FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Towards versioning profiles through time: A database benchmark

Dinis Ribeiro dos Santos Bessa de Sousa

Mestrado em Engenharia Informática e Computação

Supervisor: Tiago Boldt Pereira de Sousa

External Supervisor: João Azevedo

July 7, 2025

Towards versioning profiles through time: A database benchmark

Dinis Ribeiro dos Santos Bessa de Sousa

Mestrado em Engenharia Informática e Computação

President: Gabriel David
Referee: Ângelo Martins

July 7, 2025

Resumo

Os dados de perfil do utilizador desempenham um papel central nos sistemas modernos orientados por dados, em particular nas *Customer Data Platforms* (CDPs), onde são utilizados para personalizar as experiências do utilizador e conduzir campanhas de marketing direcionadas. No entanto, estas plataformas oferecem normalmente apenas o estado mais recente do perfil, ignorando a dinâmica temporal que pode revelar informações valiosas sobre o comportamento do utilizador e a evolução do perfil. Esta dissertação investiga a forma de armazenar dados de perfis de utilizadores ao longo do tempo, com o objetivo de apoiar a consulta histórica e a análise temporal, permitindo o acesso a estados anteriores e a revelar tendências.

O trabalho começa por examinar os conceitos fundamentais de tempo nas bases de dados, distinguindo entre tempo válido (quando os dados são verdadeiros no mundo real) e tempo de transação (quando os dados são registados no sistema). Os modelos de dados bitemporais, que integram ambas as dimensões, são explorados como um meio de captar a complexidade temporal. As bases de dados temporais, que preservam as alterações dos dados ao longo do tempo e suportam a consulta sensível ao tempo, surgem como candidatos promissores para representar perfis de utilizadores com versões. No entanto, apesar do seu potencial, poucos estudos avaliaram a sua adequação e desempenho neste contexto específico.

É efetuada uma revisão sistemática da literatura para determinar o estado da arte do modo como os perfis dos utilizadores são modelados em vários domínios. As representações vão desde modelos baseados em ontologias e conceitos até abordagens baseadas em grafos, destacando a diversidade estrutural e a riqueza semântica dos dados do utilizador. A análise identifica características únicas de perfis de utilizadores, como a heterogeneidade dos modelos e a evolução incremental dos dados.

Foi ainda efetuada uma análise do estado da arte das bases de dados temporais, centrada nos sistemas que oferecem suporte nativo para a semântica temporal, as consultas de viagem no tempo, o armazenamento imutável e a indexação temporal. Desta revisão, três sistemas representativos, PostgreSQL, XTDB v2, e TerminusDB, são selecionados para um benchmark empírico.

A principal contribuição desta dissertação é um *benchmark* que avalia a forma como estes sistemas armazenam e recuperam dados de perfis de utilizadores com versões. São comparadas duas estratégias de armazenamento: a baseada em estados, em que cada versão completa de um perfil é armazenada de forma independente, e a baseada em operações, em que apenas são guardadas as alterações efetuadas. Utilizando um *dataset* realista obtido através de dados anonimizados de uma *Customer Data Platform*, com mais de 14 milhões de atualizações em 713 milhares de perfis de utilizadores, avaliamos o desempenho do sistema nas dimensões mais relevantes, incluindo a eficiência do armazenamento, a latência de escrita e a capacidade de resposta a leituras.

Os resultados revelam compromissos entre estratégias de armazenamento e entre sistemas de bases de dados, expondo os pontos fortes e as limitações de cada abordagem. Em última

análise, este trabalho oferece uma avaliação das bases de dados temporais para o armazenamento de perfis de utilizadores, fornecendo informações práticas para sistemas que procuram incorporar temporalidade na gestão de dados de clientes.

Abstract

User profile data plays a central role in modern data-driven systems, particularly within Customer Data Platforms (CDPs), where it is leveraged to personalise user experiences and drive targeted marketing campaigns. However, these platforms typically offer only a snapshot of the most recent profile state, neglecting the temporal dynamics that can reveal valuable insights about user behaviour and profile evolution. This dissertation researches how to store user profile data over time, aiming to support historical querying and temporal analysis by enabling access to previous states and uncover trends.

The work begins by examining the conceptual foundations of time in databases, distinguishing between valid time (when data is true in the real world) and transaction time (when data is recorded in the system). Bitemporal data models, which integrate both dimensions, are explored as a means of capturing temporal complexity. Temporal databases, which preserve data changes over time and support time-aware querying, emerge as promising candidates for representing versioned user profiles. Yet, despite their potential, little research has assessed their suitability and performance in this specific context.

A systematic literature review is conducted to investigate how user profiles are modelled across domains. Representations range from ontology-based and concept-based models to graph-based approaches, highlighting the structural diversity and semantic richness of user data. The review identifies unique characteristics in user profile versioning, such as schema heterogeneity and incremental data evolution.

Additionally, a state-of-the-art review of temporal databases is carried out, focusing on systems that provide native support for temporal semantics, time-travel queries, immutable storage, and temporal indexing. From this review, three representative systems, PostgreSQL, XTDB v2, and TerminusDB, are selected for an empirical benchmark.

The main contribution of this dissertation is a benchmark evaluating how these systems store and retrieve versioned user profile data. Two storage strategies are compared: snapshot-based, where each complete version of a profile is stored independently, and operation-based, where only the differences between consecutive versions are retained. Using a realistic dataset derived from anonymised CDP data, comprising over 14 million updates across 713 thousand user profiles, we assess system performance across key dimensions, including storage efficiency, write latency, and retrieval time.

The results reveal trade-offs between storage strategies and across database systems, exposing the strengths and limitations of each approach. Ultimately, this work offers a comprehensive evaluation of temporal databases for user profile storage, providing practical insights for systems that seek to incorporate temporal reasoning into customer data management.

Acknowledgments

I would like to express my deepest gratitude to all those who supported and guided me throughout the journey of this dissertation and my academic life.

First and foremost, I sincerely thank my supervisors, Professor Tiago Boldt de Sousa and João Azevedo, for their invaluable guidance, encouragement, and insight.

A special thank you to Philippe Höij and Gavin Mendel-Gleason from TerminusDB, and Jeremy Taylor from XTDB, whose collaboration and availability made the development process smoother and helped me feel part of their communities.

To my mother, Paula, and my father, José – thank you for always having my back, and for being my role models and inspirations. To my brothers, Francisco and Rodrigo, from whom I learn every day, and who constantly challenge me to become a better person and a better engineer. To my grandmothers, Maria do Carmo and Lígia Maria – the bravest, funniest, and most loving people I have had the privilege of knowing.

To my girlfriend, Raquel – thank you for being by my side throughout this journey. Thank you for making me smile and for always bringing out the best in me.

To my friends from middle and high school, FEUP, Orfeão Universitário do Porto, and all others – thank you for the unforgettable memories and lifelong friendships. It wouldn't have been the same without you.

This dissertation is as much a reflection of your support as it is of my own efforts. Thank you all.

Dinis Ribeiro dos Santos Bessa de Sousa

"Time is the most undefinable yet paradoxical of things;
the past is gone, the future is not come,
and the present becomes the past,
even while we attempt to define it."

Charles Caleb Colton

Contents

List of Figures

List of Tables

Abbreviations

ACID	**A**tomicity, **C**onsistency, **I**solation, **D**urability
API	**A**pplication **P**rogramming **I**nterface
AWS	**A**mazon **W**eb **S**ervices
BRIN	**B**lock **R**ange **In**dex
CCPA	**C**alifornia **C**onsumer **P**rivacy **A**ct
CDP	**C**ustomer **D**ata **P**latforms
CRM	**C**ustomer **R**elationship **M**anagement
DBMS	**D**ata**b**ase **M**anagement **S**ystem
JSON	**J**ava**S**cript **O**bject **N**otation
GDPR	**G**eneral **D**ata **P**rotection **R**egulation
GIN	**G**eneralized **I**nverted **I**ndex
HDT	**H**eader-**D**ictionary-**T**riples
RDBMS	**R**elational **D**ata**b**ase **M**anagement **S**ystem
RDF	**R**esource **D**escription **F**ramework
RQ	**R**esearch **Q**uestion
SQL	**S**tructured **Q**uery **L**anguage
UUID	**U**niversally **U**nique **Id**entifier
XML	E**x**tensible **M**arkup **L**anguage

Chapter 1

Introduction

Contents

This chapter outlines the key contents of this dissertation. Section 1.1 describes the context of this work. Section 1.2 highlights the main motivations for this research. Thereafter, Section 1.3 highlights the main challenges present. Section 1.4 clarifies the main goal and enumerates the research questions that will guide this work. Finally, Section 1.5 presents the structure of the present document.

1.1 Context

Data has emerged as a strategic asset in the contemporary business landscape, driving decision-making and fostering innovation [8]. Across industry sectors, data is used to gain a competitive advantage over other businesses [9], creating business value by personalising user service and recommendations, as well as using targeted advertising [10].

Businesses increasingly accumulate vast amounts of data to extract valuable insights [11]. One notable trend is the use of historical data, where databases store not only a snapshot of current information but also a record of past states and changes that the data has undergone.

Though the importance of historical data is undeniable, storing it for its own sake can hurt the quality of data if done incorrectly. As data needs to be kept recent to provide accurate results, businesses push to increase data collection in order to gain a competitive advantage [12].

Given the growing emphasis on data-driven decision-making, data version control is rapidly gaining prominence as a critical industry focus area. Temporal databases are currently supported for both relational [13][14][15] and non-relational [16][17][18] databases. Temporal database

management systems enable storing and querying data at any point in time, as well as over defined time intervals [19].

1.2 Motivation

With the never-ending development of technology and amid the big data era, formulating effective marketing strategies has become key to business effectiveness [20].

Businesses increasingly recognise that their customers are more than just names and addresses; they are collections of information, embodied in user profiles.

User profiles consist of a collection of information associated with a user. In the marketing domain, profiles are stored in Customer Data Platforms (CDPs), which according to the CDP Institute is defined as "packaged software that creates a persistent, unified customer database that is accessible to other systems." [21]. This research, proposed by Kevel with their Customer Data Platform in mind, Kevel Audience [22], is motivated by the use cases of historical data in that domain.

User profiles are usually incrementally built based on user interactions with a brand. A profile can be a combination of user identifiers (such as cookies or email addresses) and attributes derived from the interactions with the brand (such as total purchase value or number of purchases) [23]. Information from multiple sources is unified and made accessible to other systems for analysis and managing customer interactions [21][24].

Digital marketing campaigns become more powerful and effective by relying on the data collected and provided by CDPs. Marketing campaigns are usually divided into three parts: segmentation, targeting, and positioning [25]. Digital marketing campaigns become more effective by relying on data to construct segments of customers and targeting the right users, reducing costs and improving the results of a campaign [26].

CDPs typically provide a snapshot of the most recent version of a user profile. However, there are compelling use cases for querying historical versions.

By enabling querying user profile versions, companies can detect past and emerging trends, leveraging this knowledge to anticipate future developments and optimise their strategies accordingly [27], as customer preferences and their segments are dynamic and ever-changing [28][29].

Versioning of user profiles provides an overview of a customer's evolution through time, which would allow to:

- **Analyse customer segment mobility over time**: Track how customers move between segments based on changing preferences or behaviours.
- **Identify the evolution of segments into new segments**: Understand how customer segments evolve and identify emerging trends.
- **Assess the impact of marketing campaigns on customer behaviour**: Evaluate how marketing campaigns influence customer preferences and purchasing decisions.

- **Optimise customer segmentation strategies**: Refine segmentation models based on historical data to identify more effective targeting groups.

Outside of the context of marketing and CDPs, the use of historical versions of user profiles can provide significant advantages in multiple domains:

- **Personalized recommendation systems**: Leveraging historical user data enables recommendation engines to identify preferences and enhance predictions. This approach improves user engagement and satisfaction by providing content that aligns with individual interests [30].

- **Customer retention**: By maximizing customer retention, a company can maximize its profits. Customer retention is a core issue of Customer Relationship Management (CRM), and historical data can enable better retention [31].

- **Regulatory compliance and auditing**: Maintaining detailed user data histories is essential for compliance with regulations such as the European Union's General Data Protection Regulation (GDPR)([32] and the California Consumer Privacy Act (CCPA) [33]. Temporal user profiles provide a robust audit trail, demonstrating adherence to data access, modification, and retention policies, thereby ensuring accountability and transparency.

- **Fraud detection and security**: Monitoring changes in user behaviour through historical data aids identifying anomalies and detecting fraud in domains like healthcare [34] or finances [35].

1.3 Problem

While versioning user profiles offers significant benefits, it comes at a cost. Developing efficient implementations of such systems is both time-consuming and prone to errors [36]. Each system requires a unique information model and unique constraints. There is also a general lack of information on best practices, since temporal systems are still under development.

Moreover, storage and retrieval costs associated with storing and accessing all the versions of data can be substantial, particularly for large-scale datasets [37]. It is vital to explore the trade-off between storage and retrieval: if one wants to lower storage space, it often comes at the cost of slower retrieval times [38].

Temporal database management systems, which can be used to save the incremental changes of user profiles, lack studies that explore their use cases, implementations, and comparisons, whether against non-temporal database approaches or among different temporal systems themselves. This lack of comprehensive research highlights an important area for further investigation.

1.4 Goals

Given the previously stated motivation and problem, this dissertation aims to explore the **versioning of user profiles in the context of user data management, researching efficient methods to store, query, implement, and maintain user profile versions over time**.

This work is motivated by the lack of versioning in CDPs, although it is expected that the findings in this dissertation can be extended to other domains of user data management like the ones described in Section 1.2, and possibly data in general.

The innovative aspect of this work lies in examining the unique challenges and opportunities that arise from the specificity of user profiles that provide unique constraints to work with.

The following research questions were made to conduct the research for this work:

RQ1: What is a user profile, and how can we represent user information?

This question aims to explore the models and representations of user profiles to better understand how a user profile can be represented.

RQ2: What unique constraints and characteristics are presented by user profile data when implementing versioning?

This question determines how the nature of user profile data imposes specific requirements that differ from other information pieces.

RQ3: What are the state-of-the-art open source database systems that implement temporal versioning?

This question aims to survey what open source database systems are suited for storing temporal data.

RQ4: Which metrics are commonly used to measure system performance?

This question explores how to evaluate a database system performance, by building on knowledge from database benchmarks.

1.5 Document Structure

This document is composed of 7 chapters:

- Chapter 1 outlines the context and motivation for this work, as well as presenting the general goals and research questions.
- Chapter 2 provides the necessary background on key concepts required to understand the present work.
- Chapter 3 reviews the state of the art on user profiles and temporal database management systems, ending in a discussion on how both relate and which databases can be more suited to versioning of user profiles.

- Chapter 4 presents the problem addressed in this dissertation, defining the scope, hypothesis, proposed solution, and methodology.
- Chapter 5 highlights the main considerations and assumptions that guided the development, and analyses the dataset used.
- Chapter 6 reveals the implementation details and the results, as well as a discussion of the results obtained.
- Chapter 7 reflects on the main findings and contributions of this work.

Chapter 2

Background

Contents

This chapter addresses fundamental concepts required to understand the present work. Section 2.1 describes different definitions of time, as well as unitemporal and bitemporal models. Afterwards, in Section 2.2 temporal databases are covered, presenting a definition, history and usual features. In Section 2.3, database management systems (DBMS), different models and their use cases are briefly explored. Section 2.4 highlights different methods for storing versions of information, while Section 2.5 focuses on the Customer Data Platform, outlining its role in effective marketing campaigns and the components that compose it. Finally, in 2.6 this chapter is summarised.

2.1 Time

Understanding how time is perceived and represented in information systems is the first step to introducing temporal versioning to a system.

Humans read and perceive time in different ways, using digital or physical clocks, the sun, an hourglass, or even the stars. When one wants to know "the time," there are various interpretations: what hour and minute does the clock say it is, how much time has passed, how much time is left, or when something happened.

Computers are no different. They represent time as a number (more precisely, a collection of bits) to represent the different notions of time, such as dates, intervals, or time since an epoch (time since a fixed event, one of the most common representations) [39].

The representations of time differ in their resolution (the smaller intervals of time they can represent), precision (relating to the smallest interval of time they can detect), and range of time they can represent (32 signed bits can represent up to 68 years, but when using 64 bits, 292.3 billion years in both directions, which is over twenty times the present age of the universe).

2.1.1 Dimensions of time

Time is multi-dimensional [40]. Valid time is the time an event was true in the real world, and transaction time concerns the time a fact was present in the database [41].

Both of these "times" can be thought of as more than just a timestamp but as a two axes of time that represent the period a fact was true in the real world and the period it was stored in a database until it was deleted. And so, both the valid time and the transaction time, when stored, can be thought of as intervals between two instants, indicating the start and end of a period.

In this section, we will distinguish both of these and explain how they can be used together, presenting the definitions appointed by *Christian S. Jensen et al.* in their "The Consensus Glossary of Temporal Database Concepts" document [42].

2.1.1.1 Valid Time

Valid time is the time when a fact happened or was valid in the real world, or simply when that fact was true. It is usually given by the user, application, or domain of the database system. The definition given by *Christian S. Jensen et al.* is the following:

> *"The valid time of a fact is the time when the fact is true in the modelled reality. A fact may have associated any number of instants and time intervals, with single instants and intervals being important special cases. Valid times are usually supplied by the user."* [42]

2.1.1.2 Transaction Time

When a transaction occurs in a database or when a fact is registered in a database until it is deleted. It is registered automatically by the database system and is related to its system time. *Christian S. Jensen et al.* presented this definition of transaction time:

> *"A database fact is stored in a database at some point in time, and after it is stored, it is current until logically deleted. The transaction time of a database fact is the time when the fact is current in the database and may be retrieved. As a consequence, transaction times are generally not time instants, but have duration. Transaction times are consistent with the serialization order of the transactions. They cannot extend into the future. Also, as it is impossible to change the past, (past) transaction times cannot be changed. Transaction times may be implemented using transaction commit times, and are system-generated and -supplied. While valid times may only be associated with "facts," statements that can be true or false, transaction times may be associated with any database object."* [42]

This definition presents a key property of transaction time: it is bounded on both ends [43]. Transaction time starts when the database is created and cannot extend past the current time (now). Also, it is immutable as it refers to the time a transaction was recorded, and that does not change with time and cannot be altered by users.

Conversely, valid time may be bounded or unbounded (although many applications assume a bound or a range of time) [43]. Unlike transaction time, valid time can be modified retrospectively to correct inaccuracies or align with updated knowledge about when events occurred in the real world.

It is evident that these two time dimensions are not homogeneous; valid time and transaction time have distinct semantics. Although generally some application-dependent correlations between the two times exist, valid and transaction times are proved to be orthogonal [19][40] .

2.1.2 Unitemporal model

As the name suggests, the unitemporal model uses a one-axis representation of time, using either transaction or valid time [44].

Without a temporal model, one can only retrieve from the database the facts that are currently stored in the said database.

Using a unitemporal model, with, for example, the transaction time as the time axis, the history of all the facts that were stored in the database can be retrieved, as can the time period that they were present in the database.

Looking at a practical example: *"John Doe lived in Porto at the beginning of 2024 and moved to Lisbon on January 15th."*.

There are two facts that can be taken from this example:

1. Jonh Doe lives in Porto.

2. John Doe lives in Lisbon.

The reality of John Doe's residence is depicted in Figure 2.1 as it would be in a system that provides a valid time relation, that is, a relation that only accepts valid time [42].

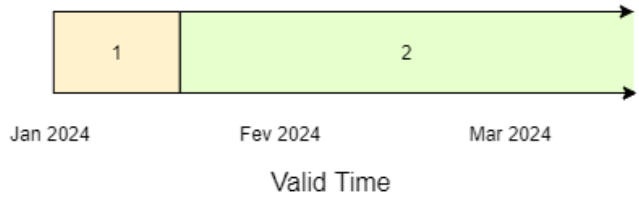

Figure 2.1: Valid time relation of Jonh Doe's residence registry

However, John Doe did not register his house change with the government until the 15th of February. Now the example can be extended to the following: *"John Doe lived in Porto at the beginning of 2024 and moved to Lisbon on January 15th. On February 15th, he registered his residence change with the governing bodies."*.

The facts continue to be the same, although from the government's point of view, if using only transaction time, the records on the database only show John Doe living in Lisbon after the 15th of February.

A representation of the transaction time relation of John Doe's residency is represented in Figure 2.2, as it would be in a system that only supports transaction time [42].

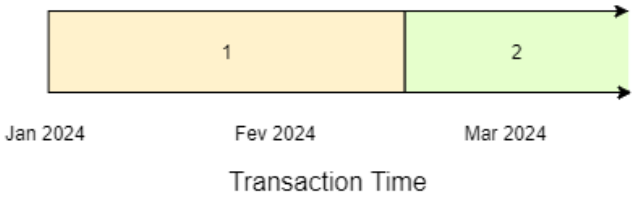

Figure 2.2: Transaction time relation of John Doe's residence registry

Using only one axis to measure the time, information can be lost. In some domains and use cases, there is the need to distinguish and store more than one dimension, introducing bitemporality.

2.1.3 Bitemporal model

A bitemporal model uses two axes of time, the valid time and the transaction time [44]. Using a bitemporal model to represent the previous example, one can preserve both the date when John moved to his new residence and the date when that change happened, as represented in Figure 2.3. The red point in the figure is the moment the change of residence is stored in the database.

This way we can faithfully represent the events and its history:

- Until February 15th, the government database indicates that John lives in Porto (indefinitely)

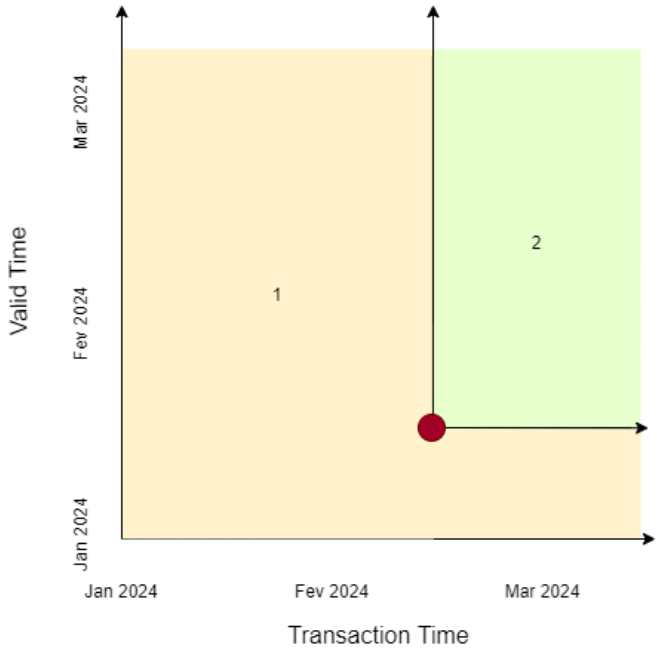

Figure 2.3: Bitemporal relation of John Doe's residence registry

- From February 15th onwards, the government database indicates that John moved out of Porto on January 15th and lives now in Lisbon.

Using a bitemporal model, we can respond to queries that ask for a portion of time (or portions of times): "as of 15th of March (system time), in how many different residences did John live between the first trimester (valid time)?".

Integrating both time dimensions at the row level enables a database to precisely capture an organization's knowledge at any point in time, even when data originates from diverse systems and undergoes various processing delays. This also provides an auditable history of how that information changed, eliminating the need for separate snapshots or data duplication [1].

This section on temporal models ends with a representation of a bitemporal relation from *Richard T. Snodgrass and Ilsoo Ahn*, which highlights a more complex example of multiple actions represented in a bitemporal model.

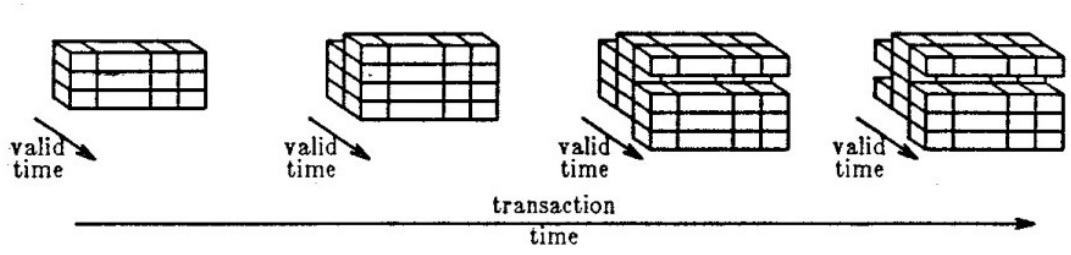

Figure 2.4: Bitemporal relation representation. From: [1]

Figure 2.4 reflects a bitemporal relation and the evolution of tuples over time. Each tuple that is stored is represented as a row of blocks. Starting from a null relation, the figure higlights changes over the course of four transactions: "*(1) three tuples were added, (2) one tuple was added, (3) one tuple was added and an existing one deleted, and (4) one tuple was modified so that it started at a later valid time*" [1].

The first step in creating reliable temporal models that faithfully capture reality is to comprehend time and its constraints and possibilities. Following the discussion of time and temporal models, the next section presents information on temporal databases.

2.2 Temporal Databases

In a traditional database, although its contents may continue to change over time, the changes are modifications to the state, with the old data being deleted from the database. The contents of a database are seen as a snapshot of the current reality [43].

In contrast, temporal databases manage data that changes over time, making it possible to store, query, and retrieve data in both present and the past. Immutability, the concept that guarantees past states of objects are not replaced but retained, closely relates to temporal databases.

This section will cover what are temporal databases, their history, and what are common features of one.

2.2.1 Definition

According to *Christian S. Jensen et al.*, "A temporal database is a database that supports some aspect of time, not counting user-defined time." [42] This definition entails that a temporal database must inherently incorporate time-related features into its structure and functionality, beyond user-defined or application-level timestamps.

Temporal databases natively support one or multiple temporal models discussed previously, either unitemporal (Section 2.1.2) or bitemporal (Section 2.1.3).

2.2.2 Historical Development

The foundations of temporal databases began taking shape in the 1970s, driven by a growing need to manage and query time-sensitive data effectively.

Concerns to traditional database systems ranged from losing past states of data due to updates made to the database [45] and the lack of tools to effectively handle time and its dimensions [43]. There was the need for an immutable database that allows queries over temporal dimensions.

Due to this, relevant research work was conducted on the topic. Works ranged from the definition of temporal concepts, the development of temporal query languages or extensions to current query languages, and the subsequent adoption of a temporal standard in the Structured Query Language (SQL) [46].

2.2.2.1 Early Developments and Temporal Concepts

In the early days of temporal database design, research was focused on time-related concepts and their formalization.

Several definitions of valid time, transaction time, and other core temporal concepts were presented [41], at first somewhat independently, but with time efforts were made to arrive at a consensus among the community [44].

Multiple temporal data models were also presented, differing in the temporal model supported (valid time, transaction time, or bitemporal) and the mechanisms used to store each of the time dimensions. In a data model, *chronon* is the term used to describe a non-decomposable time interval of some fixed, minimal duration [42]. These temporal data models varied in how they used one, two, three, or a set of chronons to represent time intervals on the axis of time [19].

2.2.2.2 Temporal Query Languages

The introduction of temporal query languages was another hot research topic during this period. There were multiple proposed extensions to existing query languages, such as TQuel [47], TSQL2 [48], and Temporal Datalog [49], defining keywords, constraints, and features needed to achieve temporal support.

Despite the multiple proposed extensions, it took quite some time for the ISO SQL committee to accept a language extension for temporal data. The SQL:2011 standard published in December 2011 introduced the ability to create and manipulate temporal tables [13].

One of the most fundamental additions was the ability to define intervals of time by a start and end timestamp with the `PERIOD FOR` keyword.

SQL:2011 included the ability to define system-versioned tables to manage transaction times, valid time period tables, and bitemporal tables. It also added the ability to add temporal constraints to ensure, for example, that valid times did not overlap for a row (system time does not share this concern, as the system handles it). This standard also extended the query capabilities by adding period predicates to express conditions involving periods.

The introduction of this temporal extension to SQL was a milestone in temporal query languages and provided a standardized framework for managing and querying time-sensitive data. However, there were multiple opportunities for improvements, such as support for period joins and support for period aggregates and grouped queries [13].

2.2.3 Features of Temporal Databases

Temporal databases incorporate features that enable them to store, manage, and query time-sensitive data efficiently. These features are designed to be additions to extend the capabilities of non-temporal databases [13][50]:

- **Support for Time Models:** Temporal databases support temporal models natively, namely unitemporal (Section 2.1.2) or bitemporal (Section 2.1.3) models.

- **Temporal Constraints**: The support for temporal models should go beyond the storage of timestamps or the existence of ranges like the ones implemented by `PERIOD FOR`. Temporal databases should enforce rules to maintain data integrity, providing mechanisms to ensure non-overlapping intervals in valid periods for the same entity and referential integrity across temporal data [13].

- **Temporal Data Types:** Temporal databases support specialized data types to facilitate temporal operations. Some of these may be common in non-temporal databases (`DATE, TIME, TIMESTAMP`) but other types may be added.

- **Time-Travel Queries:** Time-travel functionality allows the users to query the historical states of the data. This functionality enables retrieving data as it existed at a specific point in time (point-in-time queries) or over a period of time, on the supported time dimensions.

- **Temporal Indexing:** Temporal databases can implement indexing mechanisms optimized for time-based queries. Temporal indexes (e.g., interval trees or segment trees) improve the efficiency of retrieving data for specific time ranges or points, reducing the overhead associated with the retrieval of historical data [51].

- **Temporal Joins:** Temporal joins extend traditional join operations by including temporal conditions. For example, they allow combining data from different tables only when their time intervals overlap or meet specific temporal criteria.

- **Recovery Mechanisms:** Temporal databases can also support recovery mechanisms to restore a past database state in case of errors or accidental modifications.

These features provide the foundational capabilities that distinguish temporal databases from their non-temporal counterparts. After having gone through the definition, history and features of temporal databases, in the next section, different DBMS models will be explored.

2.3 Database Management Systems

In this section, we examine database management systems (DBMS), their defining characteristics, and how they differ. We will discuss their primary features and use cases, highlighting examples where specific types of DBMS excel. This analysis provides a foundation for understanding how different DBMS can be applied to various data management scenarios.

A DBMS is a software that allows users to store, manage, and retrieve information from a database efficiently [52]. It provides an interface for users and applications to access and manipulate data, typically using a query language such as SQL. Beyond simple data storage and retrieval, modern DBMS often support advanced features including transaction management, data security, concurrency control, and scalability [53].

2.3.1 Relational DBMS

Relational databases divide information into fixed relations (the tables of a database), enforcing a pre-defined schema for all of them. Each table, or relation, consists of rows (tuples) and columns (attributes), where each column enforces a specific data type. This structured approach facilitates complex querying and data manipulation using SQL, a standardized language for managing and manipulating databases.

One of the fundamental principles of Relational Database Management Systems (RDBMSs) is the enforcement of data integrity through constraints such as primary keys, foreign keys, and unique constraints. Primary keys ensure each record within a table is unique, while foreign keys maintain referential integrity between tables [54].

Relational DBMS usually support ACID (Atomicity, Consistency, Isolation, Durability) properties, which are essential for reliable transaction processing [55]. These properties ensure that all database transactions are processed reliably, maintaining the integrity of the database even in cases of system failures or concurrent access. This makes RDBMS a preferred choice for applications requiring robust transaction management, such as financial systems, order processing, and customer relationship management.

2.3.2 Document DBMS

Document databases are a type of non-relational database that stores data in the form of documents, typically using formats like the JavaScript Object Notation (JSON) or the Extensible Markup Language (XML). Unlike relational databases, document databases do not enforce a strict schema, allowing for flexibility in data modelling. Each document contains a set of key-value pairs, enabling the storage of complex and hierarchical data in a single record [56].

Comparing to relational databases, documents can provide more flexibility, high scalability, and fast read-write performance [57]. The schema-less nature of document databases is particularly advantageous for applications where data structures may evolve over time or vary across records, as it makes it possible to add new fields and modify existing ones over time.

Document databases are commonly used in web and mobile applications to store various forms of content, including articles, blog posts, and multimedia files. User profiles and personalization also significantly benefit from leveraging document databases, as they can offer a flexible and efficient approach to storing user profiles, accommodating diverse data fields, user preferences, and personalized settings seamlessly [58].

2.3.3 Graph DBMS

Graph databases are a type of database designed to store, query, and manipulate data represented as nodes, edges, and properties. This structure is particularly well-suited for modeling complex relationships and interconnected data. In a graph database, nodes represent entities, edges represent relationships between entities, and properties provide additional information about nodes or edges [59].

Unlike relational databases, which require complex joins to query relationships, graph databases enable the direct traversal of connections. Graph databases are, for that reason, suited for queries involving deep or complex relationships.

Social networks, where users are represented by their multiple relationships with other users, such as their friends or their followers; or recommendation systems, by the ease of representation of user-item interactions and between item relationships, are domains where graph databases are frequently used [58].

2.4 Versioning Strategies

This section explores the three primary strategies used to store and manage versions of data over time: operation-based, snapshot-based, and differential versioning. Each approach represents a different trade-off between storage overhead, retrieval performance, and flexibility in reconstructing historical states.

Snapshot-based versioning (also refereed to as full versioning [60]) stores complete states of data at specific points in time [61]. This approach simplifies data retrieval and is well-suited to workloads where frequent access to past versions is necessary. However, it can incur significant storage overhead, particularly in systems with large datasets or frequent updates.

Operation-based versioning captures changes as discrete operations rather than storing complete states. It is usually accompanied by a log that stores all the modifications made to an object. This method is particularly advantageous in systems where the frequency of small updates is high, and where it is desirable to replay a given sequence of changes to reconstruct any version of the data [62].

Differential versioning strikes a balance between the two previous approaches by storing the differences (or deltas) between successive states [63]. This method aims to reduce storage usage while retaining relatively fast reconstruction times. By chaining or layering deltas, differential systems can recreate any past state without needing to store every one in full [60]. However, the differential model does have to have further considerations, such as choosing distance between intermittent full states and designing the strategy to calculate the differences.

Together, these strategies represent the design space of versioning systems. Selecting the appropriate model depends on the specific workload and information needs, so that the versioning model suits the requirements of the system [60].

2.5 Customer Data Platform

Customer Data Platforms (CDPs) are specialized systems that consolidate and manage customer data to enable personalized marketing and engagement strategies. They collect and unify data from multiple sources and make it available to other systems for analysis and managing customer interactions [24][21].

To ensure that marketing campaigns remain feasible without requiring the sharing of personal user data with marketing platforms, CDPs assist businesses in gathering, analysing, segmenting, and anonymously activating their data on third-party platforms [64].

CDPs are made to combine data from many sources and provide detailed customer profiles, in contrast to standard Customer Relationship Management (CRM) systems, which mainly focus on transactional data [65].

Creating strong customer segments is vital to ensuring the effectiveness of marketing campaigns [66]. Customer Data Platforms enable brands to improve their marketing campaigns' effectiveness with the use of customer segmentation and maintain first-party data anonymously.

2.5.1 Key Components and Functions

To understand a CDP's function and operation, one should look into its components.

CDPs can be divided into five main components that interact with each other to collect data, store user profiles, and make user information available to third-party marketing platforms [2]. The five primary components are the following:

Event Tracking: The foundation for capturing customer data across multiple platforms. Records actions such as clicks, purchases, and product views. This raw event data serves as the basis for creating user profiles.

ID Matching: CDPs employ ID matching to unify customer data from disparate sources and devices. By linking platform-specific identifiers with customer-specific attributes such as email addresses, they ensure seamless integration of data across various channels. This step is critical for accurately mapping multi-device and multi-platform customer journeys.

User Profile Storage: Consolidated user data is stored as profiles in a centralized database. These profiles include both static attributes, such as demographic information, and dynamic attributes, such as recent activity. User profiles will evolve incrementally as events are tracked.

Segmentation: Segmentation enables marketers to group users into audiences based on defined rules, such as purchase history or geographic location. Boolean expressions and machine learning models can be applied to create sophisticated segments customized to specific campaigns.

Activation: The final step involves activating user profiles and segments by integrating with external platforms. This process ensures efficient and targeted advertising without sharing sensitive user data. Integration is typically achieved using hashed identifiers or anonymized IDs.

The interactions between the different components of a Customer Data Platform are represented in Figure 2.5.

To sum up, user profiles are built incrementally using the facts gathered from the event tracking component [2]. Events are the interactions of customers (clicks, purchases or product views).

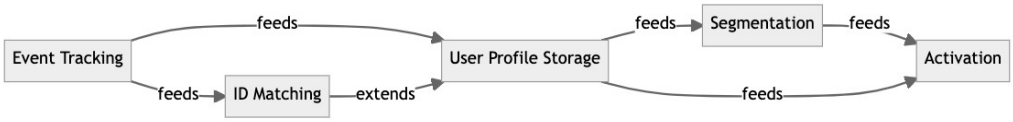

Figure 2.5: Customer Data Platforms Components and how they interact with each other. From [2]

These events produce facts that are stored in the database (the customer last viewed a product on the 14th of April and has a total purchase sum of 300€). A user profile is the collection of facts associated with a given user [23], and the id-matching component ensures that even if a user has multiple identifiers they are integrated in the same profile. The user profiles are then used to create customer segments and activated for marketing campaigns. The activation is the action of providing the advertisement platform with the user data, so it can be later used to show targeted ads.

2.5.2 User Profile Versioning in Customer Data Platforms

Recalling the motivation for the current work, enabling temporal versioning to user profiles in the context of user profile management, the main focus of this work will lie in the user profile storage component.

The storage component should be adapted to store temporal versions of the user profiles, so that user history and evolution can be taken into account when applying segmentation.

The models or techniques used for segmentation and activation are beyond the scope of this work; however, data should be readily available to these components.

2.6 Summary

This chapter outlined the foundational concepts required for understanding the work presented in this dissertation. It began by exploring the concept of time and its representation in information systems, emphasizing temporal databases and their unitemporal and bitemporal models. Different types of database management systems were reviewed, discussing their use cases and advantages, as well as different strategies to version data over time. Finally, the role and components of Customer Data Platforms (CDPs) were introduced, highlighting their functionality and relevance in managing user profiles, and a description of the components that serve as building blocks for the system was made. These concepts collectively establish the groundwork for the subsequent chapters.

Chapter 3

State of the Art

Contents

This chapter reviews the state of the art for both representations of User Profiles and Temporal Databases. Section 3.1 describes the Literature Review conducted on user profiles, and Section 3.2 the findings of the Literature Review and different user profile models. Temporal databases are described and compared in Section 3.3. Then, a discussion is made on the most suited databases for storing temporal versions of user profiles in Section 3.4, followed by Section 3.5, where the conclusions on the state-of-the-art study are presented.

3.1 Systematic Literature Review

A systematic literature review was conducted to assess the state of the art within the domain of this dissertation and to collect data that would address the research questions **RQ1** and **RQ2**. Conducting a literature review is a fundamental step in identifying current challenges and knowledge gaps in the field of study [67]. By employing explicit and systematic methods, the literature review process becomes reproducible and minimises the potential for bias [68].

3.1.1 Methodology

To implement the literature review, the following steps were taken:

1. **Formulate Research Questions:** Define the research questions to guide the review.
2. **Data Sources:** Select data sources to be included in the review.
3. **Inclusion and Exclusion Criteria:** Specify the inclusion and exclusion criteria based on relevance.
4. **Query Definition:** Build a set of search queries that express the information needs.
5. **Screening for Inclusion**: Review the relevance of documents retrieved based on title, abstract, keywords, and conclusions.
6. **Full-text Analysis**: Perform a comprehensive examination of selected documents to extract pertinent findings.

3.1.2 Research Questions

The research questions, identified in Section 1.4, aim to structure and direct the investigation of this dissertation. The research questions addressed in this review were **RQ1** and **RQ2**:

RQ1: What is a user profile, and how can we represent user information?

RQ2: What unique constraints and characteristics are presented by user profile data when implementing versioning?

3.1.3 Data sources

The literature review process used the ACM Digital Library and IEEE Xplore as digital libraries to obtain documents.

The ACM Digital Library allows access to the complete collection of ACM publications, whereas IEEE Xplore contains scientific and technical articles published by IEEE and its publishing partners.

3.1.4 Base references

To back up the review, the survey *User Modeling and User Profiling: A Comprehensive Survey* by Purificato *et al.* (2024) [69] was used, which provides a detailed overview of the evolution, techniques, and applications of user modelling and profiling. The survey explores historical and modern approaches, offering information on representations and use cases across various domains, such as recommendation systems, cybersecurity, and personalized education. The user profile models were classified according to it, and the findings from the Literature Review were complemented by the research presented in the survey.

3.1.5 Inclusion and Exclusion Criteria

The inclusion and exclusion criteria for **RQ1** and **RQ2** defined for the review are described in Table 3.1

Table 3.1: Inclusion and Exclusion Criteria for Literature Review

	Inclusion Criteria
IC1	Must have user profiles as the main focus
IC2	Publications from the last ten years (2014-2024)
IC3	Proposes or describes a user profile model or representation.
	Exclusion Criteria
EC1	Full-text access to the publication is not available
EC2	Duplicate publication
EC3	Fails to present a technique for modelling or representing user profile or user information
EC4	Describes an AI model for generating recommendations for users, without describing the user model itself

Using inclusion and exclusion criteria will help streamline the review process by establishing objective benchmarks for selecting studies that directly contribute to answering the research questions.

3.1.6 Search Queries

For **RQ1**, and **RQ2** the search query used on ACM Digital Library is presented on listing 3.1 and the search query used for IEEE Xplore is presented on listing 3.2. Although simple, these queries yielded the most relevant results for responding to **RQ1** and **RQ2**, given how common the keyword "user profile" is.

```
1    "query": { Title:("User Profile?")
2    AND Title:("Representation" OR "Model*")}
```

Listing 3.1: RQ1 Search Query for ACM Digital Library

```
("Document Title":"User Profile?") AND ("Document Title":"
    Representation" OR "Document Title":"Model*")
```

Listing 3.2: RQ1 Search Query for IEEE Xplore

3.1.7 Review Process

The review process outlined in Figure 3.1 can be described as follows:

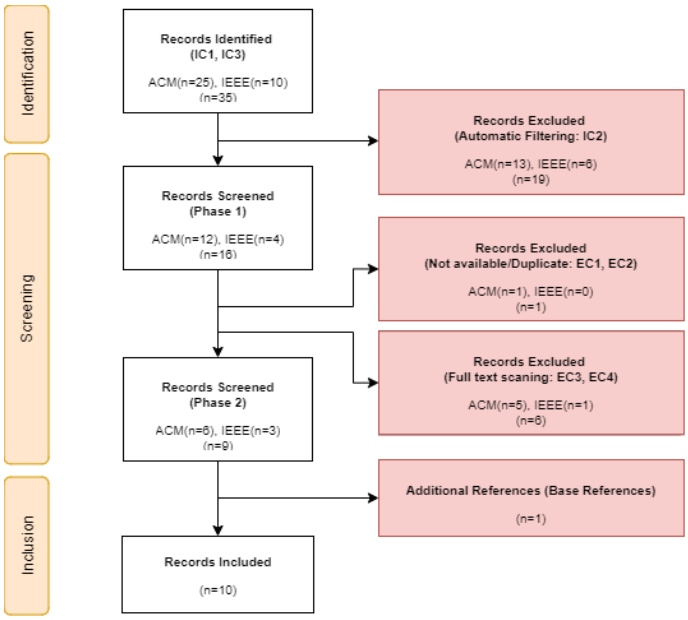

Figure 3.1: Systematic literature review

Identification: Documents were retrieved from the chosen digital libraries by using the queries presented in Section 3.1.6, respecting the inclusion criteria **IC1** and **IC3**.

Screening:

1. **Automatic filtering (IC2):** In both digital libraries, results were filtered by date, keeping results comprehended between 2014 and 2024.
2. **Filtering by title, abstract and keywords (EC1-EC4):** Removed duplicates between libraries and documents without full test access (**EC1 and EC2**) and remove results that do not present actual models or representations of user profiles (**EC3 and EC4**).
3. **Filtering based on full-text scanning (IC1 and IC3):** Removed documents that did not had user profiles has the main focus or did not discuss a model or representation of a user profile.

Inclusion: The missing base reference was added to the results to classify the retrieved user profile models.

3.2 User Profiles

This section delves into the collected information on user modelling and representation, focusing on user profile representations and the specific contexts where different representations excel. By analysing how profiles are structured, this discussion highlights the strengths and weaknesses of each approach.

3.2.1 User Profile Representations

User profile representations have evolved to encapsulate the diversity of user attributes, behaviours, and preferences. These representations typically include structured or semi-structured formats, allowing for efficient storage, retrieval, and analysis [23].

Each approach to user representation emphasizes different aspects of user data: ontology-based representations focus on semantic relationships, concept-based representations prioritize topic modelling and context extraction, and graph-based representations excel at capturing complex relationships and interactions. These methodologies provide unique benefits and are often applied based on the system's specific needs [69].

Specifically for CDPs, where customer data arises from numerous different systems, one must choose a user profile representation powerful enough to encompass all the user data collected.

Below, each of these paradigms is explored in detail, showing how they cater to diverse scenarios.

3.2.1.1 Ontology-based

Ontology-based representations are used to encapsulate the relations between concepts and entities of one or many domains [70]. In user profiles, ontologies better represent the relationship between terms (from synonymy to more complex relations like ownership or friendship).

In the context of user profiles, Rimitha *et al.* [3][71] proposed and tested an ontology to better represent user preferences for a job recommendation. This structure facilitates job search through semantic alignment with job descriptions. Similarly, Niyazova *et al.*[72] proposed an ontology tailored to social networks, capturing user relationships, activities, and interests to improve both security and expressivity.

In Figure 3.2, we can see a representation of an ontology model for the context of a job search. This ontology demonstrates what objects exist in the job search domain, what attributes they have and how the relationships between the objects.

Ontology-based representations excel in creating interpretable profiles with semantic clarity in a specific domain. They enable advanced reasoning and system integration through standardized vocabulary. However, they require a learning system method or manual intervention to create and maintain the ontologies [69].

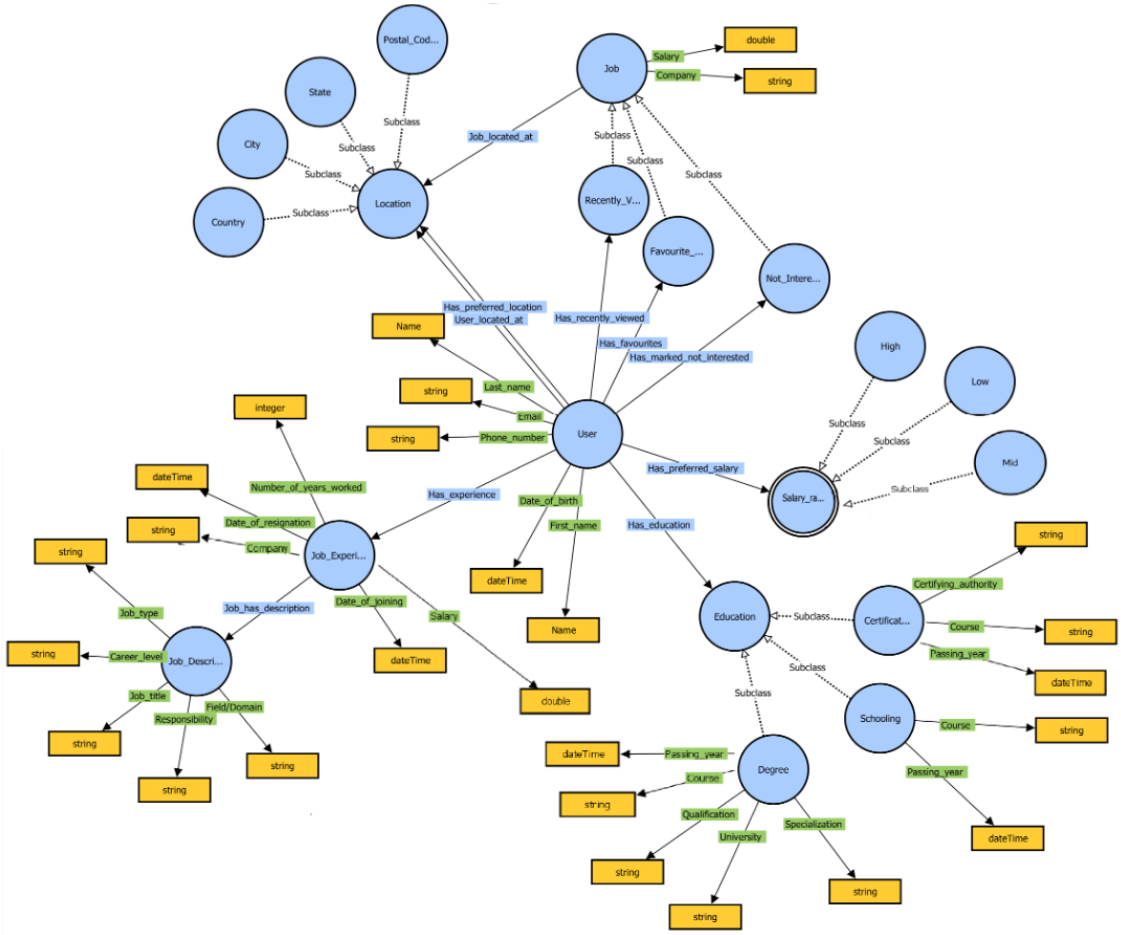

Figure 3.2: Metamodel of Job Search Ontology. From: [3]

3.2.1.2 Concept-based

Concept-based representations, like ontology-based representations, leverage the idea of representing information through concepts, entities, and the relationships between them. [69].

However, they extend the idea of relationships by capturing the hierarchy between the entities and assigning weights that indicate the user's interest in each topic, as is done in Khanthaapha *et al.* proposed model to recommend points of interest [4]. In this work, both user profiles and points of interest (the example of a restaurant as a point of interest is given) are represented as documents, from which topics are derived. The derived topics of documents used in Khanthaapha *et al.* research are in Figure 3.3. These topics are extracted from both users and points of interest, transformed into vectors, and their similarity is compared to be used for recommendations.

User profiles have also been proposed for healthcare, representing user characteristics as a collection of attributes distributed by different topics. Similarly, user profiles can be used as embedded vectors by applying weights to the different topics, enabling the use of machine learning models [73].

Figure 3.3: Word cloud of topics derived from user documents. From: [4]

3.2.1.3 Graph-based

Graph-based representations in the context of user modelling serve to store relationships with entities. Entities are depicted as nodes, and relationships between them are the edges of the graph. These models are particularly common in social networks, where they are used to represent the relationships and interactions among users.

In 2015, Han *et al.* proposed a graph-based method to represent a user profile, using data derived from a social tagging system (known as folksonomy) to represent user interests, taken from various sources using clustering [74]. Graph-based representations are also used to store triples, like the RDF (Resource Description Framework) triples. One example is the system proposed by Luo *et al.*, which combined user history with RDF data from the web to build a user profile for a recommender system [75].

More recently, Olfa Slama [5] presented a user profile model that combines the graph representation of triples and the idea of fuzziness to relations and preferences, applying it to the movie preference domain. Fuzziness enabled more flexible querying of RDF data by adding a degree to some relations to better represent their importance and using an extension of SPARQL that enables fuzzy querying [76].

Figure 3.4 represents a user profile sub graph in the domain of movie preferences. The user, Anna, possesses attributes (like age, gender, or location) and is the subject of relationships that represent her interest (she likes recent movies and considers a movie to be recent if it was filmed between 2013 and 2018).

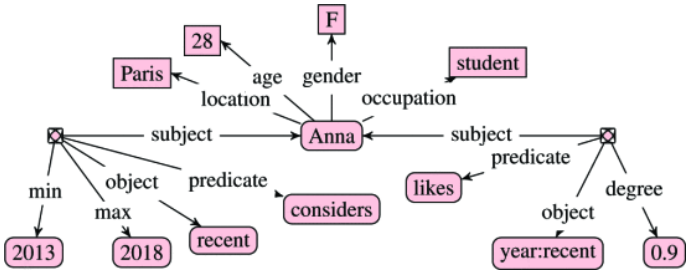

Figure 3.4: Representation of a user in a movie search system. From: [5]

This model excels at capturing intricate relationships, dependencies, and interactions within user profiles. This graph-based model facilitates the representation of user profiles, making it suited for scenarios where the modelling of complex relations is crucial, such as personalized recommendations, semantic reasoning, or dynamic social network analysis [69].

3.2.2 User Profile Uniqueness

Because of their specifics and the wide range of applications they can be used for, user profiles constitute unique pieces of information. They can aggregate information such as identifiers, demographic details, preferences, and interactions, making them highly specific not only to the individual user but to the domain itself [23].

Profiles are built explicitly by relying on information provided by the user, using online forms or questionnaires. Implicit user profiling is done by collecting the interactions and behaviours of the user without requiring direct input. More often than not, user profiles are created using both explicit and implicit user information in a hybrid manner, using the information given by the user as the starting point of the user profile and incrementally adding information as more interactions of the user are implicitly collected [69]. User profiles are so inherently dynamic and are expected to reflect the changes in users' behaviours, preferences, and interactions over time.

They may also be built with a variety of information sources and data formats, such as text, images, and audio, to represent the user [77]. Integrating such diverse data formats presents several challenges, including resolving inconsistencies, duplication, and missing data and ensuring the semantic alignment of data attributes.

Because the same user may be identified by a phone number in one system, an email address in another, and a cookie in yet another, it can even be problematic to identify the user. To reconcile these identifiers into a single profile, complex matching algorithms are needed, which frequently look beyond identifiers to determine whether two profiles refer to the same individual [78]. Furthermore, the continuous influx of data from real-time systems adds complexity to maintaining synchronization and data quality across sources.

User profiles must also adhere to certain usage and storage guidelines, and data privacy laws such as the California Consumer Privacy Act (CCPA) [33] and the General Data Protection Regulation (GDPR) in the European Union [32] must be followed.

Another factor contributing to user profiles' distinctiveness is the range of domains in which they are utilized. Uses of user profiles in recommendation systems [4][74], information retrieval [5], healthcare [73], and social networks [72] have all been demonstrated in this section.

3.2.3 Storing User Profiles

The models described in the previous section can be stored in various formats, depending on the data itself and the needed querying capabilities. The same piece of information can be represented as a combination of tables in a relational database, for transactional uses, a collection of documents in a document database if the data is not structured, or entities and relationships in a graph database

if relationships are the focus. The storage depends more closely on the domain, the structure of the information, and the workloads expected.

However, we need not just one version of the user profile, but its evolution through time. With that in mind, in the next section, different implementations of temporal database management systems will be covered, and a discussion will be made on the capabilities of each to adapt to the representation of users.

3.3 Temporal Databases

In this section, we will explore state-of-the-art temporal databases. A background in temporal databases is previously covered in Section 2.2.

This section will discuss databases that both make the implementation details of the temporal model available and are open-source, so they can be used later for evaluation. Database systems' more complete and up-to-date descriptions are found in their documentation and websites, which were used to collect database information.

Non-relational and relational temporal databases will be covered and explained, and a comparison between the database systems will be made.

3.3.1 Research Questions

This section aims to answer the research questions **RQ3** and **RQ4**.

Recovering the questions previously identified on Section 1.4:

RQ3: What are the state-of-the-art open source database systems that implement temporal versioning?

RQ4: Which metrics are commonly used to measure system performance?

3.3.2 Non-relational

In this section, we will cover temporal non-relational databases. These systems tend to be more flexible in the way they structure data than their relational counterpart. Non-relational databases excel at capturing and querying the intricate, time-evolving relationships between entities and their attributes, making them invaluable for applications involving both structural complexity and temporal dynamics.

3.3.2.1 TerminusDB

TerminusDB [16] is an open-source document graph database. It adheres to the Resource Description Framework (RDF) standard, storing data as triples representing objects and their relationships. As of the writing of this work, TerminusDB is currently on version 11, and the description corresponds to that version [16].

TerminusDB is suited for distributed and collaborative environments, where users communicate with the system trough a Git-like version control model, where every update is a commit, and push/pull/clone instructions are used to communicate differences between nodes. The time of the change is stored in the metadata of the commit, and so point-in-time queries can be made by accessing the current commit at a specific time [79]. TerminusDB provides even the ability to branch from and to different commits, and go back in time using rebase.

The database employs the Header-Dictionary-Triples (HDT) model, developed by Martinez-Prieto *et. al*, a compact RDF serialization approach designed to optimize storage and query efficiency. [80].

TerminusDB's architecture is graph-based, using nodes and edges to structure data and enforce model constraints. Updates are managed through a Git-inspired delta encoding mechanism, where changes are stored as layers incrementally built upon a base layer, recording added or deleted triples. This approach preserves immutability and facilitates efficient version control.

Additionally, TerminusDB can support data compression to optimize query performance and reduce storage requirements, offering a trade-off between storage efficiency and potential information loss in scenarios where that is acceptable [81].

Unlike other graph databases, TerminusDB enforces schema. Although this may limit the flexibility of the database, the use of a defined schema constrains the types of links, serves as documentation, offers both human and machine-readable semantics, and guarantees that software conforms to specified formats [82].

TerminusDB can be accessed via Javascript or Python Clients, a JSON Application Programming Interface (API) or even non-programmatically trough dashboards. After inserted, documents, one of the most common data formats on the internet, are connected in a graph, providing great query capabilities for highly connected data.

This approach ensures data immutability and facilitates time-travel queries while maintaining storage efficiency, making TerminusDB particularly well-suited for collaborative and versioned data management scenarios.

3.3.2.2 Datomic

Datomic [83] is a distributed and immutable database that emphasizes immutability, time travel, and rich query capabilities. At its core, Datomic stores data as *datoms*, which are immutable facts consisting of an entity, attribute, value, transaction id, and a boolean, indicating whether the *datom* is being added or retracted. These *datoms* enable efficient point-in-time queries.

Datomic maintains four different indexes automatically, allowing the efficient use of different common access patterns, querying for entities ("row" access), attributes ("column" access) and values ("graph" access) with ease. These indexes support highly efficient data retrieval for a variety of query use cases, ensuring performance in both analytical and transactional workloads [84].

In addition to indexing, Datomic also uses a transaction log that is sorted by time, ensuring that all changes are chronologically stored. This persistent log never removes data, supporting long-term data retention and enabling point-in-time queries over the transaction time.

Datomic's query language, Datalog, is a declarative, logical query language that is particularly well-suited for graph and relational queries [18]. Datomic provides Clojure clients to access the database programmatically as well as different APIs.

Datomic has three versions available: Local, Pro and Cloud. Local is a database that runs in a single process, using the local file system for storage, while the Pro and Cloud solutions are suited for distributed contexts (with the Cloud solution running on Amazon Web Services only) [83].

Although Datomic is not open-source, all available versions are free. Datomic's combination of immutability, temporal querying, and distributed scalability makes it ideal for managing time-sensitive and historical data in modern, large-scale applications.

3.3.2.3 XTDB v1

XTDB v1 [6] [85] is an open-source immutable document database that supports graph queries, by indexing the documents as triples. Triples can have pairs of valid time and transaction time associated, making XTDB v1 bitemporal, although only supporting point-in-time queries.

XTDB is schemaless and does not enforce a schema for the documents it stores. To enable graph capabilities, XTDB indexes the documents as Entity-Attribute-Values triples. This approach effectively transforms the stored documents into a graph structure, where entities are nodes and attributes serve as edges connecting these nodes to their corresponding values.

XTDB's central component is an immutable transaction log, which functions as a ledger for all data operations. This log ensures that all past transactions are preserved and accessible, supporting historical analysis and auditability. To address data privacy requirements, such as the General Data Protection Regulation (GDPR), XTDB incorporates mechanisms for evicting both active and historical data, enabling compliance without compromising the immutability principles of the database.

Moreover, documents are stored on a document stored for durability and the compute operations on the XTDB nodes.

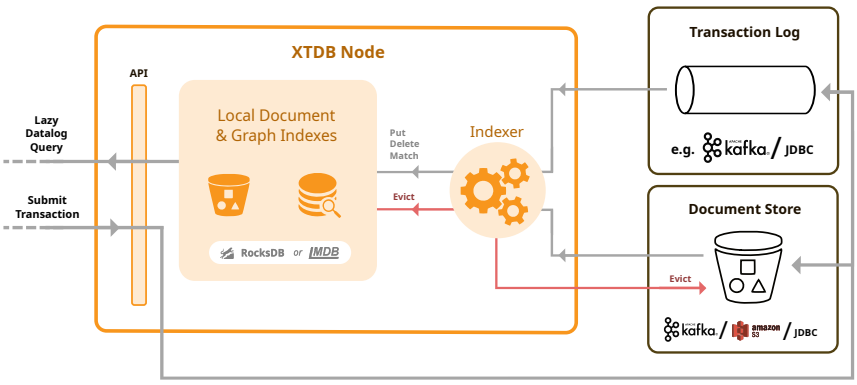

Figure 3.5: The components of a XTDB v1 database. From: [6]

The different components of a XTDB database (transaction log, document store and XTDB node), represented in Figure 3.5, can be deployed with cloud technologies or locally, assuring the database can scale to the needs of different sized projects.

XTDB v1 is a stable and production-ready product, while it is still undergoing development to improve on current service and performance levels. It focuses on advanced features such as user-defined transaction functions, speculative transactions, programmable Datalog rules, and more [7].

3.3.2.4 XTDB v2

Although still relying on the same principles, XTDB versions 1 and 2 both sufficiently different and both relevant to have the need to have them in a separate section.

XTDB v2 [7][17] relies on the same principles of XTDB v1, such as immutability, use of a dynamic schema and temporal capabilities, aiming to make them suit a mainstream audience. It is currently under active development, as the first stable release of XTDB v2 was launched during the development of this dissertation (12th of June 2025) [86].

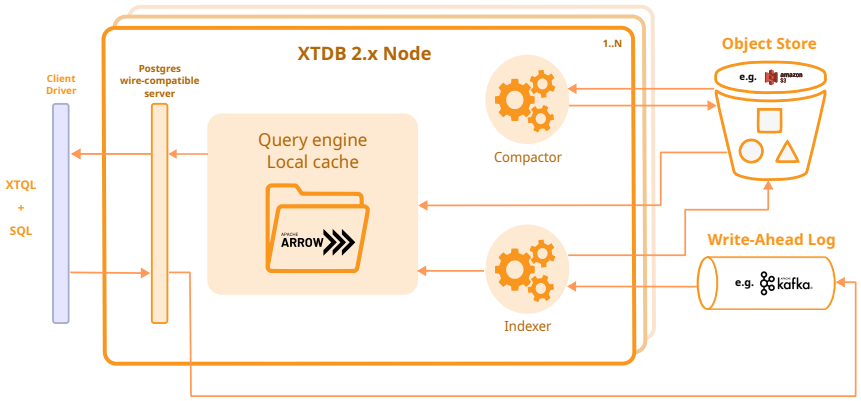

Figure 3.6: The components of a XTDB v2 database. From: [7]

On the architecture level, both the databases still rely on the same principles, having the responsibilities of the database shared between multiple components, as seen on Figure 3.6. The main difference lies on the use of columnar storage in the XTDB node, with the columnar Apache Arrow taking the place of the previous key-value store [7]. This makes XTDB v2 suited for both transactional and analytical workloads, while version 1 mostly targeted towards transactional queries.

XTDB v1 has previously been optimized for point-in-time historical queries. XTDB v2 extends temporal support with temporal joins and range scans. This unlocks the complete history of data for rich analysis [87].

The main feature that allows XTDB v2 to be suited for mainstream use is the first-class of two query languages, SQL and XTQL [88]. In addition, by exposing a PostgreSQL wire-compatible server, XTDB v2 is compatible with standard PostgreSQL tools and drivers, which allows the

support to connect to a XTDB database from multiple languages including Python, Java or Clojure [89].

3.3.2.5 SirixDB

SirixDB [90][91] is an append-only temporal database and event store designed for efficient storage and management of XML and JSON documents. Preserving the complete history of document versions, it enables linear-time reconstruction of any revision and supports advanced time-travel queries and efficient computation of differences.

This database implements a novel sliding snapshot algorithm, originally introduced in Kramis' dissertation [60]. The algorithm facilitates snapshot-based versioning by employing sliding windows, striking a balance between read and write performance. This approach improves efficiency when compared to traditional incremental or differential versioning methods [92][93].

SirixDB is open-source and actively in development. It is particularly suited for applications requiring historical data analysis or auditing.

3.3.2.6 immudb

Immudb [15][94] is an open-source database designed with built-in cryptographic proof and verification.

Its architecture prioritises data security and integrity, making it suitable for applications requiring robust audit trails and tamper-proof records. This database supports both relational, document, and key-value models, offering flexibility in managing diverse data types [95].

At its core, immudb employs mutable Merkle hash trees, enabling cryptographic verification of data without requiring trust in the database server. This structure ensures that any change or tampering with the stored data can be cryptographically detected, offering end-to-end assurance of data integrity. Moreover, this design supports compliance with privacy regulations like GDPR, allowing for selective data redaction without compromising the integrity of the database [96][97].

Immutability is a foundational principle of immudb. While new transactions can be appended, past transactions cannot be deleted or modified, preserving a complete and verifiable history. This feature is further complemented by time-travel capabilities, enabling users to retrieve historical states of records and track changes over time.

3.3.3 Relational Databases

This section examines temporal relational databases, which build upon the structured and rigorously defined nature of relational database systems. Temporal relational databases differ from their document or graph counterparts in their reliance on a fixed schema, ensuring strong consistency and adherence to relational algebra principles.

3.3.3.1 PostgreSQL

PostgreSQL is a powerful, open-source object-relational database system with over 35 years of active development, renowned for its reliability, robustness, and performance [98]. It is not a temporal database, but it offers some temporal features and capabilities and will be covered due to its relevant use in the open-source community.

PostgreSQL introduced time-travelling queries with version 9.2, in 2014, with the *Range* Type enabling querying for data whose timestamps were in a certain "range" of time [14]. Based on relational data model, PostgreSQL supports both the standard SQL types, but also arrays and unstructured data using the JSON data types [99].

For more advanced temporal features, PostgreSQL relies on extensions [100]. These extensions enable PostgreSQL to provide functionality similar to purpose-built temporal databases, including support for unitemporal or bitemporal models and querying historical states.

PostgreSQL supports multiple indexing options to enhance its ability to perform complex queries efficiently. These include B-tree, the default and most common index type, well-suited for equality and range queries, as well as sorting. Other specialised indexes include GIN (Generalized Inverted Index), which excels in indexing composite values such as arrays and JSONB for efficient element-level searches and BRIN (Block Range Indexes), a lightweight, space-efficient option for very large tables where data is naturally ordered, often used for time-series data. Each is optimised for different types of queries [101].

While PostgreSQL offers robust support for temporality through features and extensions, then again, it is not designed as a temporal database. Instead, its temporal capabilities are integrated within its general-purpose framework, making it easier for users already familiar with PostgreSQL to adopt temporal functionalities without significant overhead. This approach simplifies maintenance, integration, and migration, particularly for applications requiring temporal features as part of a broader relational data model.

3.3.3.2 Dolt

Dolt positions itself as "Git for Data" [102]. It is an open-source SQL database that enables the use of Git-like commands such as fork, clone, branch, merge, push, and pull. Dolt versioning allows users to track and query the historical states of the database seamlessly [103].

Dolt's architecture is inspired by Git's principles but tailored to the demands of database systems. Its core data structure, Prolly Tree (probabilistic B-trees), extends traditional B-trees by enabling efficient indexing and facilitating version control operations, such as merges and diffs.

Prolly Trees design meets Dolt requirements: B-tree-like performance for both reads and writes, fast diffs for efficiently comparing two versions of data, and structural sharing, which ensures that any shared portion of data between versions is stored only once. These properties enable Dolt to manage changes and resolve conflicts seamlessly while maintaining scalability [104].

Dolt has found applications across diverse domains, including the versioning of game data, medical records, machine learning training datasets, and data quality control [105].

3.3.4 Database Overview

In this section, an overview of the databases covered in the past sections will be made as well as a study on which databases might be the most suited for introducing temporal versioning to user profiles.

Table 3.2: Comparison of Temporal Databases

Name	Temporal By Design	Temporal Model	Query Language	Immutable	Supported Databases
TerminusDB (3.3.2.1)	Yes	Unitemporal	WOQL and GraphQL	Yes	Document/Graph
Datomic (3.3.2.2)	Yes	Unitemporal	Datalog	Yes	Key/Value, Graph
XTDB v1 (3.3.2.3)	Yes	Bitemporal	Datalog	Yes	Document/Graph
XTDB v2 (3.3.2.4)	Yes	Bitemporal	SQL, XTQL	Yes	Document
SirixDB (3.3.2.5)	Yes	Bitemporal	XQuery	No	Document
immuDB (3.3.2.6)	Yes	Unitemporal	SQL	Yes	Key/Value, Relational, Document
PostgreSQL (3.3.3.1)	No	—	SQL	Needs extensions	Relational
Dolt (3.3.3.2)	Yes	Unitemporal	SQL	Yes	Relational

Temporal databases share common features such as support for temporal models, immutability guarantees, multiple query languages, and various database types. However, each database has unique strengths that are suited to different use cases and requirements. In Table 3.2, a succinct description of the main characteristics of each database is shown.

3.3.5 Evaluation and Performance metrics

To assess the performance, functionality, and applicability of a temporal database implementation, evaluation metrics are used to compare the performance of the systems with each other and with the non-temporal database counterparts, which are typically the starting point.

When selecting a database for a specific use case, various factors must be considered, including the supported query languages, indexing methods, availability, consistency, durability, security features, transaction support, and licensing models [106]. For temporal DBMS, the temporal model and the immutability should also be taken into consideration.

Evaluation metrics for temporal databases are similar to the ones used in the other non-temporal databases. Ilsso Ahn and Richard Snodgrass pioneered the benchmarking of temporal databases, measuring the response times for executing temporal queries across different database types [107]. Query execution time is still a commonly used and relevant evaluation metric used to compare database performance [108][109][110].

For years now, the amount of data collected and created has been growing exponentially [111], and the need for more storage and its cost have also increased [37]. Modern temporal DBMSs, such as SirixDB and TerminusDB, address these challenges by employing innovative data structures that minimize storage overhead while maintaining temporal integrity. For instance, SirixDB uses a sliding snapshot algorithm to store differences incrementally [91]; whereas TerminusDB adopts succinct data structures to optimize versioned graph storage [81].

We therefore considered as critical metrics for evaluating the performance of temporal databases the retrieval times and storage efficiency. These metrics are vital for database performance in general but are particularly significant in the temporal domain due to the storage-retrieval trade-off

[38]. This trade-off often involves compressing or encoding historical data to save storage space at the cost of increased computational effort during data retrieval.

Storage efficiency refers to the database's ability to minimize the space required to store temporal data while maintaining its integrity and accessibility. As more historical versions and changes are recorded, temporal data inevitably grows over time. Efficient storage mechanisms may involve techniques like delta encoding, compression algorithms, or specialized temporal indexing. Storage efficiency is typically measured by the total disk space used relative to the volume of data managed, often expressed as a percentage or ratio [112]. For example, systems that reduce storage requirements by encoding only the changes between data states (rather than duplicating entire records) can achieve significant space savings. However, achieving high storage efficiency may introduce additional computational overhead during data retrieval or updates [38].

Retrieval times, on the other hand, measure how quickly a temporal database can fetch and deliver the requested data [107]. Query optimization capabilities, the database's indexing strategy, and the format in which temporal data is stored are some of the variables that affect retrieval performance. Snapshot-based systems, for instance, can frequently retrieve data more quickly for specific queries because they store complete versions of the data at particular times, but they come with a higher storage cost. Conversely, systems using changeset-based storage mechanisms may require more computational effort to reconstruct the requested state from deltas, potentially increasing retrieval times [38]. In applications where real-time or near real-time requirements are present, retrieval times, which are commonly expressed in milliseconds or seconds, are an essential metric for guaranteeing responsive systems.

3.4 How can Temporal Databases represent User Profiles?

Having gone through the main characteristics of representations of user profiles and temporal databases, this section will discuss what databases may suit the use case of the study.

User profiles can be represented in different conceptual models, and each conceptual model can be stored in various formats. However, conceptually, profiles tend to be represented more naturally as a graph of entities, as seen in Section 3.2, and so the support for graph query is a valuable feature for ease of querying for relations.

Moreover, the nature of user profile information in systems like Customer Data Platforms, where information may come from several sources, makes it so data is likely to be unstructured. [21]

Bearing the findings of previous sections in mind, XTDB (Section 3.3.2.3 and Section 3.3.2.4) and TerminusDB (Section 3.3.2.1) exhibit several features that make them strong candidates for addressing the challenge of temporal versioning in user profiles.

Both of these systems should be highly adaptable to the unstructured nature of user profiles. Using a document-oriented storage model allows them to seamlessly handle diverse schemas and attributes, accommodating the variability inherent to data from multiple sources. The support for graph queries further enhances their applicability to user profiles by capturing intricate connections

between entities, such as interactions, preferences, and relationships. Additionally, the focus of TerminusDB on changeset-based versioning, which suits user profiles that undergo frequent and incremental updates, is expected to be a key feature in reducing storage use.

Both databases provide powerful tools for temporal versioning, offering unique advantages depending on the specific characteristics and requirements of the system managing user profiles.

3.5 Summary

This chapter explores the state of the art in user profile representations and temporal databases. Through a systematic literature review, insights into diverse user profile models were gathered, highlighting their strengths and weaknesses concerning various applications. The exploration of temporal databases revealed their fundamental attributes, most important features, and methods to employ temporal versioning.

From the studied database, TerminusDB and XTDB emerge as strong candidates for managing temporal versions of user profiles. Both databases are highly adapted to the unstructured and evolving nature of user profiles by storing documents and can be used to retrieve the complex relationships user profiles have by providing graph querying capabilities. Additionally, TerminusDB offers changeset-based versioning, which is well-suited for frequent updates.

The next chapter, Chapter 4, points out the open problems identified during research, the hypothesis that will guide the experiments, and the methodology and validation techniques used.

Chapter 4

Problem Statement

Contents

This chapter addresses the problem underlying this investigation. Section 4.1 introduces the open problems related to temporal database management systems. In Section 4.2, the scope of the study is defined, while Section 4.3 presents the hypothesis to be tested. The proposed solution is described in Section 4.4, and the chapter concludes with an overview of the methodology and validation used in Section 4.5. In Section 4.6 this chapter is summarised.

4.1 Open Problems

In Chapter 3, we analysed various models to represent user profiles and revised temporal database management systems. User profile models have been extensively studied and compared in recent years, with significant insights into their structure and applications [69].

However, the literature is lacking comparison studies and benchmarks on the temporal database management systems highlighted.

Moreover, although the literature suggests that temporal data might be useful in user profile management scenarios, temporality in this context, specifically in the storage of this data, has not been covered in the research.

4.2 Scope

In light of the shortcomings highlighted in Section 4.1, this work aims to address them by comparing the performance of different temporal database management systems. The storage of user profiles serves as the guiding scenario for the experiments, given its relevance as a practical use case for temporal databases.

The storage unit of a Customer Data Platform, described in Section 2.5, provides the primary inspiration for this experiment. While the study is centred on this specific use case, the findings are expected to generalise to other user profile management scenarios.

4.3 Hypothesis

Building on the open problems discussed in Section 4.1 and the objectives outlined in Section 1.4 the following hypothesis has been formulated:

> *There exists a measurable and consistent trade-off between minimizing storage space and optimizing retrieval performance across different temporal data models.*

This hypothesis will be tested by evaluating selected DBMS across key performance metrics such as storage efficiency, query retrieval times, and scalability, while storing and retrieving user profiles, represented using different temporal models.

4.4 Proposed Solution

To validate the hypothesis, this work proposes integrating temporality into a database of user profiles and evaluating its performance using various DBMS. The evaluation will compare performance metrics and highlight the advantages and trade-offs of each system, providing insights into their suitability for the application for profile management.

Beyond performance analysis, this study aims to assess the feasibility of implementing temporal database systems in an industry setting. The research will use Kevel's Customer Data Platform, Kevel Audience [22], particularly its User Profile Storage component, as a industry example to guide the development and evaluation of the proposed solution.

This proposed solution aims to both test the hypothesis and provide actionable recommendations for selecting and designing temporal database systems for user profile management and similar applications.

4.5 Methodology and Validation

On their work "Experimental Models for Validating Technology", Zelkowitz and Wallace [113] defined four different general models to conduct experiments (quoted from their publication):

Scientific method: Scientists develop a theory to explain a phenomenon; they propose a hypothesis and then test alternative variations of the hypothesis. As they do so, they collect data to verify or refute the claims of the hypothesis.

Engineering method: Engineers develop and test a solution to a hypothesis. Based upon the results of the test, they improve the solution until it requires no further improvement.

Empirical method: A statistical method is proposed as a means to validate a given hypothesis. Unlike the scientific method, there may not be a formal model or theory describing the hypothesis. Data is collected to verify the hypothesis.

Analytical method: A formal theory is developed, and results derived from that theory can be compared with empirical observations.

In this dissertation, an **Engineering Method** is employed, by comparing how different database candidates perform against each other, and then the results are put against the hypothesis.

The technique described as **Dynamic Analysis** [113] is used to validate this work, by implementing a benchmark between the identified database candidates. Benchmark suites are often utilised to collect representative execution behaviour across a broad set of similar products, as is the case for our experiment. The workload for this benchmark is based on anonymized data from Kevel Audience.

All the experiments should be easily replicated. A description of the experiment and its replication package can be found on Chapter 6, together with the discussion of the benchmark results.

4.6 Summary

This chapter outlined the existing problems in this work domain and the proposed plan to build a solution. It began with an exploration of the open problems in temporal database management systems, highlighting a notable gap in their comparative analysis, especially in practical use cases like user profile management. The scope of the study was then defined, narrowing the focus to the evaluation of temporal DBMS performance in the context of managing user profiles, specifically within the framework of a Customer Data Platform.

The hypothesis suggested that for different temporal data models a measurable and consistent trade-off should be encountered between minimizing storage space and optimizing retrieval performance.

Finally, a description of the proposed solution and its goals, as well as describing the experimental model for this dissertation, using the nomenclature employed by Zelkowitz and Wallace [113].

Chapter 5

Customer Data Platform Storage

Contents

This chapter presents the workload that guides and later evaluates the Proposed Solution (Section 4.4). Section 5.1 describes the interface exposed by the storage component, focusing on its endpoints. Section 5.2 discusses the role of time in Customer Data Platforms and its implications for data modelling. Section 5.3 characterises the types of attributes typically stored in user profiles and their associated merge strategies and contrasts snapshot-based and update-based storage approaches. Section 5.4 analyses the real-world workload used in this study. Finally, Section 5.5 summarises the main points of the chapter.

5.1 Usage

Recalling the description of a Customer Data Platform in Section 2.5, we focus here specifically on the User Profile Storage component. The Event Tracking component adds information to the User Profile Storage and the information stored in the User Profile Storage is used by the Activation and Segmentation components [2].

There are two main endpoints through which one can interact with the storage component, one to update a user and other to retrieve.

`PUT` A `PUT` request is made on every event associated with a user. The payload should include the ID of the user to be updated plus a dictionary of the attributes and values to be updated.

`GET` A `GET` request is made to get the information on a certain user. The most common use case is to get only one user at a time. Given an ID, should return a dictionary with the attributes and values of the user.

These are the operations that will be supported by the system to be evaluated. In the next section, we will discuss how time can be incorporated into these requests.

5.2 Time in Customer Data Platforms

Customer Data Platforms store data from multiple sources. It is often the case that the facts are stored retroactively [44], meaning after they are valid in the real world, but the delay between real-world occurrence and data arrival varies by source. Imagine the case of a retailer, with both in-person store and online presence. In Figure 5.1, an example a possible order of facts from the different sources may be stored is shown.

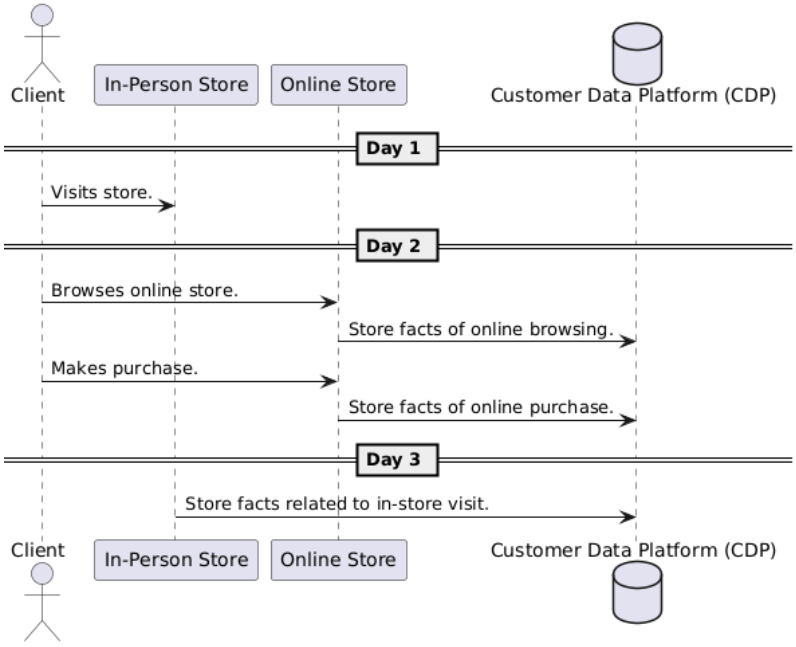

Figure 5.1: Customer Data Platform - Multi-Source Data Collection

From the perspective of the user profile management, the valid time of the events is the timeline to be used to understand the evolution of the user profile at the different times. So, using the example from Figure 5.1, if we ask on Day 4 what the Client has done, we should be able to know that the client first visited the store in-person on Day 1 and then browsed and made a purchase using the online store on Day 2.

To add this context of temporality to Kevel's use case, an optional timestamp is to be added to both the `PUT` and `GET` requests when communicating with the storage layer. For the `PUT` request, the timestamp represents the valid time of the event (if omitted, it is assumed to equal the

transaction time). For the `GET` request, the valid time of when the profile is to be read (the default is NOW).

In this context, only the valid time is required to be stored, effectively needing unitemporality for valid time only.

Although not a priority for Kevel's use case, bitemporality has useful use cases in the context of CDPs. It removes the need to store snapshots of data to be able to perform rollbacks or inspect past states. One can also query for all the updates that were made retroactively by more than a given amount of time (a week) to uncover which sources of information have been lacking. The ability to run what-if queries can be also useful: "what would be the segment of a customer had we made same activation we did last week with the data we now possess?".

In the next section, the user profile format and the different strategies to store and retrieve user profiles will be covered.

5.3 Attributes

User profiles are represented as simple collection of attributes. Each attribute is associated with a merge strategy and maintains a current value. The possible merge strategies for the Kevel Audience use case can be seen on Table 5.1.

Merge strategies define how attributes should be updated in a user profile given a new `PUT` request. New values don't usually replace old values (unless the strategy so specifies). Merge strategies enable incremental updates to profiles without requiring the authors of `PUT` requests to inspect the existing state of profiles prior to actually performing the requests.

To better understand the merge rules, we use the following variables: $current_t$ is the timestamp given in the request, $last_t$ the timestamp associated with the last time the attribute was changed, $value$ the value in the request and $attr$ the current value associated with the given attribute.

Table 5.1: Attribute Types and merge strategies

Merge Strategy	**Type**	**Merge Rule**
Most-recent	Any	$if(current_t > last_t) : attr = value$
Older	Any	$if(current_t < last_t) : attr = value$
Sum	Numeric	$attr = attr + value$
Max	Numeric	$if(value > attr) : attr = value$
Or	Boolean	$if(value) : attr = true$

The merge strategies that one attribute uses is given with the attribute creation and cannot be changed. All the attributes and their types are given at startup in the form of a JSON file, where keys are the attribute names and the values the corresponding merge strategies, as seen on Listing 5.1.

```
1 {
2     "16c3027744ff5a086a69c91ab8022086": "most-recent",
```

```
    "7a9ec0f97cd6f47b044e72673d493a68": "most-recent",
    "00caf394edb777c2d16cf3f41537e61b": "sum",
    "80461e990df28df8f262252d08c93e86": "sum",
    "6dabd267409ffbb9cb7d0163fc3df51a": "most-recent",
    "de2aee66f48fcea81c1dd69a7a40bb42": "or",
    ...
}
```

Listing 5.1: Snippet of file that defines attributes and merge strategies

A representation of a customer profile is depicted on Figure 5.2. Kevel's use case focuses on storing attributes in a key/value relation. Additionally, there can also exist links between different users, to unify customer data from different sources, as the same customer can be identified by a different id in different platforms [2]. This action is common in Customer Data Platforms, as covered on Subsection 2.5.1.

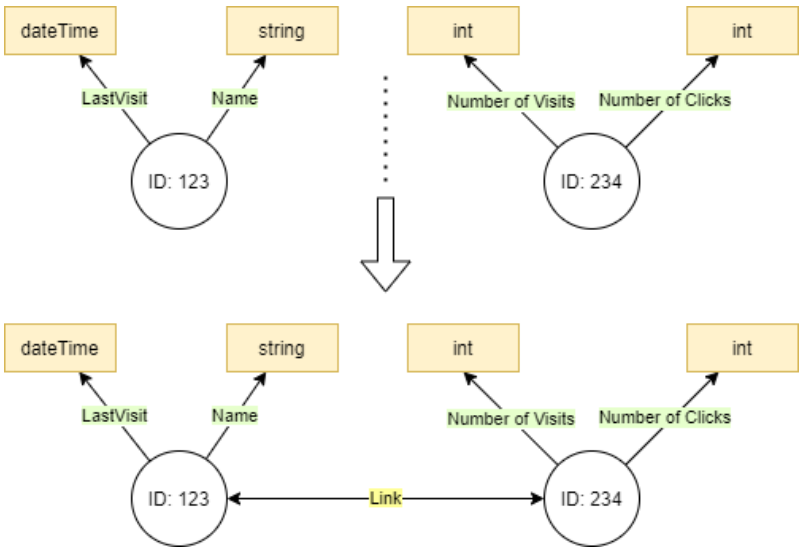

Figure 5.2: Customer profile example representation, key/value and in between entity relations - Linking profiles trough ID matching

The linking of profiles ensures customers are represented in a more complete, holistic view. After the linkage, whenever a request is made to retrieve the attributes from either of the IDs, the information from all linked profiles will be merged and shown. It was opted to keep the linking of user IDs out of the scope for the study described on Chapter 6 due to time constraints.

5.3.1 Snapshot- and Operation-based storage

Two of the approaches to store information, identified in 2.4 are the snapshot- and the operation-based. In the context of user profiles, they can be stored as (1) a collection of snapshots or full states or (2) a collection of operations or modifications. By storing all the states, one is potentially

duplicating information and increasing storage costs, but reducing retrieval time. One can also store storing all modifications, expectedly achieving a lower storage space, but at a cost of a slower retrieval time, as the state is reconstructed on retrieval [38][60].

The minimal difference between these two approaches happens for users that are updated a single time, as the storage occupied and retrieval time between modes is expected to be the most similar. The biggest expected difference should be in large user profiles with frequent small updates (snapshot-mode will have full states ready to be retrieved in lower time, at the cost of more storage due to the duplication of information between states).

There are only two operations that can be performed in the described system, `PUT` and `GET` operations, and only the `PUT` modifies the data. The operations stored in the operation-based approach will be all the updates to the user profile (all the `PUT` operations).

There are two class models we will test, as seen on Figure 5.3. To recreate a user profile at any given time, we can either store all the states of a user profiles over the valid time, or store all the updates associated with a user. The implementation on each database will differ, due to each database constraints, but will follow both models as closely as possible. All the information can be represented with only one table, and so there are no further concerns on temporal referencing and foreign key violations [114].

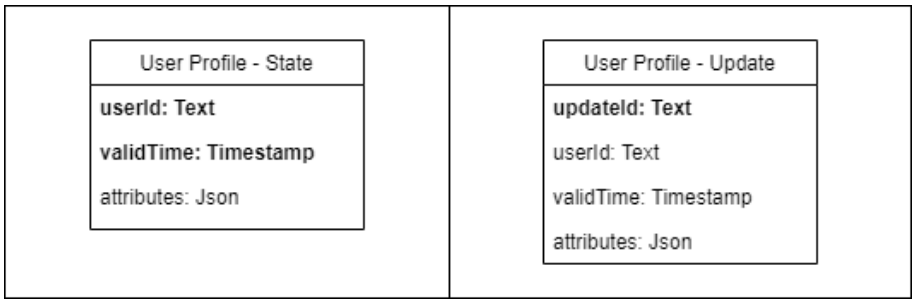

Figure 5.3: Customer Data Platform - Class model

On Figure 5.4, the green arrow represents the last operation to reach the database. We will use this example to highlight the differences between storing a user profile as multiples snapshots or multiple operations.

Figure 5.4: Operations over valid time axis

In Table 5.2, the differences in operations of the two approaches are highlighted. Snapshots and modifications are stored complete, with no further application side algorithms for reducing space, calculate differentials or perform compressions.

All reconstructions are performed on the application side, as the compression, space-efficiency and retrieval time of the databases is what is subject to test. Hybrid approaches, such differential-based versioning [63] were left out for with that reasoning in mind.

Table 5.2: Comparison of `PUT` and `GET` operations for snapshot- and operation-based storage strategies

Snapshot-based	
PUT	**GET**
1. Select last snapshot known at the given valid time (1) 2. Construct and insert the new snapshot (5) 3. Update all the snapshots after the valid time (2, 3, 4)	1. Read the latest snapshot (4)
Operation-based	
PUT	**GET**
1. Insert the operation in place (2)	1. Read all the past operations (1, 2, 3, 4, 5) 2. Starting with an empty profile, perform all operations and return the user profile

5.3.2 Merge strategies

Either when doing a `PUT` for the snapshot-based approach or a `GET` for the operation-based approach, constructing the user profile at a given time is similar: get the latest snapshot known (either by reading the last snapshot as of the valid time requested or performing all operations up to that point) and then merge with the just received operation. In the Listing 5.2, the algorithm for merging an update with a past snapshot is shown, following the rules highlighted in Table 5.1.

The attributes are represented by an object with attributes and values, either if they are from an update or from a snapshot. To determine the merge strategy for a given attribute-value pair, the merge strategy is first selected from a dictionary `rules`, built from the file that defines the merge strategies seen on 5.1.

```
1 def merge_with_past(past_attributes, update, rules):
2     for (attr, value) in update.items():
3 
4         #Get the update rule
5         rule = rules[attr]
6 
7         #More recent than past_attributes
8         if rule == "most-recent":
```

```
            past_attributes[attr] = value

        #New value if the attribute is empty
        elif rule == "older":
            past_attributes[attr] = past_attributes.get(attr, value)

        #Get value and sum
        elif rule == "sum":
            past_attributes[attr] = past_attributes.get(attr, 0) +
                value

        #Update if value > current
        elif rule == "max":
            past_attributes[attr] = max(past_attributes.get(attr,
                float('-inf')), value)

        #Logical OR
        elif rule == "or":
            past_attributes[attr] = value or past_attributes.get(
                attr, False)

    return past_attributes
```

Listing 5.2: Merging incoming update with a past snapshot

We have also seen that, when updates arrive unordered, the update may affect future snapshots (snapshots after the current update valid time).

To change a more recent snapshot given an update to the past, only the "most-recent" and "older" rules have to be changed, all other rules are commutative and associative. The changes are highlights in Listing 5.3.

```
def merge_with_future(future_attributes, current_attributes, rules):
    for (attr, value) in current_attributes.items():
        rule = rules[attr]

        #If exists, is more recent, else update
        if rule == "most-recent":
            future_attributes[attr] = future_attributes.get(attr,
                value)

```

```
9           #Older than
10          elif rule == "older":
11              future_attributes[attr] = value
12
13          #The same as merge_with_past
14          ...
15
16      return future_attributes
```

Listing 5.3: Merging current snapshot with a future snapshot

5.4 Workload

In this section the workload used for this study is analysed in detail to better characterise its structure and relevance.

The workload is based on anonymised data from Kevel Audience, based on `PUT` requests received trough time. Over 14 million JSON objects were given as the example for the user profile updates, for a total of 713 thousand users and 818 unique attributes. The distribution of updates over time can be seen on Figure 5.5 showing updates dated from the 2nd of October 2024 to the 14th of February 2025 (136 days).

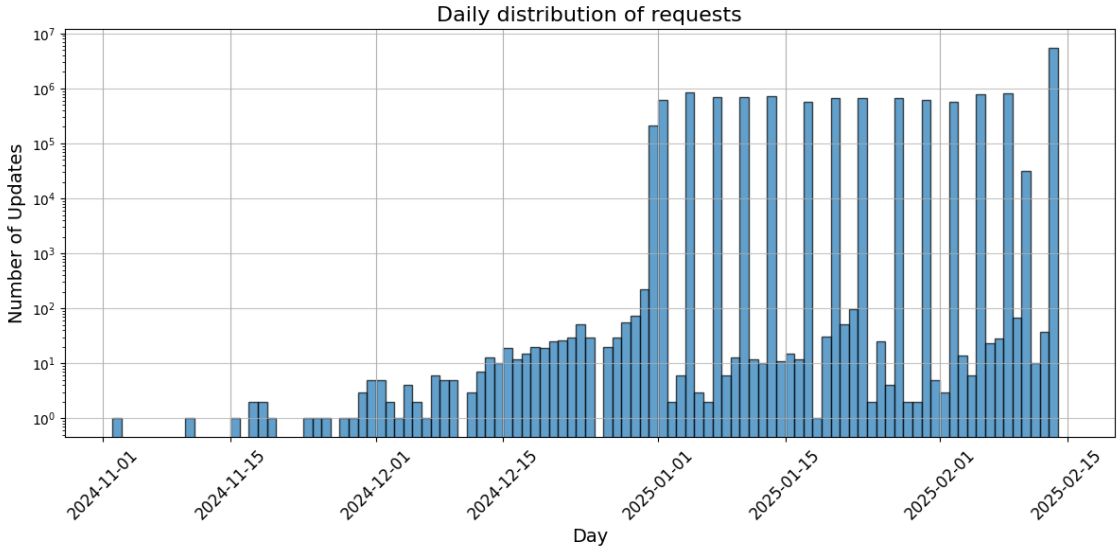

Figure 5.5: Distribution of updates over time of Kevel Audience's dataset

The overall average of rate of receiving updates over this time period equates to an average of 1.64 requests per second, or around 140 thousand daily requests. A majority of days has less than 100 received requests, highlighting the high variation in daily received updates.

From the 31st of December 2024 forward, periodically some days have an increase in the rate of PUTs received, handling a magnitude of 10 PUT requests per second. A maximum peak is encountered on the last day of recorded, with approximately a third of all requests being registered on this day.

Kevel Audience reports around 70% of PUT requests and consequently only 30% of GET. The test suite for the benchmark will follow this same distribution of requests. During testing, throughputs of 1, 10 and 100 total requests per second will simulate the different orders of magnitude of throughput encountered in the dataset, relating to normal functioning, periodic high throughput and maximum peaks.

Figure 5.6 illustrates the diversity of size of the updates that the system receives, and how the updates are distributed per user. The scale of the graphs is logarithmic to better illustrate the distribution of data across different orders of magnitude. The median value of updates per user is 8, with nearly 30% or users being updated only once, although there are several users with hundreds or thousands of updates (left of Figure 5.6). A majority of updates only changes one or two attribute at a time, although updates can be bigger (right of Figure 5.6). The number of times an attribute is updated differs significantly, with attributes appearing from one to a million updates (centre of Figure 5.6).

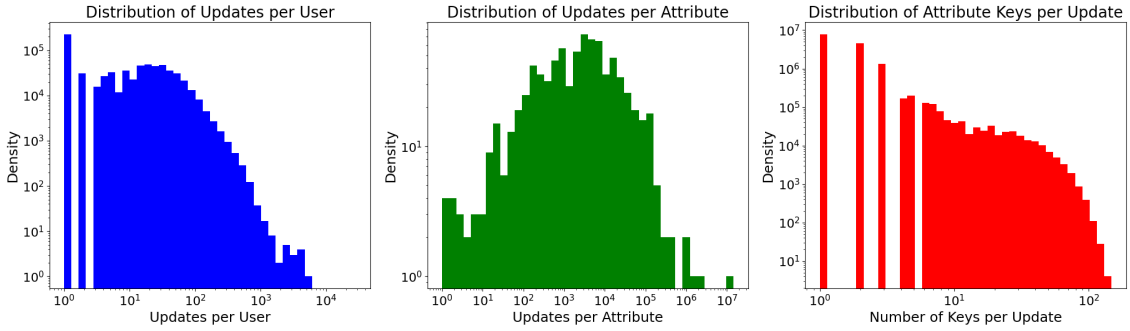

Figure 5.6: Workload analysis distribution of the: updates per user (left), updates per attribute (centre) and attributes per update (right)

5.5 Summary

This chapter better introduced the domain and workload used to evaluate the proposed storage solution. It began by detailing the interface of the storage component, specifically its read and write endpoints. It then explored the usage temporal information in Customer Data Platforms (CDPs), highlighting how time influences the interpretation and management of user data. The chapter also classified profile attributes based on their semantics and associated merge strategies. Two alternative storage strategies — snapshot-based and operation-based — were discussed in terms of their structure and implications. Finally, a real-world workload comprising over 14 million profile updates across 713 thousand users was analysed to ground the evaluation in realistic usage patterns.

Chapter 6

Database Evaluation

Contents

In this chapter, the benchmark of databases will be covered. Starting with the selection and description of the databases, in Section 6.1. The methodology of the experiment is described in Section 6.2 and the results and discussion in Section 6.3. Section 6.4 and Section 6.5 present the threats to the validity and the replication package of the experiment, respectively. The main findings of this chapter are highlighted in Section 6.6.

6.1 Databases Selected

In this section, the justification for the databases selected for test, and a description of how they are build and run is presented. The differences, highlights and limitations of each database will also be covered.

The databases that were selected for this experiment were the ones identified as suited for User Profile Management in Section 3.4, XTDB v2 and TerminusDB, as well as PostgreSQL, to serve as a baseline for comparison, as it is an industry standard for multiple applications.

The databases selected had to have support for time-travelling using valid time. Databases that only supported time-travelling trough transaction time and did not offer a way to do it trough valid time were discarded.

6.1.1 PostgreSQL

As one of the most widely used open-source databases, PostgreSQL (also discussed in Section 3.3.3.1) was selected to benchmark how newer temporal database solutions perform against this industry-standard system.

In accordance with the experimental requirements, only valid time was stored. Since bitemporality was not required, the previously discussed temporal extensions were not used [100]. The valid time reflects the moment when a user's state became active. Each entry thus represents a change in the user profile's valid state, with the at field marking the time at which this new state took effect. The schema and queries used for both `PUT` and `GET` operations are showed in Listing 6.1. The `jsonb` type was used to store the attributes as a JSON object [99]. When selecting a state at a given valid time, the state with the higher associated time lower than the given timestamp is retrieved. When inserting a state, if there already is a state at the same time, the attributes are updated.

```
1  -- schema
2  CREATE TABLE IF NOT EXISTS customer_state (
3      id TEXT NOT NULL,
4      attributes JSONB NOT NULL,
5      valid_time TIMESTAMP NOT NULL DEFAULT NOW(),
6      PRIMARY KEY (id, valid_time)
7  );
8
9  -- get the state at a given valid time (id = 1234, valid_time = 10th
       of January)
10 SELECT * FROM customer_state
11     WHERE id = 1234 AND valid_time <= 1736467200
12     ORDER BY valid_time DESC
```

```
13      LIMIT 1;
14
15  -- upsert the user profile state at a given time (id = 1234,
        attributes = JSONB Object, valid_time = 10th of January 2025)
16  INSERT INTO customer_state (id, attributes, valid_time)
17      VALUES (1234, JSONB OBJECT, 1736467200)
18      ON CONFLICT (id, valid_time)
19      DO UPDATE SET
20          attributes = EXCLUDED.attributes;
```

Listing 6.1: Schema and queries for PostgreSQL's snapshot-based representation

The operation-based schema and queries are highlighted in Listing 6.2. For this representation a generated primary key was used to ensure a different primary key for each update. This primary key just ensures the requirement of having a different primary key for each update. When retrieving a user in this mode every update associated with a given user inserted prior to a given time are collected and reconstructed in a single user profile application-side. Inserting updates never causes conflicts, as all are given a different primary key.

```
1   -- schema
2   CREATE TABLE IF NOT EXISTS customer_update (
3       id INTEGER PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
4       userId TEXT NOT NULL,
5       attributes JSONB NOT NULL,
6       at TIMESTAMP NOT NULL DEFAULT NOW()
7   );
8
9   -- get the state at a given valid time (id = 1234, valid_time = 10th
        of January 2025)
10  SELECT attributes, at FROM customer_update
11      WHERE userId = 1234 AND valid_time <= 1736467200
12      ORDER BY at;
13
14  -- insert an update at a given time (id = 1234, attributes = JSONB
       Object, valid_time = 10th of January 2025)
15  INSERT INTO customer_update (userId, attributes, valid_time)
16      VALUES (1234, JSONB OBJECT, 1736467200);
```

Listing 6.2: Schema and queries for PostgreSQL's operation-based representation

To support performant queries over large volumes of updates, two indexes were added to the the customer tables, and can be found in Listing 6.3. These will be used for the temporal operations of getting a user at a certain time (snapshot-based) or all the updates in a given time interval (operation-based). The other temporal databases already support temporal indexes by default, and so these should be in place to ensure PostgreSQL also performs to the highest level.

```
1 -- indexes for customer_state table
2 CREATE INDEX IF NOT EXISTS idx_state_id_valid_time_desc ON
     customer_state(id, valid_time DESC);
3 CREATE INDEX IF NOT EXISTS idx_state_valid_time_brin ON
     customer_state USING BRIN (valid_time);
4
5 -- indexes for customer_update table
6 CREATE INDEX IF NOT EXISTS idx_update_userid_valid_time ON
     customer_update(userId, valid_time);
7 CREATE INDEX IF NOT EXISTS idx_update_valid_time_brin ON
     customer_update USING BRIN (valid_time);
```

Listing 6.3: Indexes used for PostgreSQL

6.1.2 XTDB v2

Recovering the findings of Section 3.3.2.4, XTDB v2 is a schemaless bitemporal database, that consumes documents and allows both relation and graph like query capabilities.

Going deeper into the implementation and use, each table of the XTDB v2 database uses four built-in temporal columns to store time, two for valid time (_valid_from and _valid_to) and two for system time (_system_from and _system_to). When not explicitly defined, any document inserted will be valid between the time of insertion and the end of time.

Although it has a schemaless design, XTDB v2 has a single requirement for schema: documents must contain an _id column that uniquely identifies it. The primary key of each table of the database is comprised of the _id column together with the four built-in temporal columns. The _id has to be given by the user, as XTDB does not offer a way to generate IDs automatically, and should uniquely identify the document, being recommended to use UUIDs (Universally Unique Identifiers) [115] to identify documents [116].

All inserts work as upserts, as when insert a document on top of an already existing one, it will overwrite the existing document for the valid time range specified. With the immutability and bitemporality provided by XTDB v2, the old document will still be recorded in the database, but with a closed interval for system time, and is no longer retrieved by default.

The default case when selecting a document, if no time is specified, is to get its latest state (as of "NOW"), filtering all non-valid or historical data [117]. Selecting from any given valid or transaction time is done easily by adding a single line of SQL code, as can be seen in Listing 6.4.

```
1  -- get the current state of a user (id = 1234)
2  SELECT c.*, _valid_from, _valid_to
3      FROM customer_state AS c
4      WHERE _id = 1234;
5
6  -- get the state at a given valid time (id = 1234, valid_time = 10th
       of January 2025)
7  SELECT c.*, _valid_from, _valid_to
8      FROM customer_state
9      FOR VALID_TIME AS OF 1736467200 AS c
10     WHERE _id = 1234;
11
12 -- upsert the user profile state at a given time (id = 1234,
       attributes = JSONB Object, valid_from = 10th of January 2025)
13 INSERT INTO customer_state RECORDS {_id: 1234, attributes: JSONB
       Object, _valid_from: 1736467200};
14
15 -- upsert the user profile state at a given time interval (id =
       1234, attributes = JSONB Object, valid_from = 10th of January
       2025, valid_to = 15th of January 2025)
16 INSERT INTO customer_state RECORDS {_id: 1234, attributes: JSONB
       Object, _valid_from: 1736467200, _valid_to: 1736899200};
```

Listing 6.4: Queries used to travel trough valid time and perform insertions in XTDB v2 with snapshot-based representaiton

Using the snapshot-based model, multiple rows of data are being recorded for a user. Each update to the user state is represented as a new row that sits alongside the previous versions of that row in the same table [116]. Immutability guarantees that all the states that the client ever had are registered in the database.

For the operation-based model, since all updates have a different id and never stop being valid, there is one row per update made on the table. To get the updates that are valid at a certain point in time, similar logic applies to the one seen on the snapshot queries – get all updates when as of now and or get all updates inserted prior to a given valid time. All the queries used for the operation-based representation are depicted in Listing 6.5.

```
1  -- get all the updates as of now (userId = 1234)
2  SELECT attributes, _valid_from FROM customer_update
3      WHERE userId = 1234
4      ORDER BY _valid_from;
5
6  -- get all the updates at a given valid time (userId = 1234,
       valid_from = 10th of January 2025)
7  SELECT attributes, _valid_from FROM customer_update
8      FOR VALID_TIME AS OF 1736467200
9      WHERE userId = 1234
10     ORDER BY _valid_from;
11
12 -- insert an update for a user profile at a given time (_id = UUID,
       userId = 1234, attributes = JSONB Object, valid_from = 10th of
       January 2025)
13 INSERT INTO customer_update RECORDS {_id: UUID, userId: 1234,
       attributes: JSONB Object, _valid_from: 1736467200}
```

Listing 6.5: Queries used to travel trough valid time and perform insertions in XTDB v2 with operation-based representaiton

As discussed on Section 5.2, bitemporal support is not a priority or requirement for Kevel's use case. However, XTDB v2 provides it out-of-the-box, offering all the advantages of bitemporality discussed without having any major development implications.

6.1.3 TerminusDB

TerminusDB, previously covered on Section 3.3.2.1, had to follow a different approach. Firstly, although the support for schemaless JSON attributes is on the roadmap of implementation, we were advised against using when discussing with the TerminusDB team, as right now it is still under development.

So instead of storing all the attributes in a JSON object, all schema had to be defined. In TerminusDB, all the stored documents are stored as class objects. Optional data types, arrays and sub documents are allowed. Enforcing a schema could be limiting in some user profile representations [77], although TerminusDB reduces this impact as it allows schema migration operations, such as deleting, updating or creating new classes [118].

The Customer class was created to store the user data, as seen in Listing 6.6. Each Customer class has as a sub document of the Attributes class, defined with the attributes names and types. In the context of Kevel's use case, defining the schema does not reflect an issue, as the attributes (and merge strategies) have to be defined before being collected and stored, and so a schema was already being enforced in some way.

```
{
  "@id": "Customer",
  "@type": "Class",
  "validTime": "xsd:dateTime",
  "attributes": "Attributes",
}
```

Listing 6.6: Schema for TerminusDB snapshot-based representation

On the snapshot-based approach, to ensure the correctness of the commit graph and to avoid expensive rebase operations and keep true to TerminusDB immutability guarantees, `PUT` updates made out of order in respect to the valid time of an object were discarded – updates to a past valid time of the last update made to a user are ignored (the updates done to a user profile are then guaranteed to be in order, although a general order is not guaranteed). This also guarantees that the most recent state of any customer is in the `HEAD` of the commit graph.

When fetching the current state of a user profile, try to read the user profile from the `HEAD` and if it exists return. When using a timestamp to retrieve the user profile at a given time, when the `HEAD` corresponds to a time after the given it is needed to time-travel.

The support for time-travelling also differed significantly for the snapshot-based approach. All changes to the database are stored in a git-like commit graph. One can query any of these commits, and browse the history as it was. It is a git-like way of time-travelling, supporting point-in-time information needs.

As it is the case for Git, travelling trough time in TerminusDB means choosing a commit to read from. To get the past commits, the history endpoint [118] is the tool available, where one can retrieve the last commits – in general or the commits where a specific object was updated. The response for the request will be a list of objects like the example in Listing 6.7. Timestamp is the automatically recorded system-time of the commit, and the message is user-defined.

```
{
  "@id":"InitialCommit/hpl18q42dbnab4vzq8me4bg1xn8p2a0",
  "@type":"InitialCommit",
  "author":"system",
  "message":"create initial schema",
  "timestamp":1660919664.9129035
}
```

Listing 6.7: Example of a commit from the history endpoint

To time-travel, one must change the commit from where to read from. There is no way of

answering "what was the first/last commit where a certain condition was met" by making a simple query, since reading from multiple commits simultaneously is not supported by TerminusDB currently.

One way of making this point-in-time requests is to go to every commit where the user object was updated, retrieve its valid time and check if its less then the valid time one wants to retrieve from. This results in going trough all commits where a object was altered and retrieving the `valid time` attribute to check if its finally the object at the correct time for potentially all commits.

To make this process faster, the commit message was changed to be the valid time. Given an objects history, Figure 6.1 demonstrates how to get the most recent state for a given valid time. By keeping the valid time on the commit message, we reduce the number of accesses to the database. Additionally, since the updates done to a user profile are kept in order, a binary search over the commits can be done to get the commit to read from.

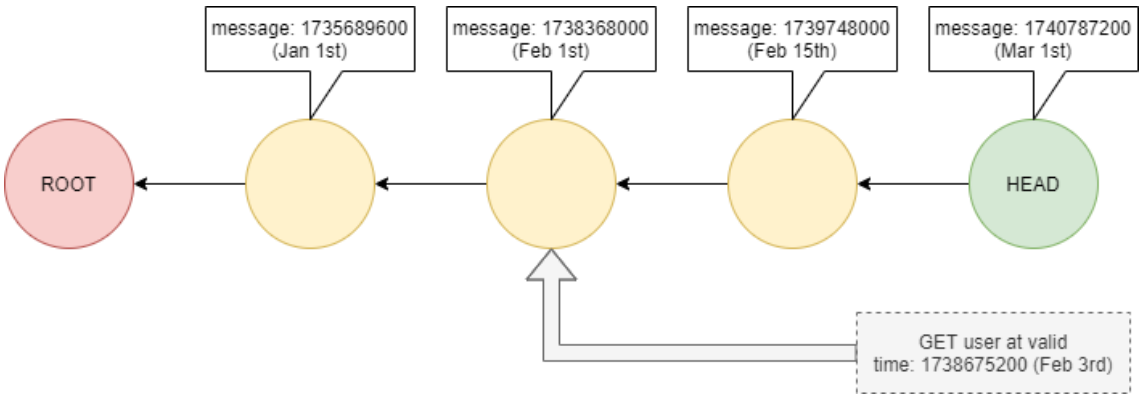

Figure 6.1: Getting the state of an object at a given valid time: TerminusDB stores all commits to the database in a git-like graph

When storing updates, on the other hand, all there was to do, similarly to the other databases, was to query for all the updates relating to a user before a given valid time.

Part of the queries and code used for the work are now part of the TerminusDB official documentation, providing a guide on how to change or create a schema as well as perform a query via the Python Client [119].

6.2 Validation

In this section, both the deployment procedures, metrics recorded and configurations used for the benchmark will be discussed.

6.2.1 Deployment

All databases were deployed using their open source "standalone" (non-distributed) Docker images, and accessed via a FastAPI [1] server. All the requests for test are made trough the server.

For the installing, configuring and running the databases for the experiment, the documentation available online was followed. In addition, for XTDB v2 and TerminusDB, valuable help from the development team and the community in general was provided in open discussion forums.

Despite having other ways of running and connecting to the selected databases, all experiments were done with the respective default Docker installation and Python clients on the server component to connect to them.

To facilitate the testing process, the Locust [2] open-source testing framework was used. Locust makes available tools to test web applications, trough a simple interface using Python. This tool allows to simulate the behaviour of multiple "users" doing concurrent requests to an endpoint. The "users" that communicate with the CDP storage endpoint in a industry example are different event tracking units. Locust allows programmers to define in code the expected throughput of requests per "user", by controlling the wait time between requests. It is important to note that Locust does not guarantee this throughput – if the tasks execution time cannot stay under the needed time, the wait time will simply be 0 before starting the next task.

The experiment is prepared to be set up both locally and on multiple machines. To exclude possible biases and increase reproducibility, the experience was made on three separate EC2 M7i instances[3] provided by Amazon Web Services (AWS). The communication between the instances of this experiment is depicted on Figure 6.2.

6.2.2 Metrics recorded

When executing the tests, the main metrics recorded were the space occupied and time taken per request, relating to the hypothesis of a trade-off between these two. For space, information was gathered both by querying the database, when possible, which was the case of PostgreSQL, that provided tools to get the size of various objects in the database trough queries[120].

To address the lack of sufficient tools to extract information on how much space each part of the database occupied for XTDB v2 and TerminusDB, the results of disk usage commands on specific docker folders were used to complement the query results (script that periodically performs data usage commands on specific files/folders, saving the results on the database machine). These results were recorded periodically during runtime.

When measuring the time, for every request, the time taken was measured. Additionally `PUT` and `GET` requests were further classified into categories based on their request type and corresponding response. The response times and number of requests for each category were grouped

[1] FastAPI framework, high performance, easy to learn, fast to code, ready for production (https://fastapi.tiangolo.com/)

[2] Locust: An open source load testing tool (https://locust.io/)

[3] Amazon EC2 M7i instances (https://aws.amazon.com/ec2/instance-types/m7i/)

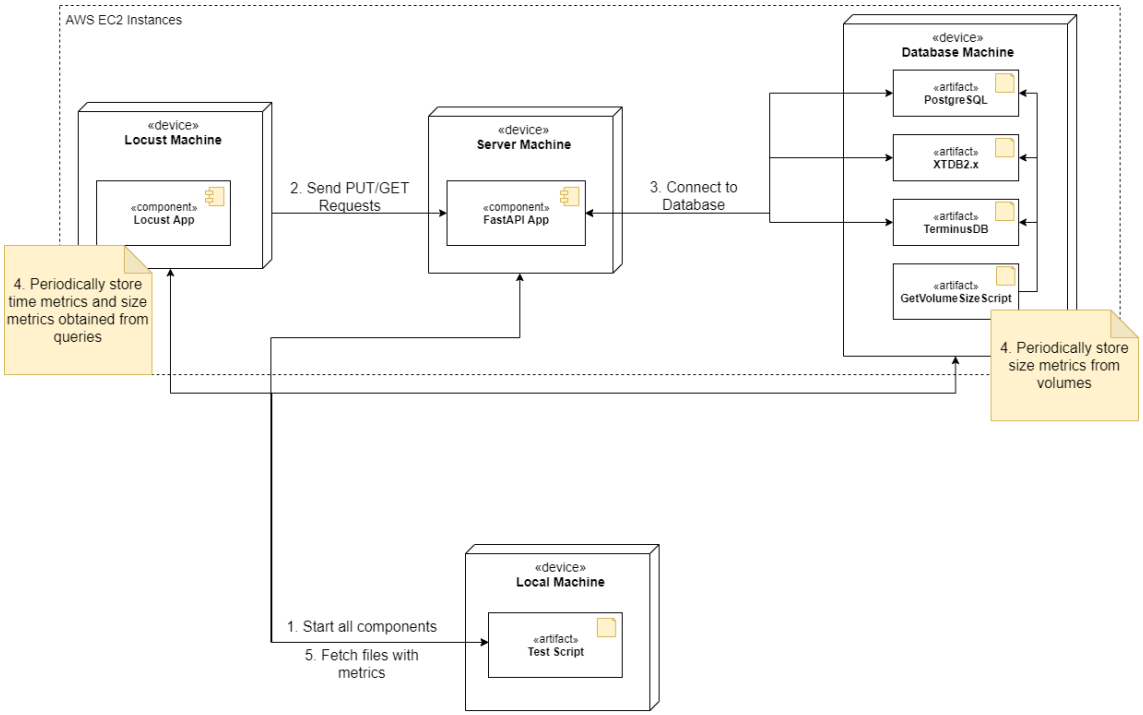

Figure 6.2: Database Benchmark Deployment: Devices, components and the communication between the parts of the system

according to the types shown in Table 6.1. Periodically, the minimum, mean, and maximum response times were recorded, along with multiple percentiles (e.g., 25th, 50th, 75th, 99th).

For every database, the operation-based mode is always classified as "MOST_RECENT", since the process of inserting an update is the same independent of the valid time of insertion. The timestamp used on the "TIMESTAMP" mode is uniformly sampled from the interval between December 1st 2024 and March 31st 2025. This timestamp corresponds either to a past state, a current state or a time where the user was not yet present in the database. When there is no user present at that point in time, the response is "NO_USER_AT_TIME".

6.2.3 Test Configuration

For each separate test, there were multiple factors that could be changed, including the database to be tested, the total time of the experiment, the time between each collection of metrics, the way of storing the updates (snapshot-based or operation-based), the number of users connected, the number of request per second (per user), the percentage of `GET`/`PUT` requests, the percentage of GET requests made without a timestamp. The flow of the experiment and its different steps are also depicted on Figure 6.2.

Table 6.2 highlights the different parameters used for the benchmark. These tests pretend to simulate the conditions covered in Chapter 5, and the reasoning behind the choice of the parameters is highlighted throughout. The test were done using all the combinations of the chosen

Table 6.1: Operation types, categories, and their descriptions.

Operation	Type	Description
PUT	MOST_RECENT	Update inserted as the latest change in the profile timeline.
	PAST	Update inserted but determined to be older than the most recent version.
	NO_UPDATE	*TerminusDB 'Snapshot' mode only*: ignore updates made to a past state of a user profile.
GET	CURRENT	Retrieves the current version of the user profile (no timestamp).
	TIMESTAMP	Retrieves the profile as reconstructed at a specific point in time.
	PAST	*TerminusDB 'Snapshot' mode only*: query with a timestamp that needs to use the History endpoint. Subset of TIMESTAMP. Used to study the time taken for travelling in TerminusDB.
	NO_USER_AT_TIME	No profile data found for the user at the specified timestamp.

parameters. A total of 8 different tests per database were expected, amounting to 48 hours of testing time.

Table 6.2: Experiment Configuration

Parameter	Value
Database	XTDB v2, TerminusDB, PostgreSQL
Mode	'Snapshot' and 'Operation'
Total time (m)	120m (2 hours)
Time collection (m)	30m
Number of Users (Locust)	1, 10 users
Rate	1, 10 requests/second
GET Percentage	30%
Percentage of GETs as of NOW	99%

6.3 Results and Discussion

In this section, the results and discussion of the experiment will be presented.

Firstly, the time taken per request will be evaluated. Both a between databases and between storage modes comparison will be made. Then, the rate of request and the number of users will be increased to uncover how the databases handle high loads. Afterwards, the space occupied per mode in each database will be compared. Finally, a discussion on what database better suits this problem and why.

6.3.1 Time taken per request

We expected to see faster retrieval times and slower update times for the 'Snapshot' mode when compared to the 'Operation' mode, taking into account the trade-off that is inherit to storing full states against operations [38].

Starting with an overview of retrieval times per request. In Figure 6.3 [4] we have a condensed overall idea of each database and mode performance, from where several key observations emerge. Notice that the logarithmic scale of the y-axis, which signifies that even small vertical differences represent substantial actual time variations.

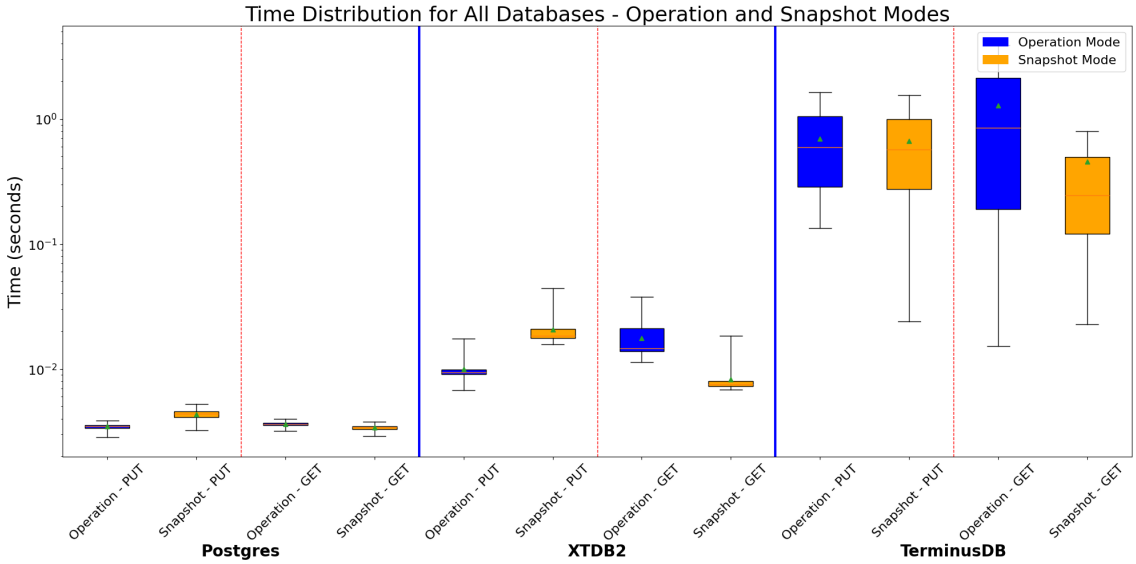

Figure 6.3: Time Distribution of requests for All Databases - 1 User, 1 Request per second (logarithmic scale)

In PostgreSQL (leftmost section of Figure 6.3), all operations exhibit extremely low median times, clustering around 3 to 5 milliseconds. The box plots for PostgreSQL are very compact, indicating minimal variability in performance for all operations under both modes.

XTDB v2 (middle section of Figure 6.3), occupies a general middle position in terms of time performance, being its response times mostly higher than PostgreSQL and lower than TerminusDB.

Finally, TerminusDB consistently (rightmost section of Figure 6.3) demonstrates the highest overall response times and variability across all tested operations and modes. A significant part of the distribution is above the 1 second mark (10^0 seconds), which implies a failure to fully comply with the 1 request per second aimed throughput.

Since TerminusDB fails to achieve even this throughput, this and subsequent comparisons will be made using the data from the 1 (Locust) User, 1 Request per second tests, the baseline for this benchmark. Affects of increasing the load will be explored in Subsection 6.3.2.

[4] The whiskers on the box plots boxes where constructed with the minimum value (lower whisker) and the 99th percentile value (upper whisker) to improve readability. Green triangles indicate the average time.

In Figure 6.3 one can also already observe a trend comparing the 'Operation' and 'Snapshot' modes.

To better analyse the 'Operation' and 'Snapshot' mode comparison, Figure 6.4, Figure 6.5 and Figure 6.6, contain the information per database also depicted in Figure 6.3.

For PostgreSQL (Figure 6.4), the time-wise impact on the difference between storage modes is minimal when compared to the other databases, since the response times are so condensed. The average `PUT` request is approximately 25% slower on 'Snapshot' mode than on 'Operation' mode, whereas `GET`s are about 7% faster than 'Operation'. Despite these specific percentages PostgreSQL's overall performance remains high regardless of the mode.

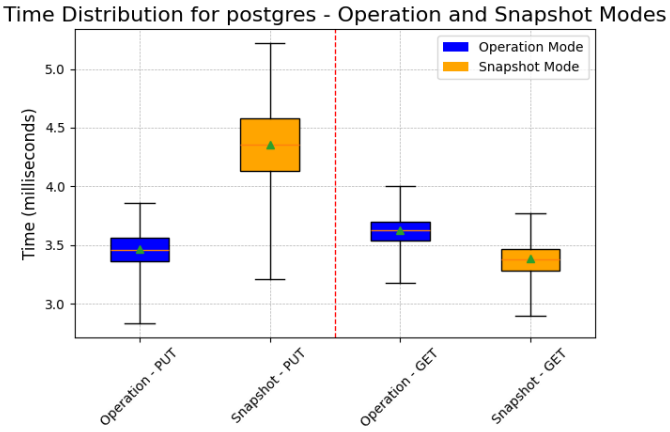

Figure 6.4: Time Distribution of requests for PostgreSQL - 1 User, 1 Request per second

Moving onto XTDB v2 (Figure 6.5) and TerminusDB (Figure 6.6), for both databases, `GET` requests achieve average slower response times in the 'Operation' mode – 52% and 64% slower than the 'Snapshot' mode respectively.

For the `PUT` requests, 'Snapshot' mode `PUT` (0.0207s) more than doubling the average time of the 'Operation' (0.0100s) for XTDB v2. TerminusDB, 'Snapshot' mode `PUT` (0.6655s) is, the only average value that appears to go against this general trend, being 4.4% faster than the 'Operation' based (0.6963s) scenario.

It becomes clearer that the `PUT` operations tend to be slower on the 'Snapshot' mode, in contrast with the faster `GET` operations. The summary of the times averages can be found in the next section, in Table 6.4.

When a timestamp was added to a `GET` request, or a `PUT` was made to a past state, the response to the request tended to be higher than the average request, as seen on Table 6.3[5]. This is most noticeable in XTDB v2 and TerminusDB, both optimized to perform actions on current data.

TerminusDB exhibits this most dramatically, particularly for `GET` requests with a timestamp in 'Snapshot' mode, where the time can escalate into tens of seconds.

[5]Since the timestamps are random, it is not guaranteed there will be data for the NO_USER_AT_TIMESTAMP category. When no data is available, it is marked with 'NA'. Also to note that PUTs to the past on 'Snapshot' mode does not apply to TerminusDB.

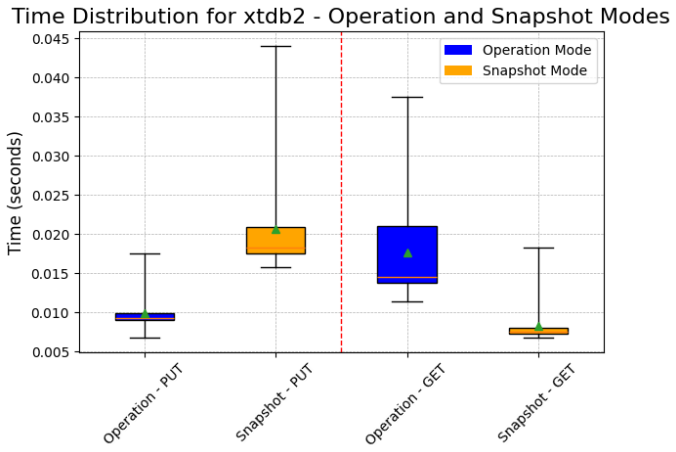

Figure 6.5: Time Distribution of requests for XTDB v2 - 1 User, 1 Request per second

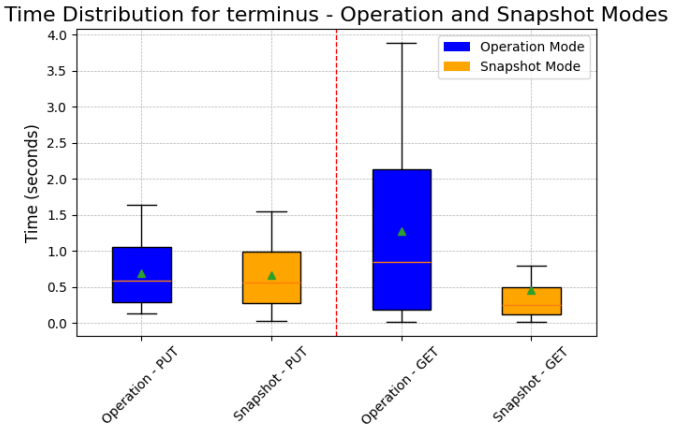

Figure 6.6: Time Distribution of requests for TerminusDB - 1 User, 1 Request per second

The magnitude of the difference in the time of the TIMESTAMP requests in TerminusDB was further investigated. We ran a test with a `GET` percentage as of "NOW" fixed at 0% so that all the requests were made using a timestamp, for 5 minutes testing time. Depicted on Figure 6.7, is the relation between the number of `PUT` requests, and consequently the number of commits, and the average time and maximum times to respond to "PAST" `GET` request, one that uses the History endpoint. It was uncovered that the History endpoint, used to get the commits where a document is modified, has a response time that increases with the number of total commits, and so for large commit graphs (with thousands of commits) the time taken can get in tens of seconds.

In the next section, the impacts of increasing the rate and the number of users will be evaluated.

6.3.2 Increasing the load

To test the databases' behaviour against an increase in load (being by an increase of users or an increase in the number of requests per second), throughput was measured against expected request rates. The expected throughput corresponds to 1, 10, 10 and 100 requests per second

Table 6.3: Average Time (s) for the different types of database operations - 1 User, 1 Request per Second

Operation		PostgreSQL	XTDB2	TerminusDB
		Time (s)		
Operation-based				
PUT	ALL	0.0035	0.0100	0.6963
	MOST_RECENT	0.0035	0.0100	0.6963
GET	ALL	0.0036	0.0176	1.2781
	CURRENT	0.0036	0.0176	1.2739
	NO_USER_AT_TIME	0.0035	0.0171	*NA*
	TIMESTAMP	**0.0036**	**0.0212**	**1.7850**
State-based				
PUT	ALL	0.0044	0.0207	0.6655
	MOST_RECENT	0.0043	0.0203	0.6809
	NO_UPDATE	—	—	0.3037
	PAST	**0.0050**	**0.0284**	**—**
GET	ALL	0.0034	0.0083	0.4564
	CURRENT	0.0034	0.0083	0.3171
	TIMESTAMP	**0.0034**	**0.0109**	**22.3426**
	NO_USER_AT_TIME	0.0033	0.0093	*NA*

(combining all users) for the four configurations tested, respectively. The expected requests are then the throughput times the number of seconds per 2-hour test: $2 \times 60 \times 60 \times troughput = 7200 \times troughput$. The actual throughputs are compared against the expected throughput (represented by the dotted green line) in Figure 6.8.

PostgreSQL consistently met all expected request rates. This robust performance aligns with its previously observed fast and stable response times and confirms that it maintains this performance. XTDB v2 failed to maintain on par with the expected number of requests for the last configuration (10 users, 10 requests per second).

As previously discussed, TerminusDB high individual response times, often exceeding one second meant it failed to meet the expected throughput for the first test. Consequently, no further load tests were conducted for TerminusDB beyond this initial low-load condition.

With the increase in the number of requests, it was noticeable that increasing the number of users, and therefore the contention for the server, the time to fulfil a request increased. As shown in Table 6.4, this behaviour is expected, taking into account that multiple users can place requests at the same time (increasing the rate of requests without increasing the number of users just sends more requests sequentially, not more requests concurrently). This heightened contention often leads to longer queueing times for requests and increased processing overhead, consequently resulting in an observed rise in the average time required to fulfil each individual request. The effects of contention provoked by simultaneous requests should be reduced by increasing the number of CPUs available, since more requests could be handled at the same time.

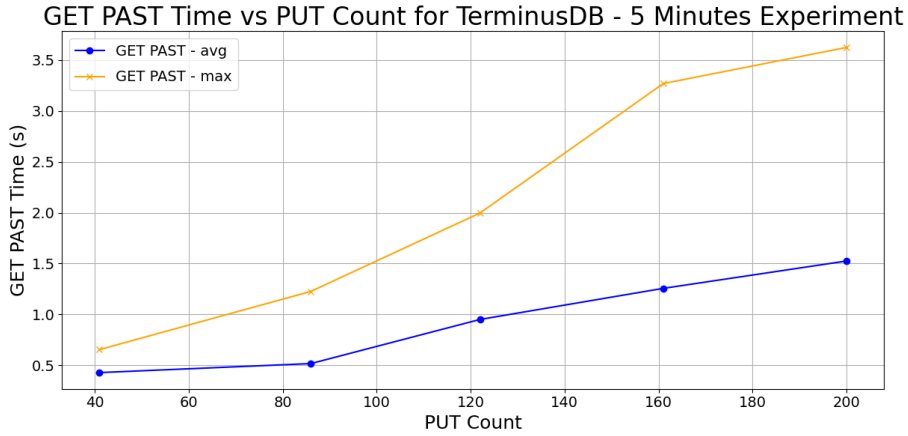

Figure 6.7: Evolution of the average time and maximum time taken per "PAST" `GET` request in relation to the number of `PUT` operations - 'Snapshot' Mode, 1 User, 1 Request per second

Table 6.4: Average Time per Request and number of all PUT operations for Postgres, XTDB v2 and TerminusDB, for both storage modes

	Operation Mode			Snapshot Mode		
Load (R, U)	**PUT Count**	**PUT (s)**	**GET (s)**	**PUT Count**	**PUT (s)**	**GET (s)**
PostgreSQL						
1, 1	5028	0.0035	0.0036	5080	0.0044	0.0034
1, 10	50485	0.0082	0.0085	50547	0.0119	0.0054
10, 1	50615	0.0035	0.0036	50226	0.0043	0.0033
10, 10	503592	0.0050	0.0052	503196	0.0064	0.0037
XTDB v2						
1, 1	4985	0.0100	0.0176	5143	0.0207	0.0083
1, 10	50354	0.1050	0.1555	50276	0.1115	0.0251
10, 1	50177	0.0092	0.0645	50185	0.0207	0.0082
10, 10	153152	0.2921	0.3823	207642	0.2970	0.0842
TerminusDB						
1, 1	4049	0.6963	1.2781	3441[a]	0.6655	0.4564

[a]Out of the 3441 request attempted to TerminusDB, 141 resulted in no update, as TerminusDB ignores updates to an object's past.

6.3.3 Space occupied

Moving onto to the space occupied by each of the databases and the two storage modes. The data used to compare space occupied will be the test with higher throughput where the database matched the expected throughput. This way the number of `PUT` requests and the space occupied between modes should be comparable, and the load on the database is the biggest possible.

The focus will be on comparing the modes of operation within the database system. The space occupied by each database is not accurately comparable, since the mechanisms of storing

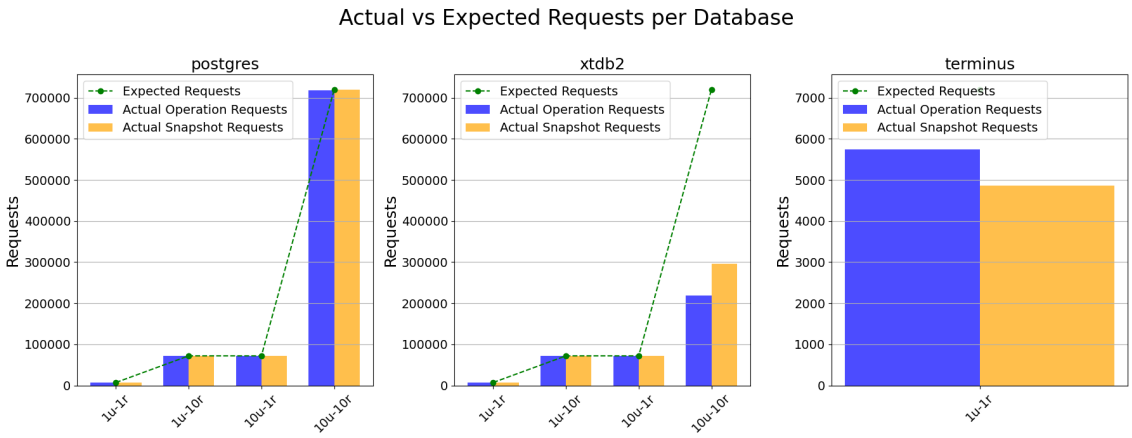

Figure 6.8: Actual vs Expected Requests Per Database for Varying Loads (u for users, r for requests per second/rate)

information are different, and due to limitations of the need to use disk usage commands used for XTDB v2 and TerminusDB the measurements are inherently flawed.

Starting with PostgreSQL, Figure 6.9 demonstrates how the storage space grows with the number of `PUT` requests made. The operation-based approach occupied more space than the snapshot-based counterpart.

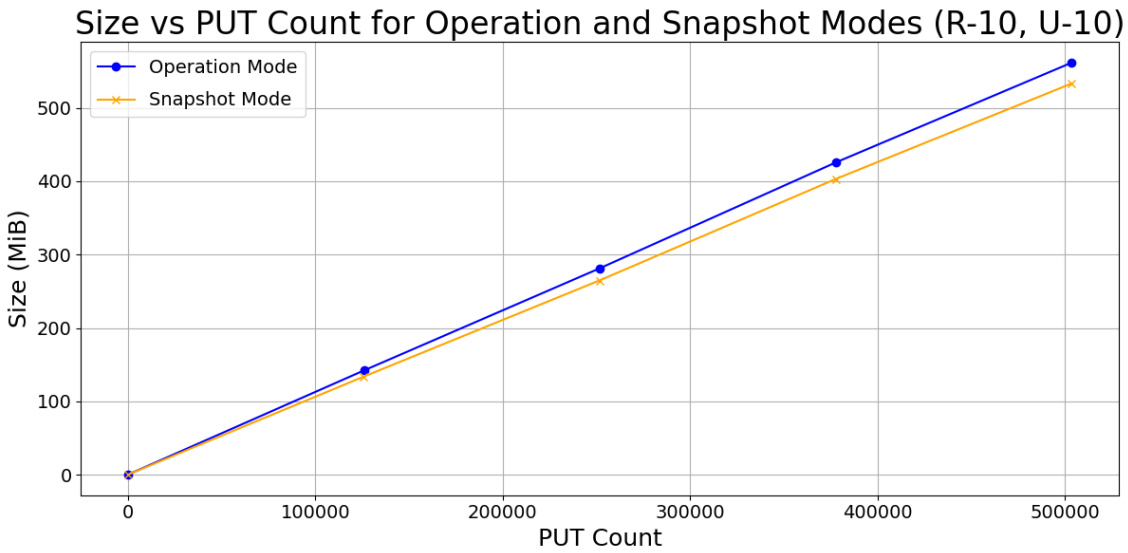

Figure 6.9: Space Usage (Total relation size) for PostgreSQL - 10 Users, 10 Requests per second

The size, measured in MiB, was recorded using the SQL commands provided by PostgreSQL [120], and is the total relation size of each test, comprising of the table data plus the index data. The Figure 6.10 demonstrates how the size is distributed between table data and indexes. PostgreSQL additionally keeps a Write-Ahead Log, metadata, and configuration files.

XTDB's space occupation is depicted on Figure 6.11. The test of 1 user and 10 requests per second was used, although the 10 users and 1 request per second test had the same throughput

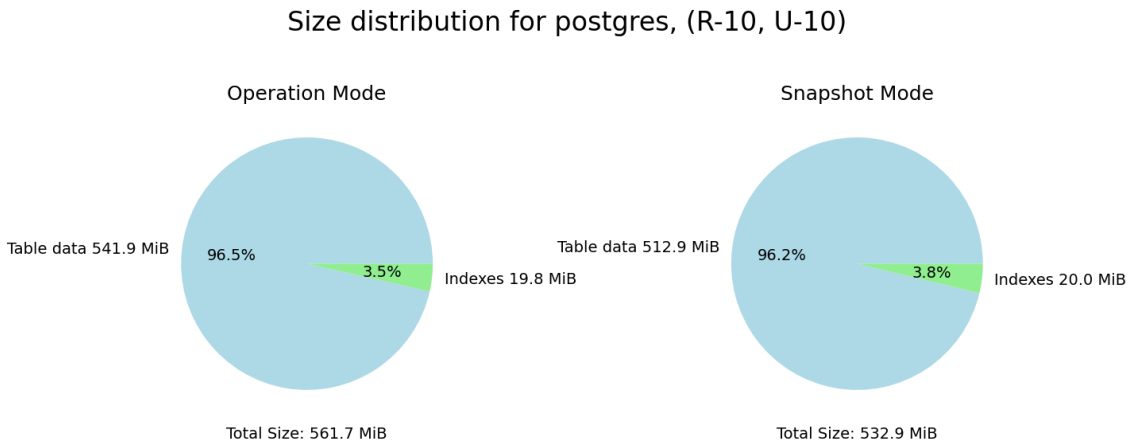

Figure 6.10: Space distribution for PostgreSQL, data and index usage - 10 Users, 10 Requests per second

that was achieved gave similar results. XTDB v2 stores the information in a growing immutable Write-Ahead Log, and when the log reaches a certain size offloads some information to storage containing the database state – including the tables, indexes, and supporting metadata. The storage consists of immutable Apache Arrow files that together represent a snapshot of the database's state at a specific point in the transaction log [121]. The log for the 'Operation' mode grows faster than the one for the 'Snapshot', but the state offloads the data to storage first, which can been seen by the steep increase of the snapshot-based approach size between the around 38000 and 50000 requests.

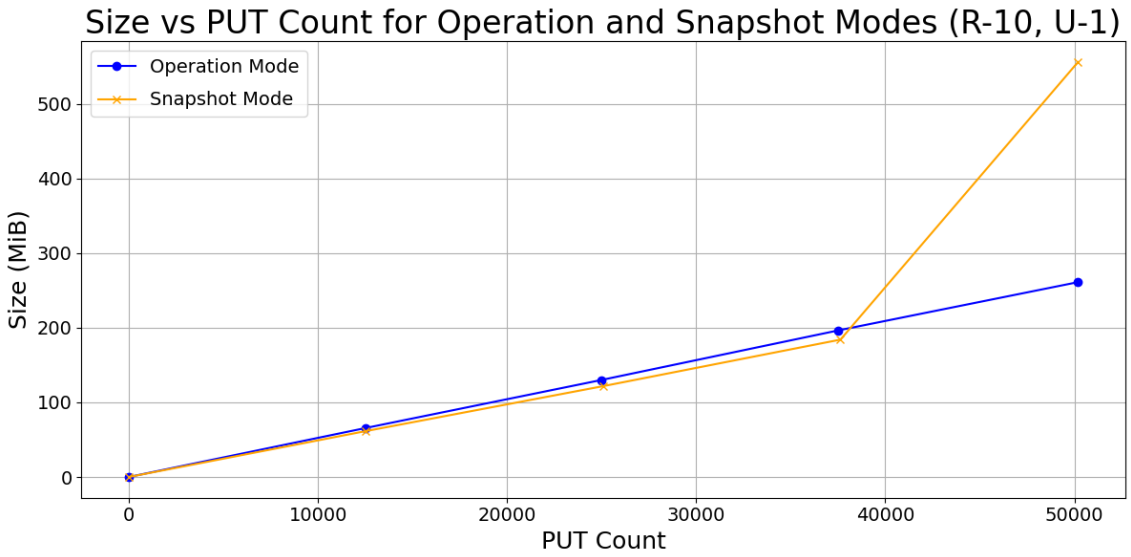

Figure 6.11: Space distribution for XTDB v2, log and buffer size - 1 User, 10 Requests per second

As the database grows in size and more storage files are created, the storage will occupy more and more of the whole space of the database, as seen on Figure 6.12.

Lastly, the evolution of size occupied by TerminusDB is on Figure 6.13. The test used for

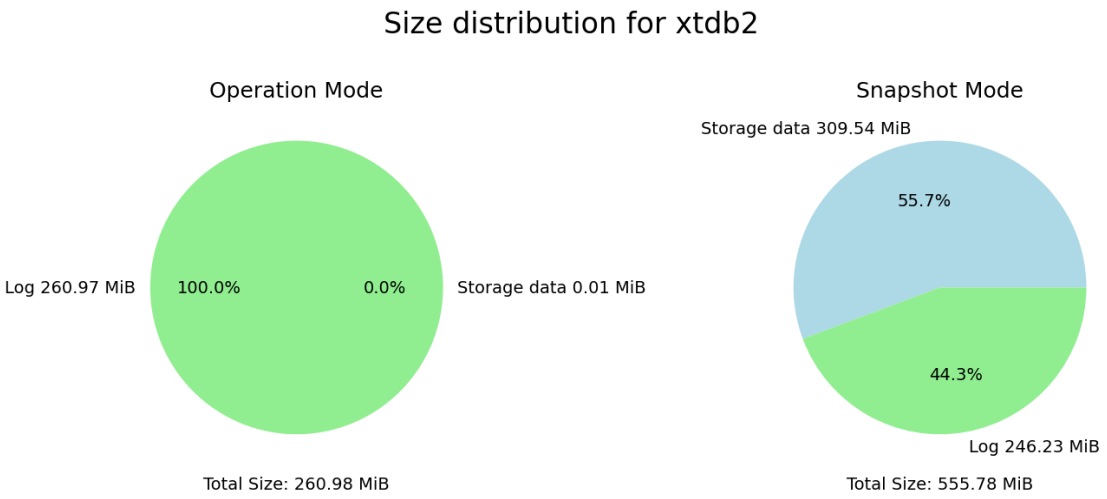

Figure 6.12: Space distribution for XTDB v2, log and buffer size - 1 User, 10 Requests per second

Terminus is the 1 User and 1 Request per second configuration, the only one completed for this database. Then again, the operation-based approach takes up more space than the snapshot-based one, for the same number of requests inserted, with a small margin. For the snapshot-based mode, only the PUTs that resulted in insertion to the database are considered.

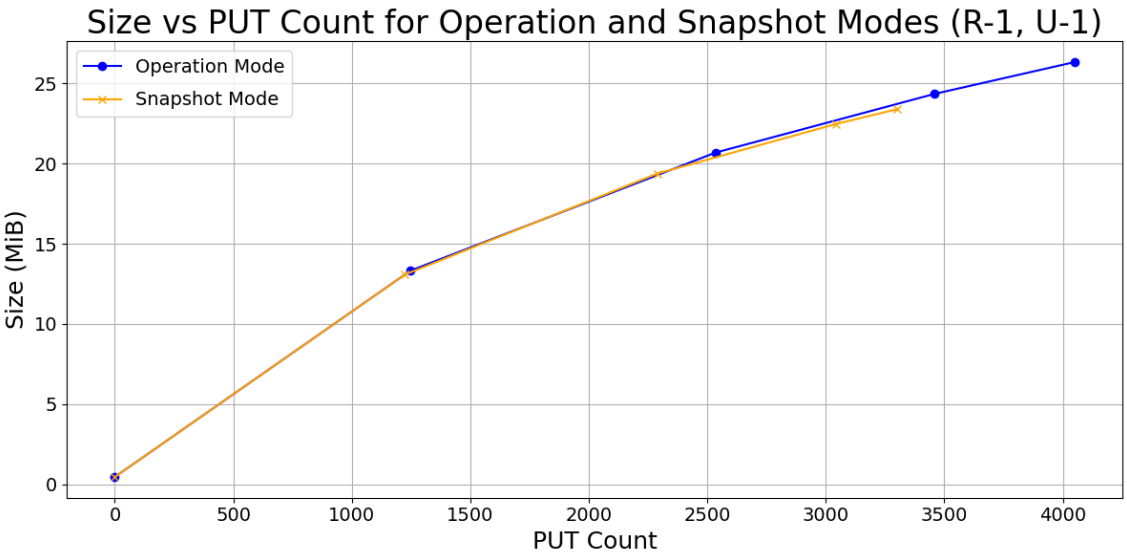

Figure 6.13: Space distribution for TerminusDB, total size - 1 User, 1 Request per second

Having analysed the results of this benchmark, we will now discuss the findings that arose from this experience.

6.3.4 Discussion

On the trade-off between the 'Snapshot' and 'Operation' modes, deciding what the best option would be depends on the use case. Having a fast retrieval time on reads is crucial to ensure

a smooth user experience and customer satisfaction [122]. Writes can be done asynchronously in background and a bigger write latency should not affect a client's experience. Ultimately, it depends on the requirements and needs of each system.

PostgreSQL was the best performer for this workload, which can be explained by its fast performance on accessing and writing information by looking for indexed values. It is known its speed and robustness when compared to other relational [123] and non-relational [124] databases for various use cases. The support for unstructured data with the JSON data types makes it suited for the workload of user profile management.

In the light of space used, snapshot-based approach on storing the user profile did not demonstrate to take more space than the operation-based approach. It is supposed that a combination of the characteristics of the dataset, where nearly 30% of users were updated only once, and using one extra column for relation by adding an 'Update ID' in addition to the 'User ID' are reasons to explain how the impact of storing full states was not the expected.

In a context where user profiles are represented by a model with connected entities – by the use of linking trough ID matching [2] or by highly connected contexts like social networks[69] – graph databases should reduce the current performance gap verified to PostgreSQL, which is more suited for fast accesses of indexed information than the other databases tested.

XTDB v2 and TerminusDB still present very compelling features for other use cases [86][125]. Additionally, all three tested databases have a active team actively improving their products together with the community and partners, as their performance and usability is only expected to improve.

6.4 Threats to Validity

The constraints and limitations encountered during the development of the experiment are reflected in the following threats to validity. Both external threats (threats related to the generalization of results) and internal threats (threats of the results being caused by factors not taken into account) were identified [126]:

The results may not be generalized to other workloads (external threat) Having tested only data and needs that were correspondent to Kevel Audience's workload, the results may not generalize for other industries and settings. This is a known common external threat when performing benchmarks [113].

The volume of the test may not be sufficient (external threat) The actual volume of the test (both in number of request as well as the size of the actual payloads involved) may not be sufficient to exhibit meaningful differences between the snapshot-based and operation-based representations.

Measuring space metrics using data usage commands (internal threat) Ideally, all the metrics on space usage should come from the database, to ensure that the results are not susceptible

to biases that are inherit to measuring folder or file sizes (for example, folders and files containing metadata that should be ignored and folders having a size bigger than the sum of its contents).

Single run for each test can be subject to bias (internal threat) All the tests for each database were only ran once. This hurts the confidence of the results, as they are more susceptible to biases and external factors. Multiple should have been made, and the results would be the average of the runs, although we were not able to replicate the tests due to time constraints.

Databases are storing different historical information (internal threat) XTDB v2 is a bitemporal database that always stores both transaction and valid time, and keeps a complete and immutable history of the database. That means that even if the state of an object is rewritten, the old state will be stored in the database, with a bounded interval of transaction time from the moment it was created until the moment it was overwritten (over both valid and transaction time). In the PostgreSQL implementation, when overwriting a past state the information is effectively deleted, and so only the states of the users over the valid time (and not over both valid and transaction time) are stored, which was the only requirement for the system. In TerminusDB, `PUT`s to past states are ignored to maintain the structure of commits ordered by valid time to allow to search for the valid state at a certain commit time.

6.5 Replication Package

To allow the full replication of the testing setup and validation steps presented in this chapter, as well as to detail the instructions on how to replicate them, an experiments replication package was built, featuring the following content [127]:

1. Code for the server application, including the queries used to interact with the different databases.
2. Code for the Locust application, to make requests for the server application.
3. Scripts for database initialization and to measure the size of specific folders.
4. Scripts to setup and start the test, including fetching the results at the end of the experiment.
5. The raw data of the results from the conducted tests.
6. A notebook to construct the visualizations presented in this work.

6.6 Summary

This chapter provided a comprehensive evaluation of three database systems – PostgreSQL, XTDB v2, and TerminusDB – under both operation-based and snapshot-based temporal storage strategies, using a real-world workload derived from anonymized user profile data.

This aimed to study the effects of the different modes of storage in time and space occupied, and how different databases would perform in the context of user profile management. PostgreSQL demonstrated the most stable and efficient performance, maintaining low latencies and high throughput even under increased load.

Finally, the internal and external threats [126] to the validity of the experience and the replication package with all the files needed to replicate the experiment and the analysis of results were presented.

Chapter 7

Conclusions

Contents

In this chapter, the main conclusions on this work are highlighted (Section 7.1) as are the contributions this work brought to the scientific and open source communities (Section 7.2). Finally, Section 7.3 describes the main challenges encountered and Section 7.4 the opportunities for future work.

7.1 Conclusions

With the growing importance of data in the context of decision-making and customer development, Customer Data Platforms are gaining relevance in the marketing domain to aid segmentation and targeting of customers, and thus improving the cost efficiency of marketing campaigns [26].

This dissertation then sets out to investigate the **versioning of user profiles in the context of user data management, researching efficient methods to store, query, implement, and maintain user profile versions over time**, as the efficient storage of user profile versions is the first step to leverage historical data to uncover trends and anticipate the future customer segments and preferences.

A review of relevant background literature, to address the concepts that serve as a backbone for the rest of this work was presented in Chapter 2, as well as a review of the state of the art on user profile representation and open source temporal databases on Chapter 3. With this research, we were able to respond to the following Research Questions:

RQ1: What is a user profile, and how can we represent user information?

A user profile is a collection of information associated with a user, built from user interactions, being a collection of identifiers (such as cookies or email addresses) and attributes

[23]. From the literature review we identified three common models to represent user information: Ontology-based, Concept-based and Graph-based [69], highlighting examples from research that used this same types of representation.

RQ2: What unique constraints and characteristics are presented by user profile data when implementing versioning?

User profiles are usually constructed of information from multiple sources and in multiple formats, resulting in a often unstructured nature [77]. User profiles are built both with information inserted by users and harvested from their interactions (explicitly or implicitly collected), changing over time to reflect changes in users' behaviours and interactions [69]. Finally, the actual characteristics and representation of each user profile are dependant on the domain they are applied, and can change depending on the information needs and uses cases [23]. User profiles must also adhere to certain usage and storage guidelines, as privacy laws such as the California Consumer Privacy Act (CCPA) [33] and the General Data Protection Act (GDPR) in the European Union [32] must be followed.

RQ3: What are the state-of-the-art open source database systems that implement temporal versioning?

The databases identified, and discussed in Section 2.2 were: TerminusDB, Datomic, XTDB v1, XDTB v2, SirixDB, immudb, PostgreSQL and Dolt. All these database systems employ temporal models, either natively or with the use of extensions.

RQ4: Which metrics are commonly used to measure system performance?

Retrieval times was the metric most commonly found in benchmarks and arguably frequently the most important to measure the performance of a system [108][109]. Together with storage usage, these two metrics are essential to measure the storage-retrieval trade-off inherit with choosing the temporal model of versioning of data [38].

Standing on the knowledge uncovered in the previous chapters, Chapter 4 defines the scope of the work as comparing the performance of database management systems, based on benchmarking different databases, and presents the hypothesis the work will aim to demonstrate:

There exists a measurable and consistent trade-off between minimizing storage space and optimizing retrieval performance across different temporal data models.

Following the problem framing and hypothesis, Chapter 5 examined the context of Customer Data Platforms (CDPs), providing a concrete use case where temporal user profiles can be applied, and also the temporal models selected for analysis, snapshot-based and operation-based.

Building on this foundation, Chapter 6 presented the empirical evaluation of three open-source database systems -— PostgreSQL, XTDB v2, and TerminusDB. The benchmarking workload, derived from anonymised CDP data containing over 14 million updates across 713 thousand users

profiles, offered a realistic testbed to validate the hypothesis across key dimensions of performance: write latency, query response time, and storage usage.

The results of the benchmark lead to the following conclusions with the proposed hypothesis in mind: snapshot-based storage strategies demonstrated better read performance, a critical advantage in customer segmentation and personalization workflows, and an overall bigger write latency. However, we could not conclude for this use case this an impact on growing storage usage between the two temporal data models.

Out of the three selected databases, PostgreSQL demonstrated the most stable and efficient performance, maintaining low latencies and high throughput in all tests.

Overall, this work demonstrates that versioning user profiles through temporal database technologies can be advantageous for applications that benefit from historical introspection and behavioural trend analysis. The benchmark results and documented implementation offer a practical reference for system designers looking to incorporate temporal data capabilities into customer-centric platforms.

The next sections outline this dissertation's broader contributions, the challenges encountered during its development, and the opportunities that remain open for future exploration.

7.2 Contributions

The work developed in this dissertation resulted in the following contributions to the software engineering state of the art, and the open source database communities:

Literature Review on User Profiles: Identified and categorized relevant examples of user profile models, according to the state-of-the-art taxonomy.

State of the Art on Open Source Temporal Database: An investigation was performed to survey what are the current open-source solutions that implement temporal databases, what are their characteristics and strengths.

Open source contributions: During the development, opportunities for open source contributions for the TerminusDB database arose, both by writing an article for documentation on the Python WOQL customer [119] and by encountering and reporting a bug on a function of the Python client [128] (corrected for v11.1.15 [129]).

Reference Implementation: The replication package can be used for more than replication purposes, serving as a reference implementation [127]. Relevant both for other benchmark experiments, and also as a example of a database and client implementation, which is especially relevant for the XTDB v2 and TerminusDB databases, that lack public examples on how to start, connect and use them.

Database benchmark: A benchmark on the databases selected, in the context of user profile management, that can serve as a guidance when choosing a database for a similar context.

7.3 Challenges

Working with in development and evolving technologies was as exciting as it was challenging. Some problems or setbacks that happened during the development and test could not have been solved if not for the help of the people behind XTDB and TerminusDB, who aided the implementation process to move forward when standstills emerge.

Somewhat ironically, time was of essence for this work, and that also limited the possibilities of tests to be made during the benchmark, both in terms of number of tests and their duration. If possible longer runs would be made to evaluate the databases on higher loads and use more of the dataset available.

7.4 Future Work

The following steps were identified to both build on top of the work conducted in this dissertation:

- With the gaining relevance of cloud computing and cloud technologies [130], and being the databases adapted for distributed environments, a logical next step would be to test the databases not only on their 'standalone' versions but also compare their performance on a distributed setting.
- Adding the ID Matching capability to the study would better represent the necessities of a Customer Data Platform [2] and other use cases where entities are related within each other.
- Testing the temporal databases against an information need that requires the use of bitemporality. This would prove to be a better test case for both XTDB v2 but also an opportunity to test PostgreSQL's temporal extensions.
- Moreover, the addition of both more databases and more information needs to the experiment would provide a more complete analysis for different databases performance on temporal data.

Appendix A

Literature Review Results

Table A.1: Results of the systematic review - RQ1

Title	Year	Data Source
A Hybrid User Profile Model for Personalized Recommender System with Linked Open Data [75]	2014	ACM Digital Library
Adaptive and multiple interest-aware user profiles for personalized search in folksonomy: A simple but effective graph-based profiling model [74]	2015	IEEE Xplore
An ontology based model for user profile building using social network [72]	2019	ACM Digital Library
Improving Job Recommendation Using Ontological Modeling and User Profiles [71]	2019	IEEE Xplore
Ontologies to Model User Profiles in Personalized Job Recommendation [3]	2018	IEEE Xplore
Personalized Queries under a Generalized User Profile Model based on Fuzzy SPARQL Preferences [5]	2019	ACM Digital Library
Recommendation of Healthcare Services Based on an Embedded User Profile Model [73]	2022	ACM Digital Library
Topic-based User Profile Model for POI Recommendations [4]	2018	ACM Digital Library
User's profile ontology-based semantic model for personalized hotel room recommendation in the web of things: student research abstract [131]	2019	ACM Digital Library

Appendix B

Detailed System Configuration

Table B.1: System Information Table

Parameter	**Value**
Databases	PostgreSQL, XTDB v2 and TerminusDB
Database Version PostgreSQL	PostgreSQL 17.4
Database Version XTDB v2	XTDB 2.0.0
Database Version TerminusDB	TerminusDB 11.1.4
EC2 Instance Types	m7i.large
Total Disk Space	484 GB
Operating System	Ubuntu 24.04.2 LTS
System Memory	7.6G
CPU Type	Intel(R) Xeon(R) Platinum 8488C
Total Cores	1
Total Threads	2

Bibliography

[1] Richard T. Snodgrass and Ilsoo Ahn. Temporal databases. *IEEE Computer*, 19(9):35–42, September 1986.

[2] Tiago Boldt Sousa. Customer Data Platforms: A Pattern Language for Digital Marketing Optimization with First-Party Data. In *Proceedings of the 27th European Conference on Pattern Languages of Programs*, EuroPLop '22, pages 1–5, New York, NY, USA, February 2023. Association for Computing Machinery.

[3] S.R. Rimitha, Vedasamhitha Abburu, Annem Kiranmai, and K. Chandrasekaran. Ontologies to Model User Profiles in Personalized Job Recommendation. In *2018 IEEE Distributed Computing, VLSI, Electrical Circuits and Robotics (DISCOVER)*, pages 98–103, August 2018.

[4] Passakorn Khanthaapha, Luepol Pipanmaekaporn, and Suwatchai Kamonsantiroj. Topic-based User Profile Model for POI Recommendations. In *Proceedings of the 2nd International Conference on Intelligent Systems, Metaheuristics & Swarm Intelligence*, ISMSI '18, pages 143–147, New York, NY, USA, March 2018. Association for Computing Machinery.

[5] Olfa Slama. Personalized Queries under a Generalized User Profile Model based on Fuzzy SPARQL Preferences. In *2019 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)*, pages 1–6, June 2019.

[6] XTDB. XTDB GitHub Repository at 1.x. https://github.com/xtdb/xtdb/tree/1.x. Accessed: 2025-03-21.

[7] XTDB. XTDB GitHub Repository. https://github.com/xtdb/xtdb, October 2024. Accessed: 2024-10-18.

[8] Maurice Mulvenna, Marian Norwood, and Alex Büchner. Data-Driven Marketing. *Electronic Markets*, 8(3):32–35, January 1998.

[9] Victoria Fast, Daniel Schnurr, and Michael Wohlfarth. Data-driven Competitive Advantages in Digital Markets: An Overview of Data Value and Facilitating Factors. *Wirtschaftsinformatik 2021 Proceedings*, February 2021.

[10] Victoria Fast, Daniel Schnurr, and Michael Wohlfarth. Regulation of data-driven market power in the digital economy: Business value creation and competitive advantages from big data. *Journal of Information Technology*, 38(2):202–229, June 2023.

[11] Enric Junqué de Fortuny, David Martens, and Foster Provost. Predictive Modeling With Big Data: Is Bigger Really Better? *Big Data*, 1(4):215–226, December 2013.

[12] Ehsan Valavi, Joel Hestness, Newsha Ardalani, and Marco Iansiti. Time and the value of data. (arXiv:2203.09118), March 2022. arXiv:2203.09118 [cs].

[13] Krishna Kulkarni and Jan-Eike Michels. Temporal features in SQL:2011. *SIGMOD Rec.*, 41(3):34–43, October 2012.

[14] PostgreSQL. PostgreSQL 9.2 released. https://www.postgresql.org/about/news/postgresql-92-released-1415/, September 2012. Accessed: 2024-10-10.

[15] Immudb. Immudb - immutable database based on zero trust, SQL and Key-Value. https://immudb.io. Accessed: 2024-10-09.

[16] Termunusdb. Terminusdb. https://github.com/terminusdb/terminusdb, October 2024. Accessed: 2024-10-08.

[17] XTDB. XTDB — immutable SQL database for data compliance. https://xtdb.com/. Accessed: 2024-10-10.

[18] Datomic. Datomic Data Model | Datomic. https://docs.datomic.com/whatis/data-model.html. Accessed: 2024-10-10.

[19] Gultekin Ozsoyoglu and Richard Snodgrass. Temporal and real-time databases: A survey. *Knowledge and Data Engineering, IEEE Transactions on*, 7:513–532, September 1995.

[20] Sun Jihua and Shi Lin. Research on Data-driven Marketing Strategy Optimization. *Journal of Business and Marketing*, 1(2):157–161, March 2024.

[21] CDP Institute. What is a CDP? https://www.cdpinstitute.org/learning-center/what-is-a-cdp/, November 2020. Accessed: 2024-10-06.

[22] Kevel. Kevel Audience | Unlock your first-party data. https://www.kevel.com/audience. Accessed: 2025-01-08.

[23] Tomasz Bujlow, Valentín Carela-Español, Josep Solé-Pareta, and Pere Barlet-Ros. A Survey on Web Tracking: Mechanisms, Implications, and Defenses. *Proceedings of the IEEE*, 105(8):1476–1510, August 2017.

[24] Marc Bourreau, Alexandre de Streel, and Inge Graef. Big Data and Competition Policy: Market Power, Personalised Pricing and Advertising, February 2017.

[25] Sally Dibb and Lyndon Simkin. Targeting, segments and positioning. *International Journal of Retail & Distribution Management*, 19(3), January 1991.

[26] Ganesh Iyer, David Soberman, and J. Miguel Villas-Boas. The Targeting of Advertising. *Marketing Science*, August 2005.

[27] John Grim Jr Robert Docters and John McGady. Segments in Time. https://www.strategy-business.com/article/9325, January 1997. Accessed: 2024-10-05.

[28] Christopher Blocker and Daniel Flint. Customer segments as moving targets: Integrating customer value dynamism into segment instability logic. *Industrial Marketing Management*, 36(6):810–822, August 2007.

[29] Mirko Böttcher, Martin Spott, Detlef Nauck, and Rudolf Kruse. Mining changing customer segments in dynamic markets. *Expert Systems with Applications*, 36(1):155–164, January 2009.

[30] Zhongqi Lu, Sinno Jialin Pan, Yong Li, Jie Jiang, and Qiang Yang. Collaborative Evolution for User Profiling in Recommender Systems. In *International Joint Conference on Artificial Intelligence*, July 2016.

[31] Bong-Horng Chu, Ming-Shian Tsai, and Cheng-Seen Ho. Toward a hybrid data mining model for customer retention. *Knowledge-Based Systems*, 20(8):703–718, December 2007.

[32] European Union. Document 32016r0679 regulation 2016/679. https://eur-lex.europa.eu/eli/reg/2016/679/oj. Accessed: 2024-10-09.

[33] California Law. California Summer Privacy Act. https://leginfo.legislature.ca.gov/faces/codes_displayText.xhtml?division=3.&part=4.&lawCode=CIV&title=1.81.5. Accessed: 2024-10-10.

[34] Irum Matloob, Shoab Khan, Habib Ur Rahman, and Farhan Hussain. Medical Health Benefit Management System for Real-Time Notification of Fraud Using Historical Medical Records. *Applied Sciences*, 10(15):5144, July 2020.

[35] B. Baesens, Véronique Van Vlasselaer, and W. Verbeke. Fraud Analytics : Using Descriptive, Predictive, and Social Network Techniques:A Guide to Data Science for Fraud Detection. August 2015.

[36] Jernej Kovše. *Model-Driven Development of Versioning Systems*. Doktor-ingenieur (dr.-ing.) dissertation, Technische Universität Kaiserslautern, 2005. Approved on 4th March 2005.

[37] Koyel Mukherjee, Raunak Shah, Shiv Kumar Saini, Karanpreet Singh, Khushi, Harsh Kesarwani, Kavya Barnwal, and Ayush Chauhan. Towards Optimizing Storage Costs on the Cloud, July 2023.

[38] Souvik Bhattacherjee, Amit Chavan, Silu Huang, Amol Deshpande, and Aditya Parameswaran. Principles of Dataset Versioning: Exploring the Recreation/Storage Tradeoff. *Proceedings of the VLDB Endowment. International Conference on Very Large Data Bases*, 8(12):1346–1357, August 2015.

[39] Elliott Hauser. UNIX time, UTC, and datetime: Jussivity, prolepsis, and incorrigibility in modern timekeeping. *Proceedings of the Association for Information Science and Technology*, 55(1):161–170, 2018.

[40] Snodgrass and llsoo Ahn. Temporal Databases. *Computer*, 19(9):35–42, September 1986.

[41] Richard Snodgrass and Ilsoo Ahn. A taxonomy of time databases. *SIGMOD Rec.*, 14(4):236–246, May 1985.

[42] Christian S. Jensen, Curtis E. Dyreson, Michael Böhlen, James Clifford, Ramez Elmasri, Shashi K. Gadia, Fabio Grandi, Pat Hayes, Sushil Jajodia, Wolfgang KÄfer, Nick Kline, Nikos Lorentzos, Yannis Mitsopoulos, Angelo Montanari, Daniel Nonen, Elisa Peressi, Barbara Pernici, John F. Roddick, Nandlal L. Sarda, Maria Rita Scalas, Arie Segev, Richard T. Snodgrass, Mike D. Soo, Abdullah Tansel, Paolo Tiberio, and Gio Wiederhold. The consensus glossary of temporal database concepts — February 1998 version. In Opher Etzion, Sushil Jajodia, and Suryanarayana Sripada, editors, *Temporal Databases: Research and Practice*, pages 367–405. Springer, Berlin, Heidelberg, 1998.

[43] Richard T. Snodgrass. Temporal databases. In A. U. Frank, I. Campari, and U. Formentini, editors, *Theories and Methods of Spatio-Temporal Reasoning in Geographic Space*, pages 22–64, Berlin, Heidelberg, 1992. Springer.

[44] Christian S. Jensen and Richard T. Snodgrass. Semantics of time-varying information. *Information Systems*, 21(4):311–352, June 1996.

[45] B Schueler. Update reconsidered. *Architecture and Models in Data Base Management Systems*, pages 161–129, 1977.

[46] Richard Snodgrass. Temporal databases status and research directions. *SIGMOD Rec.*, 19(4):83–89, December 1990.

[47] Richard Snodgrass. The temporal query language TQuel. *ACM Trans. Database Syst.*, 12(2):247–298, June 1987.

[48] Richard Thomas Snodgrass, Ilsoo Ahn, Gadi Ariav, Don Batory, James Clifford, Curtis E. Dyreson, Ramez Elmasri, Fabio Grandi, Christian S. Jensen, Wolfgang Käfer, Nick

Kline, Krishna Kulkarni, T. Y. Cliff Leung, Nikos Lorentzos, John F. Roddick, Arie Segev, Michael D. Soo, and Suryanarayana M. Sripada. TSQL2 language specification. *SIGMOD Rec.*, 23(1):65–86, March 1994.

[49] Mehmet A. Orgun. On temporal deductive databases. *Computational Intelligence*, 12(2):235–259, May 1996.

[50] Johann Gamper, Matteo Ceccarello, and Anton Dignös. What's New in Temporal Databases? In Silvia Chiusano, Tania Cerquitelli, and Robert Wrembel, editors, *Advances in Databases and Information Systems*, pages 45–58, Cham, 2022. Springer International Publishing.

[51] Erik D. Demaine, John Iacono, and Stefan Langerman. Retroactive data structures. *ACM Transactions on Algorithms*, 3(2):13, May 2007.

[52] Prabhjot Prabhjot and N. Sharma. Overview of the Database Management System. *International Journal of Advanced Research in Computer Science*, 2017.

[53] Simon Loesing, Markus Pilman, Thomas Etter, and Donald Kossmann. On the Design and Scalability of Distributed Shared-Data Databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 663–676, New York, NY, USA, May 2015. Association for Computing Machinery.

[54] Thomas Connolly and Carolyn Begg. Database systems. A practical approach to design, implementation and management. January 2010.

[55] Jim Gray. The transaction concept: Virtues and limitations. In *Readings in Database Systems*, pages 140–150. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, June 1988.

[56] Jagdev Bhogal and Imran Choksi. Handling Big Data Using NoSQL. In *2015 IEEE 29th International Conference on Advanced Information Networking and Applications Workshops*, pages 393–398, March 2015.

[57] Divya Chauhan and Kartik Bansal. Using the Advantages of NOSQL: A Case Study on MongoDB. *International Journal on Recent and Innovation Trends in Computing and Communication*, 5:90–93, February 2017.

[58] Pankaj Gupta and Prakashkumar Patel. Demystifying Databases: Exploring their Use Cases. *International Journal of Computer Science and Engineering*, 10:43–53, June 2023.

[59] N.S. Patil, P. Kiran, N.P. Kavya, and K.M. Patel. A Survey on Graph Database Management Techniques for Huge Unstructured Data. *International Journal of Electrical and Computer Engineering*, 81:1140–1149, April 2018.

[60] Marc Yves Maria Kramis. *Evolutionary Tree-Structured Storage: Concepts, Interfaces, and Applications*. Doctor of natural sciences dissertation, Universität Konstanz, 2014. Approved on 22nd April 2014.

[61] Eunji Lee, Jee E. Jang, Taeseok Kim, and Hyokyung Bahn. On-Demand Snapshot: An Efficient Versioning File System for Phase-Change Memory. *IEEE Transactions on Knowledge and Data Engineering*, 25(12):2841–2853, December 2013.

[62] Haifeng Shen and Chengzheng Sun. A log compression algorithm for operation-based version control systems. In *Proceedings 26th Annual International Computer Software and Applications*, pages 867–872, August 2002.

[63] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. FASTER: A Concurrent Key-Value Store with In-Place Updates. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 275–290, New York, NY, USA, May 2018. Association for Computing Machinery.

[64] Seth Earley. The Role of a Customer Data Platform. *IT Professional*, 20(1):69–76, January 2018.

[65] Javier Lopez. Empowering Business Strategies with Customer Data Platforms: The Intersection of Big Data and Customer Intelligence for Enhanced Decision Making. *Australian Journal of Machine Learning Research & Applications*, 4(1):1–16, May 2024.

[66] Sally Dibb and Lyndon Simkin. Judging the quality of customer segments: segmentation effectiveness. *Journal of Strategic Marketing*, 18(2):113–131, 2010.

[67] Angela Carrera-Rivera, William Ochoa, Felix Larrinaga, and Ganix Lasa. How-to conduct a systematic literature review: A quick guide for computer science research. *MethodsX*, 9:101895, January 2022.

[68] Hannah Snyder. Literature review as a research methodology: An overview and guidelines. *Journal of Business Research*, 104:333–339, November 2019.

[69] Erasmo Purificato, Ludovico Boratto, and Ernesto William De Luca. User Modeling and User Profiling: A Comprehensive Survey, February 2024.

[70] D. Jacquette. *Ontology*. Central Problems of Philosophy. McGill-Queen's University Press, 2002.

[71] S.R. Rimitha, Vedasamhitha Abburu, Annem Kiranmai, Marimuthu C, and K. Chandrasekaran. Improving Job Recommendation Using Ontological Modeling and User Profiles. In *2019 Fifteenth International Conference on Information Processing (ICINPRO)*, pages 1–8, December 2019.

[72] R. Niyazova, Al. Aktayeva, and L. Davletkireeva. An ontology based model for user profile building using social network. In *Proceedings of the 5th International Conference on Engineering and MIS*, ICEMIS '19, pages 1–4, New York, NY, USA, June 2019. Association for Computing Machinery.

[73] Jianmao Xiao, Xinyi Liu, Jia Zeng, Zhiyong Feng, and Yuanlong Cao. Recommendation of Healthcare Services Based on an Embedded User Profile Model. *Int. J. Semant. Web Inf. Syst.*, 18(1):1–21, October 2022.

[74] Keejun Han, Juneyoung Park, and Mun. Y. Yi. Adaptive and multiple interest-aware user profiles for personalized search in folksonomy: A simple but effective graph-based profiling model. In *2015 International Conference on Big Data and Smart Computing (BIGCOMP)*, pages 225–231, February 2015.

[75] Yang Luo, Boyi Xu, Hongming Cai, and Fenglin Bu. A Hybrid User Profile Model for Personalized Recommender System with Linked Open Data. In *2014 Enterprise Systems Conference*, pages 243–248, August 2014.

[76] Olivier Pivert, Olfa Slama, and Virginie Thion. An extension of SPARQL with fuzzy navigational capabilities for querying fuzzy RDF data. In *2016 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)*, pages 2409–2416, Vancouver, BC, Canada, July 2016. IEEE Press.

[77] Aleksandr Farseev, Liqiang Nie, Mohammad Akbari, and Tat-Seng Chua. Harvesting Multiple Sources for User Profile Learning: A Big Data Study. In *Proceedings of the 5th ACM on International Conference on Multimedia Retrieval*, ICMR '15, pages 235–242, New York, NY, USA, June 2015. Association for Computing Machinery.

[78] Suela Isaj, Nacéra Bennacer Seghouani, and Gianluca Quercini. Profile Reconciliation Through Dynamic Activities Across Social Networks. In Paolo Giorgini and Barbara Weber, editors, *Advanced Information Systems Engineering*, pages 126–141, Cham, 2019. Springer International Publishing.

[79] TerminusDB. Time Travel Through your Database History. https://terminusdb.com/docs/time-travel-with-python/. Accessed: 2025-03-21.

[80] Miguel A. Martínez-Prieto, Mario Arias Gallego, and Javier D. Fernández. Exchange and Consumption of Huge RDF Data. In Elena Simperl, Philipp Cimiano, Axel Polleres, Oscar Corcho, and Valentina Presutti, editors, *The Semantic Web: Research and Applications*, pages 437–452, Berlin, Heidelberg, 2012. Springer.

[81] TerminusDB. Succinct Data Structures & Delta Encoding for Modern Databases. https://terminusdb.com/blog/succinct-data-structures-for-modern-databases/, January 2023. Accessed: 2024-10-18.

[82] Gavin Mendel-Gleason. Knowledge Graph Schema Design Patterns and Principles. https://terminusdb.com/blog/knowledge-graph-schema-design/, March 2023. Accessed: 2025-03-21.

[83] Datomic. Datomic - Overview. https://www.datomic.com/. Accessed: 2024-11-27.

[84] Datomic. Indexes | Datomic. https://docs.datomic.com/indexes/index-model.html. Accessed: 2025-03-21.

[85] XTDB. XTDB Docs · XTDB Docs. https://v1-docs.xtdb.com/main/. Accessed: 2025-03-21.

[86] XTDB. Launching XTDB v2 — time-travel SQL database to simplify compliance · XTDB. https://xtdb.com/blog/launching-xtdb-v2.html. Accessed: 2025-06-13.

[87] XTDB. XTDB v2. https://xtdb.com/v2.html. Accessed: 2024-11-26.

[88] XTDB. Introducing XTQL. https://docs.xtdb.com/xtql/tutorials/introducing-xtql.html. Accessed: 2025-03-22.

[89] XTDB. Language Drivers. https://docs.xtdb.com/drivers.html. Accessed: 2025-03-22.

[90] SirixDB. SirixDB GitHub Repository. https://github.com/sirixdb/sirix, November 2024. Accessed: 2024-11-27.

[91] SirixDB. SirixDB - An Evolutionary, Temporal, JSON Store. https://sirix.io/. Accessed: 2024-11-27.

[92] SirixDB - Documentation. https://sirix.io/docs/index.html. Accessed: 2024-10-21.

[93] Johannes Lichtenberger. Why And How We Built a Temporal Database System Called SirixDB (Open Source) From Scratch. https://hackernoon.com/why-and-how-we-built-a-temporal-database-system-called-sirixdb-open-source-from January 2019. Accessed: 2024-10-21.

[94] Immudb. Immudb GitHub Repository. https://github.com/codenotary/immudb, December 2024. Accessed: 2024-12-02.

[95] Immudb. Immudb - The lightweight, high-speed immutable database. https://immudb.io. Accessed: 2025-03-22.

[96] Michael Paik, Jerónimo Irazábal, Dennis Zimmer, Michele Meloni, and Valentin Padurean. Immudb: A Lightweight, Performant Immutable Database. https://codenotary.s3.amazonaws.com/Research-Paper-immudb-CodeNotary_v3.0.pdf.

[97] Immudb. Immudb release v1.2 - immudb. https://immudb.io/blog/immudb-release-1-2. Accessed: 2024-12-02.

[98] PostgreSQL. PostgreSQL. https://www.postgresql.org/, December 2024. Accessed: 2024-12-02.

[99] PostgreSQL. 8.14. JSON Types. https://www.postgresql.org/docs/17/datatype-json.html, May 2025. Accessed: 2025-06-20.

[100] PostgreSQL. Temporal Extensions - PostgreSQL wiki. https://wiki.postgresql.org/wiki/Temporal_Extensions. Accessed: 2024-12-02.

[101] PostgreSQL. Chapter 11. Indexes. https://www.postgresql.org/docs/17/indexes.html, May 2025. Accessed: 2025-06-21.

[102] Dolt. Dolt GitHub Repository. https://github.com/dolthub/dolt, October 2024. Accessed: 2024-10-08.

[103] Dolt. What Is Dolt? | Dolt Documentation. https://docs.dolthub.com, May 2024. Accessed: 2024-12-02.

[104] Dolt. Overview | Dolt Documentation. https://docs.dolthub.com/architecture/architecture, April 2024. Accessed: 2024-12-02.

[105] Zach Musgrave. Dolt Use Cases | DoltHub Blog. https://dolthub.com/blog/2024-10-15-dolt-use-cases/, October 2024. Accessed: 2024-12-02.

[106] Noa Roy-Hubara. The Quest for a Database Selection and Design Method. In *International Conference on Advanced Information Systems Engineering*, 2019.

[107] Ilsoo Ahn and Richard Snodgrass. Performance evaluation of a temporal database management system. *SIGMOD Rec.*, 15(2):96–107, June 1986.

[108] Chunxi Zhang, Yuming Li, Rong Zhang, Weining Qian, and Aoying Zhou. Benchmarking on intensive transaction processing. *Front. Comput. Sci.*, 14(5), October 2020.

[109] Benymol Jose and Sajimon Abraham. Performance analysis of NoSQL and relational databases with MongoDB and MySQL. *Materials Today: Proceedings*, 24:2036–2043, 2020.

[110] B. Sirish Shetty and K. Akshay. Performance Analysis of Queries in RDBMS vs NoSQL. *2019 2nd International Conference on Intelligent Computing, Instrumentation and Control Technologies (ICICICT)*, 2019.

[111] Seref Sagiroglu and Duygu Sinanc. Big data: A review. In *2013 International Conference on Collaboration Technologies and Systems (CTS)*, pages 42–47, May 2013.

[112] Sami M. Halawani and Nashwan A. Al-Romema. Memory Storage Issues of Temporal Database Applications on Relational Database Management Systems. *Journal of Computer Science*, 6(3):296–304, March 2010.

[113] Marvin V. Zelkowitz and Dolores R. Wallace. Experimental Models for Validating Technology. *Computer*, 31(5):23–31, May 1998.

[114] Richard T. Snodgrass. *Developing Time-Oriented Database Applications in SQL*. The Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann, San Francisco, Calif, 2000.

[115] Kyzer R. Davis, Brad Peabody, and P. Leach. Universally Unique IDentifiers (UUIDs). Request for Comments RFC 9562, Internet Engineering Task Force, May 2024.

[116] XTDB. Key concepts. https://docs.xtdb.com/concepts/key-concepts.html. Accessed: 2025-06-02.

[117] XTDB. What is XTDB? · XTDB Docs. https://v1-docs.xtdb.com/concepts/what-is-xtdb/. Accessed: 2024-11-26.

[118] TerminusDB. TerminusDB - API Spec. https://terminusdb.org/docs/openapi/. Accessed: 2025-06-17.

[119] Dinis Bessa de Sousa. Python WOQL Customer Data Processing Example - TerminusDB Documentation. https://terminusdb.org/docs/python-woql-customer-data-processing-example/. Accessed: 2025-06-07.

[120] PostgreSQL. Disk Usage - PostgreSQL wiki. https://wiki.postgresql.org/wiki/Disk_Usage. Accessed: 2025-06-12.

[121] Backup and Restore. https://docs.xtdb.com/ops/backup-and-restore/overview.html. Accessed: 2025-06-22.

[122] Vamsi Thatikonda and Hemavantha Rajesh Varma Mudunuri. Building Data-Intensive Applications: Scalability, Performance and Availability. *The Review of Contemporary Scientific and Academic Studies*, 3, October 2023.

[123] Sanket Vilas Salunke and Abdelkader Ouda. A Performance Benchmark for the PostgreSQL and MySQL Databases. *Future Internet*, 16(10):382, October 2024.

[124] Antonios Makris, Konstantinos Tserpes, Giannis Spiliopoulos, Dimitrios Zissis, and Dimosthenis Anagnostopoulos. MongoDB Vs PostgreSQL: A comparative study on performance aspects. *GeoInformatica*, 25(2):243–268, April 2021.

[125] TerminusDB. TerminusDB vs Neo4j - Graph Database Performance Benchmark. https://web.archive.org/web/20241209054258/https://terminusdb.

com/blog/graph-database-performance-benchmark/, December 2024. Accessed: 2025-06-22.

[126] Robert Feldt and Ana Magazinius. *Validity Threats in Empirical Software Engineering Research - An Initial Survey.* January 2010.

[127] Dinis Bessa de Sousa. Towards versioning profiles through time: A database benchmark – Experimental Replication Package. Zenodo, June 2025. Available at: https://zenodo.org/records/15757822.

[128] Dinis Bessa de Sousa. Wrong path leads to error on size function · Issue #2175 · terminusdb/terminusdb. https://github.com/terminusdb/terminusdb/issues/2175. Accessed: 2025-06-07.

[129] Philippe Höij. Issue/2175 path missing delimiter by hoijnet · Pull Request #2201 · terminusdb/terminusdb. https://github.com/terminusdb/terminusdb/pull/2201. Accessed: 2025-06-13.

[130] Ling Qian, Zhiguo Luo, Yujian Du, and Leitao Guo. Cloud Computing: An Overview. In Martin Gilje Jaatun, Gansen Zhao, and Chunming Rong, editors, *Cloud Computing*, pages 626–631, Berlin, Heidelberg, 2009. Springer.

[131] Ronald Ojino. User's profile ontology-based semantic model for personalized hotel room recommendation in the web of things: Student research abstract. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, SAC '19, pages 2314–2316, New York, NY, USA, April 2019. Association for Computing Machinery.