Aalto University
School of Science

Master's programme in Computer, Communication and Information Sciences

Extending a data management system with multiversion indexing, branching, and snapshots

Filip Stenbacka

Master's Thesis
2024

Aalto University
School of Science

Author Filip Stenbacka

Title Extending a data management system with multiversion indexing, branching, and snapshots

Degree programme Computer, Communication and Information Sciences

Major Machine Learning, Data Science and Artificial Intelligence

Supervisor DSc Kerttu Pollari-Malmi

Advisors MSc Teemu Rantanen, MSc Kim Nyberg

Collaborative partner Trimble Solutions Corporation

Date 27.05.2024 **Number of pages** 72+1 **Language** English

Abstract

Software developers have long been accustomed to using version control in their projects for managing files, but similar solutions for databases are not as well established. This thesis extends a pre-existing proprietary data management system, called Lumo, with snapshots and branches for version control purposes. Additionally, a suitable multiversion index structure is needed to index data in each snapshot (version), and certain key performance criteria are recognized for the structure to adhere to. The criteria demand efficient loading, releasing and cloning of snapshots, as well as efficient access to any data in any snapshot. Furthermore, the amount of shared data between snapshots should be maximized to reduce memory consumption.

In this thesis the pre-existing Hash Array Mapped Trie (HAMT) is identified to be a suitable base data structure for the multiversion index. The Multiversion Hash Array Mapped Trie (MHAMT) is introduced as an extension to the HAMT, specifically implemented for multiversion use within Lumo. The MHAMT allows nodes to be shared between several tries, and also adds support for non-unique keys and hash collisions. The MHAMT is shown to satisfy the key performance criteria well.

The MHAMT implementation is validated by testing it using real-world data, resulting in a Lumo data model consisting of 769 snapshots, each with their own MHAMT index. The memory consumption of the MHAMTs are analyzed, and it is shown that the MHAMTs consume around 90 % less memory compared to the theoretical case where nodes cannot be shared between tries. The trie structures are examined as well, which shows the MHAMTs contain more nodes than necessary. The reason for the unbalanced tries is thought to be the hashing algorithm used by Lumo not being optimized for the test data.

The MHAMT implementation introduced in this thesis can still be enhanced. Path compression could be applied to the tries by removing unnecessary one-child nodes, and full nodes could have their bitmaps removed completely. The proposed solution of managing duplicate keys could be improved further, and adding multiversion indexing support for ordered data will also be a priority in future Lumo development.

Keywords database, multiversion, index, branch, snapshot, MHAMT

Aalto-universitetet
Högskolan för
teknikvetenskaper

Författare Filip Stenbacka

Titel Komplettering av ett datahanteringssystem med multiversionsindex, förgreningar och ögonblicksbilder

Utbildningsprogram Computer, Communication and Information Sciences

Huvudämne Machine Learning, Data Science and Artificial Intelligence

Övervakare TkD Kerttu Pollari-Malmi

Handledare DI Teemu Rantanen, DI Kim Nyberg

Samarbetspartner Trimble Solutions Corporation

Datum 27.05.2024 **Sidantal** 72+1 **Språk** Engelska

Sammandrag

Mjukvaruutvecklare har sedan länge varit vana med versionskontroll inom sina projekt för att hantera filer, men liknande lösningar för databaser är inte lika väletablerade. Detta diplomarbete kompletterar ett redan existerande proprietärt datahanteringssystem, kallat Lumo, med ögonblicksbilder och förgreningar för versionskontrollsändamål. Dessutom behövs en lämplig multiversionsindexstruktur för att indexera datan i varje ögonblicksbild (version), och några nyckelprestandakriterier identifieras som strukturen bör följa. Kriterierna kräver effektiv laddning, frigivning och kloning av ögonblicksbilder, samt effektiv åtkomst till alla data i alla ögonblicksbilder. Därtill bör mängden delad data mellan ögonblicksbilder maximeras för att reducera minnesförbrukningen.

I detta diplomarbete identifieras den existerande datastrukturen Hash Array Mapped Trie (HAMT) som en lämplig basstruktur för multiversionsindexet. Datastrukturen Multiversion Hash Array Mapped Trie (MHAMT) introduceras, som är en utvidgad HAMT specifikt implementerad för multiversionsanvändning inom Lumo. MHAMT tillåter att noder delas mellan flera träd, och lägger även till stöd för icke-unika nyckelvärden och hashkollisioner. MHAMT visar sig tillfredsställa nyckelprestandakriterierna.

MHAMT-implementationen valideras genom att testa den med hjälp av verklig data, vilket resulterar i en Lumo-datamodell bestående av 769 ögonblicksbilder, som vardera har sitt egna MHAMT-index. Minnesförbrukningen för MHAMT-träden analyseras och det visas att förbrukningen är cirka 90 % mindre jämfört med det teoretiska fallet där noder inte kan delas mellan träden. Trädstrukturerna undersöks också, vilket visar att träden innehåller fler noder än nödvändigt. Anledningen till de obalanserade MHAMT-träden antas vara att hashalgoritmen som används av Lumo inte är optimerad för testdatan.

MHAMT-implementationen som introduceras i detta diplomarbete kan fortfarande förbättras. Vägkomprimering kan tillämpas på MHAMT-träden genom att avlägsna onödiga enbarnsnoder, och kompletta noder kan få sina bitmapobjekt helt borttagna. Även den föreslagna lösningen för att hantera nyckeldubbletter kan förbättras ytterligare, och att inkorporera stöd för multiversionsindexering för sorterad data kommer också att vara en prioritet inom den framtida Lumo-utvecklingen.

Nyckelord databas, multiversion, index, förgrening, ögonblicksbild, MHAMT

Preface

I want to thank my advisors Teemu Rantanen and Kim Nyberg, as well as Trimble Solution Corporation, for the opportunity to work on this interesting topic, and for making it possible to work on the thesis while doing other work. I also want to thank my supervisor Kerttu Pollari-Malmi for her incredibly patient guidance, and the regularly organized meetings that were, at times, the only things driving my work forward.

This thesis has been long in the making, immense thanks to Teknologföreningen, especially Urdsgjallar and Urds Maskinister v.k., for not allowing me finish it sooner. Thanks to Alexandra for her unconditional support, and for proof-reading the final product. Finally, I want to thank Nudel and Müžłì for being so fluffy.

Otnäs, 27.05.2024

Filip Stenbacka

Contents

Abbreviations

AMT	Array Mapped Trie
API	Application programming interface
BIM	Building information modeling
BST	Binary search tree
BT-tree	Branched and Temporal Tree
CHAMP	Compressed Hash-Array Mapped Prefix-tree
CLI	Command line interface
COW	Copy-on-write
DBMS	Database management system
DBV	Database version
GIS	Geographic information system
GUID	Globally unique identifier
HAMT	Hash Array Mapped Trie
IoT	Internet of Things
IP	Internet Protocol
JS	JavaScript
JVM	Java Virtual Machine
LC-trie	Level-compressed trie
LHAM	Log-structured history data access method
LPC-trie	Level and path compressed trie
LSM-tree	Log-structured merge-tree
LSMV-tree	Log-structured multiversion tree
MHAMT	Multiversion Hash Array Mapped Trie
MiB	Mebibyte, 2^{20} bytes
MV-IDX	Multi-Version Index
MVB-tree	Multiversion B-tree
MVCC	Multiversion concurrency control
OB+tree	Overlapping B+tree
OOD	Object-oriented database
ORM	Object-relational mapping
PTS	Partitioned B-tree
ROW	Redirect-on-write
SI	Snapshot isolation
SQL	Structured query language
TS	Tekla Structures
TS-LSM-tree	Time-stamped log-structured merge-tree
TSB-tree	Time-Split B-tree
TST	Ternary search tree
UI	User interface

1 Introduction

Trimble Inc. is an American industry technology company which aims to connect the physical and digital worlds through hardware and software solutions across the agriculture, construction, geospatial and transportation sectors [1]. Trimble acquired the industry-leading and Espoo-based construction software company Tekla in 2012, which for decades have had a whole unit dedicated to developing and maintaining libraries, or components, for the rest of the company to use in their products. The oldest of these components which are still in use is a data management component that implements an in-memory database management system (DBMS). The DBMS has been designed for generic data, but has mostly been used for building information modeling (BIM) and geographic information system (GIS) data.

Recent efforts have been made to bring the now timeworn data management component into this century, not only to enhance the feature set for the current users, but also to attract a new and wider userbase inside Trimble. The new development focuses on creating an entirely new data management component, called Lumo, from the ground up. With only a small team working on Lumo, development is relatively slow, but the plan is to eventually be able to completely replace the old data management component with Lumo in all Trimble products which currently use it, while also targeting new use cases, but extra care has been taken to make the transition as painless as possible. Lumo in its current state can already be used as a DBMS, but not to the same extent as the old component.

This thesis focuses on a narrow slice of the development of Lumo, which is the design, implementation and testing of extending the Lumo database with multiversion capabilities. More specifically, the goal is to implement support for efficient snapshots and branches using purposely designed multiversion indexes. Here, snapshots are read-only views into certain database states, and branches are a way of modifying data starting from a certain snapshot. Git [2] is a commonly used analogy when describing how users will perceive and use Lumo multiversion capabilities.

This thesis is organized as follows. Chapter 2 begins by describing the current data management component in more detail, as well as the reasons for starting developing Lumo. The chapter also gives a detailed description of Lumo itself, its internal workings, and the various development phases of the project.

Chapters 3 and 4 delve into the main topics of this thesis in more detail. Related work and current implementations regarding snapshots, branches, multiversion databases, and multiversion indexes are discussed, as well as providing descriptions of various use cases, both general ones and ones more specific to Lumo. Some application programming interface (API) considerations and requirements surrounding the new implementations themselves are also discussed.

Chapter 5 starts off by describing the existing interfaces for the Lumo database classes, and continues with descriptions of the interfaces for the new classes which are added for Lumo's new multiversion capabilities. Some implementation details surrounding the interfaces are also discussed. After that the multiversion index chosen for Lumo is justified, and the implementation described in detail.

In Chapter 6 the new Lumo multiversion capabilities are benchmarked by measuring

memory usage and examining the index structure using real-world BIM data. Chapter 7 discusses the results of the experiment and describes the future direction of Lumo development. Lastly, the thesis is summarized and conclusions drawn in Chapter 8.

2 Background

2.1 Trimble Solutions and Tekla

This thesis is written in collaboration with Trimble Solutions, for whom I have worked since the summer of 2019. The company was formerly known as Tekla before being acquired by Trimble Navigation (now Trimble Inc.) back in 2012, but changed its name to Trimble Solutions after the acquisition. Tekla had been developing software for the construction industry since the late 1960s and had established itself as a well-known figure in the industry well before the acquisition. Tekla's accomplishments are in part due to the success of Tekla Structures (TS), its industry-leading BIM modeling software for structural engineering and construction, which to this day still is the largest project being worked on in the current office.

2.2 Data management system in Tekla products

Building software by utilizing ready-made components or building blocks is nowadays common practice, but Tekla adopted this strategy for its products at a very early stage. Separating responsibilities such as user interface (UI), graphics rendering, and data management into distinct components and reusing them between products leads to less duplicate code being written and to an overall more efficient product development. Tekla had, as Trimble Solutions still has to this day, a whole unit dedicated to researching new technologies and developing and maintaining components and libraries, which are made available for the rest of the company to use within their products, for example Tekla Structures.

One of the original components in Tekla products is the data management component, which was introduced in the late 1980s. It has since then grown into a comprehensive database management system (DBMS) split into several components with different levels of abstraction, ranging from the original low-level component to more recent higher-level components. The DBMS is centered around a proprietary relational database called DBvirtual, which can handle authoring of large and complex data models with generic data, but has traditionally been used for mainly GIS and BIM data. DBvirtual is an in-memory database, but also supports saving data to disk. It is also an embedded database, meaning it is run closely integrated with the application using it, usually within the same process. The DBMS includes an object-relational mapping (ORM) layer which gives users an easier and more modern way of interacting with the data. The ORM is written separately in three languages, namely C++, C# and JavaScript, but all implementations provide an identical API. The DBMS does not use structured query language (SQL), but has its own schema description language for data models and uses its own query engine for building queries within the applications. Typed data accessor classes can be generated for all three languages based on the database schema.

The DBMS lacks certain features compared to more modern data management solutions, especially with regards to which data types which are natively supported. For example, the only ”non-primitive” data types available are fixed size arrays of

chars and bytes. Defining and using a data model in a dynamic way is also very cumbersome. Implementing new features to mitigate the shortcomings would be next to impossible in a legacy code base such as this, and long-term maintenance is also an open question which would need to be solved.

While there are many Trimble products using DBvirtual currently, Tekla Structures is the most prominent one, which leads this thesis to focus heavily on use cases for TS specifically. Tekla Structures is used to create BIM models which represent various real-world objects, such as a building including everything from individual steel beams to concrete details. These BIM models, here referred to as TS models, can easily be several gigabytes in size. When a TS model is opened in Tekla Structures it is loaded, either fully or partly, into the DBvirtual database, from where the BIM data is fetched to be displayed on the screen. The state of the TS model in the database is seen as the ground truth and all modifications to the model are made directly to data in the database. DBMS performance in terms of speed and memory usage are both very important metrics when rendering objects on screen directly from a database, which is why DBvirtual was made to be an embedded in-memory database. DBvirtual also implements a decentralized change management system for multi-user data model editing, supporting offline usage, which is the base technology used in Tekla Model Sharing [3], an occasionally connected collaborative tool allowing multiple users to work on the same TS model simultaneously.

2.3 Recent development: Lumo, phase 1

A new set of components is in the works under the umbrella name Lumo. They are meant to be used both on their own and in conjunction with DBvirtual. See Figure 1 for a simplified overview of the components and how they relate to each other. The plan is for the Lumo components to partly, or in some cases completely, replace older components, while bringing much welcomed new features to the current data management system. The development of Lumo is being done in phases, where each development phase has their own goals set out in advance. The first development phase consists of getting the basic functionality working for each component and integrating them with DBvirtual. It can be difficult to define where one phase ends and the next starts, but it is safe to say that phase one is more or less complete, and the results from it have already been taken into use. Work towards this thesis falls under phase two (see Section 2.4). It must be noted that Lumo is a complex piece of software with many moving parts, and since the intricate internal details of Lumo is not the focus of this thesis, descriptions of Lumo's inner workings will be presented in a simplified manner.

2.3.1 Lumo.Core

The main component in the new development is called Lumo.Core. It allows users to define data models using a rich type schema, and create, store and modify objects according to their own schema. The component is divided into separate smaller units (classes), each with their own responsibilities, allowing for easier testing, modification

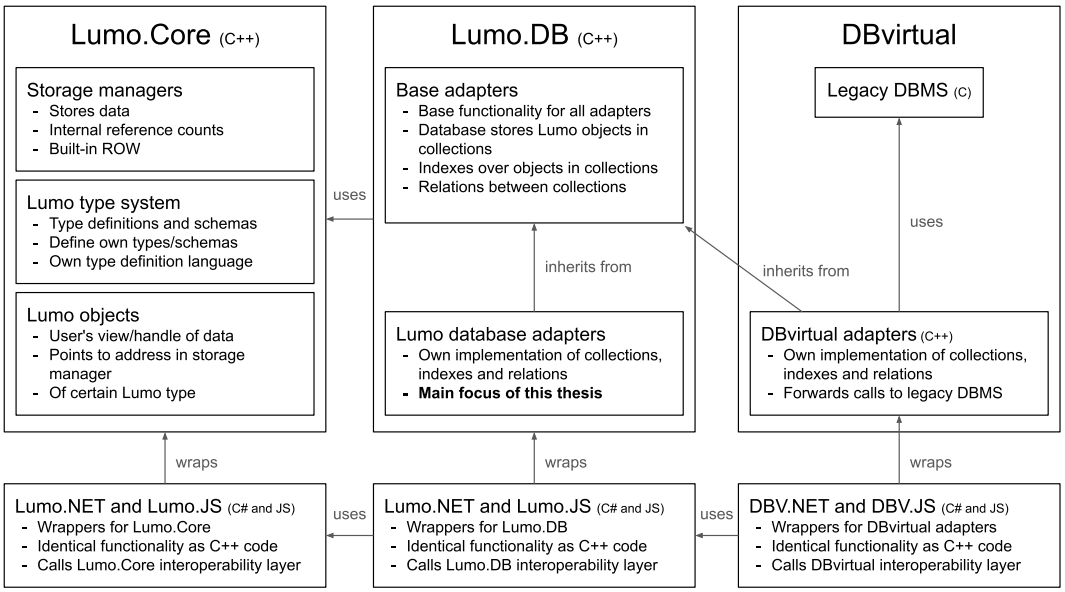

Figure 1: Short and simplified overview of the different Lumo components and how they relate to each other.

and replacement of individual units without touching the whole code base. The different units include a *type manager* (for defining, manipulating and storing *Lumo types*), a *storage manager* (for storing and retrieving data), *type renderers* (for defining how objects of a certain Lumo types are laid out for storage) and *Lumo objects* (for wrapping data for user consumption). This division of responsibilities also means that a unit can be implemented in different ways depending on the use case, as long as it adheres to the predefined interface. For example, one storage manager could be implemented to store data into memory while another could store data to disk, and a set of type renderers could be implemented differently to allow for more efficient storage of read-only data compared to modifiable data.

The Lumo type schema supports not only primitive types and structs containing these, but also sub-structs, strings, enums, fixed size arrays, dynamically sized lists and any nested combination of these. Optional types, types by reference, struct inheritance, polymorphic references and default values are supported as well. Lumo supports three ways of defining a data model, the first of which involves instantiating and defining Lumo types at run-time and then registering them to the type manager. The second approach is to write the data schema in Lumo's own data description language and give that as a string to the type manager, which then parses the schema and creates the Lumo types on its own. The final way is to define the data schema using Lumo objects of a certain built-in type, which can be manipulated just like any other Lumo object. Being able to define a Lumo data schema using Lumo objects is an integral part of the type system, because it simplifies the saving, loading and transmitting of the Lumo data model since the Lumo types can in that form be handled in the same way as the rest of the data.

The Lumo objects are what applications use to interact with the data in the storage

manager. In simple terms, a Lumo object is given an internal *Lumo address* pointing to some data within the storage manager, as well as a Lumo type and corresponding type renderer which it can use to actually interpret the data in the correct way. Lumo.Core supports deduplication of Lumo objects, as in two Lumo objects having the same storage manager addresses can be made to point to the same place if the data at both addresses is identical. There is no limit on how many Lumo objects can point to the same data, and internal reference counts make sure that data is not released until all Lumo objects pointing to it are thrown away. Lumo uses redirect-on-write (see Section 3.1.2 for an explanation of the technique) to make sure that modifications done by one Lumo object do not change the state of another Lumo object pointing to the same data. Lumo objects also come in two flavours, read-only and writable, making it easy to restrict users to not be able to modify data which is meant to be read-only. Lumo can also generate typed Lumo object wrapper classes on the programming language level which simplify the handling of Lumo objects. An example which reads the `id` field of a Lumo object using the normal dynamic API could be `obj.getValueByNameInt32("id")`, while a generated typed API would simplify that to `typed_obj.id`.

Lumo.Core stores Lumo objects as is, in a representation which is closely related to how the user sees them in the programming language, however complex the type of the Lumo objects may be. This means Lumo.Core could be described as an object-oriented database (OOD) since it adheres to some, but not all, common views of what an OOD should look like [4]. However, Lumo.Core lacks certain essential features such as data indexing and querying for it to be used as a DBMS in any traditionally meaningful way, but many of these shortcomings are rectified with Lumo.DB (see Section 2.3.3). On the other hand, Lumo.Core does carry similar features to well-known OODs such as GemStone/S [5] and ObjectDB [6]. Lumo.Core can also be described as an embedded database, just like the DBvirtual. This is because it is not a standalone application but rather a component which can be embedded into other applications.

2.3.2 Lumo.NET and Lumo.JS

The Lumo.Core component is written in C++ and implements an interoperability layer which exposes the API in a way that it can be used from other languages. This has made it possible to write a C# wrapper for the Lumo.Core component called Lumo.NET. A similar wrapper has been written as a JavaScript (JS) library (TypeScript, to be precise), for use in web applications, called Lumo.JS. For using the native Lumo.Core component from JavaScript in the web browser it is compiled to a WebAssembly module using the emscripten [7] toolchain. The two wrappers expose a nearly identical API to the native component and implement the same classes, but calls to the native side do most of the heavy-lifting. The .NET wrapper is almost on par with Lumo.Core in terms of performance, while Lumo.JS has a bit more overhead. Extra care has been taken to ensure that object lifetimes within Lumo are handled correctly even if an application mixes code from multiple languages.

2.3.3 Lumo.DB

The Lumo.Core component can in some cases be used on its own, for example if the lack of indexing and relations is of no concern, but in many use cases efficient data access like in a "real" DBMS is needed. This is where Lumo.DB comes into the picture, sitting on top of Lumo.Core, providing an interface for integrating Lumo.Core into other existing database solutions. Lumo.DB provides base adapter classes for a database, collections, indexes and relations, which are meant to be extended to create custom adapters for specific database back-ends. Instances of these custom adapters would then be used by the user to interact with the data in the database in various ways, for example inserting data into a collection or selecting data from a collection using an index. Lumo.DB also has wrappers written in both C# and JS in an identical manner as for Lumo.Core. The Lumo.DB interface is described in more detail in Section 5.1.

There currently exists a custom Lumo.DB adapter for DBvirtual, which is the link that enables it to take advantage of most of the new features that Lumo provides. All data is stored in the Lumo storage manager as Lumo objects, but references to the Lumo objects are inserted into DBvirtual tables at run-time. In a similar manner the index and relation adapters are basically implemented to just convert and forward the calls to DBvirtual.

2.4 Upcoming development: Lumo, phase 2

Phase two of the Lumo development consists of, among other things, turning Lumo into a fully fledged DBMS on its own without having to use any external integrations. This will be done by implementing a custom Lumo.DB adapter, called *Lumo database adapter*, which implements collections, indexes and relations on its own.

Another feature which will be part of the new Lumo database adapter is being able to store snapshots of the whole database. A purpose-built multiversion index will allow for efficient creation, removal, and usage of snapshots, while also keeping the snapshots as memory-efficient as possible. The snapshots will be designed to be read-only, whereas branches can be opened from any snapshot for doing modifications.

The work done towards this thesis will be part of the second Lumo development phase, and will mainly focus on the definition and implementation of a Git-like multiversion extension of the Lumo database adapter. This includes implementing snapshots, branches and the multiversion index, and benchmarking the usage and efficiency using real project data.

3 Snapshots and branches

3.1 Snapshots

In this thesis, as well as in general, the term *snapshot* simply refers to a certain database state which has been stored and can be viewed at a later time. Snapshots are often implemented to be static read-only views of the database, which is also the case in Lumo.

3.1.1 Use cases for snapshots

Snapshots can be useful in a variety of situations, and use cases can be divided into several different groups. Snapshots can either be used in a stand-alone fashion, where the existence of other snapshots is irrelevant, or in a context where several snapshots make up a kind of timeline or tree-structure. A timeline of snapshots is a sequential usage of snapshots where they are used in a linear fashion and every snapshot has a maximum of one parent and one child. The opposite would be a parallel usage, where multiple snapshots can represent different views of the same point in time. Snapshots can also be either temporary or long-lived depending on the use case. All of these kinds of use cases are of interest in this thesis.

Traditional use cases where snapshots are utilized in a stand-alone manner include reporting purposes, for example by allowing reports to be written based on a certain database state, or by attaching a snapshot to a bug report to give the developers an idea what kind of data produced a certain bug. These kinds of snapshots are also temporary in nature. Permanent snapshots are on the other hand used for instance when maintaining historical data on timeline. Many version control systems also allow storing historical data in a tree structure. Snapshots are also traditionally used for safeguarding reasons by creating snapshots as recovery points in the event of a user or administrative error.

Use cases important to this thesis are those which are relevant for the applications currently using DBvirtual (described in Section 2.2). More specifically, in Tekla Structures snapshots could be used in tracking the history of a TS model, or as a way of isolating users or processes accessing the same model. A use case for the latter would be to be able to create a temporary snapshot of a TS model which the application can use to render the model to the screen or process in some other way, for instance creating a 2d drawing of the model, while the user can continue manipulating the model as normal.

Some transactional databases use snapshots as a way to isolate transactions from each other using a mechanism called snapshot isolation (SI) [8]. When a transaction T is started using SI a snapshot is created for T to read from, and T can only read data from this snapshot. Writes done by T will be reflected in the state of the snapshot, but no concurrent transactions can see or modify what T sees. Transactions using SI can be aborted at commit if T modified data which was also modified by other transactions that were committed during the lifetime of T. It has been shown that committing concurrent transactions using SI can cause the resulting database state

to not satisfy certain integrity constraints, even though the constraints are satisfied when the transactions are run alone. An example would be a database containing two objects, where an integrity constraint is that the database cannot be empty, and two transactions are run concurrently which delete one object each. This has been dubbed the *write skew* anomaly. [9]

Snapshots used for SI are, in contrast to traditional snapshots, not read-only, since the writes done by the transaction modify the snapshot. In addition, using SI means that every commit produces a state which will later become a snapshot once a new transaction is created, while traditional snapshots usually are created only when demanded by the user. Traditional snapshots can also be long-lived while snapshots used for SI only live as long as the transaction is open. SI has been adopted by many popular data management systems, such as MySQL [10], PostgreSQL [11], MongoDB [12] and the distributed SQL database NuoDB [13].

3.1.2 Snapshot implementations

Snapshots could essentially be implemented by copying the entire database, but the implementation is usually done in a more storage-efficient way. In practice there are basically two ways to implement snapshots, with copy-on-write (COW) or redirect-on-write (ROW). In both implementations no data is copied at all when the snapshot is created; the snapshot only holds pointers to the data in the source database. Writing to the source database while having snapshots is where the two methods differ. COW will copy the original data to a new place for the snapshot to start pointing at and write the new data in the original place, while ROW will keep the original data and pointer from the snapshot as is and write the new data in a new place for the source database to start pointing to. [14–16]

Snapshots are supported by many industry-leading DBMSs, such as Microsoft SQL Server [17] and Oracle Database [18], which both implement snapshots using COW. Also many file systems support snapshots, for example zfs [19], btrfs [20] and WAFL [21], which all use ROW. There are also DBMSs that use ROW for snapshots, for example the distributed SQL database YugabyteDB [22]. PostgreSQL [11] and MariaDB [23] only support snapshots on the file system level, as in letting the file system do the heavy-lifting, while MongoDB [12] and MySQL [10] only support snapshots by copying the entire database content.

Looking specifically at other embedded databases, a few can be found that implement snapshots to some degree. For example, SQLite supports snapshots for the purpose of isolating the current writing transaction from read-only transactions [24]. LevelDB, written at Google, is another embedded database that supports snapshots [25]. However, both of these implementations are for short-term use only, and it is not expected for the user to hold on to the snapshot for longer periods of time.

Using the COW or ROW mechanisms when handling snapshots means that the initial cost of creating a snapshot is next to zero, since the snapshot only contains the changes made to the source since its creation. This means that the snapshots increase in size as more and more modifications are made to the source database. However, since the snapshots themselves do not contain all the actual data, only references to

it, they should not be confused with backups. Using ROW does fewer operations on write compared to COW, but will lead to the data in the source database becoming more fragmented over time. Oracle warns about using snapshots on a database which has a production-level workload [18], since snapshots decrease performance of the database due to the extra copy operations when writing data. However, this concern is mostly relevant for databases which write data directly to disk, especially so if the disk has a slow write speed, and do not affect in-memory databases to the same degree.

One goal of this thesis is to implement snapshots for Lumo in the Lumo database adapter, which works out to be a very simple task because the underlying object handling mechanism in Lumo.Core already implements ROW. Creating a snapshot of a database is as simple as taking a reference to each Lumo object that should be included and storing them inside the snapshot. Modifying any Lumo object in the source database will internally store the new object in a new place and let the snapshot keep referencing the original one. The Lumo object references in the snapshot will keep all used data in the storage manager from being released. In Lumo the snapshots will be considered read-only. However, one aspect that ROW does not help much with is that any indexes over objects in the database will still have to be copied entirely when a snapshot is created, leading to a lot of duplicate data between snapshots.

Another place where snapshots are proficiently used is in Git, a popular distributed version control system, where each commit can be thought of as a new non-temporary snapshot. A Git repository contains a few kinds of objects, the most relevant here being the blob, tree and commit objects. A blob contains the content of a committed file, and every new committed version of the file corresponds to a separate blob. Tree objects correspond to the folders in the repository and hold references to blobs and/or other trees. There is no limit on how many times a blob or tree can be referred to. Commit objects contain a reference to a single root tree object which holds all the actual content of the snapshot. Commits also hold a reference to a parent commit (except for the initial commit) and can therefore easily reference the same trees and blobs as the parent if those were not changed in the commit. When a new commit is created, which includes modifications of a file in the repository, Git leverages redirect-on-write. This means that when a new blob is created for the new version of the file, the blob for the old version remains untouched so that the older commits still can refer to it. Git also uses delta compression to pack the objects in the database as compact as possible, but compression of Lumo databases will not be explored in this thesis. [26, 27]

Dolt calls itself the world's first SQL database with Git-like version control for its data. In Dolt, all versions of all data, as well as all commits, are stored in data structures called Prolly Trees. The exact construction of a Prolly Tree is explained later in Section 4.2.1, but the relevant fact here is that it is a tree structure where the hash of the node content is used to reference the node in a key-value store. Each table in a Dolt database stores its data in its own Prolly tree, and the data root node is stored alongside the table schema in a separate table node. Table nodes are then hashed and inserted into a database node, which corresponds to a certain state of the database, much like a snapshot. Commits in Dolt work similarly as in Git, but point to database nodes instead of blobs and trees, and the use of Prolly Trees makes it very efficient to compare the content of two commits. [28]

3.1.3 API considerations

Lumo is not a traditional relational database where one would only interact with the data through queries and procedures written in a query language. Lumo is better described as an OOD where the data is represented in the form of objects of different types rather than be split into several tables. This means that the Lumo interface is by nature very closely related to the object-oriented programming languages themselves, like C++ and C# in this case, and designing the API for interacting with snapshots need to take this fact into consideration. Prominent OODs could be looked at to get an idea what a good and intuitive API looks like, but none of them seem to implement snapshots in a user-accessible way other than for backup purposes. Any SQL database ORM that would implement snapshots in a meaningful way have not been found either.

Multiple Lumo objects can point to the same data internally, and as previously mentioned, Lumo uses internal reference counts to keep track of which data is still in use and which can be released. Lumo object lifetimes are therefore crucial to get right, especially since Lumo objects residing in C# or JavaScript code should also keep the data alive. This needs to be reflected in the snapshot API as well with explicit methods to release the database connected to the snapshot or to throw away the entire snapshot.

Embedded databases have an even more intimate relation to the programming language, so existing snapshot APIs from those need to be taken into account as well. The C++ interface for SQLite uses the method `sqlite3_snapshot_get(*db, *snapshot)` to create a snapshot and `sqlite3_snapshot_open(snapshot)` to start a read transaction from it [24]. The respective methods using a LevelDB database in C++ are `db->GetSnapshot()` and `db->NewIterator(snapshot)` [25].

Git is primarily used through a command line interface (CLI). The most basic workflow is to use `git commit` to create snapshots and `git checkout <commit>` to view a snapshot (even though there is seldom a use case for checking out a commit rather than a branch) [27]. This is the way Lumo is intended to work too, except that the interface has to be defined as methods on the programming language classes instead. Storing every single change as a separate snapshot is not efficient, no matter the implementation, so the snapshots have to be created only when it makes sense, either automatically or preferably manually by the user for example via some kind of UI operation.

Dolt, offering a Git-like version control system for a database, provides many of the same Git commands through a CLI, and the implementations are made to be as similar to Git as possible. Since Dolt is an SQL database, the same commands are also available as SQL procedures, but the operations are the same regardless of which interface is used. [28]

Since Git (and Dolt) commits are considered permanent, they cannot be deleted without rewriting the whole history. In Lumo however, snapshots should be able to be used in a short-lived manner and be thrown away when not needed anymore. Therefore the new API needs some way to delete a snapshot completely, without messing up its parent or children.

3.2 Branches

In this thesis a *branch* refers to a writable version of a specific snapshot, or in other terms a pointer to a snapshot which one wants to modify. Modifications made to a snapshot through a branch are stored within the branch and do not change the snapshot in any way, and the branch state can at any point be stored as a new snapshot. Branches, as defined in this thesis, can only be used in conjunction with snapshots since they need a snapshot as a starting point. However, the concept of a branch only makes sense if snapshots are utilized in a parallel manner, as in if there can exist several snapshots describing the same point in time. If snapshots can have only one child, only the latest snapshot would be able to have a branch, which would defeat the purpose of branches. Another way of looking at branches is to think of them as long running transactions isolated from all other branches. All snapshots made from a branch belong to that branch, and if the branch is deleted before merged into another branch, all of its snapshots are deleted as well.

3.2.1 Use cases for branches

Branches come into the picture when one wants to be able to add modifications to a snapshot which otherwise is read-only, especially when there is a need to have more than one such modification path open simultaneously. One use case for branches is any multi-user application where users want to do modifications to data without having to care about changes made by others. The users would then be able to have their own branch where they can make and commit changes on their own, while also being able to switch back and forth to other branches to check out other versions of the data. Branches can then be merged together in the future. This kind of workflow has been used by software developers for a long time, Git being the most prominent solution nowadays, but adding support for this into Lumo would give the same type of control over any kind of data in general.

Using branches within Tekla Structures would for example allow users to work with the same shared TS model on separate branches, and then merge their changes to a common branch at certain intervals or after they have reached a specific milestone. Similarly, branches would also allow development of multiple alternative designs to a certain part of a model simultaneously in different branches, which can easily be compared later by switching between the branches. Branches would also benefit Tekla Model Sharing in a similar way, allowing users to work on completely separate branches, all served from the same online model.

3.2.2 Branch implementations

Branch objects in Git simply hold a reference to a commit (snapshot) and can be thought of as separate contexts inside the Git repository. A branch can be created starting from any commit, and doing additional commits while connected to a branch will cause the branch to always point to the latest commit in the chain. Branches can also be checked out with the `git commit <branch>` command, meaning the user can easily jump from one context to another, and it is the commit referenced by the branch

that acts as the view for that specific context. Git relies heavily on the branch objects to keep track of the commits in the repository to the degree that creating commits while not currently connected to a branch can lead to problems since nothing is referencing the commit. Deleting a branch will not actually delete all commits that were made through that branch, but if the commits are not referenced by something else, they will eventually be garbage collected and removed from the repository. [27]

Since Dolt tries to imitate the mechanisms of Git as closely as possible, the implementation of branches there is the same as in Git, at least on a conceptual level. Branches point to database nodes (commits), new commits can be created on a branch, and branches can be merged. [28]

PlanetScale is an SQL database that offers a branching workflow for applying updates to the database schema. A branch in this case is simply an isolated copy of a database, excluding the actual data, which can be used to apply and test schema changes. The updated schema on the branch can later be merged back to the original database. Note that the branching and merging only applies to the database schema, not the data itself, which makes this implementation fall outside the scope of this thesis. [29]

4 Multiversion databases and indexes

A multiversion database has the capability to store more than one version of the same data, or alternatively stated, it can keep track of how the data in the database changes over time. Updates to an item in the database will add a new version of this item, and deletions mark the item as deleted without removing the previous versions. An index over a set of items in a multiversion database contains all versions of the items, while index operations return only one version, i.e. the version that is visible in the current context. Most traditional indexes, however, are not designed to store more than one version, so if multiple versions of the data need to be accessed in a timely manner, specially designed multiversion indexes need to be used. The idea with multiversion indexes is to be able to access any past version as fast, or almost as fast, as the current version while minimizing memory usage. As is the case with any index, when building a multiversion index there is a trade-off between query time, update time and storage space.

4.1 Use cases for multiversion databases

Storing more than one version of data in a database is in many cases not necessary, but can be useful in certain situations. One widely used use case for data versioning is as a solution for accessing data concurrently, a technique referred to as multiversion concurrency control (MVCC) [30]. Using MVCC lets different processes use different versions of the same data for the purpose of not interfering with each other, and the different versions are usually only stored during the lifetime of the current transaction (see the definition of snapshot isolation in Section 3.1.1). If the versions are stored for longer periods of time, all old versions can be thought of as snapshots, which gives data versioning similar use cases as snapshots, described in Section 3.1.1. Queries in a multiversion database that search for all objects alive at a certain version will in this thesis therefore be called snapshot queries.

On the other hand, snapshots and branches do not have to be indexed to be useful, but using multiversion indexes will make storing and accessing snapshots more efficient regardless of use case. In Tekla Structures, multiversion indexes could be used to efficiently compare two versions of a TS model, or jump back and forth between two versions of a model in an instant. Another possible use case regarding TS would be to enhance Tekla Model Sharing to be able to serve clients having their own separate states of the joint model.

4.2 Multiversion index implementations

If a database contains several versions of the same data, the fundamental problem is how to keep track of which versions of which objects that go together. Creating and maintaining explicit links between objects is not practical when the number of objects and object versions grows. One approach is to introduce the notion of database versions (DBV), where each object in the database refers to a certain DBV. Modifications to any object, or by committing a transaction, would in this case create a new DBV

derived from the current version, creating a new node in a DBV tree. Since a new DBV only contains a few differences to the previous version, most objects could be shared between the two versions. Only the modified objects compared to the parent DBV need to be stored in the child DBV, meaning, if an object is not found in a DBV, it must be fetched from the parent DBV instead. [31]

4.2.1 Multiversion indexes using B-trees

B-trees are by far the most popular data structure used for indexes [32, 33]. Traditional B-trees are however not capable of indexing multiversioned data, but many B-tree-like multiversion indexes have been suggested.

One straightforward solution for adding multiversion support to any index, including B-trees, is modifying the records to also store a pointer to the previous version of the record. In practice, the different version of a record would then form a linked list, and in the case of B-trees the leaf nodes could point to the beginning of these linked lists of records instead of a single record. The current version of a record would be as fast to fetch, but versions further into the past would take longer to fetch. [34]

Jouini and Jomier [35] reviewed three different families of multiversion index structures derived from B-trees using the DBV approach, the first of which they call the B+V-tree. This index uses a modified B^+-tree [36] where each object is indexed by its key-version pair. When a leaf overflows in the B+V-tree the objects are split by the key, clustering different versions of the same object together if possible. The second reviewed family was a slightly modified Overlapping B+tree (OB+tree) [37], which creates new B^+-trees for each database version but lets consecutive B^+-trees share unchanged nodes. This index clusters objects by database version. The third family reviewed in the paper was a bi-dimensional index structure called Branched and Temporal Tree (BT-tree) [38]. The paper estimated and simulated both the query and storage costs of all three index structures. The conclusion was that the B+V-tree has the most efficient perfect match lookup, while the BT-tree has the most efficient snapshot queries. The B+V-tree was also deemed to have the smallest memory footprint of the three index structures which were tested. [35]

An asymptotically optimal multiversion B-tree (MVB-tree) was suggested by Becker *et al.* [39], where a DBV interval is stored together with every key. Each modifying operation (insertion or deletion) increases the version by one and stores it alongside the updated key, either as the start of the version interval for new keys or as the interval endpoint for deleted keys. Items in an MVB-tree can be queried with a key-version pair. The MVB-tree does not support modifications using transactions, but has later been extended to support them [40, 41].

Time-Split B-trees (TSB-tree) introduced by Lomet and Salzberg [42] stores records by key and transaction timestamp, and divides nodes into a current version and historical data. When a leaf node is full it can be split either by key, as in a normal B-tree, or by timestamp. A timestamp split in a TSB-tree is done by choosing a suitable timestamp to split at. Then all records valid only before that timestamp go into a historical node, while records only valid after the split time remain in the current node. Records valid both before and after the split time are duplicated into

both nodes. Temporal queries are therefore based on transaction time. There is no direct connection between successor and predecessor records since they can be spread randomly in both historical and current nodes. [43]

Using B-trees in a system that also uses COW mechanisms can be problematic due to a couple of reasons. Normally, if a B-tree leaf node is updated, it is just locked and modified. However, if COW is used, and the node is referenced to more than once, it will have to be copied before it can be modified. This will trigger the ancestor node to also be copied, which in turn will copy its ancestor, and so on. The single change to a leaf node will propagate up to the root node and will require locking of all nodes in between, which becomes an issue for concurrency. If B+trees are used, the problem becomes even more severe, since then there are references between each leaf node as well, which means that in the worst case, the whole tree has to be copied. Rodeh [15] did, however, show that B-trees can be modified to work well with COW by using top-down B-trees [44] rather than the traditional bottom-up B-trees and by using modified B-tree algorithms specifically written to respect COW.

Gottstein *et al.* [45] proposed the Multi-Version Index (MV-IDX) where the version information is stored directly in the index, rather than on the indexed tuple, meaning that decisions regarding the visibility of a data items can be answered in-memory without accessing the tuple. It was shown that MV-IDX works well in use cases where data updates are frequent and the DBMS stores a high amount of invisible tuple versions. It was not as effective in workloads with fewer update operations.

The Prolly Tree (Probabilistic B-tree), first introduced by the creators of Noms [46], and later extended on for Dolt [47], is a B-tree variation meant for version controlled databases. Dolt stores all its data in Prolly Trees, everything from the content of a table to the content of a commit, as already explained in Section 3.1.2. The number of values in each node is dynamic and determined probabilistically. The key values are laid out in a sorted order and the chunk boundaries are then determined by calculating a hash for each key value. The hash depends on the key value and the size of the current chunk, and if the hash is lower than a predefined value, a new chunk boundary is created. Each chunk content is then hashed and stored in a content address block store, and identical chunks are only stored in the store once. The next level of the Prolly Tree contains the highest key values from each child chunk, and their content addresses (the chunk hashes), upon which the same chunk boundary algorithm is applied. The root node is found once there is only one chunk on a level.

Prolly Tree insertions and deletions can, since the chunk boundaries are determined based on the key value and the chunk size, result in new chunk boundaries, but only with a small probability. For the same reason, updates to an existing object in the tree will never result in new chunk boundaries. A modification will, however, always change the content hash of a leaf chunk, which in turn will propagate changes up to the root node, resulting in a new table state which can be referenced by a commit. All new chunks will be stored in the content address store with their new content hash, but chunks which have not been modified do not have to be stored again. All old chunks will still remain in the content address store to preserve the history. Dolt has tuned their probabilistic boundary condition to give an average chunk size of 2000 bytes. The Prolly tree is history independent, i.e. the tree, including chunk sizes and

hashes, are the same no matter which order values are inserted, deleted or updated. This enables fast diff calculations and merges between two Prolly Trees, which is the reason Prolly Trees are used in Dolt. [28, 47]

4.2.2 Multiversion indexes using LSM-trees

The log-structured merge-tree (LSM-tree), first proposed by O'Neil *et al.* [48], is a data structure optimized for write-heavy applications, and consists of one memory component and one or more components on disk. The various components are (usually) tree-structures and are tuned to the characteristics of the storage media they are stored on. Modifications of an LSM-tree are first stored into the memory component, and once it has reached a certain size, parts (or all) of it are moved into a new disk storage component on the first level. The merging is done in batches to leverage the better write performance of hard drives when writing data chunks in a sequential manner rather than using random access writes as other common data structures rely on. Disk storage components on the same level can in turn be merged into a larger component for the next level. Searching for a specific key will first search the memory component and then the storage components starting from the first level, i.e. the latest modifications.

The LSM-tree is not designed for multiversion data, but extension propositions for such support have been made throughout the years. Muth *et al.* [49] proposed an extension to the LSM-tree called the log-structured history data access method (LHAM) which supports temporal queries on timestamped data. Like with the LSM-tree, incoming data is stored in a memory component and is later migrated to various disk components. Cao *et al.* [50] implemented the time-stamped LSM-tree (TS-LSM-tree) for their time-stamped event DBMS called Timon. The TS-LSM-tree keeps the most recent data in memory and has special handling for late data. A B-tree implementation called the Partitioned B-tree (PTS), based on the B^+-tree [36], with similar separation of data into in-memory and secondary storage components as the LSM-tree, was proposed by Riegger *et al.* [51]. The PTS was successfully implemented in a multiversion DBMS, as was shown to have better throughput relative to B^+-trees.

LSM-trees are used for indexing in many key-value storage systems, for example LevelDB [25] and RocksDB [52], but none of them support multiversion data or temporal queries. Zhao *et al.* [53] introduced an enhanced version of LevelDB called MVLevelDB, which supports temporal queries on multiversion data specifically in Internet of Things (IoT) applications using the proposed log-structured multiversion tree (LSMV-tree).

4.2.3 Trie-based data structures

B-tree is not the only data structure of interest when looking to implement a multiversion index, it just happens to be the most common structure for indexes in general. Another data structure worth investigating are *tries* [54, 55] (or prefix trees), which is another tree data structure. Searching a trie is done by partitioning the key and using the parts one after the other to follow the links between the trie nodes, starting from the root

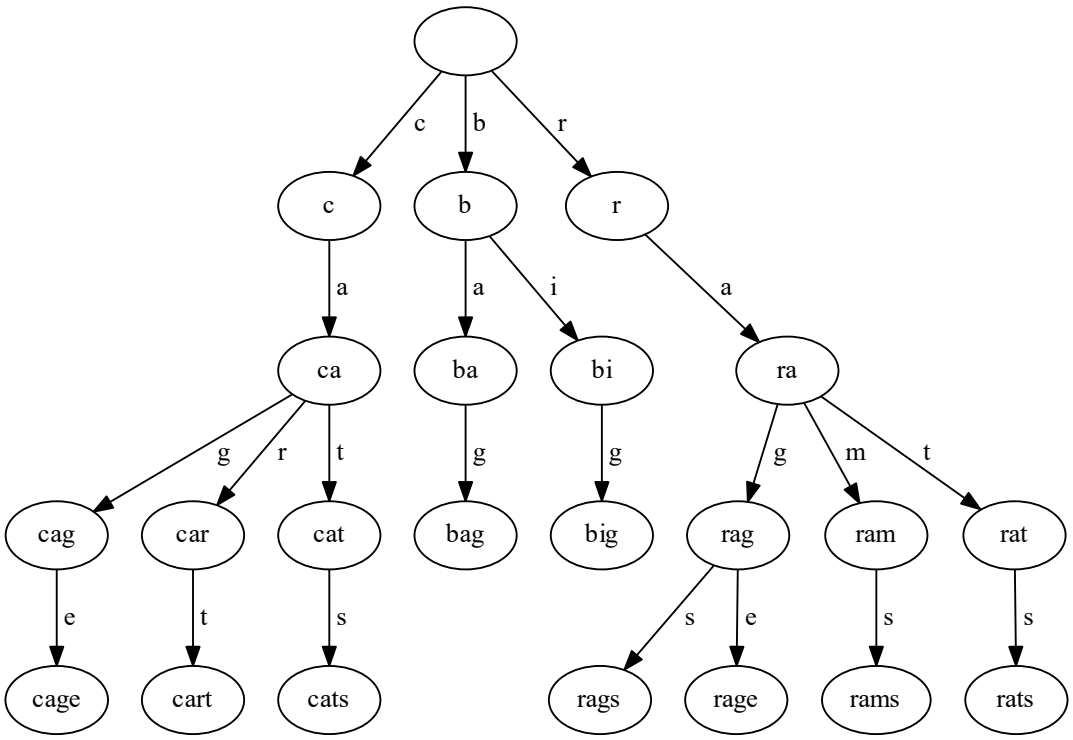

Figure 2: An example trie where each node corresponds to a key in the form of a string value. Each node stores only one character, but for the sake of making the diagram easier to follow, the whole word (so far) is fully written for each node instead. Using only small letters each node has 26 children, but empty nodes are not shown in this diagram. Finding the correct node using for example the key *cat* involves partitioning the key into characters and using them one by one to follow the correct path down the trie, starting from the root node. The nodes that would be traversed in this case are the root node, *c*-node, *ca*-node and finally the *cat*-node. Trying to use the key *dog* will result in a valid path not being found already at the root node, while using the key *carts* would result in a non-existing path not being found until reaching the `cart`-node.

node, until the whole key is used up. The nodes of a trie do not store their associated key as a whole, like in a binary search tree (BST) [56, 57], but only the last partition of it, which is needed to get there from its parent. Thus, it is the position of the node, or the path to it, that defines the key with which the node is associated. This in turn means that all children of a specific node are associated with keys which have the same prefix, and that the lookup complexity for tries is proportional to the size of the key. The *node cardinality*, i.e. the amount of child nodes per node, is dependent on the specific key partitioning used. In the case of for example string keys a key would be used up one character at a time to traverse the trie, hence the cardinality would be the amount of letters available in the character space used. An example of a simple trie which uses string keys is shown in Figure 2.

Storing a trie data structure can take up a lot of space unnecessarily since the tries

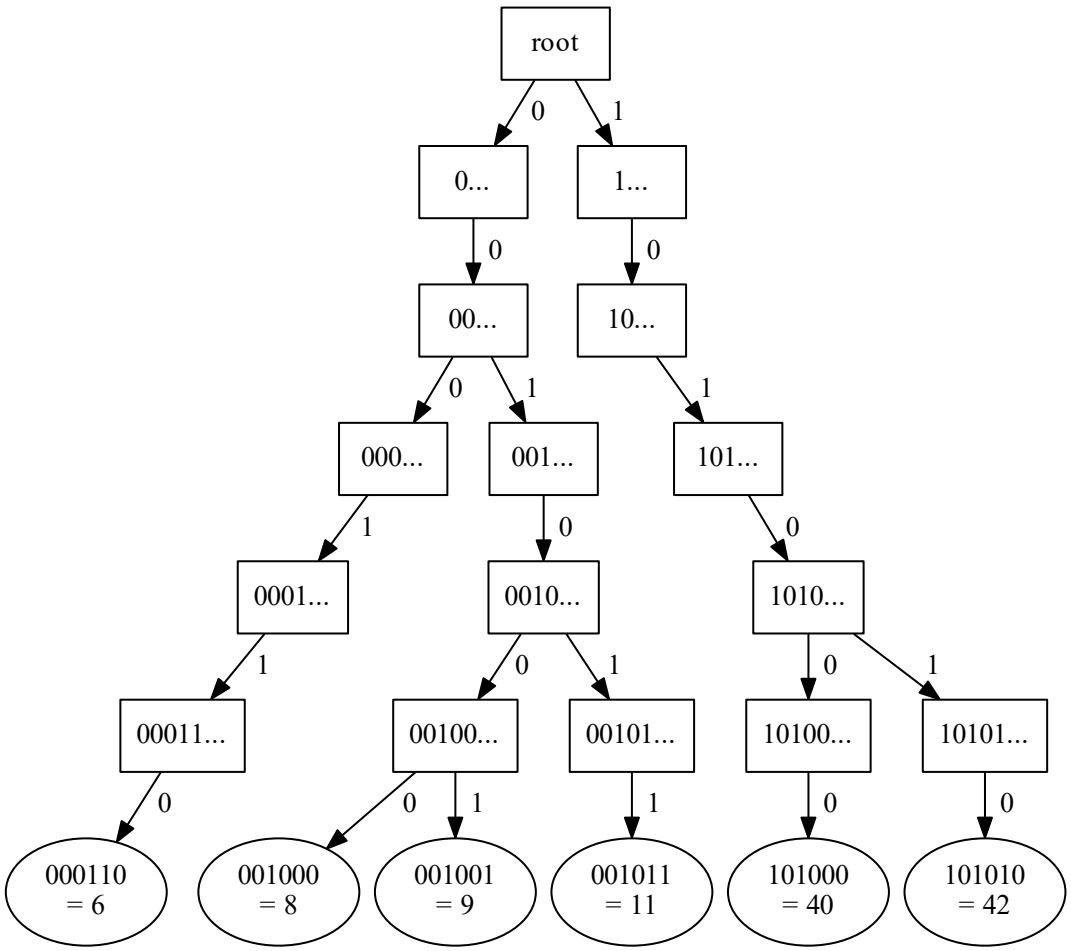

Figure 3: A binary trie storing six elements, here represented with ellipses. The keys used with the trie are all assumed to be six bits long, and are consumed one bit at a time starting from the highest bit (leftmost bit). Edge labels show which of the two paths (0 or 1) that specific edge represents, and the nodes, represented with rectangles, show the key prefix needed to get to that specific node.

can be very sparsely populated, meaning most of the space is taken up by empty nodes, which becomes more notable as the node cardinality increases. See for example the trie in Figure 2 where only three of 26 children on the root node are populated. Several variations on trie have been proposed which reduce the total memory footprint.

Bentley and Sedgewick [58] proposed the ternary search tree (TST) as a more memory-efficient trie alternative, where the nodes are arranged in a similar manner to a BST but have three children per node instead of only two. The original paper focused solely on using string keys with the TST, but any kind of key could be used. Being a trie, traversing a TST is done by partitioning the key and comparing the parts one by one to the node values. The three child nodes represent the three available paths depending on the result of the comparison, i.e. less than, equal or greater than. TST is shown to use less memory than normal tries, and is particularly appropriate when the keys are long strings.

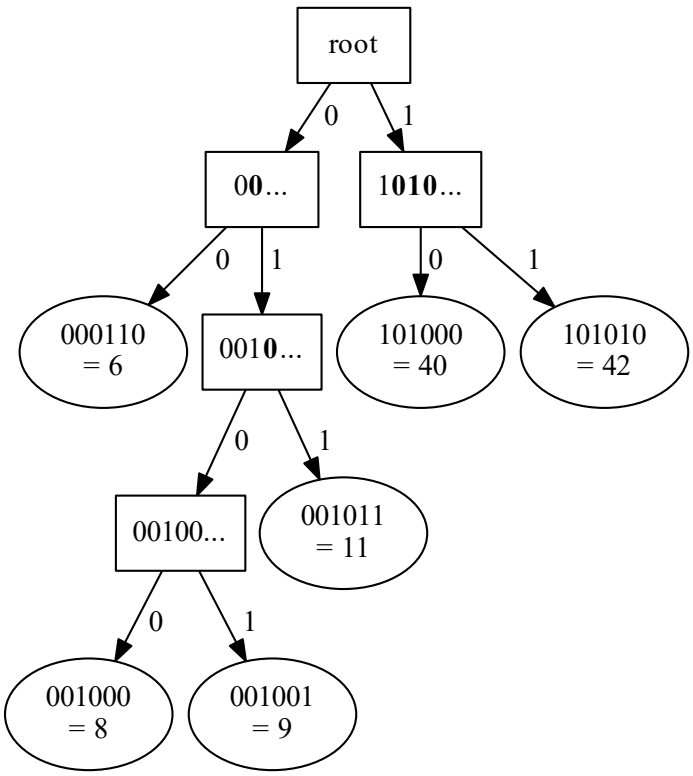

Figure 4: A path-compressed version of the binary trie seen in Figure 3, usually called a Patricia trie. Nodes with only one child have been removed, leaving only full nodes in the trie. Nodes show the key prefix needed to get to that specific node in normal font, and a possible additional prefix p common for all of its children in bold font.

One popular method of decreasing the memory usage of a trie is by applying path compression on it. Path compression of trie involves the removal of any internal node in the trie which has only one child, let us refer to such nodes as *one-child nodes* for future reference, and moving that child to the parent node instead. As a result of this, nodes also need to store the longest key prefix p which is common for all its children, and traversing from any node to one of its children $|p|$ parts of the key need to be skipped before doing the next comparison. A path compressed binary trie is usually referred to as a Patricia trie [59], where a binary trie has a node cardinality of two and the key therefore is partitioned into individual bits. Patricia tries do not have any internal nodes with empty children. Figure 3 shows an uncompressed binary trie and Figure 4 shows the same trie after path compression. [60]

Another compression method for tries is level compression, proposed by [61], and the resulting trie structure is called a level-compressed trie (LC-trie). The paper focuses solely on using the compression on binary tries, but comments that generalization to a trie with any cardinality is straightforward. The LC-trie is meant to be used with keys of equal length, as in there being no key which is a prefix of another key. This means that keys cannot be associated with internal nodes in the trie, only with separate leaf elements stored as children to nodes, and that a node child can either be another node

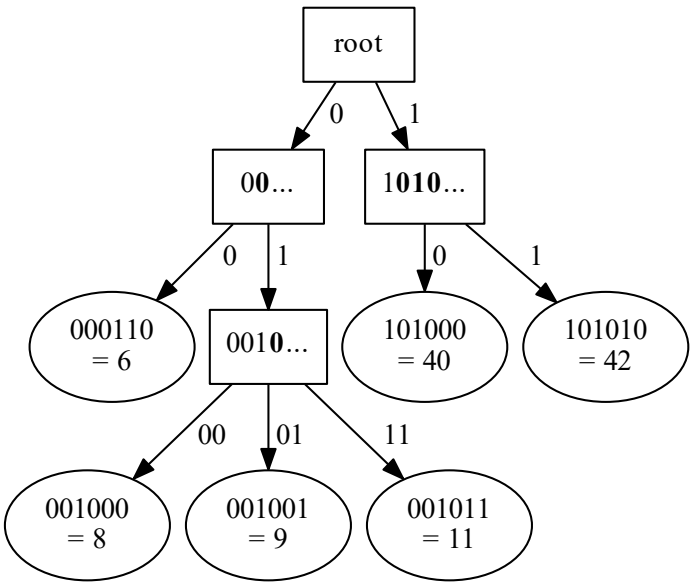

Figure 5: A level-compressed version of the Patricia trie seen in Figure 4. The trie is not a perfect LPC-trie since there are empty children and the root node only has a cardinality of two, but it still conveys how level compression is done in practice. Here, level compression has only been applied to nodes that would have had more than half of their children filled afterwards. Only node `0010` complied with this rule and had its cardinality doubled from 2^1 to 2^2. Note that node `00` could not be the target of level-compression because its child node `0010` included a common prefix p (in bold font), which would have resulted in two of the new children of node `00` occupying the `10`-path.

or a leaf element. Level compression of a trie involves doubling the cardinality of certain nodes and removing all of their direct child nodes, whose children in turn are moved up one level to become children of the node with increased capacity instead. Doubling the cardinality of a node will also make it consume one extra bit from the key during comparison. However, when level-compression is applied to a path-compressed trie, extra care needs to be taken to ensure that none of the removed child nodes are storing an extra prefix p, since that would make some of the paths from the parent node non-unique when taking the extra bit from the key into consideration. Which nodes get increased capacity is somewhat implementation-dependent, but for example a node with cardinality 2^k could have its cardinality doubled to $2^{(k+1)}$ if it would result in the node having more than half its $2^{(k+1)}$ children filled. A common use case for LC-tries is Internet Protocol (IP) address tables for internet routers [62, 63].

The level and path compressed trie (LPC-trie), introduced by Nilsson and Tikkanen [64], is a binary trie that combines, as the name suggests, both path compression and level compression. A perfect LPC-trie cannot have any nodes with empty children, in addition to the root node having a cardinality of more than two. The Linux kernel currently uses LPC-tries in its implementation of IP route tables [65]. Figure 5 shows level compression on top of the Patricia trie in Figure 4.

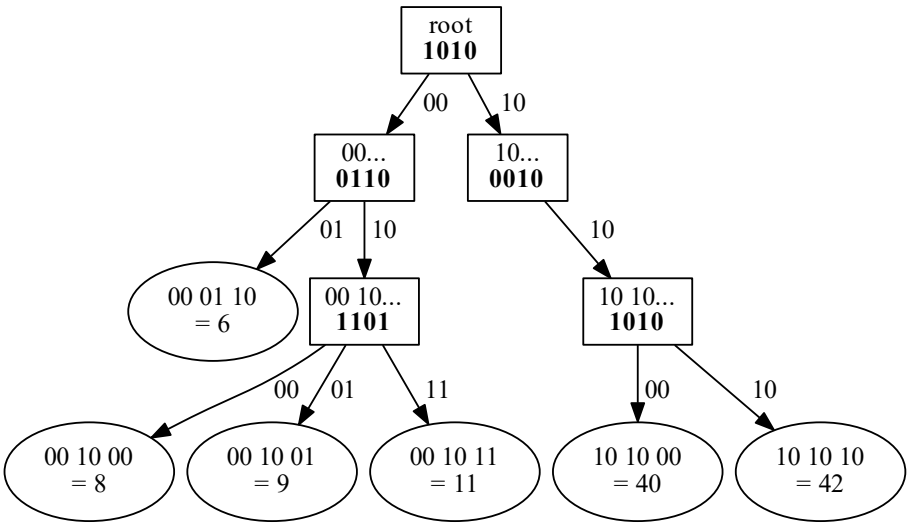

Figure 6: A HAMT which uses 6-bit long keys (hashes), and partitions them into 2-bit segments for traversal down the trie. The same elements as in the binary tree in Figure 3 have been inserted into the HAMT. Nodes include their bitmap in bold text, and the bitmaps are read from left to right, as in the leftmost bit represents the `00`-path.

4.2.4 Array Mapped Trie and derived data structures

Bagwell [66] proposed the Array Mapped Trie (AMT) as another solution to the trie sparsity problem by reducing the cost of empty node children to only one bit. In the AMT, all nodes have a dynamically sized array of children where only the non-empty children are stored in order. Each node also has a bitmap, where each bit represents whether or not the corresponding child exists in the array or not, which means the bitmap must have at least the same amount of bits as the cardinality of the node. The bitmap also indirectly gives the index of a certain existing child in the array, since counting the amount of bits set below its own bit in the bitmap gives the index in the array. The bitmap is, however, excluded for space-saving reasons if the amount of children is less than some value n (the value $n = 5$ was used for nodes with a cardinality of 16), and the children are instead searched sequentially. According to Bagwell [66] AMT performs 3–4 times faster while using 60 % less memory than TST, and is also faster than LPC-tries. Modifying the diagram in Figure 2 to use an AMT would remove all empty child nodes, but instead each node would need to store an additional 26-bit large bitmap (one bit per possible character).

Bagwell [67] later proposed the Hash Array Mapped Trie (HAMT) which uses an AMT as the underlying data structure. The difference to the AMT is that in a HAMT the key is first hashed to a fixed-sized hash, for example a 32-bit hash, before it is partitioned and used to descend down the trie. HAMT also uses bitmaps for all nodes, not needing a minimum amount of children on a node for it to get a bitmap. The only exception is that the root node does not have a bitmap, and stores its children in a statically sized list instead. See Figure 6 for an example of a HAMT which uses a 6-bit hash partitioned into 2-bit segments, giving the nodes a cardinality of four.

Searching in a HAMT is done by hashing the key and using k bits at a time to traverse down the trie. When the search encounters a node, the k next bits of the hash are used as an integer to index into the bitmap of the node. If the bit is not set, the key does not exist in the HAMT, while if it is set the search continues. The number of set bits below the indexed bit are counted and the result is used to index into the child array of the node. The child is either a leaf element, at which point the search terminates by comparing the key of the element to the search key, or another node, after which the same process repeats but with the next k bits of the hash. The exception to this process is in the very beginning, where the child array of the root node is indexed directly using the first k bits of the hash, since the root node does not have a bitmap. [67]

Inserting an element into the HAMT is done using the same steps as when searching for it. If an empty spot is found as a result of the search (a zero is encountered in a bitmap), the element is inserted in the child array of that node. This involves allocating a new array with one extra space, copying the old array to it, freeing the old array, inserting the new child in the correct place, and setting the corresponding bit in the bitmap of the node. Unless the node where the empty spot was found is the root node, then it is as simple as just inserting the element at the correct index. If another value is found in the search however, the existing leaf element is replaced with a new node instead, in which the existing element is inserted by calculating the hash and using the next k bits of it. The same k bits of the new element are then used for insertion into the new node, and if there still is a collision the same steps are followed again to insert another node. This ensures that there is no node in the trie with an element as the only child. In the worst case there is still a collision after all bits of the hashes have been used up, at which point another hash must be used to differentiate the two keys. [67]

Removing an element from a HAMT is done by first finding the correct element in the trie. If the node it is on has more than two children, the element is simply removed. This involves allocating a smaller child array for the node and releasing the old, and unsetting the respective bit in the bitmap. If the number of children is two, and the remaining child after deletion is another leaf element, the remaining child is moved up one level to completely replace the node in its parent. Nodes are never moved up the trie this way because a node is always associated to a certain level of the trie, i.e. to a specific partition of the hash, and moving them vertically in the trie would ruin that. From this follows that there can exist one-child nodes in the trie, but the lone child must be another node. [67]

A critical step of any action using the AMT (and HAMT) is the counting of set bits in the bitmaps. This must be done as efficiently as possible to ensure good performance. Modern CPU architectures have a CTPOP (Count Population) instruction which does exactly this, and it is therefore important to investigate how to force this behaviour through the programming language used for implementing the AMT. [66, 67]

A persistent data structure is a data structure that retains all previous versions of itself when modified [68], a property which could prove very useful for implementing an index in a multiversion database where all versions of the data need be stored anyway. HAMTs can be made persistent and such adaptations of HAMTs can in fact be found as the standard library implementation of hash trees in some programming languages, for example the Java Virtual Machine (JVM) languages Clojure [69] and

Scala [70]. The first proposed persistent HAMT is called Ctrie [71, 72] (or concurrent hash-trie), the implementation of which is also both lock-free and thread-safe, and is the implementation used in Clojure. Prokopec [73] proposed Cache-Tries as an enhanced version of Ctries that leverages caches better to achieve more efficient operations. The Scala implementation uses another implementation called Compressed Hash-Array Mapped Prefix-tree (CHAMP) [74, 75], which is based on a modified HAMT implementation optimized specifically for the JVM.

4.2.5 Support for duplicate keys in multiversion indexes

Many use cases benefit from enforcing uniqueness among the indexed values, but some require storing and indexing values with identical keys. One common use case for non-unique indexes is when indexing over a foreign key, assuming the relation is many-to-one or many-to-many. If the keys are compared after hashing, then potential hash collisions also need to be taken into account when dealing with duplicate keys.

B-trees can, depending on the specific implementation, store duplicate keys. This can be done by for example storing duplicates in a linked list, or by maintaining a counter associated with the key in question [76]. However, none of the B-tree derived multiversion indexes mentioned in Section 4.2.1 seem to support duplicate keys in any meaningful way.

In the case of tries, the key is partitioned and used up one part at a time until an existing value or empty spot is found in the trie. During insert, if an existing value is found and the whole key has not already been used up, the next part of the key is used to differentiate the values further. If a collision occurs even after using the whole key the inserted value is assumed to be identical to an existing one, resulting in an error, since tries do not inherently support inserting duplicate values.

For HAMTs, where the key is hashed before partitioning, different keys can produce the same hash, which means that hash collisions also need to be considered when talking about duplicate keys. The proposed solution for hash collisions is to discern the values further by using a second hash algorithm, assuming the values are actually unique [66, 67]. The enhanced HAMT implementations of Ctrie [72] and Cache-Tries [73] use special list nodes to store values with the same hash, so both duplicate objects and actual hash collisions are resolved. The list nodes basically function as persistent linked lists. In CHAMP [75] values with the same hash are similarly stored in a vector. In all tree implementations the true key values are compared if a list node is found during a search, and after removing an element from a list node the node will be deleted if its size is less than two.

5 Software implementations

This section describes the software implementations done for this thesis, as well as already existing implementations related to the thesis work. Section 5.1 describes the already existing interfaces for *Lumo databases*, which have been defined and implemented in a combined effort by my team before work on this thesis started. Section 5.2 on the other hand, describes the newly implemented interfaces for integrating multiversion into Lumo, which is work that has been done toward this thesis in particular. Section 5.3 goes into detail how special metadata objects can be leveraged when saving Lumo data in general, and how the metadata for multiversion data is defined. Lastly Section 5.4 discusses a few criteria regarding the multiversion index implemented in Section 5.5.

5.1 Overview of Lumo database adapters and Lumo schema

As explained in Section 2.3.3, using Lumo as a DBMS is accomplished by inserting and accessing data through adapters, which work as links between Lumo and the implementations of the DBMS back-ends. Lumo objects are inserted into collections, but the adapter in question determines exactly how the collection manages the objects. The same applies for the index and relation adapters, exactly how these function under the hood with regards to the Lumo objects is adapter-dependent. The following sections describe simplified versions of the interfaces which the different adapters have to adhere to, including a list of properties and methods for each interface using TypeScript syntax.

5.1.1 `ILumoDatabase`

This is the top-level adapter which holds on to all collections and relations defined by the user. The interface for the database adapters can be found in Code block 1.

5.1.2 `ILumoCollection`

Collections, identified by a name, hold Lumo objects of a certain Lumo type (or types inherited from it), and have methods for inserting Lumo objects and fetching all existing objects. Collections can also hold various indexes which index all objects in the collection. It is assumed that each collection has at least one index, which means that the responsibility of actually tracking the Lumo objects in the collection can be passed completely to the indexes. Supported operations on a collection are insertion of an object, which also will update each of the indexes, as well as fetching of all objects. See Code block 1 for the corresponding interface.

5.1.3 `ILumoIndex`

A *Lumo index*, the interface for which can be seen in Code block 1, allows for quick access to all objects in a collection and uses a specific column in the collection Lumo

Type as its key column. The key column can be any column, even a column on any sub-object. The key column is given as an array of numbers, where the first number corresponds to a column on the top-level object. If the aforementioned column is a sub-object, the second number in the key column path corresponds to a column on that object and so on. Having multiple key columns is also possible, so the key column

Code block 1: Simplified interface for Lumo databases, collections and indexes

```
interface ILumoDatabase {
    collections: ILumoCollection[];
    relations: ILumoRelation[];
}

interface ILumoCollection {
    name: string;
    type: ILumoType;
    indexes: ILumoIndex[];

    // Insert a Lumo object into the Collection
    insertObject: (obj: LumoObject) => boolean;

    // Fetch all Lumo objects in the Collection
    selectAll: () => LumoObject[];
}

interface ILumoIndex {
    name: string;
    paths: number[][];
    unique: boolean;
    ordered: boolean;

    // Fetch one Lumo object from the Index
    selectObject: (obj: LumoObject) => LumoObject;

    // Fetch all Lumo objects within a certain range
    selectRange: (a: LumoObject, b: LumoObject) => LumoObject[];

    // Update one Lumo object in the Index
    updateObject: (obj: LumoObject) => boolean;

    // Delete one Lumo object from the Index
    deleteObject: (obj: LumoObject) => boolean;
}
```

path is actually an array of arrays of numbers.

Using an index it is possible to fetch a Lumo object by providing a Lumo object with the key column(s) set to the appropriate value(s). Updating and deleting objects from the collection using the index is done by providing Lumo objects in a similar manner.

Indexes are either unique or non-unique, referring to if each object in the index must have a unique set of key column values or not. Indexes can also be ordered, which will allow for using range queries to fetch several Lumo objects with the key value(s) within a certain range, assuming that the key column value(s) can be ordered to begin with. The range select is also used to fetch several objects from a non-unique index, since the normal select will only fetch the first match.

The indexes will store references to its Lumo objects in some sort of data structure, but which structure that is and how it is searched given a set of key column values is implementation-dependent. This data structure is the most important piece of the puzzle when making an efficient multiversion database implementation.

5.1.4 `ILumoRelation`

In Lumo.DB, a relation connects, or relates, objects in one collection to objects in another collection. There is no separate table or collection needed for storing the relation, but rather a foreign key (one or more columns, same as for the `ILumoIndex`) to the related object is always stored directly as part of the main object. Since a relation also can be traversed in reverse (if allowed by the definition), from the related object back to the main object, the relation actually comprises two relation *endpoints*. Each endpoint is linked to an index on one of the related collections, and the indexes are

Code block 2: Simplified interface for Lumo relations and relation endpoints

```
interface ILumoRelation {
    name: string;
    from: ILumoRelationEndpoint;
    to: ILumoRelationEndpoint;
}

interface ILumoRelationEndpoint {
    name?: string;
    paths: number[][];
    collection: ILumoCollection;
    index: ILumoIndex;
    otherEndpoint: ILumoRelationEndpoint;

    // Fetch Lumo objects from the other endpoint's Index
    execute: (obj: LumoObject) => LumoObject[];
}
```

used to do the actual lookups when the relation is executed. Both endpoint indexes must refer to an identical set of key columns, so that the foreign key at one endpoint can be used to fetch the related Lumo object from the index of the other endpoint. If the index of the other endpoint is non-unique, the relation is a one-to-many relation. The interfaces for relations and relation endpoints can be found in Code block 2.

5.1.5 Lumo schema for defining custom types

Lumo schema is a description language for describing Lumo types in a simple and user-friendly way. The schema is written in a separate file or directly as a string in the programming language, and can be parsed by Lumo to generate the specified Lumo types and make them available in the application.

Code block 3 shows two Lumo types described using Lumo schema. The two Lumo types are meant to describe authors with a name and a unique id, and books having a name and a specific author. The author of a book is set by storing the id of the author object directly into the book object. The Lumo type system cannot, and does not, restrict or validate the actual data in Lumo objects as long as the data conforms to the type descriptions. For example, nothing stops the user from creating two author objects with the same id, or setting the `authorId` field of a book to be a non-existent author id. These are restrictions laid upon the Lumo objects by the Lumo.DB database once they are used in that context.

The `struct` keyword signifies that the described Lumo type is a struct type and the first parameter after the keyword is the name of the struct type in question. If the struct type is inherited from another struct, the base struct type name is given as a second parameter, but this feature is not used in the example.

The following indented lines (`<name> <type_name>`) are named columns of the struct. Type names which start with `::` signifies that the type in question is a built-in type and can be used out of the box. These include several different integer types (various sizes of signed and unsigned integers, for instance `::int16` and `::uint32`), two float types (`::float` and `::double`), a boolean type (`::boolean`), a string type (`::string`), and a type which represents an internal Lumo address within the storage manager (`::address`).

Code block 3: (Example) Defining custom Lumo types using Lumo schema

```
struct Author
  id   ::uint64
  name &::string

struct Book
  id       ::uint64
  name     &::string
  pages    ::uint16
  authorId ::uint64
```

The column type names can have one or more prefixes which alter their behaviour in some way, and prefixing a type name will in practice create a new separate type derived from the named type. Column data is normally stored in-place within the parent object, but if the &-prefix is used, the column data is instead stored by reference, meaning that an internal storage manager address pointing to where the column data can be found will be stored in-place in the parent object instead. Storing something by reference is useful since the same column data can then be referenced by many parent objects simultaneously, leading to less copies of the data. A column can be made into an optional column using the ?-prefix, allowing the user to know if the column has been set or not. The last supported prefixes are `[`ϕ`]` and `[]`, which represent arrays of fixed size ϕ and dynamically sized lists of the named type respectively. String and dynamically sized lists are the only types which must always be stored by reference, since their sizes are by nature not fixed.

5.1.6 Lumo schema for defining databases

The Lumo schema is extended to also support describing Lumo database structures in a similar manner, which can be parsed by Lumo.DB to generate a Lumo database, including its collections, indexes and relations. The Lumo schema in Code block 4 sets up a Lumo.DB database using the example Lumo types defined in Code block 3.

Collections are described using the `collection` keyword, where the collection name, and the Lumo type that can be stored in the collection, are given as parameters. Indexes in the collection are described using the `index` keyword, where the name of the index is given as the first parameter. The second parameter is either `unique` or `nonunique`, and the third `ordered` or `unordered`. Lastly, the `path=` parameter is the path to the key column inside the Lumo type of the collection, and can be a comma-separated list of column names if the key column is not on the top-level object.

Relations between collections are described using the `relation` keyword, followed by the `to` and `from` endpoint descriptions. The endpoints have the name of the collection it refers to as the first parameter. The `name=` parameter is used to later refer to the endpoint, and is required for the `from`-endpoint but optional for the `to`-endpoint.

Code block 4: (Example) Defining a database using Lumo schema

```
collection A Author
  index AuthorIdIndex unique unordered path=id

collection B Book
  index BookIdIndex unique unordered path=id
  index BookAuthorIdIndex nonunique unordered path=authorId

relation BookAuthorRelation
  from B name=author path=authorId
  to   A name=books  path=id
```

Defining `name=` parameters for both endpoints means that the relation is bidirectional, otherwise it can only be used in the forward direction. The `path=` parameter determines which key column to use when the relation is executed, the caveat being that there has to be a suitable index defined on the collection for that specific key column path. An index name could be directly given using an `index=` parameter instead of the `path=` parameter. The uniqueness of the indexes connected to the endpoints determine if the relation is a one-to-one, one-to-many, many-to-one, or many-to-many relation. For instance, a many-to-many relation would use non-unique indexes for both its endpoints.

As an example, take the `BookAuthorRelation` relation defined in Code block 4. To get the author of a book, the `Book.author` endpoint is accessed, which uses the id in `Book.authorId` to look up the correct author from the `AuthorIdIndex` index. Using the relation in reverse yields that `Author.books` uses `Author.id` to find the related books using the `BookAuthorIdIndex` index, which is a non-unique index, meaning the result can be more than one book. Simply put, the `BookAuthorRelation` is a one-to-many relation in the forward direction and acts as a many-to-one relation in reverse.

5.2 Introduction of Lumo model adapters

The introduction of snapshots and branches into Lumo will change how users interact with data stored in Lumo, since all snapshots and branches should have their own database, acting as separate views into the storage manager. How uses interact with the Lumo databases themselves should, however, not change at all. The sharing of Lumo objects between many databases is no problem, since Lumo internally leverages redirect-on-write to keep one view of the data from changing the state of another view. Additional views into the same data will require keeping track of which Lumo objects are visible to the new view, but does not require copying of the actual data unless changes are made to it. Even if the resource penalty for additional databases is not that high, care should still be taken to always have as few databases, snapshots and branches instantiated simultaneously as possible.

5.2.1 `ILumoModel`

The Lumo database is no longer the top-level object, since there needs to be something on top which manages the snapshots, branches and the separate views into the data, and the Lumo database is instead placed inside each snapshot and branch. This is where the new *Lumo model* comes into the picture, which is tasked to be responsible for creating, deleting, and keeping track of snapshots and branches. The interface for Lumo models is defined in Code block 5. One implementation detail that is not visible in the interface itself, is that the model should be able to selectively instantiate object representations of the snapshots and branches. Instantiating every snapshot in a large model at start-up would use up many resources unnecessarily. Enough information (see Section 5.3) need to be stored internally about the snapshots and branches to avoid having to instantiate actual object representations for them until actually requested by the user.

Code block 5: Simplified interface for Lumo models

```
interface ILumoModel {
    root: ILumoSnapshot;
    snapshots: ILumoSnapshot[];
    branches: ILumoBranch[];
    tags: string[];

    // Get a certain Snapshot or Branch
    getSnapshotById: (id: number) => ILumoSnapshot;
    getSnapshotByTag: (tag: string) => ILumoSnapshot;
    getBranchByName: (name: string) => ILumoBranch;

    // Delete a certain Snapshot or Branch
    deleteSnapshot: (snapshot: ILumoSnapshot) => void;
    deleteBranch: (branch: ILumoBranch) => void;

    // Save the entire Model state as a Lumo metadata object
    save: () => ModelMetadata;
}
```

5.2.2 `ILumoSnapshot`

Snapshots (see interface in Code block 6) have one parent snapshot and zero or more snapshot children, forming a tree structure just like Git commits. The root snapshot of the model is the exception to this rule, since it does not have a parent. In the future snapshots will also be able to have more than one parent if it is the result of a merge between two other snapshots, but merging is not yet implemented and not a part of this thesis. Snapshots have a unique identifier in the form of an integer, for the time being, but that might be changed to some other form of identifier in the future, for example a globally unique identifier (GUID).

As previously explained, a snapshot references a database that has a specific view into the data. This database is read-only and cannot be used to modify any data. Furthermore, the database on a snapshot must be explicitly instantiated. This is because instantiating a database, including all Lumo objects and indexes, can be rather resource-intensive and should be avoided if possible. An instantiated database can also be released manually when not needed anymore. Another benefit of having to explicitly instantiate and interact with a Lumo database is that that way the database API can be implemented independently from whether or not the Lumo multiversion features are used.

A branch can be started from any snapshot, enabling the user to add modifications to the data referred to by snapshots without changing the data visible to them. Many branches can point to the same snapshot. Additionally, snapshots can be tagged with a string tag so they can easily be found later, which is another Git-like feature.

Unlike Git commits, Lumo snapshots can in certain circumstances be deleted. The root snapshot and snapshots with more than one *descendant* cannot be deleted, where descendants includes both snapshot children and branches pointing directly to this snapshot, but any other snapshot can be removed from the model. Deleting a snapshot from the middle of a tree is thereby possible, and doing so will make its descendant (maximum of one snapshot or branch per the aforementioned rule) start pointing to its parent instead.

Temporary snapshots, snapshots that are created and deleted soon thereafter, have many use cases, and are the reason deletion of snapshots is supported at all. This assumes the deletion of a snapshot is efficient and does not require much clean-up. Temporary snapshots could be used, for example, to capture the state of the model to be processed elsewhere, for example to render the database content to a screen, while the user is still able to continue modifying his own version of the model in parallel. When the rendering is done, the temporary snapshot can be thrown away and the next frame can start to be rendered based on a new snapshot.

The goal is to be able to have tens of thousands, if not hundreds of thousands, of snapshots in a single model, so snapshots need to be as light and efficient as possible. Even if all snapshots are not instantiated into memory simultaneously, the information about which Lumo objects a snapshot is able to view still need to be stored somewhere. Decreasing the total model size therefore involves sharing as much information between related snapshots as possible, decreasing the amount of duplicate information stored.

Code block 6: Simplified interface for Lumo snapshots

```
class ILumoSnapshot {
    id: number;
    parent: ILumoSnapshot;
    children: ILumoSnapshot[];
    database?: ILumoDatabase_ReadOnly;
    tags: string[];

    // Instantiate/get or release the read-only Database
    getOrInstantiateDatabase: () => ILumoDatabase_ReadOnly;
    releaseDatabase: () => void;

    // Start a new Branch from this Snapshot
    startBranch: (name: string) => ILumoBranch;

    // Add or remove a tag for this Snapshot
    addTag: (tag: string) => void;
    removeTag: (tag: string) => void;
}
```

Code block 7: Simplified interface for Lumo branches

```
class ILumoBranch {
    name: string;
    snapshot: ILumoSnapshot;
    database?: ILumoDatabase;

    // Instantiate/get or release the Database for this Branch
    getOrInstantiateDatabase: () => ILumoDatabase;
    releaseDatabase: () => void;

    // Store the current Database state as a new Snapshot
    createSnapshot: () => ILumoSnapshot;

    // Throw away all modifications since the last Snapshot
    revertToSnapshot: () => void;
}
```

5.2.3 `ILumoBranch`

A branch (see interface in Code block 7), identified by its name, contains a reference to a certain snapshot, and can be used to add modifications to the data starting from the snapshot's view of the data. The branch will at first have the exact same view as the snapshot it references, but it will gradually diverge as changes are made. Modifications done within the branch do not change the snapshot's view of the data. Branches must, however, have their database explicitly instantiated, for the same reason as with snapshots.

After modifying the data using a branch, the modified view of the data can be captured into a new snapshot. The branch will then automatically step forward and start referencing the new snapshot instead. Modifications can also be thrown away completely and the branch reset to the previous snapshot.

Branches can be deleted from the model the same way as snapshots can. Branches keep their parent snapshots alive in the sense that deleting a branch from the model will also delete every snapshot up the tree until a snapshot with a second descendant is encountered.

5.3 Saving and loading a Lumo model

Lumo does not store anything to disk by default, but instead keeps all data in-memory. However, saving the data to disk is still a vital feature for any in-memory database and loading data even more so, which is why this is possible in Lumo as well. The save and load functionality and all its intrigues described in Section 5.3.1 were implemented in advance outside of this thesis, but the metadata discussed in Section 5.3.2 was designed and implemented specifically for this thesis.

5.3.1 The Lumo save functionality

With Lumo it is possible to save the storage manager content as a binary representation to, for instance, a file, and then load it again in a new application to return to the same state. The storage manager does not, however, store any information about what types of Lumo objects it contains, it only knows the binary representations of the objects, because storing type information alongside each data point would become unfeasible very fast. This means that when loading some unknown Lumo data it would be impossible to know how to decode it without a bit of extra context.

To combat this problem it is possible to, at save time, provide a set of extra Lumo objects which, unlike the rest of the objects, are saved with retained information about their types. These extra Lumo objects are called *metadata* objects because they are meant to be used to give extra context regarding the rest of the saved data. The extended information, which is saved for a metadata object, is its internal address and the name of its Lumo type, so that it can be found and parsed when the Lumo data is loaded.

However, just because the name of the Lumo type is stored does not mean that the system that loads the data knows what that type looks like, because to be able to use a Lumo object of a certain Lumo type, that exact Lumo type has to be registered to the system. Since all Lumo data need to be loadable and usable without knowing the schema of the data in advance this information needs to be included in the save as well. This is accomplished by turning all Lumo types used within the data into Lumo objects of a built-in schema and storing them into the storage manager. These special *Lumo type objects* are then provided as metadata for the save, and since their Lumo types are not user defined, but built-in, they can be restored and used to register the Lumo type they represent when loading the data.

5.3.2 Model metadata

The ability to store information in metadata objects also comes in handy when implementing save and load for an entire Lumo model, which includes storing the database state for each snapshot and branch, as well their parent/child relations. The code in Code block 8 shows the Lumo schema that defines the metadata used for this purpose. The Lumo schema syntax is described in Section 5.1.5.

The `DatabaseMetadata` Lumo type represents how a database is set up, i.e. it describes the collections, indexes, and relations of the database. Such a metadata object is what is used to create a Lumo database, which is why every snapshot and branch need to have this information at hand. The metadata also contains information about not only the database structure, but also the content of the database. The content is defined in the `IndexMetadata` object in one of two ways. If the multiversion index defined in Section 5.5 is used, then the content can simply be added to the metadata using the `root` column, but that is explained in more detail in Section 5.5.8. However, if the multiversion index is not used, for instance if the index is ordered or if no multiversion functionality is used, then the `objects` column is used instead. The `objects` column is simply a list containing the internal address for each Lumo object that is part of the index. This approach is undesirable due to the lists not being able to share any data between them, resulting in data potentially being duplicated.

Code block 8: Lumo schema defining the built-in metadata types for Lumo models

```
struct ModelMetadata
  snapshots &[]&SnapshotMetadata
  branches  &[]&BranchMetadata
  tags      &[]&TagMetadata

struct SnapshotMetadata
  id       ::uint64
  parentId ::uint64
  database &DatabaseMetadata

struct BranchMetadata
  name       &::string
  snapshotId ::uint64
  database   &DatabaseMetadata

struct TagMetadata
  name       &::string
  snapshotId ::uint64

struct DatabaseMetadata
  name        &::string
  collections &[]&CollectionMetadata
  relations   &[]&RelationMetadata

struct CollectionMetadata
  name    &::string
  type    &::Type
  indexes &[]&IndexMetadata

struct IndexMetadata
  name    &::string
  paths   &[]&[]::uint32
  unique  ::boolean
  ordered ::boolean
  objects ?&[]::address
  root    ?::address

struct RelationMetadata
  name ?&::string
  from &RelationEndpointMetadata
  to   &RelationEndpointMetadata

struct RelationEndpointMetadata
  collectionName &::string
  name           ?&::string
  paths          &[]&[]::int32
```

5.4 Multiversion index considerations

When enhancing Lumo databases with multiversion capabilities, the single most important question to answer is how to keep track of which Lumo objects exist in which snapshot (or branch). The chosen solution will have a very large impact on the performance of the database, both in terms of speed, memory usage, and possible feature set available.

5.4.1 Caveats working with Lumo object addresses

Instantiating a Lumo object from an address is not entirely free, and a Lumo object takes up far more memory than just the address. For the purpose of this thesis it is assumed that it is entirely adequate to store Lumo objects as addresses until the objects are explicitly requested by the user, even if having all Lumo objects already instantiated would speed up certain actions.

If a Lumo object address is copied without the storage manager knowing about it, the manager might later release objects that are still referenced somewhere, leading to invalid accesses if those unaccounted references are ever dereferenced. Inaccurate internal reference counts can also lead to the redirect-on-write mechanism of the storage manager not working as intended, leading to in-place writes in cases where a copy and redirect would be needed. The opposite occurs if an address is deleted without the storage manager being notified. For this reason any copying and deleting of Lumo object addresses also requires that its reference count is modified via the storage manager.

5.4.2 Key performance criteria

For the sake of simplicity, let us only consider Lumo objects of one Lumo type, or in Lumo database terms, Lumo objects part of a single collection. Let us also assume that the Lumo objects in the collection need to be indexed on a certain key column, and that the index implementation, once the database has been instantiated, uses a simple hash map for looking up the key column value. *The naive solution* for keeping track of objects in a snapshot would then be to, for each snapshot, simply store the address of each Lumo object part of that snapshot in a list. The following criteria need to be taken into account when choosing and evaluating a possible solution.

1. **Efficient loading/instantiating of snapshots.** Due to the non-zero cost of instantiating Lumo objects, this should be avoided if possible. However, some use cases might warrant the immediate instantiation of all Lumo objects in a snapshot. The naive case would need to do this to be able to build the index, since no key column data is stored alongside the addresses in the list. If the Lumo objects are kept instantiated, those can directly be used as the values in the index hash map, but if the addresses are copied into the hash map as the values instead, an extra cost of increasing the reference counts via the storage manager need to be taken into account.

2. **Efficient cloning of snapshots.** Cloning is done when a new snapshot is created from a branch and when a new branch is started from a snapshot. For the naive case this would not only mean copying the entire list of addresses, but also going through each address in the list and increasing the reference count for it. This extra work during a clone should be avoided if possible. Cloning a snapshot should not require instantiating the actual content of the snapshot either.

3. **Efficient saving/releasing of snapshots.** Throwing away a snapshot is a supported feature for the Lumo model, the reasoning for which was provided in Section 5.2.2, and doing so, especially for short-lived temporary snapshots, should be as efficient as possible. In the naive case this would result in each Lumo object address in the list needing its reference count decreased, since the addresses are being deleted. If the snapshot is saved instead of thrown away the list of addresses can instead simply be moved to the Lumo metadata.

4. **Efficient access to any object in any snapshot.** No matter how many snapshots there are in total, accessing any one object in them should be as fast and efficient as possible. Furthermore, whether the snapshot is the oldest or the most recent one should not have an effect on the access speed. This criteria also relates to the speed at which the correct object is found within the snapshot. In the naive case the access speed is very slow if the index hash map has not been created yet, because there is no relation between the address stored in the list and the content of the Lumo object it points to. Once the index has been created though, the access speed is as fast as the hash map allows it to be.

5. **Maximize the amount of shared data between snapshots.** The naive solution is terrible in this aspect since each snapshot has its completely separate list of addresses, even if the lists would be identical. If two or more snapshots and their databases are instantiated, the index hash maps will also contain large amount of duplicate information. Sharing identical parts of the address lists and hash maps between snapshots would decrease the memory usage.

6. **Minimize the amount of data needed in addition to the addresses.** The naive solution is perfect in this regard since the only data stored by the snapshots are the addresses themselves. If another data structure would be used instead of a simple list there might be information other than the addresses, such as nodes in a tree/trie, which would need to be stored as well. The goal is to have the addresses be as large of a part of the total amount of data as possible.

It is self-evident that any indexes in the multiversion database should be as fast as possible. In the naive case, the data structure for the index was a hash map, which could be seen as fast enough in most cases, and could be made even more efficient with another choice of data structure. On the other hand, if the data structure would inherently support multiversion data, and if it would be efficient to store alongside each snapshot, this index data structure could replace the address list in the naive case.

5.5 Multiversion Hash Array Mapped Trie (MHAMT)

The Multiversion Hash Array Mapped Trie is a modified HAMT implementation designed specifically for Lumo, following the key performance criteria presented in Section 5.4.2. Section 7.1 will later discuss how well the key performance criteria hold with regards to the MHAMT.

5.5.1 Justifications for using the HAMT as base structure

Reasons for basing the multiversion index on the HAMT are, among others, the high potential for fast lookups on any version and the large scale memory sharing between multiple tries. Furthermore, other base structures can include caveats which make them unsuitable, such as having a slow lookup on old versions if the number of versions is large. Integration with Lumo in particular can also be a hindrance for some structures, since Lumo indexes do not restrict which kinds of key column types that can be used, and there is no assurance that the key values are order-able. The index also needs to support non-unique keys, something that might not be feasible to implement on some structures, restricting the index structure even further, and structures that are dependent on timestamps are out of the question completely.

5.5.2 MHAMT implementation details

The values stored in an MHAMT are Lumo object addresses. The MHAMT nodes are themselves also Lumo objects and are referenced to by other nodes using their address. This means that all children of a node are Lumo object addresses, either an address to another node or to an object of the collection type. To know which child is a node and which is a value, the highest bit of the child address is used to differentiate the two alternatives. This works because the children are set to be 64 bits in length and Lumo object addresses will never use all 64 bits.

The MHAMT nodes are internally stored as Lumo objects of the type `[]::uint64`, i.e. a dynamically sized list of 64-bit unsigned integers. The list contains each child Lumo address of the node, as well as an extra element used as a *header*. The header is used to store the bitmap of the node, but also other information about the node which will be explained later. The header size is 64 bits like the children only for ease of implementation. When referring to the number of children that a node has, the term *node size* will be used.

To find the correct place in the MHAMT for a certain Lumo object, its key value is first hashed. The key value is found in a certain column of the Lumo object, but this is something which is defined in, and taken care of by, the Lumo index holding the MHAMT (see Section 5.1.3). The hashed key value is then partitioned into three-bit segments that are used to traverse down the trie using the method described in detail in Section 4.2.4. Segments of three bits give a node cardinality of eight and a maximum trie depth of 21 (a maximum of $21 \cdot 3 = 63$ bits are used), assuming a 64-bit hash and the root node being on level 0.

The MHAMT implements the usual data structure methods for fetching, inserting, updating, and removing data. Fetching data can be done using two separate methods,

Code block 9: Simplified interface for the MHAMT

```
interface MHAMT {
    // Clone this trie
    clone: () => MHAMT;

    // Find the first object with a specific key within this trie
    find: (obj: LumoObject) => LumoObject | null;

    // Find all objects with a specific key within this trie
    findAll: (obj: LumoObject) => LumoObject[];

    // Insert an object into this trie, always successful
    insert: (obj: LumoObject) => void;

    // Update an existing object in the trie
    update: (obj: LumoObject, new_obj: LumoObject) => boolean;

    // Remove an existing object from this trie
    erase: (obj: LumoObject) => boolean;
}
```

depending on if the result should contain all matching objects or just the first one, the reason being that the MHAMT supports non-unique key values. See Code block 9 for the relevant parts of the implemented MHAMT interface.

5.5.3 Shared MHAMT nodes and reference counts

MHAMT nodes can be shared between several tries, effectively sharing each node and value below that node. Tries cannot directly modify a shared node, since that would inadvertently change the view of other tries as well. If a trie wants to modify a shared node, it first has to make a copy of the node for itself. However, since MHAMT nodes themselves are Lumo objects, the storage manager will keep an internal reference count for each of them, and trying to modify a shared node will actually, due to Lumo's built-in support for ROW, automatically make a copy of it before modifying it.

Even if the copying of a node itself is automatic, it does not mean that the copy process is over. Making a copy of a node also means that there now exists an extra set of references to its children, which are still unaccounted for. To finalize the copy process the trie therefore has to make sure to increase the reference counts for each child of the copied node, child nodes and value objects alike, otherwise objects could automatically be disposed of even though references to them still exist.

Modifying a shared node will, as explained, create a new node. This new node node will have a different address than the original node, which needs to be updated in the parent node as well. If the parent node is shared as well, it too will need to

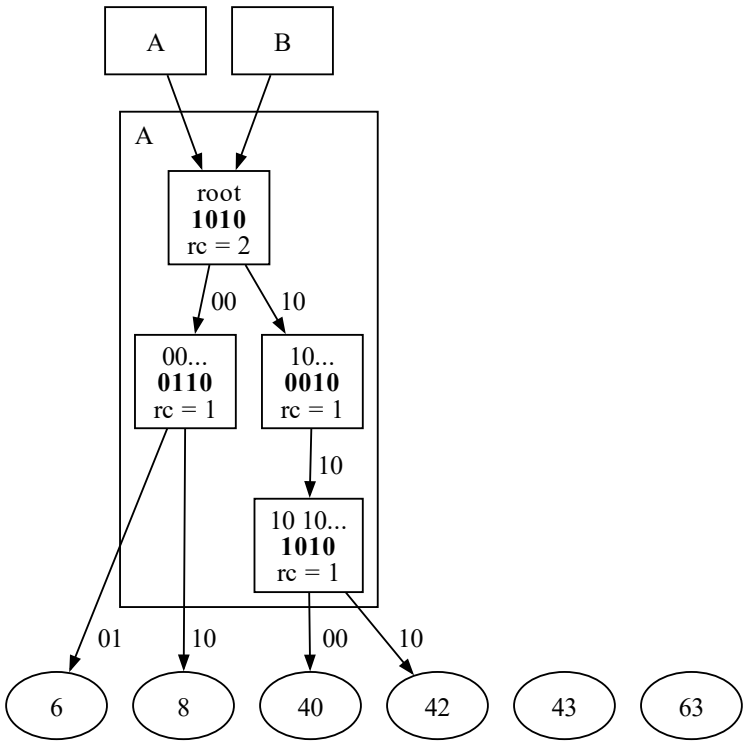

Figure 7: MHAMT *A* has had four values inserted and has due to that four nodes which it owns. Trie *A* has then been cloned to get trie *B*, resulting in the root node of trie *A* to have a reference count of two. The graph shows the state of the tries after cloning. The reference count is shown for each node, but otherwise a similar graph construction has been used as for the HAMT in Figure 6 on page 30.

be copied. The process of updating node addresses in parent nodes continues until a non-shared node is encountered. Going the other way, from root node to modified node, every node from the first shared node onwards will need to be copied.

It is worth mentioning that, due to this node sharing mechanism, the MHAMT can become a persistent data structure if needed. As explained in Section 4.2.4, a persistent data structure retains all previous versions of itself when modified, which is true for the MHAMT as long as the root node is kept shared. This can be done by always taking an extra reference to the root node after making a modification, which will force the next modification to make copies of all nodes it modifies, leaving the current state (version) intact. However, preserving exactly all versions of the MHAMT is not needed, only the versions that correspond to snapshots, which is why taking an extra root node reference will only be done when the state is stored as a snapshot.

5.5.4 Node ownership

The notion of a node owner would be an alternative way of deducing when a node needs to be copied. The logic could be that a trie could freely modify nodes it owns but would need to make its own copies of nodes owned by other tries. However, that

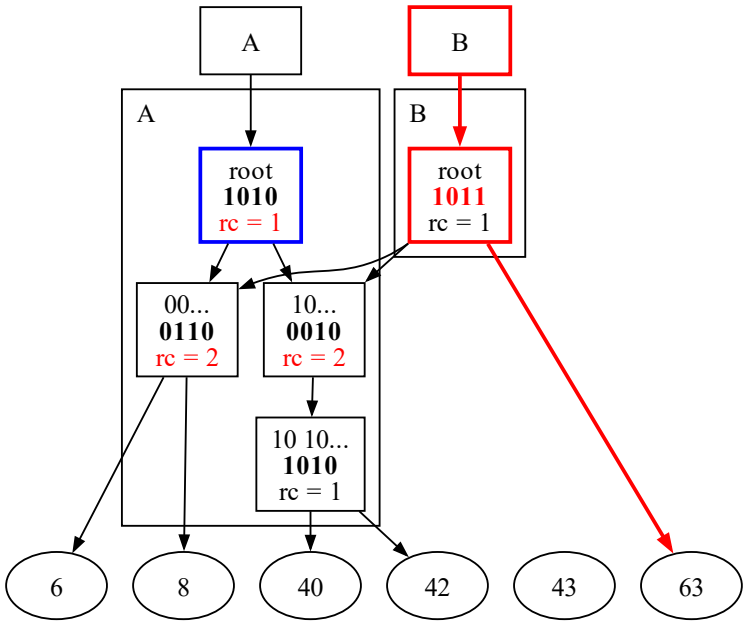

Figure 8: The state of the MHAMTs in Figure 7 after the value (hashed key value) 63 has been inserted into trie *B*. The value could be inserted directly on the root node, but since the original root node (colored blue) of *B* was shared with *A*, it first had to make a copy of it. In this case, due to Lumo's built-in ROW support, the copying of the node was done automatically when *B* tried to update the node content. However, the trie had to manually increase the reference counts of the root node children.

does not solve the problem of shared nodes, since situations where a trie would need to copy a node it already owns due to another trie also using it can still arise.

Node ownership is nevertheless an important metric when it comes to benchmarking the MHAMT and seeing how it behaves with certain data. Keeping track of which trie created which node allows the user to analyze the trie structures between operations. The user could check, for instance, how many new nodes are created when inserting a value into a cloned trie. This is achieved by, in the header of each node, storing an integer id of the trie which created it. Since node ownership does not serve a functional purpose, only an analytic one, it is only used for testing purposes.

5.5.5 A simple MHAMT example

Figures 7–10 show a series of modifications to two simplified MHAMTs, of which the second one starts off as a clone of the first one. Nodes are shown as rectangles and are placed inside larger boxes representing the spheres of ownership of the tries. The nodes show the partial hash used to get to it from the root node and, in bold, the bitmap which keeps track of which child indexes exist, just like in Figure 6 on page 30. In addition, the internal reference count for each node is shown as well, even though that information is not stored in the nodes themselves like the graphs would suggest. The reference counts are instead kept track of by the storage manager and requested by the nodes on demand. Inserted Lumo objects are shown as ellipses and are not owned by

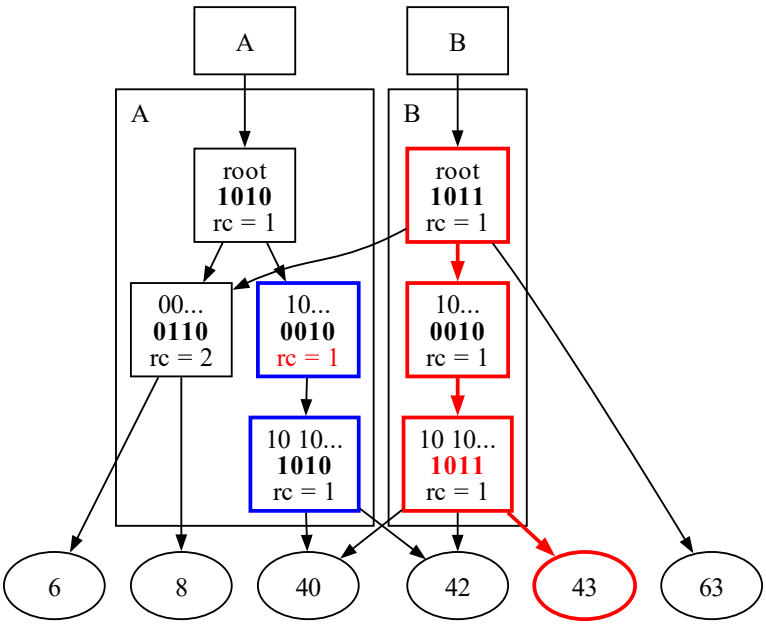

Figure 9: The state of the MHAMTs in Figure 8 after the value (hashed key value) 43 has been inserted into trie *B*. The value could be inserted into a node on the third level, but since there was a shared node in the path from the root node it first had to be copied. Since the internal reference count for that node was originally 1, trie *B* first had to take an extra reference to the node (increasing the count to 2) before modifying it, to be able to take advantage of Lumo's built-in ROW. When adding the address of the new value into the node, ROW automatically created a copy of the node, which in turn needs to have its address stored in its parent node. Since the parent node was shared, that also had to be copied before updating the child node address. Lastly, the new parent node address had to be updated into the root node of trie *B*, but that could be done without further copies since only trie *B* had a reference to it.

any trie. The value depicted inside an inserted Lumo object is a six bit long hashed key value of the object, which has been used two bits at a time, starting from the highest bits, to find the correct place in the trie. This is where the simplification lies, since in a real MHAMT the hash is 64 bits and is used up three bits at a time starting from the lowest bits, but to make the graphs more comprehensible the simplified structure was chosen. Items (nodes, node content, references/edges and values) which have changed due to the modifications since the last graph are colored red, while a node shown in blue means that it has been copied due to the modified trie not owning it.

5.5.6 MHAMT support for hash collisions and non-unique keys

The ability to insert two objects with the same key into the same MHAMT is an important feature in many use cases, but is not supported by the HAMT. Since it is in reality the hash value of the key that is used to place the object in the trie, solving the non-unique key case would automatically also solve the case where hash values are identical unexpectedly, i.e. hash collisions.

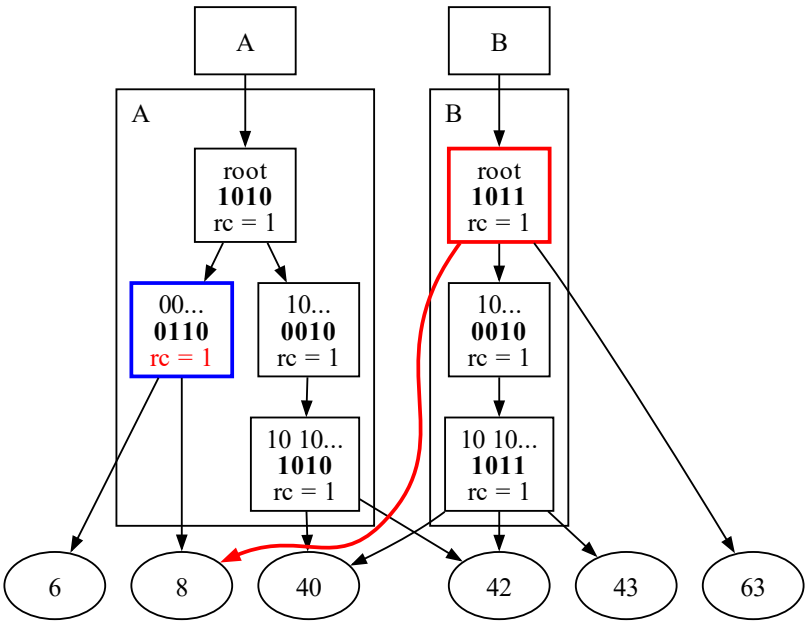

Figure 10: The state of the MHAMTs in Figure 9 after the value (hashed key value) 6 has been removed from trie B. Since the node containing the value in question was shared, trie B first had to make its own copy of the node. Removing the address of the value object in question from the new node resulted in the node only having one child left, which meant that the node could be deleted and the remaining value moved up to the parent node. The parent node, in this case the root node of trie B, was not shared, so it could be updated without copying it.

To support this, a new node type is introduced, a *list node*, which can only store addresses to value objects, not addresses to other nodes. The addresses are stored in a sorted order to make searching for a specific address faster while sacrificing performance on inserts and deletions. Every value in the list node will additionally produce the same key hash value. To be able to signify that a node is in fact a list node, a one bit flag needs to be introduced and stored in the header of each node. Still regarding the header, note that no bitmap is needed for the list node, but the number of children is needed to be stored instead. When a list node is encountered during a search, more than one value can be returned as a result, but extra care has to be taken to ensure that all objects returned have the correct key and are not simply hash collisions. See Figure 11 for a simple example of the creation of a list node in a shared MHAMT.

5.5.7 Lumo-specific details in MHAMT operations

The MHAMT implementation described in this thesis is designed with Lumo in mind, and is therefore very dependent on the inner workings of Lumo. Nonetheless, implementing a more generic MHAMT is possible. The following operations and logic are in the current implementation performed entirely by Lumo, and need to be given extra thought if an implementation independent from Lumo is to be conceived.

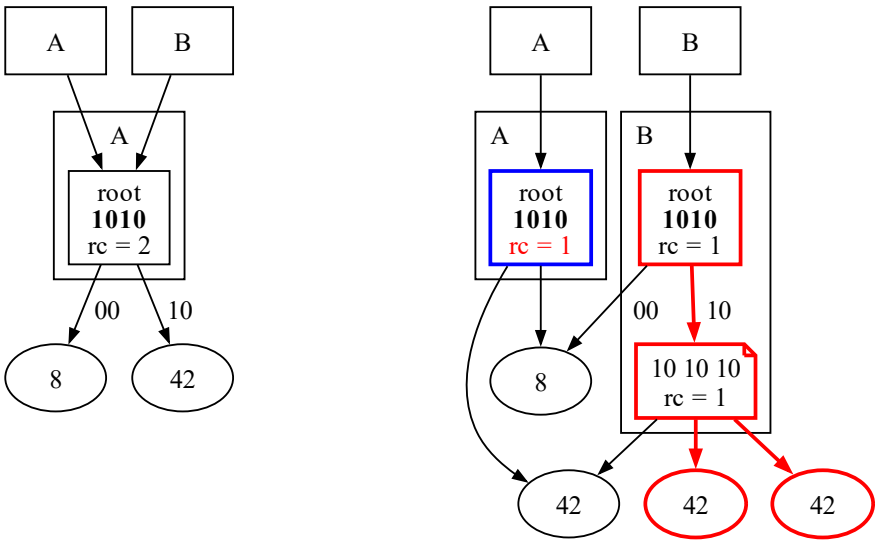

Figure 11: The left graph shows the initial state of two MHAMTs, while the graph to the right shows the state after two additional Lumo objects with the key hash value 42 have been inserted into trie *B*. Since there already existed a key 42, a list node was created and inserted into the root node instead of the original key, resulting in a copy of the root node. The original key and the two new identical keys have then been added to the new list node, but the original key is still referenced by the root node of trie *A*. Note that the list node does not use a bitmap, and that the hash '10 10 10' is not actually stored on the node either.

Hashing When traversing an MHAMT, a partitioned hash is used, more specifically a 64-bit hash. This hash is acquired by hashing the key column values of the Lumo object that is searched for, and the hashing is done entirely by Lumo. It is important to avoid an excess amount of unnecessary one-child nodes in the MHAMT, since one-child nodes are not as memory-efficient as other nodes. This is achieved by using a as uniformly distributed hashing algorithm as possible. If there are two values in the MHAMT and their hashes are identical except for the last few bits, many one-child nodes will be added to the trie until the differentiating bits are compared. As an example, the node '10...' in Figure 7 on page 48 is a one-child node. Furthermore, it is of course preferable to minimize the amount of hash collisions as well, and thereby minimize the amount of list nodes, since large list nodes can be slow to search through.

Equality checks Since hash collisions are possible, it is important to have a mechanism of checking equality between key values without hashing. This is only used when a list node is encountered during a search, because hash collisions are always stored in list nodes. Since Lumo objects allow for complex schemas to be used, and almost any set of columns can act as key columns for the MHAMT, the equality check can be quite laborious as well, which is why it is handled by Lumo.

Modifying duplicate objects Objects with the same key are allowed in an MHAMT, and they are stored within the same list node. Since they have the same key, it is impossible to tell them apart using the key only, which makes updates and deletes of duplicate objects difficult. That is why another feature of the Lumo objects is used to tell the duplicates apart, namely the Lumo address of the objects. This means that to be sure that the modification is successful, it is imperative that the exact object which currently exists in the trie is provided when performing the operation. This is done by first finding the correct object from the trie and then cloning it, creating a new Lumo object that points to the same place in the storage manager, i.e. having the same Lumo address.

Node storage Since MHAMT nodes are implemented as Lumo objects, and are thereby stored in Lumo's storage manager, they need to be fetched and instantiated to be used. While traversing down the trie, each new node is therefore instantiated and kept alive by the trie until the search or modification operation is complete, after which all of the fetched nodes are disposed of. Handing over responsibility of memory management to Lumo is advantageous because it lets the MHAMT leverage all of Lumo's built-in features surrounding this, such as reference counting and ROW.

Cloning tries To use the multiversion features of the MHAMT it is necessary to be able to clone an existing trie. In this implementation, this operation is extremely easy, because the only thing needed is for the new trie to take a reference to the root node of the cloned trie.

Releasing tries Releasing an entire MHAMT involves checking if its nodes are shared with other tries or not. The release process starts by releasing a reference to the root node, and if that was a shared node, the process simply stops. If, however, the root node was not shared, the node will automatically be released by the Lumo storage manager, which in turn will delete all child references stored in the node without actually releasing them. Therefore, to avoid leaking memory in the storage manager, each root node child (both nodes and values) need to manually have their reference counts decreased. The release process needs to be done for each non-shared node in the trie, until a shared node is encountered.

5.5.8 MHAMT integration with Lumo databases

Lumo indexes (as described in Section 5.1.3) using the MHAMT implementation only need to keep track of the Lumo address of its trie's root node. Modifications to a node seen by more than one trie will result in a new root node for the index, in which case the new address is kept track of instead.

Trie traversal Any operation on the MHAMT index involves searching the trie for a specific hash. The hash is produced by hashing the key value of a Lumo object, which is then partitioned and used to traverse the trie. The trie traversal starts from the root

node of the index, and nodes are loaded from the storage manager along the way. The result of the search can either be a value on a normal node, a value on a list node, or nothing at all.

Selections Selections in an MHAMT index are done by providing a Lumo object with the key value column(s) set to the appropriate search value(s). If a Lumo object address is ultimately found, it is instantiated and its equality validated before returning a clone of it. If an address is not found, or if the found key value is not equal to the requested one, nothing is returned. If the address was found in a list node though, then each value in the list is checked for equality until a match is found, which is then cloned and returned as usual. Since the MHAMT index can store non-unique key values, it is possible to fetch several values in one call, but in that case the range select method must be used.

Inserts Inserting a new Lumo object into the MHAMT index is always successful, assuming the object is of the correct type, since storing duplicate keys is supported by the MHAMT. If an existing Lumo object is found in the trie it either has an identical key value or it is a hash collision. In any case, a new list node is introduced into the trie, assuming the existing object was not already in a list node.

Deletes and updates To delete an existing value from an MHAMT index a Lumo object with the corresponding key value(s) is to be provided. The index selects the existing object from the MHAMT to check if it exists, and erases it from the trie if it is found. The deletion from the trie is done using the Lumo address of the selected object, because a simple equality check is not enough to identify which object the user attempts to delete if duplicates exist. Updates are done in the same manner as deletes, except that the found Lumo address is replaced by the address of the new object instead of being erased.

Saving to metadata Once a snapshot is created, or if the state of a branch is saved, the state of the database should be stored, which is achieved by saving the state of database indexes. Saving the state of the MHAMT index is as simple as storing the Lumo address of the root node into the `root` column in the model metadata (see Section 5.3.2). As long as a reference to the root node is kept in the metadata, the root node will be kept alive and the whole MHAMT will be impervious to outside modifications. The root node address can then be loaded from the metadata at a later point in time to be able to access the state of the index.

6 Benchmarks and results

To verify that the MHAMT implementation works as expected, and to benchmark its real-world performance, it was tested with real BIM data. The performance aspect most relevant to this thesis is the memory usage of the tries themselves. Performance related to speed of operations or similar is certainly important too, but large parts of the time using the MHAMT will be spent in Lumo doing operations such as hashing and fetching nodes from the storage manager, which are outside the scope of this thesis. The resulting trie structure after certain operations are also worth investigating, even if it is not entirely dependent on the MHAMT implementation itself.

6.1 The test model

Lumo is not used in production within Trimble products yet, so any real-world data stored natively in Lumo is hard to come by. Recent efforts have on the other hand resulted in a conversion tool which can convert existing BIM models of the TrimBIM [77] format used within Trimble products to Lumo objects of a predefined schema.

To be able to test the MHAMT using real-world data, a rather large existing TrimBIM model was chosen as a test subject and was converted and stored as a Lumo model using the aforementioned conversion tool. More specifically, the TrimBIM model in question was the result of the real construction project of the shopping centre Lippulaiva in Espoo, which was built using Tekla Structures. The Lippulaiva data consist of entities that represent certain geometric structures at certain positions, and having certain properties. The entities use a byte array of size 16 (the Lumo type `[16]::uint8`) as a unique identifier column. Entities are stored in an MHAMT index, using the identifier column as the key column. It must be noted that, due to the uniqueness of the identifier columns and the risk of hash collisions being almost zero, no list nodes are present in the resulting MHAMT indexes.

The Lippulaiva model was originally built and saved incrementally using Tekla Model Sharing, resulting in 769 increments, which are called Model Sharing diffs. The diffs are saved individually in a way where only the differences to the previous version are included. Diffs consist of not only inserts of new entities, but also deletions and updates of old entities. The first version of the Lippulaiva model is 8.2 MiB, and taking the remaining 768 diffs into account the last version is 164 MiB. During the conversion to Lumo, each increment of the model was converted into its own Lumo snapshot, resulting in the final Lumo model having 769 snapshots. Each snapshot stores its own MHAMT index that keeps track of which entities exist in that specific snapshot. The TrimBIM file format was designed specifically for storing BIM data as compact as possible [77], so it is no surprise that the resulting Lumo model is almost six times as big, or 961 MiB, when saved to disk compared to the TrimBIM model, especially since there has not yet been any attempts to optimize the model save size in Lumo.

6.2 Memory usage

6.2.1 MHAMT size

The memory usage m_s (in bytes) of an MHAMT node of size s (number of children) can be expressed as

$$m_s = 8(s + 1). \tag{1}$$

This is because the node header and all s child addresses are 64 bits, or 8 bytes, each. To calculate total memory usage M of an entire MHAMT, all different node sizes need to be accounted for, giving the formula

$$M = \sum_{s=0}^{8} n_s m_s = \sum_{s=0}^{8} 8n_s(1 + s) = 8 \sum_{s=0}^{8} (n_s + n_s s) = 8 \left(N + \sum_{s=0}^{8} n_s s \right), \tag{2}$$

where N is the total amount of nodes in the trie and n_s is the amount of nodes of size $s \in [0, 8]$. The node size is between 0 and 8 because normal MHAMT nodes can have a maximum of eight children. However, only the root node of an empty trie can have size 0, so $n_0 = 0$ for all non-empty tries.

The Equation (2) does not, however, take into account all list nodes, because their size can be larger than eight. The final sum in the equation would need to change to an infinite sum to capture all possible sizes of list nodes. On the other hand, the sum represents the total amount of Lumo addresses stored inside the nodes of the trie. Another way to express this, which would also take into account all list nodes, would be to sum the total amount values V stored in the trie and the total amount of nodes N, disregarding the root node since its address is not stored anywhere in the trie itself. This gives an alternative formula for the total memory usage of an MHAMT:

$$M = 8(N + (N + V - 1)) = 8(2N + V - 1) \tag{3}$$

One of the main benefits when using MHAMT with regards to memory usage is that nodes can be shared between tries. Cloning a trie does not necessitate any new nodes being created since it can use all the same nodes as the original trie, and it is only when the new trie is modified it has to create its own nodes. This means that when calculating the memory consumption of a specific MHAMT, only its own nodes should be considered, not the nodes owned by previous tries. The nodes owned by a specific trie can be deduced using the method described in Section 5.5.4.

6.2.2 Lumo model size

In a Lumo model where many snapshots index the same data using MHAMT, i.e. having many versions with the same data, the node sharing mechanism of the MHAMT is quite effective. Figure 12 shows the cumulative memory usage of the MHAMT indexes for the test model in the actual shared case compared to a theoretical case where nodes cannot be shared between tries. In the shared case, only the nodes owned by the snapshot itself have been considered, meaning nodes owned by other snapshots have been omitted in the calculations. The chosen test model only has 769 snapshots,

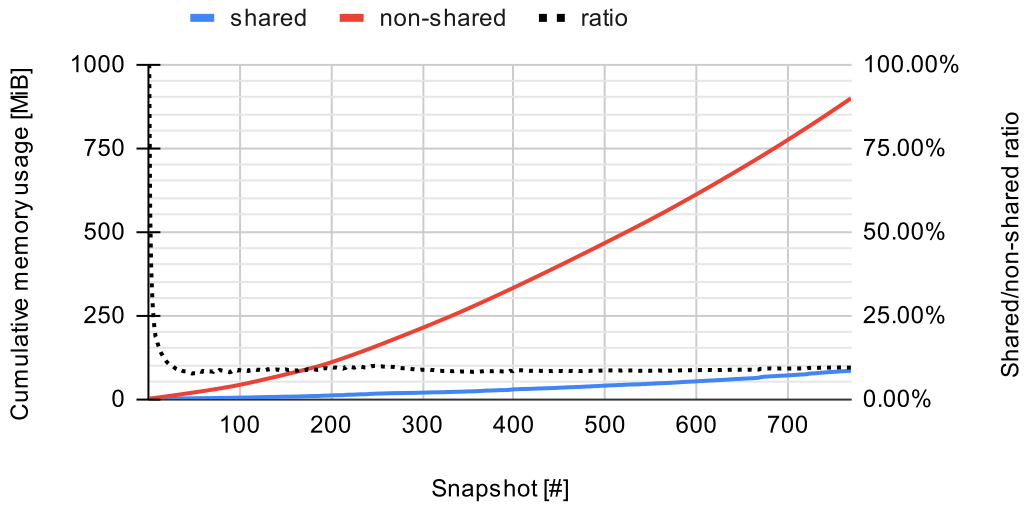

Figure 12: Comparison of the cumulative memory usage by MHAMTs between the actual case where MHAMT nodes can be shared between tries, and a theoretical case where no sharing is allowed. The memory usage has been calculated using Equation (2). The ratio between the shared and non-shared cases are shown with the black dotted line.

but the experiment still gives a good indication that the MHAMT node sharing feature is quite effective in reducing the total memory consumption.

The first snapshot of the test model introduces 24122 entities split on 11369 MHAMT nodes, and the last snapshot of the model has a total of 124311 entities and 59631 nodes. Each snapshot in the model has on average 130 (median 35) more entities than the previous version, and an MHAMT index with 63 (median 19) more nodes on average. On average, each snapshot only owns 6.0 % of the nodes in its trie, which leads to each snapshot needing more than 90 % (see ratio in Figure 12) less memory when sharing nodes with other snapshots compared to if no sharing was allowed. The final total memory consumption in the two cases are 83.9 MiB and 898 MiB respectively, a decrease of 90.7 % across the whole model.

6.3 Trie structure

Analyzing the structure of the MHAMTs in the test model reveals how well the tries are balanced. An MHAMT that has no unnecessary one-child nodes and uses up 100 % of the capacity of each level of its trie is in this thesis regarded as well-balanced. Such a trie should have its values on as low a level as possible, but continuously move them to higher levels as new values are inserted. The balance of an MHAMT is entirely dependent on the data and which key value columns and hashing algorithm are used, and is therefore difficult to draw conclusions about with this small data set. Additionally, these are all external traits which can be chosen independently from the MHAMT implementation, but the balance is nonetheless examined here to prove that

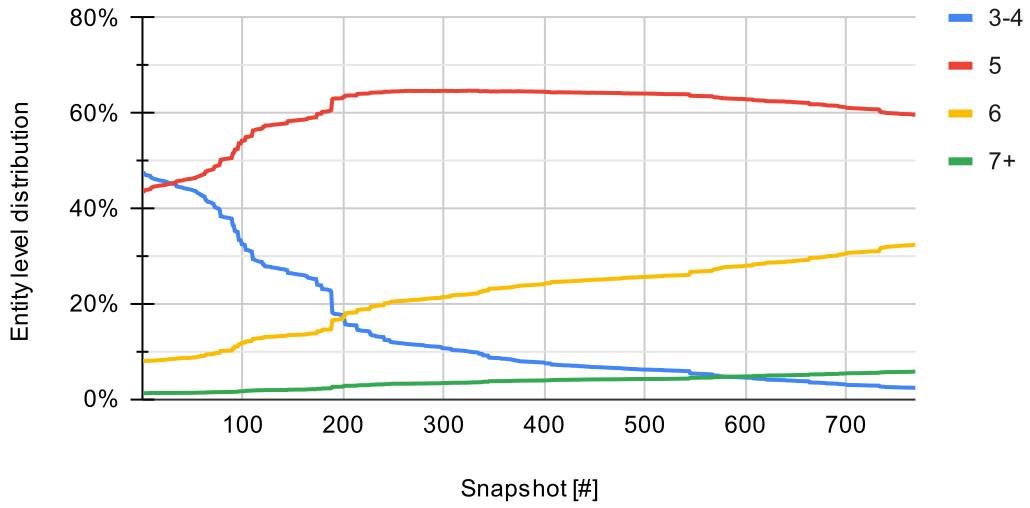

Figure 13: The storage level distribution for all entities in the MHAMT for each snapshot in the test model. The lines represent different entity levels, where the level of an entity in reality means the level of the node on which the entity is stored (root node is on level zero). All entities are included in the graph, so the lines sum to 100 %. No entities are ever stored on levels 0–2, so they are omitted from the graph, and likewise for levels eleven and higher. The last snapshot to store an entity on level three is snapshot number 188.

the current implementation works reasonably well with real-world data.

The initial 24122 entities in the test model should in theory all fit on nodes on the fourth level or below (root node is on level zero), since that level can contain a maximum of $8^4 = 4096$ nodes on which fit a maximum of $8^5 = 32768$ children. The fifth level nodes have space for $8^6 = 262144$ children, which is enough to store the 124311 entities of the final snapshot. However, in Figure 13 it can be seen that not even half of the entities are stored in nodes on levels four or lower, and that entities are stored on level seven and higher right from the start. Two entities are even stored on level nine in the first snapshot, which increases to 104 values by the last snapshot, which even has 14 entities on level ten. Figure 14 shows the average node size per trie level and indicates further that the MHAMTs are not very well-balanced. The contents of nodes on level four and five are investigated further in Figure 15.

Examining the size distribution of nodes can also reveal how well-balanced an MHAMT is. According to Figure 16, 78 %–86 % of all nodes in the test model are half full or less, taking into account all nodes across all snapshots. One-child nodes always have a size of one, and in the test model those represent 6 %–7 % of all nodes. Full nodes are to be preferred, because they store the most information relative to their memory usage, but they only amount to 4 %–7 % of all nodes at any given snapshot.

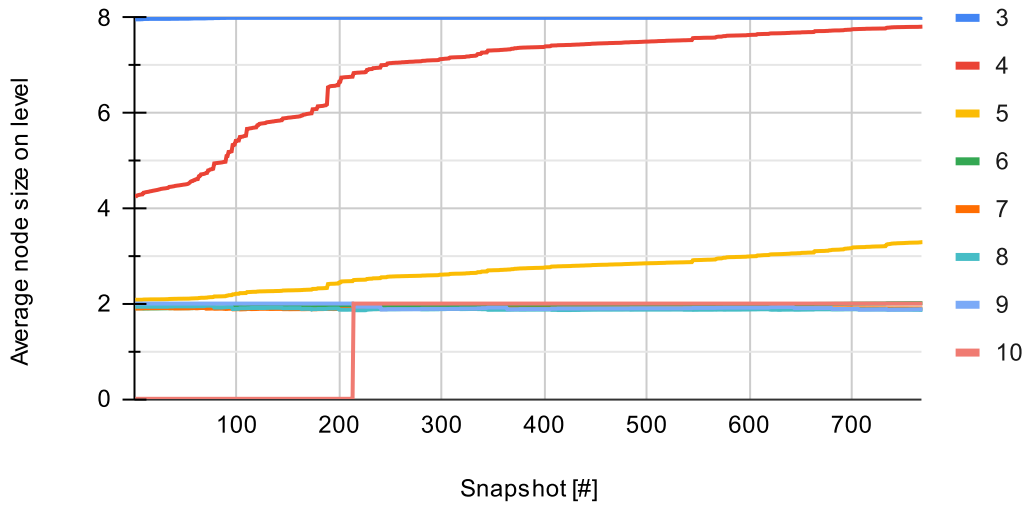

Figure 14: The average node size on each trie level across all snapshots of the test model. The lines represent different levels of the MHAMTs, and levels less than three are left out due to always being full. It can be seen that the first snapshot to store an entity on level ten is number 214.

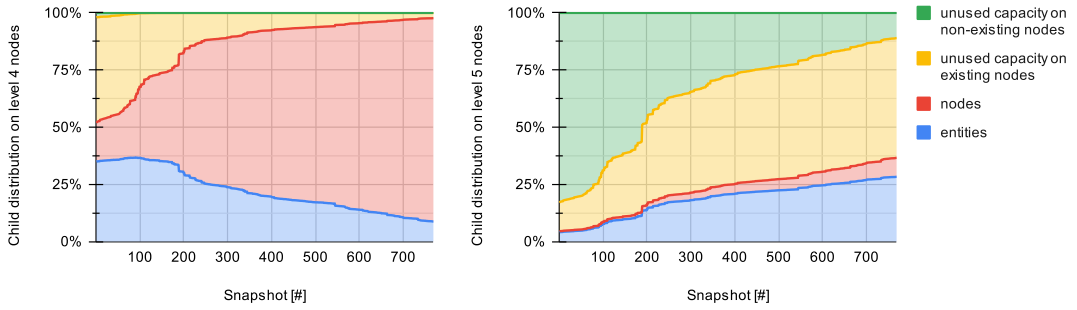

Figure 15: Two graphs showing the content of MHAMT nodes on levels four and five, respectively, throughout the test model. A large yellow (2) area indicates that the nodes on the level are far from full, and a small green (1) area means most of the nodes on the level exist.

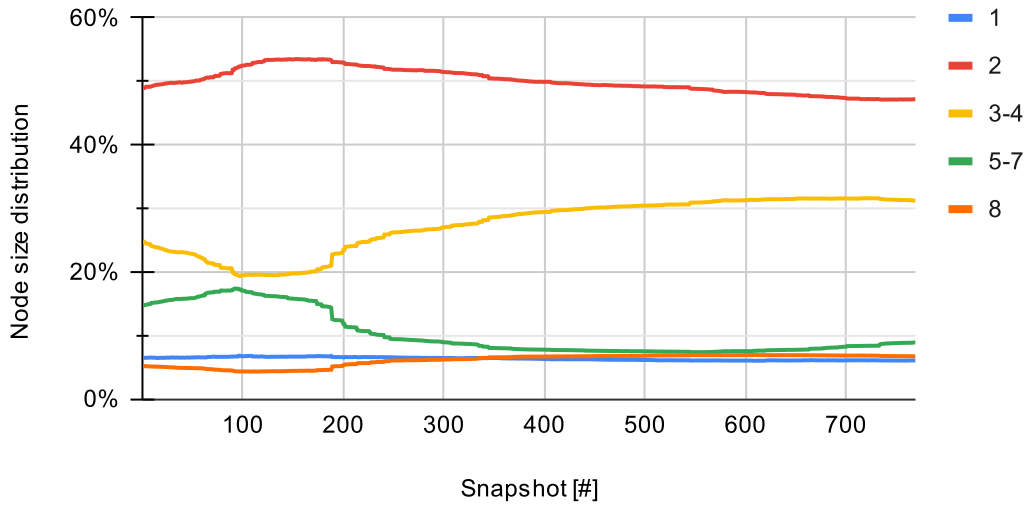

Figure 16: The node size distribution across each snapshot of the test model. Lines represent either one node size, or two to three sizes combined.

7 Discussion

The Lumo multiversion capabilities introduced and tested in this thesis are satisfactory and on a good level for future Lumo development to build upon. The testing using real-world construction data did, however, reveal a few shortcomings, which are discussed later in this chapter. Potential solutions and future enhancements are also suggested.

7.1 MHAMT key performance criteria examined

In Section 5.4.2 on page 44, a few key performance criteria were proposed for potential multiversion indexes to be assessed against. Here the MHAMT implementation is evaluated using the same criteria, and it is assumed that the Lumo databases in the snapshots only contain multiversion indexes that use the MHAMT implementation. It is clear that the MHAMT implementation described in this thesis in theory satisfies all key performance criteria:

1. **Efficient loading/instantiating of snapshots.** Instantiating a snapshot from Lumo model metadata simply means giving the MHAMT root node addresses to the multiversion indexes. There is no need to instantiate any Lumo objects at load time.

2. **Efficient cloning of snapshots.** Cloning a snapshot is done by taking extra references to the MHAMT root nodes and giving them to the new snapshot. No snapshot content requires instantiation during cloning.

3. **Efficient saving/releasing of snapshots.** Saving a snapshot is as simple as cloning it and storing its MHAMT root node addresses into the Lumo metadata. Throwing away a temporary snapshot requires releasing all MHAMT root node references, and if the root nodes are shared, which they should be in most cases, the release process is done. The process of releasing a non-shared trie is a bit more involved, but should not happen very often.

4. **Efficient access to any object in any snapshot.** Finding a Lumo object from an MHAMT is efficient, and there is no difference if the search happens in the oldest or newest snapshot.

5. **Maximize the amount of shared data between snapshots.** MHAMT nodes can be shared by many tries, and in the case of identical clones all data is shared. Modifying a shared MHAMT will diverge the contents of the tries, but most of the nodes will still be shared. This means that only only the changes made since the last snapshot will have an effect on the memory usage of the index.

6. **Minimize the amount of data needed in addition to the addresses.** MHAMT nodes have a header of nine bits, which is 1.7 % of the node size assuming the node is full.

7.2 Possible MHAMT enhancements specific to Lumo

In this implementation of MHAMT, Lumo is responsible for storing and keeping track of the nodes for the tries. The nodes are stored in the storage manager, but no attempts have been made to store them close to each other. They are simply stored in the same order they are created, and are mixed with other Lumo data as well. Taking advantage of caching when loading several nodes in a row, by for example implementing allocation pools inside the storage manager, would undoubtedly lead to more efficient trie traversals.

Lumo is also responsible for hashing the key values of the Lumo objects, and having a well behaving hashing algorithm which minimizes the number of nodes in the MHAMT is important. Any part of a Lumo object could in theory be used as a key value, anything from an integer or a list of bytes, to a string or a child object, or a combination of them all. Choosing the most optimal hashing algorithm and strategy for a given data set is therefore a difficult task and needs to be investigated further.

7.3 MHAMT node content enhancements

The nodes in an MHAMT can have up to eight children, either values or other nodes, but the optimal choice of node cardinality has not been investigated in this thesis. Having larger nodes would lead to needing fewer of them, but would need larger bitmaps per node, so it is definitely a trade-off to consider. Taking into account that shared nodes are copied once they are modified, larger nodes could lead to more data needing to be copied for a single change as well.

Due to the usage of a bitmap for keeping track of existing children, which is one of the core features of the HAMT, the cost of empty children is only one bit per child. However, once a node is full it is very unlikely to ever shrink again, especially once it is shared with other tries, in which case the bitmap becomes unnecessary. One possible improvement would be to not store the bitmap at all once the node is full, saving at least one byte per full node. However, depending on how well-balanced the trie is, the memory savings might not be as exceptional as one might believe, at least looking at the real-world MHAMT example in Chapter 6, where the percentage of full nodes in a given snapshot trie is only 4 %–7 % (see Figure 16).

The node header is in the current implementation 64 bits, even if only the bitmap and the list node flag are stored in it, amounting to nine bits. This is due to ease of implementation, allowing the node to be represented with the `[]::uint64` Lumo type. For list nodes not even the bitmap is needed, but there a counter for the number of children is needed instead. In a more optimized implementation the header would be smaller, and the node child addresses could probably be shrunk in size too, decreasing the memory footprint of a node significantly.

7.4 MHAMT trie structure enhancements

Inserting two objects whose hashed key values are identical up to, but not including, the very last bits of the hash will with the current MHAMT implementation produce a

large amount of one-child nodes. This is because the hashes are compared three bits at a time, and if the bits are identical, a new node is inserted before the next three bits are compared. These one-child nodes are empty except for the address to the next node, which is not very memory-efficient. This should not happen very often with a uniformly distributed hashing algorithm, but it would nonetheless be a good idea to be able perform some path compression by removing one-child nodes. This could be done by introducing a separate node type which could handle a larger hash partition than three bits, removing, or at least reducing, the need for one-child nodes. The real-world MHAMT example showed that 6 %–7 % of all nodes have size one and are therefore one-child nodes (see Figure 16), but the occurrence rate and impact of one-child nodes need to be investigated further before any complexity regarding them is introduced into the MHAMT implementation.

The list nodes introduced in Section 5.5.6 also offer a possible point of enhancement, since the current implementation can kill performance if the list sizes are huge. This is because searching for a specific element in the list is quite slow, even if the elements are ordered. Having to make copies of a large list node every time it is modified is not a desired consequence either. One possible solution would be to not have any list nodes at all, but instead simply keep using normal nodes. The trick would be to change the list node flag to instead signify that a node, and all of its children, store objects with the same key hash. Once such a node has been hit in a search, rather than continue using the key hash, the Lumo address of the object would be partitioned and used to traverse the rest of the trie instead.

It must be mentioned that the test model did not include any duplicate keys at all, neither deliberately by using a non-unique index nor unplanned in the form of hash collisions. Real-world testing of the list node therefore still needs to be carried out before any conclusions about its performance can be drawn.

7.5 Missing MHAMT features

The current MHAMT implementation only supports unordered indexes. This is a serious shortcoming since there are many use cases for Lumo where having a multiversion index for ordered data and range queries would be useful. It still remains to be seen whether the MHAMT can be enhanced to support ordered data, for all or for at least certain key value types, or if a whole separate multiversion index solution is required in addition to the MHAMT.

8 Conclusions

The focus of this thesis was on implementing multiversion indexing in a proprietary database management system called Lumo, and simultaneously extending it with support for Git-like functionality such as snapshots and branches. Lumo is still in active pre-release development and it is built for handling generic data, but the precursor to Lumo was mainly used for BIM data, which is why such use cases are important and used as examples throughout the thesis.

Existing snapshot implementations and their use cases were researched, and a suitable API for our own implementation was investigated. Branches, like those used in Git workflows, were studied in a similar manner. Existing multiversion index structures were studied as well, and the Hash Array Mapped Trie (HAMT) was recognized as a good candidate base structure for the Lumo multiversion index. The HAMT is a trie structure where the keys are hashed and partitioned, and then used up one segment at a time to traverse down the trie. A HAMT node contains a bitmap which indicates the existence of the children of the node, leading to empty children having a memory cost of only one bit each. The motives behind choosing HAMT as the base structure above other structures were justified.

The current state of Lumo database adapters was provided, and the new Lumo model implementation explained, including interfaces for interacting with snapshots and branches. The Lumo save and load mechanism was also discussed, especially with regards to how it is performed with the new Lumo model.

Key performance criteria for the new multiversion index were then defined, and the need for efficient loading, saving and cloning of snapshots, and efficient access to any object in any snapshot, were considered essential. Maximizing the amount of shared data between versions of an index was also deemed important.

The Multiversion Hash Array Mapped Trie (MHAMT) was introduced, an extended version of the HAMT which is stored as Lumo objects in the Lumo storage manager, just like any other data in Lumo. MHAMT uses a concept of shared nodes, by leveraging the reference count mechanism provided by Lumo, to give nodes the ability to be shared between tries. If a trie wants to modify a shared node, a copy is created, but this is done effortlessly thanks to Lumo's ROW mechanism. Cloning an MHAMT is as simple as taking another reference to the root node of the original trie, making the root node into a shared node in the process. Diagrams depicting the before and after states of a simple MHAMT were presented, showing how certain operations on shared nodes are handled. A way of supporting non-unique keys and resolving hash collisions was also proposed, as well as a way of, for testing purposes, deducing and keeping track of which nodes a trie owns. It is worth mentioning that the MHAMT can behave as a true persistent data structure as long as its root node is always kept shared, but that the normal use case in a Lumo model is to only retain the versions which correspond to snapshots.

It was noted that the MHAMT implementation is very much dependent on Lumo, and the operations carried out by Lumo were discussed in an effort to illuminate what needs to be given extra thought if a more generic MHAMT implementation is to be constructed. These included operations such as hashing and equality checks, as

well as the logic around node storage and handling of shared nodes. Furthermore, the integration of MHAMT as an index structure in the existing Lumo database was discussed, and various index operations were described in detail.

The MHAMT implementation was proven to work reliably by testing it using BIM data from a real-world construction project. After storing the BIM data as a Lumo model, the resulting model consisted of 769 Lumo snapshots, each containing one MHAMT index. The total memory consumption of each MHAMT was analyzed, and the result was compared against a theoretical case where the tries were unable to share any nodes. A key observation was that the node sharing feature of the MHAMT decreased the total memory consumption with 90.7 % across the whole model, down to 83.9 MiB from the theoretical 898 MiB. Additionally, it was shown that each MHAMT only owned on average 6.0 % of its nodes, while the rest remained unmodified and could be shared from previous tries. It was noted however, that the memory footprint of a node could be decreased in several ways, for example by shrinking the size of the node header or by not storing the bitmap at all for full nodes.

Another aspect of the MHAMT examined using the test model was the resulting trie structures, which are dependent on the hashing algorithm used and the characteristics of the data. The storage level of values in the tries, across all snapshots, were analyzed. It was discovered that values were stored on much higher levels than needed, probably due to the hashing algorithm used not being as uniformly distributed with regards to the test data as it could have been. For instance, the MHAMT in the first snapshot would in theory have been able to store all values on level four, but in reality there were values stored as high as the ninth level. Not even half of the values were stored on level four or below in the first snapshot.

Regarding the trie structure, the node sizes per level were analyzed as well. This showed that 78 %–86 % of all nodes, depending on which snapshot was looked at, were at most half full. Additionally, only 4 %–7 % were full. As larger nodes are more cost effective than smaller nodes, this aspect of the trie structure was deemed undesirable.

The MHAMT implementation itself introduced in this thesis was considered a success, since it successfully satisfied all the performance criteria presented earlier. However, a few possible enhancements were presented, such as applying path compression to remove one-child nodes, and a new solution for handling duplicate keys. Making the nodes more compact in general, especially with regards to the node header, was proposed as well. Testing different cardinalities of MHAMT nodes to find the optimal one was also suggested. Lastly, an important feature missing entirely from the MHAMT implementation was determined to be the lack of support for storing ordered data. It remains to be seen if the MHAMT can be enhanced to support range queries, or if another multiversion index solution optimized for ordered data has to be used in Lumo alongside the MHAMT.

References

[1] Trimble Inc., *About Trimble: Transforming the way the world works*. [Online]. Available: https://www.trimble.com/en/our-company/about/overview (visited on 05/14/2024).

[2] L. Torvalds and Git contributors, *Git repository on GitHub*. [Online]. Available: https://github.com/git/git (visited on 05/14/2024).

[3] Trimle Inc., *Tekla Model Sharing*. [Online]. Available: https://www.tekla.com/products/tekla-model-sharing (visited on 05/14/2024).

[4] M. Atkinson, D. DeWitt, D. Maier, F. Bancilhon, K. Dittrich, and S. Zdonik, "The object-oriented database system manifesto", in *Deductive and Object-Oriented Databases*, W. Kim, J.-M. Nicolas, and S. Nishio, Eds., North-Holland, 1990, pp. 223–240. DOI: 10.1016/B978-0-444-88433-6.50020-4.

[5] GemTalk Systems, *GemStone/S 64 Bit Programmer's Guide*, 2020. [Online]. Available: https://downloads.gemtalksystems.com/docs/GemStone64/3.6.x/GS64-ProgGuide-3.6/GS64-ProgGuide-3.6.htm (visited on 05/14/2024).

[6] ObjectDB Software, *ObjectDB 2.8 Developer's Guide*. [Online]. Available: https://www.objectdb.com/java/jpa (visited on 05/14/2024).

[7] Emcripten contributors, *Emscripten, A complete Open Source compiler toolchain to WebAssembly*. [Online]. Available: https://emscripten.org/ (visited on 05/14/2024).

[8] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil, "A critique of ANSI SQL isolation levels", *SIGMOD Record*, vol. 24, no. 2, pp. 1–10, 1995. DOI: 10.1145/568271.223785.

[9] M. J. Cahill, "Serializable isolation for snapshot databases", Ph.D. dissertation, The University of Sidney, School of Information Technologies, 2009.

[10] Oracle, *MySQL 8.0 Reference Manual*, pages "17.7.2.3 Consistent Nonlocking Reads" and "19.1.2.5 Choosing a Method for Data Snapshots". [Online]. Available: https://dev.mysql.com/doc/refman/8.0/en/ (visited on 05/14/2024).

[11] The PostgreSQL Global Development Group, *PostgreSQL 15.1 Documentation*, pages "13.2. Transaction Isolation" and "26.2. File System Level Backup". [Online]. Available: https://www.postgresql.org/docs/15/ (visited on 05/14/2024).

[12] MongoDB Inc., *MongoDB Manual*, pages "Read Isolation, Consistency, and Recency" and "Read Concern "snapshot"". [Online]. Available: https://www.mongodb.com/docs/manual/ (visited on 05/14/2024).

[13] Dassault Systèmes SE, *NuoDB Documentation: Supported Transaction Isolation Levels*. [Online]. Available: https://doc.nuodb.com/nuodb/latest/sql-development/working-with-transactions/supported-transaction-isolation-levels/ (visited on 05/14/2024).

[14] A. Silberschantz, P. B. Galvin, and G. Gagne, "Operating system concepts", in Ninth. Wiley, 2013, pp. 408–409, ISBN: 978-1-118-06333-0.

[15] O. Rodeh, "B-trees, shadowing, and clones", *ACM Transactions on Storage*, vol. 3, no. 4, pp. 1–27, 2008. DOI: 10.1145/1326542.1326544.

[16] W. Xiao, Q. Yang, J. Ren, C. Xie, and H. Li, "Design and analysis of block-level snapshots for data protection and recovery", *IEEE Transactions on Computers*, vol. 58, no. 12, pp. 1615–1625, 2009. DOI: 10.1109/TC.2009.107.

[17] Microsoft, *SQL Server Technical Documentation: Database Snapshots (SQL Server)*, 2023. [Online]. Available: https://learn.microsoft.com/en-us/sql/relational-databases/databases/database-snapshots-sql-server (visited on 05/14/2024).

[18] T. Chien. "Snapshots are NOT backups, Comparing storage-based snapshot technologies with recovery manager (RMAN) and fast recovery area for Oracle databases". (2020), [Online]. Available: https://www.oracle.com/database/technologies/rman-fra-snapshot.html (visited on 05/14/2024).

[19] Oracle Corporation, *Oracle Solaris ZFS Administration Guide: ZFS Snapshots*. [Online]. Available: https://docs.oracle.com/cd/E19253-01/819-5461/gbcbn/index.html (visited on 05/14/2024).

[20] BTRFS contributors, *BTRFS Documentation: btrfs-subvolume*. [Online]. Available: https://btrfs.readthedocs.io/en/stable/Subvolumes.html (visited on 05/14/2024).

[21] D. Hitz, J. Lau, and M. Malcolm, "File system design for an NFS file server appliance", Network Appliance, Tech. Rep., 1994, Revision C 3/95.

[22] YugabyteDB Inc., *YugabyteDB Documentation: Point-in-time Recovery*. [Online]. Available: https://docs.yugabyte.com/stable/manage/backup-restore/point-in-time-recovery/ (visited on 05/14/2024).

[23] MariaDB Foundation, *MariaDB Server Documentation: Backup and Restore Overview*. [Online]. Available: https://mariadb.com/kb/en/backup-and-restore-overview/#filesystem-snapshots (visited on 05/14/2024).

[24] Dwayne Richard Hipp, *C-language interface specification for SQLite*. [Online]. Available: https://www.sqlite.org/capi3ref.html (visited on 05/14/2024).

[25] J. Dean and S. Ghemawat, *LevelDB repository on GitHub*. [Online]. Available: https://github.com/google/leveldb (visited on 05/14/2024).

[26] D. Stolee, "GitHub blog: Commits are snapshots, not diffs", 2020. [Online]. Available: https://github.blog/2020-12-17-commits-are-snapshots-not-diffs/ (visited on 05/14/2024).

[27] S. Chacon and B. Straub, *Pro Git*, Second. Apress, 2022, Version 2.1.359-2-g27002dd. [Online]. Available: https://git-scm.com/book/en/v2 (visited on 05/14/2024).

[28] DoltHub Inc., *Dolt Documentation*, pages "Dolt – Git", "Commands" and "Storage Engine". [Online]. Available: https://docs.dolthub.com/ (visited on 05/14/2024).

[29] PlanetScale Inc., *PlanetScale documentation*. [Online]. Available: https://planetscale.com/docs (visited on 05/14/2024).

[30] P. A. Bernstein and N. Goodman, "Multiversion concurrency control—theory and algorithms", *ACM Transactions on Database Systems*, vol. 8, no. 4, pp. 465–483, 1983. DOI: 10.1145/319996.319998.

[31] W. Cellary and G. Jomier, "Consistency of versions in object-oriented databases", in *Proceedings of the 16th VLDB Conference (VLBD '90)*, Morgan Kaufmann, 1990, pp. 432–441.

[32] R. Bayer and E. McCreight, "Organization and maintenance of large ordered indices", in *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control (SIGFIDET '70)*, Association for Computing Machinery, 1970, pp. 107–141, ISBN: 9781450379410. DOI: 10.1145/1734663.1734671.

[33] D. Comer, "Ubiquitous B-Tree", *ACM Computing Surveys*, vol. 11, no. 2, pp. 121–137, 1979. DOI: 10.1145/356770.356776.

[34] D. Lomet, R. Barga, M. Mokbel, and G. Shegalov, "Transaction time support inside a database engine", in *Proceedings of the 22nd International Conference on Data Engineering (ICDE '06)*, IEEE Computer Society, 2006, p. 35, ISBN: 0-7695-2570-9. DOI: 10.1109/icde.2006.162.

[35] K. Jouini and G. Jomier, "Indexing multiversion databases", in *Proceedings of the Sixteenth ACM Conference on Information and Knowledge Management (CIKM '07)*, Association for Computing Machinery, 2007, pp. 915–918. DOI: 10.1145/1321440.1321574.

[36] P. L. Lehman and S. B. Yao, "Efficient locking for concurrent operations on B-Trees", *ACM Transactions on Database Systems*, vol. 6, no. 4, pp. 650–670, 1981. DOI: 10.1145/93605.98744.

[37] T. Tzouramams, Y. Manolopoulos, and N. Lorentzos, "Overlapping B^+-trees: An implementation of a transaction time access method", *Data & Knowledge Engineering*, vol. 29, no. 3, pp. 381–404, 1999. DOI: 10.1016/S0169-023X(98)00046-9.

[38] L. Jiang, B. Salzberg, D. Lomet, and M. Barrena, "The BT-Tree: A branched and temporal access method", in *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB '00)*, Morgan Kaufmann, 2000, pp. 451–460.

[39] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer, "An asymptotically optimal multiversion B-tree", *The VLDB Journal*, vol. 5, no. 4, pp. 264–275, 1996. DOI: 10.1007/s007780050028.

[40] E. Soisalon-Soininen, *Maintaining and querying snapshots for transaction-time versioned databases*, Unpublished manuscript, 2020.

[41] S. Syrjänen, "Multiversion indexing in database systems", Master's thesis, Aalto University, School of Electrical Engineering, 2021.

[42] D. Lomet and B. Salzberg, "The performance of a multiversion access method", *SIGMOD Record*, vol. 19, no. 2, pp. 353–363, 1990. DOI: 10.1145/319628.319663.

[43] C. Riegger, T. Vinçon, and I. Petrov, "Multi-version indexing and modern hardware technologies: A survey of present indexing approaches", in *Proceedings of the 19th International Conference on Information Integration and Web-based Applications & Services (iiWAS '17)*, Association for Computing Machinery, 2017, pp. 266–275. DOI: 10.1145/3151759.3151779.

[44] L. J. Guibas and R. Sedgewick, "A dichromatic framework for balanced trees", in *Proceedings of the 19th Annual Symposium on Foundations of Computer Science (SFCS '78)*, IEEE Computer Society, 1978, pp. 8–21. DOI: 10.1109/SFCS.1978.3.

[45] R. Gottstein, R. Goyal, S. Hardock, I. Petrov, and A. Buchmann, "MV-IDX: Indexing in multi-version databases", in *Proceedings of the 18th International Database Engineering & Applications Symposium (IDEAS '14)*, Association for Computing Machinery, 2014, pp. 142–148. DOI: 10.1145/2628194.2628911.

[46] A. Boodman, *Noms repository on GitHub: Noms technical overview*. [Online]. Available: https://github.com/attic-labs/noms/blob/master/doc/intro.md (visited on 05/14/2024).

[47] T. Sehn, "DoltHub blog: Prolly Trees", 2024. [Online]. Available: https://www.dolthub.com/blog/2024-03-03-prolly-trees/ (visited on 05/14/2024).

[48] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (LSM-tree)", *Acta Informatica*, vol. 33, pp. 351–385, 1996. DOI: 10.1007/s002360050048.

[49] P. Muth, P. O'Neil, A. Pick, and G. Weikum, "The LHAM log-structured history data access method", *The VLDB Journal*, vol. 8, pp. 199–221, 2000. DOI: 10.1007/s007780050004.

[50] W. Cao *et al.*, "Timon: A timestamped event database for efficient telemetry data processing and analytics", in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*, Association for Computing Machinery, 2020, pp. 739–753. DOI: 10.1145/3318464.3386136.

[51] C. Riegger, T. Vinçon, and I. Petrov, "Indexing large updatable datasets in multi-version database management systems", in *Proceedings of the 23rd International Database Applications & Engineering Symposium (IDEAS '19)*, Association for Computing Machinery, 2019, pp. 1–5. DOI: 10.1145/3331076.3331118.

[52] Facebook Database Engineering Team, *RocksDB repository on GitHub*. [Online]. Available: https://github.com/facebook/rocksdb/ (visited on 05/14/2024).

[53] X. Zhao, K.-Y. Lam, C. Zhu, C.-Y. Chow, and T.-W. Kuo, "MVLevelDB: Using log-structured tree to support temporal queries in IoT", *IEEE Internet of Things Journal*, vol. 9, no. 10, pp. 7815–7825, 2021. DOI: 10.1109/JIOT.2021.3113994.

[54] R. de la Briandais, "File searching using variable length keys", in *Papers Presented at the March 3–5, 1959, Western Joint Computer Conference (IRE-AIEE-ACM '59)*, Association for Computing Machinery, 1959, pp. 295–298. DOI: 10.1145/1457838.1457895.

[55] E. Fredkin, "Trie memory", *Communications of the ACM*, vol. 3, no. 9, pp. 490–499, 1960. DOI: 10.1145/367390.367400.

[56] P. F. Windley, "Trees, forests and rearranging", *The Computer Journal*, vol. 3, no. 2, pp. 84–88, 1960. DOI: 10.1093/comjnl/3.2.84.

[57] J. Culberson and J. I. Munro, "Explaining the behaviour of binary search trees under prolonged updates: A model and simulations", *The Computer Journal*, vol. 31, no. 1, pp. 68–75, 1989. DOI: 10.1093/comjnl/32.1.68.

[58] J. L. Bentley and R. Sedgewick, "Fast algorithms for sorting and searching strings", in *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '97)*, Society for Industrial and Applied Mathematics, 1997, pp. 360–369. DOI: 10.5555/314161.314321.

[59] D. E. Knuth, "The art of computer programming, volume 3: Sorting and searching", in Second. Addison-Wesley, 1998, pp. 492–512, ISBN: 0-201-89685-0.

[60] D. R. Morrison, "PATRICIA – Practical Algorithm To Retrieve Information Coded in Alphanumeric", *Journal of the ACM*, vol. 15, no. 4, pp. 514–534, 1968. DOI: 10.1145/321479.321481.

[61] A. Andersson and S. Nilsson, "Improved behaviour of tries by adaptive branching", *Information Processing Letters*, vol. 46, no. 6, pp. 295–300, 1993. DOI: 10.1016/0020-0190(93)90068-K.

[62] S. Nilsson and G. Karlsson, "Fast address look-up for internet routers", in *Proceedings of the IFIP TC6/WG6.2 Fourth International Conference on Broadband Communications: The Future of Telecommunications (BC '98)*, Chapman & Hall, Ltd., 1998, pp. 11–22. DOI: 10.1007/978-0-387-35378-4_2.

[63] S. Nilsson and G. Karlsson, "IP-address lookup using LC-Tries", *IEEE Journal on Selected Areas in Communications*, vol. 17, no. 6, pp. 1083–1092, 1999. DOI: 10.1109/49.772439.

[64] S. Nilsson and M. Tikkanen, "Implementing a dynamic compressed trie", in *Proceedings of the 2nd International Workshop on Algorithm Engineering (WAE '98)*, Max-Planck-Institut für Informatik, 1998, pp. 25–36. DOI: 10.5555/314161.314321.

[65] Linux, the kernel development community, *The Linux kernel 6.6.0: LC-trie implementation notes*. [Online]. Available: https://docs.kernel.org/networking/fib_trie.html (visited on 05/14/2024).

[66] P. Bagwell, "Fast and space efficient trie searches", École Polytechnique Fédérale de Lausanne, Switzerland, 2000.

[67] P. Bagwell, "Ideal hash trees", École Polytechnique Fédérale de Lausanne, Switzerland, 2001.

[68] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan, "Making data structures persistent", in *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing (STOC '86)*, Association for Computing Machinery, 1986, pp. 109–121. DOI: 10.1145/12130.12142.

[69] R. Hickey, *Clojure repository on GitHub: PersistentHashMap.java*. [Online]. Available: https://github.com/clojure/clojure/blob/master/src/jvm/clojure/lang/PersistentHashMap.java (visited on 05/14/2024).

[70] EPFL and Lightbend Inc., *Scala repository on GitHub: HashMap.scala*. [Online]. Available: https://github.com/scala/scala/blob/2.13.x/src/library/scala/collection/immutable/HashMap.scala (visited on 05/14/2024).

[71] A. Prokopec, P. Bagwell, and M. Odersky, "Cache-aware lock-free concurrent hash tries", École Polytechnique Fédérale de Lausanne, Switzerland, 2011. DOI: 10.48550/arXiv.1709.06056.

[72] A. Prokopec, N. G. Bronson, P. Bagwell, and M. Odersky, "Concurrent tries with efficient non-blocking snapshots", in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '12*, vol. 47, Association for Computing Machinery, 2012, pp. 151–160. DOI: 10.1145/2370036.2145836.

[73] A. Prokopec, "Cache-Tries: Concurrent lock-free hash tries with constant-time operations", in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '18)*, Association for Computing Machinery, 2018, pp. 137–151. DOI: 10.1145/3178487.3178498.

[74] M. J. Steindorfer and J. J. Vinju, "Optimizing Hash-Array Mapped Tries for fast and lean immutable JVM collections", *ACM SIGPLAN Notices*, vol. 50, no. 10, pp. 783–800, 2015. DOI: 10.1145/2858965.2814312.

[75] M. J. Steindorfer, "Efficient immutable collections", Ph.D. dissertation, University of Amsterdam, 2017.

[76] G. Graefe, "Modern B-Tree techniques", *Foundations and Trends® in Databases*, vol. 3, no. 4, pp. 203–402, 2011. DOI: 10.1561/1900000028.

[77] Trimle Inc., *TrimBIM technology*. [Online]. Available: https://developer.tekla.com/documentation/trimbim-technology (visited on 05/14/2024).