FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	System for real-time data broker's historical state replay
Student:	Bc. Peter Večeřa
Supervisor:	Mgr. Martin Major
Study program:	Informatics
Branch / specialization:	Web Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2025/2026

Instructions

Totem is a data broker whose main purpose is to facilitate very fast data exchange between real-time applications. Thanks to its tree structure and push mechanism, the service is capable of consuming or producing large amounts of data with minimal latency.

The aim of the work is to create the Totem Archiver service, which will be able to store data produced by the Totem service and subsequently search through it efficiently. The service will allow the reconstruction of the Totem tree structure at any point in the past, which Totem itself does not allow.

At any moment, Totem contains millions of values and at the same time, thousands of them change every second – hence, reconstructing the state at a specific time will require careful design and the use of big data technologies.
1. Familiarize yourself with and then introduce the Totem service.
2. Analyze the functional and non-functional requirements, discuss possible database solutions.
3. Based on the analysis, propose a suitable solution meeting the requirements from the previous point.
4. Implement and test the proposed solution.
5. Propose suitable metrics including user response speed and the amount of stored data.
6. Summarize the achieved results, suggest possible future improvements based on the

---

Electronically approved by Ing. Jaroslav Kuchař, Ph.D. on 19 February 2024 in Prague.

experiences gained.

Kleppmann, M. (2017). Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. O'Reilly Media. (2017)

Electronically approved by Ing. Jaroslav Kuchař, Ph.D. on 19 February 2024 in Prague.

Master's thesis

SYSTEM FOR REAL-TIME DATA BROKER'S HISTORICAL STATE REPLAY

Bc. Peter Večeřa

Faculty of Information Technology
Department of Software Engineering
Supervisor: Mgr. Martin Major
May 8, 2024

Czech Technical University in Prague
Faculty of Information Technology
© 2021 Bc. Peter Večeřa. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis: Večeřa Peter. *System for real-time data broker's historical state replay.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2021.

Contents

List of Figures

List of Tables

List of code listings

I would like to thank, above all, my thesis supervisor Mgr. Martin Major for his willingness, professional advice, and time he devoted to supervising this thesis. Furthermore, I would also like to thank my family and friends for their support.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. I further declare that I have concluded an agreement with the Czech Technical University in Prague, on the basis of which the Czech Technical University in Prague has waived the right to conclude a licence agreement on the utilization of this thesis as a school work pursuant to Section 60(1) of the Copyright Act. This fact does not affect the provisions of Section 47b of the Act No. 111/1998 Coll., on Higher Education Act, as amended.

In Prague on May 8, 2024

Abstract

The thesis focuses on creating a microservice capable of archiving and subsequently retrieving data produced by the Totem service. At the outset, key functional and non-functional system requirements will be identified, based on which the service design will be presented. A crucial factor in the design is the massive number of messages that the service must efficiently archive. The design must also consider that the data will be stored without retention. The final emphasis will be on ensuring that the proposed solution can search the data quickly while remaining cost-effective. The service has been tested and subsequently deployed in a production environment, confirming its robust architecture that remains stable and efficient even under high load.

Keywords API, databases, scalability, microservices, Athena, Neo4j

Abstrakt

Práce se zaměřuje na vytvoření mikroslužby, která bude schopná archivovat a následně vyhledávat data produkovaná službou Totem. Na začátku budou identifikovány klíčové funkční a nefunkční požadavky na systém, na jejichž základě bude představen návrh služby. Klíčovým faktorem při návrhu je obrovský počet zpráv, které musí služba efektivně archivovat. Při návrhu je také nutné zohlednit, že data budou ukládána bez retence. Závěrečný důraz bude kladen na to, aby navrhované řešení bylo schopno nad daty rychle vyhledávat, ale zároveň nebylo příliš nákladné. Služba byla testována a následně nasazena do produkčního prostředí, kde se potvrdila její robustní architektura, která zůstává stabilní a výkonná i při vysoké zátěži.

Klíčová slova API, databázy, škálovateľnosť, mikroslužby, Athena, Neo4j

List of abbreviations

API	Application Programming Interface
ACID	Atomicity, Consistency, Isolation, Durability
AWS	Amazon Web Services
CSV	Comma-Separated Values
DSL	Domain-Specific Language
ECS	Elastic Container Service
FR	Functional Requirement
HTTP	Hypertext Transfer Protocol
I/O	Input/Output
ISO	International Organization for Standardization
JDBC	Java Database Connectivity
JSON or XML	JavaScript Object Notation or Extensible Markup Language
JVM	Java Virtual Machine
LZ4	LZ4 Compression Algorithm
NFRs	Non-Functional Requirements
NOSQL	Not Only SQL
OOM	Out Of Memory
ORDBMS	Object-Relational Database Management System
RDBMS	Relational Database Management System
RAM	Random Access Memory
REST	Representational State Transfer
RESTful	Representational State Transfer Fulfillment
S3	Simple Storage Service
SQL	Structured Query Language
TSDB	Time Series Database
TCP connections	Transmission Control Protocol Connections
UC	Use Case

Introduction

In today's modern digital era, the importance of storing historical data for later analysis cannot be overstated. This capability propels modern companies forward, enabling them to leverage accumulated information to drive decision-making and innovation. As part of this broader context, the Totem service in our company serves as a pivotal data broker, capable of real-time data transfer with minimal latency and handling vast amounts of data.

While an archival service currently exists within our organization, it primarily serves the purpose of data storage. However, due to the sheer volume and complexity of data produced by the Totem service, integrating these data into the existing system would render search operations virtually impossible. Therefore, addressing this challenge is crucial for enhancing our data retrieval capabilities and ensuring the sustainability of our data management practices.

This thesis specifically focuses on the archiving of data generated by the Totem service and will explore solutions that not only facilitate efficient storage but also enable effective querying capabilities over the archived data. These efforts aim to optimize our archival processes and enhance accessibility, ensuring that our data assets can be fully utilized for analytical and operational purposes.

The thesis is structured into six chapters. The first chapter focuses specifically on the Totem service. In this chapter, readers are introduced to the design of the service and its benefits. It presents the internal structure of the service as well as implementation details. The second chapter addresses the analysis of the newly emerged service, specifically its functional and non-functional requirements. Subsequently, potential database solutions that could meet these requirements are analyzed. The chapter concludes with a discussion of the challenges encountered. The third chapter describes in more detail the design of the database solution and the API that the service will provide to users. The chapter also mentions the technologies employed and concludes with a high-level overview of the proposed architecture. The fourth chapter describes the process of implementing various components, emphasizing the chronology of the chapters and the flow of data within the service for a clearer understanding. The fifth chapter focuses on testing the service from a functional perspective and includes the results of the performance metrics achieved by the service. The sixth and final chapter discusses possible extensions and enhancements to the newly developed service.

Goals

The primary goal of this thesis is to design and implement the Totem Archiver service, focusing on the efficient storage and querying of massive data flows characteristic of the Totem service. Key objectives include optimizing query performance through advanced indexing strategies and data partitioning and ensuring scalability to handle growing data volumes.

Furthermore, the service aims to support fast and accurate reconstructions of the Totem state at any given moment, utilizing custom-tailored data management techniques for effective historical data analysis.

Benchmarking and performance tuning based on varied use cases will guide system optimization, ensuring the Archiver meets the dual demands of rapid data retrieval and cost-effective storage, thereby enhancing decision-making processes with swift, reliable access to historical data insights.

Chapter 1

Understanding the Totem service

This chapter introduces the Totem service, outlining its purpose, design principles, and the technological innovations that enable it to facilitate fast and reliable data exchange for real-time applications. It delves into the core aspects of Totem's architecture, such as its in-memory data management and tree-based structure, which are pivotal for its performance. Additionally, the chapter discusses Totem's implementation of the publisher-subscriber model, designed to minimize latency and maximize efficiency. The overarching goal is to provide a foundational understanding of Totem's operation, highlighting its unique approaches to overcoming common data brokerage challenges.

1.1 Detailed look at the architecture of Totem

Totem is an advanced data broker, designed with speed and reliability at its core, serves as a central hub for data transactions across many real-time applications. Unlike its predecessors, Totem eschews persistent storage in favor of an in-memory approach, ensuring data flows freely and swiftly without the hindrances of disk-based lag. This design choice boosts performance and aligns perfectly with the demands of applications that thrive on immediacy and up-to-the-minute data accuracy.

Totem's architecture has a unique design tailored to optimize data exchange speeds while maintaining a high degree of reliability. The tree-based structure at its foundation is both elegant and functional, facilitating a clear and logical organization of data. In this structure, leaves represent data points, and the paths leading to these leaves define their location within the vast data landscape Totem manages. This setup is crucial for the publisher/subscriber pattern Totem employs, enabling efficient and targeted data distribution and collection.

1.2 In-memory approach

Totem's architecture is strategically designed to hold its state entirely in memory, a decision that significantly contributes to its remarkable speed in data exchange processes. By bypassing the need for disk-based storage, Totem ensures rapid access to data, allowing for near-instantaneous

updates and retrieval. This in-memory approach is extremely useful for real-time applications where latency can greatly affect performance and user experience.

However, this design choice comes with a trade-off; in the event of a system failure, Totem lacks the ability to recover its previous state automatically. Regardless, Totem's primary function as a real-time data broker is to prioritize speed and efficiency over state recovery, aligning with use cases where the most current data exchange is more valuable than historical data continuity.

1.2.1 What about snapshots?

In Totem's in-memory architecture, creating snapshots for state recovery is impractical due to operational constraints. For instance, if an application crash, such as an OOM error occurs, generating a snapshot upon shutdown isn't feasible. Additionally, relying on outdated snapshots can lead to inaccuracies, as they may not reflect the system's current state, given Totem's real-time nature. This mismatch could introduce data inconsistencies upon recovery. Hence, traditional methods like snapshots or integrating with persistent storage are limited in their effectiveness for Totem, underscoring the need for innovative approaches tailored to its unique in-memory and real-time operation.

1.3 Internal data structure

A core feature of Totem's design is the concept of a `patch`, which is represented as a combination of a `path` and a `value`. This innovative approach allows for granular updates and retrievals, minimizing data transfer volumes while maximizing relevance and precision.

A `path` is defined as a non-empty sequence of segments, with each segment representing one layer within the tree structure. Together, these segments delineate the route to a `leaf` node, which contains a specific value. `Leaves` are uniquely characterized by data types, as they are integral to the path and represent its terminal segment. Given that paths are essentially lists of strings within the system, leaves are explicitly annotated with their corresponding data types. For instance, a leaf holding an integer value of 1 is denoted as `INT(1)`.

The example `root/source/7/markets/1/selections/2/quote/back/0/price/DOUBLE(0.5)` illustrates how specific patch can be, ensuring that only the most pertinent data is exchanged, in line with the system's high-efficiency ethos.

1.3.1 Efficiency of tree-based data organization

The tree-based structure of Totem significantly enhances the efficiency of data organization and retrieval, leveraging the inherent advantages of a tree's hierarchical nature. In a tree structure, the number of layers, or the depth, grows logarithmically with the addition of elements—meaning that even with a large number of nodes (n), the depth of the tree would only be $\log(n)$. This characteristic ensures that insertion and retrieval operations can be performed quickly, as navigating through a minimal number of layers allows fast access to any node.

Furthermore, the tree structure supports flexible and efficient data retrieval patterns, which is especially beneficial for operations that require accessing multiple child nodes. By employing wildcards in path segments, represented by *null*, Totem allows for the retrieval of all child nodes under a specific branch with minimal effort. This feature is handy for subscribers interested in a broad set of data updates, enabling them to receive comprehensive information with a single query. Such capabilities ensure that Totem meets the demands for high-speed data processing and offers versatile and powerful data access mechanisms suited for real-time applications.

1.4 Enriching data types and message structure

Totem's support for a broad spectrum of data types demonstrates its versatility and readiness to cater to diverse application needs. From fundamental types like Strings and Integers to more complex ones such as BigDecimal and Timestamp, Totem strives to provide support for a wide variety of data types.

This adaptability is further enhanced by the use of the MessagePack format for message serialization. Compact and efficient, MessagePack allows Totem to maintain its swift data handling capabilities without sacrificing detail or depth in the information exchanged. Each message's composition, encompassing a type, payload, and metadata, ensures that data is not just transferred but done so with context and precision, reinforcing the system's robustness.

1.4.1 Why MessagePack?

The choice of the `MessagePack` format for message serialization within Totem significantly enhances its performance, especially when compared to more traditional formats such as `JSON` or `XML`. `MessagePack` stands out due to its binary nature, which ensures compactness and speed in data serialization and deserialization processes [1]. For instance, consider a scenario where Totem needs to serialize a data structure containing various types of information, including strings, integers, and floating-point numbers. In `JSON`, this data structure might be represented in a relatively verbose text format, which, while human-readable, results in larger message sizes and, consequently, slower network transmission and processing times. With its additional tagging and structure, XML would further worsen these issues, leading to even larger message sizes.

In contrast, MessagePack can represent the same data structure in a much more compact binary format. This compactness significantly reduces the size of the messages being exchanged, enabling faster data transmission over the network and quicker processing by the receiving application. For a real-time system like Totem, which handles millions of values and thousands of changes every second, the efficiency gains from using `MessagePack` can be substantial. By reducing the overhead associated with serialization and deserialization, Totem can maintain its high throughput and low latency, ensuring that real-time applications relying on it can operate smoothly and efficiently. This strategic choice of serialization format is crucial to Totem's ability to facilitate fast data exchanges between real-time applications, underscoring its commitment to performance and reliability.

1.5 Publisher-subscriber pattern

The Totem service employs a highly efficient publisher-subscriber pattern, a fundamental communication paradigm designed to enhance real-time data exchange across distributed systems.

In this model, publishers are responsible for generating and dispatching data updates, while subscribers sign up to receive notifications about changes relevant to their specific interests. This decoupling of data production and consumption allows for a dynamic, scalable, and flexible system architecture.

Totem leverages this pattern to its full potential by facilitating immediate data push over TCP connections, ensuring subscribers receive updates with minimal latency. This mechanism is crucial for applications requiring real-time responsiveness, as it eliminates the need for polling and ensures that data flows seamlessly and efficiently from publishers to subscribers, maintaining the system's overall performance and reliability.

1.5.1 Optimizing latency through subtree segmentation

In addressing the latency challenge within the Totem storage system, a novel approach has been proposed to optimize the handling of data updates by introducing multiple storage subtrees. This strategy aims to mitigate the bottleneck effect caused by the sequential processing of messages by dividing the storage component into distinct subtrees. Each producer is allocated a specific subtree for message production, facilitating parallel processing and significantly reducing the risk of system-wide slowdowns due to extensive updates from a single client.

This solution not only promises to maintain message order within individual subtrees, ensuring consistency and reliability but also minimizes the impact of errors and operational failures in confined areas of the system. By segmenting the storage structure, Totem can achieve its goal of processing messages with a maximum latency of 2ms while also limiting the blast radius of costly operations, thus enhancing the system's overall performance and scalability.

Chapter 2

Requirement analysis

The chapter serves as the foundational cornerstone of the thesis, explicitly addressing the role of requirement analysis in developing the Totem Archiver service. This stage is crucial for understanding what the Totem Archiver must achieve to facilitate efficient data storage, retrieval, and query functionalities within the specified operational context. It sets the stage for aligning the project's objectives with stakeholder expectations, ensuring that subsequent design and implementation efforts are precisely targeted to meet these articulated needs. By delving into the intricacies of requirement analysis, this section underscores its significance not just as a preparatory step but as a continuous reference point throughout the project lifecycle, guiding design decisions, development priorities, and validation strategies to implement the Totem Archiver successfully.

2.1 Functional requirements

As part of the Totem Archiver analysis, a strategic approach has been adopted to categorize functional requirements into two primary categories: *Read* and *Write* operations. This distinction is made to clarify the system's capabilities and streamline the design and implementation processes. Below is a detailed description of these categories and their subsequent subdivision.

The *Write* category encompasses functionalities related to ingesting and storing data updates from the Totem service, foundational to the Totem Archiver's purpose of ensuring that all data transmitted by Totem is efficiently archived for future access.

The *Read* functionality within the Totem Archiver is strategically divided into two main subcategories to address specific data retrieval needs: retrieval at a precise timestamp and retrieval over a defined time range. This categorization is designed to enhance the system's usability by providing tailored access mechanisms suited to different types of analytical inquiries.

- **Retrieval at a specific time**: This subcategory enables clients to perform queries targeting a specific moment in time. It facilitates the accurate extraction of the data state at critical instances, supporting in-depth historical data analysis and verification.
- **Retrieval within a time range**: Expanding the scope of inquiries, this subcategory allows for the analysis of data evolution over specified intervals. It is crucial for identifying trends, detecting patterns, and conducting thorough evaluations of data changes across time.

2.1.1 Data updates persistence

Requirement description: The Totem Archiver shall possess the capability to persistently store data updates received from the Totem service. Each update, characterized by its unique data path, value, and timestamp, must be efficiently processed and securely archived. This functionality encompasses the Archiver's ability to handle a continuous influx of patches, ensuring they are accurately recorded and stored in a manner that supports quick retrieval and analysis.

Justification: The persistent storage of data updates is paramount for the Totem Archiver, serving as the backbone of its historical data archiving capability. This process enables the Archiver to act as a reliable repository of the state changes in the data managed by the Totem service, thereby facilitating retrospective analyses, compliance audits, and data recovery needs. The efficiency and security of this data persistence process directly impact the Archiver's ability to serve as a dependable source of historical data, ensuring that users can trust the integrity and availability of the archived information over time.

2.1.2 Ingestion of updates and offset management

Requirement description: The Totem Archiver must be designed to ingest updates produced by the Totem service, which are published to Kafka. This functionality includes the capability to reliably consume these updates, ensuring that no data is lost or duplicated in the process. The Archiver must also commit offsets periodically to Kafka, which serves as a checkpoint mechanism to track the consumption progress within the stream of updates. This process is essential for managing the state of data ingestion, allowing the Totem Archiver to resume data consumption from the last committed offset in case of a system restart or failure.

Justification: Efficient and reliable ingestion of updates from Totem is critical for the Archiver's role in capturing and storing data changes over time. Periodic offset committing is fundamental for maintaining the integrity and consistency of the ingestion process. It ensures that the Archiver accurately reflects the state of data in Totem, facilitating precise data recovery and continuity in archival operations. This functionality supports robust data management practices, safeguarding against data loss and enabling seamless system maintenance and recovery procedures.

2.1.3 Retrieve a single data path at specific time or time range

Requirement Description: Empowers users to fetch the value of a particular data path at a designated timestamp or across a defined period, enabling precise and contextual data review.

Justification: This functionality is essential for users who need to understand the state or changes of a specific data element at crucial points in time. By enabling both precise and period-based queries, it supports a wide range of analytical tasks, from audit trails to trend analysis, ensuring users have the necessary tools for detailed data investigation.

2.1.4 Retrieve multiple data paths at specific time or time range

Requirement Description: Facilitates concurrent retrieval of values from multiple data paths for a given timestamp or within a selected timeframe, providing a comprehensive view of related data points.

Justification: Offering simultaneous access to multiple data paths at specific times or over intervals addresses complex analytical needs where relationships or correlations between different data points are scrutinized. This capability is pivotal for multi-dimensional analysis, allowing users to paint a fuller picture of the data environment at critical moments or throughout specified periods.

2.1.5 Wildcard search at specific time or time range

Requirement Description: Supports the use of wildcard patterns for querying data paths, offering flexibility in identifying and extracting data values at specific moments or over periods.

Justification: The ability to perform wildcard searches introduces unparalleled flexibility and efficiency in data retrieval processes. It caters to scenarios where users might not know the exact data paths or need to explore data patterns without rigid constraints. This functionality significantly enhances the system's usability for exploratory data analysis and pattern detection, broadening the analytical capabilities of the Totem Archiver.

2.2 Non-Functional requirements

Non-functional requirements (NFRs) critically shape the Totem Archiver's performance, usability, and reliability, setting standards for how the system operates rather than what functionalities it provides. These requirements ensure the system's efficiency, adaptability, and sustainability, focusing on user experience and system resilience. Addressing NFRs is essential for the Totem Archiver's success, influencing its long-term viability and operational integrity in a dynamic technological landscape.

2.2.1 Performance

Requirement Description: The Totem Archiver must outperform the Totem service in terms of data processing speed despite the added requirement for persistence. NFR includes superior response times and throughput, ensuring data is archived and queryable with minimal latency.

Justification: To provide a seamless user experience and support real-time data analysis needs, surpassing Totem's performance is critical, especially considering the Archiver's role in handling persistent data storage.

2.2.2 Scalability

Requirement Description: The system must be designed to support concurrent operation, allowing for horizontal scaling to accommodate growing data volumes and user demand without degradation in performance.

Justification: As data volumes and the number of users increase, the ability to scale out seamlessly is essential for maintaining system efficiency and responsiveness.

2.2.3 Reliability

Requirement Description: In the event of a system failure, the Totem Archiver must be capable of a rapid recovery, resuming operations from the last known good state without data loss.

Justification: High reliability and minimal downtime are paramount to ensure continuous data availability and integrity, supporting critical decision-making processes.

2.2.4 Data Timeliness

Requirement Description: Data updates produced by the Totem service must be archived and made queryable within 5 minutes of their creation.

Justification: This requirement ensures that the system supports near real-time data analysis, enabling users to make informed decisions based on the most current data.

2.2.5 Cost-Effectiveness

Requirement Description: The development and operational costs of the Totem Archiver should be optimized to deliver a cost-effective solution without compromising on performance and functionality.

Justification: Balancing cost with performance is crucial in a resource-constrained environment, ensuring the solution is sustainable and delivers value for money.

2.2.6 Metrics and monitoring

Requirement Description: Comprehensive metrics and monitoring capabilities must be integrated into the system, providing visibility into its performance, usage, and operational status.

Justification: Effective system monitoring and metrics collection are vital for ongoing performance optimization, troubleshooting, and capacity planning.

2.2.7 Infrastructure

Requirement Description: The Totem Archiver will be deployed on cloud infrastructure, specifically AWS, requiring compatibility with its services and best practices.

Justification: Leveraging cloud infrastructure offers flexibility and scalability but requires careful integration to utilize cloud capabilities fully and efficiently.

2.2.8 Technology

Requirement Description: The system should be user-friendly, easy to maintain, and developed using Scala 3 with the ZIO framework, ensuring smooth integration with existing technologies and infrastructures.

Justification: A focus on usability and maintainability ensures the system is accessible to its intended users and can be efficiently updated and supported over time. The choice of Scala 3 and ZIO framework aligns with modern, functional programming paradigms, offering robustness and developer productivity.

2.3 Use cases

Use cases are an integral part of the system design process, serving as a bridge between user requirements and the functional specifications of a system. They provide a structured and detailed narrative that outlines how users (or other systems) interact with the system to achieve specific goals. By focusing on user actions and system responses, use cases help identify the functionalities a system must possess and clarify the system's expected behavior under various scenarios. This approach not only aids in ensuring that the system meets all user needs and facilitates communication among stakeholders, developers, and designers by providing a common language and reference point.

This section outlines the high-level use cases for the Totem Archiver, focusing on the primary actors involved and the system functionalities they engage with. This elucidation of interactions is crucial for defining the scope of the system, guiding its development, and ensuring it fulfills its intended purposes effectively.

2.3.1 Archiving data updates

Primary Actor: Totem Service

Summary: This use case covers the foundational capabilities of data updates persistence and the management of data stream ingestion, including offset management, by the Totem Archiver. It emphasizes the system's ability to securely store incoming data updates from the Totem service.

Covered Functional Requirements:

- Data updates persistence 2.1.1
- Ingestion of updates and offset management 2.1.2

2.3.2 Precise data retrieval

Primary Actor: Client (User or System)

Summary: Focuses on the client's ability to retrieve data for a specific path or multiple paths at a precise moment or over a defined period, showcasing the Archiver's capability for timely and accurate data retrieval.

Covered Functional Requirements:

- Retrieve a single data path at specific time or time range 2.1.3
- Retrieve a multiple data path at specific time or time range 2.1.4

2.3.3 Flexible data querying

Primary Actor: Client (User or System)

Summary: Highlights the system's adaptability in processing searches with wildcard patterns for data paths at particular times or within set time intervals, allowing for effective data navigation and analysis.

Covered Functional Requirements:

- Wildcard search at specific time or time range 2.1.5

For verifying the fulfillment of all functional requirements, see table 2.1. Each requirement must be covered by at least one use case.

	UC1	**UC2**	**UC3**
FR1	+		
FR2	+		
FR3		+	
FR4		+	
FR5			+

Table 2.1 Relationship between Use Cases (UC) and Functional Requirements (FR)

2.4 Database options

Analyzing the functional and non-functional requirements (FRs and NFRs) of the Totem Archiver leads to a critical discussion on the selection of an appropriate database solution. The chosen database must align with the system's demand for high performance, reliability, scalability, and cost-effectiveness. Here's an exploration of potential database solutions considering these criteria.

Relational databases (RDBMS)

Relational databases offer structured data storage, complex query capabilities, and strong **ACID** compliance [2].

- Atomicity
- Consistency
- Isolation
- Durability

However, while they are highly reliable and support complex transactions, they may face scalability challenges and incur significant costs at scale, especially when deployed in cloud environments. For high-volume, high-velocity data like that of the Totem Archiver, an RDBMS might struggle to maintain performance without substantial optimization and scaling efforts.

NOSQL databases

NoSQL databases represent a flexible and scalable approach to data storage, designed to overcome the limitations of traditional relational database systems. Their schema-less nature is particularly adept at handling unstructured or semi-structured data, making them an efficient choice for accommodating the varied data updates from the Totem service. These databases excel in performance and scalability, catering well to the dynamic and distributed environments often associated with modern web and cloud applications [3].

NoSQL technology stores data in a flexible schema and is easily scalable, addressing the constraints of relational database systems. It is typically designed to utilize multiple servers, making it a frequent choice in distributed settings. There are several types of NoSQL databases, each with unique characteristics:

Document: Similar to key-value stores but store values as documents. A document is a semi-structured entity that groups multiple attributes under one key, rather than storing each attribute under a separate key. This approach facilitates retrieving documents not only by keys but also based on attribute values, offering more flexibility in data querying and organization.

Key-value: Utilize a simple model that allows storing and retrieving values using a unique key. This model's straightforward nature enables fast lookup times, making it suitable for scenarios where quick data access is paramount.

Column oriented: Share some concepts with relational databases, such as rows and columns. A column, the basic unit of storage, contains a name and a value that can be scalar or complex types like sets, lists, or maps. Columns can be grouped into collections of related columns, and a set of columns forms a row. Rows can have identical or different columns without requiring a predefined fixed schema. However, they do not support table joins, which might limit some types of relational operations.

Graph: Store data using nodes (vertices) and relationships (edges) between them. A vertex represents an object with an identifier and a set of attributes, while an edge represents a relationship between two vertices, including attributes related to the relationship. Graph databases excel in scenarios where relationships between data points are as important as the data points themselves.

Time series databases (TSDB)

Given the emphasis on temporal data retrieval in the FRs, time series databases are a compelling option. TSDBs are optimized for storing and querying time-stamped data, offering high write throughput and efficient time-based queries [4]. They are well-suited for handling high volumes of data generated in real-time, as is the case with the Totem Archiver. TSDBs can provide the performance and scalability needed, but their specialized nature may limit general-purpose data management and querying capabilities, potentially impacting cost-effectiveness depending on the specific use case and scale.

Distributed databases and data warehouses

Distributed databases and cloud-native data warehouses offer solutions that inherently support scalability and performance. They can handle large volumes of data across distributed environments, providing the flexibility to scale up resources as needed. While offering significant advantages in terms of performance and scalability, the cost implications of these platforms can vary widely based on the operational load, data volume, and query complexity [5].

Hybdrid and multi-model databases

Hybrid or multi-model databases that combine features of RDBMS, NoSQL, and TSDB could offer a balanced solution, enabling the Totem Archiver to leverage the strengths of each model where it fits best. This approach allows for flexible data modeling, efficient data storage and retrieval, and scalability. Although these databases strive to combine the strengths of various models, they might not always match the performance of specialized single-model databases for certain use cases [6]. Managing such a complex system could introduce overhead and challenges in maintaining cost-effectiveness and operational simplicity.

Polygot persistance

Polyglot persistence is a strategic approach to database architecture where different data storage technologies are used simultaneously to handle varied data types and processing needs efficiently [7]. This method capitalizes on the strengths of multiple database systems to optimize the storage, retrieval, and management of data in complex applications like the Totem Archiver.

Polyglot persistence allows for the selection of specific database technologies tailored to the unique requirements of each data type or transaction pattern within an application. For example, while a time-series database might be optimal for managing time-stamped data generated by the Totem Archiver, a graph database could be better suited for handling complex relationships between data points.

The advantage of using polyglot persistence lies in its flexibility and efficiency. By leveraging the optimal database technology for each aspect of the application's data needs, polyglot persistence can lead to improved performance, scalability, and potentially lower costs. However, this approach also comes with challenges, such as increased complexity in database management and the need for specialized skills to handle diverse database systems effectively. It is crucial to carefully balance these factors to ensure that the benefits outweigh the complexities involved.

2.5 Challenges and considerations

The requirement gathering and analysis phase for the Totem Archiver project presents several challenges, stemming from the nature of the system's expected performance, the anticipated growth in data volume, and budget constraints. Below, we discuss these challenges in detail, along with strategies to address and mitigate them, ensuring a thorough and robust requirement analysis process.

Challenges

Continuous growth of the database
As the database grows due to continuous data ingestion, read and write operations may slow down while the operational costs are poised to increase, potentially straining the project's budget.

Maintaining high performance
Finding a database solution capable of handling the anticipated throughput without incurring exorbitant costs poses a significant challenge, especially given the requirement for the system to outperform the Totem service in terms of data processing and retrieval speeds.

Index inefficiency
Traditional indexing strategies may become less effective as the database grows, impacting query performance. Additionally, the ever-increasing data size directly influences operational costs, presenting a significant challenge in maintaining a cost-effective yet performant system. If indexes are not regularly monitored and maintained, they can become counterproductive, potentially slowing down the database instead of improving its performance. Over time, there exists a tipping point at which the database engine may determine that leveraging an index is less efficient than performing a full table scan.

Strategies for mitigation

Optimizing data storage
Utilize data compression to reduce the physical size of the database, which can lead to cost savings in storage and potentially improve I/O performance for read and write operations.

Leveraging modern database technologies: A polyglot persistence approach to use the most appropriate database technologies for different types of data and access patterns. This can include combining different types of databases together to balance performance and cost.

Conclusion

Addressing the challenges of database management in the Totem Archiver project requires a multifaceted approach, focusing on optimizing data storage, leveraging modern database technologies, and ensuring efficient indexing and query processing. By implementing these strategies, it's possible to mitigate the impact of a continuously growing database on performance and operational costs, ensuring the system remains robust, scalable, and cost-effective. This approach underpins a solid foundation for the requirement analysis process, enabling the project to adapt to evolving needs and technical constraints.

Chapter 3

Design of the totem archiver service

3.1 Database selection for the Totem Archiver service

The Totem Archiver, designed to complement the Totem service by efficiently archiving its data, faces the challenge of managing vast amounts of information while maintaining high performance and cost-effectiveness. This chapter delves into the data model tailored for storing Totem's hierarchical and temporal data, explains the rationale behind our database solution choice, and contrasts it with other considered options.

Data model for storing totem information
To effectively manage the data produced by the Totem service, we devised a data model that captures the essence of its hierarchical structure and the temporal nature of its data. The model is designed to facilitate rapid access to historical data, supporting complex queries that span across different time frames and hierarchical levels. This required a database solution capable of handling rich relationships and large-scale data storage.

Rationale for selection

After careful deliberation, we decided on a polygot database solution that leverages the strengths of Neo4j, a graph database, alongside Amazon S3 for bulk data storage, complemented by Amazon Athena for efficient data querying. This strategic choice was driven by several core commitments:

Efficient data processing and storage:
The Totem service is expected to generate vast amounts of data. Handling this data efficiently, without compromising on speed or accessibility, was a paramount concern. Neo4j offers exceptional performance in managing complex relationships and hierarchies inherent in our data, making it an ideal choice for the relational aspects of our storage needs. For the bulk data generated, Amazon S3 provides a scalable, reliable solution, ensuring we can store large volumes of data without concern for physical hardware limitations.

Cost-effectiveness:
Cost efficiency in managing and storing the projected data volumes was a critical factor in our decision-making process. S3 offers a cost-effective solution for large-scale data storage, with a pricing model that scales with our needs [8]. This allows us to manage costs more predictably as our data grows. Furthermore, Amazon Athena's serverless querying service enables us to perform complex analyses directly on S3-stored data without the need for additional data processing infrastructure, optimizing our operational costs.

High performance in data retrieval:
A key commitment in our selection was to ensure excellent performance in data retrieval. The combination of Neo4j and Athena + S3 addresses this commitment by providing a dual approach to data management. Neo4j enables rapid traversal of connected data points for relationship-heavy queries, while Athena allows for quick, flexible queries on the vast datasets stored in S3. This approach ensures that we can meet the performance expectations for both complex relational queries and large-scale data analyses, providing timely insights to users and maintaining system responsiveness.

Neo4j and Athena + S3

The decision to integrate Neo4j with Athena + S3 represents a balanced approach to meeting the Totem Archiver's requirements. Neo4j's capabilities in graph-based data modeling and querying align with our need to efficiently manage complex data relationships. At the same time, the combination of Athena and S3 offers a powerful, scalable platform for storing and analyzing the extensive datasets generated by the Totem service, ensuring we can access and query our data efficiently and cost-effectively. This custom solution stands as a testament to our commitment to delivering a high-performing, scalable, and cost-efficient archiving service.

Integration approach

Neo4j excels in modeling the intricate relationships inherent in Totem's data, providing a flexible and intuitive framework for representing hierarchical structures. By storing only metadata and relationships in Neo4j and offloading the actual data payloads to S3, we achieve significant cost savings and scalability.
This setup allows us to exploit the best of both worlds: Neo4j's powerful querying capabilities for relationship-heavy operations and S3's virtually unlimited and cost-effective storage space.

Pros and cons of the chosen solution

Advantages

- **Scalability and cost-effectiveness:** By offloading the primary data storage to S3, we significantly mitigate scalability concerns and reduce operational costs, leveraging S3's efficiency in handling large data volumes.
- **Efficient handling of data relationships:** Neo4j's graph model naturally aligns with the hierarchical and relational nature of Totem's data, enabling sophisticated queries and analyses.

Disadvantages

- **Introduction of new technology:** Adopting Neo4j represents a learning curve and integration challenge, as it introduces a new technology into our existing technology stack, requiring new expertise and potentially affecting development timelines.
- **Complexity in data synchronization:** Managing data across Neo4j and S3 adds complexity to the architecture, necessitating robust synchronization mechanisms to ensure data integrity.

Honorable mentions

Postgres and TimescaleDB

Despite Postgres and TimescaleDB being key components of our current technology stack, they were not chosen as the main storage solutions for the Totem Archiver. The anticipated volume of data from the Totem service posed scalability and cost challenges for these databases. Specifically, TimescaleDB and Postgres' proportional scalability could result in performance bottlenecks as data volumes increase. Moreover, the expenses related to scaling up, ensuring high availability, and managing large datasets might become excessively high. Therefore, we opted for a polygot approach that better aligns with the Totem Archiver's needs to address these concerns while ensuring the system remains scalable and cost-effective.

3.2 Totem Archiver API

The Totem Archiver introduces a RESTful API designed to facilitate efficient data updates and queries. A key feature of this API is:

- **POST /api/v1/update-info**

This endpoint is crafted to handle complex data path updates with temporal validity. Its versatility and adaptability to process multiple data paths within a single request stand out.

3.2.1 Request description

The request body 3.1 is a `JSON` object with the following structure:

```
{
  "flexiblePaths": [
    [
      "segment1", // optional
      "segment2", // optional
      ... // additional segments
    ],
    ... // additional flexible paths
  ],
  "validFrom": "YYYY-MM-DDTHH:mm:ss.XXXZ",
  "validTo": "YYYY-MM-DDTHH:mm:ss.XXXZ" // optional
}
```

Code listing 3.1 API request body

- `flexiblePaths`: An array of arrays representing the flexible paths to be updated.
  - Each inner array represents a single path.
  - Each element within the inner array represents a segment in the path.
    - A string value specifies a literal segment.
    - `null` acts as a wildcard segment, matching any value.
- `validFrom`: A timestamp representing the start time of validity for the update.
- `validTo`: A timestamp representing the validity end time for the update (exclusive).

Request example

Code listing 3.2 showcases an example of a primitive request body, for two paths:

- `["a", "b", "c"]`
- `["b", null, "c"]` (where the second segment acts as a wildcard)

```
{
  "flexiblePaths": [
    ["a", "b", "c"],
    ["b", null, "c"]
  ],
  "validFrom": "1970-01-01T00:00:05.000Z",
  "validTo": "1970-01-01T00:00:10.000Z"
}
```

Code listing 3.2 API request example

3.2.2 Response description

The response 3.3 is a `JSON` object with the following structure:

```
"updateInfos": [
{
  "path": [
    "segment1",
    "segment2",
    // additional segments
  ],
  "pathInfo": [
    {
      "value": "TYPE(VALUE)",
      "validFrom": "YYYY-MM-DDTHH:mm:ss.XXXZ",
      "validTo": "YYYY-MM-DDTHH:mm:ss.XXXZ"
    },
    ... // additional objects
  ]
},
... // information for other paths
]
```

Code listing 3.3 API response body

- `updateInfos`: An array containing information about each path.
  - Each element represents information for a single path.
- `path`: A specific path composed from segments.
- `pathInfo`: An array containing information for each update applied to the path.
  - Each element represents a single update.
  - `value`: The updated value (a string, integer, etc., depending on the data type).
  - `validFrom`: The start time of validity for the update.
  - `validTo`: The end time of validity for the update.

Response example

Code listing 3.4 shows an example of a primitive response body.

```
"updateInfos": [
{
  "path": ["a", "b", "c" ],
  "pathInfo": [
    {
      "value": "STRING(Initialized)",
      "validFrom": "1970-01-01T00:00:05.000Z",
      "validTo": "1970-01-01T00:00:07.123Z"
    },
    {
      "value": "STRING(Processed)",
      "validFrom": "1970-01-01T00:00:07.123Z",
      "validTo": "1970-01-01T00:00:08.491Z"
    },
    {
      "value": "STRING(Cleared)",
      "validFrom": "1970-01-01T00:00:08.491Z",
      "validTo": "1970-01-01T00:00:10.000Z"
    },
  ]
},
{
"path": ["b", "x", "c" ],
  "pathInfo": [
    {
      "value": "INT(42)",
      "validFrom": "1970-01-01T00:00:05.000Z",
      "validTo": "1970-01-01T00:00:05.789Z"
    },
    {
      "value": "INT(-1)",
      "validFrom": "1970-01-01T00:00:09.562Z",
      "validTo": "1970-01-01T00:00:10.000Z"
    },
  ]
  }
]
```

Code listing 3.4 API response body

A few observations from this response body can be made:

I. Identification of paths matching a wildcard search:

The response indicates that for a wildcard search pattern of `["b", null, "c"]`, only one path, `["b", "x", "c"]`, matches this criterion within the specified time range.
This observation suggests that at the time of the query, there were no other paths in the system that fit the format `["b", null, "c"]`. If there were any other paths matching this pattern, they must have been updated outside of the queried time frame, specifically after the latest `validTo` timestamp mentioned in the response.

II. A gap in path updates:

We observed a gap in path updates.
Between timestamps `1970-01-01T00:00:05.789Z` and `1970-01-01T00:00:09.562Z`. During this period, the path still existed in the system, but its value was null. This indicates that a value associated with the path was deleted at some point within this timeframe.

III. Synchronization of query range with update validity:

The response data ensures that the first update in the sequence for any given path begins exactly at the `validFrom` timestamp specified in the query, and similarly, the last update in the sequence concludes at the `validTo` timestamp requested. This alignment indicates a precise filtering mechanism employed by the system to return updates. It directly correlates the query's time range with the validity period of the updates, ensuring that the provided updates span the entire requested timeframe without exceeding or falling short of it. This behavior is crucial for several reasons:

- **Precision in data retrieval:** It guarantees that the updates returned are strictly within the bounds of the requested timeframe, providing a complete and accurate snapshot of the changes to each path during the specified interval.
- **Consistency in reporting:** By aligning the updates' validity directly with the query parameters, the system ensures consistent and predictable reporting behavior, which is particularly valuable for time-sensitive analyses and auditing purposes.

Conlcusion

Totem Archiver's API exemplifies the commitment to providing a robust, flexible, and user-friendly interface for managing historical data.

3.3 Technology stack

This chapter outlines the technology stack behind the Totem Archiver service, briefly describing the selected technologies, their roles, and their integration. We detail why each technology was chosen. This overview offers insights into the strategic technology decisions underpinning the Totem Archiver's architecture.

3.3.1 Scala

For the development of the Totem Archiver service, we chose Scala 3.3.3, an evolution from the previously used Scala 2.13, showcasing significant syntactical advancements and establishing itself as the new standard in our Scala development practices. This choice not only reflects our commitment to leveraging cutting-edge technology but also aligns with Scala's compatibility and efficient operation on the Java Virtual Machine (JVM). Scala's unique combination of functional programming paradigms [9] with object-oriented principles [10] allows for a more expressive, concise, and maintainable codebase, which is particularly beneficial for the complex processing requirements of the Totem Archiver.

Key features of Scala, such as case classes for seamless data modeling, pattern matching for clearer logic expressions, and a strong emphasis on immutability, support safer concurrency and facilitate a more straightforward approach to managing program state [11]. These features are instrumental in achieving the high performance and reliability goals set for the Archiver service.

3.3.1.1 ZIO

The Totem Archiver service leverages ZIO, a powerful Scala library for building concurrent, asynchronous, and effectful applications. ZIO offers a robust foundation for managing complex workflows and interactions with external systems, making it an ideal choice for the Archiver's architecture.

Key concepts:

- **Effects:** ZIO encapsulates side effects, such as network calls, file I/O, and logging, within a managed context [12]. This promotes a clear separation of concerns between pure computations and side effects, leading to more predictable and testable code.
- **Fiber-based concurrency:** ZIO utilizes lightweight fibers for concurrent execution of tasks [13]. Fibers offer greater efficiency compared to traditional threads, allowing the Archiver to handle a high volume of concurrent operations with minimal overhead.
- **Functional programming style:** ZIO embraces a functional programming approach, encouraging the use of immutable data structures and pure functions. This fosters code that is easier to reason about, debug, and maintain, especially in a concurrent environment, due to **referential transparency**. Referential transparency guarantees that a function always produces the same output for the same input, regardless of external factors [14]. This predictability makes ZIO functions highly **testable**.

Benefits:

- **Error handling:** ZIO provides a structured approach to error handling through its error channeling mechanism [15]. This empowers developers to gracefully handle and propagate errors throughout the application, resulting in a more robust and resilient system.
- **Composable effects:** ZIO effects are composable, meaning they can be combined and chained together to build complex workflows [16]. This composability simplifies the construction of intricate data pipelines within the Archiver service.
- **Testability:** ZIO's core design principles, particularly its effect isolation and emphasis on pure functions, make writing unit tests exceptionally straightforward. Tests can focus solely on the function's logic without worrying about external dependencies or side effects.

Why was ZIO used?

The Totem Archiver service deals with concurrent data processing, interacts with various external systems, and requires robust error handling. ZIO's core features, particularly its effect management, lightweight concurrency, and functional programming style, perfectly align with these requirements. Additionally, ZIO's focus on referential transparency and composability significantly simplifies testing, ensuring a high degree of code quality and maintainability.

While other libraries like Akka were considered for building concurrent applications, ZIO's emphasis on testability was a major deciding factor. Akka, while powerful, can introduce complexities when it comes to testing due to its actor-based model. ZIO's functional approach and effect isolation streamline the testing process, allowing developers to write clear and concise unit tests for the Archiver service.

3.3.1.2 JsonIter

The Totem Archiver service incorporates Jsoniter, a high-performance JSON parsing library, for efficiently handling data serialization and deserialization tasks. Jsoniter offers a significant performance advantage over commonly used alternatives like Circe or Spray-json, making it an ideal choice for applications like the Archiver that deal with large volumes of JSON data.

Even Though both Circe and Spray-json are well-established JSON libraries in the Scala ecosystem, Jsoniter excels in terms of raw speed [1]. This performance boost is crucial for the Archiver service, where timely processing of JSON data is essential for maintaining a high level of throughput.

3.3.1.3 Sttp client

We rely on Sttp Client, a mature and widely adopted Scala library [18], for making HTTP requests within the Totem Archiver service. Sttp offers a comprehensive feature set for building robust and efficient HTTP interactions, making it the natural choice for our development needs.

Given Sttp's extensive use across various internal projects within the company, adopting it for the Archiver service fostered code reuse and consistency. Sttp's established position within

[1] See [17] for reference.

our development ecosystem and its richness in features made exploring alternative libraries less practical. This standardization not only simplifies maintenance efforts and leverages existing expertise within the company when managing HTTP communications.

3.3.1.4 Tapir

The Totem Archiver service utilizes Tapir for API definition and documentation generation. Tapir is a Scala library that facilitates the creation of robust and well-documented APIs [19].

Tapir focuses on defining API endpoints using a type-safe approach, promoting code clarity and reducing errors. Additionally, Tapir automatically generates API documentation, streamlining communication between developers and consumers of the Archiver service's functionalities.

3.3.2 S3

The Totem Archiver service leverages Amazon Simple Storage Service (S3) as its primary cloud storage solution. S3 is a scalable and reliable object storage service offered by Amazon Web Services (AWS) [20].

S3 provides a cost-effective and highly available platform for storing the vast amount of data archived by the Totem Archiver service. Here is why S3 was an ideal choice:

- **Pay-as-You-Go:** S3 utilizes a pay-as-you-go pricing structure, eliminating the need for upfront investments in physical storage infrastructure [21]. This aligns perfectly with the Archiver's needs, as we only pay for the storage space we use.
- **Scalability:** S3 boasts immense scalability, allowing us to seamlessly scale storage capacity as the volume of archived data grows [21]. This scalability ensures that the Archiver can accommodate future growth without storage limitations.
- **Reliability:** As an AWS service, S3 offers exceptional reliability and durability. The data archived within S3 is replicated across multiple geographically dispersed facilities, minimizing the risk of outages and data loss [21]. This reliability is crucial for ensuring the long-term preservation of the archived information.

By leveraging S3's cost-effectiveness, scalability, and reliability, the Totem Archiver service benefits from a robust and future-proof storage solution without the burden of managing physical disk space or worrying about data integrity.

Parquet

The Totem Archiver service leverages Apache Parquet, a columnar data storage format, to optimize archived data for space efficiency and rapid retrieval [22]. Parquet stores data in a column-oriented manner, unlike the traditional row-based approach of CSV files. This columnar organization allows for efficient compression of individual data columns, leading to significant reductions in storage requirements compared to uncompressed formats.

This space efficiency translates to cost savings for the Totem Archiver service. By minimizing the amount of data stored, we optimize our cloud storage utilization on Amazon S3 3.3.2.

Parquet supports various compression codecs, including *Gzip*, *Snappy*, *Brotli*, *Zstd*, and *LZ4* [23]. We thoroughly evaluated these codecs within the context of the Archiver service's specific needs. The evaluation focused on two key metrics: compression ratio (amount of space saved) and processing speed (compression and decompression times).

Our testing revealed that all the codecs achieved very similar results in terms of compression ratio and decompression speed. *Snappy*, however, emerged as the most favorable choice due to its:

- **Industry standard:** Widely adopted as a compression format within the extensive data ecosystem, offering compatibility advantages across various tools and technologies.
- **Balanced performance:** Offers a good balance between compression ratio and processing speed, making it suitable for our use case where both space efficiency and performance are critical.
- **Simplicity:** Relatively simple compression algorithm, which can be advantageous regarding resource consumption.

Therefore, Snappy compression was chosen as the default compression format for Parquet files within the Totem Archiver service. This selection ensures efficient storage utilization while maintaining acceptable data archiving and retrieval processing speeds.

3.3.3 Athena

The Totem Archiver service integrates Amazon Athena, a serverless interactive query service from Amazon Web Services (AWS), to empower users with efficient and scalable data exploration capabilities. Athena allows us to query the Parquet files directly archived in Amazon S3 (section 3.3.2) using familiar SQL syntax, eliminating the need for complex data movement or transformation steps before analysis [24].

Why athena?

Athena is a perfect match for our needs as it seamlessly aligns with our core technology principles:

- **Scalability:** Athena's serverless architecture removes the burden of infrastructure management, allowing us to scale query processing effortlessly as data volume grows.
- **Cost-effectiveness:** We only pay for the queries we execute, making Athena a cost-optimized solution for ad-hoc data exploration on our vast archive.
- **Seamless integration:** Native integration with S3 allows Athena to directly query archived data without data movement, minimizing latency and facilitating real-time insights.

External tables and Hive partitioning

A key benefit of using Athena with Parquet files is the ability to create external tables. These tables reference data directly within S3, enabling Athena to query it without physical data movement.

Furthermore, Athena supports Hive partitioning, a technique for organizing data files based on timestamps [25]. In the Totem Archiver service, we leverage minute-granularity Hive partitioning, where data resides in folders structured by year (YYYY), month (MM), day (DD), hour (HH), and minute (mm). This approach significantly optimizes query performance and fulfills the fourth non-functional requirement (section 2.2.4):

- **Faster results:** When users submit queries for specific timeframes, Athena can efficiently locate relevant data segments based on the partition path, minimizing data scanned and accelerating query execution.
- **Precise data access:** Minute-granularity partitioning ensures users can conduct highly specific queries, retrieving data with the desired level of detail. This empowers in-depth data analysis and extraction of valuable insights from the archived information.

By integrating Athena's capabilities with our data storage strategy, the Totem Archiver service provides a powerful and user-friendly platform for interactive data exploration. Users can leverage familiar SQL syntax to unlock the valuable knowledge stored within the archive, fostering informed decision-making and maximizing the utility of the archived data.

3.3.4 Neo4j

The Totem Archiver service relies heavily on Neo4j, a powerful and versatile graph database. Unlike traditional relational databases that organize data in tables with rows and columns, Neo4j excels at representing interconnected information – precisely what our totem data entails.

Neo4j's true value lies in its ability to model complex relationships between data entities [26]. In our case, these entities represent the various segments that make up a totem path. By leveraging Neo4j's graph structure, we can efficiently create and store these hierarchical relationships, essentially forming a digital representation of the tree-like structures inherent to totem data.

Furthermore, Neo4j empowers us to create indexes over our S3 storage. These indexes act as efficient lookup mechanisms, allowing us to pinpoint the precise location of specific totem paths within the vast S3 data lake. This significantly reduces retrieval times when querying the archive.

Our approach involves storing the entire totem path within Neo4j, with updates happening at a granular level within minute buckets. If no update occurs within a minute, the path remains unchanged. However, if an update is detected, Neo4j efficiently appends the new information to the corresponding leaf node. These leaf nodes essentially hold timestamps, serving as a record of the path's first occurrence within each minute bucket.

It's important to note that while nodes store the actual segment values, relationships within the graph utilize hashed node values. This optimization technique enhances traversal efficiency within the Neo4j graph, allowing for faster retrieval of specific totem paths. A deeper dive into the specifics of this hashing mechanism will be explored in the implementation phase (section 4.2.4).

3.3.5 Postgres

The Totem Archiver service leverages PostgreSQL, a powerful, open-source object-relational database management system (ORDBMS), as its secondary data storage solution. PostgreSQL boasts a rich feature set, exceptional reliability, and a strong reputation within the database community [27].

Core database and familiarity

PostgreSQL is the cornerstone of our company's data storage infrastructure. This widespread adoption translates to a deep understanding and expertise with PostgreSQL among our development teams. Utilizing PostgreSQL for the Archiver service capitalizes on this existing knowledge base, streamlining development and ongoing maintenance efforts.

Supportive data and data integrity

PostgreSQL will store critical `supportive data` that holds significant informational value within the Totem Archiver service. This data encompasses essential details such as timestamps of totem outages. Preserving this information is paramount as it underpins a core concept in data integrity: temporal validity.

Temporal validity refers to the ability to determine data validity at a specific point in time. The Archiver service establishes a clear timeline by recording totem outage timestamps, differentiating between valid and invalid data during outage periods. This distinction is crucial for ensuring the trustworthiness and reliability of the archived data.

In essence, PostgreSQL serves as a robust foundation for storing and managing the Totem Archiver service's supportive data.

3.3.6 Kubernetes

For orchestrating several of our backend services, including the critical Neo4j graph database, we leverage Kubernetes. Kubernetes is an open-source system designed to automate the deployment, scaling, and management of containerized applications [28].

Unlocking scalability and efficiency

Imagine an orchestra conductor skillfully managing a complex symphony. Kubernetes functions similarly for containerized applications. It groups containers, which are lightweight, self-contained units of software, into logical units called pods. Kubernetes then expertly manages these pods across a cluster of machines, ensuring they run efficiently and reliably. This containerization approach offers significant benefits, particularly for Neo4j:

- **Effortless scaling:** Kubernetes empowers us to effortlessly scale services like Neo4j up or down based on demand. This elasticity ensures Neo4j can handle fluctuating workloads effectively, whether processing a surge of data or operating during periods of lower activity.

- **Simplified deployments:** Containerized applications are inherently portable. This means Neo4j can be easily deployed across different environments without modification. This simplifies deployments and streamlines management across our infrastructure.
- **Resource optimization:** Kubernetes optimizes resource utilization by efficiently allocating resources to containerized applications like Neo4j. This translates to a more efficient use of our computing infrastructure, reducing costs and maximizing resource availability.

By leveraging Kubernetes for orchestration, Neo4j gains a robust and scalable foundation. This approach fosters efficient resource utilization, simplified deployments, and the ability to seamlessly scale Neo4j to meet evolving needs.

3.3.7 ECS

The Totem Archiver service leverages Amazon ECS (Elastic Container Service), a managed container orchestration service offered by Amazon Web Services (AWS). ECS simplifies the deployment, scaling, and management of containerized applications, making it an ideal choice for hosting the Totem Archiver within the AWS cloud environment.

As a managed service, ECS takes care of the heavy lifting associated with container orchestration. This frees up our development team to focus on core functionalities of the Archiver service, rather than managing the underlying infrastructure. Benefits of utilizing ECS for the Totem Archiver include:

- **Simplified deployment and management:** ECS offers a user-friendly interface and automated features that streamline the deployment and ongoing management of the containerized Archiver service.
- **Seamless integration with AWS services:** Since the Totem Archiver resides within the AWS cloud, utilizing ECS fosters tight integration with other AWS services we leverage. This cohesive environment simplifies data flows and streamlines overall system operation.
- **Scalability on demand:** ECS empowers us to effortlessly scale the Archiver service up or down based on processing needs [29]. This elasticity ensures the Archiver can handle fluctuating workloads efficiently.

By employing ECS for orchestration, the Totem Archiver benefits from a robust, scalable, and AWS-integrated platform. This approach ensures efficient management, seamless integration with surrounding services, and the ability to adapt to changing processing demands.

3.3.8 Apache Kafka

The Totem Archiver service utilizes Apache Kafka, a distributed streaming platform [30], as the cornerstone of its data ingestion pipeline. Kafka acts as a high-throughput message queue, ensuring reliable and scalable data delivery from the Totem Service, an external component, to the Archiver service.

Efficient data flow

The Totem Service publishes data batches, consisting of multiple ”patches” that contain path and value information, as messages onto a specific Kafka topic. This topic serves as a dedicated channel for data exchange between the Totem Service and the Archiver service.

Consuming and processing with offsets

The Archiver service subscribes to the designated topic as a Kafka consumer. As the Totem Service publishes messages, the Archiver service efficiently consumes them, acknowledging successful processing through a mechanism known as commits.

These commits leverage offsets, a concept within Kafka that tracks the consumer's progress in consuming messages from a topic. When the Archiver service commits an offset, it essentially signifies that it has successfully processed all messages up to that specific point within the Kafka topic.

Reliable delivery and guaranteed processing

Kafka's robust architecture ensures at least once delivery semantics. This means that each message published by the Totem Service is guaranteed to be delivered to the Archiver service at least once.

Furthermore, by committing offsets after successful data storage in S3, Neo4j, and PostgreSQL, the Archiver service ensures that it does not reprocess already archived data. This mechanism prevents data duplication and streamlines the overall data ingestion process.

Apache Kafka serves as a critical intermediary, facilitating a reliable and scalable data flow from the Totem Service to the Archiver service. Utilizing offsets empowers the Archiver to commit its progress, guaranteeing successful data processing and preventing duplicate archiving.

3.4 Architectural overview

This section presents a high-level overview of the Totem Archiver service's architecture 3.1, detailing the interacting components, data flow, communication mechanisms, and how the chosen technologies contribute to the overall design.

External data source (Totem service):

Sends batches of patches containing path and value information to Apache Kafka, a message broker.

Write component (Write Service):

- Consumes messages from Apache Kafka.
- Enriches the data by adding `validFrom` and `validTo` timestamps.
- Maintains a local cache that aggregates data within a one-minute window.

- Processes updates:
  - If the update is older than the current minute threshold (outdated):
    - Writes the aggregated data for the minute to a Parquet file using the File Service.
    - Uploads the Parquet file to Amazon S3 using the Persistence Service.
    - Updates an index in Neo4j with only the first occurrence of distinct paths within the minute.
  - If the update is within the current minute threshold:
    - Updates the local cache for the current minute.

  In case of outages, persists supportive data (timestamps) in PostgreSQL.

Shared storages:

- Neo4j: A graph database used to model the hierarchical relationships.
- Amazon S3: An object storage service used to store the archived Parquet files.
- PostgreSQL: A relational database used to store critical supportive data for data integrity.

Read component:

- Receives data retrieval requests from an API.
- Utilizes Neo4j to locate the specific paths within S3 based on the request.
- Constructs appropriate queries to retrieve relevant information from S3 using Amazon Athena, an interactive query service for S3 data.
- Fetches additional supportive data from PostgreSQL to ensure data integrity and validity of the retrieved information.
- Formats the data by merging intervals, trimming overflowing timestamps, filtering out null values, and integrating supportive data.

Overall architecture:

This architecture leverages Apache Kafka for efficient message queuing, enriches and aggregates data within the Write Service, utilizes Neo4j for efficient path lookups, stores Parquet files in S3 for long-term storage, and retrieves data using Athena queries on S3. PostgreSQL serves as a secondary storage for critical metadata.

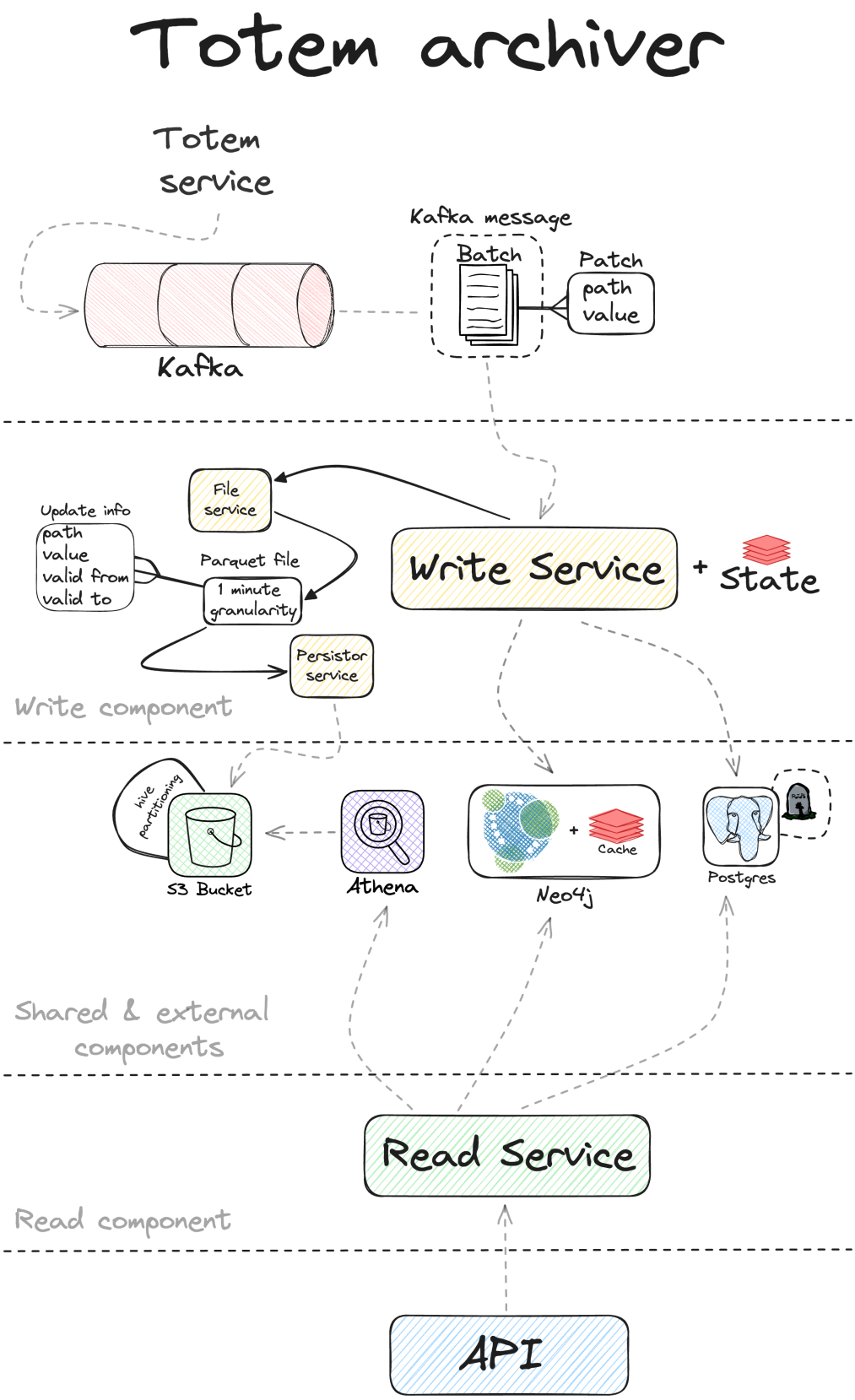

Figure 3.1 Simplified architectural overview of the Totem Archiver

Chapter 4

Implementation

This chapter details the implementation of the Totem Archiver service. It explores the key steps and decisions that transformed initial concepts into a working system. The focus lies on component integration, architectural choices, and programming strategies.

Code snippets illustrate implementations and challenges encountered. Problem-solving approaches and lessons learned are discussed, offering valuable insights for future projects.

This chapter is structured around the two main components of the Totem Archiver service: the read and write components as they could potentially be split into two independent micro-services. It is organized in a manner that mirrors the flow of data through the system, providing a logical progression that aids in understanding. This approach ensures that readers are equipped with the foundational knowledge required to grasp more advanced concepts discussed later in the text.

Finally, the chapter reflects on the implementation process, contributing practical knowledge to software engineering. By documenting successes and setbacks, it offers a balanced perspective for future developers tackling similar complexities. The discussion goes beyond technical details, including strategic insights and best practices from iterative development and testing.

4.1 Building blocks

This subsection lays the foundation for understanding more complex structures within the Totem Archiver. It delves into the core building block, the `UpdateInfo` class, and explains its key components in detail.

Path

This section introduces the concept of an opaque type within the Totem Archiver service, specifically the `Path` type. An opaque type (a feature introduced in Scala 3) offers a layer of abstraction by hiding its underlying implementation details while maintaining compatibility with its upper bound.

Listing 4.1 showcases the formal definition of the Path type using Scala's opaque type syntax:

```
opaque type Path <: Vector[String] = Vector[String]

object Path:
 def apply(segments: Vector[String]): Path = segments
```

Code listing 4.1 Path

Here, `Path` is declared as an opaque type with an upper bound of `Vector[String]`. This ensures that any function expecting a `Vector[String]` argument can also accept a Path argument without compilation errors.

The `Path` type plays a crucial role within the Totem Archiver service, frequently appearing in various components. By defining it as an opaque type, the code benefits from several advantages:

- **Improved readability:** Using a dedicated type like `Path` enhances code readability by explicitly conveying its purpose and domain-specific meaning compared to the more generic `Vector[String]`.
- **Enhanced maintainability:** An opaque type promotes better code maintainability. If the internal representation of Path needs modification in the future, such changes remain encapsulated within the type definition, minimizing potential ripple effects throughout the codebase.
- **Overall code quality:** The use of opaque types fosters a more robust and well-structured codebase, contributing to improved overall code quality.

UpdateInfo

Listing 4.2 showcases the `UpdateInfo` case class, which serves as the domain model for a patch. Patches are the fundamental building blocks of both Totem and the Totem Archiver. Here's a breakdown of its essential fields:

```
case class UpdateInfo(path: Path, value: Option[String], validFrom: Instant, validTo:
    ↪ Option[Instant] = None)
```

Code listing 4.2 Update Info

- **path** (`Path`): (refer to code listing 4.1 for details on Path) This field represents a string vector (segments) defining a path within a tree structure. The order of these segments is crucial, with the last element being the leaf node. As you move towards the vector's beginning, you traverse layers upwards in the tree.
- **value** (`Option[String]`): This optional field indicates the value associated with the specified path. While the declared type is String, the actual value can be stored as a string representation of different data types like integers (e.g., `INT(42)`) or booleans (e.g., `BOOL(true)`)
- **validFrom** (`Instant`): This field specifies the timestamp from which the provided value for the path becomes valid.
- **validTo** (`Option[Instant]`): This optional field denotes the timestamp until which the value remains valid. Initially, when creating an UpdateInfo object, the `validTo` value is unknown. It's typically set in subsequent patches for the same path, effectively setting the previous patch interval validity.

4.2 Write component

This section delves into the intricate process of data ingestion, processing, and archiving within the Totem Archiver service. This multi-step journey leverages various services to ensure efficient and reliable data handling. We'll progressively unveil these services, explaining their specific roles in the overall workflow.

4.2.1 Apache Kafka

Initially, it appeared we might need to develop our own decoder for data deserialization. Fortunately, we didn't have to pursue this route. The team responsible for maintaining the Totem service provided a client that significantly simplifies the process. This client takes a decoder and a configuration containing all the necessary settings to connect to the Apache Kafka topic. They even supplied a pre-defined decoder implementation, sparing us the complexity of dealing with MessagePack decoding, which would have added another layer of complexity.

```
1  object StorageEventReader extends CommonMessagePackReader {
2
3   private implicit val updatedItemReader: Reader[UpdatedItem] = product4Reader(
       ↪ UpdatedItem.apply)
4
5   private val initializedReader: Reader[Initialized] = product1Reader(Initialized.apply)
6   private val shutdownReader: Reader[Cleaned] = product1Reader(Cleaned.apply)
7   private val tickedReader: Reader[Ticked] = product1Reader(Ticked.apply)
8   private val updatedReader: Reader[ItemsUpdated] = product3Reader(ItemsUpdated.apply)(
       ↪ implicitly, seqReader[Seq[UpdatedItem], UpdatedItem], implicitly)
9
10  val reader: Reader[StorageEvent] = enumReader(
11   StorageEventDiscriminator.initialized -> (initializedReader <* skippingReader),
12   StorageEventDiscriminator.shutdown -> (shutdownReader <* skippingReader),
13   StorageEventDiscriminator.ticked -> (tickedReader <* skippingReader),
14   StorageEventDiscriminator.updated -> (updatedReader <* skippingReader),
15  )
16 }
```

Code listing 4.3 Storage Event Reader

Code listing 4.2.1 details the implementation of the `StorageEventReader` class. This reader utilizes a discriminator approach to identify the message type based on the first character of the message sequence. The service can distinguish between four message types:

1. **Initialized**: This message signifies the initial startup of the Totem service.
2. **Shutdown**: This message indicates a graceful shutdown of the Totem service.
3. **Ticked**: This message acts as a heartbeat, sent every 10 seconds (configurable).
4. **Updated**: This message represents a batch of data updates (refer to figure 3.1 for a high-level overview)[1].

[1] Note: Only the most pertinent message type for our use case is illustrated in the figure.

4.2.2 File service

This section explores the interaction between the Totem Archiver and file service (refer to code listing 4.4 for the API definition). The file service is responsible for serializing instances of the `UpdateInfo` class (Listing 4.2) into Parquet files.

The Totem Archiver leverages the `parquet4s`[2] library to handle serialization of `UpdateInfo` objects into Parquet format. This library provides a robust solution by utilizing the underlying Hadoop client. However, an initial challenge arose due to a logging backend incompatibility between parquet4s and the Totem Archiver application. This incompatibility caused difficulties during the development phase.

To address the logging incompatibility mentioned above, a solution was implemented involving the exclusion of the conflicting logging dependency from the `parquet4s`. The introduction of a bridge component followed this. This bridge acts as a translation layer, effectively mediating communication between the two disparate logging backends, ensuring seamless operation.

```
1 trait FileService:
2   def create(filename: String): Task[Path]
3   def write(filePath: Path, data: Vector[UpdateInfo]): Task[Unit]
4   def delete(filePath: Path): Task[Boolean]
```

Code listing 4.4 File Service

The file service API is straightforward and doesn't have much complexity. However, readers might be curious about how the data is serialized, which is a bit more challenging.

```
1  private object ParquetCodecs:
2    given ParquetRecordEncoder[UpdateInfo] = (entity: UpdateInfo, resolver:
       ↪ EmptyRowParquetRecordResolver, configuration: ValueCodecConfiguration) =>
3      RowParquetRecord(
4      "path" -> BinaryValue(entity.path.serializePath),
5      "value" -> entity.value
6        .map(BinaryValue.apply)
7        .getOrElse(NullValue),
8      "valid_from" -> LongValue(entity.validFrom.toEpochMilli),
9      "valid_to" -> entity.validTo
10       .map(instant => LongValue(instant.toEpochMilli))
11       .getOrElse(NullValue)
12     )
13   given ParquetSchemaResolver[UpdateInfo] = (cursor: Cursor) =>
14     List(
15     PrimitiveType(
16       Repetition.REQUIRED,
17       PrimitiveType.PrimitiveTypeName.BINARY,
18       "path"
19       ),
20     PrimitiveType(
21       Repetition.OPTIONAL,
22       PrimitiveType.PrimitiveTypeName.BINARY,
23       "value"
24       ),
25     PrimitiveType(
```

[2] https://github.com/mjakubowski84/parquet4s

```
26        Repetition.REQUIRED,
27        PrimitiveType.PrimitiveTypeName.INT64,
28        "valid_from"
29        ),
30      PrimitiveType(
31        Repetition.OPTIONAL,
32        PrimitiveType.PrimitiveTypeName.INT64,
33        "valid_to"
34        )
35      )
```

Code listing 4.5 ParquetCodecs

Since `UpdateInfo` (code listing 4.2) possesses a unique structure and encoding, a custom Parquet codec was implemented for its efficient serialization. Here, we discuss several key considerations made during the codec's development:

Path serialization:

The Path field within `UpdateInfo` is serialized using the `JsonIter` library (Refer to subsection 3.3.1.2). Initially, an escaping approach was considered. However, this method proved to be less robust and offered minimal benefits in terms of storage space, particularly when data compression is employed. Since data compression is a standard practice within the Totem Archiver, the escaping approach's potential space savings were negligible

- **Timestamps serialization:** The `validFrom` and `validTo` fields within `UpdateInfo` are serialized as longs (INT64) instead of timestamps. This selection prioritizes storage efficiency and compression effectiveness. Additionally, this approach aligns with the serialization strategy for similar data types within the Neo4j database used by the Totem Archiver. Consistent type usage across the database stack enhances code readability and maintainability. Notably, this choice of INT64 serialization proves particularly beneficial in the long term, especially considering the potential lack of data compression within Neo4j.
- **Naming:** The Totem Archiver service adheres to custom field naming conventions to enhance code readability and maintainability. While terms like "path" and "value" might be commonly used as keywords, specific prefixes provide additional context.
- **Prefixing for timestamp fields:** Timestamp fields within the `UpdateInfo` class (see code listing 4.2) utilize the `valid_` prefix (e.g., `valid_from`, `valid_to`). This prefix explicitly conveys the purpose of these fields, differentiating them from standard timestamp objects and highlighting their role in denoting data validity periods.

  These naming conventions promote consistency throughout the codebase. Prefixes like `valid_` not only improve readability but also align with standard database syntax (`snake_case`) used by Athena queries, ensuring seamless integration with that platform. This consistency fosters improved code maintainability and reduces potential confusion for developers working on the Totem Archiver service.

4.2.3 File persistor service

The File Persistor Service features a concise API (refer to code listing 4.6) with a single primary method. Due to the service's dependency injection architecture, implementation is straightfor-

ward. Of particular interest is the S3 key construction mechanism (See code listing 4.7)

```
1 trait FilePersistor:
2  def persistFile(path: Path, destination: Destination): IO[Error, Unit]
```

Code listing 4.6 File Persistor Service

The key is generated using the `env` and `destination` variables (see lines 5-12 in code listing 4.7). The `env` enum represents the service's runtime environment (dev, test, stage, or prod). The `destination` case class designates the precise S3 storage location. This mechanism's simplicity will be explored in a later section.

```
1   private def uploadToS3(path: Path, destination: Destination): IO[Error,
      ↪ PutObjectResponse] =
2   val uploadAction = for
3    _ <- ZIO.logInfo(s"Uploading ${path.getFileName} to S3...")
4    _ <- ZIO.attemptBlocking(fileSizeInKB(path)) @@ S3FilePersistor.s3UploadSize
5    key = "%s/year=%04d/month=%02d/day=%02d/hour=%02d/minute=%02d/data.parquet".format(
6     env.value,
7     destination.year,
8     destination.month,
9     destination.day,
10    destination.hour,
11    destination.minute
12   )
13   putObjectRequest = PutObjectRequest.builder().bucket(bucket.value).key(key).build()
14   res <- ZIO
15    .attemptBlocking(s3Client.putObject(putObjectRequest, path))
16    @@ S3FilePersistor.s3UploadDuration.trackDurationWith(_.toMillis.toDouble)
17  yield res
```

Code listing 4.7 Method for data upload to aws s3

4.2.4 Neo4j repository

Integrating Neo4j repository functionality and custom procedures proved to be a significant challenge in developing the Totem Archiver Service. As a new technology within our team and company, this aspect required substantial time and research to master effectively. In this section, I'll delve into the details of the current implementation's writing capabilities, discussing the complexities involved and the strategies employed to ensure optimal performance. Additionally, I'll provide a retrospective analysis of previous iterations, highlighting their limitations and the lessons learned that shaped the current, more robust solution.

Figure 4.1 illustrates the Neo4j data model, providing a clear visual depiction of how data is structured within this graph database. This figure showcases the arrangement of nodes, relationships, and properties that collectively form the graph. Nodes represent entities, each labeled and characterized by various properties that define their attributes. Relationships, that connect these nodes, are also annotated with properties that describe the nature and dynamics of the links.

Figure 4.1 depicts a high-level overview of the data model. Let's review some examples to understand this structure better and the motivation behind this.

Why root node?

Neo4j Community Edition's original limitation of a single database per instance drove the development of a custom root node. This node effectively separates data from different environments (test, stage, prod), providing a clear organizational structure. Although we subsequently discovered and enabled Neo4j's multi-database capability, our approach offers a distinct performance advantage. By pinpointing the precise starting point of graph traversals, we achieve a guaranteed $\mathcal{O}(1)$ lookup. This significantly streamlines searches, especially within large and complex datasets.

What does A_{hash} mean?

Segment names are hashed into relationship types using the xxHash[3] library. Unfortunately, this library generates only 32-bit hashes. We employed a bitmask to extract the last 16 bits, preserving the desirable properties of the hash while adhering to Neo4j's relationship type constraint.

Our initial approach relied on a single, universal relationship type called `:NEXT`. However, as historical data accumulated, nodes with numerous neighbors became a bottleneck. Traversing these neighbors in search of a specific relationship proved costly, leading to $\mathcal{O}(n)$ complexity in the worst case.

To drastically enhance performance, we devised a strategy to significantly reduce each node's degree (number of neighbors). Seeking improvement, we shifted to a design where nodes remained empty, and relationships encoded their only property (name). This aimed to achieve near-constant $\mathcal{O}(1)$ lookup between nodes. Neo4j's internal neighbor table, structured as a map where keys are relationship types, would theoretically limit each node's `:NEXT` relationships to one, allowing for efficient retrieval.

Unfortunately, Neo4j Community Edition's limit of 2^{16} (65,536) relationship types constrained this otherwise highly performant solution. To optimize within this limitation, we adopted a hashing strategy. Segment names are hashed into relationship types, retaining $\mathcal{O}(n)$ complexity but reducing the search space by a significant constant factor.

What do n,v,m properties mean?
It's important to highlight that our primary goal has been to minimize disk space usage. From the beginning, it was clear that the benefits of using more descriptive property keys would be significantly outweighed by the additional disk space required. Consequently, we opted for abbreviated key names right from the start to conserve space efficiently. For instance, 'n' stands for 'name,' 'v' for 'values,' and 'm' for 'metadata.'

How many neighbors can a single node have?

Node connectivity within the graph is highly variable. Nodes can have anywhere from a single neighbor (like leaf nodes with only a parent connection) to a theoretically unbounded number of neighbors. Certain crucial nodes experience a rapid and continuous expansion of their relationships, making it impractical to estimate a fixed range. For example, some nodes might consistently increase their outgoing degree by an average of 100,000 relationships per day.

What is the purpose of leaf nodes?

[3] *Java xxHash provides blazing-fast, non-cryptographic hashing functions for various data types* [31].

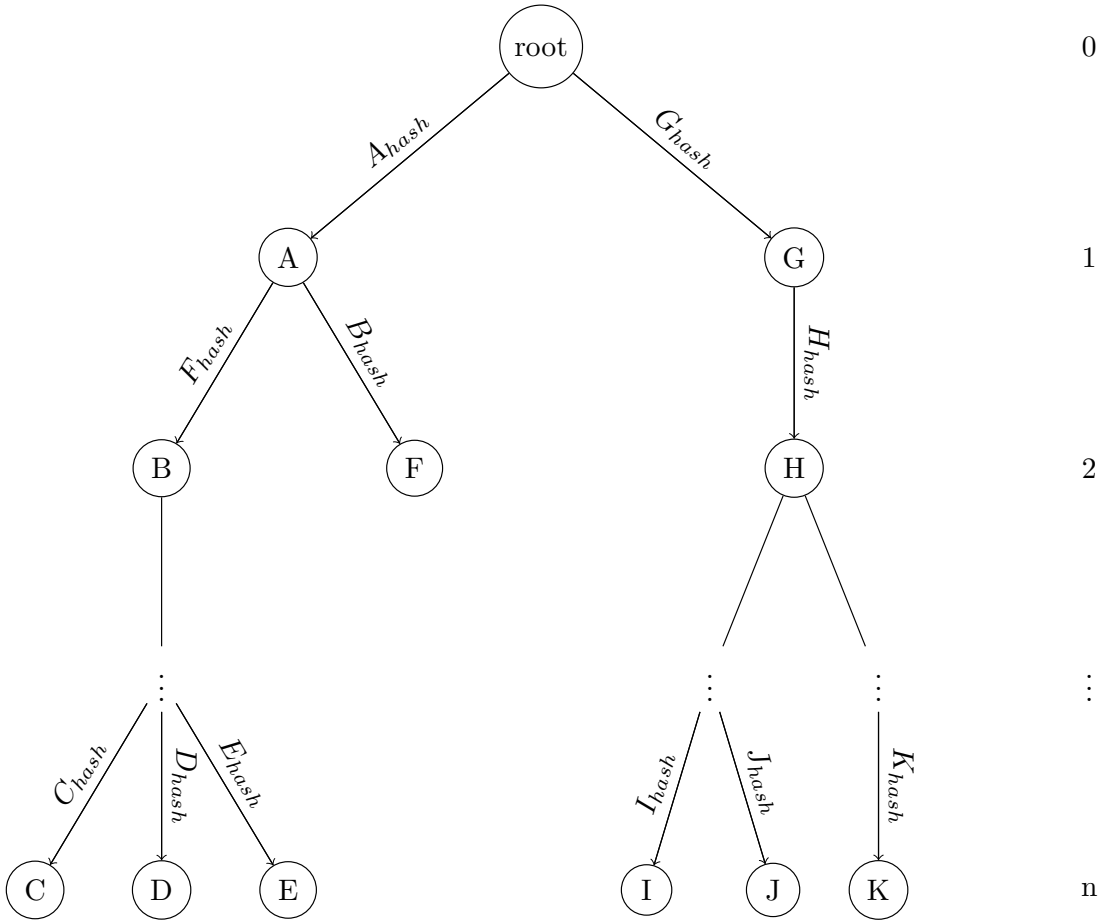

Figure 4.1 Neo4j data model

Every node in this structure is equally important, but some nodes, especially the leaf, hold more information than others. A leaf node typically holds this:

- **n:** The segment name is a universal property identifying each node.
- **v:** A sequence of long values (millis since epoch). These represent the first occurrence timestamps (in minutes) of specific paths within the data.
- **m:** A corresponding sequence of boolean values, acting as metadata. These flags indicate whether a path value in the **v** sequence had last update corresponding to a null value in the minute. Their primary purpose is to streamline the data sent back to the Totem Archiver.

Example:
Consider node **C** with the following properties:
- **n:** *price* (its segment name)
- **v:** *[42, 64200]* (first occurrence timestamps in minutes)
- **m:** *[false, true]* (corresponding metadata flags)

Node C resides at depth four within the graph, with its path being A/B/X/C.

To illustrate data retrieval:

- **A:** Searching for path: A/B/X/C at time 100. We'd retrieve the path A/B/X/C and the value 42. The 'false' flag in the m array indicates a valid value exists at this timestamp.

- **B:** Searching for path: A/B/X/C at time 6500. No value would be retrieved. The *true* flag in m signifies a `null` update. This optimization avoids returning data points that provide no value to the API.

Caching mechanism:

We utilize a caching mechanism to enhance the performance of transactions. Maintaining consistency is crucial, as the cache holds information across multiple transactions. We update our cache dynamically so that transactions can benefit from it. However, this approach has some disadvantages due to the lack of native support in Neo4j, requiring us to develop a custom solution to handle potential transaction crashes and rollbacks. To address this, we considered two approaches:

Version I. CombinedCache
This version would use two structures to ensure cache integrity, comprising of a `globalCache` and a `supportingStructure`. The `supportingStructure` could be a set of added keys during transactions (referred to as `addedKeys`) or a simple map to temporarily hold new cached data (referred to as `localCache`). Each would manage rollback and commit operations differently:

1. **addedKeys:**
   - **commit:** Clears the set.
   - **rollback:** Iterates through the set and invalidates keys in `globalCache`.
2. **localCache:**
   - **commit:** Iterates through the map and transfers items to `globalCache`.
   - **rollback:** Clears the map.

Drawbacks: Both strategies have major disadvantages. They could potentially cause OOM issues during batch insertions due to additional overhead and stress on the heap. Moreover, if a transaction ends abruptly, it may leave the cache in an inconsistent state.

Version II. Self-repairing cache
This version simplifies the previous approach by using only one cache. It operates as before but includes a `try-catch` block around node retrieval. If this operation throws an exception, it indicates that there is an inconsistency between the cache and the database. The problematic key is then invalidated, and the process moves to the next one. This method does not introduce additional overhead and has the advantage of being self-repairing. Refer to code listing 4.8 lines 6-10 for details on this approach.

Cache structure and lookup:

Our cache employs lists of strings as keys, representing a `path`, and strings as values, representing `node element ids` (code listing 4.8 visualizes the cache lookup process). For an incoming path like *A/B/C/C* see example below:

- 1. We search the cache for the full path *A/B/C/D*.
- 2. If not found, we search for *A/B/C*.
- 3. If not found, we search for *A/B*.
- 4. Finally, we search for *A*.

```
private Node findInCache(
   LinkedList<String> remainingSegments, LinkedList<String> consumedSegments) {
  while (!remainingSegments.isEmpty()) {
   String cachedNodeId = cache.getIfPresent(remainingSegments);
   if (cachedNodeId != null) {
    try {
     return tx.getNodeByElementId(cachedNodeId);
    } catch (NotFoundException e) {
     log.warn("Error retrieving node from cache for segments: %s", remainingSegments);
     cache.invalidate(remainingSegments);
    }
   }
   consumedSegments.addFirst(remainingSegments.removeLast());
  }
  return null;
 }
```

Code listing 4.8 Cache look up

If there's no match at any level, we must check and potentially create these nodes within our `processNextNode` method (code listing 4.9 depicts the node processing algorithm). The process is initiated by calculating a hash of the child node's value. Subsequently, the algorithm searches for a pre-existing relationship between the current and child nodes. If a relationship is discovered, the algorithm must further verify a specific property value (`segmentValue`) due to the potential for multiple nodes sharing the same hashed edge label (`segmentHash`). If no matching relationship exists, the algorithm creates the relationship and generates the child node.

```
private Node processNextNode(String segmentValue, int depth, Node currentNode) {
  RelationshipType segmentHash = Utils.getRelationshipType(segmentValue);
  Optional<Node> maybeNextNode =
    currentNode.getRelationships(Direction.OUTGOING, segmentHash).stream()
      .map(Relationship::getEndNode)
      .filter(node -> node.getProperty(NameKey).equals(segmentValue))
      .findFirst();
  return maybeNextNode.orElseGet(
    () -> createNextNode(segmentValue, depth, currentNode, segmentHash));
 }
```

Code listing 4.9 Process next node

4.2.4.1 Lessons learned

The following text explores early approaches that ultimately proved unsuitable. This iterative development process played a crucial role in refining our current solution, with insights gained from previous versions' limitations.

Iteration I:
Initially, our approach relied on simple Cypher queries (code listing 4.10 shows Cypher merge query for one path.), which, in retrospect, were markedly inefficient for our application's needs. These queries operated prohibitively slowly, making it impractical to process substantial amounts of data within a single minute. Consequently, our application risked longer processing times than the Totem Archiver, leading to potential data loss due to Apache Kafka's data retention limitations. Furthermore, this version did not address metadata updates, as the concept had not yet been introduced.

```
1 MERGE (a:‘0‘ {n: ’A’})-[:NEXT]->(b:‘1‘ {n: ’B’})
2 MERGE (b)-[:NEXT]->(c:‘2‘ {n: ’C’})
3 MERGE (c)-[:NEXT]->(d:‘3‘ {n: ’D’})
4 ON CREATE SET d.v = 42
5 ON MATCH SET d.v = d.v + 42
```

Code listing 4.10 Cypher query

Iteration II:
At this juncture, it became evident that a more expressive tool was necessary to adequately meet our specific requirements. Neo4j, with its capability to write custom procedures at a very low level, presented an ideal solution.

Initially, we implemented a straightforward graph traversal strategy, beginning at the root and progressively moving to the leaves. Our navigation relied solely on the node property ’n’, necessitating iteration through all outgoing edges (relations) to determine the subsequent path. Although this method had its limitations, it was significantly more effective than previous efforts. Notably, it facilitated the storage of information at a pace surpassing that at which the Totem Archiver could produce it.

Iteration III:
Although the previous iteration was deemed adequate at the time, concerns about its potential obsolescence arose due to the anticipated increase in client subscriptions to Totem. This expansion would inevitably lead to more updates and, consequently, an increased volume of data requiring processing.

Recognizing the need for enhancements, we identified caching as a pivotal strategy for improving processing speed. Utilizing cache would allow us to retrieve data from RAM, rather than relying on slower disk storage, thereby achieving significant gains in speed.

The development of this caching solution underwent several iterations:

1. The initial version utilized a simple hashmap, which proved to be highly inefficient. This approach employed nodes as keys, with values stored in another map where the node property ’n’ served as the key and the values were nodes themselves. Without delving into excessive detail, it is sufficient to say this method was resource-inefficient.

   A major drawback of this initial caching approach was its lack of an upper bound. Consequently, when an outlier emerged—say, one with five times the typical size—it could exhaust the available heap memory.

2. In an effort to optimize our caching strategy, we devised a more conventional cache-like solution, wherein paths served as keys and nodes as corresponding values. This approach demonstrated a reduced resource footprint.

   For instance, an update involving the path ”A/B/C” would be handled by storing three distinct keys: ”A”, ”A/B”, and ”A/B/C”. Each key would map to node references for ”A”, ”B”, and ”C” respectively, ensuring a structured and efficient caching system.

3. Despite improvements, occasional out-of-memory (OOM) errors persisted due to the presence of data outliers. To address this issue, we adopted Caffeine, a high-performance caching library renowned for its near-optimal efficiency.

   Caffeine provides a comprehensive API that supports various caching strategies. We opted for a straightforward configuration: a maximum size limit without a weighting function, ensuring that each element is treated equally, and a simple expiration policy. This approach effectively prevents any occurrences of out-of-memory (OOM) errors.

4. While our caching strategy approached its operational limits, we conceived an additional refinement: maintaining the cache across transactions. In the existing implementation, the cache was reconstructed for each transaction, a process that, although effective, was resource-intensive. We hypothesized that by reusing cache data, which is highly time-sensitive, we could achieve greater efficiency and conserve resources.

   To implement this innovation, a fundamental modification of our model was necessary. Specifically, values could no longer be stored as references to nodes due to their variability across transactions. Instead, storing identifiers (IDs) would ensure consistency and usability in subsequent transactions, as these IDs remain stable even when the actual node references change.

5. While the cache had been optimized for space and time complexity, its design left it susceptible to transaction failures. Due to its ad-hoc construction, the lack of a rollback mechanism could lead to the presence of invalid data in the cache, potentially causing subsequent transaction failures.

   We considered several solutions. One approach involved invalidating keys when a value (node ID) was not found, introducing an additional check during each cache retrieval. However, we ultimately adopted a more robust solution:

   We continue to populate the cache during the insertion process but now also track inserted keys throughout the transaction. In the event of a transaction failure, we can selectively invalidate these keys, ensuring the cache remains in a consistent state.

Iteration IV:
At this stage, while we have enhanced our querying capabilities, questions remain regarding possible improvements to our model. Indeed, there is room for optimization. Previously, we employed a generic relationship type, denoted as `:NEXT`. To improve search performance and decrease the resulting dataset size, we now employ hashed relationship types. This adjustment refines the specificity of queries, thereby facilitating faster and more efficient data retrieval.

4.2.5 Tombstone repository

A repository that maintains records of totem outages is crucial for ensuring data consistency when users make requests. Given the impracticality of manually nullifying every update valid at the time of an outage, we utilize a specialized mechanism known as the Tombstone Repository. This repository features a straightforward API, as illustrated in Code Listing 4.11.

```
1 trait TombstoneRepository:
2 def upsert(tombstone: Tombstone): Task[Unit]
3 def findInRange(validFrom: Instant, maybeValidTo: Option[Instant]): Task[Vector[
      ↪ Tombstone]]
```

Code listing 4.11 Tombstone repository API

The `upsert` method in this API takes a `Tombstone`, which is essentially a wrapper over an `Instant`. As messages are read from Apache Kafka, the timestamp of the last message received is continually updated. When an *Initialized* message is received, we insert or update the timestamp into the repository to mark the point of data validity up to the occurrence of an outage.

4.2.6 Recapitulation of data flow

1. Data is ingested through Apache Kafka.
2. Data aggregation continues until we encounter the first message that exceeds our defined threshold.
   - If an 'Initialized' message is detected, we record the `lastMessageAt` timestamp in the `Tombstone Repository`.
3. The data is written into a parquet file.
4. This file is then uploaded to S3.
5. We extract and insert the first occurrence of each path for a given minute into Neo4j in batches.
6. The offset is committed back to Apache Kafka.

4.3 Read Component

This section explores the sophisticated process of data retrieval, querying, and presentation within the Totem Archiver service. This multi-faceted approach employs a variety of services to guarantee effective and accurate data access. We will systematically detail these services, describing their particular functions in the entire data access workflow.

4.3.1 REST API

The REST API serves as the gateway to the read component, featuring a single endpoint 3.2 implemented using Tapir 3.3.1.4. One of the key advantages of Tapir is its ability to auto-generate the API documentation directly from the code. This ensures that the documentation remains current and is easy to maintain (code listing 4.12). The request is parsed, validated, and forwarded to the `ReadUpdateInfoService`.

```
1  object UpdateInfoEndpoint:
2   private val updateInfoPath = "update-info"
3   private val base = ApiRootV1.in(updateInfoPath)
4
5   def all(using JsonCodec[UpdateInfoRequest], JsonCodec[UpdateInfoResponse], JsonCodec[
       ↪ ApiError]): AllEndpoints =
6    Vector(listUpdateInfo)
7
8   def listUpdateInfo(using
9    JsonCodec[UpdateInfoRequest],
10   JsonCodec[UpdateInfoResponse],
11   JsonCodec[ApiError]
12  ): StandardEndpoint[UpdateInfoRequest, UpdateInfoResponse] =
13   base.post
14    .in(customCodecJsonBody[UpdateInfoRequest].example(Example.updateInfoRequest))
15    .out(customCodecJsonBody[UpdateInfoResponse].example(Example.updateInfoResponse))
16    .errorOut(oneOf(errorInternalServerError, errorBadRequest))
17    .tag(updateInfoPath)
```

Code listing 4.12 Update info endpoint definition

4.3.2 Neo4j repository

We have already discussed a lot about Neo4j and its applications in our architecture, including implementation specifics. Now, let's shift our focus to the read operations. Read operations do not alter the graph's layout, making them highly parallelizable. Our Totem Archiver service sends multiple concurrent read-only requests. By using methods that specifically limit access to reading, we inform Neo4j explicitly that no transaction log will be necessary in the event of a crash, as the graph structure will remain unaffected. code listing 4.13 displays the `Update Info Location Repository` API methods. We will particularly concentrate on the latter method `findUpdateInfoLocationByFlexiblePaths`.

```
1  trait UpdateInfoLocationRepository:
2   def batchInsertPathFirstOccurrences(
3    paths: Vector[PathFirstOccurrence]
4   ): IO[UpdateInfoLocationRepository.Error, Unit]
5
6   def findUpdateInfoLocationByFlexiblePaths(
7    flexiblePaths: Vector[FlexiblePath],
8    validFrom: Instant,
9    validTo: Option[Instant]
10  ): IO[UpdateInfoLocationRepository.Error, Vector[UpdateInfoLocation]]
```

Code listing 4.13 Update info location API

Refer to code listing 4.14, lines 7-8, for a visual representation of Neo4j API communication. Notice how we always parameterize our queries. This offers performance benefits and protection against potential exploits, and it aligns with Neo4j best practices. We create a session beforehand and ensure proper closure for optimal resource management. `ZIO.attemptBlocking` is a wrapper that helps manage concurrent, potentially exception-throwing operations. It signals non-blocking execution on a separate thread pool (`ZIO.Attempt`) and potential exception handling (`ZIO.Blocking`). Lines 11-20 in code listing 4.14 demonstrate how we process the result and convert it to our domain structure. These lines execute concurrently, achieved by using `ZIO.foreachPar`—acting as a map rather than a traditional foreach, transforming rather than discarding the result.

```
1  private def fetchUpdateInfoLocations(findQuery: CypherQuery): IO[Error, Vector[
       ↪ UpdateInfoLocation]] =
2    ZIO.scoped:
3     for
4      session <- ZIO.succeed(driver.session(SessionConfig.forDatabase(env.value)))
5      neo4jRecords <- ZIO
6       .attemptBlocking:
7        session.executeRead: tx =>
8         tx.run(findQuery.definition, findQuery.parameters).asScala.toVector
9       .mapError(error => FindFailure(findQuery, error.some))
10      .ensuring(ZIO.succeed(session.close()))
11     updateInfoLocations <- ZIO.foreachPar(neo4jRecords): neo4jRecord =>
12      neo4jRecord.values().asScala.toList match
13       case rawPath :: rawValues :: _ =>
14        val path = Path(rawPath.asList[String](_.asString()).asScala.toVector)
15        val locations = rawValues.asList(_.asLong()).asScala.toVector.map(Instant.
       ↪ ofEpochMilli)
16        ZIO.succeed(UpdateInfoLocation(path, locations))
17       case _ => ZIO.fail(ParseFailure(neo4jRecord))
18    yield updateInfoLocations
```

Code listing 4.14 Fetch update info locations

On the Neo4j server, we have defined a custom procedure, as shown in code listing 4.16. The arguments `validFrom & validTo` are essential for filtering out irrelevant update information directly on the Neo4j side, thereby reducing data transfer size and enhancing performance.

```
1 @Procedure(name = "find", mode = Mode.READ)
2 @Description("CALL find('123', ['A','B','C'], 42, 100)")
3 public Stream<UpdateInfo> find(
4  @Name("id") String id,
5  @Name("path") List<String> path,
6  @Name("validFrom") Long validFrom,
7  @Name("validTo") Long validTo)
8  throws Exception
```

Code listing 4.15 Find procedure definition

Refer to code listing 4.16 for a clearer visualization of how data is searched. We traverse the graph iteratively, differentiating among three potential scenarios:

1. **Leaf case**: When a leaf node is reached, extract the information, filter timestamps within the range, add them to our accumulator, and proceed (see line 9 in code listing 4.16).
2. **Wildcard case**: If a wildcard segment is encountered, add every neighboring node to the queue (refer to line 17 in code listing 4.16).
3. **Else case**: This is the most common scenario where we search for a specific neighbor somewhere within the graph (line 24 in code listing 4.16 illustrates this case).

```
1  private List<UpdateInfo> findBucket(
2     Deque<TraversalState> nodesToVisit,
3     Long validFrom,
4     Optional<Long> validTo,
5     List<UpdateInfo> acc) {
6    while (!nodesToVisit.isEmpty()) {
7     TraversalState traversalState = nodesToVisit.remove();
8
9     if (traversalState.isLeaf()) {
10     LeafData leafData = getProperties(traversalState.node);
11     LeafData formattedLeafData = Utils.findInRange(leafData, validFrom, validTo);
12     if (formattedLeafData != null) {
13      acc.add(
14        new UpdateInfo(
15          traversalState.consumed, leafData.getTimestamps(), leafData.getMetadata()));
16     }
17    } else if (traversalState.isWildcard()) {
18     for (Node nextNode : getOutgoingNeighbours(traversalState.node, Optional.empty()))
       ↪ {
19      List<String> newConsumedPath = new ArrayList<>(traversalState.consumed);
20      newConsumedPath.add(getNodeProperty(nextNode));
21      nodesToVisit.addLast(
22        new TraversalState(nextNode, traversalState.getTailRemaining(), newConsumedPath)
       ↪ );
23     }
24    } else {
25     String propertyValue = traversalState.getFirstRemaining();
26     List<Node> neighbors =
27       getOutgoingNeighbours(
28         traversalState.node, Optional.of(Utils.getRelationshipType(propertyValue)));
29
```

```
30      Optional<Node> nextNode =
31        neighbors.stream()
32          .filter(n -> n.getProperty(NameKey).equals(propertyValue))
33          .findFirst();
34
35      if (nextNode.isPresent()) {
36       nodesToVisit.add(
37         new TraversalState(
38           nextNode.get(),
39           traversalState.getTailRemaining(),
40           traversalState.addConsumed(propertyValue)));
41      } else {
42       log.info("Segment " + propertyValue + " not found for path: ...");
43      }
44     }
45    }
46    return acc;
47   }
```

Code listing 4.16 Find procedure definition

The `findInRange` method in the `Utils` class locates relevant `LeafData` instances within a specified range (see code listing 4.16, line 11). `LeafData` encapsulates `values` (`List<Long>`) and `metadata` (`List<Boolean>`). Here's how the process works:

1. **Sorted search**: We perform a binary search on the `values`, leveraging its guaranteed sorted order.
2. **Index adjustment**: Indices are modified for our specific filtering needs.
3. **Subsets extracted**: Based on the adjusted indices, subsets of `values` and `metadata` are extracted.
4. **Invalidating leaf data**: Leaf data is set to null (invalidated) in two cases:
   a. `Values` is empty.
   b. Single `true` element in the extracted subset of `metadata`.[4]

4.3.3 Amazon Athena integration

To integrate Athena with our application, we utilize the jOOQ API. jOOQ, which stands for Java Object Oriented Querying, is an SQL library designed for Java and also compatible with Scala. It abstracts SQL types into Java types, enabling us to work more intuitively with SQL operations within Java applications. Importantly, jOOQ supports a wide variety of SQL dialects, including Trino, which is utilized within Athena. This compatibility is crucial for ensuring smooth interactions between our application and Athena.

In our current implementation, we employ the *Athena JDBC Driver 3.0*. This version offers substantial improvements over its predecessors, notably the active fetching of query results back to the application, a feature that significantly enhances our data retrieval processes.

[4] In this scenario, we would retrieve only one value from S3, which would be null—a value we discard in the API anyway. Therefore, there is no reason to return it.

As depicted in code listing 4.17, we use this driver to establish a connection to Amazon Athena. Our architecture involves multiple schemas for different operational environment:

- *totem_archiver_test*
- *totem_archiver_stage*
- *totem_archiver_prod*

Each schema contains an external table named *update_info* that operates over distinct files in an S3 bucket.

Within jOOQ, we manage a single schema at a time. Therefore, when building the repository, it is crucial to specify the output schema based on the environment in which our application is running (see code listing 4.18). This approach allows us to dynamically adjust the schema according to the deployment environment, ensuring that our database interactions are contextually relevant and accurate.

```
1  import com.amazon.athena.jdbc.AthenaDataSource
2
3  def layer(athenaConfig: AthenaConfig, awsConfig: AwsConfig): ZLayer[Environment,
       ↪ Throwable, DataSource] =
4    ZLayer.scoped:
5     ZIO.serviceWithZIO[Environment]: env =>
6      val athenaEnvConfig = athenaConfig.enrichWithEnv(env)
7      ZIO.attempt:
8       val athenaDataSource = new AthenaDataSource()
9       athenaDataSource.setRegion(awsConfig.region)
10      athenaDataSource.setDatabase(athenaEnvConfig.schema)
11      athenaDataSource.setCredentialsProvider(athenaEnvConfig.credentialsProvider)
12      athenaDataSource
```

Code listing 4.17 Athena datasource

This code provides a reusable, injectable component for establishing Athena connections within a ZIO application [5].

```
1  def layer: ZLayer[DataSource & Environment, Nothing, UpdateInfoRepository] =
2    ZLayer.fromFunction((ds: DataSource, env: Environment) =>
3     UpdateInfoRepositoryImpl(
4      DSL.using(
5       ds,
6       SQLDialect.TRINO,
7       new Settings()
8        .withRenderMapping(new RenderMapping()
9          .withSchemata(new MappedSchema()
10           .withInput(TotemArchiver.TOTEM_ARCHIVER.getName)
11           .withOutput(TotemArchiver.TOTEM_ARCHIVER.getName + env.suffix)
12         )
13       )
14     )
15    )
16   )
```

Code listing 4.18 Creation of update info repository

[5] *Note: Athena uses trino SQL dialect*

How is Athena query constructed?

During the implementation phase, we encountered a significant challenge in constructing an optimal query. This involved an iterative process where we tested various strategies. AWS provides metrics for Athena query execution, such as data scanned and time spent, which are crucial indicators. The amount of data scanned is significant since it directly affects costs. At the same time, the time spent is critical for ensuring that our tool efficiently locates data on S3, leveraging the Neo4j index it generates.

It's important to mention that we were always using these 3 conditions no matter the iteration, as they were fundamental for efficient querying, those were:

- **path condition**: specifying path
- **valid from / to condition**: specifying range
- **update location condition**: specifying bucket, in which we should search

Iteration I:

In our initial iteration, we utilized the `SELECT` statement combined with the `UNION ALL` operator. This method proved inefficient for data scanning, often scanning the same data multiple times. Each path's SELECT statement was constructed separately, potentially leading to oversized queries due to Amazon Athena's query size limits. A more significant issue arose when multiple paths were stored in the same bucket, causing redundant scans that increased costs and reduced processing speed.
Refer to code listing 4.19 to see how the query is structured for just two paths. Note that the `valid_from & valid_to` condition remains constant. The first query reveals that the values *A/B/C/D* have not changed for over a month, indicating that this is our only and most recent data location. Additionally, lines 27-30 (and 60-63) are optional conditions used only when the user specifies a date range."

```
1  SELECT
2   "totem_archiver"."update_info"."path",
3   "totem_archiver"."update_info"."value",
4   "totem_archiver"."update_info"."valid_from",
5   "totem_archiver"."update_info"."valid_to",
6   "totem_archiver"."update_info"."year",
7   "totem_archiver"."update_info"."month",
8   "totem_archiver"."update_info"."day",
9   "totem_archiver"."update_info"."hour",
10  "totem_archiver"."update_info"."minute"
11 FROM "totem_archiver"."update_info"
12 WHERE (
13  "totem_archiverv"."update_info"."path" = 'A/B/C/D'
14  AND "totem_archiver"."update_info"."year" = 1970
15  AND "totem_archiver"."update_info"."month" = 1
16  AND "totem_archiver"."update_info"."day" = 1
17  AND "totem_archiver"."update_info"."hour" = 0
18  AND "totem_archiver"."update_info"."minute" = 0
19  AND (
20   (
21    "totem_archiver"."update_info"."valid_from" <= 2768490000
22    AND (
23     "totem_archiver"."update_info"."valid_to" > 2768490000
24     OR "totem_archiver"."update_info"."valid_to" is null
25    )
```

```
  )
  OR (
   "totem_archiver"."update_info"."valid_from" >= 2768490000
   AND "totem_archiver"."update_info"."valid_from" < 2768495000
  )
 )
)
UNION ALL
SELECT
 "totem_archiver"."update_info"."path",
 "totem_archiver"."update_info"."value",
 "totem_archiver"."update_info"."valid_from",
 "totem_archiver"."update_info"."valid_to",
 "totem_archiver"."update_info"."year",
 "totem_archiver"."update_info"."month",
 "totem_archiver"."update_info"."day",
 "totem_archiver"."update_info"."hour",
 "totem_archiver"."update_info"."minute"
FROM "totem_archiver"."update_info"
WHERE (
 "totem_archiverv"."update_info"."path" = 'A/B/C/E'
 AND "totem_archiver"."update_info"."year" = 1970
 AND "totem_archiver"."update_info"."month" = 2
 AND "totem_archiver"."update_info"."day" = 2
 AND "totem_archiver"."update_info"."hour" = 1
 AND "totem_archiver"."update_info"."minute" = 1
 AND (
  (
   "totem_archiver"."update_info"."valid_from" <= 2768490000
   AND (
    "totem_archiver"."update_info"."valid_to" > 2768490000
    OR "totem_archiver"."update_info"."valid_to" is null
   )
  )
  OR (
   "totem_archiver"."update_info"."valid_from" >= 2768490000
   AND "totem_archiver"."update_info"."valid_from" < 2768495000
  )
 )
)
```

Code listing 4.19 Amazon Athena query (UNION ALL approach)

Iteration II:

To address the issues from the first iteration, we considered using a single `SELECT` statement with multiple `IN` operators. However, a significant drawback emerged, which we initially overlooked. The amount of data scanned increased dramatically as more distinct values were added to the sets.

Refer to code listing 4.20 for a clear visualization of this problem. It shows that we are scanning across two months, days, hours, and minutes, resulting in $2 * 2 * 2 * 2 = 16$ possible combinations. This approach led to a substantial waste of resources. Now imagine we had 30 minutes instead of just two; that would result in scanning across $2^3 * 30 = 240$ files, which would take more time and increase the costs.

```
1  SELECT
2    "totem_archiver"."update_info"."path",
3    "totem_archiver"."update_info"."value",
4    "totem_archiver"."update_info"."valid_from",
5    "totem_archiver"."update_info"."valid_to",
6    "totem_archiver"."update_info"."year",
7    "totem_archiver"."update_info"."month",
8    "totem_archiver"."update_info"."day",
9    "totem_archiver"."update_info"."hour",
10   "totem_archiver"."update_info"."minute"
11  FROM "totem_archiver"."update_info"
12  WHERE
13   ("totem_archiver"."update_info"."path" IN ('A/B/C/D', 'A/B/C/E')
14   AND "totem_archiver"."update_info"."year" IN (1970)
15   AND "totem_archiver"."update_info"."month" IN (1,2)
16   AND "totem_archiver"."update_info"."day" IN (1,2)
17   AND "totem_archiver"."update_info"."hour" IN (0,1)
18   AND ("totem_archiver"."update_info"."minute" IN (0,1))
19   AND (
20   (
21    "totem_archiver"."update_info"."valid_from" <= 2768490000
22    AND (
23     "totem_archiver"."update_info"."valid_to" > 2768490000
24     OR "totem_archiver"."update_info"."valid_to" is null
25    )
26   )
27   OR (
28    "totem_archiver"."update_info"."valid_from" >= 2768490000
29    AND "totem_archiver"."update_info"."valid_from" < 2768495000
30   )
31  )
32 )
```

Code listing 4.20 Amazon Athena query (IN approach)

Iteration III:

The use of the `IN` operator across individual fields turned out to be a poor choice, as it led to inefficient searching. To improve this, we devised a strategy where each bucket location had its own specific condition. For a detailed view of this implementation, refer to code listing 4.21 lines 14-18.

```
1  SELECT
2    "totem_archiver"."update_info"."path",
3    "totem_archiver"."update_info"."value",
4    "totem_archiver"."update_info"."valid_from",
5    "totem_archiver"."update_info"."valid_to",
6    "totem_archiver"."update_info"."year",
7    "totem_archiver"."update_info"."month",
8    "totem_archiver"."update_info"."day",
9    "totem_archiver"."update_info"."hour",
10   "totem_archiver"."update_info"."minute"
11  FROM "totem_archiver"."update_info"
12  WHERE
13   ("totem_archiver"."update_info"."path" IN ('A/B/C/D', 'A/B/C/E')
14   AND ("totem_archiver"."update_info"."year",
```

```
    "totem_archiver"."update_info"."month",
    "totem_archiver"."update_info"."day",
    "totem_archiver"."update_info"."hour",
    "totem_archiver"."update_info"."minute") IN ((1970, 2, 2, 1, 1), (1970, 1, 1, 0, 0)
    ↪ )
  AND (
  (
   "totem_archiver"."update_info"."valid_from" <= 2768490000
   AND (
    "totem_archiver"."update_info"."valid_to" > 2768490000
    OR "totem_archiver"."update_info"."valid_to" is null
   )
  )
  OR (
   "totem_archiver"."update_info"."valid_from" >= 2768490000
   AND "totem_archiver"."update_info"."valid_from" < 2768495000
  )
 )
)
```

Code listing 4.21 Athena query (IN approach II.)

Iteration IV:

Unfortunately, the enhancements we made to our query did not deliver the performance improvements we anticipated. Unable to pinpoint the reason for this shortfall, we decided to simplify the query. We defined each location condition separately using straightforward `=`, `AND` operators. This approach still involved consolidating the year, month, day, hour, and minute into a single condition, but we connected these conditions with an `OR` operator instead (see figure 4.22).

```
SELECT
  "totem_archiver"."update_info"."path",
  "totem_archiver"."update_info"."value",
  "totem_archiver"."update_info"."valid_from",
  "totem_archiver"."update_info"."valid_to",
  "totem_archiver"."update_info"."year",
  "totem_archiver"."update_info"."month",
  "totem_archiver"."update_info"."day",
  "totem_archiver"."update_info"."hour",
  "totem_archiver"."update_info"."minute"
 FROM "totem_archiver"."update_info"
 WHERE
  ("totem_archiver"."update_info"."path" IN ('A/B/C/D', 'A/B/C/E')
  AND (
      (
      "totem_archiver"."update_info"."year" = 1970
      AND "totem_archiver"."update_info"."month" = 2
      AND "totem_archiver"."update_info"."day" = 2
      AND "totem_archiver"."update_info"."hour" = 1
      AND "totem_archiver"."update_info"."minute" = 1
      )
    OR (
      "totem_archiver"."update_info"."year" = 1970
      AND "totem_archiver"."update_info"."month" = 1
      AND "totem_archiver"."update_info"."day" = 1
```

```
26        AND "totem_archiver"."update_info"."hour" = 0
27        AND "totem_archiver"."update_info"."minute" = 0
28        )
29      )
30    AND (
31    (
32     "totem_archiver"."update_info"."valid_from" <= 2768490000
33     AND (
34      "totem_archiver"."update_info"."valid_to" > 2768490000
35      OR "totem_archiver"."update_info"."valid_to" is null
36     )
37    )
38    OR (
39     "totem_archiver"."update_info"."valid_from" >= 2768490000
40     AND "totem_archiver"."update_info"."valid_from" < 2768495000
41    )
42   )
43  )
```

Code listing 4.22 Amazon Athena query (Simplified IN approach)

4.3.4 Tombstone repository

The Tombstone Repository API has already been presented in figure 4.11. However, our current focus is on the `findInRange` API method. This method identifies all totem outages within a specified range and includes the `Tombstone` with the largest timestamp among records earlier than `validFrom`. To illustrate this, I have prepared two images.

Figure 4.2 demonstrates what the data would look like without this strategy of persisting totem outages. At first glance, everything might appear normal, but this would mislead the user requesting the information, as it fails to account for two significant outages.

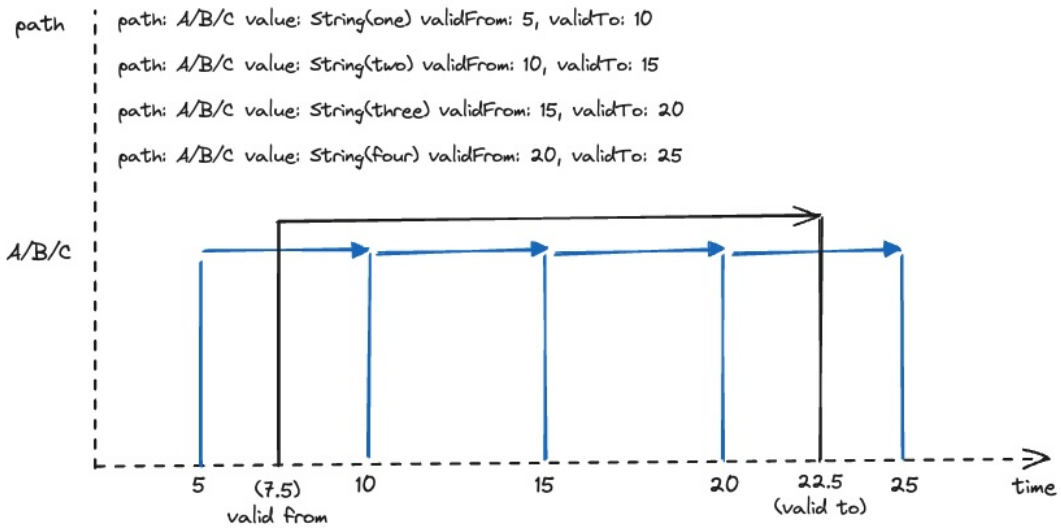

Figure 4.2 Data without tombstones

Figure 4.3 demonstrates how the Tombstone mechanism addresses data inaccuracies caused by outages. Notice the inclusion of a `Tombstone` with the largest timestamp preceding the requested range. This effectively nullifies updates prior to the outage, ensuring accurate results for the user.

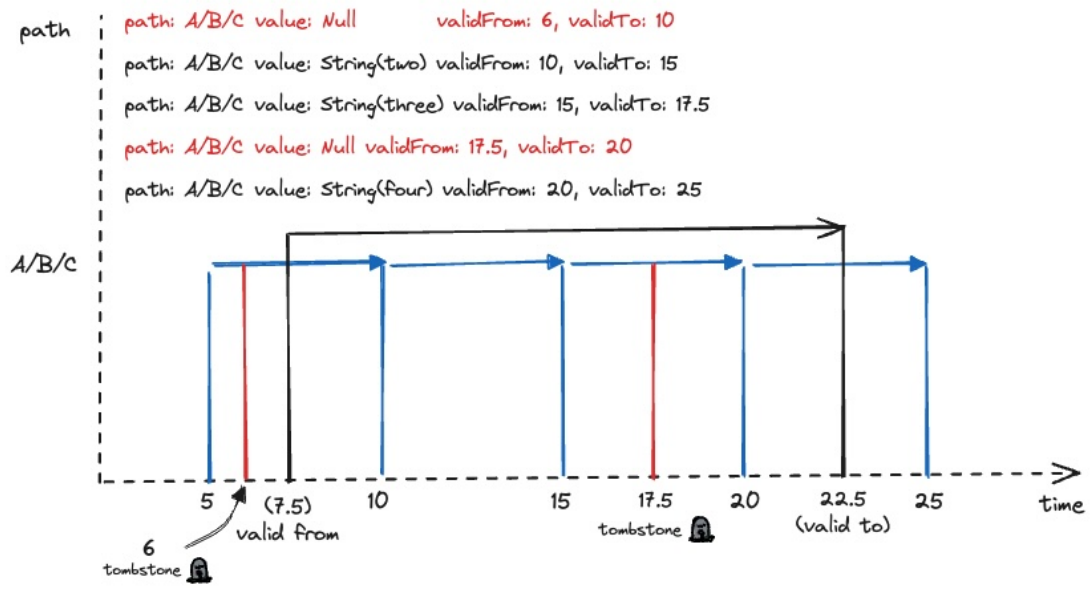

Figure 4.3 Data with tombstones

4.3.5 Recapitulation of data flow

1. A HTTP request is received.
2. The request is deserialized, validated, and then directed to the `ReadUpdateInfoService`.
3. We retrieve the data using a defined find procedure in Neo4j.
4. We identify the files containing the data on S3 and perform queries using Athena.
5. The data is grouped by `path`, and intervals are connected.
6. Relevant tombstones are identified, and the intervals are adjusted accordingly.
7. We exclude any update information containing null values.
   - Optionally, we adjust the `valid from / to` of the `UpdateInfo` based on the `requested valid from / to`.

Chapter 5

Testing & Metrics

This chapter concentrates on the testing and evaluation of the current implementation. For evaluation, we have developed various metrics to assess the performance of the implemented solution more effectively. Testing is divided into two main categories:

Unit testing: We created comprehensive unit test suites to examine each component in isolation. This helps us pinpoint potential errors within the application and proactively minimize issues when we introduce new features.

End-to-end testing: E2E testing simulates real-world user interactions, validating the system as a whole. This ensures all components work together seamlessly and provide the intended user experience.

5.1 Testing

Let's begin by examining our unit testing strategy. We rigorously tested critical components of the Totem Archiver to ensure their individual functionality. Adhering to best practices allowed us to effectively mock necessary traits, providing full control over the testing environment and isolating component behavior.

We need to set up a local environment for this type of testing for this type of testing. We run our initialization script to populate the container with the necessary data. For PostgreSQL, we employ Flyway to manage our test data; each time we execute the test suite, Flyway creates the required test data.

Testing with Neo4j proved to be more challenging because it lacks a direct equivalent to Flyway. As a result, we have to spin up a separate Neo4j instance capable of running a Cypher query that we send to our database. While there might be more elegant solutions, this approach suffices for our needs.

We utilize two docker-compose files to establish our local testing environment:

- **docker-compose.yml:** Defines the core services:
  - Neo4j
  - PostgreSQL
  - MinIO (an S3-compatible container)
- **docker-compose.test.yml:** Executes initialization scripts to populate Neo4j, PostgreSQL and MinIO with essential integration test data.

Refer to the code listing 5.1 for context, lines 2-9, as they depict how we spin up our local environment. Due to the error-prone nature of time series data, we meticulously tested all possible scenarios, especially edge cases, to guarantee correct behavior. The reader comments within the code further clarify the testing conditions.

```
1   object UpdateInfoLocationRepositorySpec extends ZIOSpec[UpdateInfoLocationRepository
      ↪ & Driver]:
2  override def bootstrap: ZLayer[Any, Any, UpdateInfoLocationRepository & Driver] =
3   Runtime.setConfigProvider(ApplicationConfig.defaultProvider) >+>
4    TestContainers.dockerComposeLayer >+>
5    TestContainers.updatedConfigLayer.flatMap: env =>
6     DataSourceFactory.layer(env.get[ApplicationConfig].neo4j) >+>
7      Environment.layer(env.get[ApplicationConfig].env) >+>
8      UpdateInfoLocationRepositoryImpl.layer(env.get[ApplicationConfig].batchSize)
9
10  override def spec = suiteAll("UpdateInfoLocationRepository") {
11   // path timestamps: --------/X--------X--------X
12   // fetch interval: -------X|
13   test("path timestamps are all greater than valid from, validTo is none") {
14    val validFrom = timestamps.head.minusMillis(1)
15    for
16     updateInfoLocationRepository <- ZIO.service[UpdateInfoLocationRepository]
17     result <- updateInfoLocationRepository.findUpdateInfoLocationByFlexiblePaths(Vector
        ↪ (firstFlexiblePath), validFrom, None).exit
18    yield assert(result)(fails(isSubtype[UpdateInfoLocationRepository.Error.NoRecords](
        ↪ anything)))
19   }
20   // path timestamps: --------|X--------X--------X
21   // fetch interval: --------|X
22   test("path timestamp is equal to valid from, validTo is none") {
23    val validFrom = timestamps.head
24    for
25     updateInfoLocationRepository <- ZIO.service[UpdateInfoLocationRepository]
26     updateInfoLocations <- updateInfoLocationRepository.
        ↪ findUpdateInfoLocationByFlexiblePaths(Vector(firstFlexiblePath), validFrom,
        ↪ None)
27    yield assert(updateInfoLocations)(
28     hasSameElements(Vector(UpdateInfoLocation(firstPath, Vector(timestamps.head))))
29    )
30   }
31   // path timestamps: --------|X--------X--------X
32   // fetch interval: --------|-----X
33   test("path timestamp is slightly bigger than to valid from, validTo is none") {
34    val validFrom = timestamps.head.plusMillis(1)
35    for
36     updateInfoLocationRepository <- ZIO.service[UpdateInfoLocationRepository]
```

```
37      updateInfoLocations <- updateInfoLocationRepository.
        ↪ findUpdateInfoLocationByFlexiblePaths(Vector(firstFlexiblePath), validFrom,
        ↪ None)
38     yield assert(updateInfoLocations)(
39      hasSameElements(Vector(UpdateInfoLocation(firstPath, Vector(timestamps.head))))
40     )
41    }
42    // path timestamps: --------|X--------X--------X
43    // fetch interval: X-------|O
44    test("fetch interval to is smaller or equal than smallest timestamp in path") {
45     val validFrom = epochInstant
46     val validTo = timestamps.head
47     for
48      updateInfoLocationRepository <- ZIO.service[UpdateInfoLocationRepository]
49      result <- updateInfoLocationRepository.findUpdateInfoLocationByFlexiblePaths(Vector
        ↪ (firstFlexiblePath), validFrom, validTo.some).exit
50     yield assert(result)(fails(isSubtype[UpdateInfoLocationRepository.Error.NoRecords](
        ↪ anything)))
51    }
52    // path timestamps: --------|X--------X--------X
53    // fetch interval:        X|-|O
54    test("fetch interval validFrom is smaller but to is bigger than smallest timestamp in
        ↪  path") {
55     val validFrom = timestamps.head.minusMillis(1)
56     val validTo = timestamps.head.plusMillis(1)
57     for
58      updateInfoLocationRepository <- ZIO.service[UpdateInfoLocationRepository]
59      updateInfoLocations <- updateInfoLocationRepository.
        ↪ findUpdateInfoLocationByFlexiblePaths(Vector(firstFlexiblePath), validFrom,
        ↪ validTo.some)
60     yield assert(updateInfoLocations)(
61      hasSameElements(Vector(UpdateInfoLocation(firstPath, Vector(timestamps.head))))
62     )
63    }
64    // path timestamps: --------X--------X--------X
65    // fetch interval:     X-----------------|O
66    test("fetch interval covers every update") {
67     val validFrom = timestamps.head
68     val validTo = timestamps.last.plusMillis(1)
69     for
70      updateInfoLocationRepository <- ZIO.service[UpdateInfoLocationRepository]
71      updateInfoLocations <- updateInfoLocationRepository.
        ↪ findUpdateInfoLocationByFlexiblePaths(Vector(firstFlexiblePath), validFrom,
        ↪ validTo.some)
72     yield assert(updateInfoLocations)(
73      hasSameElements(Vector(UpdateInfoLocation(firstPath, timestamps)))
74     )
75    }
76    // path timestamps: --------|X--------X--------X
77    // path2 timestamps: --------X-----------------|
78    // fetch interval:     X-----------------|O
79    test("fetch interval covers every update every path") {
80     val validFrom = timestamps.head
81     val validTo = timestamps.last.plusMillis(1)
82     for
83      updateInfoLocationRepository <- ZIO.service[UpdateInfoLocationRepository]
84      updateInfoLocations <- updateInfoLocationRepository.
```

```
    ↪ findUpdateInfoLocationByFlexiblePaths(
85    Vector(firstFlexiblePath, secondFlexiblePath),
86    validFrom,
87    validTo.some
88   )
89  yield assert(updateInfoLocations)(
90   hasSameElements(Vector(UpdateInfoLocation(firstPath, timestamps),
      ↪ UpdateInfoLocation(secondPath, Vector(timestamps.head))))
91  )
92 }
93
94 test("fetch path that is not in database") {
95  val validFrom = epochInstant
96  val validTo = timestamps.head
97  for
98   updateInfoLocationRepository <- ZIO.service[UpdateInfoLocationRepository]
99   result <- updateInfoLocationRepository.findUpdateInfoLocationByFlexiblePaths(Vector
      ↪ (unknownFlexiblePath), validFrom, validTo.some).exit
100  yield assert(result)(fails(isSubtype[Error.NoRecords](anything)))
101 }
102}
```

Code listing 5.1 Update info location repository spec

The next phase of the testing involved end-to-end testing (E2E), which was somewhat less rigorous. This was due to the difficulty of simulating precise conditions, as isolating the test proved to be inefficient in terms of time spent. Consequently, we conducted on-the-fly testing of the Totem Archiver service. This approach meant that our database was constantly changing, which required careful timing of intervals to ensure consistent results for each test run.

We designed several test scenarios, each targeting a different use case:

- A specific time and path
- A specific time and a path with wildcards
- A time range and a specific path
- A time range and a path with wildcards

The results were as expected: the most specific queries were the fastest, while those with broader time ranges and wildcards were the slowest. This aligns with the varying amounts of data retrieved.

Our tests pinpointed the primary bottleneck: Athena consistently took at least 15 seconds to process each query.

This situation may evolve as Neo4j continues to expand. The tests mentioned were conducted when there was approximately a week's worth of data in Neo4j, amounting to around 500 million nodes and edges. And future evaluation would make sense to see how results differ after more data is stored.

5.2 Metrics

This section explains how we keep track of the Totem Archiver service and measure its performance. We use various metrics to gain insights into the behavior of different I/O operations.

For this, we use Prometheus to gather metrics and Grafana to visualize them:

Prometheus is a powerful open-source monitoring and alerting toolkit. It collects and stores its metrics as time series data, i.e., data with timestamps. Prometheus uses a flexible query language called PromQL that lets users select and aggregate data in real-time [32]. It is highly effective for recording any numerical data over time, making it well-suited for monitoring the performance and health of various systems.

Grafana is an open-source platform for monitoring and observational data visualization. It is widely used for graphing and visualizing time series data. Grafana allows users to create dashboards with panels that depict specific metrics over time, pulled from an array of possible data sources including Prometheus [33]. This capability makes it an indispensable tool for dynamic analysis of metric data, providing insights through graphs, charts, and alerts based on the aggregated data.

Together, Prometheus and Grafana form a robust monitoring toolset, where Prometheus handles data collection and storage, and Grafana focuses on visualization and analysis. This combination allows for a comprehensive view of systems' performance metrics in a clear and actionable format.

One of the most important metrics we track is the execution time of the query that inserts data into Neo4j. This is crucial because if the upsert process takes longer than our current storage interval (which is 1 minute), we may lose data. To avoid this, we have set up an alert that notifies us if the process takes longer than expected. Other metrics we track include the time it takes to upload the parquet file to S3, the size of the parquet file that is being uploaded, and the number of paths that are inserted into it. These are all metrics regarding our write component.

We have also created metrics for the read component. These include the time it takes for our Neo4j index to retrieve the locations of our data, the execution time of Athena queries, and the retrieval of tombstones (although this is relatively insignificant as totem outages are rare).

Refer to figure 5.1 which illustrated the data upsertion process to Neo4j. This snapshot reflects the metrics at a point when approximately one billion nodes and relationships are stored in our Neo4j database. It will be interesting to observe how these metrics evolve over time.

Refer to figure 5.2 which illustrates the count of `UpdateInfo` entries written to the parquet file. The parquet metrics interval ranges can become quickly obsolete, and we should further optimize the interval ranges to better capture the count of written `UpdateInfos`.

Please refer to figures 5.3 and 5.4, which show the size of the parquet file and the time it takes to upload it to S3, respectively. These metrics could be further refined to capture the characteristics of the data more precisely.

Conclusion:

The Totem Archiver service maintains strong performance amid growing data volumes from an increasing number of publishers. To sustain this, continuous improvements and regular performance evaluations are crucial. Enhancing data processing and storage strategies, along with advanced monitoring, will ensure that the system scales effectively and remains robust in the face of increasing demands.

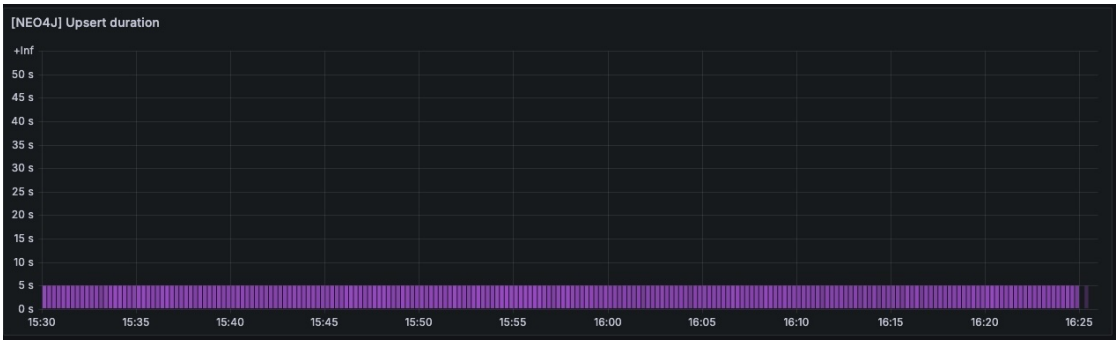

Figure 5.1 Neo4j query execution duration

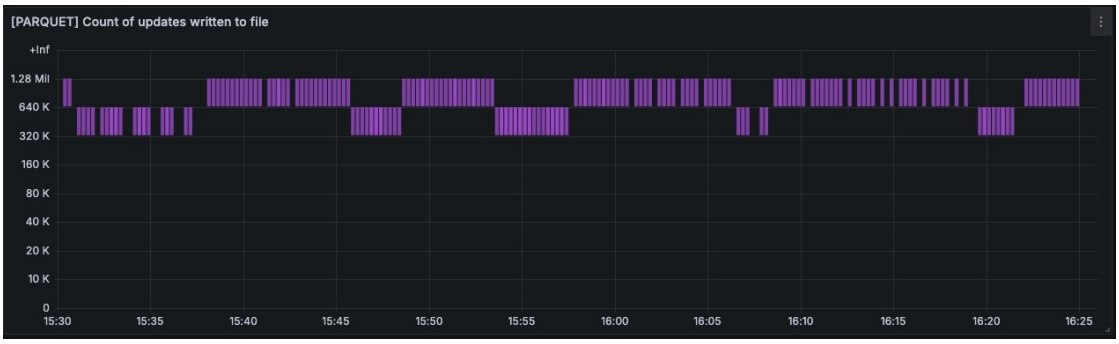

Figure 5.2 Count of update infos written to parquet file

Figure 5.3 Parquet file size

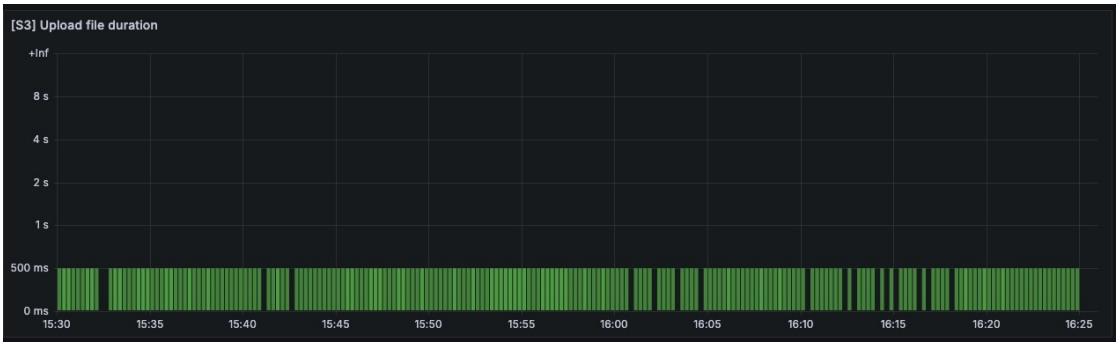

Figure 5.4 Parquet file upload duration to S3

Chapter 6

Future work

In this chapter, we will explore potential future enhancements to refine our current solution and optimize it. By examining the limitations of our current implementation, we can gain a clearer understanding of the necessary optimizations. This analysis will help us better tailor our enhancements to meet the evolving demands efficiently.

Let's examine potential limitations of our current solution and identify features that could significantly enhance its functionality.

Split the Totem Archiver into two independent services

At present, the Totem Archiver handles both writing and reading operations simultaneously. As other teams begin to fully utilize this service, this dual responsibility could lead to issues. Ideally, we would separate these functions into two independent services to ensure that one does not impact the performance of the other.

Endpoint for heavy loads

Currently, the Totem Archiver service has only one endpoint, which poses a risk of an OOM error if a request retrieves a large volume of data. This could cause the read component to affect the write component, potentially leading to a system crash. To mitigate this, we could introduce an additional endpoint that leverages the fact that Athena stores all its results in S3. This solution has a small limitation.
In the solution where we simply return a link to the S3 result location, we do not account for Totem outages, which could lead to inconsistencies in the data, therefore these outages would need to be as part of the response.

Neo4j internal limits

Currently, we are using the Neo4j Community edition, which, as I've previously noted (see 4.2.4), comes with significant storage limitations as illustrated in figure 6.1. While these limitations may not pose an immediate concern, our design's lack of data retention means we will eventually encounter these restrictions.

The simplest solution would be to upgrade to a Neo4j Enterprise license, which would, for example, enable the block entity format as seen in figure 6.2. However, this option is prohibitively expensive and may not be justified by our use case.

An alternative strategy could involve setting up an additional instance of Neo4j. We would maintain the original instance for read-only operations, and adapt our read component to retrieve data not just from the new instance (which would continue receiving new data) but also from the older one. We could further refine this system by establishing a specific timestamp marking when we transitioned to the new database. This would be intentionally useful in subsequent read operations to avoid unnecessary scans of the older database.

Table 2. Aligned format entity limits

Name	Limit
Property keys	`2^24` (16 777 216)
Nodes	`2^35` (34 359 738 368)
Relationships	`2^35` (34 359 738 368)
Properties	`2^36` (68 719 476 736)
Labels	`2^31` (2 147 483 648)
Relationship types	`2^16` (65 536)
Relationship groups	`2^35` (34 359 738 368)

Figure 6.1 Neo4j Community edition aligned format [34]

Table 8. Block format entity limits

Name	Limit
Nodes	`2^48` (281 474 976 710 656)
Relationships	`∞` (no defined upper bound)
Properties	`∞` (no defined upper bound)
Labels	`2^31` (2 147 483 648)
Relationship types	`2^30` (1 073 741 824)
Property keys	`2^31` (2 147 483 648)

Figure 6.2 Neo4j Enterprise edition block format [35]

Conclusion

The aim of this master's thesis was to develop a service capable of storing totem historical data, enabling users to recreate the state of the totem archiver for a desired time range or specific point in time.

Initially, an in-depth analysis of the totem service was conducted to gain a comprehensive understanding of its operation, internal data structure, data generation, and other relevant aspects. This analysis provided valuable insights into the service's functionality and the implications for the proposed solution.

Following the analysis, user requirements for the system were defined. Use cases provided a clearer understanding of user needs, shaping the functional requirements, which were relatively straightforward. However, non-functional requirements presented a significant challenge, demanding a robust system with specific technologies. The analysis of potential databases played a critical role in the successful implementation of the system.

The previous analysis identified crucial areas of the Totem Archiver service that required attention. This insight informed the proposal of a robust, scalable design that balances cost efficiency with performance. A user-friendly yet powerful API was also developed.

Throughout the implementation phase, we adhered to software engineering best practices to create software that is easily testable and maintainable. The service was deployed in a test environment, where the underlying infrastructure was fine-tuned, and end-to-end testing was conducted. During this phase, production data was utilized to simulate real-world scenarios. It was observed that the drop in performance concerning the database size was non-linear, resulting in good scalability. Key I/O component metrics were collected and visualized using Grafana.

The final chapter highlights potential improvements and future developments to enhance the Totem Archiver. I have outlined the most significant features that would benefit the Totem Archiver.

Bibliography

1. *It's like JSON. but fast and small.* [Online]. 2024. [visited on 2024-04-02]. Available from: https://msgpack.org/.
2. *What is a relational database?* [Online]. 2024. [visited on 2024-04-03]. Available from: https://cloud.google.com/learn/what-is-a-relational-database.
3. *What is a NoSQL database?* [Online]. 2024. [visited on 2024-04-03]. Available from: https://cloud.google.com/discover/what-is-nosql.
4. *What Is a Time-series Database (TSDB)?* [Online]. 2024. [visited on 2024-04-03]. Available from: https://www.purestorage.com/knowledge/what-is-a-time-series-database.html.
5. *Data Warehouse vs. Database: Understanding the Differences* [online]. 2024. [visited on 2024-04-03]. Available from: https://www.astera.com/type/blog/data-warehouse-vs-database/.
6. *The Rise of Multi-Model Databases* [online]. 2024. [visited on 2024-04-05]. Available from: https://medium.com/@igniobydigitate/the-rise-of-multi-model-databases-6e26c173c830.
7. *Polyglot persistence vs multi-model databases for microservices* [online]. 2024. [visited on 2024-04-09]. Available from: https://circleci.com/blog/polyglot-vs-multi-model-databases/.
8. *Cost-effective Use cases Benefits of Amazon S3* [online]. 2024. [visited on 2024-04-06]. Available from: https://adex.ltd/cost-effective-use-cases-benefits-of-amazon-s3.
9. *Scala is functional* [online]. 2024. [visited on 2024-04-07]. Available from: https://docs.scala-lang.org/tour/tour-of-scala.html.
10. *Scala is object-oriented* [online]. 2024. [visited on 2024-04-07]. Available from: https://docs.scala-lang.org/tour/tour-of-scala.html.
11. *Scala— Case Classes and Pattern Matching* [online]. 2024. [visited on 2024-04-07]. Available from: https://zeesh-arif.medium.com/scala-3-case-classes-and-pattern-matching-d68910651c75.
12. *ZIO* [online]. 2024. [visited on 2024-04-07]. Available from: https://zio.dev/reference/core/zio/.
13. *Introduction to ZIO Fibers* [online]. 2024. [visited on 2024-04-07]. Available from: https://zio.dev/reference/fiber/.

14. *Referential Transparency* [online]. 2024. [visited on 2024-04-07]. Available from: https://zio.dev/reference/error-management/imperative-vs-declarative/#referential-transparency.
15. *Handling Errors* [online]. 2024. [visited on 2024-04-07]. Available from: https://zio.dev/overview/handling-errors/.
16. *Chaining* [online]. 2024. [visited on 2024-04-07]. Available from: https://zio.dev/reference/core/zio/.
17. *JsonIter benchmark* [online]. 2024. [visited on 2024-04-07]. Available from: https://plokhotnyuk.github.io/jsoniter-scala/.
18. *sttp: the Scala HTTP client you always wanted!* [Online]. 2024. [visited on 2024-04-08]. Available from: https://sttp.softwaremill.com/en/stable/.
19. *Hello, Tapir ZIO ZIO HTTP* [online]. 2024. [visited on 2024-04-08]. Available from: https://pramodshehan.medium.com/hello-tapir-zio-zio-http-c5dca727e0ab.
20. *Amazon S3* [online]. 2024. [visited on 2024-04-08]. Available from: https://aws.amazon.com/s3/.
21. *Mastering the Art of Storage: A Comprehensive Guide to Amazon S3* [online]. 2024. [visited on 2024-04-08]. Available from: https://medium.com/@ismail.lamaakal/mastering-the-art-of-storage-a-comprehensive-guide-to-amazon-s3-80fc4d1a1efd.
22. *Apache Parquet* [online]. [visited on 2024-04-09]. Available from: https://parquet.apache.org/.
23. *Compression.md* [online]. [N.d.]. [visited on 2024-04-09]. Available from: https://github.com/pengfei99/ParquetDataFormat.
24. *Amazon Athena Features* [online]. 2024. [visited on 2024-04-09]. Available from: https://aws.amazon.com/athena/features/.
25. *Partitioning data in Athena* [online]. 2024. [visited on 2024-04-09]. Available from: https://docs.aws.amazon.com/athena/latest/ug/partitions.html.
26. *What Is a Graph Database?* [Online]. 2024. [visited on 2024-04-09]. Available from: https://www.graphable.ai/software/what-is-neo4j-graph-database/.
27. *What is PostgreSQL?* [Online]. 2024. [visited on 2024-04-10]. Available from: https://aws.amazon.com/rds/postgresql/what-is-postgresql/.
28. *Overview* [online]. 2024. [visited on 2024-04-11]. Available from: https://kubernetes.io/docs/concepts/overview/.
29. *Unleashing the Power of AWS ECS: Revolutionizing Container Management* [online]. 2024. [visited on 2024-04-11]. Available from: https://medium.com/@bilal325/unleashing-the-power-of-aws-ecs-revolutionizing-container-management-f02507300d7.
30. *APACHE KAFKA* [online]. 2024. [visited on 2024-04-11]. Available from: https://kafka.apache.org/.
31. *xxHash* [online]. 2024. [visited on 2024-04-10]. Available from: https://github.com/Cyan4973/xxHash.
32. *What is Prometheus?* [Online]. [visited on 2024-04-20]. Available from: https://prometheus.io/docs/introduction/overview/.
33. *What is Grafana?* [Online]. [visited on 2024-04-20]. Available from: https://www.redhat.com/en/topics/data-services/what-is-grafana.
34. *Aligned format* [online]. [visited on 2024-04-22]. Available from: https://neo4j.com/docs/operations-manual/current/database-internals/store-formats/.
35. *Block format* [online]. [visited on 2024-04-22]. Available from: https://neo4j.com/docs/operations-manual/current/database-internals/store-formats/.

Concents of the attachment

- `readme.txt` ..... brief description of the content of the SD card
- `src`
  - `totem-archiver` ..... Totem Archiver source code
    - `README.md` ..... brief description of Totem Archiver
    - `neo4j-procedures` ..... Neo4j custom procedures
    - `docs` ..... OpenAPI specification
- `thesis` ..... Source form of the thesis in LaTeX format
- `text` ..... thesis text
  - `thesis.pdf` ..... Text of the thesis in PDF format