In Search of an Efficient Data Structure for a Temporal-Graph Database

Tobie Morgan Hitchcock
Kellogg College, University of Oxford

November 2019

A dissertation submitted in partial fulfilment of the requirements for the degree of Master of Science in Software Engineering

Abstract

The use of immutable, versioned data stores is playing an ever more significant role in a number of application domains and business use cases. Decreases in data storage pricing has led to the ability to store more data, and in many cases to store the data in an immutable way - so that future changes to the dataset augment, rather than mutate the data. There are currently a variety of different methods for storing historical, versioned data, each with its own benefits and negatives.

The objective of this dissertation is to analyse the current landscape of versioning within key-value data stores and graph databases, exploring and analysing the efficiency of the current data structures used for data storage. I aim to propose a new approach for the underlying data structure which should allow for more efficient querying, and optimised data storage characteristics, whilst at the same time offering more advanced methods for querying the dataset. I will then explore and analyse the design of a key-value store for use in both a single-node, and a distributed architecture, going on to provide a practical implementation, using software engineering techniques, of an embedded key-value data store. Finally, although a direct comparison between the different approaches is not necessarily possible or easy to define, I will analyse the theoretical efficiency and will compare and benchmark the proposed approach against the alternative data stores.

Acknowledgements

I should like to express my sincere gratitude to all the tutors on the Software Engineering Programme, with particular thanks to my supervisor Dr. Peter Bloodsworth, who has given me unfailing encouragement and support. I should also like to record my thanks to Mr. Richard Pettinger, whose endorsement was an early catalyst for my change in career path. Last, but by no means least, I should like to thank my family - Martin, Ingrid, Jaime and Raj - for their unstinting patience and help.

Declaration

The author confirms that this dissertation does not contain material previously submitted for another degree or academic award; and the work presented here is the author's own, except where otherwise stated.

Table of contents

Table of figures

Definition of key terms

Throughout the course of this dissertation I shall make use of a few terms which are related to each other, and which may at times be used interchangeably with each other within the context of this dissertation.

Immutable data: When used within the context of a database, immutable data is data which does not change once it has been stored in the database.

Historical data: This term denotes data that is accessed using a previous versioning identifier or timestamp.

Versioned data: Versioned data is similar to immutable data, but previous versions can be altered.

Temporal data: Temporal data is similar to versioned data in that it has previous defined versions, which are versioned according to some measure of time in the past or the future. When referring to temporal data, I may also talk about *valid time* (the time at which the data value was valid in the real world) and *transaction time* (the time at which a data value was inserted or changed within the database) separately.

In addition, I shall make use of a number of key terms for the different parts or layers of the proposed implementation. When looking at related research and literature, these terms are used interchangeably with one another, and as a result I have formally defined them here so that each reference is elucidated.

Database: Sometimes known as a database (DB) management system, in the context of this dissertation a database is a system which offers advanced query functionality and data access methods.

Data store: In this dissertation, I make use of this term when discussing key-value (KV) data stores, or distributed data stores. These are typically focused on storage and retrieval rather than advanced query functionality.

1. Introduction

The use of immutable, versioned data stores (sometimes known as temporal data stores) is playing an ever more significant role in a number of application domains and business use cases, promising a number of advantages [1]. Decreases in storage pricing [2] have led to the ability to store large amounts of data, and in many cases have led to the ability to store the data in an immutable way - so that any future changes to the dataset augment, rather than change, any data already residing in the data store. This has led to the possibility of storing all historical versions of each database record, with the ability to time-travel to a specific version as it was at a specific time. However, this desire to store an increasingly large amount of data brings with it a number of technical challenges [3], some of which I will look to analyse and overcome in this dissertation.

The use cases for immutable datasets and versioned data stores are numerous [4]. At a programming level, immutable datasets allow for an increase in concurrency, as newer data is only added to the dataset, and does not alter any previous data, whilst allowing for multiple data readers and writers. These data structures include concurrent B-trees [5], concurrent tries [6], and log-structured merge trees [7]. Immutable, append-only data stores also allow for simpler characteristics when dealing with distributed transactions, making use of multi-version concurrency control [8] across distributed data stores, allowing for disparate nodes to make changes concurrently to a particular piece of data without affecting any ACID [9] guarantees, and enabling decentralization of distributed cloud databases. In addition to improved concurrency control and simpler distributed characteristics, immutable data stores can also lead to improved performance characteristics, with benefits to caching, memory, and on-disk storage.

The use cases for versioning within databases can also be seen at the application level, where the distinct characteristics can lead to a multitude of benefits applicable across a number of domains. Any application in which there is a need for temporal analysis, spatio-temporal analysis, or the desire to detect changes over time across a dataset, can benefit from versioned, immutable data stores. This includes, but is not limited to, analytics, bioinformatics [10], geo-information systems [11], spatio-temporal wildlife tracking [12], spatio-temporal urban tracking [13], time-series analysis, forecasting, and collaborative systems. In a study conducted in 2012, IBM found that using versioning capabilities provided by a data store, rather than in the application logic, could reduce development time from months to hours [14].

When it comes to the business level benefits of immutable datasets, there is a plethora of use cases and examples. From a security and permissions perspective [15], immutable, versioned data stores allow for the ability to view and analyse which changes have been made to a dataset, and by whom, leading to tamper-proof change detection [16], and system logging of activity [17]. In addition to the security perspective is the historical querying functionality, where temporal queries are able to be run on a dataset as it appeared at a particular point in time, allowing for an accurate audit-trail [18] of all changes made to the dataset - something which is often complex to perform on normal data stores.

There are a number of different methods for storing historical, versioned data, each with its own benefits and negatives, and each usually designed for a specific use-case. Some data stores such as RocksDB [19] are designed for high concurrency read and writes and, as a result, have the added benefit of versioning within the data-structure to achieve this. Other data stores such as Google Spanner [20], CockroachDB [21], and TiKV [22] make use of versioned data in order to support distributed transactions [23]. Other databases such as temporal databases [24] make use of temporal indexes [25] allowing data to be stored and indexed according to when it was inserted into the database (*transaction time*), or when it was valid in the real world (*valid time*). As data is organised by time, it is possible to store and analyse data as it appeared, or will appear, at a particular time in the past, present, or future, but this can result in inefficiencies when it comes to other query types. In a similar manner, time-series based data stores such as InfluxDB [26] use time-structured merge trees [27], to enable storage of data by temporal indexing, putting the onus of responsibility for non-temporal queries onto the user. Finally, only a few data stores - such as Datomic [28], ChronoGraph [29], and ChronoDB [30] - are designed specifically with versioning in mind.

1.1 Motivation for this research

The motivation for this project was drawn from a business project where a need for a temporal, graph database, and a lack of suitable tools on the market, led to the development of a proprietary spatio-temporal document-graph database. In this particular case, the system was needed to analyse the tracking of golf shot data across a golf course, enabling course architects to perform real-time analytical queries against the historical tracking information. As one can see in the simplified image to the right, as the golf course features change year-on-year (and in some specialised cases, on a daily basis) the positioning and metadata concerning the players' shots would no longer be relevant according to how the golf course looks at the current time. As a result, when performing a graph database query which analyses the shot progression along the length of the hole, and filters shots based on the whereabouts of hazards and other features, one needs to be aware of the layout of the course, along with any weather conditions, as it was when the shot was initially tracked.

Further analysis then needs to cross-relate this with the player statistics which were present for each particular shot, at the time that the golf ball was struck. With the added necessity that the query would need to be ad-hoc, with filter parameters, and geolocation boundaries being specified dynamically by the end-user, meant that upfront analysis of the dataset was not possible. The addition of a temporal graph database, enabled vertices and edges within the graph to fetch and 'connect with' other vertices - but instead of seeing the data as it is currently, the query is able to define a specific version to retrieve, resulting in a multi-temporal query across the connected edges of the graph.

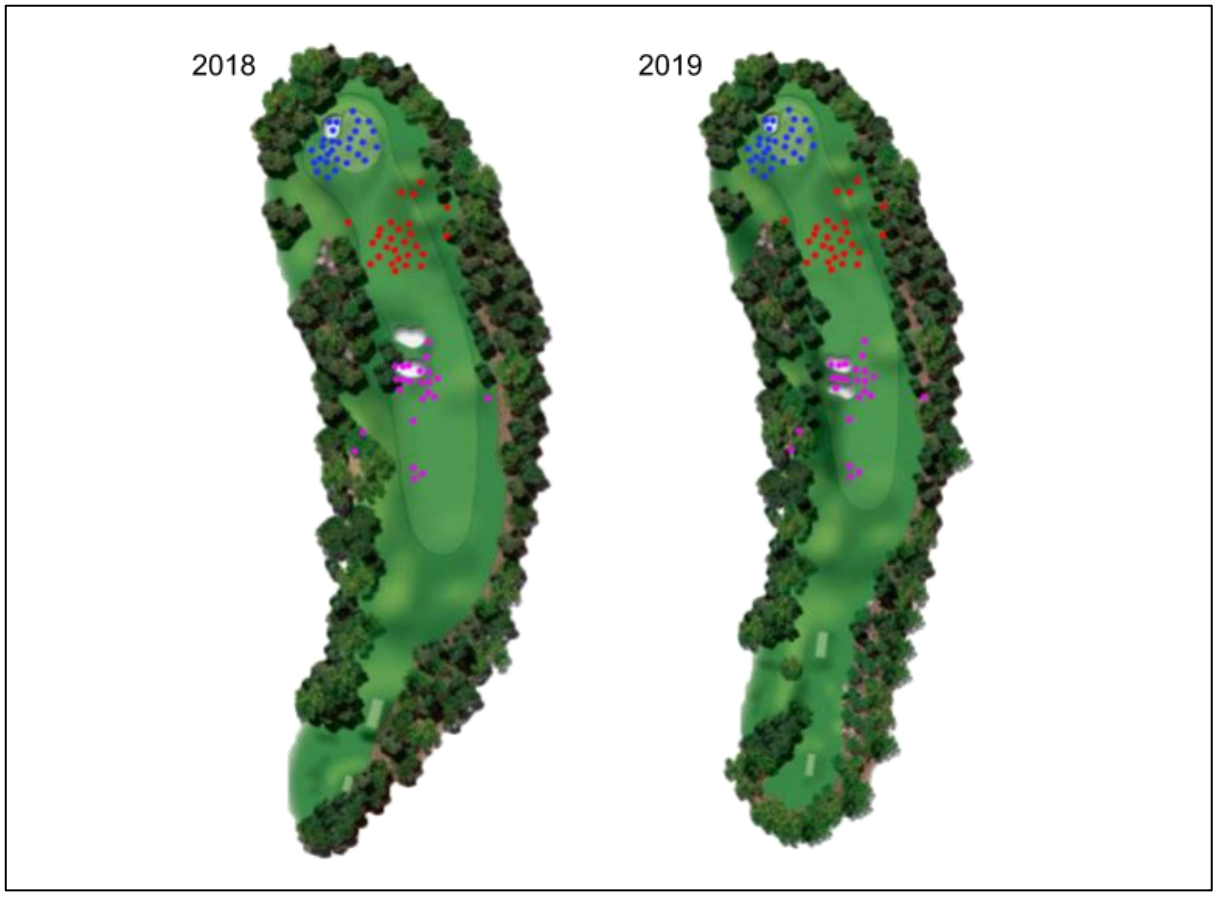

Figure 1 - Golf course analysis over time, according to temporal course data

1.2 Research rationale

Looking at graph databases specifically, the need to query highly connected data (which closely mimics the real world) with direct and indirect reachability queries has grown substantially in recent years. In Figure 2, it is clear to see that the popularity of graph databases has been steadily increasing over the last 6 years. In addition, the interest in time-series databases over the last 2 years has risen similarly, as shown in Figure 3, as developers and systems architects look to different techniques to store IoT time-series event data. Graph databases are able to process connected datasets more efficiently and allow for modelling the data in multiple ways depending on the requirements, whilst enabling real-time querying over large datasets. Augmented with the ability to perform historical analysis of how the graph looked at a particular point in time, analysing which relationships were present between nodes and comparing the data with how it looks today, temporal graph databases offer a number of advantages to many application domains, including bioinformatics, geoinformatics, machine learning, and artificial intelligence. As dataset sizes increase year-on-year, and with the growing popularity of not just NoSQL databases as a whole but with the increasing desire for complex analysis over inter-connected datasets, the need for performant temporal database functionality will grow in parallel.

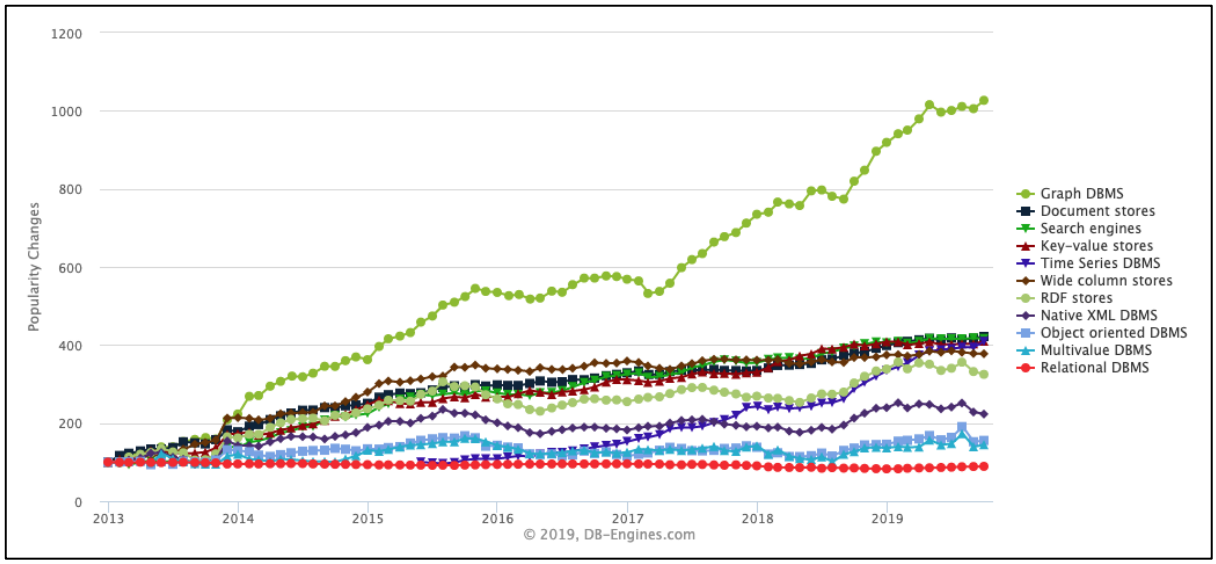

Figure 2 - Popularity increase of different database types over a 7-year period

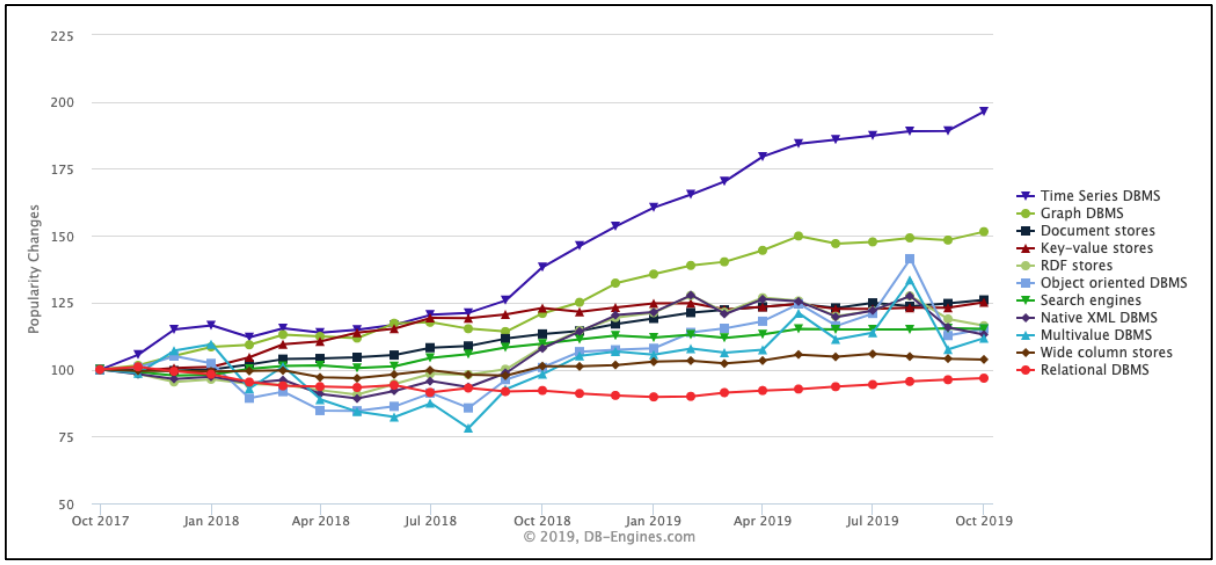

Figure 3 - Popularity increase of different database types from 2017 to 2019

When adding versioning functionality within a data store, a number of technical challenges arise due to the increase in data and because of the underlying data structure design. As a result, the efficiency of these types of stores is usually affected by the amount of versioned data which is present, with some storing the historical data only for a certain amount of time before it is discarded [31], whilst others allow only certain types of queries to be run over the datasets. The proposed approach in this dissertation should allow for historical queries on a key-value store to be as efficient, with regards to storage, performance, and memory-usage, as data stores which store only the latest dataset, regardless of the number of versions stored for each database record. In addition, it should allow for complex traversal queries with the added query dimensions of *valid* time and *transaction* time. This in turn will allow for temporal graph database relationship queries to be effected without any degradation in performance.

1.3 Project objectives

This Software Engineering MSc project aims to achieve a thorough examination and analysis of current approaches to versioned, historical data querying, and will present a multitude of current techniques for both data-storage and versioned data-storage, each with their own benefits and weaknesses. In analysing the different techniques, we will see that the issue of temporal querying within a graph database presents its own difficulties, which are not fully covered by the current approaches.

Throughout the project, I will make use of a number of software engineering methodologies, including requirements engineering, to analyse the use-cases and requirements for the database and data store. I will then proceed to the design and implementation states making use of agile development methodologies, designing the architecture using a service-oriented architecture approach, before implementing the proposed design using small releases, simple design layers, and with a test-driven and benchmarking approach. Using these proven approaches should allow me to realise the objectives defined in the requirements gathering stage of the project.

In completing this dissertation project, I will demonstrate that the proposed approach has related benefits to semantic and interconnected data (document-graph databases), time-series data (temporal databases), and real-world application benefits with regards to real-time analytical and cloud-based data processing, and performs at a similar level to other approaches, whilst offering a larger number of querying functionality abilities. This dissertation draws from techniques and theories taught in Algorithmics (ALG), Semantic Technologies (STC), Concurrent Programming (CPR), Cloud Computing and Big Data (CLO), and Service Oriented Architecture (SOA) modules, along with pre-existing knowledge, and knowledge gained through personal or business research.

1.4 Project evaluation

Analysis of all aspects of the project, including the graph database, distributed database architecture, and embedded data store, is out of scope for this project, due to the inability to closely compare and evaluate the implementations. On the graph database side, there are only a few alternative databases, each of which has a very different feature-set and query characteristics, making it hard to perform equivalent tests, comparisons, or benchmarking. Similarly, with regards to the distributed database side, finding a distributed key-value database which allows for range-based queries, with the added dimension of temporal versioning, is not possible, preventing any meaningful tests and benchmarks at this level.

As a result, I shall be looking at and analysing the implementation of the underlying temporal key-value store, which can be used as the embedded storage layer directly within a graph database node, or within a distributed key-value database node. I will compare this project's implementation with a number of different mature key-value store implementations, many of which are used within large-scale enterprise projects, and most of which do not have the added aspect of versioning implemented within them (which I foresee as being a benefit rather than a cost to the benchmarking of these implementations).

Using the same development language to test alternative solutions should lead to a fair comparison of the benchmarked products, as there should be the same runtime costs and memory usage requirements across all of the different tools against which the project will be compared.

The success of this project, which is analysed towards the end of this dissertation, will depend upon a number of different factors. First, the implementation must work successfully within the context of a graph database, meeting the user and query requirements which I research and set out in Chapter 3. Secondly, the testing and benchmarks should result in roughly equal or improved results compared with the alternatives. Finally, from a functionality perspective, the proposed implementation should be able to be improved upon to allow for uses and developments within future work and within a distributed setting. We shall consider this further work in Chapter 6; however, this is out of scope due to the restricted nature of this project.

1.5 Dissertation outline

In this chapter we have looked at the benefits and use-cases of immutable data and versioning within data stores, and how the domain of connected data stores or graph databases could benefit specifically. In the next chapter I shall look at the different techniques and methods by which versioning can be implemented within the current landscape of key-value data stores and graph databases, with a discussion surrounding the advantages and disadvantages of each method.

Chapter 3 will set out the scenarios and use cases in which an end-user might query the database, setting out the desired database behaviour for each query, and the necessary requirements that the query should meet. We will then go on to look at the necessary requirements of the system which would be needed in order to facilitate these end-user scenarios. In Chapter 4 I will then proceed to suggest an alternative data structure-based approach which, when implemented within a key-value data store, should offer more advanced query functionality, and improved query efficiency, whilst still offering the same features as current methodologies. Next, I will propose a practical implementation of this improved data structure technique, and its application within a versioned, graph data store. Further analysis will look at how this key-value store can be augmented to operate within a distributed architecture.

Following this in Chapter 5 I will analyse the implementation, benchmarking and testing it against alternative data stores, and examining whether the requirements have been met. In Chapter 6 I shall reflect upon the success of the project, detailing the problems which were faced when implementing the proposed solution, analysing the design decisions made, and suggesting alternative approaches or potential improvements that could be made to the work proposed in this dissertation. I shall then put forward a number of potential avenues for future research, and will examine whether any ethical, legal, or professional issues should be considered in the context of this project.

2. Versioned data

In order to meet the goals of this project, one must first understand the different methods by which storing and querying a versioned dataset can be accomplished. In this chapter I shall look at a number of different approaches to managing versioned data, analysing the resulting characteristics of each method, and detailing the key challenges which one would expect to encounter when using these methods when implementing versioning within a graph database. I will then reflect upon these comparisons in the chapters following the implementation.

2.1 Current techniques for data versioning

The differing techniques for versioned data storage, querying, and analysis can be broadly categorized into four groups: those which operate independently of the system which is to be versioned (including backup and archiving); those which operate at a high level and which can be enacted within application code; those which can be applied to the datastore in which the data resides; and those which can be implemented at a lower level within the data structure that holds the dataset. It is important to note that not all of the methods explored in this section are directly comparable to each other.

2.1.1 Data warehousing and archiving

The simplest method for versioning data, which is implemented separately from any database, is to use data warehousing tools to periodically backup and archive the dataset. This approach requires almost no understanding of the application, database, or dataset to setup, and is therefore able to be implemented in any scenario where data needs to be accessed as it existed at a particular point in time, but no complex changes to the software architecture can be made. Using this approach, a full or incremental backup is performed periodically against a database, resulting in an archive of the database as it existed at the time. As we can see in Figure 4, it is then possible to query the data as it was at a specific time, by loading in the respective data archive, and performing queries on the dataset.

Although this method is relatively simple to implement, it leads to a number of disadvantages which result in it being an incorrect fit for our use case. First, depending on the archiving method used, a large amount of duplicate data could be present across the different dataset archives, leading to larger than necessary storage requirements. Secondly, as this method results in periodical exports of the dataset (for example every day, or every hour), the archives will miss granular changes made between the backup times, resulting in a loss to the versioned data, and leading to inaccurate historical queries. Thirdly, the ability to run cross-temporal queries (across differing dataset versions), or changeset analysis (retrieving changes made to a data value over time) is removed, as each query must be run on a separate dataset version, with no knowledge of other versions of the dataset.

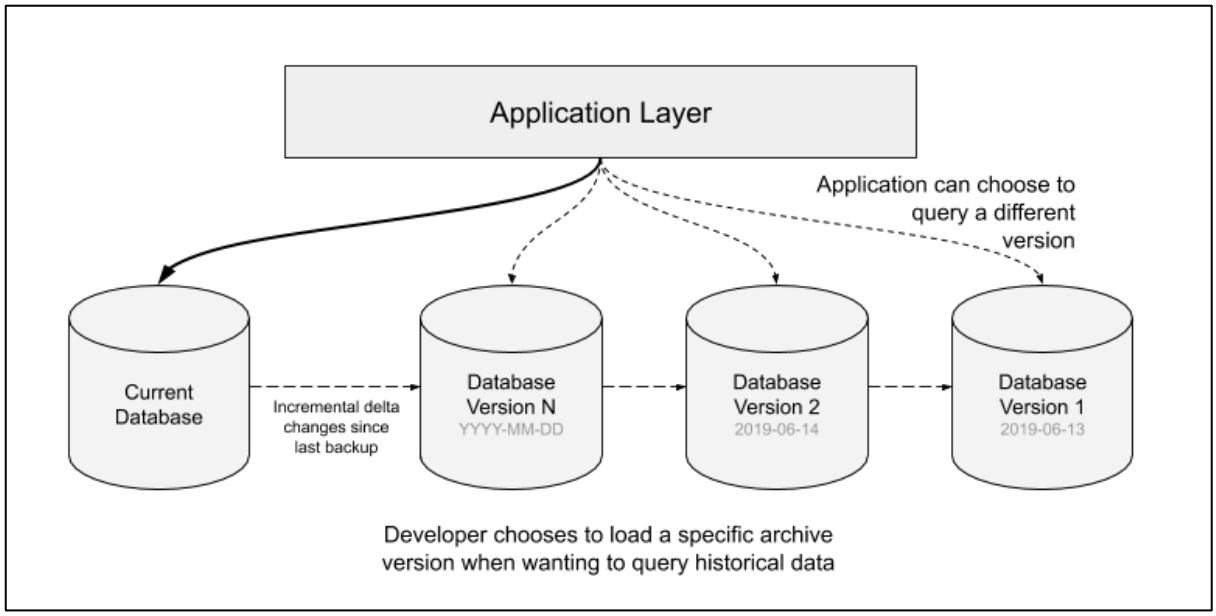

Figure 4 - Versioned queries within the application layer using data archiving

2.1.2 Row-based data versioning

An alternative technique for implementing data versioning is to implement it within the logic at the application level. There are two approaches for application-level data versioning: row-based, and table-based. With a row-based approach, when a change is made to the data, the application layer would create a duplicate version of the row in the database, specifying a validity timestamp for the previous data within the `StartTime` and `EndTime` columns. When a query is made for the current version of a record, it will search for the row where the `EndTime` time is `NULL`. When performing historical queries, a query should filter records based on the `StartTime` and an `EndTime` of each row.

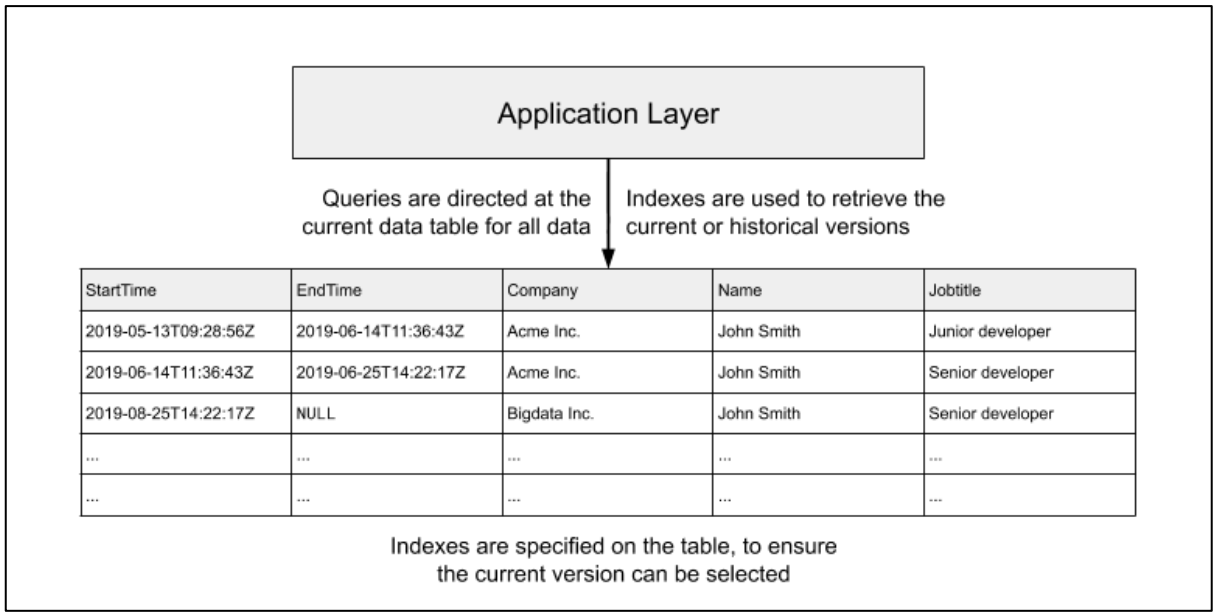

Figure 5 - Row-based data versioning, using columns for specifying versioned data

Although every change to each database record is captured, allowing for queries across all granular changes to the dataset, there are a number of problems with this approach. First, there is now a need to create indexes based on the `StartTime` and `EndTime` columns, affecting the performance of general queries, and alternative indexed queries. Secondly, due to an increase in the amount of data in the table, queries and joins between tables will become less performant, with the database having to filter out a large proportion of rows which are not relevant to the query version. When performing simple queries with only sparse changes to data, this method works satisfactorily. However, when it comes to large datasets with frequent changes, and interconnected relationships, this method is unable to offer the performance that is required.

2.1.3 Table-based data versioning

As with row-based data versioning, table-based versioning works in a similar way, but by storing the duplicate versions of each changed record in a secondary table. Using this alternative approach means that queries on the current dataset are not affected by the number of previous versions. Instead, when an application needs to query the historical data, it will direct queries at the secondary table, as shown in Figure 6. Although an improvement on row-based data versioning, this approach still has a number of problems which lead to it not being a good method for our use case.

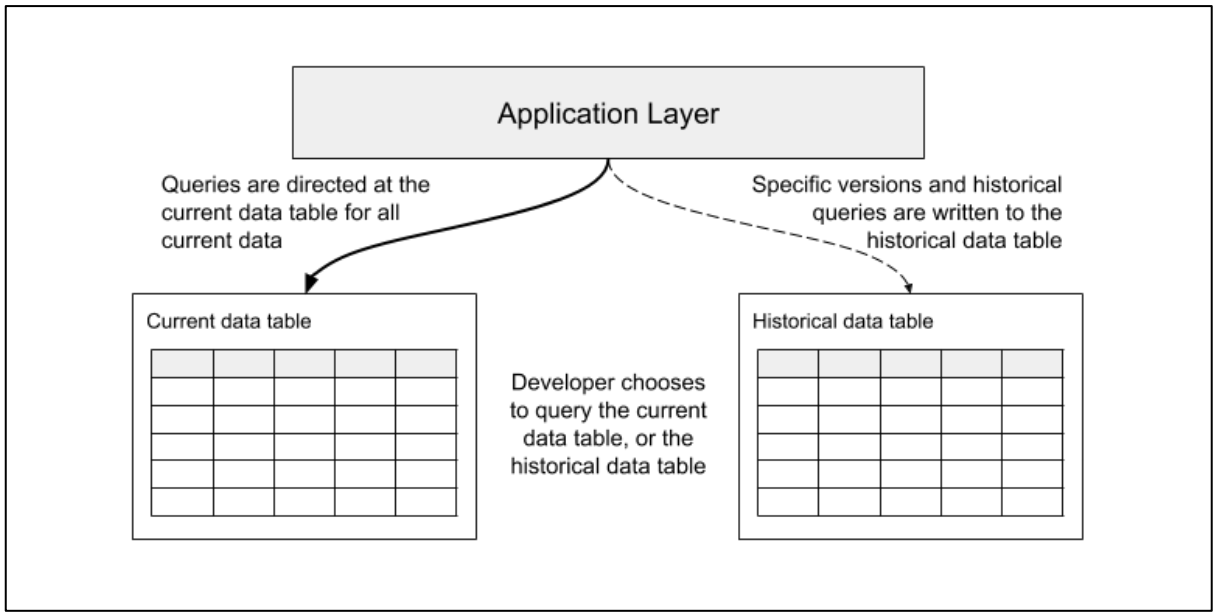

Figure 6 - Table-based data versioning, using a secondary table for versioned data

2.1.4 Historical or temporal tables

Some databases offer temporal tables, introduced in SQL:2011 [32], to ensure that previous versions of a record are kept when modifications are made. Temporal tables are secondary system-versioned database tables which transparently store historical data in a secondary table but allow for all queries to be directed at the primary table. This approach ensures that data changes, and versioned queries, are able to be added more easily to the application layer, yet when it comes to performance, does not improve upon the previous row-based or table-based techniques.

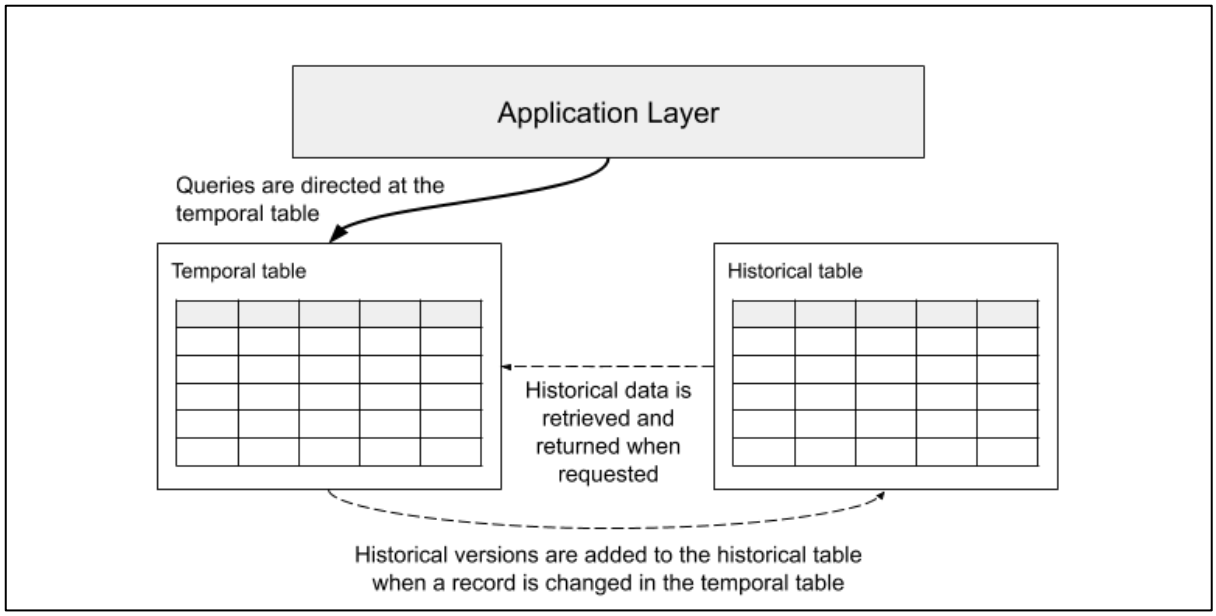

Figure 7 - Data versioning using historical or temporal tables introduced in SQL:2011

As we saw with row-based and table-based queries, as the number of changes to the dataset increase, so too does the query response time. In addition to this, Microsoft SQL Server [33] has a number of limitations when using and querying temporal tables [34], with restrictions on primary keys, foreign keys, and column constraints, affecting query functionality and query response times. As with the previous techniques, temporal tables do not offer the query functionality or the query performance [35] which would be used to combine current and historical data within highly interconnected graph database queries.

2.1.5 Event sourcing

An alternative technique for implementing data versioning is to store data changes as a stream of events. When there is a need to update the database, instead of modifying any data, an *event*, which contains the information on something that has happened in the past, is created and submitted to an event-store [36]. The event-store in turn stores the event in an append-only store, which can lead to improvements with transaction consistency on high-demand applications, and also allows for a fully transparent audit trail of all changes to the data. As shown in Figure 8, the insertion of this event store will then trigger a change in the main database, updating the current 'view' of the data with the change which occurred. In addition, this data can be published to other sources, allowing other materialized views into the data to be updated when an event which concerns them takes place.

In addition to the benefits perceived by having a complete auditable log of changes, this technique can also have an effect on performance, allowing queries on current data to be unaffected by the event logging, and reducing the chances of data update conflicts and concurrent transaction lock attempts [37]. However, although there are some noticeable benefits to this approach over the previous techniques, due to the historical events being stored as an append-only event log, historical queries still need to analyse and filter out a large amount of data, which once again leads to reduced functionality and performance, preventing the use of this approach for historical graph queries.

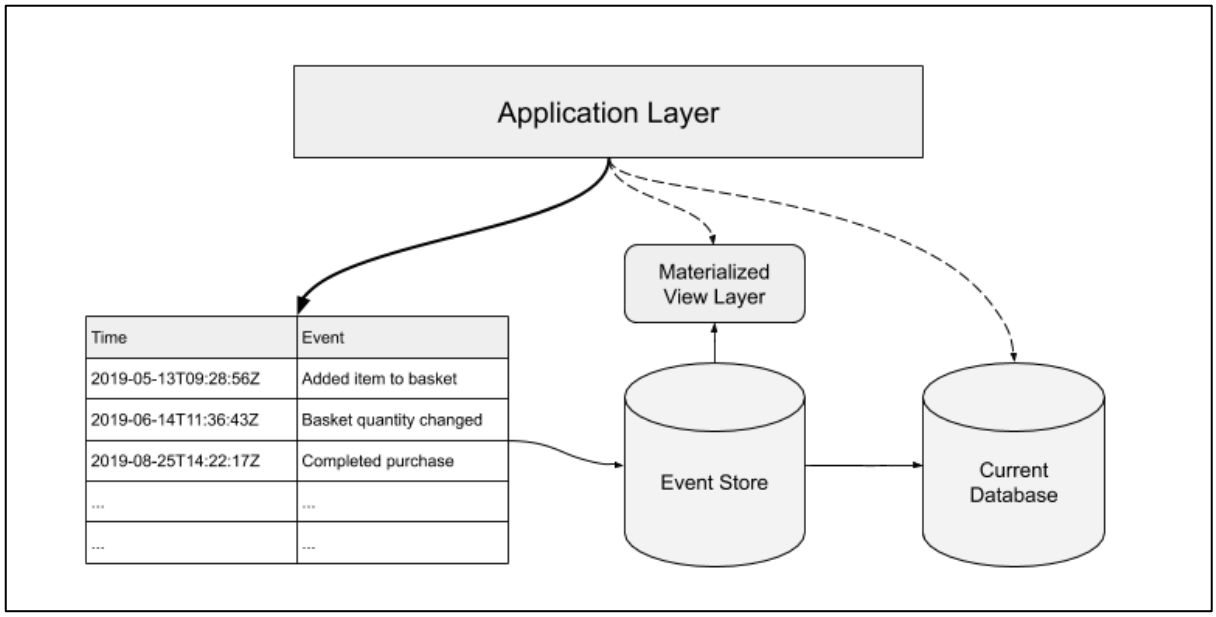

Figure 8 - Capturing data changes using event-sourcing methodologies

2.1.6 Time-series databases

Time-series databases store time-series event data indexed by timestamp. Instead of a record existing only once within the database, each insertion into the database creates a new record, optimised for storage by the time of entry. Typically, time-series databases are similar to an event log, discouraging or preventing modifications of data once stored. Since these database types are designed for running analytical queries of changing data over time, they are not optimised for accessing or querying the data using alternative parameters, suiting large scale temporal analysis, rather than selecting changes to a particular record over time. As a result of this, although we may draw on techniques used when implementing time-series databases, their specific use as a platform for our use case is not appropriate.

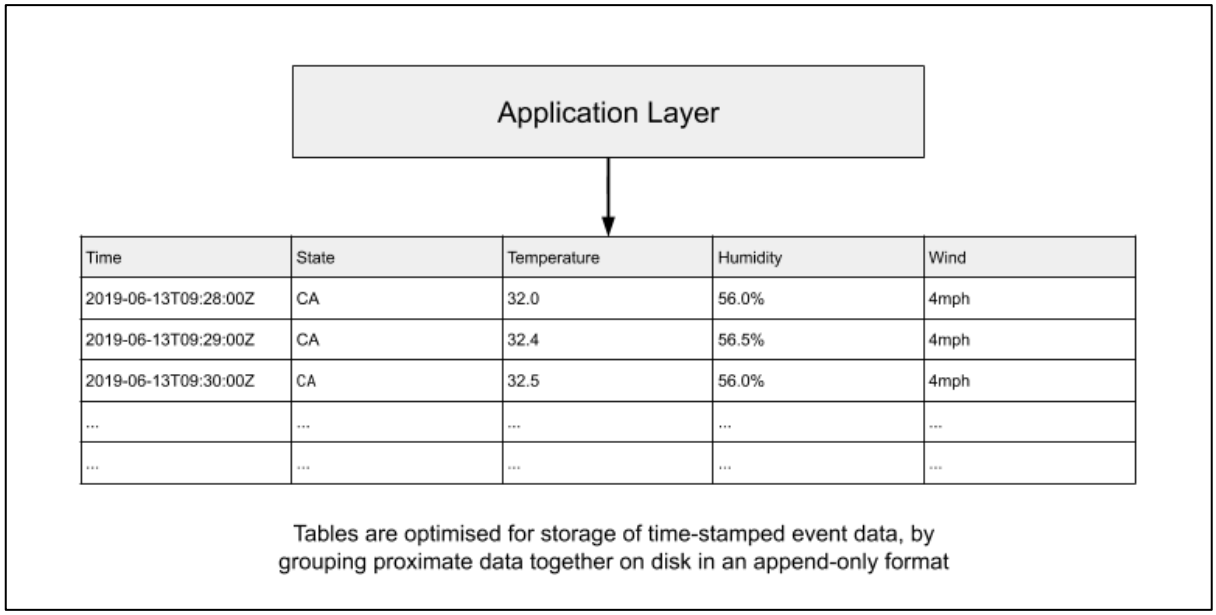

Figure 9 - Time-stamped data in a time-series database table

2.1.7 Time-based versioned graphs

When it comes to graph databases, storing the historical versioning of data within the graph itself has its own challenges [38]. Figure 8 shows the representation of a simple dataset in a graph database, where all vertices and edges are assigned a `start` and `end` property. When the data is current, the `end` property will be `NULL`, and relationships can be traversed normally. As we can see in Figure 11, when an edge or vertex is altered, the `end` property is time-stamped, and a duplicate node is created. When traversing over the graph, edges with an `end` property that does not satisfy the version timestamp are ignored. However, each change to the graph increases the number of vertices and edges and vertices, leading to an increase in the complexity of the graph. This added complexity, along with the eventual scalability issues and performance problems when running queries across the graph, results in this technique being unsuitable for our use case.

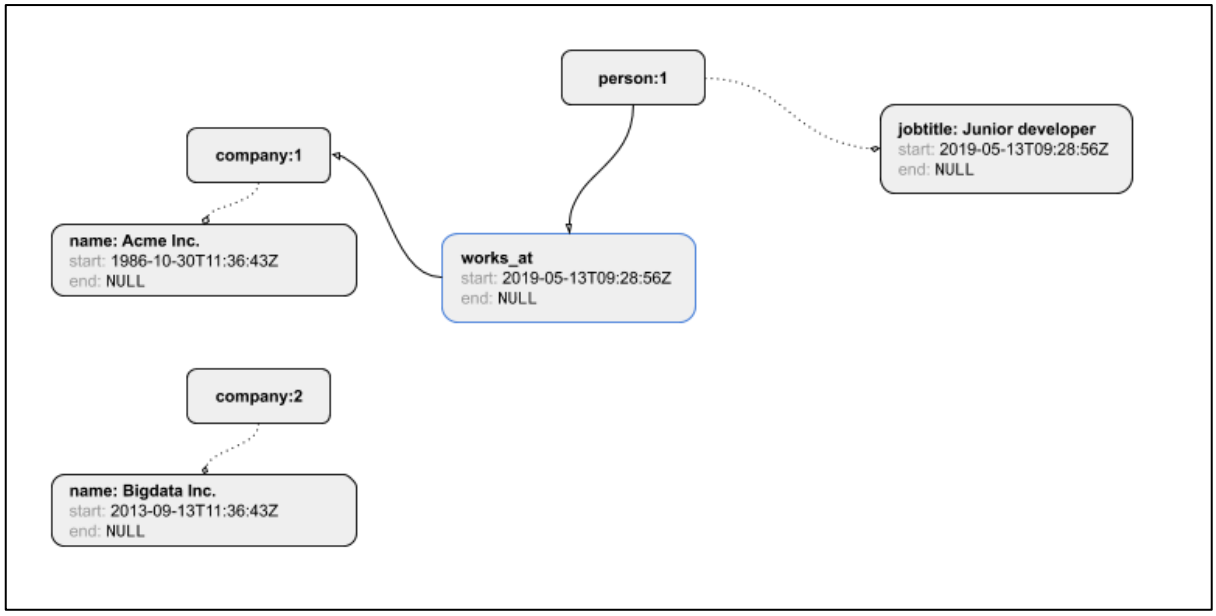

Figure 10 - Time-based versioned graphs, before making a change to the graph

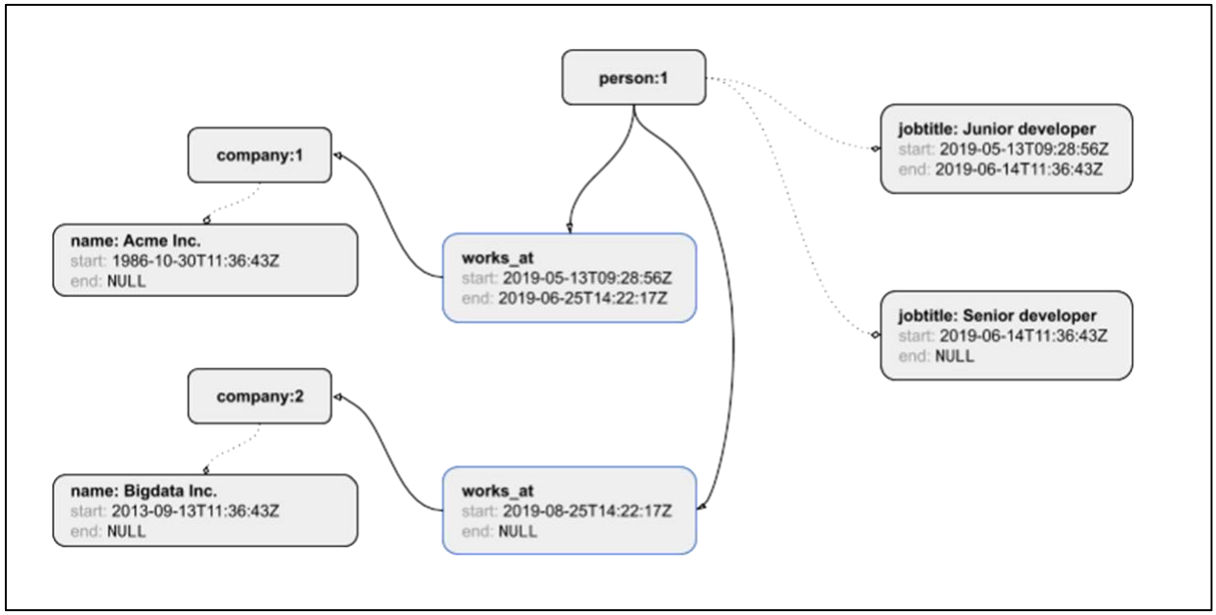

Figure 11 - Time based versioned graphs, after making a change to the graph

2.1.8 Key-value based multi-version indexing

In order to enable multi-version concurrency for concurrent writes to the data structure, and for high-availability of writes in a distributed environment, a data structure can be augmented with a time or version index parameter, which is used for detecting more recent changes to a key once a transaction is committed. In current key-value store implementations, such as LMDB [39], RocksDB [19], BerkeleyDB [40], and BadgerDB [31], the version is stored within the key-value store, rather than within the data structure, by concatenating the versions with the keys. When querying for a single key, or multiple keys, the database will filter out keys which contain a previous timestamp. In this way, the database (or the data nodes in a distributed database environment) can fetch a key at a particular version and can ensure that no other version has since been inserted by another node.

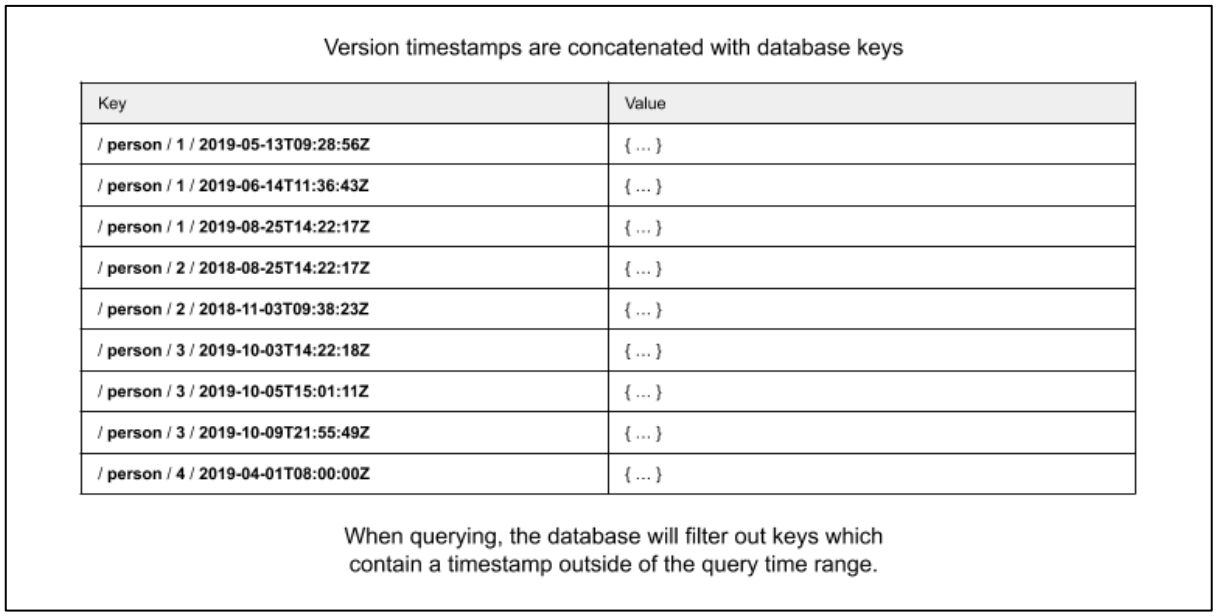

Figure 12 - Key-value based index versioning, showing four keys and multiple versions

We can see in Figure 12, how the keys in a key-value data store might look when multiple versions of each key are stored. Fetching a specific record, at a specific version, is as simple as concatenating the key with the version timestamp and retrieving the value at the respective node. For range queries, the datastore must filter out (or skip over) versions which are not relevant for the query, selecting the preceding version to the query timestamp.

This approach works well for MVCC requirements where a key-value version must remain available in the data store for the maximum amount of time needed (as specified by the database) to prevent slow-running or disparate transactions from committing correctly. In most cases these versioned values can be garbage collected once transactions that relied on them have stopped running. However, storing the versioned values for longer periods of time will lead to negative performance trends for lookup queries, and additional performance problems with range queries, as an increasing number of key-value versions will need to be filtered out as the data is iterated over.

In *Scalable versioning for key-value stores* [41], Haeusler presents an idea where the key-value store is augmented with timestamp information in order to support key-based temporal versioning within a graph, using a B-tree data structure to store the data. Similarly, in *Scalable time-versioning support for property graph databases* [42], Vijitbenjaronk et al. make use of LMDB [39], an underlying key-value store implemented using a copy-on-write B-tree, to add support for temporal graphs. With these approaches, the distribution of keys with temporal versions does not affect the performance of the data store with regards to lookup queries, whether there is a single key with many versions, or many keys with many different versions - with the complexity remaining at the optimal `O(log n)`. However, with regards to range queries, the search time will increase simultaneously with the number of versions for each key, both for current versions and on historical versions, requiring an added complexity of `O(v)`, where `v` is the number of versions for each record. The characteristics of this approach are set out in Figure 13. As a result of these performance characteristics for range queries, this method does not suit the use cases or requirements set out in Chapter 3.

Increase in versions	Performance of querying current data, depends on the total number of versions stored for each key
Point query on current data	`O(log n)` complexity
Point query on historical data	`O(log n)` complexity
Range query on current data	Added complexity of `O(v)` for each record, where `v` is the number of versions which need to be skipped to find the correct version
Range query on historical data	Added complexity of `O(v)` for each record, where `v` is the number of versions which need to be skipped to find the correct version

Figure 13 - Performance characteristics of the proposed solution in [42]

2.1.9 Partially persistent and fully persistent data structures

Persistent data structures [43] are often used for immutability and for solving the problem of concurrency. They are often used within databases to enable the ability to perform concurrent writes on a data structure, while other actors are reading from the same data structure. While actors are reading from the data structure, a writer can modify the tree, using pointer switching, without affecting the consistent view that the readers have into the data.

Once a persistent data structure has been altered, a new root node is returned, with which the data structure can be queried, presenting a view into the dataset according to the latest change, whilst using unaltered nodes from the previous version of the tree. Due to this, it is possible to store all of the different root nodes, allowing for all previous versions to be queried, and enabling storage of versioned data, whilst allowing efficient [44] queries over all versions of the dataset. Figure 14 shows an example of a Radix tree [45] (sometimes referred to as a Patricia tree) being altered, with the blue nodes showing the new changes to the tree, whilst nodes which aren't changed are referenced from the new data structure.

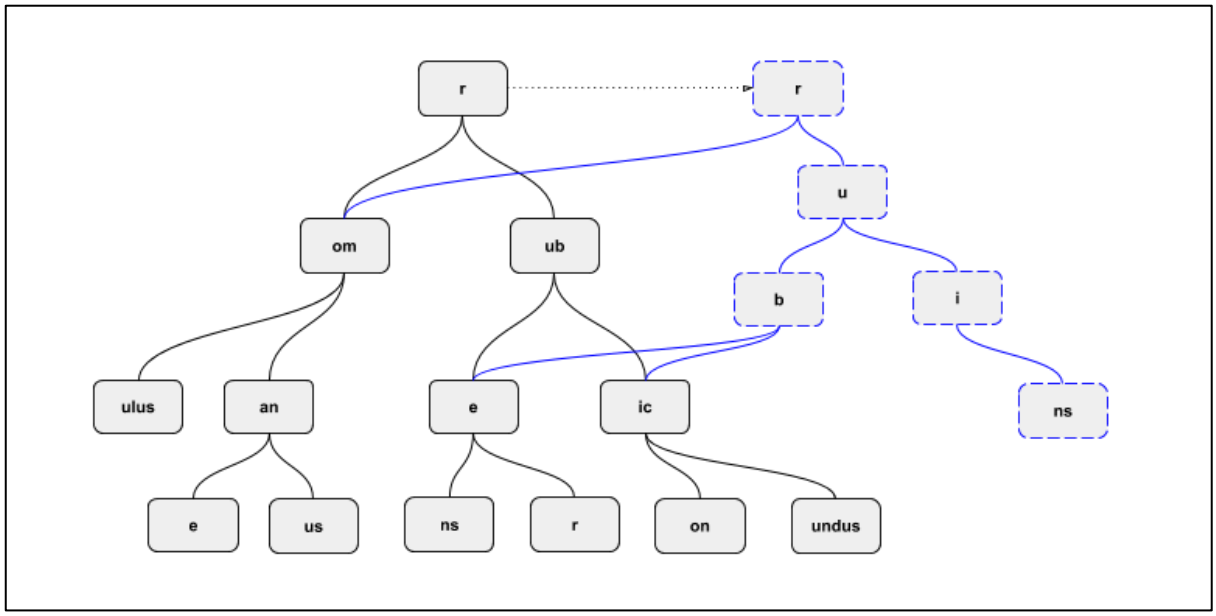

Figure 14 - Persistent Radix tree, showing node copying after adding the word 'ruins'

This method of versioning can introduce a few performance disadvantages when compared to a mutable data structure, where changes are performed on the nodes and paths in-place - as the path up to the root node would first need to be copied before returning the new data structure. If being able to access and query all versions of the data structure is a requirement, there are also issues which one needs to be aware of when storing and accessing the different root nodes of the data structure, as it would then be necessary to store these root nodes in their own tree rather than an array, so that accessing a previous version is itself an `O(log n)`, not a `O(n)` operation. As a way of mitigating this issue, copy-on-write data stores such as LMDB [39], WiredTiger [46], and BoltDB [47] prevent or limit the possibility of rolling back to a previous version, storing only a limited number of prior versions as a way of supporting long-lived transactions and delayed commits. However, due to the benefits of immutability, concurrency, and versioning, this approach merits further investigation, and I shall build upon these ideas later in the project.

2.1.10 Augmented data structure

The final technique which I will look at in this chapter is the augmentation of a data structure so that version information can be stored within the structure itself. In *Transaction time support inside a database engine* [48], Lomet et al. show how transaction-time support can be added to SQLServer, implementing the versioning within the data-structure persistent storage pages, using a pointer based on-disk linked list, which allows for previous versions to be retrieved by traversing the version chain. This method allows for historical data to be located alongside current data, and enables large histories to be maintained, whilst at the same time remaining transparent to the overlying data store. This technique is not necessarily limited to certain types of data structures, allowing different types to be used together to achieve the desired aims. This method for versioning is an interesting approach, aspects of which I shall build upon in my proposed implementation.

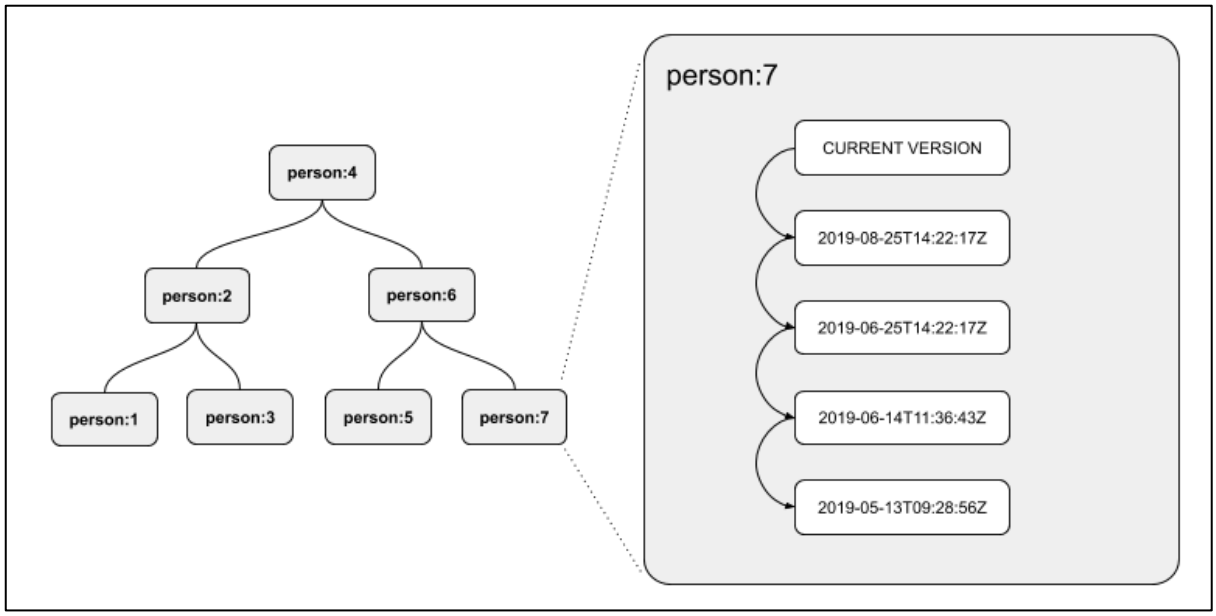

Figure 15 - B-tree data structure with linked-list for node versioning

As one can see in Figure 16, the characteristics are similar to, but slightly different from, the key-value based indexing approach that I analysed in Section 2.1.8. In this approach, lookup and range queries on current data are not affected by the number of versions stored for each record, instead remaining at the optimal `O(log n)` complexity. However, when performing a lookup or range query on historical data, each version chain must be followed from the most recent version, to the desired version. As a result, the performance of these queries degrade as the number of versions is increased, requiring an added complexity of `O(v)`, where `v` is the number of versions for each record, and degrading the further back in time that the query needs to analyse. This performance is additionally affected as previous versions may be located within a separate 'page' on disk, needing extra traversals to fetch the desired version.

Increase in versions	Performance of querying current data, is not affected by the total number of versions stored for each key
Point query on current data	`O(log n)` complexity
Point query on historical data	`O(log n + v)` complexity, where `v` is the number of previous versions which need to be traversed to find the correct version
Range query on current data	As efficient as without versioning
Range query on historical data	Added complexity of `O(v)` for each record, where `v` is the number of previous versions which need to be traversed to find the correct version

Figure 16 - Performance characteristics of the proposed solution in [44]

2.2 Discussion

There are a number of different techniques with which to store, analyse, and query versioned datasets, each of which has its own specific set of advantages and disadvantages. After analysing the characteristics of the different approaches above, it became apparent that a multi-technique approach would need to be used to meet the requirements of this project. As a result, I shall extend upon ideas found in several of the approaches, incorporating ideas from event sourcing, and time-series databases, and building upon techniques used for building persistent and augmented data structures. The focus of this project shall be to amalgamate these ideas, improving upon some of the techniques discussed in order to meet the requirements set out in the following chapter.

In the following chapters I shall develop the idea of keeping the key-value storage layer separate from the graph database querying layer, and will seek to improve upon the method of augmenting the data structure to allow for versioning within the storage layer, while attempting to improve upon the querying functionality and implementation efficiency, so that the amount of data does not affect the throughput speed of data retrieval.

3. Requirements and behaviour

In this chapter I shall look at a number of scenarios in which an end-user might query the database, setting out the desired database behaviour for each query, and the necessary requirements that the query should meet. I shall then look specifically at both the functional and non-functional requirements of the key-value store, before taking a closer look at the necessary query functionality and data access requirements which would be needed in order to facilitate these end-user use cases.

3.1 Scenario use cases

Historical graph database fraud detection
The first use case concerns the retrieval and analysis of data within a graph database, with a specific desire to query the dataset as it was at a particular historical point in time. In this scenario, an end-user wants to analyse a dataset containing financial share trading data, in order to detect patterns which would lead to fraud detection. In this particular case, a recent uncovering of a fraudulent action by a financier in the past, has led to the desire to analyse with whom they had communications and connections, at the time of the event. As the graph reflects current connections and transactions, it has resulted in a constantly updated graph which reflects the real-world but does not show information as it appeared at a time in the past. As a result, the database user wants to run queries across the graph using the vertices, edges, and properties which were current at the time of the event. By using a temporal graph database, the end-user would be able to specify the timestamp version for the desired window into the dataset, with the ability to query and analyse the data with no other application-side modifications.

Golf course architecture analysis
The second use case concerns the scenario detail in Section 1.1 of this dissertation. In this use case, a golf course architect wishes to analyse the areas of a golf course which need to be improved so that more proficient players find the course more difficult, while less advanced players are able to play the course without being affected by the professional-level hazards. In this case the architect wants to analyse the yearly-changing position of a group of bunkers, detecting how it affects the playing style of each level of golfers. In order to effect this analysis, the architect wants to filter the tracked data according to the design and layout of the course as it was when the shot was tracked, without having to run a separate query for every time that the golf course changed (which in some cases can be quite often). Instead, they wish to run a single query, filtering the data based on the temporal changes of the course, and depending on the time that each individual shot was taken.

3.2 Graph database operation behaviour

In order to set out the requirements for the proposed implementation, one needs to look first at the behaviour of the graph database, and how it interacts with the key-value data store in order to insert, modify, and retrieve data. This will enable an analysis of how a query is formed in the graph database query processing layer, and how that layer would, in turn, query the key-value store using transactions. As this is a topic unto itself, we will briefly look at a few simplified queries, demonstrating how the query would be converted into key-value store operations. To begin with, a user would submit their query to the graph database in the form of a text-based SQL-like query language. This query is then parsed and converted into a set of **`insert`**, **`select`**, or **`delete`** operations, for either single key-value items or ranges. Below are a number of queries, each with a brief explanation and a pseudo-code summary of key-value operations needed to fulfil the query. We shall look at the key-value store API in greater detail in Section 4.4.

A single record select query

This query would select a single record by its ID from the key-value store and would return the value to the end user. The query would default to using the current system time and would therefore fetch the version which is less than or equal to the current timestamp.

```
SELECT * FROM person:bmrq6o;
```

The key-value store operation is a simple get request, which would return a single record by searching within the key-value store using the path provided.

```
db.Get(
        time.Now(),
        "/db/person/bmrq6o",
);
```

A multiple record select query

This query would select the range of records in the ‘person’ table from the key-value store, iterating over each record, and returning the values to the end user. The query would default to using the current system time and would therefore fetch each record version which is less than or equal to the current timestamp.

```
SELECT * FROM person;
```

The key-value store operation is a get range request, which would return all keys which are greater than the first key, and less than the second key within the key-value store.

```
db.GetR(
        time.Now(),
        "/db/person/0x01",
        "/db/person/0xff",
);
```

A single record current-time update query

This query would update a single record in the key-value store and would return any previous value to the end user. If no time is specified, then the query would default to using the current system time - inserting the data with a current version timestamp.

```
UPDATE person:bmrq6o SET name = "John Smith";
```

The key-value store operation is a simple put request, which would enable inserting a value at the specified key and version.

```
db.Put(
        time.Now(),
        "/db/person/bmrq6o",
        "{name:'John Smith'}",
);
```

A single record historical-time update query

This query would update a single record in the key-value store and would return any previous value to the end user. If a version time is specified along with the data insertion, then the query would use this timestamp when inserting the version into the database.

```
UPDATE person:bmrq6o SET name = "John" VERSION "2019-07-27T13:00:00Z";
```

In this instance, the key-value store operation is a simple put request once again. However, the first argument has a specific timestamp for use as the version.

```
db.Put(
        "2019-07-27T13:00:00Z",
        "/db/person/bmrq6o",
        "{name:John}",
);
```

A graph traversal query from a single graph vertex

This query would enable us to fetch any outgoing edges for a specific vertex in the graph. In this instance, we are fetching all people who were liked by the person record from which the graph traversal originates, at the time specified in the query. This is an example of retrieving values as they were stored in the key-value store using *valid time*.

```
SELECT ->like->person FROM person:bmrq6o VERSION "2019-07-27T13:00:00Z";
```

Here the key-value store operation is another get range request, using the current timestamp. In this case the key paths are identifying a specific range of keys to iterate over, in order to retrieve the values for the graph database edges.

```
db.GetR(
        "2019-07-27T13:00:00Z",
        "/db/person/bmrq6o/out/like/0x01",
        "/db/person/bmrq6o/out/like/0xff",
);
```

A transaction time query for multiple records

The final query example that we shall look at specifies a transaction at a particular version. This enables us to search the records in the key-value store using *transaction time*. In this query we are searching for all golf shots, where the previous shot was played from inside a bunker. Importantly, this query will fetch the data as it was visible in the database at the specified version timestamp, independent of any data changes made since that time.

```
BEGIN TRANSACTION AT "2019-07-27T13:00:00Z";
LET $bunkers = (SELECT * FROM course_features WHERE type = "bunker");
SELECT * FROM golf_shot WHERE ->previous INSIDE $bunkers;
CANCEL;
```

In this scenario, a read-only transaction is created at the specified *transaction time* version. This transaction would access the key-value store using a specific historical commit, with access (in a read-only way) to data as it was at the time. As a result, the range selection method can specify a current timestamp (to get the latest values), without it affecting the query. After the data has been retrieved and processed, the transaction is cancelled in order to free up any resources.

```
db.BeginAt("2019-07-27T13:00:00Z");
db.GetR(
        time.Now(),
        "/db/person/golf_shot/0x01",
        "/db/person/golf_shot/0xff",
);
db.Cancel();
```

3.3 Data store requirements

In implementing the key-value data store, one must adhere to a set of functional and non-functional requirements in order to meet the needs of the use-cases, and furthermore to ensure that the key-value store can safely be used in environments where minimum data guarantees are necessary.

Requirement 1: The primary requirement is for the database to guarantee a minimum level of usability and validity, by supporting and enforcing ACID [9] (Atomicity, Consistency, Isolation, and Durability) properties. As a consequence of this requirement, the database must enforce the use of transactions within which operations to the data store can be effected - allowing multiple select, update, or delete statements which either succeed and commit as a whole, or fail without affecting the database. The second property, consistency, must be ensured so that the key-value store is not left in a damaged state if a failure event were to occur. With regards to isolation, the database should support multiple concurrent readers, and writer, guaranteeing that the changes being made to the dataset by the writer are not visible to the readers. Finally, the issue of durability is important, as committed modifications to the dataset should not be lost in the event of a system failure. As a consequence, contrary to the implementation of some key-value stores such as LevelDB [49], the data store must support multi-key, multi-operation, application-defined transactions.

Requirement 2: As an extension of *Requirement 1*, the data store should be able to store its data to disk after every commit, and in addition must have the ability to stream the total dataset to a file, without affecting writes or other readers (backup).

Requirement 3: By extending *Requirement 2*, the key-value store should be able to start up with an initial dataset, by loading and processing a data storage / backup file. This would allow the database to be stopped and restarted, continuing where it left off.

Requirement 4: The database must be performant for read-heavy analysis-based workloads, and must also support a write-heavy environment, so that it can be used for storing time-series event data. In addition, the performance should not deteriorate as the number of versions increases.

Requirement 5: As the amount of data that is generated, collected, and analysed is increasing year-on-year, and with the added storage requirements needed when storing versioned, immutable data, an important requirement for the data store is to be able to handle large datasets. Even though it is now possible to utilize servers with terabytes of RAM, the database should be able to store a greater amount of data than will fit into a single server node's memory.

3.4 Data access requirements

In order to meet the requirements of the queries specified in the scenario use cases set out in Section 3.1, the data store must support the following requirements with regards to data storage, access, and retrieval.

Requirement 6: As with all databases, the primary data access requirement is to be able to query for a particular record as it is stored in the data store at its **current version** `version(0)`. This requirement is augmented with the need to query a record as it appeared at a **particular version** in the past `version(timestamp)`.

Requirement 7: In addition to performing point queries for a particular version, the data store must support historical queries in order to retrieve **all versions** for a record, or **selected versions** for a record which were applied between `version(start, finish)`.

Requirement 8: By extending *Requirement 1*, when presented with a range query for records which lie within `range(min, max)` or are prefixed by `prefix(key)`, the data store must be able to retrieve each record within the range, fetching the record at its **current version** `version(0)`, or as it appeared at a **particular version** in the past `version(timestamp)`.

Requirement 9: By extending *Requirement 2*, when presented with a range query for records which lie within `range(min, max)` or are prefixed by `prefix(key)`, the data store must be able to retrieve **all versions** for each record, or **selected versions** for each record which were applied between `version(start, finish)`.

Requirement 10: In order to operate effectively as a data store layer within a database, the data store must also support **insertion** queries. When inserting, the data store should support inserting at the **current version** `version(0)`, or inserting a historical version at a **particular version** in the past `version(timestamp)`.

Requirement 11: Next is the ability to perform **deletion** queries on the data store. This should enable a record to be removed from the data store both in an immutable way (so we know it is now deleted but we can still view previous versions), and in a mutable way (so all versions of the record are removed completely from the data store). One should be able to remove both single records, and records which lie within `range(min, max)` or are prefixed by `prefix(key)`.

Requirement 12: The time taken to retrieve a historical version of a database record must not increase as the number of stored versions increase. Additionally, the time taken to retrieve a single record must not be affected by the number of record versions.

Requirement 13: A range query over a number of keys must not be affected by the number of versions stored for each key in the range. As the amount of data increases, and as the total number of immutable versions for each record increases, the speed of performing a range query over the current or a historical version should not be affected.

Requirement 14: The final requirement for data access fulfils a need for accessing the dataset at a version timestamp set by the system, rather than version identifiers specified by a client at modification time. This would allow an end-user to query the dataset by *transaction time*, seeing the dataset as it existed at that time, regardless of the historical or future version timestamps set on each value.

In conclusion it is apparent that there are a number of requirements which need to be addressed at differing levels within the database. Some requirements are appurtenant to the end-user, and directly relate to the functionality by which the database is queried. Other functionality is more obscure, affecting these initial requirements, but not necessarily directly discernible to an end-user. In the next chapter I will describe the architecture of a system which will look to address these requirements.

4. Architecture and design

In this chapter I will look at the general architecture needed to meet the requirements specified in Chapter 3 and will proceed to design a temporal key-value store, which would be the key part of a temporal graph database. Whilst I will be predominantly designing and implementing a single-node database setup, I shall also take into account the needs of a distributed database system, being mindful of theoretical requirements and implications within a distributed key-value store. This should allow for future work to apply the proposed implementation within the larger context of a distributed data store, allowing for much greater data storage and processing possibilities – a necessity, especially with the growing needs of data storage [50].

4.1 High-level design overview

As I am designing the database for use in both a single-node and a multi-node setup, I will design the system as a series of layers, each of which can act independently of each other. This approach leads to a number of advantages when it comes to the design, development, and operation of each layer. First of all, when looking at the database as a single-node setup, a multi-layered architecture can lead to a simplified design and development process, as each layer can be designed for a particular task, resulting in each layer having its own responsibility for just a small subset of the overall system. In addition, a multi-layered architecture can lead to improved reasoning, as each layer can be developed independently from each other, each with its own specific set of requirements, test cases, and release schedules.

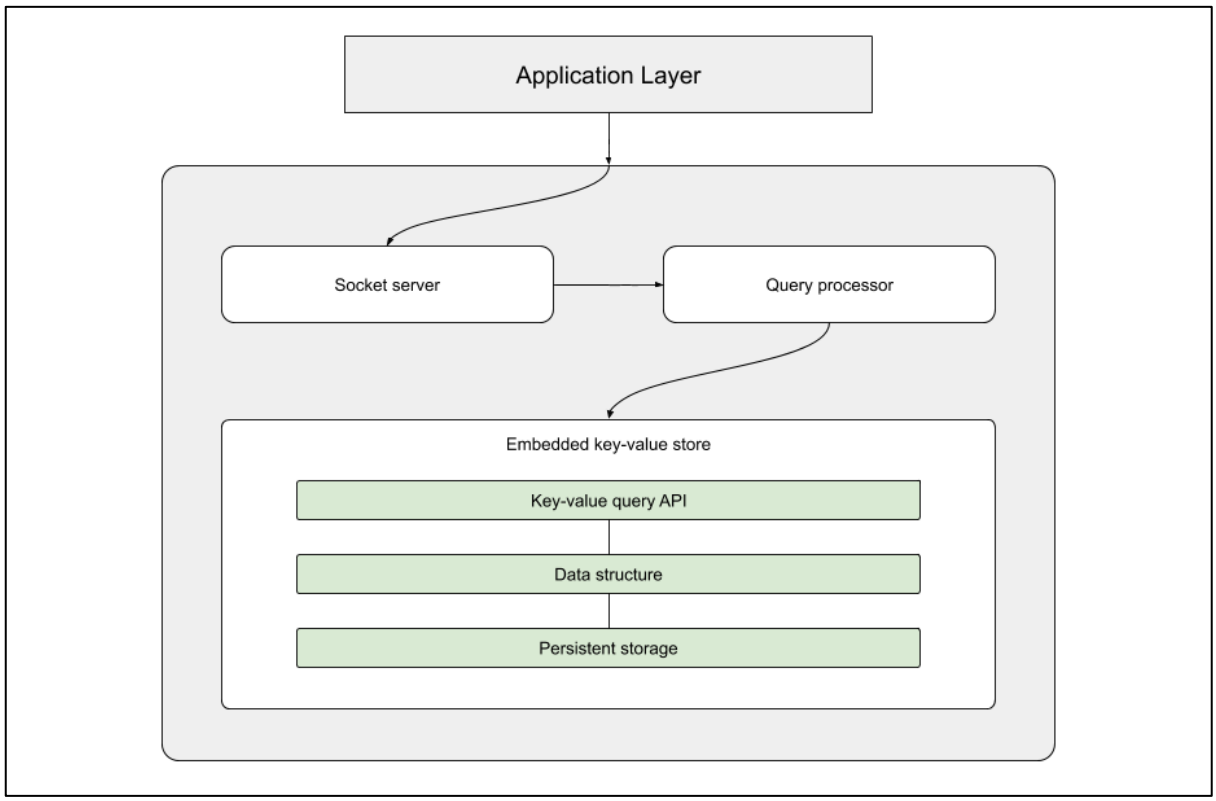

Figure 17 - Embedded key-value store in a single-node database

Figure 17 shows an example of how the embedded key-value store would fit when being used as the storage interface for a fully featured graph database. In this scenario, the graph database makes use of the embedded key-value store by implementing it within its programme. The graph database layer accepts queries from an end-user client via a TCP socket or HTTP request, before parsing the query and analysing the most efficient method possible for retrieving the desired dataset from the key-value store in order to satisfy the query, and then executes the queries, before returning the processed results back to the client.

The key-value store itself is separated into three distinct parts: the key-value query layer, the in-memory data structure, and the persistent storage layer. The key-value store query layer abstracts away the complexities of the data structure, with a set API, enabling data to be inserted, deleted, retrieved, or iterated over in specific ways. This layer operates in a similar method to other key-value store interfaces including BadgerDB [31], BoltDB [47], RocksDB [19], LMDB [39], BerkeleyDB [40], and LevelDB [49], except for one key difference - the addition of a temporal versioning identifier which allows for inserting, deleting, retrieving, or iterating over the data as it was at a specific version or point in time. In addition, the embedded key-value store is responsible for storing data to persistent storage (HDD, SSD, or Network Drives), and for managing file compression and compactions. The embedded key-value store is also directly responsible for handling any transactions when reading and writing to the underlying data structure, employing multi-version concurrency control in order to offer Atomicity, Consistency, Isolation, and Durability properties [9]. Finally, if desired, the key-value store is responsible for encryption of the data before it is stored to persistent storage.

The final layer, which sits within the embedded key-value data store and defines the resulting characteristics and functionality of the data store itself, is the data structure. In order to meet the end-user requirements set out in Chapter 3, I shall propose a data structure called the *Temporally Augmented Radix Tree*, with certain characteristics which will enable point and range queries over current and historical datasets without any performance degradation for either query type as the number of versions within each node in the data store increases.

4.1.1 Multi-node distributed database setup

When it comes to a distributed setup, a multi-layered approach is essential, and can lead to further benefits. In this scenario, the storage, consensus, and querying layers of the distributed database are each split into a separate service, which leads to an improved operational process, where each layer is able to be deployed, managed, monitored, and scaled independently from each other. Nodes for processing queries can operate independently from nodes which store data, with the ability to scale up and down depending on user activity or query throughput, removing the need to re-replicate data across storage nodes when query spikes occur. In addition, storage nodes can scale up and down depending on the storage requirements and data replication factors. This stateless separation results in a highly available and highly scalable architecture.

In Figure 18, we can see how the embedded key-value store sits within the high-level architecture overview of the distributed key-value data store, and how the graph database nodes perform queries over the distributed dataset. Once the graph database node has parsed the query from the client, instead of sending queries to an embedded data store, the node connects to the key-value store across a network interface, using the same API functionality that is available with the embedded option. In a similar manner to the single-node scenario, the graph database nodes are now stateless, only storing data in memory during the course of a client transaction, before returning data to the client.

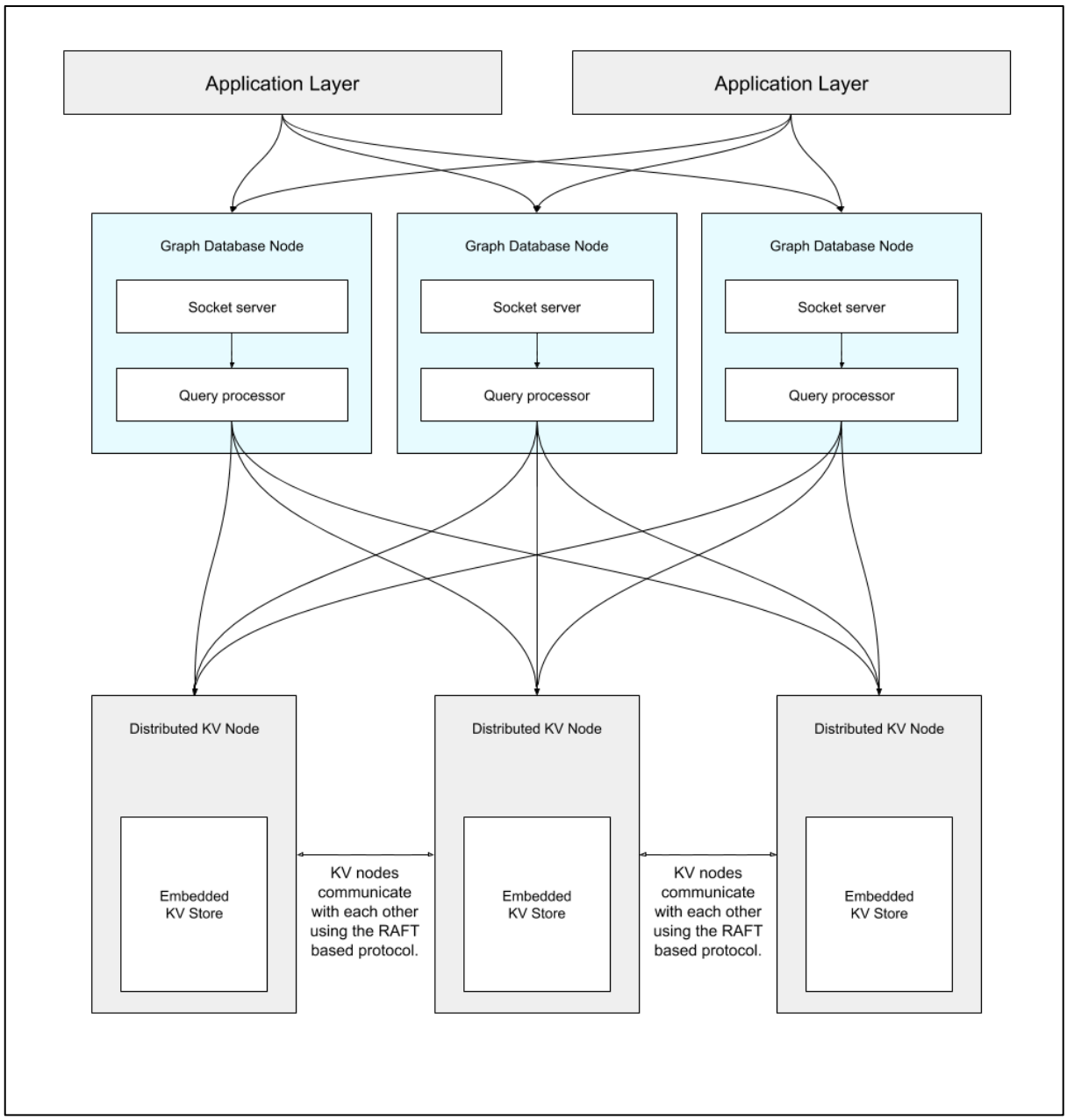

Figure 18 - Embedded key-value store within a multi-node distributed database

The distributed key-value store nodes will communicate with each other using a distributed consensus algorithm Raft [51], in order to ensure that all state between the nodes is kept consistent, and to ensure that transactions are committed to the store with Atomicity, Consistency, Isolation, and Durability properties [9]. This distributed architecture can be setup in a couple of different ways. The first method, visible in Figure 19, uses the Raft algorithm to ensure that the total data contents of all nodes are replicated and in-sync with one another, leading to a highly available architecture. In this case, when a node joins the distributed cluster, it will download a snapshot of the entire dataset from another node, before being able to accept read requests. If a node in the cluster were to fail, then by using the Raft algorithm, the cluster would select a new leader, and would continue to accept reads and writes to the dataset, syncing consistently between the nodes. Although this method leads to a higher availability architecture than a single node can offer, the issue still remains where the dataset cannot be larger than the storage and memory capabilities of the smallest member node in the cluster. I shall investigate this problem further in Chapter 6.

Although the implementation of a distributed key-value store is not the focus of this project, it is important to base design decisions around the necessities and requirements which would arise. In this scenario, a node which is joining the cluster for the first time, or which is reconnecting after a crash or network partition, would need to quickly compare its version of the dataset with the dataset on a current node in the cluster. As a result of this, three factors need to be possible: the first is that a dataset must be able to be compared quickly with another dataset; the second is that a snapshot of the entire dataset, or a subset of the dataset, needs to be able to be sent and received over the network by different nodes; and finally a dataset should be able to be loaded quickly so that a node may recover and join the consensus group in a timely fashion. In addition, each data store should be able to support multi-version concurrency control at a previous timestamp, so that a distributed transaction can fetch the dataset at a consistent time throughout the cluster. These implementation requirements lead to a number of design decisions which will be incorporated into the proposed data structure and data store design in Sections 4.2 and 4.3.

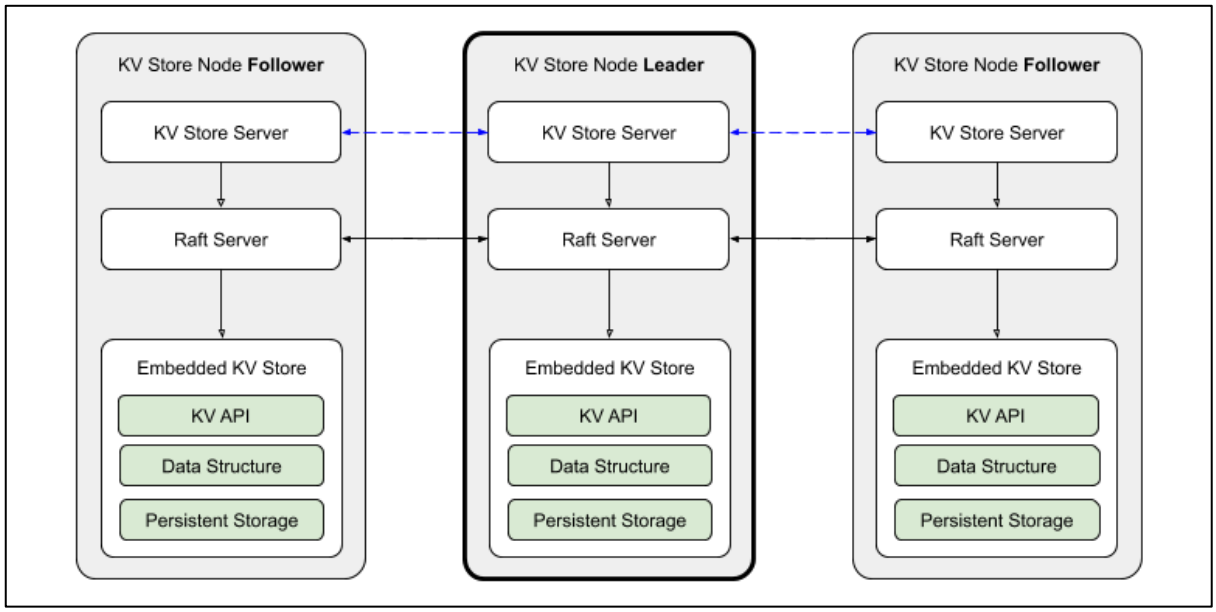

Figure 19 - A highly available distributed database architecture using Raft

4.1.2 Problems with a layered approach

Although using a layered approach has its advantages, it is important to take into account the disadvantages that this approach can bring [52]. When separating the querying layer from the underlying key-value storage, the functionality of each layer is defined and accessed through a specific API, dictating what functionality can and can't be called upon from an external layer. This can lead to implications when it comes to querying, compression, and storage, as each layer is prevented using intricate functionality implemented within another layer. With regards to data storage, for example, the embedded key-value store specifies that data must be encoded and stored as binary data. This works well when storing to disk, or when sending and storing over a network, but forces us to encode in-memory data. This in turn limits us from being able to make use of the data structure without first encoding any in-memory data to a binary format.

In addition, when it comes to querying the dataset, the layered approach introduces some negative effects. With this approach a node must fetch as much data as it thinks it needs in order to satisfy a particular query - fetching data ahead of time, even if it ends up not needing to process all of the data. This is not too much of a problem in a single-node setup, as the result sets are able to be returned in a matter of nanoseconds. However, when looking at a distributed store, requesting, receiving, and processing more data than is necessary immediately has an effect on both the graph database query layer, the distributed key-value storage layer, and on the network. Instead, a way of mitigating this effect would be to pass more information surrounding the query onto the storage layer, resulting in the storage nodes having a greater understanding of the query itself, leading to more efficient processing, and only needing to send back relevant data, minimising round-trip times, and reducing data transfer costs.

On reflection however, although these disadvantages introduce a number of technical difficulties, I settled upon following the layered approach to developing this database, leading to the ability to separate out the data store from the querying layer. As a result, I shall look more deeply into the impact of this design decision in Chapter 6, with a suggestion of how to overcome the disadvantages of layer separation.

4.2 Data structure design

The design of the data structure is arguably the most important part of implementing the database. When setting out to design a new database, there are a number of data structures to choose from, each with its own specific characteristics and intentional usage patterns. Popular data structures such as B-trees [53], or close variants thereof (such as B+-trees and RB-trees), are heavily used, for storage or indexing, in a number of databases including MySQL [54], PostgreSQL [55], BoltDB [47], and were designed with a focus on storage using spinning-disk hard drives . Other potential data structures include R-trees for geo-spatial indexing, LSM-trees for heavy-write scenarios on solid-state disk drives, hash tables for efficient single-item lookup, skip-lists for improved linked lists characteristics, as well as many other different structures suited for different use cases. As an integral part of the data store, the data structure defines the characteristics and performance traits for the overlying implementation. In searching for the right data structure for this project, it was important to satisfy the requirements set out in Chapter 3. The proposed implementation - a *Temporally Augmented Radix Tree* - shown in Figure 20 makes use of a combination of approaches in order to meet the performance, functionality, and storage requirements. This structure is composed of three layers, which all work together to form the full data structure, enabling concurrent access, with both *valid time* and *transaction time* capabilities.

During the early discovery phase of this project, a couple of different data structures were tested as a storage structure. One alternative which was investigated was the R-tree [56] as it had a number of benefits: first it could be stored on disk; secondly it could be setup to have multi-dimension indexing; and finally, its use as a bi-temporal indexing structure [57] had already been proposed. However, due to a number of factors, the search for an alternative structure was continued. First, as the index structure would be operating in main-memory, a structure with space efficiency was sought, whilst R-trees were designed for storage to disk. Secondly, due to the complexity of indexing multi-dimensional data, the R-tree has a performance cost when it comes to rebalancing the tree after a modification. Log Structured Merge trees [7] were also considered, given their suitability for high-write scenarios, as these structures typically store the top layer of the tree in memory, with lower levels being written to disk. While they are suitable for high-write scenarios and solid-state disk storage, I continued to search for a structure which could easily allow copy-on-write functionality, while being able to be stored fully in main-memory.

In addition, a number of different data structure types were tried for the node versioning. The first method made use of a skip-list [58] in order to store versions in decreasing order - similar to a linked list, but enabling a theoretical average complexity of `O(log n)` to be attained when searching for a specific version. However, three main factors led to the choice of a different structure: first, the probabilistic nature of a skip-list, along with the fact that in our use case, versions were predominantly inserted at the beginning of the list; secondly, the fact that copy-on-write persistence within the list results in copying of the entire data structure; and finally, because in the context of this domain, there is a need to store only unsigned 64-bit integers for each version.

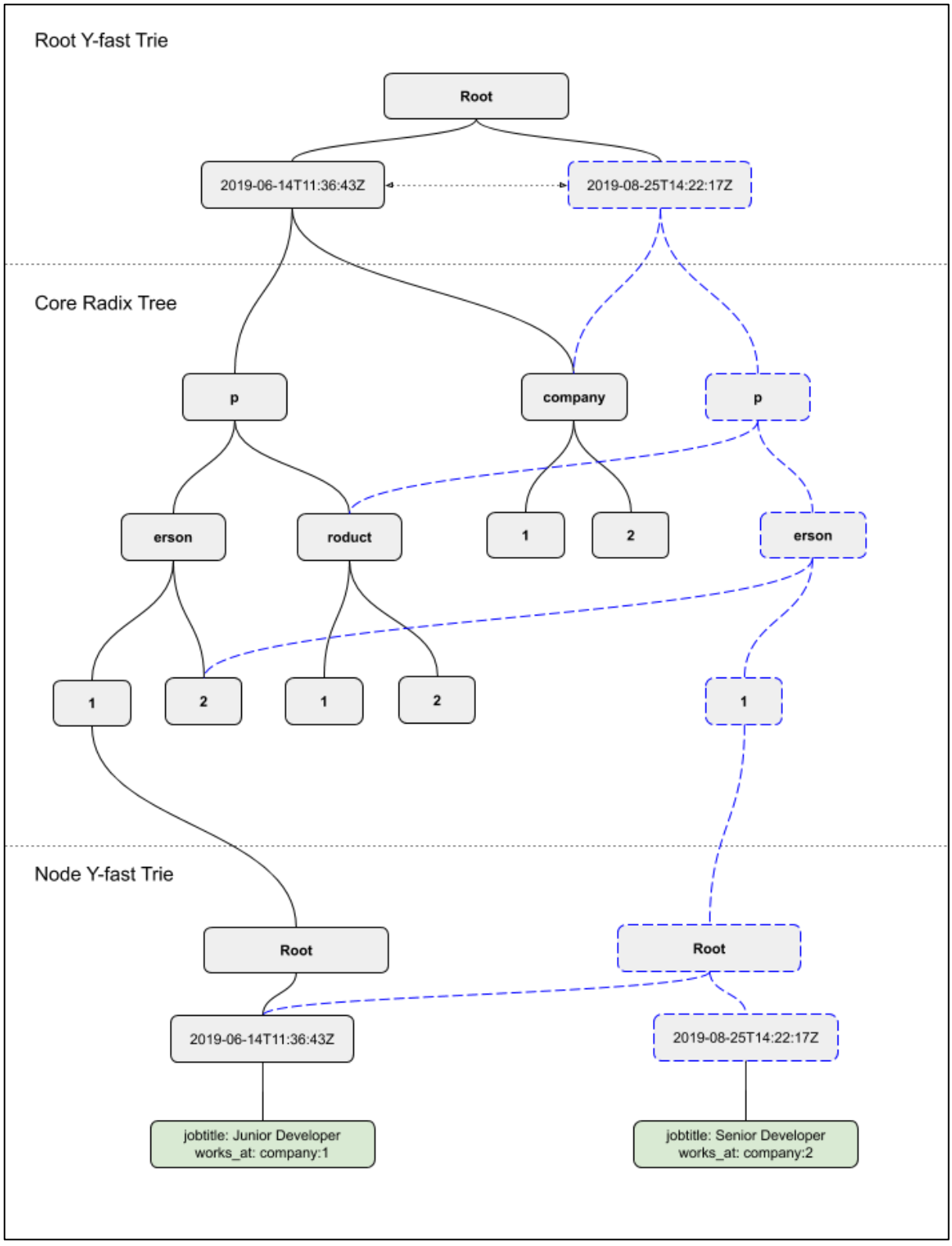

Figure 20 - Proposed implementation of a Temporally Augmented Radix Tree

4.2.1 Core data structure

At the core of the data structure is a copy-on-write, compressed prefix, persistent Radix tree [45]. The Radix tree stores the separated parts of the key-value keys within each node of the tree. Given that each key-value item stored in the database is similar, each sharing common path prefixes, we can easily benefit from the Radix tree prefix compression characteristic which results in only a few nodes being used to represent long data store keys. This prefix compression in turn enables us to store the data structure in main memory, allowing for performant traversal of the tree, and simplified copy-on-write semantics, with regards to the data structure paths, when making modifications to the tree. In addition, as the keys are stored within the tree itself, as opposed to being stored in the leaf nodes, the Radix tree offers the benefit of not having to rebalance the tree after every modification - a side-effect which should result in implementation simplifications and performance gains within the data store.

There are three main design decisions which are implemented within the edge nodes of the Radix tree. First, each node stores a pointer to the parent node, and an array of pointers to children nodes, as shown in Figure 21. This allows the structure to be iterated through, as each node can simply traverse up to the parent node, or down to the previous or next child nodes. In addition, the compressed prefix of a part of the key (which the node represents) is stored inside the node. This enables us to form the key from a combination of the hierarchical prefix values on each node, allowing us to retrieve the key as we iterate through the tree, without having to store the key multiple times at each leaf node.

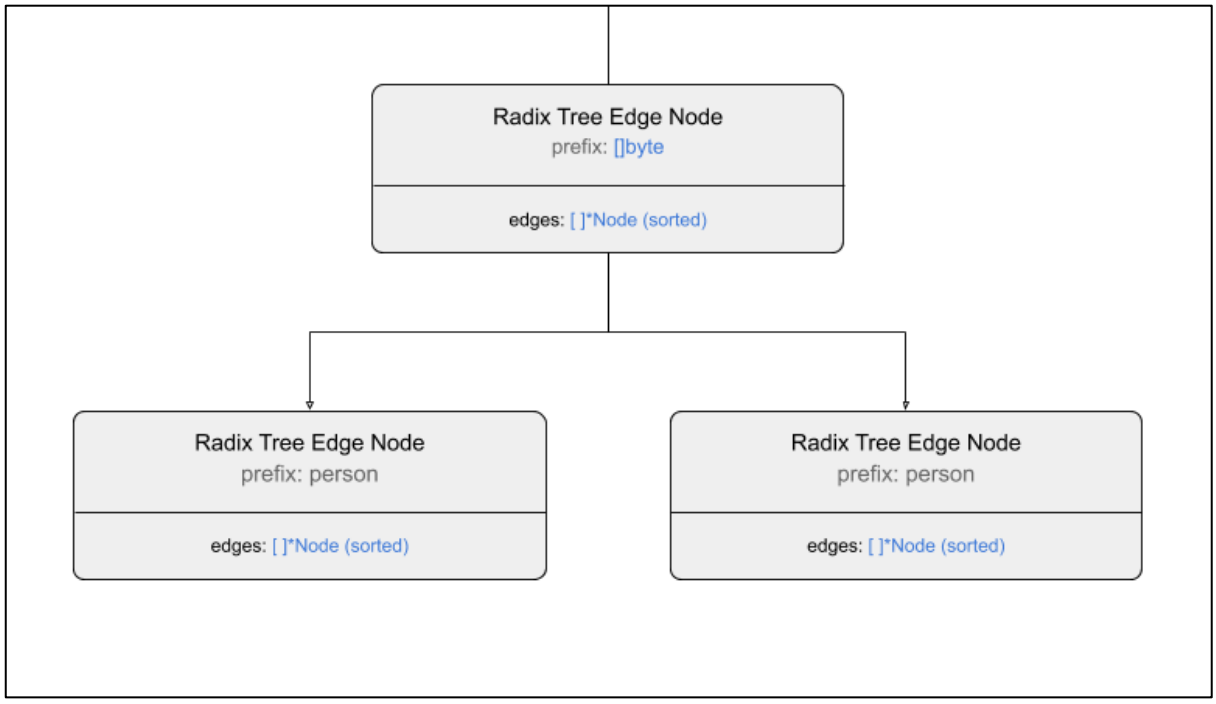

Figure 21 - A Temporally Augmented Radix Tree Edge Node

Like the edge nodes, the leaf nodes of the tree, visible in Figure 22, also store a pointer to their parent node in order to enable traversal to alternative paths in the tree. The key-value binary values themselves are not stored directly within the leaf node; instead a pointer to a Y-fast trie [59] dictates how the data is stored. The Y-fast trie stores the versions of each change to the data, by storing the data value along with the version timestamp as an unsigned 64-bit integer representation of a nanosecond-base time. The Radix tree has two other additional fields which are used to improve lookup times for the first and last versioned values. The `min` field stores a pointer directly to the earliest version, and the `max` field stores a pointer directly to the latest version, enabling us to do two things: first, if the database is being used without versioning, performing a lookup for a key-value item with `version(0)` should be able to retrieve the item with `O(1)` slowdown; and secondly, when using versioned values, we should be able to fetch the latest known version of a key-value item, without having to do a lookup in the Y-fast trie, by fetching the maximum value directly, again with `O(1)` slowdown.

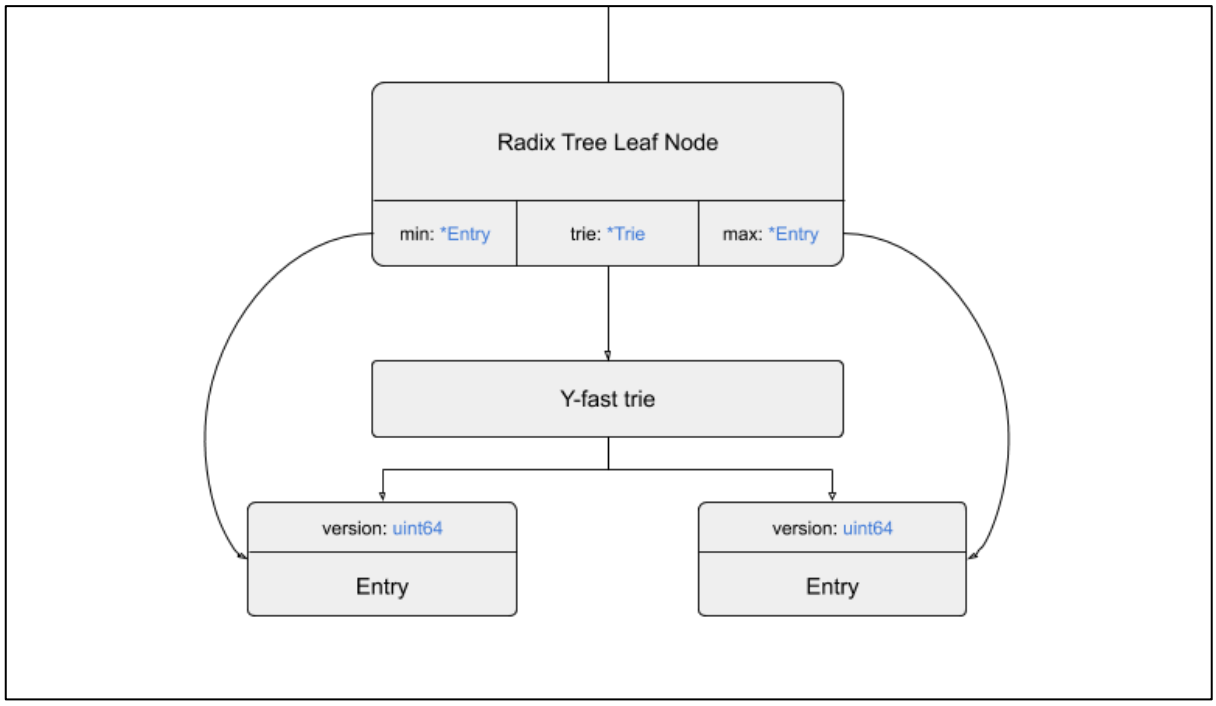

Figure 22 - A Temporally Augmented Radix Tree Leaf Node

As can be seen in Figure 20 and Figure 23 the data structure is a copy-on-write persistent tree, prompting a new set of nodes from the altered node up to the root node, whenever a modification is made to the tree. The negative side effect of this approach is that path copying can be costly if having to duplicate a large number of nodes to get to the root node - something which should be somewhat mitigated due to the Radix tree prefix compression, ensuring that the number of nodes needing to be copied is reduced. On the other hand, this approach leads to a number of benefits. First, the copy-on-write with atomic pointer switching allows a writer to atomically apply changes to the new tree, whilst multiple readers are still reading the previous versions of the tree, unaware that a new version is being created. Additionally, and most importantly, this approach leads to the ability to store every modification made to the data structure over time - an important factor when building a temporal data store.

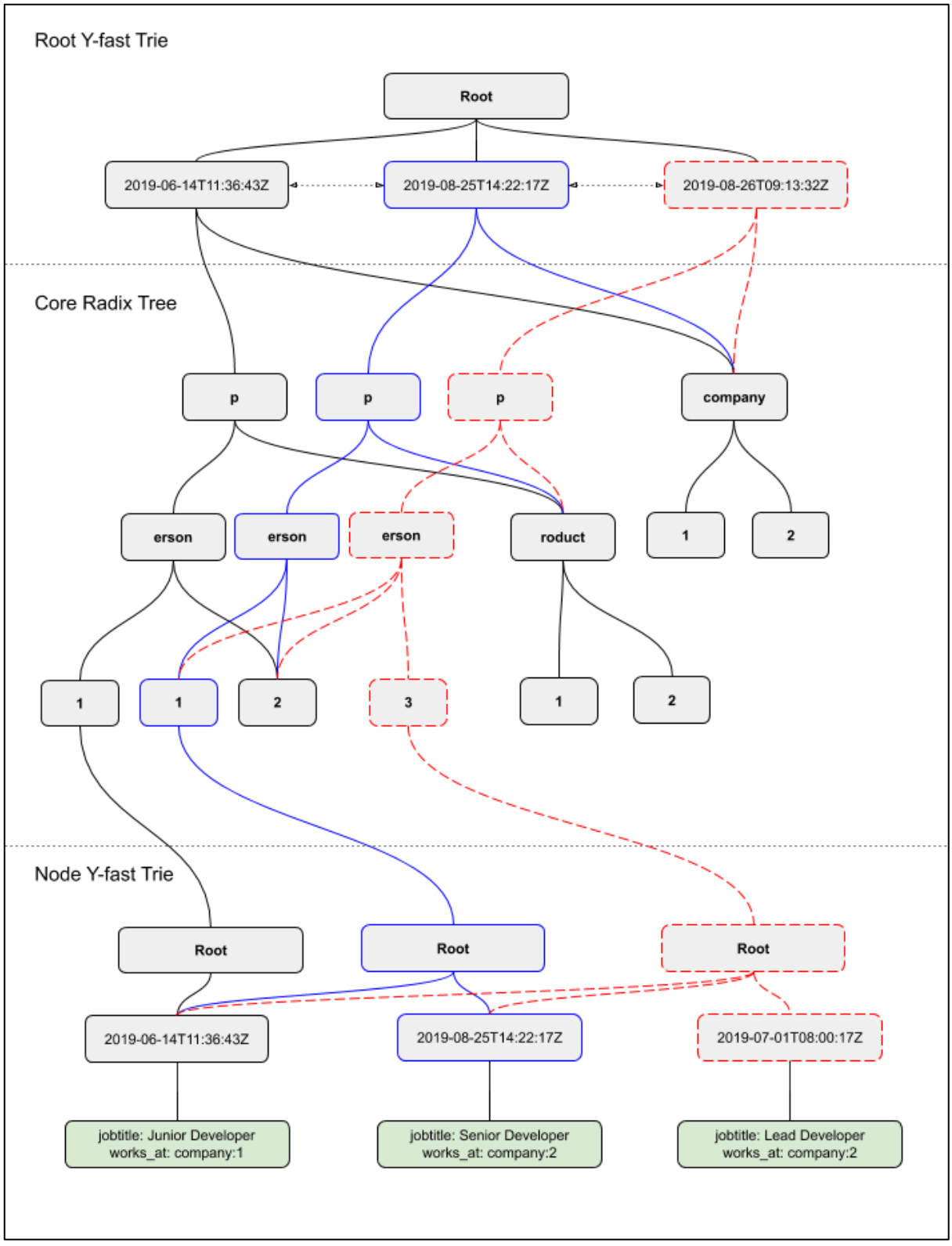

Figure 23 - A temporal modification within a Temporally Augmented Radix Tree

4.2.2 Data structure versioning

The data structure makes use of a Y-fast trie in two separate places in order to support the functionality of both *valid time* and *transaction time*. The first Y-fast trie exists within every leaf node of the tree and stores the different *valid* time versions using 64-bit integer indexing. Changes to a key-value item are inserted into the Y-fast trie using the version identifier specified by the client, or by using the system time (to denote the current time) if no version is specified. If the versioning identifier is less than the `min` value, or greater than the `max` value in the radix tree node, then the node's `min` and `max` pointers are updated to point to the minimum and maximum values in the Y-fast trie respectively. This change is then traversed up the tree, until it reaches the root node, returning a new tree, which encompasses the new changes within it.

When a new copy of the tree has been made, and a new root node has been generated, the in-memory pointer is inserted into the root Y-fast trie. In a similar manner to the embedded Y-fast tries, this root Y-fast trie stores different versions using a 64-bit integer. However, in this case the version timestamp is not specified by the client making the request, instead being generated using the system time. This difference in behaviour between the root and node Y-fast tries permits the distinction between *valid time* (specified on a node update), and *transaction time* (specified on a transaction update).

Importantly, due to the immutability of the persistent data structure, as each transaction is committed and each set of mutations is stored within the root Y-fast trie, the data structure storage grows ever larger over time. The decision to use a Radix tree as the core data structure, enables, through the use of compressed trie prefixes, a smaller number of copies for each change from the leaf to the root of the tree. In addition, with the same input data, the Radix tree should be structured in the same manner irrespective of the insert order and would not need to be re-balanced on every change, resulting in fewer path differences between each new copy of the tree. As a result, the reasonably low memory usage with each new change to the data structure should allow for a large number of keys to remain in RAM. This design decision will be tested further in Chapter 5.

4.2.3 Data structure discussion

Although not explicitly listed as a set of requirements in Chapter 3, an important modification to the data structure could be made in order to support a multi-node distributed data store setup, detailed in Section 4.1.1. By utilising methods found in Merkle Trees [60], each node would store the hash of the sum of its edges, and would update this hash value on every modification (due to copy-on-write, the hash of only a small number of nodes would need to be recomputed). This would enable us to compute whether two separate key-value stores, each on a different node in a distributed cluster, are exact copies of each other - forcing a data sync if the data structure hashes do not match. An example of this can be seen in Figure 24.

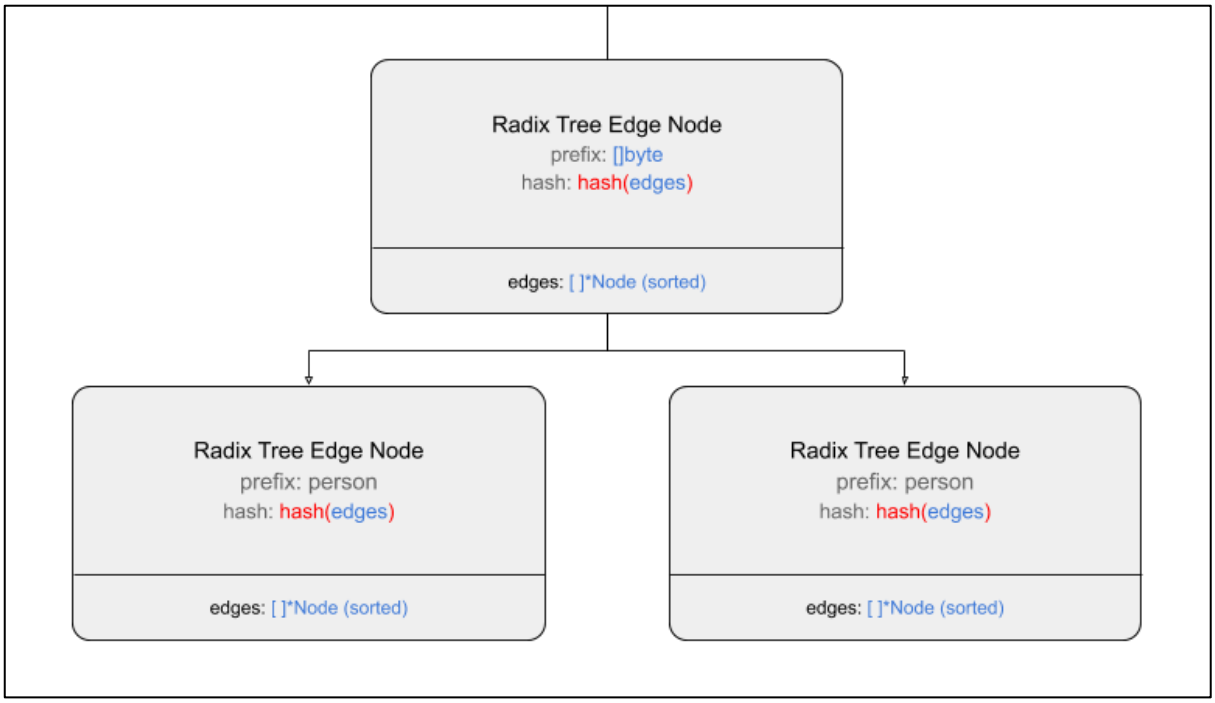

Figure 24 - Merkle Tree support in a Temporally Augmented Radix Tree

The insertion of a `min` and `max` field on a leaf node, ensuring that a query is able to retrieve the oldest and newest values directly, without querying the Y-fast trie, seems like a reasonable method for ensuring the 'current' and 'initial' versions are always accessible with an `O(1)` slowdown. However, on reflection, as the data store enables the insertion of future facts and events (by specifying a unsigned 64-bit integer versioning timestamp for a future time when inserting a value), the pointer stored in the `max` field may not necessarily be the correct version according to the current time. Although the Y-fast trie will still enable retrieval with a `O(log log U)` where U is the maximum value in the domain or `O(log 64)` cost, it is an issue which one needs to be aware of when employing future versions within the data store.

The Temporally Augmented Radix Tree, proposed in this chapter, forms the basis of the data store which will be discussed in the next section. With the copy-on-write functionality allowing for multiple concurrent readers and writer, versioning support for both valid time and transaction time, prefix compression for improving data structure memory usage, and mechanisms to allow for finding the minimum and maximum value version, the data structure is a suitable candidate for meeting the requirements in Chapter 3 - which I shall analyse later in this dissertation. In the following section we will look at how the data structure fits within the greater context of a key-value data store, in order to satisfy the requirements, and allowing it to be compared and benchmarked with other systems.

4.3 Data store design

In this implementation, the Temporally Augmented Radix Tree would be embedded within the data store, visible in Figure 25, providing versioned, in-memory organisation and management of the data, enabling sorted data storage for both reading and writing. The data store itself manages access to and modification of the data structure, using a number of techniques, in addition to controlling the persistence and durability of the dataset - effectively being responsible for the ACID guarantees which are a key requirement of the key-value data store. Finally, the data store is responsible for the encryption, decryption, compression, and decompression of all data that is to reside within the data structure. In this section we will look at the design of the data store and will analyse the reasoning behind some of the design decisions made.

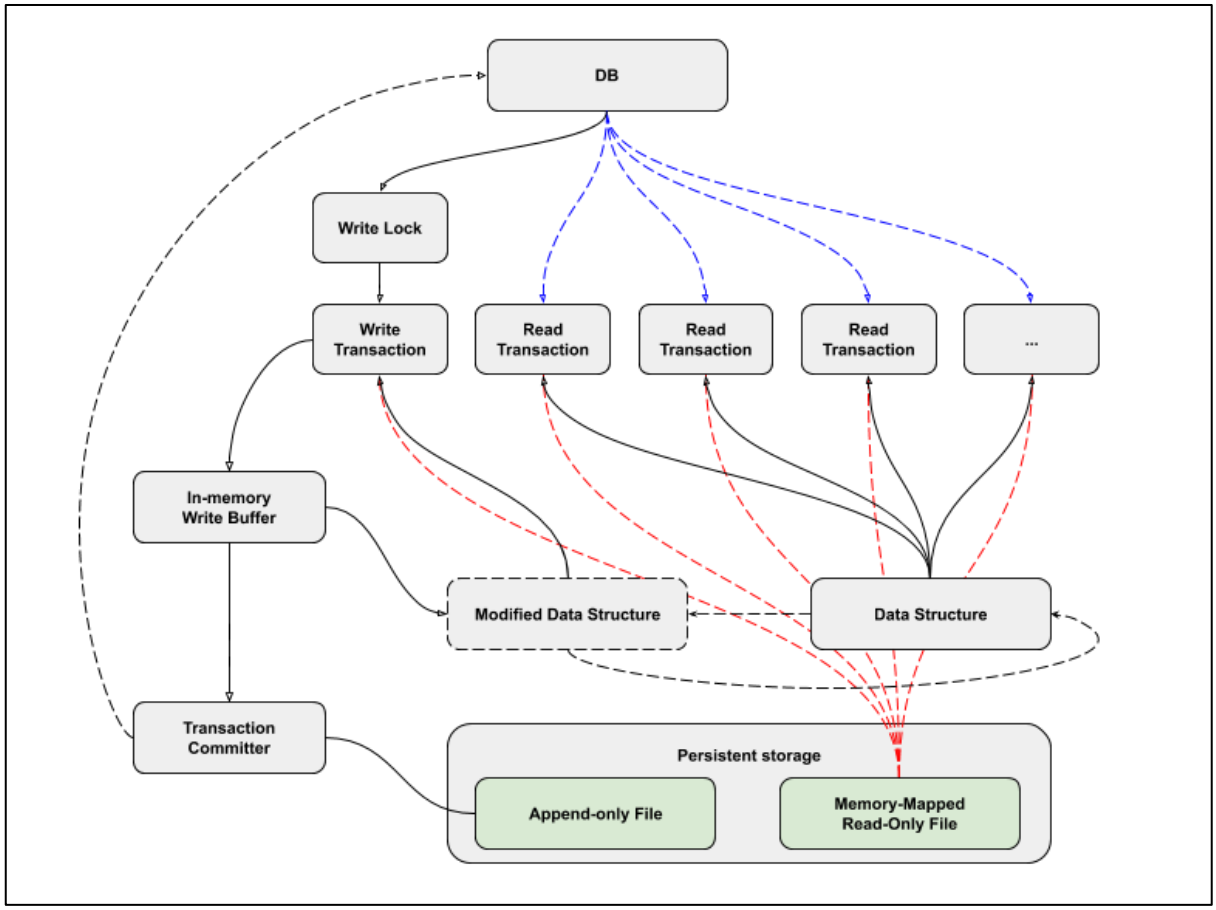

Figure 25 - Data store implementation, showing concurrent read and write transactions

In the design of the key-value store, a couple of assumptions were made: first, due to the tremendous performance improvements seen with solid-state-disks (SSDs) [61], and with the declining costs of SSD storage [62], that the data store would be persisting to SSD storage devices; and secondly, that the store would be used in a write-heavy, read-heavy scenario. As a result, the design decisions have been based around the performance characteristics of SSDs, in order to achieve a high throughput for both retrieving and modifying data. To achieve this, as we can see in Figure 26, the data store separates out the storage of keys from the storage of values, using an approach detailed by Lu et al., in WiscKey [63] - reducing IO amplification when writing to the storage device.

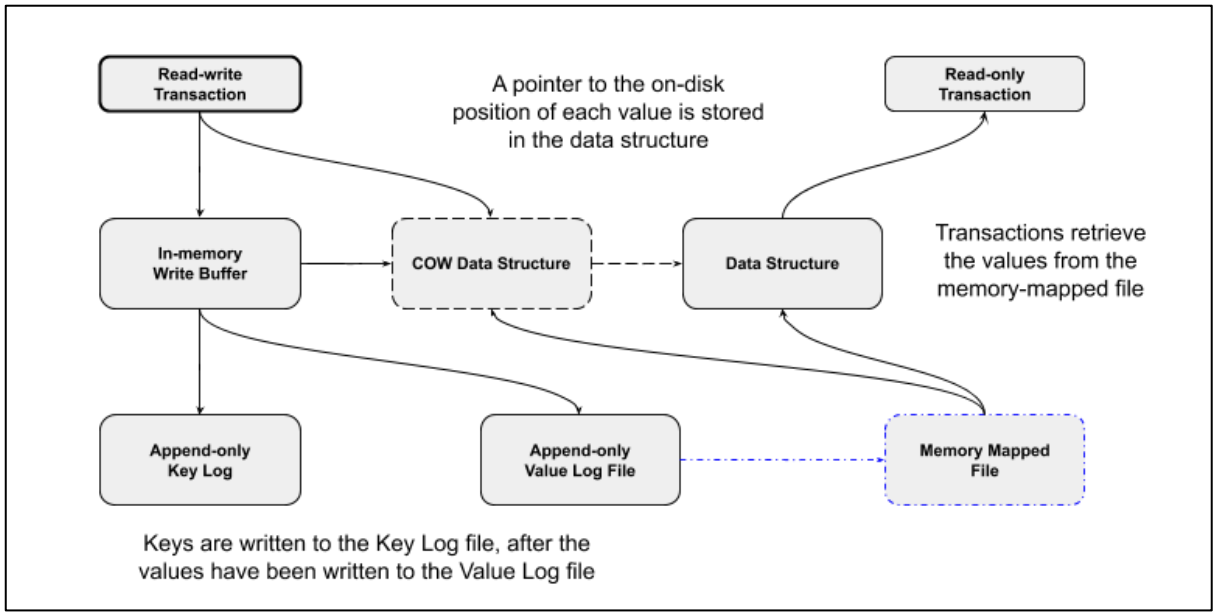

Figure 26 - The processes for reading and writing data within the key-value store

The transaction commit process, visible in Figure 26, is an integral part of the key-value store, without which the read and write storage performance would not be attainable. In this process, all changes made to the copy-on-write data structure are written to an in-memory data buffer, which stores a representation of the key-value changes, in binary form. Once the transaction is ready to commit, the write buffer separates out the keys and values, writing each to a different append-only log file (values will be written first, so as not to prevent atomicity or consistency) in two operations. The values are written to the value log, in binary form, as one large entry, with the values stored immediately after the preceding value.

The key-log file will store the operations made within the transaction, starting with the system version time of the transaction itself. Next, the modification operations are inserted, using **`PUT`** (for inserts), **`DEL`** (for deletes), or **`CLR`** (for clearing of all values) to signify what operation is to be made, and followed by the key that was modified. For items which were updated (**`PUT`**), the key is followed by two values: the first is a big-endian encoded unsigned 64-bit integer of the size of the value; and the second is another big-endian encoded integer of the position of the value within the value-log file. An example of these operations (in non-binary form) is visible in Figure 27. Finally, instead of storing the actual values within the data structure, the leaf nodes now store the two unsigned 64-bit integers detailing the size and position of the value within the value-log file. By using the technique of separating out keys and values, and by using append-only storage for persistence, the data store is designed for sequential writes, and random-access workloads, which should satisfy the high-write performance needs specified in **Requirement 4**.

```
TXN 2019-07-27T13:00:00Z
PUT /db/table/1 468 1937481
PUT /db/table/2 468 1937949
DEL /db/table/3
DEL /db/table/4
PUT /db/table/5 468 1938417
```

Figure 27 - Example contents of a key-log file in human-readable format

When a transaction retrieves an entry from the store it makes use of the memory mapped value-log, in-turn, retrieving the values from the file - by loading the specific size of data, at the specified position within the file. In this way, a key is stored completely in memory within the trie, using trie-based prefix compression, resulting in low memory usage, and enabling performant iteration, whilst each value is loaded from the file on demand when a request for a key-value item is made.

Exporting and importing into the key-value store can be enabled by extending the functionality already offered by the transaction and storage process. For exporting, the data store iterates through the key-log file, streaming the contents to an IO-writer, followed by the contents of the value-log file. This technique enables the ability to sync the database over a network, without affecting any current read or write transactions, satisfying the needs of **Requirement 2**. For importing, the data store offers two options: first, to start-up and load the data from a file on disk; and secondly to load the data from an IO-reader (allowing over the network importing). For both of these options, the data load will reset the data store contents, resulting in a clean memory state. This would enable a database to import the contents of dataset across a network in order to remain in-sync with other nodes, satisfying **Requirement 3**.

On reflection, the downside to this approach (instead of storing the whole data structure on disk and accessing it using memory mapping) is that we have to ingest all of the keys into the in-memory data structure on start-up, before being able to perform any reads or writes against the data store, resulting in a slower loading time. I shall investigate this issue, benchmarking it against alternative implementations in Chapter 5.

4.3.1 Atomicity, Consistency, Isolation, Durability

In order to meet the requirements, set out in Chapter 3, the data store must guarantee all aspects of the ACID set of properties for database transaction management. By designing for these four database transaction properties, we satisfy the stipulations specified under **Requirement 1**. The first of these, atomicity, must be ensured so that multiple changes can be made to the dataset, which either succeed as a whole, or don't succeed at all. By using a copy-on-write data structure we can ensure that any mutations that are made to the dataset create a copy of the altered nodes, instead of altering the tree in-place. Within the data-store we can use this to our advantage, making use of an in-memory write buffer to collect all of the changes made to the tree from within the transaction, before flushing these changes to persistent storage in a single operation.

The aspect of consistency, where the database is not left in an invalid state, either due to a failing transaction or because of a system failure, is ensured with the use, once again, of both the copy-on-write characteristic of the data structure, and with specific methods for persistent storage. When any modifications are committed, and once the total contents of the in-memory buffer are written to persistent storage, the data store will atomically insert the new root node into the root Y-fast trie. By committing the write buffer to storage in one operation, before the new transaction root is stored in memory, one can guarantee the consistency of the data.

Isolation - where one transaction is unable to see the partial modifications of another transaction - is once again guaranteed with the design of the data structure. The data store will prevent multiple read-write transactions, by employing the use of a read-write locking mechanism to ensure that only a single writer can create a new copy of the dataset at one time. Meanwhile, the lock does not prevent other concurrent read-only transactions from accessing the latest version. At the point that the transaction is committed, the pointer to the data structure root node will be inserted atomically within the Y-fast trie which stores the *transaction time* versions of the dataset - and on success, future transactions will be able to see the modified data.

4.3.2 Data-store discussion

An initial prototype implementation of the key-value store did not make use of memory mapping files in order to read data from persistent storage. Instead, as is shown in Figure 28, the data values were stored in memory within the data structure, with modifications written to disk in an append-only format. Although the same functionality and ACID properties were guaranteed, this approach was sufficient for datasets where the total size of all keys, values, and programming language pointers was sufficiently small enough to fit into RAM. In an environment where the total dataset size exceeded the available RAM, the operating system would need to use virtual swap memory, causing a considerable data access slowdown. On reflection, in order that the key-value store could be used in scenarios where memory constraints were less than the size of the dataset, and so that the comparison with other competing key-value stores (presented in Chapter 5) was fair, **Requirement 5** was introduced. As a result, the design was changed to use a separate key-log and value-log, as detailed earlier in this Section, satisfying **Requirement 5**.

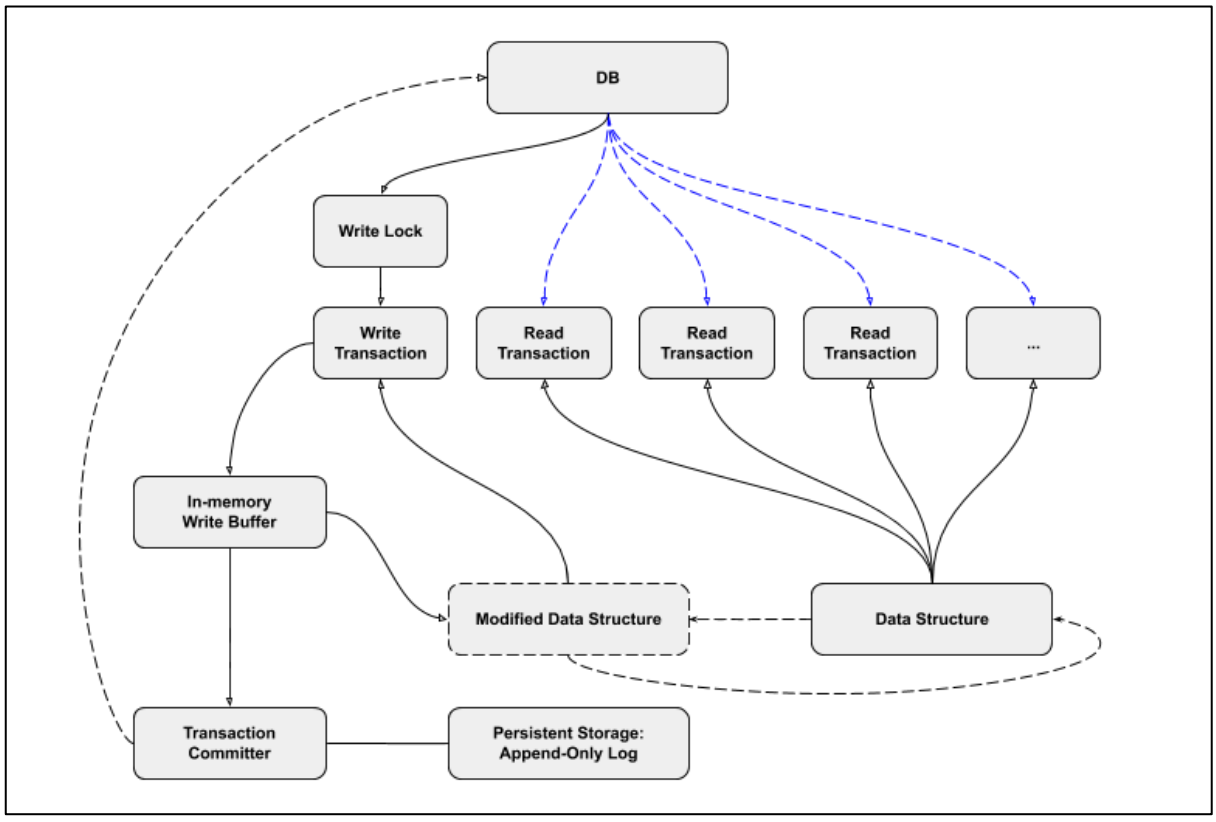

Figure 28 - Initial in-memory implementation, with persistent storage log

4.4 Key-value store API

In order to satisfy the requirements, set out in Chapter 3, the key-value store must support inserting, selecting, and deleting single keys or key ranges. The methods are grouped into 4 categories: **Put** methods which enable inserting a single key-value item, or a range of items into the store; **All** methods which enable selecting a single key-value item, or range of items, retrieving all of the versions stored for each item; **Get** methods which enable selecting a single key-value item, or range of items from the store; **Del** methods which remove a single key-value item, or range of items, by specifying a `NULL` value at the specified timestamp; and **Clr** methods which removes a single key-value item, or range of items, by clearing all current and historical values for the matching keys from the store. A description of the different methods is visible below. For each of the **Put**, **Get**, and **Del** methods, the first argument is an unsigned 64-bit version identifier, which signifies at which timestamp the version should be inserted, selected, or deleted. With the **All** methods, no version identifier is necessary, as these sets of methods will retrieve all versions of a key. Similarly, with the **Clr** methods which remove all items for a key, or a set of keys, no version identifier is needed. Using these 4 sets of methods should be enough to support the query requirements set out in Chapter 3.

4.4.1 Key-value transactions

The following methods give a client of the key-value store API full control over the transaction management process of the data store. These four methods allow for the creation of read-write transactions, concurrent read-only transactions, and transactions for iterating over the dataset as it appeared according to a historical *transaction time* in the past. With the **Begin** method, a transaction is created, preventing other concurrent write transactions, through the use of a lock, and along with the **Cancel** and **Commit** methods helps to satisfy **Requirement 1**. The **BeginAt** method, on the other hand, supports read-only transactions exclusively, fetching the correct tree root from the root Y-fast trie, allowing us to satisfy the needs in **Requirement 14**.

Begin creates a read or read-write transaction at the current system time.

```
func (db *DB) Begin(writeable bool) (*TX, error)
```

BeginAt creates a new read-only transaction at the specified *transaction time* version.

```
func (db *DB) BeginAt(version uint64) (*TX, error)
```

Cancel discards the changes made within a transaction.

```
func (tx *TX) Cancel() error
```

Commit commits the changes made within a transaction to the data store and to disk.

```
func (tx *TX) Commit() error
```

4.4.2 Key-value operations

The **Put**, and **PutC** methods enable inserting data into the data within a transaction. With these methods, data can be inserted at the current *valid* time version (specified by the system time), or by modifying the value at a particular version in the past or the future, fully satisfying the needs of **Requirement 10**.

Put inserts a single key-value item at the specified version.

```
func (tx *TX) Put(ver uint64, key, val []byte) (*KV, error)
```

PutC marks a key-value item as deleted, if the value is equal to the specified value.

```
func (tx *TX) PutC(ver uint64, key, val, exp []byte) (*KV, error)
```

The **Get**, **GetP**, and **GetR** methods allow retrieval of data from the data structure and from persistent storage, either as a single entry, a group of entries which match the specified prefix, or a range of entries. The **Get** methods will automatically traverse the data structure to retrieve the items and will fetch the corresponding values from the value-log file, before returning the matching entries to the client. These methods satisfy **Requirement 6**, and **Requirement 9**, with both current access and historical versioned access abilities.

Get retrieves a single key-value item.

```
func (tx *TX) Get(ver uint64, key []byte) (*KV, error)
```

GetP retrieves the range of rows which are prefixed with `key`.

```
func (tx *TX) GetP(ver uint64, key []byte, max uint64) ([]*KV, error)
```

GetR retrieves the range of `max` rows between `beg` and `end`.

```
func (tx *TX) GetR(ver uint64, beg, end []byte, max uint64) ([]*KV, error)
```

The **Del**, **DelC**, **DelP**, and **DelR** methods all enable the marking of data as deleted within the store. As the data store is immutable, instead of actually removing the data from the tree, a new version is created at the current or specified version, with a **`NULL`** value, signifying that the value is deleted, or does not exist, and satisfying **Requirement 11**.

Del marks a single key-value item as deleted.

```
func (tx *TX) Del(ver uint64, key []byte) (*KV, error)
```

DelC marks a key-value item as deleted, if the value is equal to the specified value.

```
func (tx *TX) DelC(ver uint64, key, exp []byte) (*KV, error)
```

DelP marks the range of rows which are prefixed with `key` as deleted.

```
func (tx *TX) DelP(ver uint64, key []byte, max uint64) ([]*KV, error)
```

DelR marks the range of `max` rows between `beg` and `end` as deleted.

```
func (tx *TX) DelR(ver uint64, beg, end []byte, max uint64) ([]*KV, error)
```

The **Clr**, **ClrP**, and **ClrR** methods enable clearing a single key-value entry, clearing all entries whose key matches the prefix, and clearing a range of keys respectively. The **Clr** methods completely remove a key-value entry, including all of its versions, from the data store, allowing for versioned data to be removed for all future transactions. As a result, no version identifier needs to be specified when using these methods. It is important to note that the historical, immutable data will still be available by accessing the previous transactions, using a specific *transaction time* version identifier.

Clr completely removes a single key-value item.

```
func (tx *TX) Clr(key []byte) (*KV, error)
```

ClrP completely removes the range of rows which are prefixed with `key`.

```
func (tx *TX) ClrP(key []byte, max uint64) ([]*KV, error)
```

ClrR completely removes the range of `max` rows between `beg` and `end`.

```
func (tx *TX) ClrR(beg, end []byte, max uint64) ([]*KV, error)
```

The **All**, **AllP**, and **AllR** methods enable us to retrieve all of the versions for a single key-value entry, multiple entries whose key matches the prefix, or for a range of keys respectively. These methods return all of the versions for all of the matching key-value entries, partially satisfying **Requirement 7** and **Requirement 9**.

All retrieves a single KV item, fetching the current version, and all historical versions.

```
func (tx *TX) All(key []byte) ([]*KV, error)
```

AllP retrieves the range of rows which are prefixed with `key`, fetching all versions.

```
func (tx *TX) AllP(key []byte, max uint64) (kvs []*KV, error)
```

AllR retrieves the range of `max` rows between `beg` and `end`, fetching all versions.

```
func (tx *TX) AllR(beg, end []byte, max uint64) ([]*KV, error)
```

4.4.3 Key-value store iteration

On reflection, the **All**, **AllP**, and **AllR** methods prevent us from having a more fine-grained control over the data retrieval process, allowing one to fetch all versions of the matching key-value entries, but preventing one from retrieving a certain specific range of versions for each entry. While working on the benchmarking and implementation comparison work in Chapter 5, it became apparent that traversing the entire structure, and storing all keys and versioned values in memory, while it iterated through the structure, caused a large number of memory allocations. As a result, through the use of a stack-based cursor, I decided to later add in functionality to enable manual iteration through the values (functionality which was already available on the data structure, but had not been made available in the key-value store API), ensuring that only a single key-value item was allocated in memory at a time. With this addition we can now retrieve the desired versioned data values from within a key-value entry, by iterating through the Y-fast trie, satisfying **Requirement 7** and **Requirement 9** fully.

Cursor creates a stack-based cursor for iterating over the tree.

```
func (tx *TX) Cursor() (*Cursor)
```

First moves the cursor to the first leaf node in the tree, and returns its value.

```
func (cu *Cursor) First() ([]byte, *Node)
```

Last moves the cursor to the last leaf node in the tree, and returns its value.

```
func (cu *Cursor) Last() ([]byte, *Node)
```

Prev reverses the cursor to the previous leaf node in the tree, and returns its value.

```
func (cu *Cursor) Prev() ([]byte, *Node)
```

Next advances the cursor to the next leaf node in the tree, and returns its value.

```
func (cu *Cursor) Next() ([]byte, *Node)
```

Seek jumps to the nearest item in the tree whose key is greater than the specified key.

```
func (cu *Cursor) Seek(key []byte) ([]byte, *Node)
```

4.4.4 Key-value store API discussion

On reflection, although the API defined in this section satisfies all of the related requirements set out in Chapter 3, there is a remaining issue with the Put, Del, and Clr operations, which may be confusing to end-users. Currently, the **Put** methods can be used to insert a value into the store, allowing the insertion of a `NULL` value. This is similar to the **Del** methods which insert a `NULL` value into the store to signify the deletion of an entry. As a result, if one were to require the insertion of a `NULL` value - for instance to specify that something exists, but does not have a value - then the data store as it stands will perceive that item as not existing, and will skip it when iterating through the tree using the **Get** methods. Similar issues have also manifested themselves within other data stores [64], and this is an area of the key-value store that needs to be improved in future work.

In this chapter I presented a Temporally Augmented Radix Tree data structure, along with an embeddable, immutable, versioned key-value data store, which, when implemented together, satisfy the majority of requirements set out in Chapter 3. The remaining requirements - **Requirement 12** and **Requirement 13** - shall be investigated in the following chapter, where we shall compare and benchmark the proposed implementation with other key-value data stores.

5. Implementation, benchmarking and evaluation

5.1 Implementation overview

For implementing the proposed key-value data store, I chose a language which enabled me to rapidly develop a prototype using agile methodologies, whilst still ensuring a reasonable level of performance - so that the proposed data structure could be tested as early on in the process as possible. As a result, I made use of Golang, a compiled, garbage-collected language, which makes heavy use of Communicating Sequential Processes (CSP) to enable concurrency within the language, in order to enable concurrent tasks and threads. The choice to use Golang was driven by the fact that the graph database which was to use the embedded key-value store was already written using Golang, leading to a simpler development integration. In addition, the Golang code is able to be cross compiled, quickly and effectively, for use on many different architectures, enabling the data store (and encompassing database) to run in a range of environments, including container based operating systems.

5.2 Implementation comparisons

When choosing alternative key-value stores with which to compare this prototype implementation, it was important to find products which had a reasonable level of similarity when it came to functionality within the store. As a result, I have chosen data-stores which support transaction-based workloads, making use of Serializable Isolation, or Serializable Snapshot Isolation to guarantee the ACID properties within the database. In addition, it was important to compare implementations which were developed using the same language, reducing the chance of differing results due to language differences and alternative memory implementations. As a result, for the most part, the tests make use of other Golang data stores, except for LMDB which uses a Golang-to-C bridge to connect to the LMDB library.

The databases which I have chosen to perform the benchmarks with are BoltDB, LMDB, LevelDB, and BadgerDB. The first data-store, LMDB, is a tried-and-tested embedded transactional database library written in C. Its implementation uses a B-tree to store the data in a sorted order, allowing for retrieval, range scans, and iterators. The LMDB design allows for a single-writer, and multiple concurrent readers. All operations made to the key-value data space must be made from within a read or write transaction, using memory-mapped files to sync the data to pages on disk - leaving the responsibility of the page management up to the operating system. It is important to note that the LMDB library has a slight advantage as it does not make use of the garbage collection within the Golang runtime, resulting in a more efficient implementation.

The other three stores are written in the Golang language, with no external dependencies in other languages. BoltDB is a memory-mapped implementation of a B-tree, based on the design of LMDB, allowing for nested hierarchical buckets of data. Its implementation also allows for a single-writer, and multiple concurrent readers. LevelDB is a Golang implementation of the C-based database library introduced and developed by Google, allowing for an ordered mapping of string keys to string values using a Log Structured Merge Tree. Although data does not necessarily need to be altered within a transaction, its design does allow for the use of transactions and batch writing, enabling us to remain fair with regards to benchmarking. Besides the basic insertion, deletion, and selection queries for single keys, the data store also supports forward and reverse iteration through the store, and ensures that data is automatically compressed on disk using Snappy Compression [65]. Finally, BadgerDB is another implementation of a Log Structured Merge Tree, which makes use of the WiscKey approach [63] to separate keys from values on disk - allowing for in-memory storage of the key, and fetching the values from disk when requested. The implementation makes use of Serializable Snapshot Isolation, using versioned timestamps to detect data conflicts, and allowing for multiple concurrent readers and writers.

5.3 Test results and benchmarks

In order to compare the different data-stores, it was important to keep the benchmarks as fair as possible, attempting to make use of the correct techniques within each store in order to gain the expected levels of performance. In the tests, the keys are generated using a uniquely generated ID, based on the MongoDB Object ID Algorithm [66], resulting in an incrementing random 12-byte identifier. This is then appended to the end of a database path to form a unique key: `/key/namespace/database/table/9m4e2mr0ui3e8a215n4g`. In addition, to ensure the fairness of tests, I establish that all memory is garbage collected, and cleared between each individual benchmark process.

The data used for the values for each key and version was a unique randomly generated 128-byte string, representing encoded data for a database record. To perform the differing benchmarks, we used three methods for inserting and accessing data within each store: first, 500,000 key-value entries, each with 100 historical versions (coloured blue in the charts); secondly, 100,000 items each with 500 versions (coloured red on the charts); and finally 10,000 items each with 5000 versions (coloured yellow in the charts). The total number of key-value entries for each test was 50,000,000. The benchmark results for the implementation in this project is visible using the name *RixxDB* in the ensuing charts.

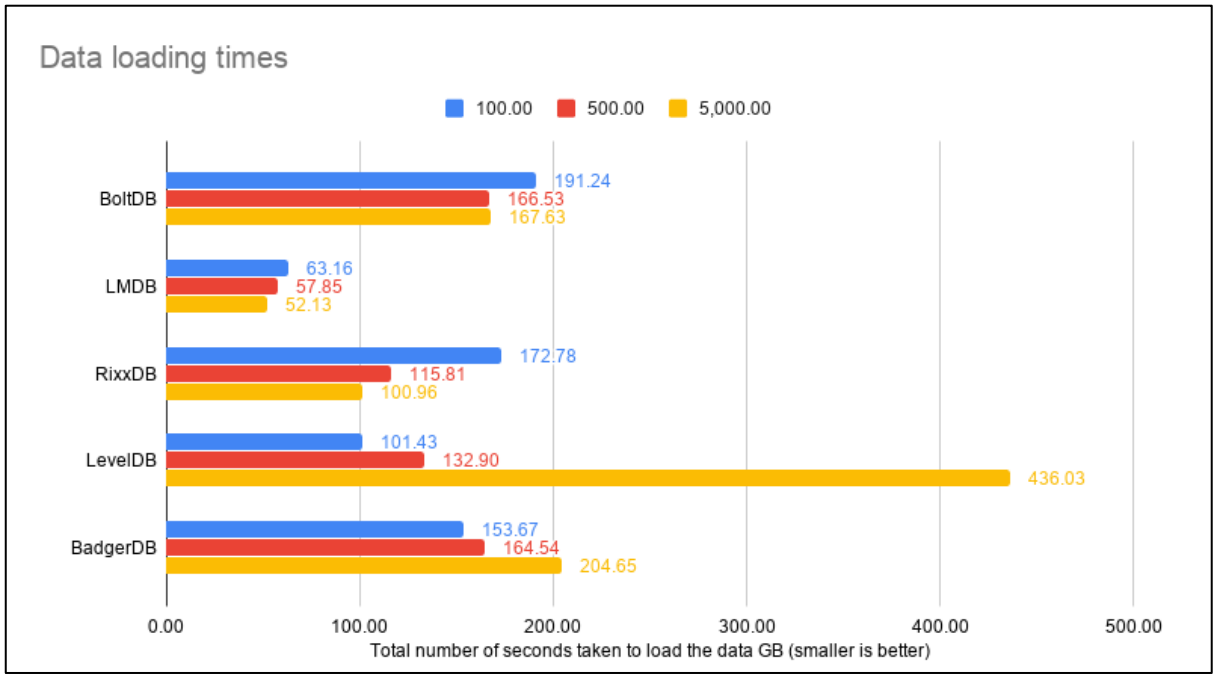

Figure 29 - Initial data insertion times for each key-value store

In this chart one can see the loading times of the different data stores when loading the three different datasets. The performance of this project's implementation (named RixxDB in the chart), deteriorates as more key-value entries, and fewer versions are used. We analyse the reasoning behind this later in this chapter.

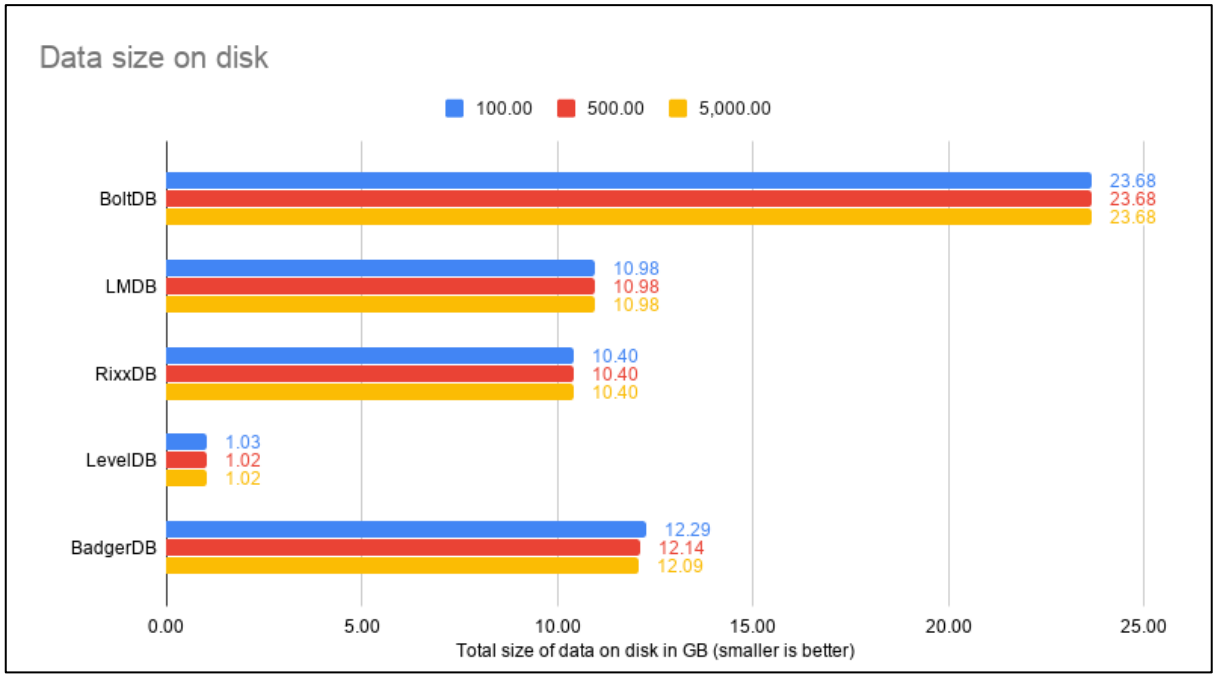

Figure 30 - Final size of persisted on-disk storage for each key-value store

The final size of the data on disk is visible in this chart - with LevelDB's Snappy Compression causing a great improvement over the other key-value stores.

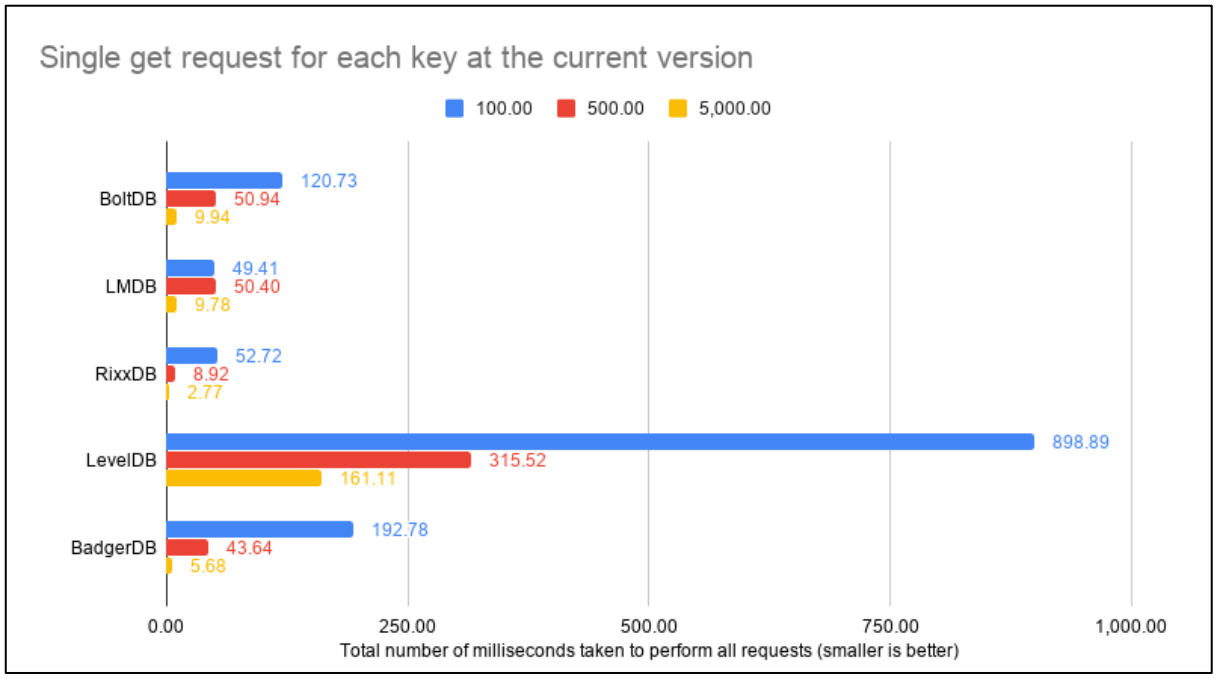

Figure 31 - Single-item retrieval for each key, fetching the latest version

In this benchmark, each generated key was fetched asynchronously, retrieving the latest stored version. The majority of the data stores perform similarly with this approach, as the `O(log n)` performance complexity of the B-trees and LSM-trees does not cause any noticeable performance issues with this amount of data.

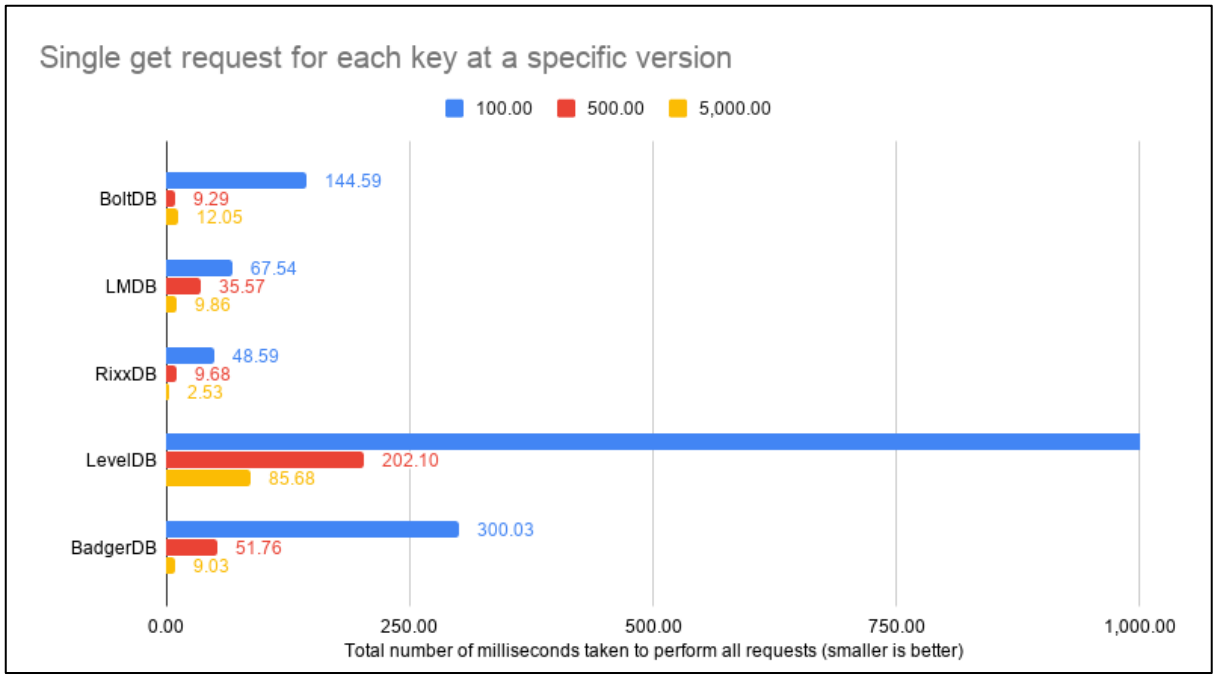

Figure 32 - Single-item retrieval for each key, fetching a specific historical version

In a similar manner to the previous chart, this benchmarked fetched each key individually, whilst using a specific version identifier. Once again, the majority of the data stores perform similarly with this approach, and in the alternative stores, there is no differentiation between fetching the latest or a historical value.

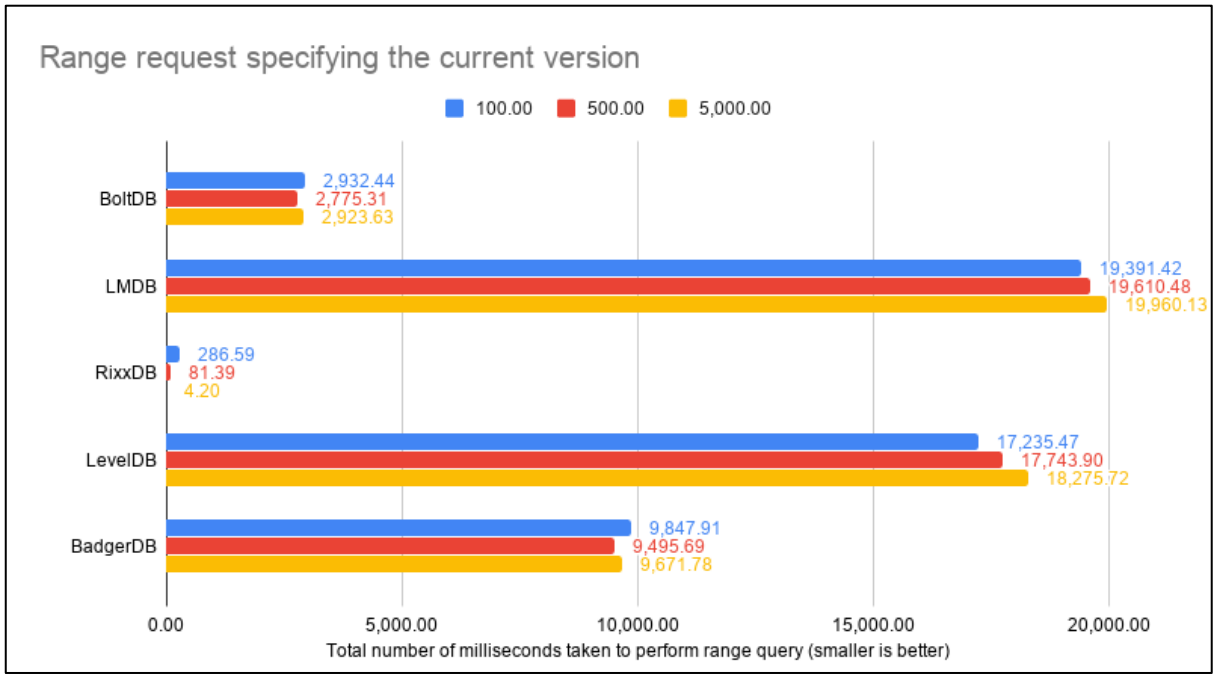

Figure 33 - Range request for all key-value entries, fetching the latest version

In this benchmark, an iterator was used to traverse the entire key space, returning only the latest value for each key. The performance difference between the proposed implementation, and the other key-value stores is distinctly apparent.

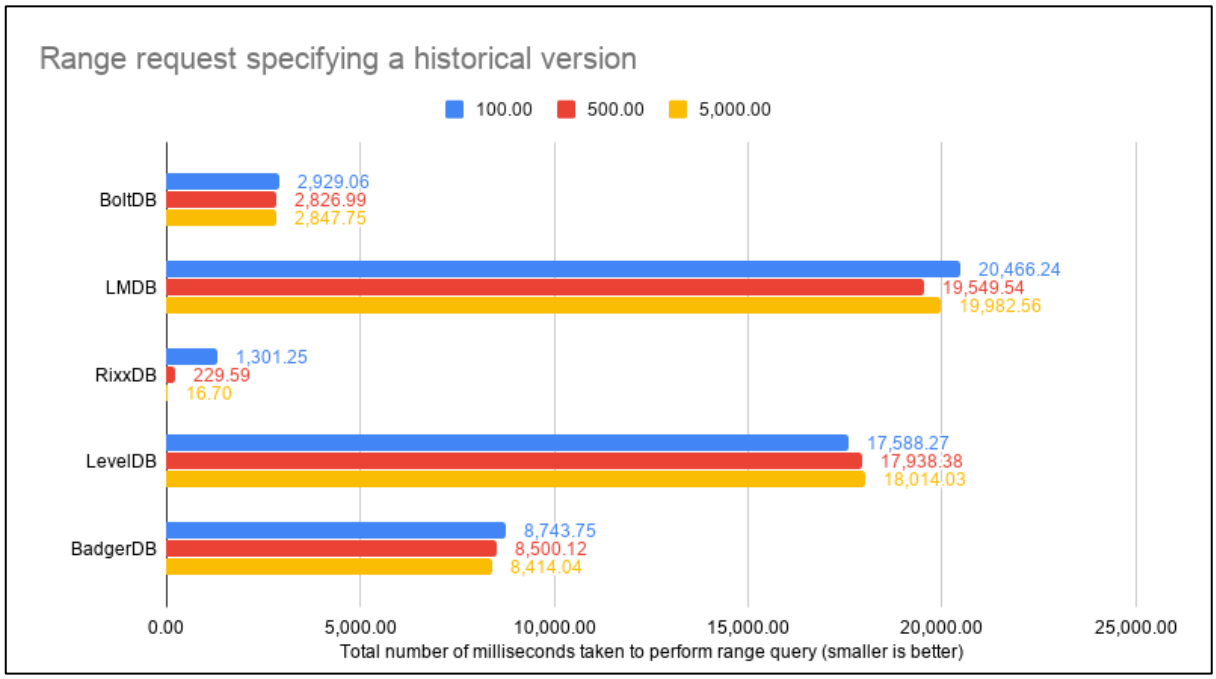

Figure 34 - Range request for all key-value entries, fetching a specific historical version

As with the previous chart, this benchmark iterates over the entire key-space, but returning a specific versioned value for each key. Once again, the contrast in performance is noticeable. In this case, my implementation is slightly slower than the previous benchmark, presumably due to the need to retrieve the value from within the Y-fast trie.

5.4 Evaluation of the implementation

As one can see from the benchmark results, the proposed data structure and key-value store perform similarly to the alternative key-value stores, enabling similar insertion times, and similar performance when it comes to single key-value entry retrieval. With regards to range queries over multi-versioned values, the benefits of the Temporally Augmented Radix Tree are clear, operating most effectively as more versions are added to the store. It is apparent, however, that there are a number of areas where the proposed design and implementation could be improved. While running the benchmarks it became apparent that the in-memory initialization time of teach node's Y-fast trie could potentially be an issue when using the key-value store to store non-versioned items or only a small number of versions for each key-value entry. On the other hand, when storing many different versions within each key-value entry, the benefits of using a Y-fast trie are apparent.

In addition, there are a number of improvements that could be made to the core Radix tree data structure which would improve the overall performance of the key-value store. To begin with, the implementation could be improved to make use of an Adaptive Radix Tree [67] design, making use of different node sizes within the tree to reduce the in-memory space used, but without reducing the overall performance. In extending this, the Adaptive Radix tree can also be made to operate in a persistent manner [68], allowing the copy-on-write characteristics of the current implementation to be retained. Additionally, in order to improve write performance within the tree, one can make use of techniques introduced by Lee et al. for creating Write Optimal Radix Trees [69], which should bring performance gains when performing copy-on-write duplication of nodes within the tree. Finally the Height Optimised Trie Index [70] approach could be used to further reduce the height of the trie leading to both improved memory use, and performance gains.

On reflection, the least performant aspects of the implementation are as a result of the copy-on-write attributes, and memory allocations of the data structure. In order to mitigate these effects, a number of different approaches could be tried. First, the implementation could benefit from a combination of path-based node copying and FAT nodes (a technique introduced by Sleator et al.) to achieve a persistent data structure with an `O(1)` slowdown and an `O(1)` complexity for each modification. Alternatively, the use of techniques introduced in three separate papers: Concurrent Hash Assisted Radix Trees [71]; Concurrent Tries with Efficient Non-Blocking Snapshots [6]; or Efficient Non-blocking Radix Trees [72] would allow for improved concurrent write access to the data structure.

Finally, the choice of Golang as the implementation language of choice resulted in a number of problems due to garbage-collection operations - a problem which is apparent when looking at alternative language benchmarks for tree-like data structures [73]. An initial plan was to implement the key-value store layer within a language which does not use garbage collection (such as Rust), and where the memory overhead of implementing a pointer-base tree structure is more performant. The choice to use Golang, however, meant I could effectively, and fairly, compare the key-value store to a number of alternative implementations, and in addition I was able to develop the prototype using faster agile methodologies.

5.5 Requirements analysis

By benchmarking the key-value store with alternative implementations, we can see that the key-value store successfully retrieves values from the data structure, without suffering from any slowdown with data access. Additionally, as the number of versions stored within each key-value entry increases, the performance of the key-value data store is not affected. When it comes to range requests, the store exceeds the performance of other stores, efficiently traversing the key-value items, and fetching the relevant version from the Y-fast trie, with only a marginal impact on performance. Similarly, to the performance of single-entry retrievals, the range requests perform well when traversing over historical records with only a slight slowdown (this slowdown is because of the need to search the Y-fast trie for the correct predecessor timestamp). As a result, the implementation of the key-value store sufficiently satisfies the final two remaining requirements - **Requirement 12** and **Requirement 13**.

6. Analysis and discussion

6.1 Alternative approaches and improvements

In building the key-value store, the intention was to provide a versioned, immutable data store supporting *transaction time* and *valid time* attributes, which performed at a reasonable or equal level when compared to lesser-functionality alternatives. In addition to some of the points which have been reflected upon within the preceding chapters, there are a number of areas where alternative approaches could be made to the design and implementation in order to effect a different set of characteristics and a different set of performance attributes from the key-value store. In this section I shall look at a number of improvements which could be made to the data store, either to suit a different use case, or in order to improve upon the current implementation.

The first alternative approach is related to the storage of different versions. In an earlier implementation of the key-value store, I chose to store each version as a delta encoding relating to the previous change. With the storage of delta diffs, instead of full values on every change, the memory and storage usage requirements of the data store would decrease, but in turn, the time taken to compute a value at a particular version would increase - as all changes would need to be computed from the initial version, or from the latest snapshot. In effect, this alternative approach is a balance between storage size, and access speed. One final thing to note is that when using deltas to store the changes to a value over time, inserting a value in-between two versions becomes a prohibitively expensive operation, especially if such a change were occurring on multiple entries within the data store.

A secondary approach to data storage therefore would be to use the key-value store to store records in a column or field format, as opposed to a document format. Up to this point, we have presumed that the data stored at `/db/person/1` is a fully encoded database record (encoded using Thrift, CBOR, JSON, or an alternative format), and that in order to retrieve multiple records from the store we would perform a prefix query for entries beginning with `/db/person/`. Alternatively, however, we could store each document field separately, so that `/db/person/1` is stored across multiple different keys - `/db/person/1/name`, `/db/person/1/age`, `/db/person/1/birthday`. With this storage approach, we could still select all records using a range or prefix query, needing to construct the data on the client side, with the added benefit of only storing changes to the specific fields which change, as opposed to storing the entire record on modification.

The third area of improvement involves the aspect of time itself. Within the store as it stands, time (in both *transaction time* and *valid time* cases) is stored as an unsigned 64-bit integer, effectively representing the time in nanoseconds since Unix Epoch time on 1970-01-01T00:00:00 UTC. The use of time in this manner works well for situations which are inserting changes into the data store using current timestamps but does not suit situations where versions may represent events before 1970 or beyond 2554-07-21T11:34:33 UTC. As a result, the versioning to represent time events would be unsuitable for a number of scientific scenarios where historical and future events could easily occur outside of this timeline. If a need for this were to arise, the key-value store could instead make use of 128-bit signed integers for the version identifiers within the store, enabling versions to be stored according to historical and future time. The resulting additional cost to the datastore (in addition to the additional storage size of big-endian encoding for all versions increasing from the 8 bytes to 16 bytes), would be an $O(\log 128)$ complexity compared to the current $O(\log 64)$ for each leaf node version search.

An additional area of improvement, which was briefly mentioned in Section 4.1.2, was the issue with the separation of the layers that make up the data structure, key-value data store, and graph database. As discussed, this separation can lead to inefficiencies and performance issues when it comes to the querying of data from the graph database, as the concept and knowledge of storage is abstracted away from this top layer. As a result, the query layer must fetch data according to the defined API methods, without being able to perform any optimisations by having direct access to the underlying storage. Although the key-value store API implementation enables a wide variety of different approaches to load data, including single queries; prefix based retrieval; range scans; and support for custom cursor based iteration; there may still be a scenario where the query would be more efficiently processed by being located within the storage layer. In addition, when operating in a distributed environment, where the graph database node is stateless - communicating with the key-value storage nodes to retrieve the desired data - this issue is compounded. One improvement, which could mitigate the effect of this layer separation, would be to allow co-processing of data within the storage layer - so that instead of analysing and processing the query externally, leading to a potential excess of data being transferred, instead certain aspects of each query could be passed directly to the storage layer. With this addition, the storage layer would need to have some understanding of the data (instead of just storing binary data as it does at the moment), but this approach could lead to the filtering of data being effected within the key-value store, before any data is transmitted over the network, substantially reducing network transmission and costs.

6.2 Broader issues and further development of this work

In Section 4.1.1 we looked at how the key-value store could be made to work in a distributed environment, introducing a highly-available architecture, using Raft [51], with which to approach this requirement. However, although this method leads to a higher availability architecture than a single node can offer, the issue still remains, where the dataset cannot be larger than the storage and memory capabilities of the smallest member node in the cluster, and ideally a sharded and partitioned approach should be followed. With the growing storage demands of an immutable database, this problem is compounded. Due to the requirements of providing a versioned, ordered key-value store, there are a number of complexities involved when scaling this across a distributed environment. An approach, which can be seen in databases such as Google Spanner [20], CockroachDB [21], and TiKV [22], is to split the store into a number of ranges, with each range being synced, managed, and rebalanced across the cluster. In contrast to splitting up the key-value store into smaller ranges, an alternative approach would be to cluster the distributed nodes in a hash ring, and use consistent hashing [74] to store each edge node of the radix tree across the cluster, in a similar manner to a Prefix Hash Tree [75]. This is undoubtedly a much greater topic of research that could form the basis of future work.

Throughout this project I have approached the topic of versioning within a key-value store, enabling access to versioned data by *valid time* and *transaction time*. Although the implementation, as it stands, satisfies the requirements for the use cases in which it was originally developed, there are a number of queries that are not yet supported, but whose implementation could benefit the success of the temporal key-value store. Currently a query such as the one below would need to iterate over all entries prefixed by **`/db/person/`** and scan over all versions of each entry, before returning the response.

```
SELECT * FROM person WHERE surname WAS "Smith";
```

In a similar manner, the following query would need to iterate over all entries prefixed by **`/db/person/`** making use of further iteration through the range of requested versions.

```
SELECT * FROM person WHERE surname WAS "Smith" BETWEEN "2019-01-01" AND
"2019-06-01";
```

As is evident from these relatively simple queries, both cross-temporal queries, and queries which perform comparative analysis over historical event data, would benefit from the introduction of improved temporal indexing, and possibly even the support of temporal reasoning methods such as Allen’s Interval Algebra [76]. This is another area which could benefit from future development.

6.3 Ethical, legal, and professional considerations

There are a number of considerations which need to be borne in mind with respect to this project with regards to data privacy, and data security. First of all, with the advent of the EU GDPR in May 2018, collecting, managing, and analysing big datasets [77] has become not only a technical design decision, but also a legal one [78]. As a result, given that I have proposed a key-value store for storing data in an immutable way, with the ability to see all versions as they were stored in a past version, the issue of data privacy, and its impact on businesses [79], has an impact on the usability of this implementation. Storing all changes that have been made to data over time, can lead to the possibility that the 'right to be forgotten' is not possible to adhere to - leaving a digital footprint which can be queried and analysed without ever being destroyed. Currently the implementation allows for clearing all versions of a key or a set of keys from the store; however, this process does not affect the *transaction time* aspect of the store, where each transaction yields a different version. This is an area which needs to be considered further.

In addition, as a large amount of historical and current data is likely to be stored in a temporal data store, the issue of data security is paramount. In the implementation in this project, the data store enables encryption and decryption of values, but keys must remain unencrypted, as the sorted ordering of the key-value store would be affected. Future considerations could be given to implementing order-preserving encryption [80] which would enable the keys to remain sorted within the data structure. Although many implementations of order-preserving encryption can lead to leakage of data [81], this would still be a benefit when compared with the current implementation.

7. Conclusion

Despite a plethora of available databases, each with its own characteristics and advantages, there is clearly still a need for an optimised embedded key-value data store which can address the unrelenting need for improved temporal data access characteristics with advanced querying functionality. In this dissertation I have explored the area of temporal, immutable data storage, within the context of data stores with a specific focus on graph databases, with a view to proposing a new data structure - the Temporally Augmented Radix Tree, which could be implemented within an embedded key-value store, for the storage and retrieval of bi-temporal data.

A primary objective of this project was to make use of software engineering methodologies to analyse the different use cases and gather the necessary requirements for the implementation of such a prototype key-value store. I consider how such a store could be implemented within the larger context of a distributed data store by using a service-oriented architecture approach. After examining the specific use cases of both temporal data, and temporal data within a graph database, I present the requirements which would need to be met in order for an implementation to operate effectively for the specified scenarios.

Having assessed the performance of the implementation by comparing and benchmarking it against other widely-used key-value stores, I concluded that the proposed approach had benefits when querying temporal data, but it also brought about some challenges. This prompted me to further reflect upon the design decisions of the implementation, analysing where the problems lay, and finally where improvements could be made.

Bibliography

[1] ‘The rise of immutable data stores’, *Next In Tech*, 08-Sep-2015. .

[2] D. S. Rosenthal, D. C. Rosenthal, E. L. Miller, I. F. Adams, M. W. Storer, and E. Zadok, ‘The economics of long-term digital storage’, *Memory of the World in the Digital Age, Vancouver, BC*, 2012.

[3] ‘Immutability, MVCC, and Garbage Collection’, *Immutability, MVCC, and Garbage Collection*. [Online]. Available: https://www.xaprb.com/blog/2013/12/28/immutability-mvcc-and-garbage-collection/. [Accessed: 17-Oct-2019].

[4] P. Helland, ‘Immutability Changes Everything.’, *ACM Queue*, vol. 13, no. 9, p. 40, 2015.

[5] P. L. Lehman, ‘Efficient locking for concurrent operations on B-trees’, *ACM Transactions on Database Systems (TODS)*, vol. 6, no. 4, pp. 650–670, 1981.

[6] A. Prokopec, N. G. Bronson, P. Bagwell, and M. Odersky, ‘Concurrent tries with efficient non-blocking snapshots’, in *Acm Sigplan Notices*, 2012, vol. 47, pp. 151–160.

[7] P. O’Neil, E. Cheng, D. Gawlick, and E. O’Neil, ‘The log-structured merge-tree (LSM-tree)’, *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996.

[8] P. A. Bernstein, P. A. Bernstein, and N. Goodman, ‘Concurrency control in distributed database systems’, *ACM Computing Surveys (CSUR)*, vol. 13, no. 2, pp. 185–221, 1981.

[9] T. Haerder and A. Reuter, ‘Principles of transaction-oriented database recovery’, *ACM computing surveys (CSUR)*, vol. 15, no. 4, pp. 287–317, 1983.

[10] D. M. Dooley, A. J. Petkau, G. Van Domselaar, and W. W. Hsiao, ‘Sequence database versioning for command line and Galaxy bioinformatics servers’, *Bioinformatics*, vol. 32, no. 8, pp. 1275–1277, 2015.

[11] Z. Shi and R. Shibasaki, ‘GIS DATABASE REVISION--THE PROBLEMS AND SOLUTIONS’, *International Archives of Photogrammetry and Remote Sensing*, vol. 33, no. B2; PART 2, pp. 494–501, 2000.

[12] F. Urbano and F. Cagnacci, Eds., *Spatial Database for GPS Wildlife Tracking Data: A Practical Guide to Creating a Data Management System with PostgreSQL/PostGIS and R*. Springer International Publishing, 2014.

[13] J. Perret, A. Boffet Mas, and A. Ruas, ‘Understanding Urban Dynamics: the use of vector topographic databases and the creation of spatio-temporal databases’, in *24th international cartography conference (icc’09)*, 2009.

[14] C. M. Saracco, M. Nicola, and L. Gandhi, ‘A matter of time: Temporal data management in DB2 for z’, *IBM Corporation, New York*, 2010.

[15] T. Nash and A. Olmsted, ‘Performance vs. security: Implementing an immutable database in MySQL’, in *2017 12th International Conference for Internet Technology and Secured Transactions (ICITST)*, 2017, pp. 290–291.

[16] R. T. Snodgrass, S. S. Yao, and C. Collberg, ‘Tamper Detection in Audit Logs’, in *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, Toronto, Canada, 2004, pp. 504–515.

[17] R. A. K. Duncan and M. Whittington, ‘Creating an Immutable Database for Secure Cloud Audit Trail and System Logging’, in *Eighth International Conference on Cloud Computing, GRIDs, and Virtualization, 19 February 2017-23 February 2017, Athens, Greece*, 2017.

[18] S. Mitra, M. Winslett, R. T. Snodgrass, S. Yaduvanshi, and S. Ambokar, ‘An architecture for regulatory compliant database management’, in *Proceedings of the 2009 IEEE International Conference on Data Engineering*, 2009, pp. 162–173.

[19] ‘RocksDB | A persistent key-value store’, *RocksDB*. [Online]. Available: http://rocksdb.org/. [Accessed: 17-Oct-2019].

[20] J. C. Corbett *et al.*, ‘Spanner: Google’s globally distributed database’, *ACM Transactions on Computer Systems (TOCS)*, vol. 31, no. 3, p. 8, 2013.

[21] ‘Cockroach Labs, the company building CockroachDB’, *Cockroach Labs*. [Online]. Available: https://www.cockroachlabs.com/. [Accessed: 17-Oct-2019].
[22] ‘TiKV’. [Online]. Available: https://tikv.org/. [Accessed: 17-Oct-2019].
[23] S. Loesing, M. Pilman, T. Etter, and D. Kossmann, ‘On the design and scalability of distributed shared-data databases’, in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 2015, pp. 663–676.
[24] R. T. Snodgrass, ‘Temporal databases’, in *Theories and methods of spatio-temporal reasoning in geographic space*, Springer, 1992, pp. 22–64.
[25] V. Kouramajian, I. Kamel, R. Elmasri, and S. Waheed, ‘The time index+: an incremental access structure for temporal databases’, 1994.
[26] ‘InfluxDB: Purpose-Built Open Source Time Series Database’, *InfluxData*. [Online]. Available: https://www.influxdata.com/. [Accessed: 17-Oct-2019].
[27] P. Dix, ‘[New] InfluxDB Storage Engine | Time Structured Merge Tree’, *InfluxData*, 07-Oct-2015. [Online]. Available: https://www.influxdata.com/blog/new-storage-engine-time-structured-merge-tree/. [Accessed: 17-Oct-2019].
[28] A. Kiel, ‘Datomic-a functional database’, 2013.
[29] M. Haeusler, T. Trojer, J. Kessler, M. Farwick, E. Nowakowski, and R. Breu, ‘ChronoGraph: A Versioned TinkerPop Graph Database’, in *International Conference on Data Management Technologies and Applications*, 2017, pp. 237–260.
[30] M. Haeusler, ‘Scalable Model Versioning, Querying & Persistence’.
[31] ‘Introducing Badger: A fast key-value store written purely in Go - Dgraph Blog’. [Online]. Available: https://blog.dgraph.io/post/badger/. [Accessed: 17-Oct-2019].
[32] K. Kulkarni and J.-E. Michels, ‘Temporal features in SQL: 2011’, *ACM Sigmod Record*, vol. 41, no. 3, pp. 34–43, 2012.
[33] ‘Microsoft Data Platform | Microsoft’, *Microsoft SQL Server - GB (English)*. [Online]. Available: https://www.microsoft.com/en-gb/sql-server/default.aspx. [Accessed: 17-Oct-2019].
[34] CarlRabeler, ‘Temporal Table Considerations and Limitations - SQL Server’. [Online]. Available: https://docs.microsoft.com/en-us/sql/relational-databases/tables/temporal-table-considerations-and-limitations. [Accessed: 17-Oct-2019].
[35] ‘Kevin Mahoney: Database Design: Immutable Data’. [Online]. Available: http://kevinmahoney.co.uk/articles/immutable-data/. [Accessed: 17-Oct-2019].
[36] ‘Event Store’. [Online]. Available: https://eventstore.org/. [Accessed: 17-Oct-2019].
[37] D. Betts, J. Dominguez, G. Melnik, F. Simonazzi, and M. Subramanian, *Exploring CQRS and Event Sourcing: A Journey into High Scalability, Availability, and Maintainability with Windows Azure*, 1st ed. Microsoft patterns & practices, 2013.
[38] Arnaud Castelltort and A. Laurent, ‘Representing history in graph-oriented nosql databases: A versioning system’, in *Eighth International Conference on Digital Information Management (ICDIM 2013)*, 2013, pp. 228–234.
[39] ‘Symas Lightning Memory-mapped Database’, *Symas Corporation*. [Online]. Available: https://symas.com/lmdb/. [Accessed: 17-Oct-2019].
[40] ‘Oracle Berkeley DB | Oracle United Kingdom’. [Online]. Available: https://www.oracle.com/uk/database/technologies/related/berkeleydb.html. [Accessed: 17-Oct-2019].
[41] M. Haeusler, ‘Scalable versioning for key-value stores’, in *Proceedings of the 5th International Conference on Data Management Technologies and Applications*, 2016, pp. 79–86.
[42] W. D. Vijitbenjaronk, J. Lee, T. Suzumura, and G. Tanase, ‘Scalable time-versioning support for property graph databases’, in *2017 IEEE International Conference on Big Data (Big Data)*, 2017, pp. 1580–1589.
[43] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan, ‘Making data structures persistent’, *Journal of computer and system sciences*, vol. 38, no. 1, pp. 86–124, 1989.
[44] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer, ‘An asymptotically optimal multiversion B-tree’, *The VLDB Journal—The International Journal on Very Large Data Bases*, vol. 5, no. 4, pp. 264–275, 1996.

[45] D. R. Morrison, ‘PATRICIA—Practical Algorithm To Retrieve Information Coded in Alphanumeric’, *J. ACM*, vol. 15, no. 4, pp. 514–534, Oct. 1968.
[46] ‘WiredTiger Storage Engine — MongoDB Manual’, *https://github.com/mongodb/docs/blob/master/source/core/wiredtiger.txt*. [Online]. Available: https://docs.mongodb.com/manual/core/wiredtiger. [Accessed: 27-Oct-2019].
[47] *boltdb/bolt*. BoltDB, 2019.
[48] D. Lomet, R. Barga, M. F. Mokbel, and G. Shegalov, ‘Transaction time support inside a database engine’, in *22nd International Conference on Data Engineering (ICDE’06)*, 2006, pp. 35–35.
[49] *google/leveldb*. Google, 2019.
[50] M. Hilbert and P. López, ‘The World’s Technological Capacity to Store, Communicate, and Compute Information’, *Science*, vol. 332, no. 6025, pp. 60–65, Apr. 2011.
[51] D. Ongaro and J. Ousterhout, ‘In search of an understandable consensus algorithm’, in *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, 2014, pp. 305–319.
[52] ‘FoundationDB’s Lesson: A Fast Key-Value Store is Not Enough’, *VoltDB*, 01-Apr-2015. .
[53] D. Comer, ‘Ubiquitous B-tree’, *ACM Computing Surveys (CSUR)*, vol. 11, no. 2, pp. 121–137, 1979.
[54] ‘MySQL’. [Online]. Available: https://www.mysql.com/. [Accessed: 27-Oct-2019].
[55] ‘PostgreSQL: The world’s most advanced open source database’. [Online]. Available: https://www.postgresql.org/. [Accessed: 27-Oct-2019].
[56] A. Guttman, *R-trees: A dynamic index structure for spatial searching*, vol. 14. ACM, 1984.
[57] R. Bliujute, C. S. Jensen, S. Saltenis, and G. Slivinskas, ‘R-tree based indexing of now-relative bitemporal data’, in *VLDB*, 1998, vol. 98, pp. 345–356.
[58] W. Pugh, ‘Skip lists: a probabilistic alternative to balanced trees’, *Communications of the ACM*, vol. 33, no. 6, 1990.
[59] D. E. Willard, ‘Log-logarithmic worst-case range queries are possible in space Θ(N)’, *Information Processing Letters*, vol. 17, no. 2, pp. 81–84, Aug. 1983.
[60] R. C. Merkle, ‘Method of providing digital signatures’, US4309569 (A), 05-Jan-1982.
[61] M. Polte, J. Simsa, and G. Gibson, ‘Comparing performance of solid state devices and mechanical disks’, in *2008 3rd Petascale Data Storage Workshop*, 2008, pp. 1–7.
[62] V. Kasavajhala, ‘Solid state drive vs. hard disk drive price and performance study’, *Proc. Dell Tech. White Paper*, pp. 8–9, 2011.
[63] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, ‘WiscKey: Separating Keys from Values in SSD-conscious Storage’, presented at the 14th {USENIX} Conference on File and Storage Technologies ({FAST} 16), 2016, pp. 133–148.
[64] C. Vicknair, D. Wilkins, and Y. Chen, ‘MySQL and the Trouble with Temporal data’, in *Proceedings of the 50th Annual Southeast Regional Conference*, 2012, pp. 176–181.
[65] ‘snappy’, *snappy*. [Online]. Available: http://google.github.io/snappy/. [Accessed: 01-Nov-2019].
[66] ‘ObjectId — MongoDB Manual’, *https://github.com/mongodb/docs/blob/master/source/reference/method/ObjectId.txt*. [Online]. Available: https://docs.mongodb.com/manual/reference/method/ObjectId. [Accessed: 01-Nov-2019].
[67] V. Leis, A. Kemper, and T. Neumann, ‘The adaptive radix tree: ARTful indexing for main-memory databases.’, in *ICDE*, 2013, vol. 13, pp. 38–49.
[68] A. Dave, J. E. Gonzalez, M. J. Franklin, and I. Stoica, ‘Persistent Adaptive Radix Trees: Efficient Fine-Grained Updates to Immutable Data’.
[69] S. K. Lee, K. H. Lim, H. Song, B. Nam, and S. H. Noh, ‘{WORT}: Write Optimal Radix Tree for Persistent Memory Storage Systems’, in *15th {USENIX} Conference on File and Storage Technologies ({FAST} 17)*, 2017, pp. 257–270.

[70] R. Binna, E. Zangerle, M. Pichl, G. Specht, and V. Leis, ‘HOT: A Height Optimized Trie Index for Main-Memory Database Systems’, in *Proceedings of the 2018 International Conference on Management of Data*, New York, NY, USA, 2018, pp. 521–534.
[71] W. Pan, T. Xie, and X. Song, ‘HART: A Concurrent Hash-Assisted Radix Tree for DRAM-PM Hybrid Memory Systems’, in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2019, pp. 921–931.
[72] V. Velamuri, ‘Efficient Non-blocking Radix Trees’, 2017, pp. 565–579.
[73] ‘binary-trees | Computer Language Benchmarks Game’. [Online]. Available: https://benchmarksgame-team.pages.debian.net/benchmarksgame/performance/binarytrees.html. [Accessed: 01-Nov-2019].
[74] D. Karger *et al.*, ‘Web caching with consistent hashing’, *Computer Networks*, vol. 31, no. 11–16, pp. 1203–1213, 1999.
[75] S. Ramabhadran, S. Ratnasamy, J. M. Hellerstein, and S. Shenker, ‘Prefix hash tree: An indexing data structure over distributed hash tables’, in *Proceedings of the 23rd ACM symposium on principles of distributed computing*, 2004, vol. 37.
[76] J. F. Allen, ‘Maintaining knowledge about temporal intervals’, in *Readings in qualitative reasoning about physical systems*, Elsevier, 1990, pp. 361–372.
[77] T. Z. Zarsky, ‘Incompatible: The GDPR in the age of big data’, *Seton Hall L. Rev.*, vol. 47, p. 995, 2016.
[78] U. Pagallo, ‘The Legal Challenges of Big Data: Putting Secondary Rules First in the Field of EU Data Protection’, *Eur. Data Prot. L. Rev.*, vol. 3, p. 36, 2017.
[79] C. Tankard, ‘What the GDPR means for businesses’, *Network Security*, vol. 2016, no. 6, pp. 5–8, Jun. 2016.
[80] A. Boldyreva, N. Chenette, Y. Lee, and A. O’neill, ‘Order-preserving symmetric encryption’, in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, 2009, pp. 224–241.
[81] N. Chenette, K. Lewi, S. A. Weis, and D. J. Wu, ‘Practical Order-Revealing Encryption with Limited Leakage’, in *Fast Software Encryption*, Berlin, Heidelberg, 2016, pp. 474–493.

Glossary

ACID (Atomicity, Consistency, Isolation, Durability)
A set of properties for database transactions, which define a sequence of operations and functionality for modifying data within the context of a database, to ensure a minimum level of guarantees to the dataset.

API (Application Programming Interface)
An interface that defines the communication parameters between a client and a server, or a client and a programming library, with the intention of simplifying the implementation.

DB (Database)
A database is a data storage system used to store, update, and query the data on a computer system.

DBMS (Database Management System)
Software that handles the storage, retrieval, and updating of data in a computer system.

GDPR (General Data Protection Regulation)
The EU GDPR is a regulation in EU law on data protection and privacy for all individual citizens of the European Union.

HDD (Hard Disk Drive)
A non-volatile, electro-magnetic computer or server storage medium, which uses spinning disks and a magnetic storage head to read and write data. These devices are typically slower than solid-state disks.

IoT (Internet of Things)
The system of interconnectivity of devices across the internet. These devices are usually used to generate sensor data for the systems in which they are embedded.

KV (Key Value)
A key-value database, or key-value store, is a data storage system used for storing, retrieving, and managing a mapping of keys with values.

RAFT (Distributed Consensus Algorithm)
Raft is a distributed consensus algorithm, similar to Paxos, which is designed to simplify the management of state across nodes in a distributed cluster.

RAM (Random access memory)
Computer or server volatile memory, used to store working data, rather than data that needs to be persisted.

SSD (Solid State Drive)
A computer or server storage device containing non-volatile flash memory, used in place of a hard disk due to its much greater speed.