UiT The Arctic University of Norway

Faculty of Science and Technology
Department of Computer Science

Reducing Memory Overhead in CRDT for Collaborative Editing

Andrei Ivanov

INF-3990: Master's thesis in Computer Science - May 2024

Supervisors

Main supervisor: Weihai Yu Department of Computer Science

Abstract

For peer-to-peer collaborative editing systems utilizing the Conflict-free Replicated Data Type memory consumption can become a significant issue, particularly when handling large files with extensive editing histories. It can lead to performance problems and hinder user productivity, especially in environments with limited resources. This thesis addresses the problem of excessive memory usage and proposes a novel solution to mitigate this issue by applying partial persistence to CRDT data structure. Through a series of experiments and evaluations, the effectiveness of the proposed approach is demonstrated. This research contributes to the advancement of memory-efficient collaborative systems, offering potential benefits for users working with large documents in resource-constrained environments.

Contents

List of Figures

1 Introduction

1.1 Context

Collaborative editing systems have become indispensable tools for facilitating real-time collaboration among users across diverse domains, including document editing, software development, project management, and online collaboration platforms. These systems enable multiple users to work together on shared documents, projects, or content concurrently, allowing for seamless communication, coordination, and knowledge sharing.

Traditional approaches to collaborative editing, such as Operational Transformation (OT), have been widely adopted to ensure consistency and concurrency control in distributed environments. OT-based systems maintain a central server that coordinates user edits by transforming and applying operations. While effective in many scenarios, OT-based systems reliance on central server introduces a single point of failure problem, hindering availability and reliability of collaborative editing environments.

In recent years, Conflict-free Replicated Data Types (CRDTs) have emerged as a promising alternative to traditional approaches for achieving eventual consistency in collaborative editing systems. CRDTs offer a decentralized approach to concurrency control, allowing replicas to independently update their local state without coordination from a central server. By leveraging mathematical properties and commutative operations, CRDTs ensure that replicas eventually converge to a consistent state, even if the presence of concurrent updates and

network partitions.

This thesis focuses on addressing a specific challenge in CRDTs, particularly concerning the ever-growth of local memory as data within the model continues to expand. As collaborative editing sessions progress and more updates are applied to the data model, the size of local memory grows even when edited text is deleted. This growth can lead to numerous problems, posing significant challenges for CRDT-based systems in distributed environments.

1.2 Problem Definition

Conflict-free Replicated Data Types offer a decentralized approach to achieving eventual consistency in distributed systems, making them well-suited for collaborative editing applications. However, CRDT-based systems face specific challenges and limitations related to memory management, particularly concerning the continuous growth of data updates and the resulting increase in memory consumption.

As collaborative editing sessions progress and more updates are applied to the data model, the size of the local memory in each node of the CRDT-based system grows continuously. This accumulation of historical data includes the complete history of updates, operations, and metadata necessary to ensure eventual consistency across distributed replicas. The continuous growth of data updates introduces several challenges, such as memory exhaustion, performance degradation and scalability constraints.

To address those challenges, there is a need for effective memory management solution that optimizes memory usage, prevents memory exhaustion, and ensures efficient utilization of resources.

1.3 Goals

This project is centered around the resolution of a memory ever-growth problem in CRDT-based systems and achieving enhanced efficiency and scalability in collaborative editing applications.

It involves modifying an already implemented CRDT by applying partial persistence to it. This approach aims to address the challenge of memory ever-growth by selectively offloading data from local memory to disk storage. By introducing partial persistence of data updates, this CRDT variant will optimize memory

usage and mitigate memory exhaustion, thus enhancing the scalability and practical deployment of CRDT-based solution in collaborative editing applications.

1.4 Methodology

The methodology employed a combination of research, design, development, and testing phases to iteratively refine the solution.

The approach is centered around the concept of partial persistence, which involves offloading less relevant data from local memory to disk storage while maintaining the integrity and functionality of the collaborative editing system.

The design and implementation processes followed an iterative development approach, consisting of multiple iterations focused on specific aspects of the solution. Each iteration involved analysis and planning to assess the current state of the solution, identify areas for improvement, and define objectives and strategies for implementation.

Throughout the implementation process, collaboration and feedback played a crucial role in guiding decision-making and driving improvements. Regular discussions with project supervisor provided valuable insights, suggestions, and critiques that informed the direction and refinement of the solution.

2

Technical Background

2.1 Collaborative Editing Systems

Collaborative editing systems have progressed the way individuals collaborate and work together on shared documents, projects, and content. From simple text editors to complex collaborative platforms, these systems have evolved significantly over the years, leveraging innovative technologies and architectures to facilitate seamless collaboration among users.

2.1.1 Evolution of Collaborative Editing Systems

Collaborative editing systems have undergone a remarkable evolution, driven by advancements in network technologies, distributed systems, and user interface design. This evolution can be broadly categorized into the following phases:

- **Early Collaborative Text Editors.** The early days of collaborative editing witnessed the emergence of simple text editors that allowed multiple users to edit a shared document concurrently. The very first computer system to implement real-time collaborative editing, NLS (oN-Line System), was implemented by a team of researchers at Stanford Research Institute in 1962, led by Douglas Engelbart. [1] Other examples include the SubEthaEdit [30], DocSynch [20] and Gobby [24].

- **Web-based Collaboration Platforms.** With the advent of the internet and web technologies, collaborative editing systems transitioned to web-based platforms, offering users the ability to collaborate on documents in real time through web browsers. Examples include Google Docs[19], Microsoft Office Online [27], and Dropbox Paper [28].

- **Cloud-based Collaboration Tools.** The progress in cloud computing and storage technologies led to the emergence of cloud-based collaboration tools, which provided users with seamless access to shared documents from any device or location. Examples include Microsoft SharePoint [29], Box [16], and Zoho WorkDrive [31].

2.1.2 Characteristics of Collaborative Editing Systems

Collaborative editing systems are designed to enable multiple users to collaborate on shared documents in real time. To fulfill this purpose effectively, these systems must possess certain characteristics and meet specific requirements [2], such as:

- **Conflict Resolution.** Conflict resolution mechanisms are essential for handling concurrent edits made by multiple users to the same document. In collaborative editing systems, conflicts may arise when users edit the same portion of a document simultaneously or when network issues result in out-of-order updates. Effective conflict resolution ensures that conflicting edits are resolved in a consistent manner, preserving the integrity and coherence of the shared document.

- **Support for Concurrent Users.** Collaborative editing systems must be capable of supporting concurrent users collaborating on the same document simultaneously. This requires scalable architectures and efficient data structures to handle the coordination and synchronization of edits across multiple users. Additionally, collaborative editing systems must provide robust user management and access control features to regulate user permissions and ensure that only authorized users can edit or access sensitive content.

- **Versioning and History Tracking.** Versioning and history tracking capabilities are required for documenting and managing changes made to shared documents over time. Collaborative editing systems typically maintain a version history of the document, allowing users to review past revisions, revert to previous versions, and track the evolution of the content. Versioning and history tracking features provide accountability and traceability, enabling users to understand the context and chronology

of edits made by themselves and other collaborators.

2.1.3 Architectures of Collaborative Editing Systems

Collaborative editing systems employ two fundamental architectural paradigms to support real-time collaboration among users: the client-server architecture and the peer-to-peer (P2P) architecture:

- **Client-Server Architecture.** In a client-server architecture, clients interact with a central server to access and edit shared documents. Example include Google Docs and Microsoft Office Online.

- **P2P Architecture.** Peer-to-peer architectures distribute documents editing responsibilities among multiple nodes in a decentralized manner, allowing users to collaborate directly with each other without relying on a central server. Examples include Conclave [17] and Hyperpad [25].

2.2 Conflict-free Replicated Data Types

Conflict-free Replicated Data Types represent a fundamental advancement in distributed systems, providing a solution for achieving eventual consistency in replicated data across distributed environments. At their core, CRDTs embody principles and concepts that enable concurrent updates to data without the need for synchronization or coordination between replicas, thereby resolving conflicts seamlessly.

In order to understand the workings of CRDTs, it is essential to grasp the underlying principles that manage their design and operation. These principles serve as the guiding rules for achieving eventual consistency in distributed systems [6]:

- **Commutativity.** When merging the state of two replicas, A and B, it is essential that merging A into B produces the same result as merging B into A. This property ensures that the order of merge operations does not affect the final state of the data, enabling replicas to converge towards a consistent state regardless of the order in which updates are applied.

- **Idempotence.** In scenarios where network issues or message duplication occur, it is crucial that applying a value multiple times has the same effect as applying it once. This property ensures that replicas can safely apply updates without fear of introducing inconsistencies or diverging from

the intended state, even in the presence of network anomalies, message duplication or in cases when the same update comes from different replicas.

- **Associativity.** Regardless of the order in which updates are applied to replicas, the final state of the data should remain consistent. Whether merging instances A and B first and then merging the result with instance C, or vice versa, the final state should be identical. This property guarantees that replicas can collaborate with different replicas.

2.2.1 CRDTs Classification

Initially, CRDTs were categorized in two classes [4]: state-based CRDTs and operation-based CRDTs. Each approach offers distinct advantages and trade-offs, applicable to different use cases and scenarios in distributed systems. Delta-CRDTs were introduced in 2015 with the aim of reducing the footprint of state-based counter CRDTs while retaining the majority of their benefits [7].

2.2.1.1 State-based CRDTs

State-based CRDTs represent shared data as a complete state that is replicated across all replicas in the system. Each replica maintains its local copy of the shared state and periodically sends its state to other replicas. When receiving state updates from other replicas, a replica merges these updates with its local state to achieve convergence.

However, this approach can lead to inflation over time, as metadata accumulates to ensure convergence during synchronization.

Despite the potential for increased storage overhead, state-based CRDTs offer robust eventual consistency guarantees. They are tolerant of duplicate messages, as redundant updates can be detected and discarded during the merge process.

The visual representation of a state-based Counter CRDT that uses *max* merge function can be seen in Figure 2.1. In this figure two peers, *a* and *b*, modify and propagate the state. The state contains a map that stores a counter for each peer, only the origin peer can modify the corresponding counter in map. It can be seen that the last state update by the peer *b* is propagated twice. That is the example of message duplication, which is being resolved by the merge function that chooses the counter with the highest value (counter is not intended to be

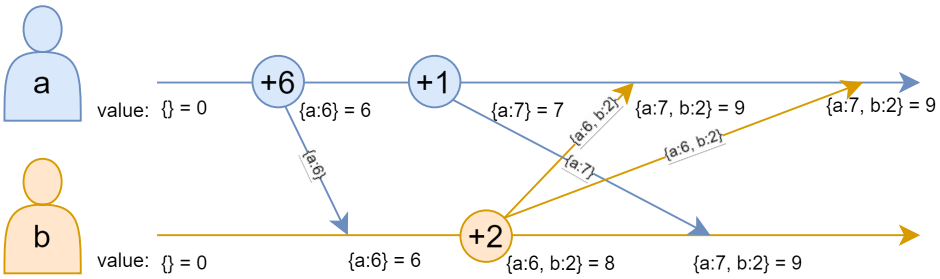

Figure 2.1: An example of state-based Counter CRDT.

decreased in this approach).

2.2.1.2 Operation-based CRDTs

Operation-based CRDTs represent updates to shared data as a sequence of operations. Each replica applies these operations locally and propagates them to other replicas. By ensuring that operations commute with each other, operation-based CRDTs guarantee eventual convergence across replicas.

This approach can also suffer from inflation, as well as state-based approach, due to the accumulation of operation logs, but it provides efficient scalability by minimizing the amount of metadata stored at each replica.

Message duplication can pose challenges to maintaining consistency and correctness in operation-based CRDTs. When replicas receive duplicate messages containing the same operation, they must ensure that applying the operation multiple times does not lead to unintended changes or inconsistencies in the shared data.

Figure 2.2 demonstrates an example of operation-based Counter CRDT. In this figure, two peers perform operations and propagate them instead of sending the whole state.

2.2.1.3 Delta State CRDT

Delta State CRDTs represent a variation within the state-based CRDT family that focuses on minimizing communication overhead by transmitting only the changes (deltas) made to the replicated data structure. This approach is particularly beneficial in scenarios where the size of the data structure is large and transmitting the entire state would be inefficient.

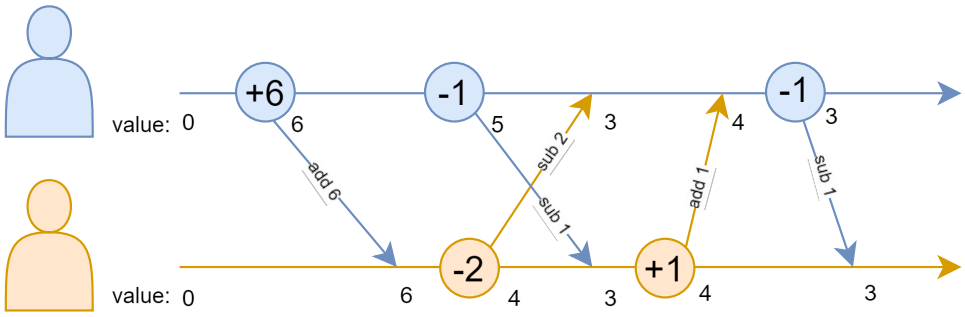

Figure 2.2: An example of operation-based Counter CRDT.

In Delta State CRDTs, each replica maintains its local copy of the data structure and tracks the changes made to it over time. Instead of propagating entire states or individual operations, replicas exchange deltas, which represent the specific modifications made to the data structure since the last synchronization.

2.2.2 Types of CRDTs

In CRDT design, various data types could be implemented using state-based, operation-based or delta-state approaches. The most common are Counter, Set and List CRDTs [10].

2.2.2.1 Counter CRDTs

Counter CRDTs represents a counter that can be incremented or decremented by individual processes without coordination. In a state-based approach. each replica maintains a local counter value, and changes are propagated by merging the values during synchronization. Operation-based implementations track increments and decrements as individual operations, which are then propagated and applied to replicas.

2.2.2.2 Set CRDTs

Set CRDTs represents a set of unique elements, supporting addition and removal operations. In a state-based approach, each replica maintains its local copy of the set, and changes are merged during synchronization to ensure convergence. Operation-based implementations represent additions and removals as individual operations, allowing replicas to apply these changes independently and achieve eventual consistency.

2.2.2.3 List CRDTs

List CRDTs represents an ordered list of elements, supporting insertion, deletion, and reordering operations. State-based implementations maintain a copy of the list at each replica, and changes are merged to synchronize replicas. Operation-based approaches track individual insertions, deletions, and moves as operations, enabling replicas to independently apply changes and converge over time.

2.2.3 Applications of CRDTs

CRDTs are being used in diverse applications across various domains, offering robust solutions for achieving eventual consistency and seamless replication of data in distributed environments. Some of the key applications where CRDTs are utilized are presented below:

- **Distributed Databases.** CRDTs play a crucial role in some distributed database systems, enabling efficient replication and synchronization of data across multiple nodes. By leveraging CRDTs, distributed databases can achieve multi-master replication [9] and support offline operation [12, 13], ensuring data availability and integrity in diverse and dynamic environments.

- **Collaborative Editing.** CRDTs are extensively used to support real-time collaborative editing of documents or shared content in collaborative software applications [14, 15]. Whether it's collaborative document editing, shared whiteboards, or collaborative code editing, CRDTs enable multiple users to make concurrent updates without conflicts, facilitating seamless collaboration and content creation [3].

2.3 Memory Size in CRDTs

One significant challenge faced by CRDT-based systems is the continuous growth of local memory size as updates are performed on the shared data model [11]. This issue arises due to the nature of CRDTs, which state updates are inflationary to ensure eventual consistency across distributed replicas. While this approach guarantees data integrity and convergence, it also results in the ever-growth of data state within the local memory of each node in the system.

This kind of problem in CRDT systems often arises due to the accumulation of

tombstones. Tombstones are metadata markers used to indicate the deletion of elements in the data structure [5]. When a user deletes a portion of text, a tombstone is created to record this deletion. Tombstones are typically retained indefinitely to ensure that all replicas converge to the same state.

As users continue to edit the document, more tombstones accumulate in the system. They consume memory resources, even though they represent deleted elements and do not contribute to the visible content of the document. Over time, especially in scenarios with frequent edits or long-lived documents, the memory occupied by tombstones can become significant and lead to memory ever-growth issues.

The accumulation of tombstones within the local memory of each node poses several challenges, such as:

- **Memory Exhaustion.** As updates to the data model accumulate over time, the size of the local memory grows continuously. This growth can lead to memory exhaustion, especially in systems with limited memory resources or high update rates.

- **Performance Degradation.** The increasing size of local memory can impact system performance, resulting in longer response times for numerous types of operation. Retrieving, processing and navigating through large volumes of data may introduce latency and overhead, affecting the overall responsiveness of the system.

- **Scalability Constraints.** The accumulation of historical data may impose scalability constraints on the CRDT-based system, limiting its ability to support large datasets or accommodate a growing number of concurrent users. Scaling the system to handle increasing data volumes while maintaining performance and reliability becomes a significant challenge.

2.4 Overview of the Wyde Collaborative Editing System

Wyde is a collaborative editing subsystem integrated into Emacs, designed to facilitate real-time collaborative editing with the usage of delta-based CRDT. The key feature of Wyde is a support of selective undo/redo operations with dynamic and appropriate granularity. This feature faces the previously stated problem of memory ever-growth, as it requires permanent remain of tombstones, the data that was marked as deleted, and prevents applying of the

solution that involves tombstone garbage collection.

In this section, an overview of the internal organization of the Wyde system is provided, which is pivotal for understanding the following description of interactions with Wyde's data model. A more detailed description of Wyde's architecture and functionalities can be found in the original paper [8].

In Wyde, documents undergo concurrent updates from multiple peers across different sites. Each peer comprises essential components including a view of the document, a model representing the document's structure, a log maintaining the operation history, and several queues for managing incoming operations.

The view represents the document as a string of characters. Users can insert or delete substrings at specific positions within the view and undo earlier operations selected from the log.

Peers concurrently receive two types of operations: local operations generated by the user and remote operations sent from other peers. Local operations immediately affect the peer's view of the document. Local operations and received remote operations, are stored in queues for later processing. During synchronization cycles, queued operations are integrated into the model, with the resulting changes reflected in the view. Integrated operations are being logged. Throughout the editing session, the operations log is continuously saved and updated in a JSON file for the purpose of model restoration upon opening the file.

Unique identifiers play a crucial role in Wyde's operation tracking. Each peer is assigned a unique identifier known as the peer identifier *pid*. Operations originating from a peer are tagged with an update counter known as the log counter *lc*, which is incremented with each integrated local operation. Additionally, each editing session is assigned a unique session identifier *sid*, enabling the unique identification of operations using a tuple, combined of those properties (*sid, pid, lc*).

The model in Wyde serves as a representation of editing operations and their relationships. It is structured as layers of interconnected nodes, each node encapsulating a set of characters. Conceptually, characters within the model possess unique identifiers that are fully ordered.

Nodes within the model are organized into layers, with nodes at the lowest layer representing insertions and containing the inserted characters. Conversely, nodes at higher layers signify deletions, with each higher-layer node (outer node) effectively removing characters contained within lower-layer nodes (in-

ner nodes).

Every node in the model contains two essential identifiers: *cid-l* denotes the identifier of the leftmost character within the node, while *cid-r* represents the identifier of the rightmost character. Identifiers for characters situated between the edges of the node are not stored in the model. These identifiers play an important role, since they determine the place of the node within the model. The presence of two nodes with the same *cid-l* or *cid-r* will provoke a conflict, so all of the nodes that were added in the model, must remain, which makes it necessary to keep deleted nodes in the local memory as tombstones. Additionally, insertion nodes include a string, denoted as *str*, which contains the characters inserted by the corresponding operation.

Subsequent operations within the Wyde model may result in the splitting of existing nodes. Nodes that originate from the same operation share a common *op* element, which serves as the descriptor for the operation. This descriptor encompasses essential information such as the identifier and type of the operation, along with sets denoted as *P* (for parents) and *C* (for children).

The *P* set contains references to the descriptors of the operation's ground operations, while the *C* set consists of operations that are built upon the current operation. A ground operation *op-g* of a given operation *op* is one whose existence relies on the existence of *op*. This parent-child relationship between operations establishes dependencies among them.

Additionally, the descriptor includes an undo element that contains a set *U* of identifiers for its undo operations. This set may contain multiple identifiers, reflecting scenarios where multiple peers concurrently undo the same operation. Furthermore, an undo element may itself possess its own undo element, especially when the original operation is redone. Consequently, the undo elements of an operation form a chain, and the operation is effectively undone if the length of the chain is an odd number.

In the Wyde model, an insertion operation is considered self-visible if it has not been effectively undone, whereas a deletion operation is self-visible if it has been effectively undone. An operation is deemed visible if it is self-visible, and all its ground operations are visible. Similarly, a character within the model is considered visible if all operations affecting it are visible.

The Wyde model incorporates three types of links among nodes:

- *l-r* links maintain the left-right character order,
- *opl-opr* links connect nodes belonging to the same operation,

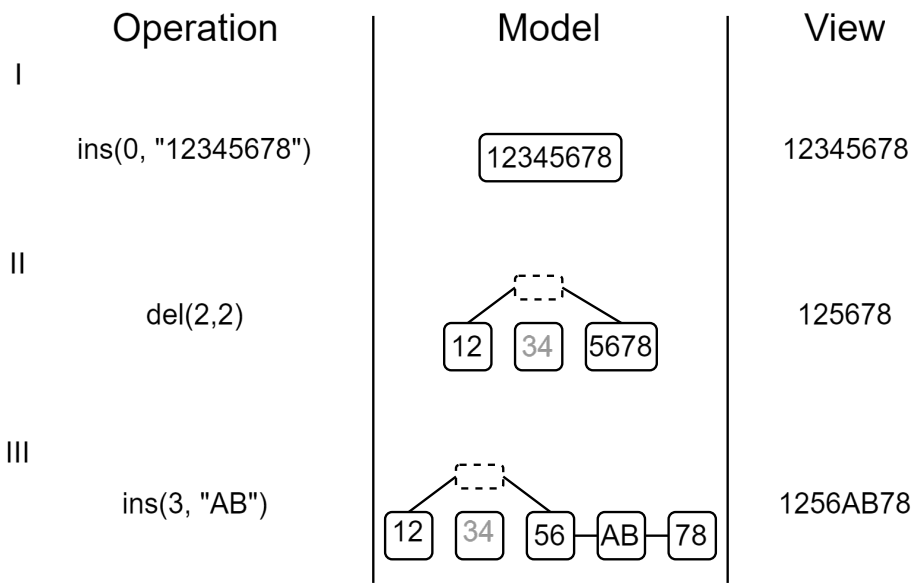

Figure 2.3: An example of model updates in Wyde.

- *i-o* links maintain inner-outer relations among nodes.

The outermost nodes and nodes within the same outer node are interconnected via *l-r* links. During synchronization between the view and the model, the view mirrors the concatenation of all visible characters from the outermost nodes through the *l-r* links.

The model utilizes two hashtables, designed to efficiently store information about nodes and operations within the system, called *nodes* and *ops*. These hashtables are used for fast access to specific entities based on their unique identifiers. The node's identifier is composed of a concatenated tuple, combining node's *cid-l* with node's operation identifier. Upon passing an entity's unique identifier, the data structure is returned from the corresponding hashtable. This architectural design enables Wyde to rapidly access nodes and operations on demand.

Figure 2.3 shows an example of performing updates to the Wyde model. Rectangles with solid border and black text represent visible insertion nodes. A rectangle with dotted border is a deletion node. It contains an invisible underlying insertion node with a light gray text.

3

Partial Persistence Approach

3.1 Hypothesis and Expectations

The hypothesis for addressing the identified problem resolves around the offloading of certain portions of the local memory to disk storage. Specifically, the focus is on relocating nodes linked to deletion operations, as these nodes often contain significant amounts of data that may no longer be relevant or actively used. By transferring these less relevant nodes to disk storage, it is anticipated that the memory consumption of the local system can be efficiently reduced, thereby mitigating memory exhaustion and enhancing system performance. This would make the CRDT partially persistent, as nodes on the disk are planned to stay immutable.

The primary expectation is to achieve a significant reduction in memory storage utilization, proportional to the amount of memory consumed by the nodes relocated to disk storage, It is anticipated that the memory savings will be nearly equivalent to the memory footprint of the nodes removed from the local memory.

However, in order to be able to access the nodes offloaded to the database, the creation of special structures for referencing the nodes stored on disk is required. Although those structures may consume some amount of local memory, it is

expected to be substantially smaller compared to the removed nodes, with each pointer potentially replacing dozens or even hundreds of nodes. As a result, the overall impact on memory consumption is expected to be minimal, with the benefits of memory optimization outweighing any potential increase in memory overhead due to pointers creation.

Additionally, expectations include the potential optimization of performance in terms of navigation across the model. If a chain of multiple nodes is replaced by a pointer referencing a region of nodes on disk, it is expected to take less time to navigate from the first node to the last node of the model.

3.2 Architecture of the Partially Persistent CRDT

The architecture of the solution revolves around the concept of offloading memory to the disk storage for nodes in the upper layers of the model, specifically targeting nodes that are less likely to be frequently accessed in the future. Nodes that are linked with deletion operations match this criterion. To maintain the integrity of the model, a pointer replaces the deleted node or the region of deleted nodes, occupying their place in the model. These pointers are referred to as proxy nodes.

A visual representation of the proposed solution is depicted in Figure 3.1. This illustration showcases the model's state both before and after the conversion of a deletion node and its associated inner nodes into a proxy node. Essential information for the converted nodes, necessary for their restoration in local memory, is stored in the database. The proxy node, denoted as P and depicted as a circle, contains a unique identifier of the offloaded node d, enabling it's retrieval from the database on demand.

Proxy nodes serve as placeholders for the deleted nodes, facilitating efficient retrieval when needed. They also offer a navigation performance enhancement by allowing the referencing of a sequence of linked nodes through a single proxy node. Each proxy node maintains information about the leftmost and rightmost nodes of the deleted region. In cases where the proxy node refers to a single node, these values are identical. Figure 3.2 demonstrates the conversion of a region of proxy nodes.

The solution includes the capability to convert any node in the model to a proxy node. This conversion process involves saving all relevant information about the node, including its inner nodes and linked operations, to the database. Subsequently, the node is removed from the local memory.

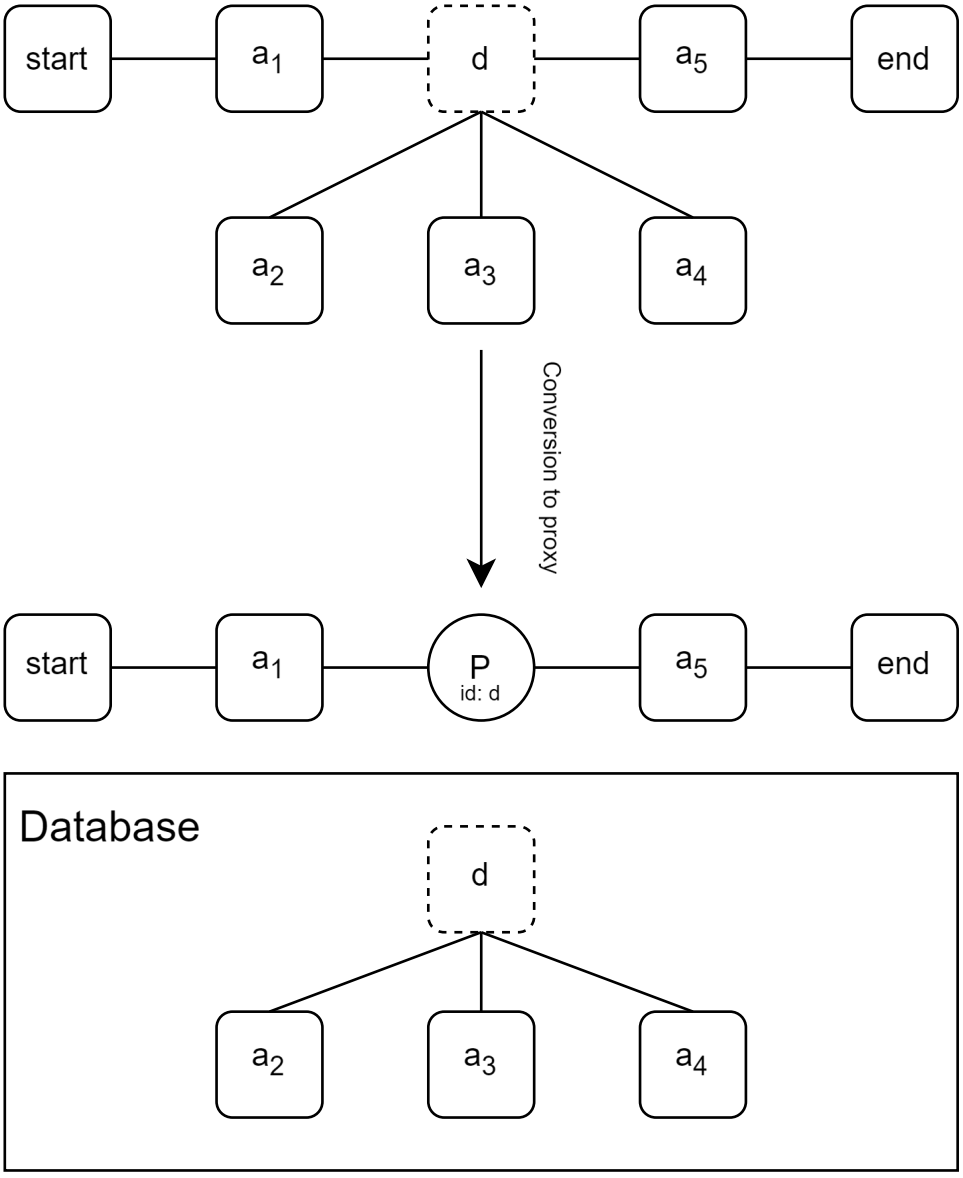

Figure 3.1: A visual representation of the node-to-proxy conversion.

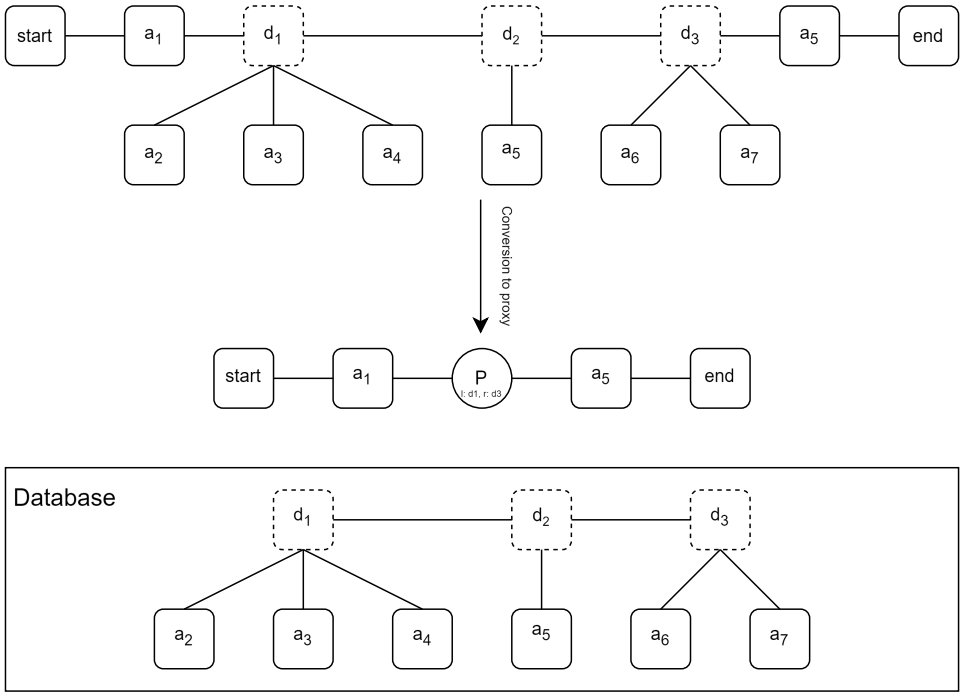

Figure 3.2: A visual representation of the region-to-proxy conversion.

In case of a need for accessing nodes that have been converted to proxy nodes (for example in case of undo operation or the remote operation that involves the converted node), the proxy node can be converted back, leading to the restoration of all previously deleted nodes to the model. This process ensures that the model remains intact and accessible, even after nodes have been offloaded to the database.

3.3 Early Development and Exploration

Before beginning the development of the solution, extensive preliminary preparations were performed. This section provides an overview of the early development stages and exploration activities conducted in preparation for the project.

3.3.1 Learning Emacs Lisp

As a prerequisite for working on the project, a comprehensive understanding of Emacs Lisp was essential. Emacs Lisp serves as the primary programming

language for implementing functionalities within the Wyde editor, the target platform for integrating the Partial Persistence Approach. To acquire proficiency in Emacs Lisp, dedicated efforts were made to study the language's syntax, semantics, and best practices. During learning process various knowlege resources, tutorials, and official documentation were utilized to grasp the fundamentals of Emacs Lisp programming [18, 21].

3.3.2 Studying the Wyde Editor Codebase

Following the acquisition of foundational knowledge in Emacs Lisp, the next phase involved a thorough examination of the Wyde editor's codebase. The Wyde editor serves as the host environment for implementing the solution, making it necessary to gain a comprehensive understanding of its architecture, design principles, and implementation details. The study of the Wyde editor encompassed various aspects, including:

- Analysis of existing data structures and algorithms used for the interactions inside the data model;
- Understanding the mechanisms for managing memory, handling data synchronization, and ensuring consistency in distributed environments.

During the exploration phase time was allocated to thoroughly understand the codebase, identifying relevant components, and getting familiar with the complexities of the existing implementation.

3.4 Development Iterations

The development iterations represent a series of steps taken to address challenges, refine strategies, and achieve the goals outlined in the project.

These iterations reflect an iterative and incremental approach to software development, where each phase builds upon the insights gained from the previous one. Throughout this process, various ideas were explored, tested, and polished to enhance the functionality and efficiency of the solution.

The following subsections outline each iteration's objectives, the development process employed, key findings, and plans for subsequent iterations. Together, they form a comprehensive narrative of the development process, revealing the challenges faced, solutions devised, and lessons learned along the way.

3.4.1 Initial Design Exploration

The initial iteration of development involved exploring a design concept aimed at significantly reducing local memory consumption by migrating the entire data model to the database. The objective was to implement a solution where all nodes in the model would be stored in the database, thus offloading them from local memory. During this iteration, the focus was on implementing functionality for saving the entire data model to the database and modifying nodes in response to changes in the model.

During this iteration, the following steps were taken:

- A set of tools for interaction with database was developed;
- Implemented functionality for saving all nodes in the model to the database;
- Developed mechanisms for modifying nodes in the database to reflect changes in the model.

After analysing the results of this iteration, the approach of migrating the entire data model to the database was considered highly risky due to the extensive codebase modifications required. Adapting all interactions within the model to work with the database instead of local memory posed significant challenges and potential problems. However, the development process resulted in the creation of a robust set of tools for database interaction, laying the foundation for future iterations.

Based on the analysis of the results from the initial iteration, a safer approach was designed to address the encountered challenges. This approach, as described in the Section 3.1, focuses on selectively offloading nodes to disk storage while leveraging proxy nodes for memory optimization.

3.4.2 Proxy Nodes Development

The second iteration focused on the development of proxy nodes functionality. The primary objectives were to implement the conversion of regular nodes to proxy nodes and vice versa.

During the development process, the following results were obtained:

- Implemented functionality for converting regular nodes to proxy nodes by saving regular nodes to the database and creating corresponding

proxy nodes;

- Developed mechanism for converting proxy nodes back to regular nodes by loading nodes from the database and inserting them in place of proxy nodes;

- Integrated proxy nodes into the Wyde model to ensure seamless interaction and compatibility;

- Developed a set of tools for making memory measurements to assess the impact of proxy nodes on memory usage.

Memory measurements conducted after the development revealed that the conversion of regular nodes to proxy nodes resulted in a reduction in memory usage. However, the observed reduction was not as extensive as initially expected. Further investigation suggested that information about operations related to nodes could still remain in the local memory, contributing to memory overhead. As a result, the need to address the storage of operations in local memory was identified as a priority for the next iteration.

3.4.3 Removal of Operations from Local Memory

The third iteration focused on addressing the issue of memory overhead caused by operations linked to deleted nodes remaining in local memory. The primary objective was to attempt to locate and remove all references to these operations, enabling them to be garbage-collected and freeing up memory.

However, these attempts proved unsuccessful, as the operations continued to be stored in memory due to the complexity and extensive nature of their references throughout the codebase.

As a result of this iteration, a further investigation was carried out, involving the closure and reopening of the file to avoid loading operations linked to nodes contained in the database. This approach is aimed to prevent unnecessary loading of operations into local memory, thus reducing memory usage.

3.4.4 Model Restoration Process Modification

The focus of the fourth iteration was on enhancing the model restoration process to reduce memory consumption of the restored file. The goal was to avoid loading irrelevant data about nodes and operations linked to proxy nodes into local memory.

In the original restoration process, the log file, associated with the file that is being restored, is read line by line, with each line containing information about a specific operation. Each read operation is then added to an operations queue. Once the entire file is read, the operations in the queue are applied to the model, reconstructing the state of the model at the end of the previous editing session.

The modified restoration process introduces a check for each operation read from the log file. If an operation is found to be contained in the database and has a relation in the table of proxy-node relations, it is skipped from being added to the operations queue. Instead, the proxy node associated with this operation is loaded from the database and directly added to the model.

This modification effectively reduced memory consumption during the restoration process by avoiding the loading of unnecessary data into local memory. That, to some degree, confirmed the suspicion that some offloaded data was not completely garbage-collected. Additionally, it leads to an increase in the speed of file restoration, as only relevant data is processed and added to the model.

The results of this iteration have shown the reduce of memory consumption of the opened file and demonstrated the increased speed of model restoration. A more detailed description of the results, including the measurements, is presented in Chapter 5.

4

Implementation

4.1 Database Interaction

4.1.1 Database Schema

The database schema for the solution is structured to manage various aspects of the data model, including nodes, operations, and proxy nodes. Each table captures specific information essential for the functionality and integrity of the system. The database schema can be seen in the Figure 4.1.

Further details regarding the utilization of the database will be presented in the following sections. It is important to notice that the work on offloading operations to the database was also performed, although its implementation details are not included in this thesis as it was not utilized in the present study. Any operations-related manipulations relevant to this research are considered as an area for future work.

4.1.2 Database Connectivity and Operations

Emacs Lisp provides support for interaction with SQLite databases through built-in *sqlite.el* library. This section describes how it was utilized, covering basic operations such as connecting to a database and executing SQL queries.

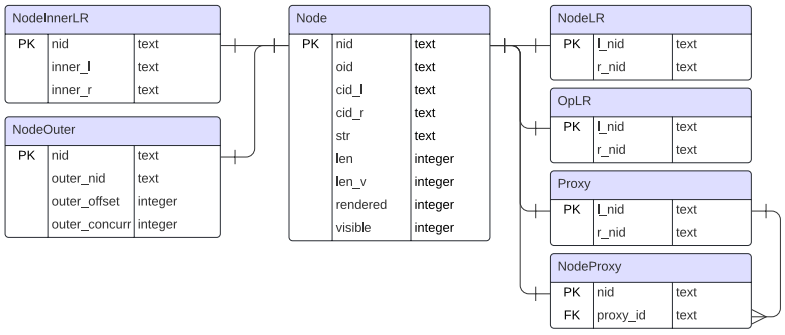

Figure 4.1: Database schema.

4.1.2.1 Establishing Database Connection

To begin working with an SQLite database, the first step is to establish a connection to it. The *wyde-db-connect* function serves for this purpose. It accepts a single argument *db-path*, denoting the path to the SQLite database file. The connection establishment process is described below:

1. Upon function invocation, the existence of the specified database file is being verified.

2. If the file exists, it proceeds to establish a connection to the existing database using *sqlite-open*.

3. In the event of a missing database file, the creation of a new database at the specified path is initiated.

4. Additionally, it applies the schema to the newly created database to ensure compliance with the data model.

Upon successful execution, *wyde-db-connect*, demonstrated in Listing 4.1 (code listings for other implemented functions will not be included in the thesis due

to their extensive length, which would occupy excessive space on a single page), yields a database connection object, enabling interaction with the database withing Emacs Lisp. In order to close the established connection, *sqlite-close* function can be called.

Listing 4.1: Function for establishing database connection

```
(defun wyde-db-connect (dbpath)
  (if (file-exists-p dbpath)
      (sqlite-open dbpath)
    ;;ELSE
    (let ((schemapath (concat wyde-dir "db/
       wyde-db-schema.sql")))
      (when (file-exists-p schemapath)
        (let ((db (sqlite-open dbpath)))
          (with-temp-buffer
            (insert-file-contents schemapath)
            (sqlite-transaction db)
            (dolist (element (split-string (
               buffer-string) ";"))
              (setq query (string-replace "\n" ""
                 element))
              (when (not (equal "" query))
                (sqlite-execute db query)))
            (sqlite-commit db))
          db)))))
```

4.1.2.2 Database Interaction Functions

The Emacs Lisp does not provide any built-in ORM tools and is limited to basic querying of the database, so three database interaction functions have been developed to facilitate communication between Emacs Lisp and SQLite database. They encapsulate such common database operations as insertion, updating and selection.

- ***wyde-db-insert-row*.** This function is designed to insert a new row into a specified table within the database. It constructs and executes an SQL insert query based on the provided arguments and returns the response, indicating the success or failure of the insertion operation.

- ***wyde-db-update-row*.** This function manages the update of an existing row within a specified table in the database. It generates and executes an SQL update query and returns the response, indicating the success or

failure of the update operation

- ***wyde-db-select-row*.** This function enables the retrieval of specific rows from a designated table within the database. It constructs and executes an SQL select query and returns the response, consisting of the selected rows matching the specified criteria.

4.1.3 Saving and Loading Nodes

4.1.3.1 Saving Nodes

The *wyde-db-save-node* function serves as the primary tool for transferring nodes from local memory into a database. It operates with two parameters: the database connection object and the node to be stored. Upon invocation, this function initiates the saving process by first employing the *wyde-db-save-single-node* function to save the contents of node in the database. Subsequently, it checks if the node contains inner nodes. If such inner nodes exist, the function recursively applies itself to each inner node, ensuring comprehensive storage of the node structure regardless of its depth within the tree.

The *wyde-db-save-single-node* function specializes in the individual saving of nodes to the database, A detailed description of how the function works is presented below:

1. Upon invocation, the function initiates the insertion of a row into the *Node* table, capturing essential attributes of the node as per the database schema.

2. It examines the node's relationships, such as left and right neighbours, and records these associations in the relevant tables *NodeLR* and *OpLR*.

3. Node possesses a reference to an outer node, this relationship is documented in the *NodeOuter* table.

4. Inner nodes relationships are being saved to *NodeInnerLR* table.

Importantly, all database interactions occur within an SQL transaction, ensuring data integrity and providing the capability to abort transactions in case of failure.

4.1.3.2 Loading Nodes

The *wyde-db-load-node* function ensures loading nodes from the database into local memory, providing the retrieval and reconstruction of nodes along with their associated relationships.

At first, fundamental information about the node is retrieved from the database based on the provided node key. If successful, a new node is instantiated in local memory with the received parameters. Information about node relationships is fetched from the relevant tables in the database, including *NodeLR*, *OpLR*, *NodeOuter*, and *NodeInnerLR*. Subsequently, the appropriate operation from the model is linked to the node.

Then, the function proceeds to establish relationships with the left and right neighbours of the node. If optional arguments *l* or *r* are provided, their values are set as neighbours. Alternatively, if database information is available, neighbouring nodes are loaded from the database using the *wyde-db-load-neighbour-node* function and set as neighbours. Neighbour nodes are also being updated. Similar procedures are followed for establishing relationships with nodes' neighbours in terms of operation (*opl* and *opr*).

If the *outer* argument is passed, information about the outer relationship is fetched from the database. The passed argument is then set as a pointer to the node's outer node, along with additional attributes retrieved from the database.

If information about inner relationships is retrieved from the database, the function recursively loads inner nodes. Pointers to the leftmost and rightmost inner nodes are then set accordingly.

If the node does not have pointers to it's operation-left and operation-right nodes, it is considered to be an edge node. In that case, it is being set as the leftmost or rightmost node for the node's operation.

Finally, the loaded node is added to the model's hashmap, ensuring its integration into the local memory structure. The function returns the loaded node as the result.

4.2 Proxy Nodes

Proxy nodes are structures existing within the model to gain access to nodes that have been saved to the database. They act as placeholders, maintaining

the integrity of the model's node chain. Each proxy node contains multiple elements:

- **Node Key.** A unique identifier used for accessing the corresponding node in the database.
- **Neighbour References.** Pointers to the right and left neighbour nodes in the model's node chain.
- **Keys of Leftmost and Rightmost Converted Nodes.** Needed for understanding where is the beginning and end of the converted region.

4.2.1 Node-To-Proxy Conversion

The conversion of a regular node to a proxy node begins with the saving of the node to the database.

Once the node is successfully saved, its neighbouring nodes are updated to reflect the changes. Pointers to the old node are adjusted to point to the newly-created proxy node, maintaining the integrity of the node chain within the model.

A new recording is added to the *Proxy* table in the database, which contains node keys for the leftmost and rightmost nodes related to the proxy node. Also, for the node that is being converted, and all of it's underlying nodes, a recording in the *NodeProxy* table in the database is created, where the value of *nid* column is a node's key and the value of *proxy_id* column is the key of the leftmost node that is related to the proxy node.

In case when the saved node was the model's current node, the model's current pointer is updated to point to the proxy node.

Finally, the process of cleaning references to the saved node is performed. This step ensures that any residual references to the old node are removed, preventing inconsistencies within the model structure.

4.2.2 Proxy-To-Node Conversion

The conversion of proxy nodes to regular nodes involves several steps and performed with *wyde-db-proxy-to-node* function.

The conversion process initiates by retrieving the node from the database using

the identifier specified in the proxy node.

After that, references to the proxy node within the model structure are adjusted to reference the newly-loaded node or edge nodes in the case of loading a chain of multiple nodes. This includes updating neighbouring nodes of regular nodes and nodes in terms of operation, as well as the model's current reference if applicable.

By performing these steps during the conversion process, the integrity of the model structure is preserved, ensuring synchronization between local memory and the database.

4.2.3 Handling Deletions

4.2.3.1 Understanding the Emacs Lisp Garbage Collector

Emacs Lisp employs a garbage collector mechanism to manage indirect memory allocation and deallocation [23]. It periodically deallocates unreferenced objects to free up memory space. Objects that are no longer reachable from the root of the execution environment are considered garbage and are suitable for collection. Therefore, in order to delete a piece of data from memory in Emacs Lisp, all references leading to it must be terminated to allow the garbage collector to reclaim the memory occupied by the object.

4.2.3.2 Identifying and Managing References to Deleted Nodes

Upon code analysis, multiple places were identified that can potentially contain references to the deleted node within the model structure:

- **Model's Nodes Hashtable.** Contains references to all existing nodes in the model.
- **Node Neighbour References.** Pointers from the neighbouring nodes to the deleted node.
- **Operation's Leftmost and Rightmost Node.** References to the deleted node as the leftmost or rightmost node of an operation.
- **Node Neighbour References in Terms of Operation.** Operation-left and operation-right pointers from nodes linked to the same operation as deleted node.

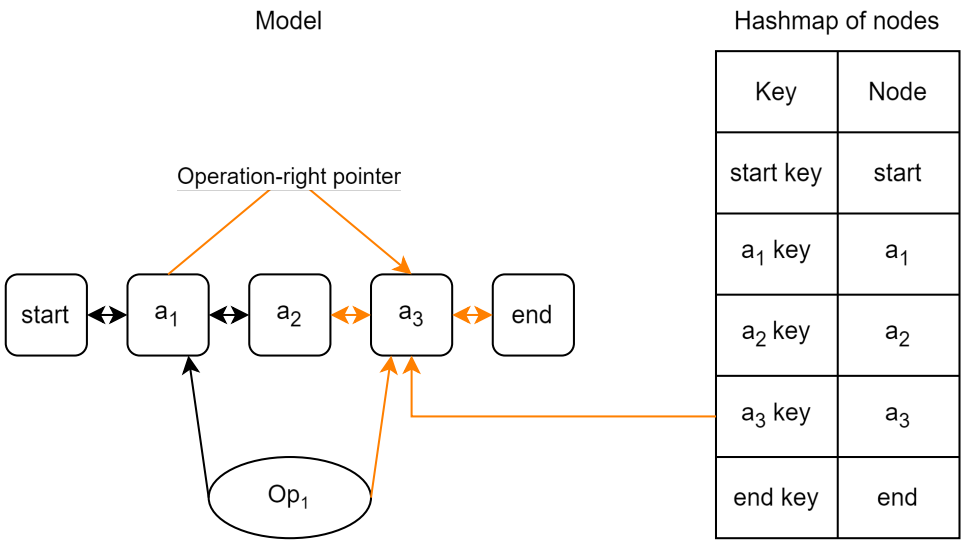

Figure 4.2: Demonstration of possible references to a node within the model.

Figure 4.2 demonstrates all potential references for the node a_3. The orange-colored arrows denote references to the node from other components within the model.

During the process of converting a node to a proxy node, these references are properly handled to ensure that the deletion process will free the local memory: references are removed from the model's hashmap, while references in other places are being adjusted to point at the proxy node.

4.3 Integration to Wyde

The integration of proxy nodes into the Wyde model posed several challenges, primarily related to accessing proxy nodes with functions designed for regular nodes. One notable complication was noticed during the process of traversing the model, where attempts to access the right neighbour of a node resulted in errors if the neighbour was a proxy node.

4.3.0.1 Implementing Advising Functions

To resolve these complications, it was initially considered to write custom functions for accessing parameters depending on the object type. However, this approach was deemed too risky due to potential inconsistencies and complex-

ities in integrating custom functions with existing codebase. Instead, Emacs Lisp's advice mechanism, was leveraged to address this issue.

Emacs Lisp provides a powerful feature called advice, which allows functions to be advised or modified dynamically at runtime. Advising functions involve adding another behaviour before, after, or around the execution of the original function, without modifying its source code[22].

4.3.0.2 Overcoming Limitations of Advising Mechanism

The implemented advising functions were relatively straightforward: they checked the type of the structure being accessed, and if it was a proxy node, the appropriate attribute of the proxy node was returned without invoking the original function, this mechanism is depicted in Figure 4.3. However, it was later discovered that the advising mechanism did not work as expected for structure slot accessing functions.

Upon further investigation, it was found that these functions were inlined by default. Inlining involves replacing a function call with the actual precompiled code of the function, which improves performance but prevents the function from being advised. As a result, the advising mechanism was unable to intercept calls to these functions and modify their behaviour.

To overcome this limitation, a *:noinline* option was added in the structure declaration. This option prevented the compiler from inlining the structure slot accessing functions, allowing them to be advised successfully. With this modification, the advising mechanism could intercept calls to these functions and apply the necessary modifications, ensuring the integration of proxy nodes into the Wyde model. It should be mentioned that the purpose of functions inlining is the execution speed increase and removing it could cause the performance degradation. However, the observations on the slot accessing functions speed after adding the *:noinline* option to the structure declaration did not show any notable speed decrease.

4.4 Model Restoration Modification

The modified model restoration process represents a reduce of memory consumption over the original process by avoiding the loading of irrelevant data related to proxy nodes into local memory. The model restoration is the process of restoring the model to the state that it had by the end of the previous editing session. It involves reading a log file containing the history of operations applied

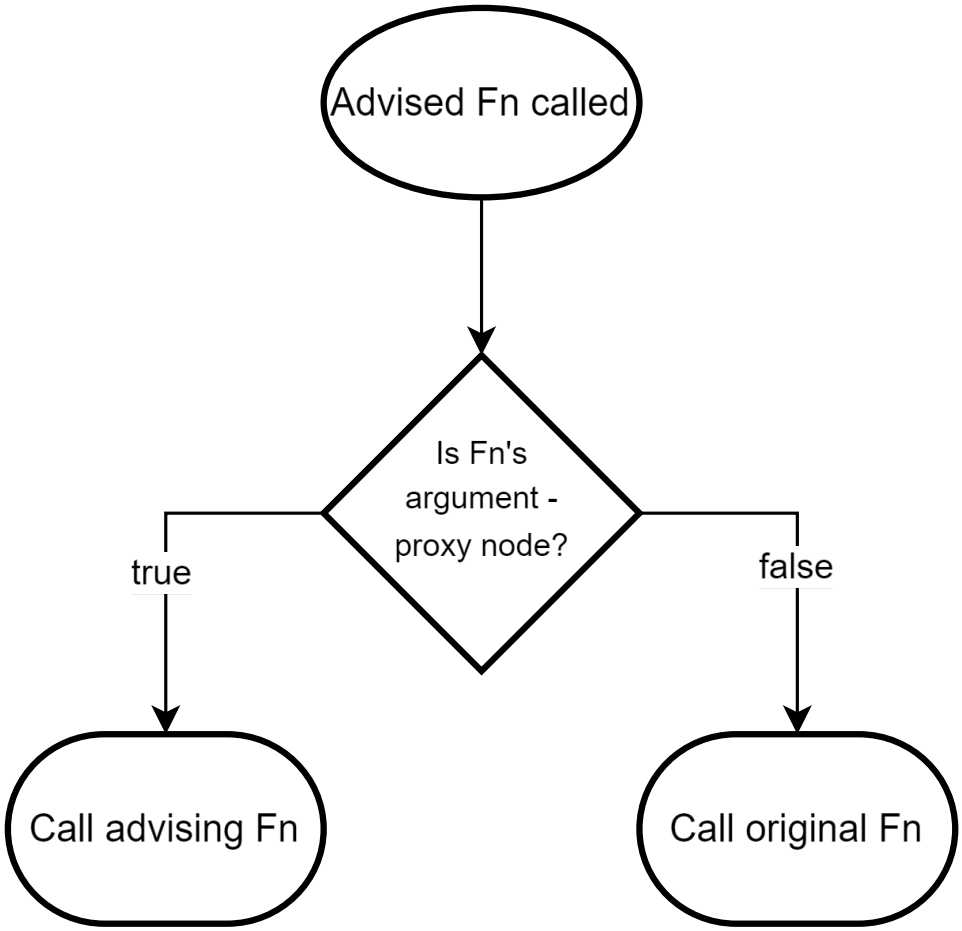

Figure 4.3: A diagram depicting the mechanism of implemented advising functions.

to the model and applying those operation to a newly-created model.

To implement the modified restoration process, several key changes were made. First, a new hashtable called *proxies* is created. This table stores all node-proxy relations from the database. It contains key-value pairs where the key is a node key and the value is a proxy node key. This hashtable is populated using a function called *wyde-db-load-proxy-table*, which retrieves entries from the *NodeProxy* table in the database.

Next, the log file, in JSON format, is read line-by-line, with each line representing a single operation. These lines are decoded from JSON format to operation structures. The decoding function *wyde-mdl-decode-json* was modified to check if the operation's node key, constructed with the data from JSON: operation id and the position of leftmost character affected by operation, is contained in the *proxies* hashtable. If so, an empty entry is added to the operations queue to prevent loading irrelevant operations during queue processing.

Once the log file is read and the queue is filled with operations, the process of applying these operations to the empty model begins. The original integration process was modified to handle situations when the operation that is being applied refers to the neighbour node that was not loaded due to its relation with a proxy node. In that case, this node is being searched in the *proxies* table and then, in case if proxy node is already present in the model, it is being searched in *nodes* hashtable. If the proxy node was not found among the model's nodes, it is loaded and inserted into the model according to the leftmost character place of it's leftmost related node and then set as the neighbour for the node of operation that is being currently applied to the model.

The result is a restored model with proxy nodes and the absence of irrelevant information about nodes and operations related to proxy nodes. A schematic visual representation of the process is depicted in Figure 4.4.

However, it is important to note that the current implementation only supports restoration of models consisting of insertion, deletion, and proxy nodes. Further improvements, especially regarding processing of undo, redo, or grouped operations in the history log, are areas for future work and enhancement of the modified file restoration process.

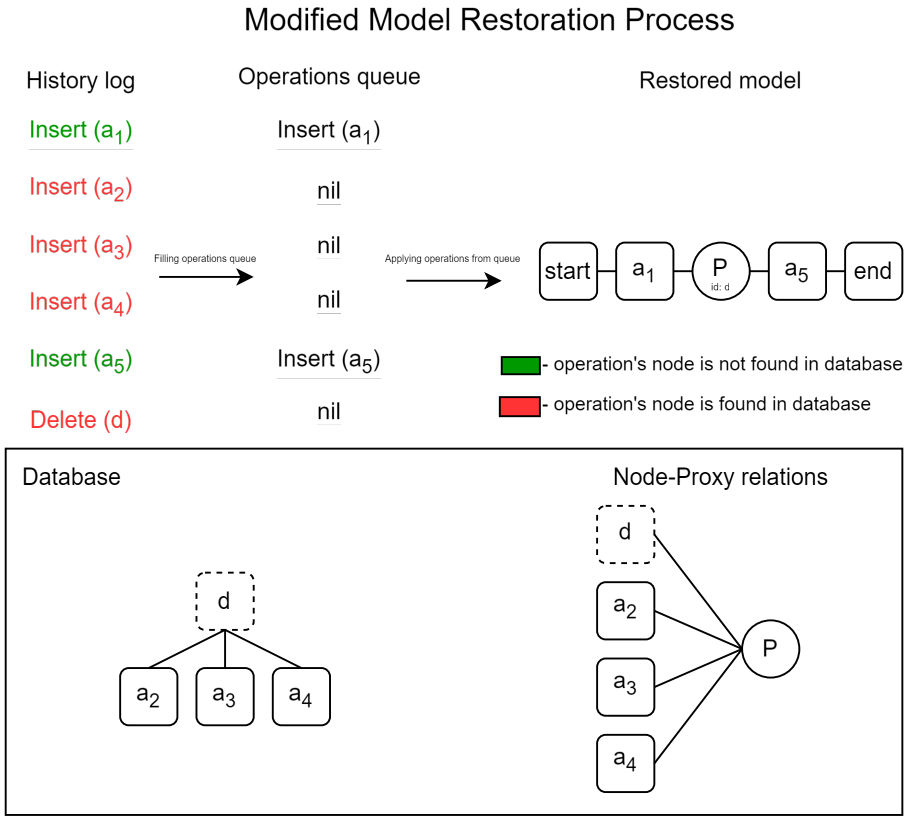

Figure 4.4: A visual representation of the modified model restoration process.

5 Evaluation

5.1 Tests Design

5.1.1 Unit Tests

Unit tests play a crucial role in ensuring a certain degree of correctness of the implemented functionality throughout the development process. These tests are written using Emacs Lisp's built-in testing framework, ERT (Emacs Lisp Regression Testing), which provides a convenient way to define and execute test cases within the Emacs environment.

Each unit test case focuses on a specific aspect of the system's functionality, covering essential operations such as operations related to saving and loading nodes from the database, operations for communication to the database and operations for conversion of regular nodes to proxy nodes and vice versa. To isolate the tests from external dependencies and ensure reproducibility, a temporal database is utilized for each test case. This database is created and initialized before the execution of each test and is deleted upon completion, ensuring a clean testing environment for each run.

For the implementation of unit tests, the *wyde-with-db-test* macro is utilized, demonstrated in Listing 5.1. This macro encapsulates the setup and teardown procedures for each test, including database initialization and cleanup.

Listing 5.1: Supplementary macro for setting up test environment

```
(defmacro wyde-with-db-test (test-name &rest body)
  (declare (indent 1) (debug t))
  `(wyde-with-debug-log test-name nil
    (wyde-with-temp-dir
      (let* ((name "testdb")
             (dbfile (concat default-directory name ".
                sqlite"))
             (db (wyde-db-connect dbfile)))
        (unwind-protect
            (progn ,@body)
          (sqlite-close db)
          (delete-file dbfile))))))
```

Listing 5.2 provides an example of unit test that checks the correctness of the *wyde-db-update-row* function by verifying that a row in the *Node* database table is successfully updated with the provided data.

Listing 5.2: Unit test example

```
(ert-deftest wyde-db-test-update-row ()
  (wyde-with-db-test "wyde-db-test-update-row"
    (sqlite-execute db "INSERT␣INTO␣Node␣(nid,␣str,␣len
       ,␣visible)␣VALUES␣('(8888␣1␣1)',␣'test',␣NULL,␣
       1)")
    (wyde-db-update-row db "Node" '("nid" "str" "len" "
       visible") '((1111 8 8) "new_str" 1 nil) "nid" "
       (8888␣1␣1)")
    (let ((answer (car (sqlite-select db "SELECT␣nid,␣
       str,␣len,␣visible␣FROM␣Node␣WHERE␣nid␣=␣'(1111␣8
       ␣8)'"))))
      (should (equal "(1111␣8␣8)" (pop answer)))
      (should (equal "new_str" (pop answer)))
      (should (equal 1 (pop answer)))
      (should (equal nil (pop answer))))))
```

5.1.2 Performance Tests

Performance tests serve the purpose of evaluating the efficiency and scalability of the implemented solution.

For assessing memory consumption metrics, the *garbage-collect* function is

employed. This built-in Emacs Lisp function initiates the garbage collection process, reclaiming memory occupied by objects that are no longer in use. This function returns data regarding memory usage. During the process of analysis, the primary focus is on the number of used *vector slots*, as in the context of the implemented solution, both nodes and operations in the model are represented as structures, utilizing vector slots for storing their fields.

To measure the execution time of functions, the built-in benchmark tool is utilized. It facilitates the precise measurement of the execution time of specified functions. Specifically, the *benchmark-run-compiled* function is used to accurately measure the execution time of functions under test conditions.

Currently, two types of performance tests are employed to gather essential metrics related to the system's performance under varying conditions.

The first performance test, *wyde-db-prf-test-plain-chain*, operates by receiving two parameters: the total number of nodes in the model and the number of nodes slated for deletion. This test involves creating a new model and executing a sequence of operations equivalent to the total number of nodes specified. Subsequently, a specified number of nodes are deleted from the model and the resulting deletion node is converted into the proxy node. Throughout this process, memory consumption metrics are measured before and after the proxy conversion, alongside the duration of the conversion process. The code for this test is demonstrated in Listing 5.3.

Listing 5.3: Conversion to proxy performance test

```
(defun wyde-db-prf-test-plain-chain (db nodes-number
   del-nodes-number)
  (let ((mdl (wyde-generate-prf-test-mdl nodes-number))
        (chars-to-del (* del-nodes-number 10)))
    (wyde-mdl-do-local-op mdl '(:del 1 ,chars-to-del))
    (wyde-peer-save (wyde-mdl-peer mdl))
    (with-wyde-profile-trace
        (print "BENCHMARK RESULTS:")
        (print (benchmark-run-compiled (
           wyde-db-node-to-proxy db mdl (wyde-node-r (
           wyde-mdl-bom mdl)))
                 (garbage-collect)))))
  t)
```

The second performance test, *wyde-db-prf-test-walk-through-mdl*, similarly constructs a model and populates it with nodes (see Listing 5.4). Multiple deletion operations are then executed, resulting in the creation of several deletion nodes.

Following this, the time required to navigate from the first node to the last in the model is measured. Subsequently, all deletion nodes are converted into a single proxy node, and the same time measurement is taken, This test provides insights into the efficiency of navigation through the model before and after the proxy conversion process.

Listing 5.4: Model navigation performance test

```
(defun wyde-db-prf-test-walk-through-mdl (db
   nodes-number del-nodes-number)
  (let ((mdl (wyde-generate-prf-test-mdl nodes-number))
        (chars-to-del 10))
    (wyde-add-advices)
    (dotimes (_ del-nodes-number)
      (wyde-mdl-do-local-op mdl '(:del 1 ,chars-to-del)
        ))

    (print "BENCHMARK␣RESULT␣OF␣INITIAL␣MODEL:")
    (print (benchmark-run-compiled (
       wyde-walk-through-mdl mdl)))

    (wyde-db-region-to-proxy db mdl (wyde-node-r (
       wyde-mdl-bom mdl))

    (print "BENCHMARK␣RESULT␣AFTER␣REGION␣CONVERSION:")
    (print (benchmark-run-compiled (
       wyde-walk-through-mdl mdl)))
    (wyde-remove-advices)
    t))
```

5.2 Experiments and Results

In this section, the description and outcomes of the experiments are presented, providing insights into the performance and effectiveness of the proposed solution. The specification of the machine used for conducting all tests is as follows:

- **Processor:** AMD Ryzen 7 5800H, 3.20 GHz, 8 CPU cores
- **RAM:** 8 GB
- **Disk:** BC711 NVMe SK hynix 512GB

- **OS:** EndeavourOS

5.2.1 Memory Consumption after Conversion to Proxy Nodes

The first experiment aimed to assess the memory consumption before and after converting regular nodes. The experiment followed a structured process, firstly, an initial memory measurement of an empty model without any additional nodes was performed. Then, the memory consumption of the same model filled with multiple nodes was measured. The next step involved a conversion of a portion of the nodes in the model to a single proxy node. The memory consumption was measured again for the modified model.

The percentage of freed memory was calculated using the formula:

$$M_f = \frac{M_a - M_p}{M_a - M_b}$$

where:

- M_f represents the percentage of freed memory,
- M_a is the memory consumption after adding nodes to the model,
- M_p is the memory consumption after conversion of regular nodes to proxy nodes,
- M_b is the memory consumption before adding nodes to the model.

Multiple measurements were conducted, varying the percentage of nodes in the model converted to proxy nodes, ranging from 25% to 100%, and the size of the models, ranging from 12 nodes to 1000 nodes. The resulting chart can be found in the Figure 5.1.

The results of the experiment revealed that the percentage of freed memory was approximately half of the percentage of converted nodes. This proportion was observed across different sizes of models. The percentage of freed memory was lower for smaller models and increased as the number of nodes in the model increased.

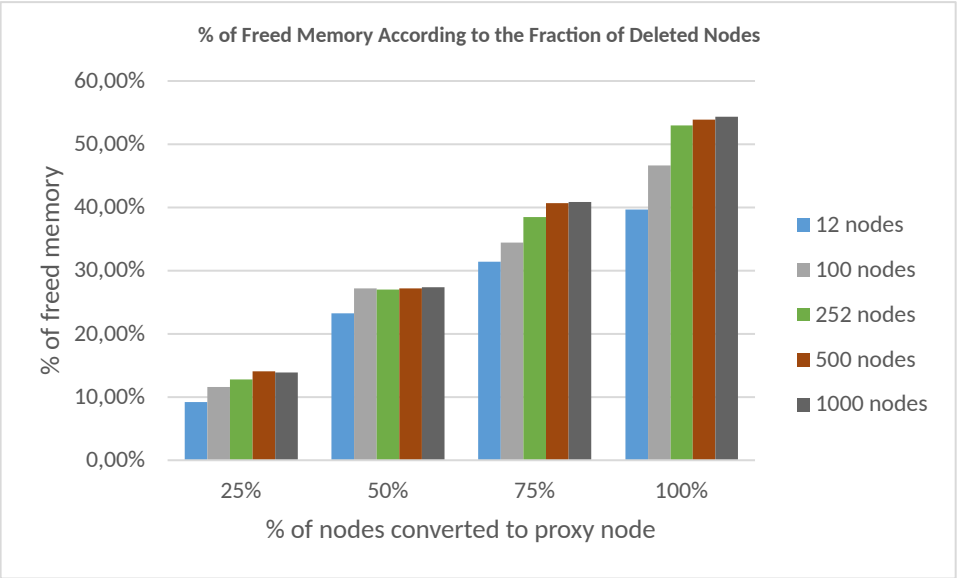

Figure 5.1: Percentage of freed memory according to the fraction of deleted nodes.

5.2.2 Memory Consumption Of Restored Model

The second experiment focused on comparing the memory consumption of a restored file using two different restoration processes: the original restoration process and the modified restoration process.

The memory consumption of an opened file was measured after applying the original restoration process without any modifications. This process typically loads all operations related to the file into local memory. After that, the same file was opened using a modified restoration process, which excludes operations stored in the database and referred to by proxy nodes from being loaded into local memory.

For this experiment, models were created consisting of a single deletion node, each with a varying number of underlying insertion nodes, ranging from 500 to 1500 nodes. After model creation, the deletion node was converted to a proxy node, ensuring that all necessary information for the modified restoration process was stored in the database.

The memory consumption was measured using the *htop* utility [26], which provides real-time monitoring of system resources, including memory usage by individual processes.

The results, depicted in Figure 5.2, indicated that the memory consumption of

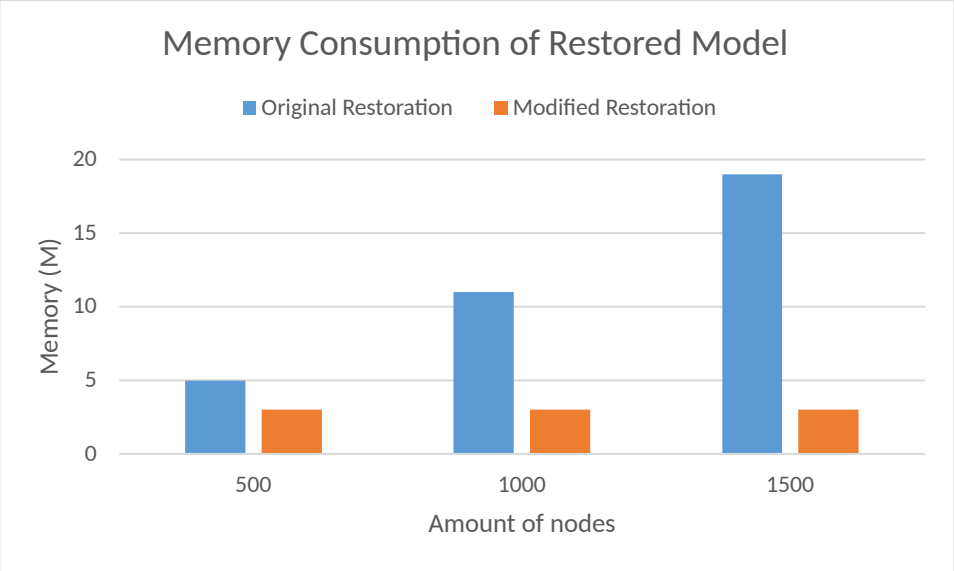

Figure 5.2: Memory consumption of restored model.

the file restored by the regular restoration process increased with the number of nodes in the model. In contrast, the file restored by the modified restoration process maintained a fixed size, indicating that unnecessary operations and nodes referred to by the proxy node were not loaded into local memory.

This experiment demonstrated the advantage of the modified restoration process in reducing memory consumption by avoiding the loading of irrelevant data into local memory.

5.2.3 Nodes Conversion Speed

The third experiment aimed to assess the speed of converting regular nodes to proxy nodes. For this experiment, a deletion node containing a varying number of underlying insertion nodes, ranging from 50 to 1000 nodes, was selected, and the time taken to convert it to a proxy node was measured. The time taken for the conversion operation was measured using the tools described in section 5.1.2.

The results are visualized in Figure 5.3. They revealed a non-linear relationship between the number of nodes converted during the conversion operation and the total elapsed time of the conversion process.

Furthermore, the experiment demonstrated that the conversion time varied based on the amount of text being deleted. Conversion of a small number

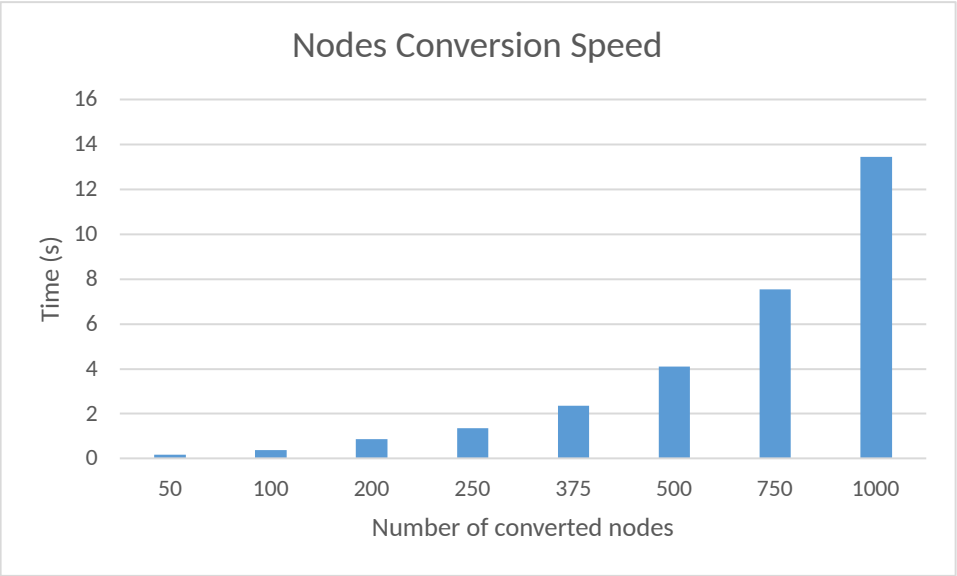

Figure 5.3: Nodes conversion speed.

of nodes, representing a small amount of text, such as a couple of sentences, took a relatively short amount of time, approximately 189 ms for 50 nodes. In contrast, conversion of a large number of nodes, representing pages of plain text, required a significantly longer time, approximately 13.7 seconds for 1000 nodes.

5.2.4 Model Restoration Speed

The fourth experiment aimed to compare the speed of model restoration between the original restoration process and the modified restoration process. Multiple measurements were conducted using different files, with file variety and structure consistent with those used in the second experiment, which measured the memory consumption of restored models.

The results, depicted in Figure 5.4, indicated that the modified restoration process showed faster restoration speed compared to the original restoration process.

Both processes showed an increase in restoration time as the model size increased. However, the elapsed time of the modified restoration process increased at a slower rate compared to the original restoration process. This behavior can be attributed to the difference in how each restoration process operates: during the original restoration process, each operation is processed,

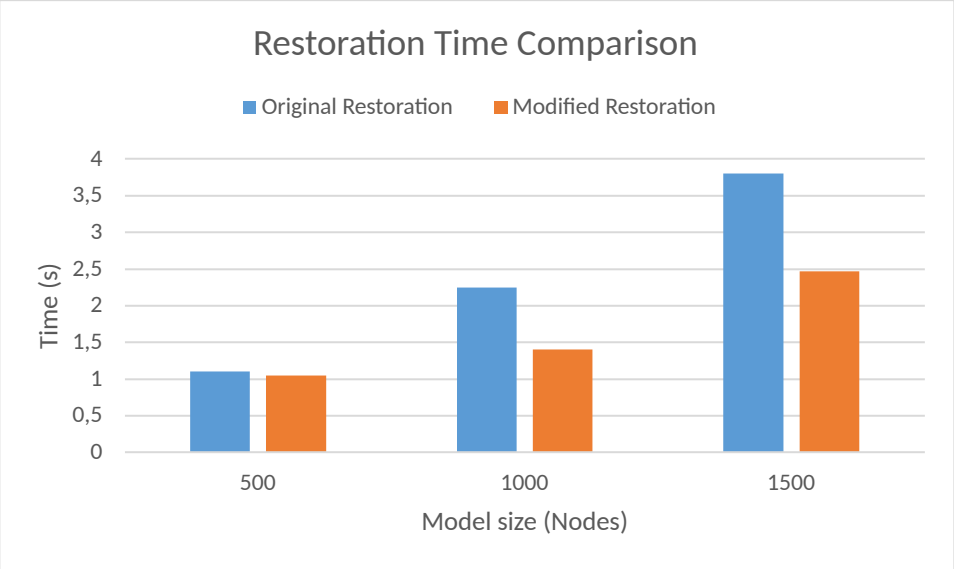

Figure 5.4: Restoration time comparison.

leading to longer restoration times, while the modified restoration only processes operations that are not associated with any proxy node.

Additionally, the time consumed by the garbage collector during the restoration processes was measured. The visual representation of the results can be seen in Figure 5.5. It revealed that a smaller proportion of total elapsed time was consumed by the garbage collector during the modified restoration process compared to the original one. This observation is depicted in a chart representing the percentage of garbage collection time from the total elapsed time of the process.

5.2.5 Model Navigation Speed

The final experiment aimed to measure the increase in the model's navigation speed after converting a fraction of nodes into a single proxy node. Navigation through the model involves traversing from the beginning node to the ending node by applying the *wyde-node-r* function to access neighbouring nodes until reaching the ending node.

The experimental procedure involved several steps. Initially, a model was configured, consisting of multiple deletion nodes ranging from 12 to 1000 nodes, with each deletion node containing a single underlying insertion node. Subsequently, the navigation speed of the model was measured before any modifications were made. Following this, a fraction of nodes forming a linked

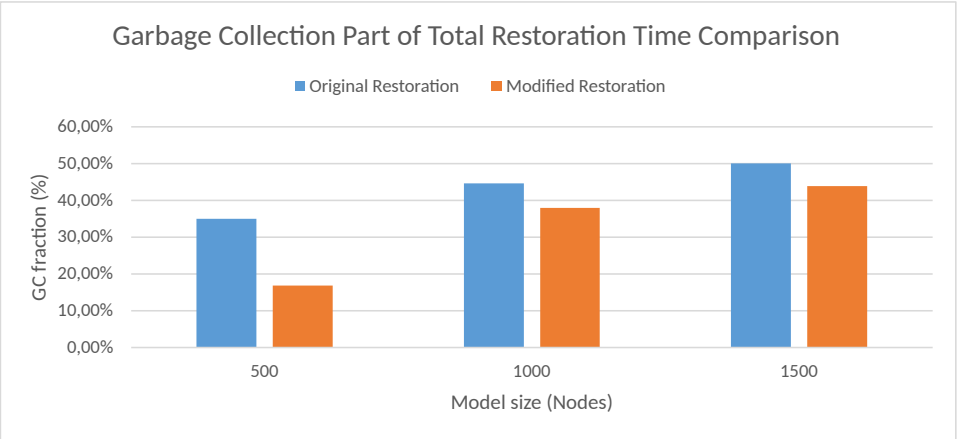

Figure 5.5: Comparison of garbage collection part of total restoration time.

list, ranging from 25% to 100% of nodes in the model, was converted into a single proxy node. Finally, the navigation speed of the modified model was measured.

The percentage increase in navigation speed was calculated using the formula:

$$I = \frac{S_b - S_a}{S_b}$$

where:

- I represents the percentage increase in speed,
- S_b is the navigation speed of the model before modifications,
- S_a is the navigation speed of the model after conversion operation.

The findings illustrated in Figure 5.6 revealed that replacing multiple nodes with a single proxy node led to an increase in navigation speed, consistent with expectations.

The degree of improvement increased with a higher fraction of deletion nodes converted to a proxy node, it shows that in scenarios where multiple deletion operations are performed in the same text region, the conversion operation would enhance program performance for operations involving model navigation.

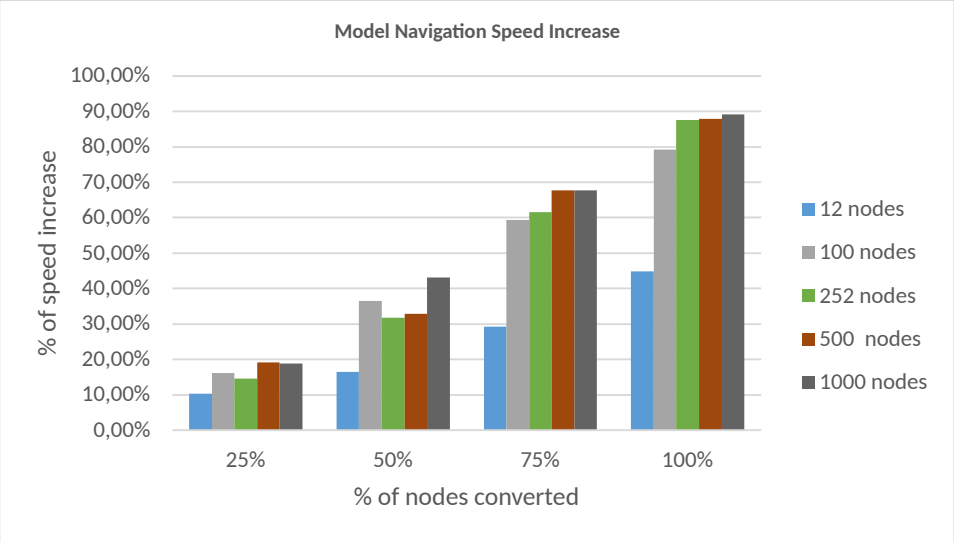

Figure 5.6: Comparison of model navigation speed increase.

5.3 Analysis

The experimental results provide valuable insights into the performance and effectiveness of the implemented solution.

The experiments demonstrate a notable reduction in memory consumption achieved by converting regular nodes to proxy nodes. This optimization allows for more efficient utilization of system resources, particularly when dealing with large-scale collaborative editing scenarios. However, there is potential for further improvements of the conversion process, in terms of current implementation, to achieve even higher memory reduction by removing operations data from the local memory along with the related nodes.

The modified file restoration process exhibits superior performance compared to the original restoration process. By avoiding the loading of irrelevant operations associated with proxy nodes, the modified process enhances the speed of file restoration operation and reduces the memory consumption of the restored file.

Converting multiple nodes into a single proxy node leads to a significant increase in navigation speed through the model. This improvement is particularly beneficial for operations involving model traversal.

The one notable concern is the performance degradation for the conversion of a large number of nodes. The time required for this operation grows exponentially,

so addressing these performance limitations through optimization techniques is crucial for enhancing the effectiveness of the proposed solution.

6 Discussion

6.1 Practical Scenarios

The proposed solution has potential applications in various real-world scenarios, particularly those involving resource-constrained environments.

In situations where users need to open large files on devices with limited memory capacity, traditional editors may struggle to handle the entire file in memory, leading to sluggish performance or even crashes. By offloading less relevant data from local memory to disk storage, the proposed solution can enable efficient editing of large files without overwhelming the system's memory resources.

Another potential application is the ability to open multiple files in parallel on machines with small memory size. Reducing the local memory footprint for each document would expand the capacity for working with multiple files concurrently.

The solution holds potential for broader application beyond collaborative editing systems, extending to any data structure stored in local memory that encounters scenarios of soft deletion. Soft deletion typically involves marking data as deleted without physically removing it, keeping it accessible for potential retrieval or historic reference.

6.2 Potential Benefits and Limitations

The solution offers several potential benefits. Firstly, it effectively reduces memory usage, enabling more efficient utilization of system resources, especially on machines with limited memory capacity.

Additionally, the introduction of proxy nodes in the model architecture enhances performance in terms of navigation through the model due to the replacement of multiple nodes with a single element.

However, the proposed approach may present a potential limitation when applied to opening files with a small portion of proxy nodes. In scenarios where the file contains only a small number of proxy nodes, the time required for establishing a database connection and executing queries to retrieve data from the database could outweigh the time saved by the optimization algorithm. As a result, the overall performance for opening small-sized files may be negatively affected, highlighting a trade-off between memory usage reduce and processing efficiency.

Another potential limitation of the approach relates to the performance degradation observed during the conversion of a large number of regular nodes to a single proxy node. The experiments results demonstrated high time consumption for the conversion of a big amount of nodes. Future research efforts should focus on optimizing the conversion process to mitigate the performance impact associated with processing a significant volume of nodes.

The drawback of the implemented solution relates to the suboptimal optimization of memory consumption. Despite offloading node data to disk storage, operations linked to deleted nodes remain in local memory. This inefficiency reduces the rate of memory savings achieved by the solution and requires further refinement to address memory management challenges.

6.3 Acquired Knowledge

Throughout the working on this master project, significant knowledge and expertise were acquired across various domains, enriching both technical skills and research methodologies. This journey has been marked by valuable lessons learned, challenges encountered, and strategies developed for effective problem-solving and decision-making.

The project provided a deep dive into collaborative editing systems, particularly focusing on the implementation and optimization of CRDT-based solu-

tions.

Engaging in the iterative development process underscored the importance of software engineering practices such as code refactoring, version control, and testing methodologies. Applying these practices ensured the robustness and maintainability of the developed solution.

Various challenges were presented during development process, including addressing memory overhead issues and balancing trade-offs between different design choices. Leveraging critical thinking, collaboration with my supervisor and a willingness to experiment with new approaches were essential strategies for overcoming these challenges.

6.4 Future Work

While the current solution represents a step forward in addressing memory management challenges in CRDT-based collaborative editing systems, there exist several promising directions for further research, development and improvement.

One potential area for improvement is to extend support for undo and redo operations for the operations that result in the creation of proxy nodes. This enhancement would provide users with greater flexibility and control over their editing history, enabling seamless navigation and manipulation of document revisions.

Another potential improvement involves enhancing the processing of remote operations received from other peers. Specifically, instead of directly applying deletion operations to the local model, a more optimized approach could involve inserting proxy nodes in place of these deletions. This strategy would be particularly beneficial when handling deletion operations that target nodes not previously present in the local model. By inserting proxy nodes directly, the system could avoid unnecessary loading of less relevant information into local memory, effectively offloading it to the database.

Additionally, future work could focus on the deletion of operations that refer to deleted nodes from the local memory. This optimization would further decrease memory consumption, enhancing the overall efficiency of the solution.

/7 Conclusion

The development and implementation of the Partial Persistence approach for CRDT data model have yielded considerable reduction in memory consumption. Through a series of iterative development cycles, key features such as the proxy conversion and the modification of the file restoration process have been successfully integrated into the system. This approach offers notable benefits, including reduced memory consumption, improved performance for model navigation, and faster model restoration.

The proposed solution represents a step forward in P2P collaborative editing systems, offering a new approach for managing large-scale documents with efficiency and effectiveness. With continued development and refinement, it has the potential to become a valuable approach for integration into other collaborative text editors that do not rely on a single server, thereby enhancing productivity and usability for users.

While the proposed solution offers memory optimization benefits, further optimization efforts are needed to refine memory management and address the challenges of potential performance degradation described in Section 6.2.

Bibliography

[1] Douglas C. Engelbart. “Workstation History and The Augmented Knowledge Workshop.” In: Proceedings of the ACM Conference on the History of Personal Workstations, 1986, pp. 73–83.

[2] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen. “Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems.” In: *ACM Transactions on Computer-Human Interaction (TOCHI)* 5.1 (1998), pp. 63–108.

[3] M. Ahmed-Nacer, C. Ignat, G. Oster, H. Roh, and P. Urso. “Evaluating CRDTs for Real-time Document Editing.” In: 11th ACM Symposium on Document Engineering., 2011, pp. 103–112.

[4] M. Shapiro, N. Pregui¸ca, and C. Baquero. “Conflict-free replicated data types.” In: Symposium on Self-Stabilizing Systems, 2011, pp. 386–400.

[5] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. *A comprehensive study of Convergent and Commutative Replicated Data Types*. Research report. Inria – Centre Paris-Rocquencourt, 2011.

[6] P. Bailis and A. Ghodsi. “Eventual consistency today: Limitations, extensions, and beyond.” In: *Communications of the ACM* 56.5 (2013), pp. 55–63.

[7] P. S. Almeida, A. Shoker, and C. Baquero. “Efficient state-based CRDTs by deltamutation.” In: Int. Conf. on Networked Systems (NETYS), 2015, pp. 62–76.

[8] W. Yu, L. Andre, and C. Ignat. “A CRDT supporting selective undo for collaborative text editing.” In: DAIS, 2015, pp. 193–206.

[9] V. Enes, P.S. Almeida, and C. Baquero. “The Single Writer Principle in CRDT Composition.” In: Proceedings of the Programming Models and Languages for Distributed Computing, PMLDC ’17, 2017, pp. 1–3.

[10] M. Shapiro. “Replicated Data Types.” In: *Encyclopedia Of Database Systems*. Springer-Verlag, 2017, pp. 1–5.

[11] D. Sun, C. Sun, A. Ng, and W. Cai. “Real differences between OT and CRDT in correctness and complexity for consistency maintenance in co-editors.” In: *Proceedings of the ACM on Human-Computer Interaction* 4.CSCW1 (2020), pp. 1–30.

[12] W. Yu and C. Ignat. "Conflict-free replicated relations for multisynchronous database management at edge." In: IEEE Int. Conf. Smart Data Services (SMDS), 2020, pp. 113–121.
[13] R. Vaillant, D. Vasilas, M. Shapiro, and T.L. Nguyen. "CRDTs for truly concurrent file systems." In: Proceedings of the 13th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage 21), 2021, pp. 35–41.
[14] G. Litt, S. Lim, M. Kleppmann, and P. van Hardenberg. "Peritext: A CRDT for Collaborative Rich Text Editing." In: *Proceedings of the ACM on Human-Computer Interaction* 6.531 (2022), pp. 1–36.
[15] Y. Mahéo, F. Guidec, and C. Noûs. "CRDT-based Collaborative Editing in OppNets: a Practical Experiment." In: 17th Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies (UBICOMM 2023), 2023, pp. 13–21.
[16] Box. URL: https://www.box.com/collaboration. (accessed: 28.04.2024).
[17] Conclave. URL: https://conclave-team.github.io/conclave-site/. (accessed: 28.04.2024).
[18] Emacs Docs. URL: https://emacsdocs.org/docs/elisp/Emacs-Lisp. (accessed: 28.04.2024).
[19] Google Docs. URL: https://docs.google.com/. (accessed: 28.04.2024).
[20] DocSynch. URL: http://docsynch.sourceforge.net/. (accessed: 28.04.2024).
[21] GNU Emacs Documentation. URL: https://www.gnu.org/software/emacs/documentation.html. (accessed: 28.04.2024).
[22] GNU. *Advising Functions (GNU Emacs Lisp Reference Manual)*. URL: https://www.gnu.org/s/emacs/manual/html_node/elisp/Advising-Functions.html. (accessed: 28.04.2024).
[23] GNU. *Garbage Collection (GNU Emacs Lisp Reference Manual)*. URL: https://www.gnu.org/software/emacs/manual/html_node/elisp/Garbage-Collection.html. (accessed: 28.04.2024).
[24] Gobby. URL: https://gobby.github.io/. (accessed: 28.04.2024).
[25] Hyperpad. URL: https://github.com/hackergrrl/hyperpad. (accessed: 28.04.2024).
[26] H. Muhammad. *htop - an interactive process viewer*. URL: https://htop.dev/. (accessed: 28.04.2024).
[27] Microsoft Office Online. URL: https://www.office.com/. (accessed: 28.04.2024).
[28] Dropbox Paper. URL: https://www.dropbox.com/paper/start. (accessed: 28.04.2024).
[29] Microsoft SharePoint. URL: https://microsoft.sharepoint.com/. (accessed: 28.04.2024).
[30] SubEthaEdit. URL: https://subethaedit.net/. (accessed: 28.04.2024).
[31] Zoho WorkDrive. URL: https://www.zoho.com/workdrive/. (accessed: 28.04.2024).