG-C achieves up to 2.24× improvement under the Zipf distribution compared to the uniform distribution at cache sizes of 8 GB and 40 GB. This is because higher cache hit ratios and fewer data exchanges under the Zipf distribution. However, at extreme cache sizes (1 MB and 100 GB), performance under the Zipf distribution degrades by up to 2.13× compared to the uniform distribution. At 1 MB, frequent data exchanges occur, whereas at 100 GB, no exchanges are required. In both cases, concurrent updates on the same objects lead to more rollbacks under the Zipf distribution, which significantly impacts performance.

For read performance, as depicted in Figure 19(b), AeonG-C consistently outperforms under the Zipf distribution compared to the uniform distribution, achieving up to 4.2× improvement. This is attributed to queries accessing a subset of hot data under the Zipf distribution, which optimizes cache utilization. Read performance improves significantly when the cache is fully populated at 100 GB. However, due to the complexity of read queries (involving multi-hop operations), smaller cache sizes still require data exchanges for intermediate results, limiting performance gains.

10 Related Work

In this section, we shall review temporal graph data management and cloud databases.

Temporal Graph Data Management involves two primary approaches. One approach integrates temporal features at the application level, utilizing commercial graph databases by attaching temporal metadata [13, 14, 16, 50]. For instance, T-GQL [14] decomposes vertices into Object, Attribute, and Value vertices while retaining conventional edges, and introduces time dimensions as properties. However, there may exist unpredictable performance due to underlying engines designed for static graphs. The second strategy focuses on system-level solutions, designing storage engines to efficiently store and access growing historical data. Two prevalent storage approaches are *Copy* and *Log*. The *Copy* approach [11, 28] stores an entire graph state whenever a batch of updates occurs, simplifying queries but incurring significant redundancy. The *Log* approach [38, 54] records every graph update activity in a log, offering a more compact solution but requiring costly reconstruction to replay logs when executing a temporal graph query. To balance these trade-offs, the *Copy+Log* approach [22, 23, 29, 33, 39, 42] combines a finite set of snapshots with a list of deltas between them. In contrast, our "anchor+delta" strategy records the actual data for only evolving vertices and edges. For storage overhead, we minimize the overall consumption by recording individual graph objects. For query efficiency, we can directly fetch historical data without the need for log replay, thereby enabling more efficient query execution.

Cloud Databases have become increasingly prevalent as database vendors shift towards offering databases as a service. Recently, a new generation of cloud-native databases [6, 12, 58] has emerged, utilizing disaggregated architectures to enhance elasticity and flexibility. These systems decouple compute and storage, linking compute nodes to shared storage via high-speed networks. While most research in this area has focused on relational databases, there have been some efforts in the context of graph databases [9, 17, 20, 46], such as Amazon Neptune [9] and Cloud Graph [20]. However, existing works primarily focus on current graph states, lacking the management of historical data.

11 Conclusion

In this paper, we propose AeonG, a new graph database that efficiently offers built-in temporal support. Based on a temporal property graph model, we propose a hybrid storage engine to store temporal data with minimal storage consumption, and introduce a native temporal query engine to enable efficient temporal query processing. Further, we extend AeonG into a cloud-native database with disaggregated compute and storage layers, thus enhancing the system elasticity and scalability in managing temporal graph data. Extensive experimental evaluations show that AeonG achieves up to 5.73× lower storage consumption and 2.57× lower latency for temporal queries against state-of-the-art systems, while introducing only 9.74% performance degradation for supporting temporal features.

Acknowledgements We sincerely thank Zhouyu Wang, Guodong Jin, and Dong Wen for their early contributions.

Funding This work was supported by the National Natural Science Foundation of China (Number 61972403, 62072458).

References

1. Abdallah, A., Maarof, M.A., Zainal, A.: Fraud detection system: A survey. J. Netw. Comput. Appl. **68**, 90–113 (2016)

2. AeonGCloud: https://github.com/hououou/aeongcloud. Accessed on 2024-09
3. Angles, R., Arenas, M., Barceló, P., Hogan, A., Reutter, J.L., Vrgoc, D.: Foundations of modern query languages for graph databases. ACM Comput. Surv. **50**(5), 68:1-68:40 (2017)
4. Angles, R., Bonifati, A., Dumbrava, S., Fletcher, G., Green, A., Hidders, J., Li, B., Libkin, L., Marsault, V., Martens, W., Murlak, F., Plantikow, S., Savkovic, O., Schmidt, M., Sequeda, J., Staworko, S., Tomaszuk, D., Voigt, H., Vrgoc, D., Wu, M., Zivkovic, D.: Pg-schema: Schemas for property graphs. Proc. ACM Manag. Data **1**(2), 198:1-198:25 (2023)
5. Angles, R., Bonifati, A., Dumbrava, S., Fletcher, G., Hare, K.W., Hidders, J., Lee, V.E., Li, B., Libkin, L., Martens, W., Murlak, F., Perryman, J., Savkovic, O., Schmidt, M., Sequeda, J.F., Staworko, S., Tomaszuk, D.: Pg-keys: Keys for property graphs. In: SIGMOD Conference, pp. 2423–2436. ACM (2021)
6. Antonopoulos, P., Budovski, A., Diaconu, C., Hernandez Saenz, A., Hu, J., Kodavalla, H., Kossmann, D., Lingam, S., Minhas, U.F., Prakash, N., et al.: Socrates: The new sql server in the cloud. In: Proceedings of the 2019 International Conference on Management of Data, pp. 1743–1756 (2019)
7. ArangoDB: https://www.arangodb.com. Accessed on 2024-02
8. Bagan, G., Bonifati, A., Ciucanu, R., Fletcher, G.H.L., Lemay, A., Advokaat, N.: gmark: Schema-driven generation of graphs and queries. IEEE Trans. Knowl. Data Eng. **29**(4), 856–869 (2017)
9. Bebee, B.R., Choi, D., Gupta, A., Gutmans, A., Khandelwal, A., Kiran, Y., Mallidi, S., McGaughy, B., Personick, M., Rajan, K., et al.: Amazon neptune: Graph data management in the cloud. In: ISWC (P&D/Industry/BlueSky) (2018)
10. bRPC: brpc. https://brpc.apache.org. 22 January 2024
11. Byun, J., Woo, S., Kim, D.: Chronograph: Enabling temporal graph traversals for efficient information diffusion analysis over time. IEEE Trans. Knowl. Data Eng. **32**(3), 424–437 (2020)
12. Cao, W., Zhang, Y., Yang, X., Li, F., Wang, S., Hu, Q., Cheng, X., Chen, Z., Liu, Z., Fang, J., et al.: Polardb serverless: A cloud native database for disaggregated data centers. In: Proceedings of the 2021 International Conference on Management of Data, pp. 2477–2489 (2021)
13. Cattuto, C., Quaggiotto, M., Panisson, A., Averbuch, A.: Time-varying social networks in a graph database: a neo4j use case. In: GRADES, p. 11. CWI/ACM (2013)
14. Debrouvier, A., Parodi, E., Perazzo, M., Soliani, V., Vaisman, A.A.: A model and query language for temporal graph databases. VLDB J. **30**(5), 825–858 (2021)
15. Dgraph: https://dgraph.io. Accessed on 2024-02
16. Durand, G.C., Pinnecke, M., Broneske, D., Saake, G.: Backlogs and interval timestamps: Building blocks for supporting temporal queries in graph databases. In: EDBT/ICDT Workshops, CEUR Workshop Proceedings, vol. 1810. CEUR-WS.org (2017)
17. Dydra: Dydra. http://dydra.com. 22 January 2024
18. Facebook, O.: Rocksdb: A persistent key-value store for fast storage environments (2019)
19. Francis, N., Green, A., Guagliardo, P., Libkin, L., Lindaaker, T., Marsault, V., Plantikow, S., Rydberg, M., Selmer, P., Taylor, A.: Cypher: An evolving query language for property graphs. In: SIGMOD Conference, pp. 1433–1445. ACM (2018)
20. Graph, C.: Cloud graph. http://www.cloudgraph.com. 22 January 2024
21. Gupta, P., Mhedhbi, A., Salihoglu, S.: Columnar storage and list-based processing for graph database management systems. Proc. VLDB Endow. **14**(11), 2491–2504 (2021)
22. Han, W., Li, K., Chen, S., Chen, W.: Auxo: a temporal graph management system. Big Data Min. Anal. **2**(1), 58–71 (2019)
23. Han, W., Miao, Y., Li, K., Wu, M., Yang, F., Zhou, L., Prabhakaran, V., Chen, W., Chen, E.: Chronos: a graph engine for temporal graph analysis. In: EuroSys, pp. 1:1–1:14. ACM (2014)
24. Harding, R., Aken, D.V., Pavlo, A., Stonebraker, M.: An evaluation of distributed concurrency control. Proc. VLDB Endow. **10**(5), 553–564 (2017). https://doi.org/10.14778/3055540.3055548. http://www.vldb.org/pvldb/vol10/p553-harding.pdf
25. Huang, D., Liu, Q., Cui, Q., Fang, Z., Ma, X., Xu, F., Shen, L., Tang, L., Zhou, Y., Huang, M., et al.: Tidb: a raft-based htap database. Proceedings of the VLDB Endowment **13**(12), 3072–3084 (2020)
26. Iosup, A., Hegeman, T., Ngai, W.L., Heldens, S., Prat-Pérez, A., Manhardt, T., Chafi, H., Capota, M., Sundaram, N., Anderson, M.J., Tanase, I.G., Xia, Y., Nai, L., Boncz, P.A.: LDBC graphalytics: A benchmark for large-scale graph analysis on parallel and distributed platforms. Proc. VLDB Endow. **9**(13), 1317–1328 (2016)
27. Kankanamge, C., Sahu, S., Mhedhbi, A., Chen, J., Salihoglu, S.: Graphflow: An active graph database. In: SIGMOD Conference, pp. 1695–1698. ACM (2017)
28. Khurana, U., Deshpande, A.: Efficient snapshot retrieval over historical graph data. In: ICDE, pp. 997–1008. IEEE Computer Society (2013)
29. Khurana, U., Deshpande, A.: Storing and analyzing historical graph data at scale. In: EDBT, pp. 65–76. OpenProceedings.org (2016)
30. Kim, J., Kim, K., Cho, H., Yu, J., Kang, S., Jung, H.: Rethink the scan in MVCC databases. In: SIGMOD Conference, pp. 938–950. ACM (2021)
31. Kulkarni, A., Teevan, J., Svore, K.M., Dumais, S.T.: Understanding temporal query dynamics. In: Proceedings of the fourth ACM international conference on Web search and data mining, pp. 167–176 (2011)
32. Kulkarni, K.G., Michels, J.: Temporal features in SQL: 2011. SIGMOD Rec. **41**(3), 34–43 (2012)
33. Kumar, P., Huang, H.H.: Graphone: A data store for real-time analytics on evolving graphs. ACM Trans. Storage **15**(4), 29:1-29:40 (2020)
34. Labouseur, A.G., Birnbaum, J., Olsen, P.W., Spillane, S.R., Vijayan, J., Hwang, J.H., Han, W.S.: The g* graph database: efficiently managing large distributed dynamic graphs. Distributed and Parallel Databases **33**, 479–514 (2015)
35. Laddada, W., Ray, C.: Graph-based analysis of maritime patterns of life. In: Proceedings of the GAST Workshop, 20th Journées Francophones Extraction et Gestion des Connaissances (EGC), pp. 1–14 (2020)
36. Leo, D.D., Boncz, P.A.: Teseo and the analysis of structural dynamic graphs. Proc. VLDB Endow. **14**(6), 1053–1066 (2021)
37. Lu, Y., Yu, X., Madden, S.: STAR: scaling transactions through asymmetric replication. Proc. VLDB Endow. **12**(11), 1316–1329 (2019). https://doi.org/10.14778/3342263.3342270. http://www.vldb.org/pvldb/vol12/p1316-lu.pdf
38. Macko, P., Marathe, V.J., Margo, D.W., Seltzer, M.I.: LLAMA: efficient graph analytics using large multiversioned arrays. In: ICDE, pp. 363–374. IEEE Computer Society (2015)
39. Massri, M., Miklós, Z., Parvédy, P.R., Meye, P.: Clock-g: A temporal graph management system with space-efficient storage technique. In: ICDE, pp. 2263–2276. IEEE (2022)
40. Memgraph: https://memgraph.com. Accessed on 2024-02
41. Mgbench: https://memgraph.com/benchgraph. Accessed on 2024-02
42. Miao, Y., Han, W., Li, K., Wu, M., Yang, F., Zhou, L., Prabhakaran, V., Chen, E., Chen, W.: Immortalgraph: A system for storage and analysis of temporal graphs. ACM Trans. Storage **11**(3), 14:1-14:34 (2015)
43. Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., Schwarz, P.: Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. ACM Transactions on Database Systems (TODS) **17**(1), 94–162 (1992)
44. Neo4j: https://neo4j.com. Accessed on 2024-02

45. Neumann, T., Mühlbauer, T., Kemper, A.: Fast serializable multi-version concurrency control for main-memory database systems. In: SIGMOD Conference, pp. 677–689. ACM (2015)
46. NuvolaBase: Nuvolabase. http://www.nuvolabase.com/site. 22 January 2024
47. Ongaro, D., Ousterhout, J.: In search of an understandable consensus algorithm. In: 2014 USENIX Annual Technical Conference (USENIX ATC 14), pp. 305–319. USENIX Association, Philadelphia, PA (2014)
48. Pang, X., Wang, J.: Understanding the performance implications of the design principles in storage-disaggregated databases. Proc. ACM Manag. Data **2**(3), 180 (2024)
49. RocksDB-Cloud: https://github.com/rockset/rocksdb-cloud. Accessed on 2024-02
50. Rost, C., Gómez, K., Täschner, M., Fritzsche, P., Schons, L., Christ, L., Adameit, T., Junghanns, M., Rahm, E.: Distributed temporal graph analytics with GRADOOP. VLDB J. **31**(2), 375–401 (2022)
51. S3, A.: https://aws.amazon.com/s3/aws. Accessed on 2024-02
52. Salihoglu, S.: Kùzu: A database management system for "beyond relational" workloads. SIGMOD Rec. **52**(3), 39–40 (2023)
53. Slatedb: https://slatedb.io/. Accessed on 2025-02
54. Steer, B.A., Cuadrado, F., Clegg, R.G.: Raphtory: Streaming analysis of distributed temporal graphs. Future Gener. Comput. Syst. **102**, 453–464 (2020)
55. Storage, A.B.: https://azure.microsoft.com/services/storage. Accessed on 2024-02
56. Takac, L., Zabovsky, M.: Data analysis in public social networks. In: International scientific conference and international workshop present day trends of innovations, vol. 1 (2012)
57. Thomson, A., Diamond, T., Weng, S.C., Ren, K., Shao, P., Abadi, D.J.: Calvin: fast distributed transactions for partitioned database systems. In: Proceedings of the 2012 ACM SIGMOD international conference on management of data, pp. 1–12 (2012)
58. Verbitski, A., Gupta, A., Saha, D., Brahmadesam, M., Gupta, K., Mittal, R., Krishnamurthy, S., Maurice, S., Kharatishvili, T., Bao, X.: Amazon aurora: Design considerations for high throughput cloud-native relational databases. In: Proceedings of the 2017 ACM International Conference on Management of Data, pp. 1041–1052 (2017)
59. Verbitski, A., Gupta, A., Saha, D., Brahmadesam, M., Gupta, K., Mittal, R., Krishnamurthy, S., Maurice, S., Kharatishvili, T., Bao, X.: Amazon aurora: Design considerations for high throughput cloud-native relational databases. In: Proceedings of the 2017 ACM International Conference on Management of Data, pp. 1041–1052 (2017)
60. Wu, Y., Arulraj, J., Lin, J., Xian, R., Pavlo, A.: An empirical evaluation of in-memory multi-version concurrency control. Proc. VLDB Endow. **10**(7), 781–792 (2017)
61. Zhu, X., Serafini, M., Ma, X., Aboulnaga, A., Chen, W., Feng, G.: Livegraph: A transactional graph storage system with purely sequential adjacency list scans. Proc. VLDB Endow. **13**(7), 1020–1034 (2020)
62. Zipf, G.K.: Human Behaviour and the Principle of Least Effort: an Introduction to Human Ecology. Addison-Wesley (1949)