GraphVault: A Temporal Graph Persistence Engine

Julian Bichl[1,2], Thomas Driessen[1], Melanie Langermeier[1] and Bernhard Bauer[2]
[1] *qbilon GmbH, Hermanstraße 5, 86150 Augsburg, Germany*
[2] *Software Methodologies for Distributed Systems, University of Augsburg, Universitatstrasse 6a, 86135 Augsburg, Germany*

Keywords: GraphVault, Temporal Graphs, Graph Query Engine, Temporal Graph Persistence, Graph Databases.

Abstract: Graph structures have gained increasing popularity in recent years as they offer comprehensive possibilities for managing and analyzing high interconnected data. In order to facilitate the orchestration of these data, graph databases have been developed enabling graphs to be stored as central entity. However, traditional graph databases and frameworks consider graphs as a inherently valid unit without temporal reference which can limit their ability to perform advanced analysis. This paper presents GraphVault, a graph persistence engine that is capable of efficiently storing graphs and reconstructing labeled property graphs over time. We present our temporal data model, which we mapped to a key-value engine using a purpose-built record design. The performance of our implementation is then compared to that of a conventional graph database.

1 INTRODUCTION

At the beginning of the 21st century, technical advancements in Big Data enabled the capturing of highly interconnected information, in which not only the individual data record but the interconnectivity of the data among each other serves as the main source of knowledge acquirement. While common data structures like tables are unsuitable for representing highly interconnected data, graphs consisting of attributed nodes and edges provide a decisive capability in structuring and analyzing connected information. Current research considers graphs as key enabler for future advances, e.g. the Gartner Inc. predicts an 80 percent use of graphs and graph technologies in shaping innovation by 2025 (Rita, 2021).

However, common graph database systems can only store static graphs. Therefore, observing and analyzing past graph mutations over time is not natively supported by most systems. This missing information represents enormous untapped potential. E.g. Financial Fraud Detection uses graph databases to identify fraud rings through reused telephone numbers or addresses (Sadowski and Rathle, 2014). Employing a temporal graph database, such rings could also be identified over time even if the fraudsters never used identical data records at the same period of time. Another example of graph databases in action are product recommendation systems enabling the generation of targeted recommendations for customers by linking products to their associated buyers. As customer buying behavior changes over time, the use of a temporal graph database could weaken the evaluation quality of past purchases or could incorporate seasonal events into the generation of new product recommendations.

These examples demonstrate the potential advantages of incorporating the temporal dimension into existing graph databases. This paper introduces GraphVault, a robust temporal graph persistence engine that can efficiently store a graph over time and rebuild it at any earlier point.

The remaining paper is structured as follows: In Chapter 2 we give an overview over different existing approaches that enabled graph tracking over time. Following this summary, our paper presents our solution in chapter 3, in which we have mapped a temporal graph model to a key-value engine. Our approach is further evaluated in Chapter 4, where we compare the query performance of past graphs to a general used graph database.

2 RELATED WORK

In recent years, there have been several solutions to connect temporal dimension with graph databases and thereby track changes over time. The following chapter provides an overview of the most relevant ap-

Bichl, J., Driessen, T., Langermeier, M. and Bauer, B.
GraphVault: A Temporal Graph Persistence Engine.
DOI: 10.5220/0012556500003690

In *Proceedings of the 26th International Conference on Enterprise Information Systems (ICEIS 2024) - Volume 1*, pages 224-231
ISBN: 978-989-758-692-7; ISSN: 2184-4992

proaches. However, the list does not claim to be exhaustive and rather points out approaches that differ significantly from each other.

The approach presented by Massri et al. is based on a columnar database and tries to optimize the calculation time during querying the graph at a given point in time, while minimizing the amount of storage used to store graph mutations (Massri et al., 2020). In their research they defined the *Copy Model* (complete graph snapshots are saved despite high redundancy), the *Log Model* (only the initial graph and the changes are saved) and the *Copy-on Write Model* (only modified graph elements are copied). To achieve a compromise between the need of computational effort for graph calcuation at a given point in time and necessary storage to persist it, they introduced a new model called *Copy+Log*. This model splits the graph history into time chunks, which themselves are structured by the *Log Model* with an initial graph and stored graph mutations. Further development optimized the querying time by implementing backwards calculable graph modifications (Massri et al., 2023). This reduced the needed disk space by reducing the initial graph to an amount of mutations applied on the initial graph of the previous chunk.

Rost et al extended their distributed graph analysis framework *Gradoop* in order to support temporal validity for graph elements (Rost et al., 2019). By adding two optional intervals to every record in their columnar database, it was possible to attach the valid and the transactional time to every node and edge. On the basis of this additional information, they were able to define temporal analysis operators executed on their Apache Flink based processing engine. These operators made it possible to query graphs at a given time, to calculate the difference of two time-varying graphs, to group graph elements with respect to the temporal dimension and to check for graph patterns considering temporal constraints.

The authors of *ImmortalGraph* developed an optimized persistence method for temporal graphs in addition to a temporal graph processing engine (Miao et al., 2015). Employing this approach, temporal graph operators can be classified according to their degree of temporal and structural complexity. The graph data is duplicated stored and persisted in a structure-local and temporal-local data structure.. To efficiently perform computations on the graph structure at a certain point in time, the graph elements are processed and persisted in a structure-local layout. If one or more graph elements are considered over time, the temporal-local layout is used to enable more efficient temporal operators.

Debrouvier et al. defined a temporal property graph model to extend a common graph database by attaching validity intervals to the graph elements as properties (Debrouvier et al., 2021). Less data redundancy is achieved by decoupling property keys and property values and internally representing them as separated nodes. Their prototype is based on the commercial graph database *Neo4j*. They further developed their graph query language *T-GQL* that translates into *Cypher* queries and were able to query different kinds of shortest paths over time.

3 GraphVault

In this chapter, we present GraphVault, a temporal database system designed for efficient graph persistence over time and instant graph reconstruction from any prior time point.

In the following, we use the definition proposed in (Angles et al., 2017) as it provides the foundational data structure for our improved temporal graph.

Definition 1.
A directed labeled property graph G is defined as

$$G = (N, E, L, P, V, \varepsilon, \lambda, \sigma)$$

where

- $N : \{N_1, ..., N_n\}$ is a set of nodes representing entities identified by unique numeric IDs.
- $E : \{E_1, ..., E_n\}$ is a set of directed edges representing relationships between nodes. Each directed edge is identified by a unique numeric ID.
- $L : \{L_1, ..., L_n\}$ is a set of labels assigned to nodes and edges denoting the type or nature of the entity or relationships. Labels are represented by string values.
- $P : \{P_1, ..., P_n\}$ is a set of properties identified by a string.
- $V : \{V_1, ..., V_n\}$ is a set of property values of any datatype.
- $\varepsilon : (\varepsilon_s \cup \varepsilon_t)$ is a set of functions with $\varepsilon_s : E \rightarrow N$ and $\varepsilon_t : E \rightarrow N$ that maps every edge to a source and target node.
- $\lambda : (N \cup E) \rightarrow L$ is a total function that maps a label to each node and edge.
- $\sigma : (N \cup E) \times P \rightarrowtail V$ is a injective partial function that maps a node or a node with a property to a corresponding property value.

■

This definition is frequently used as the fundamental data structure for common graph databases and therefore yield to our data structure for our temporal graph engine. In the academic literature, there have been a number of different approaches to the storage of graphs over time.

In their work, Salzberg and Tsotras introduced the *Copy* and *Log* methods for temporal information storage in graphs. *Copy* involves saving complete graph states at each change, leading to redundancy, while *Log* only records changes, reducing storage but increasing reconstruction computational overhead (Salzberg and Tsotras, 1999).

In order to avoid the need to store all snapshots of the modification of a graph or to compute all modifications in order to regain the graph at a given point in time, researchers used an alternative approach by assigning validity intervals to graph elements. This method is analogous to the expansion of the SQL:2011 standard that included temporal functionalities, where temporal intervals are assigned to relational records (Kulkarni and Michels, 2012). Our approach implemented in GraphVault is based on the approach proposed by (Campos et al., 2016) and the *Duration-labeled temporal graph* presented in (Debrouvier et al., 2021), which assigns validity intervals to nodes, edges, properties and property values.

Formally, our previously defined labeled property Graph G in Definition 1 is extended to:

Definition 2.
A temporal directed labeled property graph $G_{temporal}$ is defined as

$$G_{temporal} = (N, E, L, P, V, I, \varepsilon, \lambda, \sigma, \iota)$$

where

- $N, E, L, P, V, \varepsilon, \lambda, \sigma$ are defined as in Definition 1
- $\sigma : (N \cup E) \times P \rightarrow \{v_1, ..., v_n\} \subseteq V$ is a function that maps a node or an edge with a property to a set of corresponding property values.
- $I : \{(a,b] \mid a,b \in \mathbb{N}$ and $a < b\}$ As a set of validity intervals, where a and b symbolize timestamps. a is the first timestamp where the corresponding element was valid and b the point in time when the element got invalid.
- $\iota : (N \cup E \cup P \cup V) \rightarrow \{i_1, ..., i_n\} \subseteq I$ is a function that maps graph elements like nodes, edges, properties and property values to a subset of validity intervals.

■

We also define some helper functions to better specify the formal temporal graph model:

1. $active : (N \cup E \cup P \cup V) \times \mathbb{N} \rightarrow Boolean$ such that

$$\text{active}(e,i) = \begin{cases} \text{True}, & \text{if } \exists (a,b] \in \iota(e) : \\ & i \geq a \wedge i < b \\ \text{False}, & \text{else} \end{cases}$$

The function *active* indicates whether for a graph element e at any point in time i the element existed.

These definitions allows us to attach validity intervals to every graph element. There are additional constraints enforced on the model, so that it remains consistent and all graph states can be reconstructed at any time in a valid state:

2. $\forall i \in \mathbb{N}, \forall e \in (N \cup E), \forall p \in P \colon \sigma(e,p) = v \wedge v \neq \emptyset$, $active(i,v) \Rightarrow active(i,p)$
This requirement guarantees that if a property value existed at any point, the corresponding property must have also been present.
3. $\forall i \in \mathbb{N}, \forall e \in (N \cup E), p \in P \colon \sigma(e,p) \neq \emptyset$, $active(i,p) \Rightarrow active(i,e)$
The same principle can be applied to nodes and edges, as these must also be valid whenever an associated property existed at that point in time.
4. $\forall i \in \mathbb{N}, \forall e \in E, n \in N \colon \varepsilon(e) = n$, $active(i,e) \Rightarrow active(i,n)$
The remaining constraint ensures that if an edge existed at some time, the corresponding source and target nodes must exist at that time.

These requirements ensure graph structure consistency over time. However, additional rules must be defined to ensure the logical correctness of the graph through time:

5. $\forall e_1, e_2 \in E$, $\varepsilon_s(e_1) = \varepsilon_s(e_2) \wedge \varepsilon_t(e_1) = \varepsilon_t(e_2) \Rightarrow \lambda(e_1) \neq \lambda(e_2)$
This principle ensures that if two edges with the same direction exist between two nodes, their labels must be different, because otherwise this edge would have been reused.
6. $\forall i \in \mathbb{N}, \forall p \in P$, $active(p,i) \Rightarrow |\{v \mid \forall e \in (N \cup E) \colon v \in \sigma(e,p) \wedge active(v,i) = True\}| = 1$
This constrains an active property to have only one valid value at any given time.

As defined in definition 1, every node and edge is identified by a unique numeric ID. We extend this assignment to also be valid on properties and property values. In the following, the notion *x.id* identifies the ID from graph element *x*. For properties, *p.name* identifies the name from property *p*. At property values v, v.value identifies the value payload from any datatype. With these notions, further constrains can be made about the structure of the temporal graph:

7. $\forall n_1, n_2 \in (N \cup E \cup P \cup V): n_1.id \neq n_2.id$
   Ids are completely unique within the temporal graph.
8. $\forall p_1, p_2 \in P: p_1.id \neq p_2.id, n \in (N \cup E)$, $\sigma(n, p_1) \neq \emptyset \wedge \sigma(n, p_2) \neq \emptyset \Rightarrow p_1.name \neq p_2.name$
   This constraint implies that every property must be unique in its name at a given node or edge, otherwise the already existing property would have been reused.
9. $\forall v_1, v_2 \in V: v_1.id \neq v_2.id, n \in (N \cup E), p \in P$, $v_1 \in \sigma(n, p) \wedge v_2 \in \sigma(n, p) \Rightarrow v_1.value \neq v_2.value$
   This rule ensures that each property value mapped to a property is unique based on its payload.

Given these constraints, a temporal graph that is able to reconstruct the valid graph at any point in time can be defined. However, one restriction concerns the labels of nodes and edges. According to the current formal definition, nodes and edges cannot alter their labels over time. This restriction is acceptable since a node or edge typically maintains the same type over time. However, if the demand for time-varying labels arises, the limitation of our data model can be straightforwardly overcome by using a synthetic label attribute at node or edge level as these are time-varying.

Our proposed approach can be categorized as *Log* based, since we only store the changes in our temporal graph model. However, our proposed solution shows that an efficient use of the persistence engine of storing the historical changes reduces the computational complexity to a minimum, providing a graph reconstruction performance comparable to a log approach. This requires implementing data structures and algorithms that enable an efficient search and traversal of the graph in both temporal and structural dimensions.

3.1 Persistence Engine

When examining current graph databases based on available source code or published architecture, the persistence engines can be distinct in either self-written persistence engines or those relying on existing engines. Popular representatives like Memgraph[1] or ArangoDB[2] rely on key-value engines that allow them to store their graph structures efficiently using binary key-value pairs. In addition, established graph databases show that existing key-value engines have proven themselves in the graph database world. As a result, we opted to build our temporal graph analytics engine on top of an existing key-value engine.

[1]https://memgraph.com/

[2]https://arangodb.com/

Investigating existing key-value engines, two different data structures can be distinguished for storing key-value records. In several engines, the pairs are stored in the form of an Log-structured (LSM) tree. Well-known representatives of this engine type are RocksDB[3] and WiredTiger[4]. For instance, the graph database ArangoDB leverage RocksDB engine to enable efficient and high-performance insertion of large graphs. Beside LSM trees, binary trees have demonstrated their effectiveness as a data structure for storing key-value pairs. While this data structure may have slower insertion times due to its sorting process, it has a performance advantage for reading and iterating on the data as dictated by its underlying structure. A example of a well-known representative of this data structure is the *Lightning Memory-Mapped Database* (LMDB)[5].

The aim of our graph engine is to analyze graphs over time. It is important that the underlying persistence engine can iterate over a large amount of data in a performant manner, since multiple temporal modifications of the graph can result in a significant amount of data, especially when new graph elements are created or their properties change with each new graph inserted. Since key-value engines that are based on binary trees provides more efficient iteration and data read performance, LMDB as a B+ tree-based engine is chosen as the basis for GraphVault as its sorted data structure allows for fast search and retrieval of records. This decision is supported by the fact that LMDB has already been used successfully in the implementation of (temporal) graph databases. (NationalSecurityAgency, 2016; Vijitbenjaronk et al., 2017)

3.2 LMDB

LMDB, or Lightning Memory-Mapped Database, is a high performance, embedded key-value store designed for efficient data retrieval. It was originally implemented as the back-end serving database for the OpenLDAP software developed by Symas Corporation. LMDB uses a B+ tree structure to optimize the storage of keys and values, which allows for fast search and retrieval processes, resulting in high performance data management. Data access speed is improved by using memory-mapped files, which provide direct access to the virtual memory of the operating system. (Howard, 2015)

Inside LMDB, binary key-value pairs are stored in an append-only B+ tree. Using the default configuration, the records based on binary key-value pairs

[3]https://rocksdb.org/

[4]https://source.wiredtiger.com/

[5]https://symas.com/lmdb

are sorted in ascending order. Therefore, it is recommended to define the binary key structure in such a way that information that has to be queried together in one query is located next to or close to each other in the binary tree. This enables the loading of an amount of data from LMDB with just one logarithmic search.

However, compared to other graph databases, our engine choice LMDB has the disadvantage of comparatively slower inserting performance when it comes to storing larger graphs because of the required key comparison and rebalancing of the tree. Additionally, LMDB utilizes a shadow paging technique which employs a copy-on-write process to prevent in-place page updates, but restricts the number of writing threads to one. (Howard, 2015, 6).

3.3 Record Design

GraphVault is built around LMDB by reusing its transactional functionality and serializable isolation. It converts given directed labeled property graphs to key-value records. In this chapter, we introduce the record schema used for this mapping.

3.3.1 Requirements

Our temporal storage graph engine must support two key functionalities: first, the ability to incorporate real-time graph changes into the system; and second, the capability to query and analyze graphs effectively over time. This paper primarily concentrates on the engine's ability to query the complete graph at any previous moment in time. However, as we are currently working on powerful analytical features, our engine must provide the capability to perform flexible analyses that support structural properties of nodes and edges, as well as properties related to the temporal dimension. Therefore, our goal for the temporal graph engine is to store all properties of a temporal graph in an optimal manner, facilitating their retrieval within a minimal amount of time. This is accomplished through a record design optimized for queries that provides access to structure information and property values in minimal time.

3.3.2 General Design

In the following we define key-value mapping for LMDB to ensure an uniform mapping of the temporal graph's structure through a designated record schema. An entity is defined as a node, edge, property or property value. It serves as the primary element that is used to describe the key-value records in LMDB. Each entity has a unique numeric random ID. Each entity can be associated with either a primitive attribute or a property value. Entities can associate with primitive attributes or property values denoting type or value.

Furthermore, a reference is defined as a named, directed connection between two entities. For instance, an edge has two references, source and target, which describe the connection to the nodes. The ID of the target entity specifies the reference's target.

In the same manner as the attribute and reference records, the validity intervals of the referenced entities are stored as a separate record type. The interval values are saved by concatenating the start and end values.

3.3.3 Data Records

Our proposed temporal graph model of Definition 2 can be described in four different kinds of record types visualized in table 1.

Entity Records. This record type stores all entities and their associated IDs. Therefore, all records for a given entity type are in contiguous records. Thus, all IDs of a given type can be retrieved with a range query using LMDB. As the value part of the LMDB record isn't required but non-optional, it stores an empty fill bit.

Attribute Records. This record type is used to store primitive attributes from entities. Each entity can have several attributes, but only one of a type unified by its name. Our schema stores all attributes of an entity side-by-side in LMDB's binary tree. The name of the attribute is stored as UTF-8 binary encoded value. The attribute's value is binary serialized and stored in the value section of the LMDB record.

By using this record schema, it is possible to query the type of an edge within a range query, as well as the IDs of the source and target. In our temporal graph, attribute records are used to store node and edge labels, edge source and destination IDs, property labels and property values.

Reference Records. Reference records are used to store relational data between records. These reference records are identified by a name in text format (UTF-8 binary encoded) and directed between two entities. Unlike a property record, a reference record may have multiple references with the same name, with different target entity. After the binary reference name, the ID of the reference target entity is stored binary. The value part of the LMDB record stores an empty fill bit.

Table 1: The Record Design of GraphVault in LMDB.

ID of entity type	Record Type	Record Body			Value
1 (Node),	0 (Entity)	*ID*	-	-	-
2 (Edge),	1 (Attribute)		Attribute	-	Value
3 (Property),	2 (Reference)		Reference	Reference target ID	-
4 (Property Value)	3 (Interval)		Interval	-	-

Table 2: The Index Record Design of GraphVault in LMDB.

ID of entity type	Record Type	Record Body			Value
1 (Node),	4 (Attribute Index)	Attribute	Value	*ID*	-
2 (Edge),	5 (Reference Index)	Reference	Reference target ID		-
3 (Property),	6 (Interval Index)	Interval	-		-
4 (Property Value)	7 (IntervalR Index)	IntervalR	-		-

By using this schema, all reference records are stored next to each other in the binary tree. Thus it is possible to load all references of an entity at once. In GraphVault, we store in LMDB the edge's source and destination information, the association between properties and their corresponding nodes or edges as well as their corresponding property values via reference records.

Interval Records. Interval records define the temporal validity of different entities in the key-value database. They use binary concatenation to combine the start and end numerical timestamp values to determine the interval's value. An entity can be assigned multiple validity intervals, but it must always have at least one. Identical intervals cannot be assigned to an entity more than once, and only disjunctive time intervals per entity are permitted.

3.3.4 Index Records

GraphVault is designed as a temporal graph persistence engine with a query-optimized principle in mind. I.e. any inserted data can be found efficiently and quickly, even if no explicit in-memory index has been created on the property. This query optimization involves the structures of nodes and edges as well as property information and their respective property values. As a result, it is possible to efficiently query any graph data at any time without having to consider indices in advance.

To achieve this, index records are created for all properties, references and intervals with the last part of the record referring to the entity ID to which the record applies. This index record schema is outlined in table 2. Consequently, all entities that possess the same property or reference are located next to each other in the binary tree.

Unlike the normal interval record (record type 3) and its index record (record type 6), the interval index record works with the end value first, followed by the start value. This allows interval records that end before or after a certain time to be efficiently located within the binary tree.

3.4 Temporal Record Optimization

To ensure a efficient reconstructing of historical data, the rapid identification of entities that are valid at a given point in time is required, since an associated interval record covers that point in time. However, simply concatenating the binary start and end values of an interval would result in sorting the records by the start value followed by the end value. This requires considering all records in the binary tree with an identical start value consecutively to confirm no other interval record exists at the end, which is large enough to include the given time point. Therefore the idea is to insert the interval record with the longest duration for a given start value at the top of the tree, creating a sequence of intervals sorted first by the smallest start value and then by the largest end value.

This sort behavior has been achieved by inverting the end value of interval in binary form. By taking the binary big-endian representation of the numerical end value of the interval and inverting it, every 0 turns into a 1 and vice versa. As a result, the exact reversal of the order yields the representable number range, transforming the smallest number into the largest, the second smallest into the second largest, and so on.

By organizing the interval index records in this order, the first record for a start value can be used as a break point in the iteration, since the record with the longest range for a given start value is examined first to determine if a valid interval can exist for that start value at all. If not, all subsequent records with this starting value can be disregarded, and the iteration can continue with the subsequent larger starting value.

The use of this sorting of intervals in combination with a binary search across all entries of intervals makes it possible to efficiently and specifically obtain all valid entities at a point in time and thus to reconstruct the graph at a previous state.

4 EVALUATION

In order to evaluate our implementation, we compare GraphVault with the modern database ArangoDB. ArangoDB is selected as a comparison partner because it is a multi-model database that provides a graph database, but also allows flexible structuring of the data it stores by defining JSON objects as nodes and edges (Belgundi et al., 2023). This allows an easy and straightforward migration of our data structures to ArangoDB, accomplished by assigning validity intervals to nodes and edges as well as their respective properties and property values.

4.1 Setup and Dataset

The comparison is carried out on an Intel i7-1165G7 processor with 4 cores at 2.8GHz and 16 GB of RAM. The temporal graph datasets are generated by multiple iterations of instantiating a given graph metamodel that describes nodes, edges, properties and their values using a set of probability functions. Additionally, nodes and edges are randomly removed after each graph generation to further enhance variability between temporal graphs. For each temporal iteration, these graphs are then inserted into GraphVault. In the case of ArangoDB, the entire resulting temporal graph is first computed and then finally inserted into the database. For performance evaluation, five different graph sizes ranging from 10 to 10^5 nodes and three times as many edges are generated, as outlined in Table 3. In the following evaluation, each graph is created with up to three distinct properties and varying property values. After each iteration of graph generation, random nodes and edges are removed with a 30% probability.

4.2 Results and Interpretation

The focus of the performance evaluation is on the speed of the recovery of the entire graph to a previous point in time. Therefore, we compare the mean query times for different graph sizes and mutation iterations. The consideration of hard disk space is not taken into account during the design of GraphVault as its architecture allows for efficient querying of flexible queries over time with replicated data entries, which results

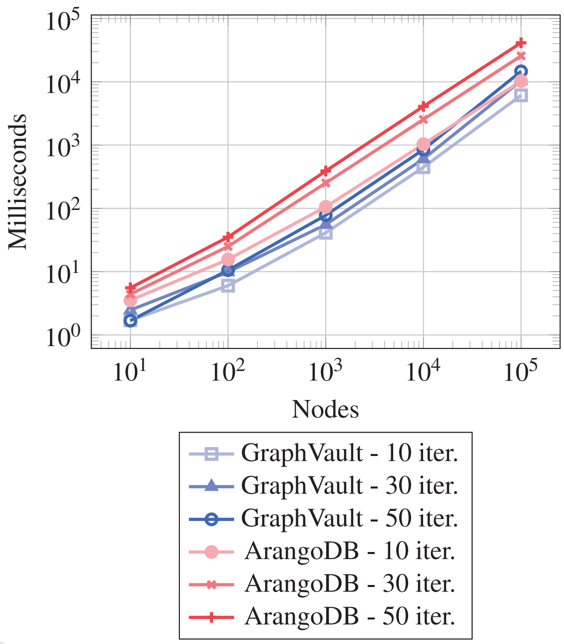

Figure 1: Comparison of execution times for GraphVault and ArangoDB with different iteration counts.

in higher memory consumption compared to similar graph databases.

Since ArangoDB currently does not allow indexes on numeric values within lists, no in-memory indexes are set on the time intervals in either system. The graphs are then generated and inserted 10, 30 and 50 times, respectively. After persisting the temporal graphs in both systems, all historical graphs are queried over all temporal iterations and the average query time is calculated. The mean query times of the comparison are shown in Figure 1 and Table 3.

Our benchmark results demonstrate that GraphVault yields a significantly faster query speed (40%-60%) for graphs as compared to ArangoDB. GraphVault's efficient ordering of the interval record notably enhances query performance, particularly for larger graphs. However, our comparison also reveals that GraphVault's query speed diminishes with an increasing number of iterations. This is because the vast number of possible edge combinations results in an increased generation of new edges, associated properties and property values with each graph iteration. These new values must be considered when conducting a temporal query.

To verify this assumption, a second experiment is conducted using GraphVault and ArangoDB with 10^5 nodes and 3×10^5 edges, and a 30% removal probability. However, the graph is generated only once before the experiment, and graph elements are dropped with the given probability per iteration before inser-

Table 3: Mean execution times for GraphVault and ArangoDB with constantly new values.

Nodes	Edges	GraphVault 10 iter. (ms)	ArangoDB 10 iter. (ms)	GraphVault 30 iter. (ms)	ArangoDB 30 iter. (ms)	GraphVault 50 iter. (ms)	ArangoDB 50 iter. (ms)
10^1	3×10^1	1.70	3.53	2.47	4.40	1.68	5.543
10^2	3×10^2	6.00	15.57	9.87	25.00	10.56	35.102
10^3	3×10^3	41.10	105.723	55.47	251.90	77.80	389.754
10^4	3×10^4	447.20	1035.00	604.73	2545.00	848.60	4089.00
10^5	3×10^5	6111.30	10255.00	10423.27	25698.00	14646.28	41401.00

Table 4: Execution times for GraphVault and ArangoDB without new values consistently getting 10^5 nodes and 3×10^5 edges inserted.

Iterations	GraphVault (ms)	ArangoDB (ms)
10	3294.60	8139.00
30	3160.80	10622.00
50	3329.64	13114.00

tion. Therefore, the graph is constantly changing, but all its components are eventually identified and can be reused in GraphVault. The average query times shown in Table 4 confirm that GraphVault requires a constant and consistent query time to restructure the graph over multiple iterations, the increased query times in Table 3 are therefore due to the growth of graph data through time.

5 CONCLUSION AND FURTHER WORK

Graphs are an excellent data structure for managing and analyzing a wide variety of data. In this paper we introduced GraphVault, a graph persistence engine that is capable of storing graph evolutions over time. We presented our extended labeled property graph data structure and explained our approach in mapping it to the key-value engine LMDB through a specific record design. Then we concluded by comparing the query speed over time between GraphVault and ArangoDB.

The next step in advancing GraphVault is the implementation of a query engine. The record design is defined with flexible queries in mind and as such, we plan to extend a common graph query language with temporal features and integrate it on top of GraphVault. The objective is to provide high performance query results that will allow us to effectively analyze graphs over time in the future. We will evaluate this feature on both generated and real-world datasets, demonstrating the potential of temporal graph analysis in practical applications.

REFERENCES

Angles, R., Arenas, M., Barceló, P., Hogan, A., Reutter, J., and Vrgoč, D. (2017). Foundations of modern query languages for graph databases. *ACM Comput. Surv.*, 50(5).

Belgundi, R., Kulkarni, Y., and Jagdale, B. (2023). *Analysis of Native Multi-model Database Using ArangoDB*, pages 923–935.

Campos, A., Mozzino, J., and Vaisman, A. (2016). Towards temporal graph databases.

Debrouvier, A., Perazzo, M., Parodi, E., Soliani, V., and Vaisman, A. (2021). A model and query language for temporal graph databases. *The VLDB Journal*, 30.

Howard, C. (2015). MDB: A Memory-Mapped Database and Backend for OpenLDAP.

Kulkarni, K. and Michels, J.-E. (2012). Temporal features in sql:2011. *SIGMOD Rec.*, 41(3):34–43.

Massri, M., Miklos, Z., Raipin Parvedy, P., and Meye, P. (2023). *Clock-G: Temporal Graph Management System*, pages 1–40.

Massri, M., Raipin Parvedy, P., and Meye, P. (2020). Gdbalive: a temporal graph database built on top of a columnar data store. *Journal of Advances in Information Technology*, 12.

Miao, Y., Han, W., Li, K., Wu, M., Yang, F., Zhou, L., Prabhakaran, V., Chen, E., and Chen, W. (2015). Immortalgraph: A system for storage and analysis of temporal graphs. *ACM Trans. Storage*, 11(3).

NationalSecurityAgency (2016). Lemongraph: Log-based transactional graph engine — github.com. [Accessed 20-10-2023].

Rita, S. (2021). Graph as The Foundation For Data, Analytics and AI. *Graph + AI Summit, October 5, 2021*.

Rost, C., Thor, A., and Rahm, E. (2019). Temporal graph analysis using gradoop. In Meyer, H., Ritter, N., Thor, A., Nicklas, D., Heuer, A., and Klettke, M., editors, *BTW 2019 – Workshopband*, pages 109–118. Gesellschaft für Informatik, Bonn.

Sadowski, G. and Rathle, P. (2014). Fraud detection: Discovering connections with graph databases. *White Paper-Neo Technology-Graphs are Everywhere*, 13.

Salzberg, B. and Tsotras, V. J. (1999). Comparison of access methods for time-evolving data. *ACM Computing Surveys*, 31(2):158–221.

Vijitbenjaronk, W., Lee, J., Suzumura, T., and Tanase, G. (2017). Scalable time-versioning support for property graph databases. pages 1580–1589.