Cedar: A Columnar LSM-Engine for Temporal Property Graphs

Yang Wang
Beihang University
Beijing, China
yangwangcs@buaa.edu.cn

Xuelian Lin
Beihang University
Beijing, China
linxl@buaa.edu.cn

Jinghe Song
Beihang University
Beijing, China
songjh@buaa.edu.cn

Jianyong Zhu
North China Electric Power University
Baoding, China
zhujy@ncepu.edu.cn

Yu Zhao
Institute of Computing Technology, Chinese Academy of Sciences
Beijing, China
zhaoyu@ict.ac.cn

Shuai Ma*
Beihang University
Beijing, China
mashuai@buaa.edu.cn

ABSTRACT

Interactions among real-world entities can be modeled using temporal graphs, which evolve dynamically over time. Ensuring efficient storage and queries in graph databases is challenging. In this paper, we design and demonstrate Cedar, an LSM-tree-based columnar engine for temporal graphs. Firstly, Cedar unifies vertices and edges as first-class temporal events into an append-only stream, transforming complex graph updates into a sequence of atomic records. Each record is encoded into a compact 32-byte fixed-length key, enabling sequential writes that mitigate random I/Os and storage overhead. Secondly, to provide native support for this temporal stream, Cedar extends the LSM-tree's SkipList with a temporal version chain, enabling the system to handle temporal graph updates without incurring the cost of external management. Thirdly, to effectively store a query temporal property graphs, Cedar transitions row-oriented events into a Zone-Columnar on-disk layout. This layout optimizes storage density via vertical compaction while leveraging columnar efficiency to accelerate temporal-analytical queries. Finally, we demonstrate Cedar's efficacy through a deployment scenario on real-world power grid, showcasing substantial improvements in ingestion throughput and temporal query latency.

PVLDB Reference Format:
Yang Wang, Xuelian Lin, Jinghe Song, Jianyong Zhu, Yu Zhao, and Shuai Ma. Cedar: A Columnar LSM-Engine for Temporal Property Graphs. PVLDB, 19(12): 4790 - 4793, 2026.
doi:10.14778/3827998.3828123

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

1 INTRODUCTION

Temporal graph data is critical to many applications, including financial fraud detection, power grid monitoring, and cybersecurity [2]. These applications demand the capability [4] from the underlying system: they must ingest high-velocity streams of evolving graph structures and dynamic properties in real-time, while simultaneously supporting complex historical analytics and causal tracing across extensive temporal windows.

Existing works on temporal graph data management have primarily focused on the trade-off between storage cost and read/write efficiency. ChronoGraph [1] supports temporal queries by materializing full graph snapshots at each time point, causing linear storage growth and high redundancy, and thus is unsuitable for high temporal updates. The log-based approach in Raphtory [7] offers reduced storage via delta logging, but it requires expensive reconstruction to answer graph queries. Later, to bridge this gap, hybrid architecture approaches such as AeonG [3] and Aion [6] employ "current+delta" or dual-storage models that allow users to configure this trade-off by parameters. These systems support transaction-time data models and provide relatively faster temporal queries. However, these systems still rely on duplicate data structures, leading to redundant storage. Furthermore, their underlying data structures are not temporal-aware, causing severe storage/write amplification, low ingestion throughput, and high temporal query latency due to random I/Os.

Collectively, managing temporal graphs using existing systems still underscores three fundamental challenges: (1) Reduce the space cost of data structures by modeling and encoding temporal graph information with a query-efficient disk layout. (2) Capable of handling high-throughput temporal graph updates by reducing write amplification as graph history grows. (3) Support efficient temporal graph queries by a native processing engine.

In this paper, we propose Cedar for managing temporal graphs whose topology and properties change dynamically. The main contributions are listed as follows:
(1) We introduce a compact, 32-byte fixed-length data format that homogenizes both vertex and edge updates. By transforming complex structural modifications into an append-only sequential event stream, this design systematically bypasses the random I/O penalties of conventional graph stores.
(2) We propose the version chain skiplist in memory to index structure specifically tailored for streaming temporal data. By integrating version chains directly into the skiplist nodes, that significantly enhances concurrent read performance and enables efficient snapshot isolation for temporal graphs.

*Shuai Ma is the corresponding author

Proceedings of the VLDB Endowment, Vol. 19, No. 12 ISSN 2150-8097.
doi:10.14778/3827998.3828123

(3) We design a disk-resident, columnar sorted string table layout within an LSM-tree-based storage engine. Combined with an optimized compaction mechanism, this design effectively mitigates I/O amplification and reduces background maintenance overhead during disk-resident data merges.

2 DATA MODEL AND ENCODING

Inspired by event sourcing, we model the temporal property graph [2] as an append-only stream of atomic events, wherein every modification to vertices, edges, and properties is encapsulated as a self-contained, context-rich event.

Definition 1 (Atomic graph event)**.** Let $\mathcal{N}$ denote the set of vertex. We define the countable domains of labels ($\mathcal{L}$), property keys ($\mathcal{K}$), and values ($\mathcal{V}$), with Ω representing a continuous, totally ordered temporal domain. The atomic graph event is formally defined as a tuple $\varepsilon = \langle s, t, o, \phi, \kappa, \tau, \delta, v \rangle$, where:

- $s \in \mathcal{N}$ denotes the source vertex of the update or connection;
- $t \in \Omega$ denotes the timestamp of the event;
- $o \in \mathcal{N} \cup \{\text{null}\}$ denotes the target vertex for edge events ({null} for vertex events);
- $\phi \in \mathcal{L} \cup \mathcal{K}$ denotes the type predicate, specifying either a structural label ($\mathcal{L}$) or a property key ($\mathcal{K}$);
- $\tau \in \{\text{NODE, EDGE, PROP}\}$ is the Event Class;
- $\delta \in \{\text{CREATE, UPDATE, DELETE}\}$ is the Operation Type;
- κ denotes the sequence number for totally ordering concurrent events at t;
- $v \in \mathcal{V}$ is the Property Value.

Definition 2 (Temporal Property Graph)**.** A temporal property graph is defined as a totally ordered history $\Sigma = \langle \varepsilon_1, \varepsilon_2, \ldots, \varepsilon_m \rangle$ of atomic graph events, linearized via lexicographical (t, κ) precedence. As the ground truth, Σ captures all topological and property updates, ensuring full traceability of the graph's evolution.

Encoding of atomic graph event. We design a unified 32B fixed-length encoding (detailed in Table 1) to accommodate massive-scale ingestion and microsecond-level temporal precision, projecting all atomic graph events into a single `Key-Value` space. Logically, this design treats time as a first-class citizen, achieving a semantic fusion of these atomic events. This hardware-conscious design also eliminates memory fragmentation as every two keys are aligned in a 64-byte CPU cache line.

Table 1: Data format of the unified 32B fixed-length key

Field	Size (Bytes)	Role
`entity_id`	8	Entity ID; big-endian for lex order
`timestamp`	8	Descending Timestamp; newer first
`target_id`	8	V: Null / EO: `dstID` / EI: `srcID`
`column_id`	2	Property/Edge type (max 65,535)
`sequence`	2	μs-level dedup sequence
`entity_type`	1	Enum: 0=V, 1=EO, 2=EI
`flags`	1	State bits: Del/Comp/Tomb/Ext
`partition`	2	Reserved (Future: partition ID)

3 STORAGE ENGINE DESIGN

Cedar's storage engine is designed with a primary focus on storage architecture under a rigorous ACID transactional framework, as illustrated in Figure 1.

3.1 Version Chain Skiplist in Memory

To overcome the limitations of traditional schemes in temporal retrieval, we design a Version Chain Skiplist (VCSL) by extending the multi-layer skiplist with vertical older pointers to manage space and time sequence (Figure 1 A). Specifically, the VCSL is strictly partitioned both logically and physically into two tiers:

(1) Entity Index Layer: Built upon a lock-free skip list, this layer is designed for highly efficient routing across the graph space. Each multi-version entity is mapped to a unique sentinel node in this layer. Each node encapsulates the composite key (`entity_id, column_id, target_id`) and maintains two pointers, `Next` and `down`, for skip-list navigation. Crucially, the node uses a dedicated atomic pointer, `latest`, to locate the entity's latest version, thereby linking the index structure to the underlying version data.

(2) Version Data Layer: This layer is composed of extremely lightweight version nodes, dedicated exclusively to the vertical tracing of historical states. To minimize storage overhead, each node is stripped down to encapsulate only the essential metadata and data payload (`timestamp, txn_version, property_value`), along with a singly linked `older` pointer that strictly maintains the version chain in descending order of timestamps.

Complexity Analysis. When querying temporal graphs of N entities, with an average K versions per entity, current flat indexing reduces the lookup complexity to $O(\log(N \times K))$. In contrast, VCSL isolates the index scale from version depth, bounding horizontal routing to $O(\log N)$. Direct access via the `older` pointer resolves frequent latest-state queries in $O(1)$. For historical time-travel queries, the descending version chain guarantees early termination, bounding complexity to $O(\log N + V')$, where V' is the number of skipped intermediate versions ($V' \ll K$). Meanwhile, VCSL mitigates hot vertex overhead by avoiding $O(K)$ full link scans. By embedding version chains into skiplist nodes, it improves concurrent reads and supports efficient snapshot isolation for temporal graphs.

3.2 Columnar Storage on Disk

Semantic Zone Partitioning. Cedar reduces read amplification in temporal graph analytics with a semantic-aware columnar layout, clustering data into five zones based on access frequency and co-retrieval patterns:

(1) Zones 0 & 2 (Topology): These zones store vertex identifiers using composite RLE-Delta encoding, leveraging power-law degree distribution for high data locality and compression.

(2) Zone 1 (Temporal Vector): This zone uses Delta-of-Delta encoding to maintain 8-byte version timestamps, enabling efficient temporal range scans at $O(\log M + k)$.

(3) Zone 3 (Metadata): A bitmap stores $\langle \tau, \delta, \kappa, \text{flags} \rangle$, allowing predicate pushdown for filtering without decompressing.

(4) Zone 4 (Property Values): A hybrid storage area manages variable-length properties through an threshold mechanism.

Cedar maintains consistency in heterogeneously encoded zones through a virtual RowID contract, which maps rows implicitly

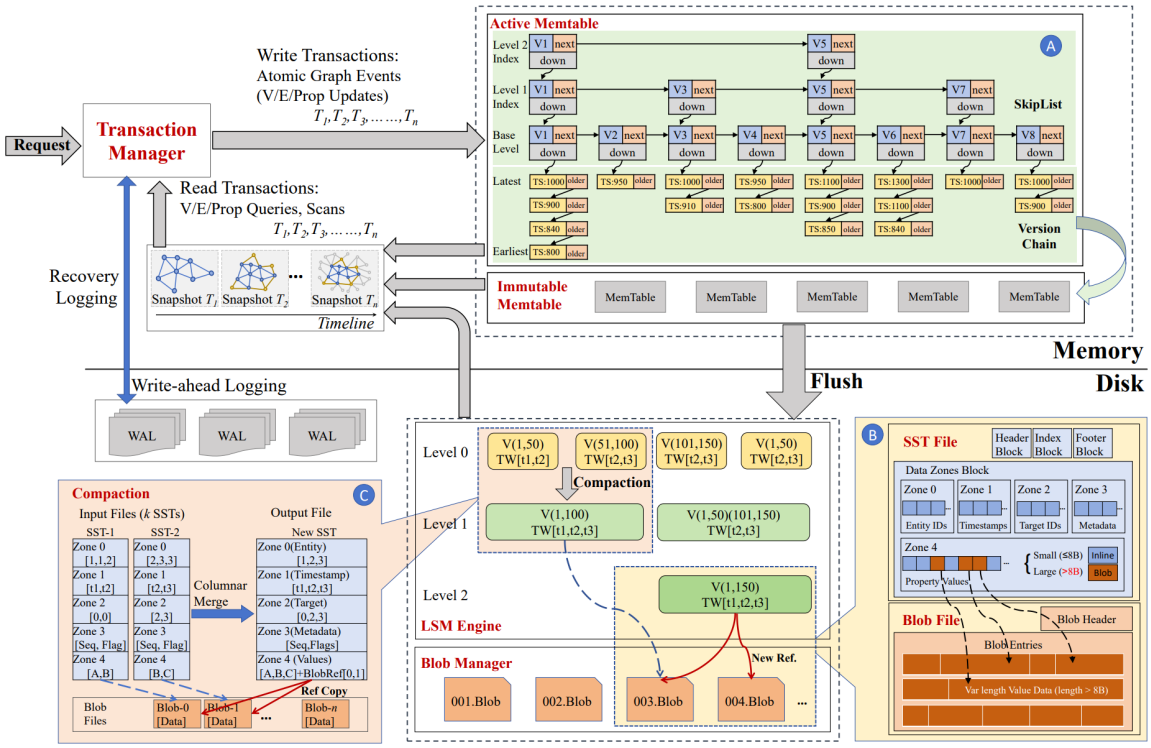


Figure 1: The design of the Cedar storage engine

within each SST. This is implemented via a zone offset table (ZOT) in each block's footer, serving as an index of restart points (row-to-offset pairs) for every zone at intervals of $K = 128$ rows. When accessing a specific row r in zone z, the system uses the global SST index to find the block, then consults the ZOT to locate the nearest restart point $p = \langle r_p, \text{off}_p \rangle$ where $r_p \le r < r_p + K$. By decompressing only the sequence from off_p, Cedar achieves $O(1)$ row localization within compressed blocks, avoiding the sequential scan overhead of traditional variable-length encoding.

Compaction mechanism. Traditional LSM-based storage engines treat key-value pairs as atomic units during compaction, which necessitates frequent and redundant serialization/deserialization of property values during the merge-sort process. Cedar exploits the immutability of historical versions and the semantic independence of zones to implement zone-level merge (ZLM). This mechanism significantly mitigates write amplification by applying differential processing strategies across zones (detailed in Figure 1(C)):

(1) Lightweight Metadata Merging: ZLM first orchestrates a columnar K-way merge for metadata zones (Zones 0–3), followed by deduplication keyed on ⟨`entity_id`, `timestamp`⟩. Given the fixed-width and regular nature of metadata, this phase is highly efficient, involving only minimal memory copies and basic logical pruning.

(2) Differential Property Processing (Zone 4): Property values ≤ 8KB are stored as compressed `inline` values within the block. For large objects (Blobs) exceeding 8KB, during high-frequency $L_0 \rightarrow L_1$ compactions, the system preserves existing reference pointers without accessing the underlying physical files, thereby achieving 0-copy forwarding.

(3) Row Synchronization and ZOT Reconstruction: Upon completion, the system performs a consistency check to ensure row alignment across all zones. It then reconstructs the zone offset table for each output data block to establish a fresh virtual `RowID` mapping, maintaining the structural integrity of the SST.

Complexity Analysis. Let N be the total number of input records, D be the number of duplicates eliminated, and B be the count of large Blobs. ZLM's time complexity is $O(N \log k)$ for metadata zones and $O(N - D)$ for Zone 4. The write amplification is modeled as $\text{WA} = [(N-D) \cdot (s_{0..3} + \mathbb{I}_{\text{inline}} \cdot s_4) + \alpha \cdot B] / \text{InputSize}$, where $s_{0..3}$ is the average metadata size, and $\alpha \ll 1$ represents the negligible overhead of pointer forwarding during $L_0 \rightarrow L_1$ transitions.

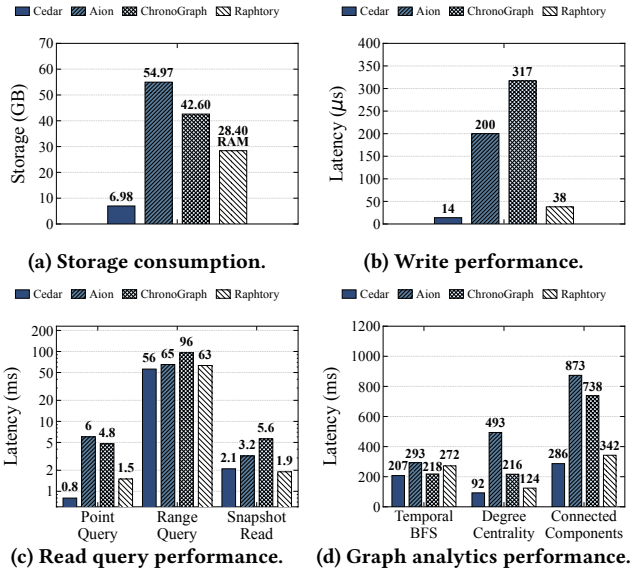
4 DEMONSTRATION

We demonstrate Cedar on a real-world European renewable energy dataset [5], which captures 60 days of power-flow dynamics. The dataset contains about 1.5K vertices and 2.2K edges; each vertex has 6 static and 5 temporal properties updated every 60 seconds, producing 1.5K records per minute and 674.97M temporal records in total. We evaluate Cedar against Aion, ChronoGraph, and Raphtory, followed by two representative use cases.

(1) Performance Evaluation. All experiments run on a workstation with an Intel Xeon Gold 5218 CPU, 64GB memory, and Ubuntu 20.04 LTS. Figure 2 reports the results.

Cedar achieves the smallest storage footprint, consuming only 6.98 GB. It reduces storage consumption by 87.3%, 83.6%, and 75.4% compared with Aion, ChronoGraph, and Raphtory, respectively, demonstrating the effectiveness of its columnar layout.

For write performance, Cedar reduces append latency to 14 μs, achieving 14.3×, 22.6×, and 2.7× speedups over Aion, ChronoGraph, and Raphtory, respectively. For temporal reads, Cedar achieves the best performance on `AS OF` point queries and `BETWEEN` range queries, with latencies of 0.8 ms and 56.0 ms. Although Raphtory is slightly faster on snapshot reads, Cedar provides comparable snapshot performance with much lower storage overhead. For graph analytics, Cedar achieves the lowest latency on all workloads, completing Temporal BFS, Degree Centrality, and Connected Components in 207 ms, 92 ms, and 286 ms, respectively. Overall, Cedar improves graph analytics performance by 2.84× over Aion, 1.85× over ChronoGraph, and 1.28× over Raphtory on average.



(a) Storage consumption. (b) Write performance.

(c) Read query performance. (d) Graph analytics performance.

Figure 2: Performance comparison of Cedar, Aion, ChronoGraph, and Raphtory.

(2) Use Case: Power Grid Management. Figure 3 illustrates a renewable energy grid management application built on Cedar.The system visualizes the overall distribution of power plants and transmission lines on a map, and supports three types of temporal access through complex Cypher queries: retrieving point-in-time vertex properties, querying structural relationships, and scanning interval-based property histories.

Figure 4 illustrates the workflow for diagnosing a short-circuit at BUS101. Cedar first rewinds the grid state to one second before the fault using `AS OF` queries to establish a pre-incident baseline. It then traces cascading propagation paths and overloaded lines within a 4-hop radius via temporal BFS, quantifies blackout impact through snapshot-consistent reads, and evaluates structural resilience using connected component analysis. This scenario highlights Cedar's ability to unify point lookups, temporal traversals, and global graph analytics in a single low-latency workflow.

5 CONCLUSIONS

We introduced Cedar to manage temporal graphs, addressing the challenges posed by dynamic data through its innovative LSM-tree-based architecture. By modeling graph updates as an append-only

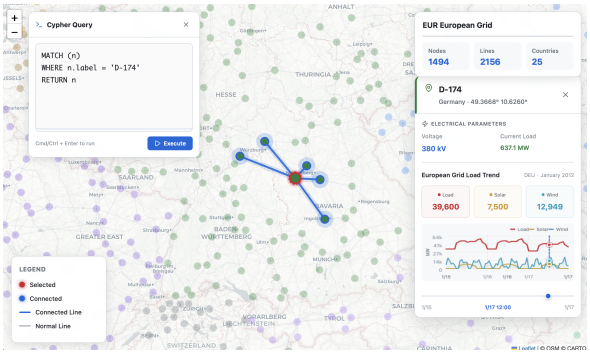


Figure 3: Exploring Temporal Graphs (Use Case)

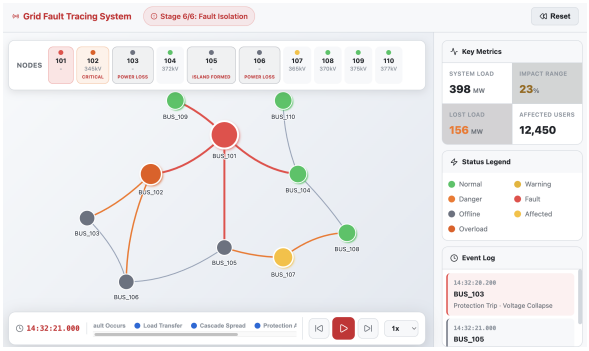


Figure 4: Power Grid Fault Tracing (Use Case)

temporal event stream, Cedar not only optimizes storage but also enhances the efficiency of querying and analysis. Experiments on real-world data have demonstrated Cedar's capability to perform complex temporal queries rapidly and scalably, positioning it as a robust solution for applications that require comprehensive handling of temporal graphs.

ACKNOWLEDGMENTS

This work is supported in part by NSFC U22B2021 & U24B20143.

REFERENCES

[1] Junghwan Byun, Sungpack Woo, and Dong-Wan Kim. 2020. Chronograph: Enabling Temporal Graph Traversals for Efficient Information Diffusion Analysis Over Time. *IEEE Transactions on Knowledge and Data Engineering* 32, 3 (2020), 424–437. https://doi.org/10.1109/TKDE.2019.2891313

[2] Hanqing Chen, Shuai Ma, Junfeng Liu, and Lizhen Cui. 2025. Discovery of Temporal Network Motifs. *IEEE Transactions on Knowledge and Data Engineering* 37, 5 (2025), 2376–2390. https://doi.org/10.1109/TKDE.2025.3538514

[3] Jiamin Hou, Zhanhao Zhao, Zhouyu Wang, Wei Lu, Guodong Jin, Dong Wen, and Xiaoyong Du. 2024. AeonG: An Efficient Built-in Temporal Support in Graph Databases. *Proc. VLDB Endow.* 17, 6 (2024), 1515–1527. https://doi.org/10.14778/3648160.3648187

[4] Haixing Huang, Jinghe Song, Xuelian Lin, Shuai Ma, and Jinpeng Huai. 2016. TGraph: A Temporal Graph Data Management System *(CIKM '16)*. 2469–2472. https://doi.org/10.1145/2983323.2983335

[5] T. Jensen and P. Pinson. 2017. RE-Europe, a large-scale dataset for modeling a highly renewable European electricity system. *Sci. Data* 4 (2017), 170175.

[6] Ali Khosravi and Yu Zhang. 2024. Aion: Log-based Transactional Temporal Graph Database. In *EDBT'24*. 337–349. https://doi.org/10.48786/edbt.2024.28

[7] Benjamin A Steer, Felix Cuadrado, and Richard G Clegg. 2020. Raphtory: Streaming analysis of distributed temporal graphs. *Future Generation Computer Systems* 102 (2020), 453–464. https://doi.org/10.1016/j.future.2019.08.019