ChronoGraph
Versioning Support for OLTP TinkerPop Graphs

Martin Haeusler, Thomas Trojer, Johannes Kessler, Matthias Farwick, Emmanuel Nowakowski and Ruth Breu

Institute for Computer Science, Technikerstraße 21a, 6020 Innsbruck, Austria

Keywords: Graph Databases, Versioning, TinkerPop, Gremlin.

Abstract: In recent years, techniques for system-time versioning of database content are becoming more sophisticated and powerful, due to the demands of business-critical applications that require traceability of changes, auditing capabilities or historical data analysis. The essence of these techniques was standardized in 2011 when it was introduced as a part of the SQL standard. However, in NoSQL databases and in particular in the emerging graph technologies, these aspects are so far being neglected by database providers. In this paper, we present ChronoGraph[a], the first TinkerPop graph database implementation that offers comprehensive support for content versioning and analysis, designed for Online Transaction Processing (OLTP). This paper offers two key contributions: the addition of our novel versioning concepts to the state of the art in graph databases, as well as their implementation as an open-source project. We demonstrate the feasibility of our proposed solution through controlled experiments.

[a]This work was partially funded by the research project "txtureSA" (FWF-Project P 29022).

1 INTRODUCTION

Graph databases offer a powerful alternative to traditional relational databases, especially in cases where most information value is found in relationships between elements, rather than their properties. Particularly with the popular and commercially succesful graph database Neo4j[1], the concept of graph databases has reached a wider audience and has inspired many other implementations, such as Titan DB[2] and Orient DB[3].

Much like SQL for relational databases, the property graph model (Rodriguez and Neubauer, 2011) Apache TinkerPop[4] (alongside the traversal language Gremlin) is the de-facto standard interface for graph databases, allowing to exchange the actual database implementation without altering the application. As these technologies mature over time, they are faced with new demands for features. Among them is the demand for system-time content versioning, primarily for the purpose of maintaining traceability of changes, providing extensive auditing capabilities, legal compliance or historical data analysis of the graph contents (e.g. trend analysis).

The concept of versioning database content is an old topic. Early work dates back to 1986 when Richard Snodgrass published his article *Temporal Databases* (Snodgrass, 1986). In the following years, several different approaches were discussed (Easton, 1986; Lomet and Salzberg, 1989; Jensen et al., 1998; Nascimento et al., 1996; Becker et al., 1996), with many of them focusing on a relational environment and SQL (e.g. Oracle's Flashback technology (Hart and Jesse, 2004), Temporal Tables in DB2 (Saracco et al., 2012) or ImmortalDB for SQL Server (Lomet et al., 2006; Lomet et al., 2008)).

Versioning for graph databases is a more recent topic (Castelltort and Laurent, 2013; Taentzer et al., 2014; Tanase et al., 2014). Castelltort et al. and Taentzer et al. have shown clearly that achieving full-featured graph versioning within a given non-versioned general purpose graph database is a challenging task. It often leads to a sharp increase of complexity in the graph structure and consequently causes issues regarding scalability, comprehensibility and performance of queries operating on the graph.

[1]https://neo4j.com/
[2]http://titan.thinkaurelius.com/
[3]http://orientdb.com/
[4]https://tinkerpop.apache.org/

Haeusler, M., Nowakowski, E., Farwick, M., Breu, R., Kessler, J. and Trojer, T.
ChronoGraph - Versioning Support for OLTP TinkerPop Graphs.
DOI: 10.5220/0006465400870097
In *Proceedings of the 6th International Conference on Data Science, Technology and Applications (DATA 2017)*, pages 87-97
ISBN: 978-989-758-255-4

With ChronoGraph[5], the first versioned TinkerPop implementation, we propose a novel solution that retains the simplicity and ease of use of graph queries on a non-versioned graph by handling the versioning process in an entirely transparent way. We are also going to showcase scenarios where versioning provides advantages in the execution of regular graph database features. As our system is already being used in practice, achieving a high performance alongside new versioning features is important. We demonstrate this through a controlled experiment.

The remainder of this paper is structured as follows: In Section 2 we present the individual requirements which we considered for our solution. In Section 3 we focus on the solution details and architecture, which is discussed in depth and compared with related work in Section 4. Section 5 presents an evaluation of our approach through experiments. Finally, Section 6 outlines our future work and Section 7 concludes the paper with a summary.

2 REQUIREMENTS

Based on our previous work with graph databases (Trojer et al., 2015) and versioning systems (Haeusler, 2016), as well as by translating versioning features in SQL (Lomet et al., 2006; Lomet et al., 2008; Hart and Jesse, 2004; Saracco et al., 2012) to their graph counterparts, we synthesized the following key requirements for a versioned graph database:

- *R1: Any Query on any Timestamp*
  In order to provide effective comparisons between individual graph versions, the essential underlying capability is to execute any given Gremlin query on any desired timestamp, without altering the query. In this way, comparisons between version a and b can invoke query q on a and b separately, and the query results can be compared directly.
- *R2: Efficient "Time Travel"*
  The ability to request the graph state at a given timestamp in the past is referred to as "Time Travel". This operation needs to be implemented in an efficient way in order to support requirement **[R1]**. Specifically, we demand that the query response time is independent of the request timestamp. In other words, requests on timestamps far in the past should not perform inherently worse than requests on recent timestamps.
- *R3: History Analysis for Vertices & Edges*
  The capability to list the timestamps at which a given vertex or edge was changed is essential for analyzing the history of the element. At these timestamps, the analysis query in question can be repeated for comparison purposes.
- *R4: Listing Change Timestamps*
  A versioned graph database should be capable of listing all changed elements in a given time range and their corresponding change timestamps, e.g. for calculating deltas. This list is not restricted to any particular element, therefore this requirement is orthogonal to requirement **[R3]**.

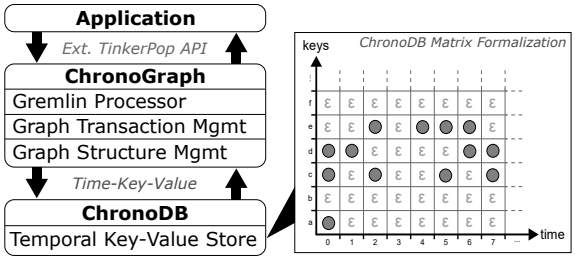


Figure 1: ChronoGraph Architecture.

3 PROPOSED SOLUTION

In this section, we present our novel versioning concepts for TinkerPop Online Transaction Processing (OLTP) graphs. We give an overview over the architecture of ChronoGraph, outline how graph data is mapped to the underlying versioned key–value store, and how the versioning concepts affect this process.

3.1 ChronoGraph Architecture

Our open-source project ChronoGraph provides a fully TinkerPop-compliant graph database implementation with additional versioning capabilities. In order to achieve this goal, we employ a layered architecture as outlined in Figure 1 a). In the remainder of this section, we provide an overview of this architecture in a bottom-up fashion.

The bottom layer of the architecture is a temporal key-value store, i.e. a system capable of working with *time–key–value* tuples as opposed to plain key–value pairs in regular key-value stores. For the implementation of ChronoGraph, we have chosen to use our own implementation called *ChronoDB*[6]. Figure 1 b) shows the matrix formalization of the store, which we have presented in our previous work (Haeusler, 2016). We refer the interested reader to our previous work, where

[5]http://tinyurl.com/chronograph-github

[6]http://tinyurl.com/chronodb-github

we provide the details of all relevant operations and data structures. In the context of this paper, it is sufficient to know that ChronoDB is a versioned key-value store that is based on a B^+-Tree structure (Salzberg, 1988) that allows to perform temporal lookups efficiently with time complexity $O(\log n)$ **[R1, R2]**.

ChronoGraph itself consists of three major components. The first component is the *graph structure* managment. It is responsible for managing the individual vertices and edges that form the graph, as well as their referential integrity. As the underlying storage mechanism is a key-value store, the graph structure management layer also performs the partitioning of the graph into key-value pairs and the conversion between the two formats. We present the technical details of this format in Section 3.2. The second component is the *transaction management*. The key concept here is that each graph transaction is associated with a timestamp on which it operates. Inside a transaction, any read request for graph content will be executed on the underlying storage with the transaction timestamp. ChronoGraph supports full ACID transactions with the highest possible isolation level ("serializable", also known as "snapshot isolation", as defined in the SQL Standard (ISO, 2011)). The underlying versioning system acts as an enabling technology for this highest level of transaction isolation, because any given version of the graph, once written to disk, is effectively immutable. All mutating operations are stored in the transaction until it is committed, which in turn produces a new version of the graph, with a new timestamp associated to it. Due to this mode of operation, we do not only achieve repeatable reads, but also provide effective protection from phantom reads, which is a common problem in concurrent graph computing. As the graph transaction management is heavily relying on the transactional capabilities of ChronoDB, we refer the interested reader to our previous work (Haeusler, 2016) for further details. The third and final component is the *query processor* itself which accepts and executes Gremlin queries on the graph system. As each graph transaction is bound to a timestamp, the query language (Gremlin) remains timestamp-agnostic, which allows the execution of any query on any desired timestamp **[R1]**.

The application communicates with ChronoGraph by using the regular TinkerPop API, with additional extensions specific to versioning. The versioning itself is entirely transparent to the application to the extent where ChronoGraph can be used as a drop-in replacement for any other TinkerPop 3.x compliant implementation. The application is able to make use of the versioning capabilities via additional methods (c.f. Section 3.4), but their usage is entirely optional and not required during regular operation that does not involve history analysis.

Table 1: TinkerPop API to Record Mapping.

TinkerPop	Record	Record Contents
Vertex	VertexRecord	id, label, PropertyKey → PropertyRecord In: EdgeLabel → EdgeTargetRecord Out: EdgeLabel → EdgeTargetRecord
Edge	EdgeRecord	id, label, PropertyKey → PropertyRecord id of InVertex, id of OutVertex
Property	PropertyRecord	PropertyKey, PropertyValue
—	EdgeTargetRecord	id of edge, id of other end Vertex

3.2 Data Layout

In order to store graph data in our versioned Key–Value Store, we need to transform the graph structure into the key–value format. For each vertex, we consider the direct neighborhood, and replace neighboring vertices by their IDs. This produces a graph fragment centered around the vertex. In contrast to the mutable TinkerPop API, these fragments need to be immutable in order to be suitable for versioning. We call such immutable representations *Records*, and there is one record type for each TinkerPop element (see Table 1). Since edges are duplicated in the lists of their incoming and outgoing vertices, we introduce the additional concept of a *EdgeTargetRecord*. It contains the ID of the other-end vertex, but no edge properties. These are stored in the *EdgeRecord* itself. This allows us to perform navigations along edges with only one ID resolution (i.e. disk access) per step. At the same time, we minimize information duplication for the edges while still allowing to iterate over all edges in a graph. The drawback is that we need to perform two ID resolutions in navigation queries that have a condition on the edge properties (c.f. Figure 2).

3.3 Versioning Concept

When discussing the mapping from the TinkerPop structure to the underlying key–value store in Section 3.2, we did not touch the topic of versioning. This is due to the fact that our key–value store *ChronoDB* is performing the versioning on its own. The graph structure does not need to be aware of this process. We still achieve a fully versioned graph, an immutable history and a very high degree of sharing of common (unchanged) data between revisions. This is accomplished by attaching a fixed *timestamp* to every *graph transaction*. This timestamp is always the same as in the underlying ChronoDB transaction. When reading graph data, at some point in the resolution process we perform a *get(...)* call in the underlying key–value store, resolving an element (e.g. a vertex) by ID. At

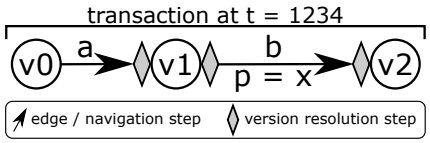


Figure 2: Example: Navigation in a Graph Version.

this point, ChronoDB uses the timestamp attached to the transaction to perform the temporal resolution. This will return the value of the given key, at the specified timestamp. We refer the interested reader to our previous work for details (Haeusler, 2016).

In order to illustrate this process, we consider the example in Figure 2. We open a transaction at timestamp 1234 and execute the following Gremlin query:

```
V(v0).out("a").outE("b").has("p","x").inV()
```

Starting from vertex *v0*, we navigate along the outgoing edge labelled as *a* to the other-end vertex. Since this edge is stored in the vertex as an *EdgeTargetRecord*, we resolve the target vertex *v1* by ID from the key–value store. At this moment, the resolution of the correct version of *v1* is taking place, requesting *v1* at the transaction timestamp, which is 1234. Navigating from *v1* to *v2* via edge *b* requires two resolutions, because we have an additional condition on a property *p* of the edge. This forces us to first resolve the edge at timestamp 1234, evaluate the condition on it, and then resolve the target vertex if the condition matches the edge. We would like to emphasize that similar loading strategies are applied by most popular graph databases; the essential difference is that the underlying storage of ChronoGraph is in addition taking care of resolving the correct version of the requested element.

The key observation here is that we *automatically* receive the correct version of the element without implementing any specific functionality in the mapping process. Since ChronoDB provides a *consistent view* on the entire database content for any given timestamp, the temporal resolution logic not only applies when loading a graph element by ID, but also when *navigating* from one element to another. We will always receive the element in the state it was at the timestamp attached to our transaction. This is a major step towards fulfilling requirement **[R1]**. As ChronoDB offers logarithmic access time to any key-value pair on any version, this is also in line with requirement **[R2]**.

3.4 TinkerPop Compatibility

The Apache TinkerPop API is the *de-facto* standard interface between graph databases and applications built on top of them. We therefore want ChronoGraph to implement and be fully compliant to this interface as well. However, in order to provide our additional functionality, we need to extend the default API at several points. There are two parts to this challenge. The first part is *compliance* with the existing TinkerPop API, the second part is the *extension* of this API in order to allow access to new functionality. In the following sections, we will discuss these points in more detail.

3.4.1 TinkerPop API Compliance

As we described in Sections 3.2 and 3.3, our versioning approach is entirely transparent to the user. This eases the achievement of compliance to the default TinkerPop API. The key aspect that we need to ensure is that every transaction receives a proper timestamp when the regular `g.tx().open()` method is invoked (see Listing **??**). In a non-versioned database, there is no decision to make at this point, because there is only one graph in a single state. The logical choice for a versioned graph database is to return a transaction on the current *head* revision, i.e. the timestamp of the transaction is set to the timestamp of the latest commit. This aligns well with the default TinkerPop transaction semantics — a new transaction $t1$ should see all changes performed by other transactions that were committed before $t1$ was opened. When a commit occurs, the changes are always applied to the head revision, regardless of the timestamp at hand, because history states are immutable in our implementation in order to preserve traceability of changes. As the remainder of our graph database, in particular the implementation of the query language Gremlin, is unaware of the versioning process, there is no need for further specialized efforts to align versioning with the TinkerPop API.

We employ the TinkerPop *Structure Standard Suite*, consisting of more than 700 automated JUnit tests, in order to assert compliance with the TinkerPop API itself. This test suite is set up to scan the declared *Graph Features* (i.e. optional parts of the API), and enable or disable individual tests based on these features. With the exception of *Multi-Properties*[7] and the *Graph Computer*[8], we currently support all optional TinkerPop API features, which results in 533 tests to be executed. We had to manually disable 8 of those remaining test cases due to problems within the test suite, primarily due to I/O errors related to illegal file names on our Windows-based development system.

[7] We do not support multi-valued properties directly as intended by TinkerPop. However, we do support regular properties of List or Set types.

[8] The Graph Computer is the entry point to the Online Analytics Processing (OLAP) API. Support for this feature may be added in future versions of ChronoGraph.

The remaining 525 tests all pass on our API implementation.

3.4.2 TinkerPop Extensions

Having asserted conformance to the TinkerPop API, we created custom extensions that give access to the features unique to ChronoGraph. As the query language Gremlin itself remains completely untouched in our case, and the graph structure (e.g. `Vertex` and `Edge` classes) is unaware of the versioning process (as indicated in Section 3.3), we are left with one possible extension point, which is the `Graph` interface itself. In order to fulfill requirements **[R1]** and **[R2]**, we need to add a method to open a transaction on a user-provided timestamp. By default, a transaction in TinkerPop on a `Graph` instance `g` is opened via one of the following methods:

```
// Variant A: Thread-bound transaction
g.tx().open();
// Variant B: Unbound transaction
Graph txGraph = g.tx().createThreadedTx();
```

Listing 1: Opening TinkerPop Transactions.

We expanded the `Transaction` class by adding two new overrides to the `open(...)` and `createThreadedTx(...)` methods:

```
Date userDate = ... ;   long userTime = ...;
// Variant A:
graph.tx().open(userDate);
graph.tx().open(userTime);
// Variant B:
Graph tx1 = g.tx().createThreadedTx(userDate);
Graph tx2 = g.tx().createThreadedTx(userTime);
```

Listing 2: Opening ChronoGraph Transactions.

Using these additional overrides, the user can decide the `java.util.Date` or `java.lang.Long` timestamp on which the transaction should be based. This small change of adding an additional time argument is all it takes for the user to make full use of the time travel feature, the entire remainder of the TinkerPop API, including the structure elements and the Gremlin query language, behave as defined in the standard. With these additional methods, together with the details presented in the previous sections, we fully cover requirements **[R1]** and **[R2]**.

We added the following methods to provide access to the history of a single Vertex or Edge:

```
Vertex v = ... ;      Edge e = ... ;
Iterator<Long> it1 = graph.getVertexHistory(v);
Iterator<Long> it2 = graph.getEdgeHistory(e);
```

Listing 3: Accessing Graph Element History.

These methods allow access to the history of any given edge or vertex. The history is expressed by an `Iterator` over the change timestamps of the element in question, i.e. whenever a commit changed the element, its timestamp will appear in the values returned by the iterator. The user of the API can then use any of these timestamps as an argument to `g.tx().open(...)` in order to retrieve the state of the element at the desired point in time. The implementation of the history methods delegate the call directly to the underlying ChronoDB, which retrieves the history of the key–value pair associated with the ID of the given graph element. This history is extracted from the primary index, which is first sorted by key (which is known in both scenarios) and then by timestamp. This ordering allows the two history operations to be very efficient as only element ID requires a lookup in logarithmic time, followed by backwards iteration over the primary index (i.e. iteration over change timestamps) until a different ID is encountered.

In order to meet requirement **[R4]**, we added the following methods:

```
long from = ...; long to = ...;
Iterator<TemporalKey> it1 = graph.
    getVertexModificationsBetween(from, to);
Iterator<TemporalKey> it2 = graph.
    getEdgeModificationsBetween(from, to);
```

Listing 4: Listing Changes within a Time Range.

These methods grant access to iterators that return *TemporalKey*s. These keys are pairs of actual element identifiers and change timestamps. Just as their element-specific counterparts, it is intended that these timestamps are used for opening transactions on them in order to inspect the graph state. Combined calls to `next()` on `it1` and `it2` will yield the complete list of changes upon iterator exhaustion, fulfilling requirement **[R4]**. Analoguous to their element-specific siblings, these methods redirect directly to the underlying ChronoDB instance, where a secondary temporal index is maintained that is first ordered by timestamp and then by key. This secondary index is constructed per keyspace. Since vertices and edges reside in disjoint keyspaces, these two operations do not require further filtering and can make direct use of the secondary temporal index.

3.4.3 Transaction Semantics

When we implemented ChronoDB, we envisioned it to be a system suitable for storing data for analysis purposes, therefore the consistency of a view and the contained data is paramount. As all stored versions are effectively immutable, we chose to implement a full ACID transaction model in ChronoDB with the

highest possible isolation level ("Serializable" (ISO, 2011)). As ChronoGraph is based on ChronoDB, it follows the same transaction model. To the best of our knowledge, ChronoGraph is currently the only implementation of the TinkerPop API v3.x that is full ACID in the strict sense, as many others opt for *repeatable reads* isolation (e.g. OrientDB, Titan Berkeley) while ChronoGraph supports *snapshot* isolation. A proposal for snapshot isolation for Neo4j was published recently (Patiño Martínez et al., 2016), but it is not part of the official version. Graph databases without ACID transactions and snapshot isolation often suffer from issues like *Ghost Vertices*[9] or *Half Edges*[10] which can cause inconsistent query results and are very difficult to deal with as an application developer. These artefacts are negative side-effects of improper transaction isolation, and application developers have to employ techniques such as soft deletes (i.e. the addition of "deleted" flags instead of true element deletions) in order to avoid them. As ChronoGraph adheres to the ACID properties, these inconsistencies can not appear by design.

4 RELATED WORK

The idea to have a versioned graph database is well-known. Several authors proposed different approaches (Castelltort and Laurent, 2013; Semertzidis and Pitoura, 2016a; Semertzidis and Pitoura, 2016b) which also highlights the importance of the problem. However, our solution is different from existing solutions in one key aspect: In contrast to other authors, we do not propose to implement the versioning capabilities within a graph itself, but rather on a lower level. This section is dedicated to a discussion of the resulting advantages and disadvantages of this design choice.

4.1 Functionality

Implementing versioning of a graph within an existing graph database is a tempting idea, given that a considerable implementation and quality assurance effort has been put into modern graph databases like Neo4j. Early works date back to 2013 when Castelltort and Laurent published their work on this topic (Castelltort and Laurent, 2013). Further work in the same vein was published recently by Taentzer et al. (Taentzer et al., 2014). However, as Castelltort's and Laurent's paper clearly shows, the versioned "meta-graph" structure can become very complex even for small examples. Changes in vertex properties are easy to represent by maintaining several copies of the vertex and linking them together by "predecessor" edges, but structural changes (addition or removal of edges) are difficult to represent within a graph structure itself. Due to this added complexity in structure, the resulting queries need to become more sophisticated as well. This can be mitigated by implementing a query translation mechanism that takes a regular graph query and a timestamp as input and transforms it into a query on the meta-graph. Aside from the inherent complexity of this mapping, the performance of the resulting transformed query will suffer inevitably, as the overall graph structure is much larger than a single graph version would be. In the common case where a query should be executed on one particular timestamp, the amount of irrelevant data (i.e. the number of graph elements that do not belong to the requested revision) in the graph increases linearly with every commit. A non-versioned general purpose graph database such as Neo4j, being unaware of the semantics of versioning by definition, has no access to any means for reacting to and/or devising a countermeasure against this problem.

Other related approaches e.g. by Semertzidis and Pitoura (Semertzidis and Pitoura, 2016b; Semertzidis and Pitoura, 2016a) or by Han et al. (Han et al., 2014), assume the existence of a series of graph snapshots as input to their solutions. These approaches do not aim for online transaction processing (OLTP) capabilities and focus on the analysis of a series of static graphs. A direct comparison to our work is therefore not feasible. However, the data managed by ChronoGraph may serve as an input to those tools, as each graph revision can be extracted individually and consequently be treated as a series of snapshots.

Our implementation is a stark contrast to existing solutions. We implement the versioning process at a lower level, in a generic versioned key–value store called ChronoDB. This store is aware of the semantics of the versioning process, and is capable of solving the problem of long histories (Haeusler and Breu, 2017), unlike the previously mentioned solutions.

To the end user, the versioning process is completely transparent, as our implementation is fully compliant with the standard TinkerPop API for non-versioned graphs. There is no need for translating one graph query into another in order to run it on a different graph version. A developer familiar with the TinkerPop

[9] Vertices that have been deleted by transaction *t1* while being modified concurrently by transaction *t2* do not disappear from the graph; they remain as *Ghost*s.

[10] Half Edges refer to the situation where an edge is only traversable and visible in one direction, i.e. the *out*-vertex lists the edge as outgoing, but the *in*-vertex does not list it as incoming, or vice versa.

API can start using ChronoGraph without any particular knowledge about the versioned nature of the graph. By offering additional methods, which are very much in line with the intentions of the TinkerPop API, we grant access to the temporal features. Additionally, ChronoGraph is fully ACID compliant with snapshot isolation for concurrent transactions, preventing common artefacts that arise in other, non-ACID graph databases, such as ghost vertices and half edges. Our solution is strongly based on immutability of existing versions, which aids in preserving traceability of changes and allows extensive sharing of data that remained unchanged between revisions.

4.2 Limitations

Our approach is tailored towards the use case of having a *versioned* graph (as opposed to a *temporal* graph), which entails that queries on a *single* timestamp are the prevalent form of read access. Even though we support additional auxiliary methods for traversing the history of a single vertex or edge, and listing all changes within a given time range, our approach is far less suitable for use cases with an emphasis on temporal analysis that require time range queries, or detection of patterns on the time axis (as in graph stream analysis (McGregor, 2014; Pigné et al., 2008)). For example, answering the question "Which elements often change together?", while possible in our solution, can not be implemented without linear scanning through the commit logs. Another example would be the query "List all vertices that have ever been adjacent to a given one", which would again involve linear iteration.

We are currently also not offering any means for distributing the graph among multiple machines (see Section 6 for details). This limits the scale of our graph to sizes manageable within the physical memory and computing resource restrictions of a single machine. Currently, the largest ChronoGraph instance used in practice that we are aware of has about 500.000 elements (vertices plus edges) in the head revision.

5 EVALUATION

Table 2 shows a very high-level look at the most relevant TinkerPop implementations in practice and compares them to ChronoGraph. Titan, Neo4j and ChronoGraph are supporting the new 3.x versions of TinkerPop, while OrientDB is still using the 2.x version. The remainder of the table can be divided into two parts. On the one hand there are distributed systems that are intended for deployment on multiple machines, and on the other hand there are systems deployed locally on one machine. In the distributed domain, the BASE[11] approach is prevalent due to its weaker consistency guarantees. For local deployments, all graph databases in the list follow the ACID principles. However, they all implement them in different ways. In particular the *Isolation* property allows for a certain degree of freedom in interpretation. Most local graph databases listed in Table 2 have *Read Committed* isolation, which is the second weakest of the four levels identified in the SQL 2011 Standard (ISO, 2011).ChronoGraph is the only implementation that provides true *Snapshot* isolation, which is the highest possible isolation level.

We assert the conformance to the TinkerPop standard in ChronoGraph via the automated test suite provided by TinkerPop which encompasses around 700 test cases. We furthermore assert the correctness of our additional functionality by adding another 1400 automated test cases, achieving a total statement coverage of approximately 70%. In order to demonstrate that the impact on performance of these new features does not harm the overall applicability of our graph database, we conducted a comparative experiment with the top three major TinkerPop implementations, alongside a second experiment that demonstrates the versioning capabilities of ChronoGraph.

5.1 Controlled Experiments

This section presents two experiments with their respective setups and results. The comparative performance experiment evaluates ChronoGraph against other popular graph databases, while the version history growth experiment focuses on the versioning capabilities of ChronoGraph and demonstrates the impact on performance as more revisions are being added.

All benchmarks were executed with 1.5GB (comparative experiment) or 3.0GB (history growth experiment) of RAM available to the JVM on an Intel Core i7-5820K processor with 3.30GHz and a Crucial CT500MX200 SSD.

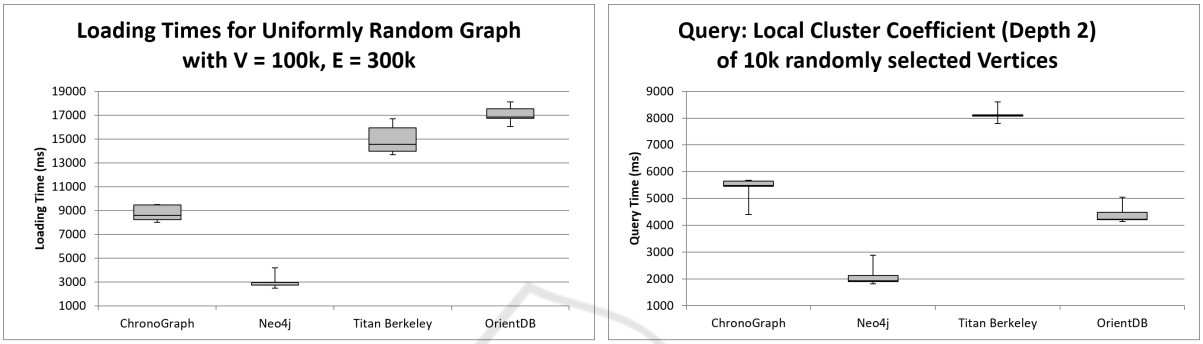
5.1.1 Comparative Performance

We generated a uniformly random graph with 100.000 vertices and 300.000 edges. We then selected a random subset of 10.000 vertices. In order to assert the reproducibility of the experiment, we persisted both the adjacency list and the identifiers of the vertex subset in a plain text file. Using the graph database under test (GUT), we loaded the text file, created the corresponding TinkerPop structure and persisted it as a new graph in the GUT. In order to have a common baseline,

[11] **B**asically **A**vailable, **S**oft State, **E**ventual Consistency

Table 2: Comparison of TinkerPop Implementations.

Graph-Database		TinkerPop Version	Deployment	Paradigm	Max. Isolation Level
ChronoGraph		3.x	Local	ACID	Snapshot
Titan	on BerkeleyDB	3.x	Local	ACID	Read Committed
	on Cassandra	3.x	Distributed	BASE	n/a
	on HBase	3.x	Distributed	BASE	n/a
Neo4j		3.x	Distributed	BASE / ACID	Read Committed
OrientDB	Local Mode	2.x	Local	ACID	Repeatable Reads
	Distributed Mode	2.x	Distributed	ACID	Read Committed



(a) Graph Loading Times.

(b) Query Response Times.

Figure 3: Performance of TinkerPop graph creation and querying.

we did not make use of vendor-specific batch-loading capabilities. The time it takes to perform this task is shown in Figure 3 a). The displayed box plot shows the results over 10 runs.

After importing the graph, we loaded the list of 10.000 random vertex IDs from the text file. For each vertex ID, we fetched the corresponding vertex from the GUT and calculated its *local cluster coefficient*, which is a two-step neighborhood query, as presented by Ciglan et al. (Ciglan et al., 2012). This experiment covers two of the primary capabilities of graph databases, which are random access and neighborhood navigation. Figure 3 b) shows the results of this benchmark, accumulated over 10 runs.

As Figure 3 clearly indicates, Neo4j is the fastest competitor in both loading and querying the graph by a wide margin. We would like to emphasize that we included Neo4j in this benchmark for reference purposes due to its popularity; its feature set is hardly comparable to the other databases in the benchmark as it primarily follows the BASE principle which allows for a wide range of optimizations that would not be possible in an ACID environment. Our ChronoGraph implementation takes second place in loading speed. OrientDB first needs to convert the graph elements into its internal document representation, while Titan is limited by the insertion speed of the underlying Berkeley DB. In terms of read access speed, ChronoGraph is the middle ground between OrientDB and Titan Berkeley in this benchmark. Here, we only deal with a single graph version, therefore a comparable speed to non-versioned graph databases is to be expected. We still suffer from a slight overhead in calculation due to the present versioning engine, which is the reason why OrientDB performs better. The main use case of Titan is in distributed environments, which explains why restricting it to a single Berkeley DB instance (the only officially supported ACID backend) causes a degradation in performance.

5.1.2 Version History Growth

In a second experiment we analyze the performance impact as an increasing amount of versions are added to the system via regular commits. For this experiment, we assume frequent small-scale changes. Even though ChronoGraph supports versioning for vertex and/or edge property values, in this experiment we only consider changes that alter the graph topology. We use the same local cluster coefficient calculation query as in the previous experiment on a graph with an increasing number of versions. To eliminate the impact of an increase in graph size, we choose the modifications randomly from a probability distribution which is designed on the fly to increase the graph size if too many deletions have occurred, and to reduce it again if too many elements have been added. All operations preserve the uniform randomness of the

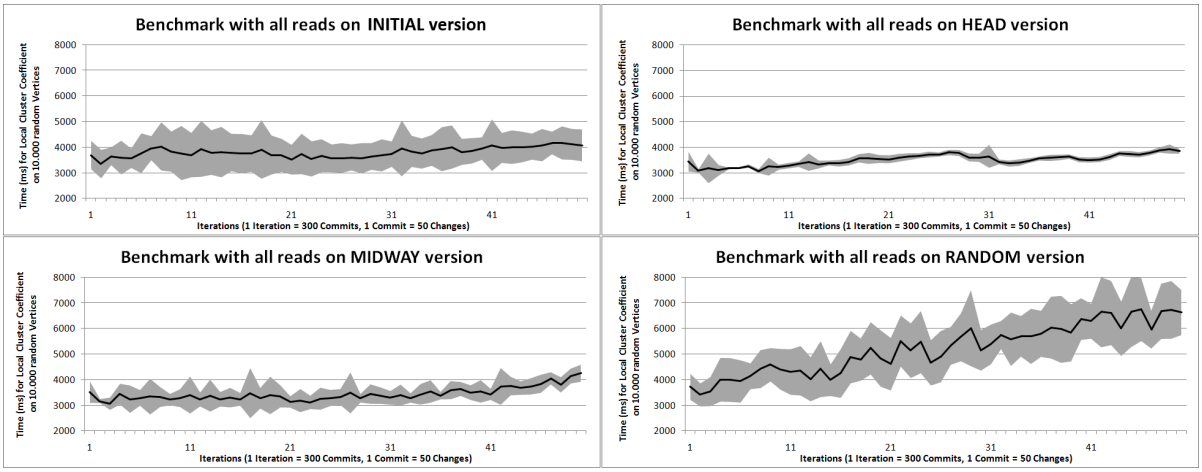


Figure 4: Query Performance with increasing number of versions.

graph structure. The four distinct events are vertex addition, vertex removal, edge addition and edge removal. The removal operations pick their target element at random. When adding a vertex, we also connect it to another, randomly chosen existing vertex via a new edge. A commit can contain any combination of these operations. In this experiment, each commit consists of 50 such structural changes. As in the previous experiment, the initial graph is uniformly random and has 100.000 vertices and 300.000 edges. The modifications in the commits alter these values. We use the random distribution of changes as a tool to keep the graph from becoming too large or too small, as we are only interested in the impact of a growing history and want to eliminate the influence of a growing or shrinking graph structure. Both the number of vertices and the number of edges is bounded within ±5% of the original numbers. The probability distribution for the change events is recalculated after every modification.

The experiment algorithm operates in iterations. In each iteration, it performs 300 commits, each of them containing 50 operations as outlined above. Then, a timestamp is picked on which a graph transaction is opened, and the local cluster coefficient query from the first experiment is calculated for 10.000 randomly selected vertices. We measure the accumulated time of these queries and repeat each measurement 10 times (keeping the timestamp constant, but selecting a different, random set of vertices each time). We would like to emphasize that the time for opening a transaction is independent of the chosen timestamp, i.e. opening a transaction on the head revision is equally fast as opening a transaction on the initial revision. The timestamp selection is done in four different ways, which gives rise to the four different graphs in Figure 4:

- INITIAL uses the timestamp of the initial commit.
- HEAD uses the timestamp of the latest commit.
- MIDWAY uses the timestamp at the arithmetic mean of the initial and latest commit timestamps.
- RANDOM chooses a timestamp randomly between the initial and latest commit (inclusive).

We granted 3GB of RAM to the JVM in this experiment (even though the benchmark program can be successfully executed with less memory) in order to prevent excessive garbage collection overhead. Compared to the first experiment, we are faced with much larger volumes of data in this case, and the test code itself has to manage a considerable amount of metadata, e.g. in order to assert the correct calculation of the probability distributions.

Figure 4 showcases a number of interesting properties of ChronoGraph. Given a graph that retains an almost constant size over time, the number of versions has a very minor effect on the query performance if the INITIAL or HEAD revisions are requested. The present increase is due to the larger number of elements in the underlying B^+-Tree structure in ChronoDB. The INITIAL version has a notably higher standard deviation than the HEAD version. This is due to the experiment setup. As new changes are committed, they are written through our versioning-aware cache. The HEAD case can take full advantage of this fact. However, for the INITIAL case, the write through eliminates older entries from the Least-Recently-Used cache which have to be fetched again to answer the queries on the initial commit timestamp.

The MIDWAY version suffers from a similar problem as the INITIAL version, however to a lesser extent because the request timestamp is closer to the latest commit than in the INITIAL case. We would like to emphasize that, while the underlying store offers the same performance for any version, the temporal cache

is limited in size and cannot hold all entries in the experiment. Therefore, choosing a timestamp that is closer to the latest commit causes an improvement in performance due to the write-through mechanics of our commit operations.

Finally, the worst case scenario is the RANDOM variant of our experiment. Here, each read batch chooses a different timestamp at random from the range of valid timestamps. This causes a considerable number of cache misses, increasing the standard deviation and query response time. As the underlying data structure in our store is a B^+-Tree that contains all revisions, which is the reason for the resulting logarithmic curve in Figure 4.

At this point, it is important to note that the versioning engine is faced with considerable volumes of changes in this experiment. After 50 iterations, the version history consists of 15,001 individual graph revisions with 50 modifications each. This translates into 750.000 high-level changes in addition to the initial version. Considering that a graph structure change translates into several key-value pair changes (e.g. a vertex delete cascades into the deletion of all connected edges, which cascades into adjacency list changes in the vertices at the other end), the number of atomic changes is even higher. Each of these atomic changes must be tracked in order to allow for per-element history queries. If we assume that each high-level graph change on average entails 3 changes in the underlying key–value store, this results in a store that has more than five times the number of elements compared to the initial graph. As Figure 4 shows, this increase in data volume does not entail an equivalent increase query response times, which demonstrates the scalability of our approach with a high number of revisions **[R2]**.

6 FUTURE WORK

The existing versioning capabilities are based on the underlying assumption that the history of each element is immutable, i.e. the past does not change. This principle will considerably ease the distribution of the versioned graph data among multiple machines, as newly incoming commits can only alter the database contents in the head revision. These commits are furthermore restricted to operate in an append-only fashion. This allows for highly effective caching and data replication without risking to ever encounter stale data. We plan on implementing such capabilities in ChronoGraph in the future.

Another feature which we intend to support in upcoming releases is the TinkerPop `GraphComputer` interface. It is a generalized interface for computation on distributed graphs, geared towards online analytics processing (OLAP) by utilizing map–reduce patterns and message passing. Thanks to the abstraction layer provided by Gremlin, this will also allow our users to run queries in a variety of languages on our graph, including SPARQL [12] (Barbieri et al., 2010).

Finally, we are also applying ChronoGraph in an industrial context in the IT Landscape Documentation tool Txture[13]. In this collaboration, we have successfully connected high-level modeling techniques with our versioned graph database. Txture assists data centers in the management of their IT assets, and in particular their interdependencies. At its core, Txture is a repository for these asset models, and ChronoGraph acts as the database of this repository, providing the required query evaluation, persistence and versioning capabilities. Txture is currently in use by several industrial customers, including the data center Allgemeines Rechenzentrum (ARZ) and a globally operating semiconductor manufacturer. We will publish a case study on the role of ChronoGraph within Txture in the near future.

7 SUMMARY

In this paper, we presented our concepts for supporting transparent system time versioning in TinkerPop OLTP graphs. We provided an overview over the core concepts, the persistence format and the underlying versioning engine of our open source proof-of-concept implementation called ChronoGraph. We presented our vendor-specific extensions to the TinkerPop API that grant access to the versioning capabilities and demonstrated that this design aligns very well with the standard TinkerPop API. In the related work section, we also presented a comparison to existing efforts and technologies and outlined the underlying principles of ChronoGraph as a novel approach to the graph versioning problem. This discussion and the following evaluation section show that ChronoGraph improves on the state-of-the-art as a new member of the TinkerPop OLTP family by introducing new versioning and querying capabilities, strict adherence to the ACID principles, as well as providing snapshot-level transaction isolation, without severely compromising performance in comparison to other popular graph databases.

Overall, we made two key contributions in this paper. The first contribution is the introduction of our novel versioning concepts into the world of graph databases, which additionally allow for snapshot-level

[12] https://www.w3.org/TR/sparql11-query/
[13] www.txture.io

transaction isolation and conformance to all ACID properties. The second contribution is the ChronoGraph implementation itself, which is a full-featured and thoroughly tested TinkerPop 3.x implementation that is freely available under an open source license. Our experiments have shown the competitive performance of this implementation.

REFERENCES

Barbieri, D. F., Braga, D., Ceri, S., Valle, E. D., and Grossniklaus, M. (2010). C-SPARQL: a continuous query language for RDF data streams. *Int. J. Semantic Computing*, 4(1):3–25.

Becker, B., Gschwind, S., Ohler, T., Seeger, B., and Widmayer, P. (1996). An asymptotically optimal multiversion b-tree. *The VLDB Journal*, 5(4):264–275.

Castelltort, A. and Laurent, A. (2013). Representing history in graph-oriented NoSQL databases: A versioning system. In *Eighth International Conference on Digital Information Management (ICDIM 2013), Islamabad, Pakistan, September 10-12, 2013*, pages 228–234.

Ciglan, M., Averbuch, A., and Hluchy, L. (2012). Benchmarking traversal operations over graph databases. In *Data Engineering Workshops (ICDEW), 2012 IEEE 28th International Conference on*, pages 186–189. IEEE.

Easton, M. C. (1986). Key-sequence data sets on indelible storage. *IBM Journal of Research and Development*, 30(3):230–241.

Haeusler, M. (2016). Scalable versioning for key-value stores. In *DATA 2016 - Proceedings of 5th International Conference on Data Management Technologies and Applications, Lisbon, Portugal, 24-26 July, 2016.*, pages 79–86.

Haeusler, M. and Breu, R. (2017). Sustainable management of versioned data. In *Proceedings of the 24th PhD Mini-Symposium*. Budapest University of Technology and Economics.

Han, W., Miao, Y., Li, K., Wu, M., Yang, F., Zhou, L., Prabhakaran, V., Chen, W., and Chen, E. (2014). Chronos: a graph engine for temporal graph analysis. In *Proceedings of the Ninth European Conference on Computer Systems*, page 1. ACM.

Hart, M. and Jesse, S. (2004). *Oracle Database 10G High Availability with RAC, Flashback & Data Guard*. McGraw-Hill, Inc., New York, NY, USA, 1 edition.

ISO (2011). SQL Standard 2011 (ISO/IEC 9075:2011).

Jensen, C. S., Dyreson, C. E., Böhlen, M., Clifford, J., Elmasri, R., Gadia, S. K., et al. (1998). *Temporal Databases: Research and Practice*, chapter The consensus glossary of temporal database concepts — February 1998 version, pages 367–405. Springer Berlin Heidelberg, Berlin, Heidelberg.

Lomet, D., Barga, R., Mokbel, M., and Shegalov, G. (2006). Transaction time support inside a database engine. In *Proceedings of the 22nd ICDE*, pages 35–35.

Lomet, D., Hong, M., Nehme, R., and Zhang, R. (2008). Transaction time indexing with version compression. *Proceedings of the VLDB Endowment*, 1(1):870–881.

Lomet, D. and Salzberg, B. (1989). Access Methods for Multiversion Data. *SIGMOD Rec.*, 18(2):315–324.

McGregor, A. (2014). Graph stream algorithms: a survey. *SIGMOD Record*, 43(1):9–20.

Nascimento, M., Dunham, M., and Elmasri, R. (1996). M-IVTT: An index for bitemporal databases. In Wagner, R. and Thoma, H., editors, *Database and Expert Systems Applications*, volume 1134 of *Lecture Notes in Computer Science*, pages 779–790. Springer Berlin Heidelberg.

Patiño Martínez, M., Sancho, D., Jiménez Peris, R., Brondino, I., Vianello, V., and Dhamane, R. (2016). Snapshot isolation for neo4j. In *Advances in Database Technology (EDBT)*. OpenProceedings. org.

Pigné, Y., Dutot, A., Guinand, F., and Olivier, D. (2008). GraphStream: A tool for bridging the gap between Complex Systems and Dynamic Graphs. In *Emergent Properties in Natural and Artificial Complex Systems. Satellite Conference within the 4th European Conference on Complex Systems (ECCS'2007)*, volume abs/0803.2.

Rodriguez, M. A. and Neubauer, P. (2011). The graph traversal pattern. In *Graph Data Management: Techniques and Applications.*, pages 29–46.

Salzberg, B. (1988). *File Structures: An Analytic Approach*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Saracco, C., Nicola, M., and Gandhi, L. (2012). A matter of time: Temporal data management in DB2 10. *IBM developerWorks*.

Semertzidis, K. and Pitoura, E. (2016a). Durable graph pattern queries on historical graphs. In *Proc. IEEE ICDE*.

Semertzidis, K. and Pitoura, E. (2016b). Time Traveling in Graphs using a Graph Database. In *Proceedings of the Workshops of the (EDBT/ICDT)*.

Snodgrass, R. T. (1986). Temporal databases. *IEEE Computer*, 19:35–42.

Taentzer, G., Ermel, C., Langer, P., and Wimmer, M. (2014). A fundamental approach to model versioning based on graph modifications: from theory to implementation. *Software and System Modeling*, 13(1):239–272.

Tanase, I., Xia, Y., et al. (2014). A highly efficient runtime and graph library for large scale graph analytics. In *Proceedings of Workshop on GRAph Data Management Experiences and Systems*, GRADES'14, pages 10:1–10:6, New York, NY, USA. ACM.

Trojer, T., Farwick, M., Häusler, M., and Breu, R. (2015). Living Models of IT Architectures: Challenges and Solutions. *Software, Services and Systems*, 8950:458–474.