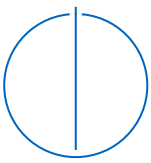
SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

GDPR Metadata Indexing Optimization

Shrief Hussien



SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

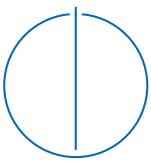
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

GDPR Metadata Indexing Optimization

Optimierung der DSGVO-Metadatenindexierung

Author:	Shrief Hussien
Examiner:	Prof. Pramod Bhatotia
Supervisor:	Dr. Dimitrios Stavrakakis
Submission Date:	30.09.2025



I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, 30.09.2025 Shrief Hussien

Acknowledgments

I would like to thank my supervisor, Dr. Dimitrios Stavrakakis, for his guidance and support throughout this work. His feedback and advice were valuable in shaping the direction of the research and in bringing this thesis to completion.

I would also like to thank my parents for their unwavering support throughout this journey. They are the reason behind my success and the person I am today, and I am deeply grateful for their encouragement and love.

Abstract

The General Data Protection Regulation (GDPR) requires operations such as retrieving all records for a user, grouping data by purpose, and deleting expired records, which conventional key–value stores can not serve efficiently, as they are optimized only for primary-key access. This thesis presents an extensible indexing layer that augments key–value stores with specialized structures for GDPR workloads. A skip list supports retention-based deletion, a B+ tree enables subject-level queries, and an inverted index handles purpose-based lookups. Together, these indices eliminate the need for costly full scans and ensure that queries and updates remain efficient under concurrent access. Using YCSB workloads, the evaluation demonstrates that each index structure is best suited to a distinct class of GDPR queries, balancing between throughput, scalability, and memory usage. The work contributes both the design and implementation of the indexing layer, as well as an experimental study that identifies which structures best support specific compliance tasks. The source code is available to support reproducibility and further research.[1]

[1] Link to repository: `https://github.com/sherifhussien/gdpr-index`

Contents

1 Introduction

Modern data-intensive systems, such as those in healthcare, digital advertising, and the training of large language models (LLMs), process enormous amounts of personal data. The GDPR imposes strict requirements regarding how this data is collected, stored, accessed, and deleted [Reg16]. It strengthens the rights of European Union (EU) citizens by granting them rights such as the right of access, rectification, erasure, objection, and data portability. Non-compliance carries significant financial risk, with penalties of up to €20 million or 4% of the global annual revenue. For example, Google was fined €50 million in 2019 for failing to obtain legitimate user consent for personalized ads [Sat19], and British Airways faced a £184 million fine the same year for insufficient protection of customer data [Lun19]. Accordingly, systems are required to maintain rich metadata about users, declared purposes, sharing policies, and retention durations. The volume of the metadata grows rapidly and must remain searchable with low latency, even when many clients access the system simultaneously. Addressing these challenges is crucial not only for complying with regulatory requirements but also for ensuring that systems continue to operate efficiently at scale.

The GDPR introduces workloads that depend heavily on metadata, which differ from those that conventional storage systems are designed to handle [Sha+19b]. Compliance requires queries such as retrieving all records for a user, grouping data by specific purpose, or deleting records once their retention period has ended. To meet these needs, relational databases can offer more substantial support because secondary indices accelerate metadata lookups. Key–value stores, by contrast, are optimized only for primary-key access [Mic], which means GDPR queries often degrade into costly full scans. This gap highlights the need for secondary indexing techniques that bring metadata-aware querying to key-value stores while preserving their performance and scalability.

Existing work on GDPR compliance has mainly focused on retrofitting general-purpose storage systems with additional features. Shah et al. [Sha+19a] retrofit Redis with compliance mechanisms and show that strict real-time enforcement reduces throughput by up to 20×, indicating the performance cost of synchronous per-request logging. Shastri et al. [Sha+19b] translate GDPR articles into database workloads and show that Redis, PostgreSQL, and a commercial database system perform poorly on these workloads, with performance degrading and scalability declining as the volume

of personal data increases. Agarwal et al. [Aga+21] present GDPRizer, a tool that retrieves all data linked to an individual from legacy relational databases, reducing manual effort but limited by its reliance on schema annotations and application-specific heuristics. Collectively, these studies show that current systems can provide partial compliance but suffer from severe performance and scalability issues, and they lack native support for core GDPR tasks such as retention-based deletion, subject-level data retrieval, and purpose-based grouping of records.

While prior studies highlight the challenges of achieving GDPR compliance, they primarily focus on retrofitting general-purpose key–value stores, such as Redis, with compliance features. These solutions do not overcome the central limitation that key–value stores are optimized for primary-key access and have no native support for secondary lookups. As a result, queries that are essential for compliance, such as retrieving all records for a user, grouping by purpose, or deleting data based on retention time, often fall back to full scans, which degrade both performance and scalability. Shastri et al. [Sha+19b] show that PostgreSQL performs much better than Redis on GDPR workloads because its secondary indices speed up metadata-based queries. Another challenge is that GDPR metadata is diverse. Attributes such as user identifiers, processing purposes, and retention times differ in structure and access patterns, so a single index type cannot serve all needs efficiently.

To this end, we propose an indexing layer that maintains multiple specialized indices, allowing users to enable those most relevant to their workloads. This thesis strives to answer the following critical question: *how can we design index structures for GDPR metadata that provide low-latency queries, utilize memory efficiently, and scale reliably as demand increases, while remaining independent of the underlying key–value store*.

The system is structured around an Index Manager that provides a standard interface to clients and coordinates queries across the underlying index structures. It selects the correct index based on the query type and returns the result, while hiding low-level synchronization details from higher layers. The design of the Index Manager is generic and not limited to a fixed set of indices. Any metadata attribute can be supported by adding an appropriate index through the same interface. In this thesis, three indices were implemented: a *Retention Index* based on a lock-free skip list, a *Subject Index* implemented as a B+ tree with lock coupling for writers and shared locks for readers, and a *Purpose Index* implemented as a sharded inverted index with fine-grained bucket-level locking to reduce contention. The framework is extensible, meaning further index types can be added without requiring changes to the higher layers. All three structures were implemented from scratch in C++ to provide complete control over memory management and synchronization.

In summary, this thesis demonstrates that incorporating a metadata-aware indexing layer on top of a key-value store enables both practical and efficient GDPR queries. The

system is built around an Index Manager that manages several specialized indices, each suited to a different type of metadata. We evaluated the three index structures and conducted experiments using the Yahoo! Cloud Serving Benchmark (YCSB) workloads, demonstrating that these indices enhance throughput while maintaining manageable memory usage under concurrent access. The main contributions of this work are the design and implementation of an extensible indexing layer for key-value stores, as well as an experimental study that evaluates skip lists, B+ trees, and inverted indexes, identifying which structure best supports which GDPR-related queries.

2 Background

2.1 GDPR Overview

The GDPR, which came into Effect in the EU on 25th May 2018, was enacted to address the growing misuse of personal data [Reg16]. It governs the entire lifecycle of the personal data from storage and collection to processing and eventual deletion.

The regulation not only strengthens the rights of EU citizens, granting them rights such as the right of access, rectification, erasure, objection, and data portability, However, it also imposes strict obligations on organizations that process personal data. These obligations include obtaining explicit consent before processing, reporting data breaches within 72 hours of detection, and maintaining detailed records of processing activities.

Non-compliance with the GDPR may result in substantial financial penalties of up to €20 million or 4% of global annual revenue, whichever is higher. An example of this was in January 2019, when Google was issued a €50 million fine because it failed to obtain legitimate consent from users for personalized ads [Sat19]. Similarly, British Airways was fined £184 million in July 2019 for failing to take sufficient measures to protect customer data [Lun19].

The GDPR is composed of 99 articles, which define its binding legal requirements, and 173 recitals, which offer explanation and interpretative context. While the articles set out the rules that organizations must comply with, the recitals clarify their purpose and intended scope.

Broadly, the articles can be grouped into five categories: the principles of data processing, the rights of data subjects, the obligations of data controllers and processors, the powers of supervisory authorities, and the enforcement measures. The first category establishes the general principles of lawful and fair data processing, while the second details the rights of individuals to access, rectify, erase, or port their personal data. The third specifies the responsibilities of organizations that process personal information, and the fourth defines the role of supervisory authorities in monitoring compliance across EU Member States. Finally, the fifth category provides the accountability and enforcement framework, including mechanisms for dispute resolution and the imposition of sanctions.

2.2 Indexing Techniques

Predicate queries refer to queries that apply a condition to select only certain record from the database. Predicate queries that attempt to extract only a subset of the records in a table can be achieved by scanning the entire table, examining each record, and returning only those that meet the query condition. When the table is sorted on the attribute being referenced by the query, however, a full scan is unnecessary because qualifying records are typically located much more rapidly, in logarithmic rather than linear Time.

A more efficient alternative to scanning is to use specialized index structures that enable direct referencing of the qualifying records. Indexes can be defined over one or more attributes, and when they are built over non-primary-key attributes, they are referred to as secondary indexes. A typical secondary index contains an entry for each distinct value of the indexed attribute. Each entry may be thought of as a key-value pair, where the key is an attribute value and the value is a list of base record references.

Secondary indexes can be implemented through several mechanisms. The most basic approach is to maintain an inverted list as a system table, where the indexed attribute serves as the key and the associated record identifiers are stored in an auxiliary column [Dsi+17]. Such a structure allows efficient lookups for specific attribute values. More advanced implementations utilize tree-based data structures, in which internal nodes direct searches toward leaves that contain attribute values and their corresponding pointers. Both inverted lists and tree-based indexes support point queries, which retrieve records matching a single attribute value, as well as range queries over intervals of values[Dsi+17]. For highly selective point queries, hash-based indexes are often preferred, since they map attribute values directly to record identifiers and provide high-speed access.

In some cases, an index may be sufficient to answer a query without consulting the base table, particularly when the index stores the required attributes. However, the effectiveness of indexes depends heavily on the amount of records satisfying the predicate. When the qualifying set is small, indexes provide rapid identification and access to the relevant tuples. When a significant fraction of records satisfy the predicate, a sequential scan may be more efficient, as it allows batch reads and avoids the overhead of numerous index traversals [Dsi+17].

2.2.1 Key–Value Stores

Key–value (KV) stores are the most basic form of NoSQL database. They expose a simple interface with operations such as `put`, `get`, and `delete`, where each entry is an opaque value associated with a unique key. Because the system does not impose a

schema and data can be retrieved directly by key, KV stores achieve extremely low-latency access and are easy to scale horizontally. This simplicity makes them a common choice for latency-sensitive tasks such as caching, session management, and large-scale data pipelines.

Different KV systems make different trade-offs between latency, durability, and scalability. Redis is an in-memory KV store designed for sub-millisecond access times and heavy concurrent workloads. It is widely used as a cache or transient data store, offering high throughput but requiring additional persistence mechanisms for durability. RocksDB, in contrast, is a persistent, embedded KV store built on log-structured merge (LSM) trees. It is optimized for write-heavy workloads and large storage volumes, relying on sequential writes and background compaction to maintain performance. Together, Redis and RocksDB illustrate a key trade-off in KV design, where Redis prioritizes speed with in-memory storage. At the same Time, RocksDB focuses on durability and scalability with disk-backed structures.

The main limitation of KV stores is their restricted query model. They handle primary-key lookups efficiently but lack native support for secondary attributes. Queries such as retrieving all records for a user, filtering by timestamp, or grouping by declared purpose require scanning the entire dataset. This is particularly problematic for GDPR workloads, which are dominated by metadata queries involving attributes such as subject identifiers, purposes, objection flags, and expiration times. Relational databases support such queries through secondary indexes but face higher complexity and maintenance costs, which limit scalability under high update rates. KV stores trade query flexibility for speed and simplicity. To close this gap, new secondary indexing techniques are needed that extend KV stores with efficient metadata lookups while preserving their scalability and predictable performance.

2.3 Sharding

Sharding is a database architecture pattern that partitions large datasets across multiple independent database instances, known as shards. Each shard stores a distinct subset of the data, typically determined by a sharding key such as user ID, geographic region, or another attribute that ensures even distribution of records. This design enables horizontal scaling, allowing a system to accommodate increasing workloads by adding shards rather than upgrading a single, monolithic database.

The primary motivation for adopting sharding is to improve performance and scalability. By distributing both data and query load across multiple servers, sharding mitigates contention and eliminates the bottlenecks inherent in centralized storage. It also enhances fault tolerance, since the failure of a single shard does not necessarily

compromise the availability of the entire system.

However, implementing sharding introduces several challenges. Selecting an effective sharding key is critical: poor choices may lead to data skew and hotspots, where specific shards receive disproportionate traffic. Queries that span multiple shards (cross-shard queries) are inherently more complex and often incur additional latency. Furthermore, maintaining strong consistency and supporting distributed transactions across shards typically requires specialized coordination protocols, which may impose overhead or necessitate trade-offs in isolation guarantees.

Despite these challenges, sharding has become a fundamental technique in large-scale systems such as social networks, e-commerce platforms, and cloud services, where massive data volumes and high levels of concurrency exceed the capacity of single-node databases. Modern distributed databases, including MongoDB and Cassandra, offer built-in sharding mechanisms that automate partition management, query routing, and, in many cases, rebalancing across shards.

2.4 GDPR compliance for modern databases

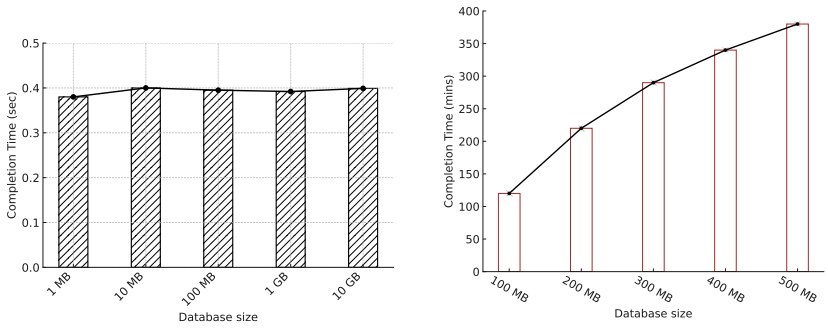
The GDPR introduces workloads that go beyond standard CRUD(Create, Read, Update, Delete) operations [Sha+19b]. Systems must handle queries on metadata, such as user IDs, processing purposes, objections, sharing information, and expiration times. For example, this includes retrieving all records for a user, filtering by purpose, or deleting expired data. These workloads differ from traditional ones that rely mainly on primary keys.

These requirements translate into several fundamental challenges for storage systems:

- **Secondary lookups.** Compliance queries typically target attributes such as user identifiers, purposes, or expiration times. Because traditional KV stores only index primary keys, they cannot serve these queries without expensive full scans.
- **Many-to-many mappings.** GDPR metadata involves mappings such as between purposes and records or between subjects and the third-party organizations that receive their data. Systems must manage these mappings efficiently for both queries and updates.
- **Retention enforcement.** Records must be deleted immediately once their retention period ends. This creates a stream of background deletions that must coexist with regular read, update, and insert workloads without degrading performance.
- **Concurrency at scale.** GDPR queries rarely run in isolation. They overlap with regular application traffic, increasing contention on shared data structures and putting pressure on transaction and synchronization mechanisms.

Together, these characteristics yield workloads that are metadata-driven, deletion-heavy, and highly concurrent, which stand in sharp contrast to the primary-key workloads that conventional key–value stores are designed to handle.

Effect of scale. Shastri et al. [Sha+19b] evaluated how modern databases behave under GDPRbench, a benchmark designed to capture GDPR-specific workloads, and compared it to traditional YCSB benchmarks. While YCSB targets single-record lookups by primary key, GDPRbench issues metadata-based queries that can return many records, such as all data linked to a user. These heavier queries achieve lower throughput, so GDPRbench is run with smaller datasets and fewer operations than YCSB. Their study highlights a clear performance gap between key–value and relational systems under these conditions.

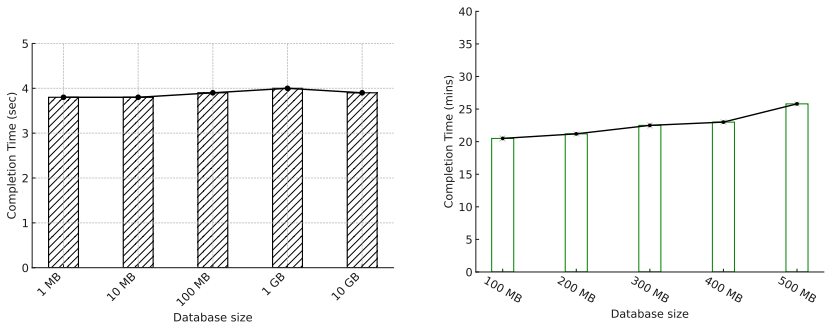


(a) YCSB Workload-C (b) GDPRbench Customer

Figure 2.1: Time taken by Redis to complete 10K operations as database size grows [Sha+19b].

For Redis, completion time under YCSB workloads remains constant as the database grows (Figure 2.1a), since records are always accessed directly by primary key. Under GDPRbench, however, completion time increases almost linearly with database size (Figure 2.1b), because every query must scan a large share of the dataset. As the dataset grows, this scanning overhead becomes the dominant bottleneck, a limitation standard to key–value stores that lack secondary indexes.

In contrast, PostgreSQL performs better due to its native support for secondary indexes. As shown in Figure 2.2b, completion time still rises as the dataset grows, but at a slower rate than Redis, which is almost constant. Secondary indexes prevent full scans for metadata lookups; however, maintaining multiple indexes introduces significant overhead at large scales.



(a) YCSB Workload-C (b) GDPRbench Customer

Figure 2.2: Time taken by PostgreSQL to complete 10K operations as database size grows [Sha+19b].

2.5 Motivation

These findings show a fundamental limitation of existing systems. Relational databases utilize secondary indexes to accelerate metadata queries, whereas key–value stores are limited to primary-key access. As a result, operations such as retrieving all records for a user, filtering by purpose, or deleting expired data run efficiently in relational systems but degrade into costly full scans in key–value stores. This contrast exposes a gap. Relational systems support compliance queries through indexing, but key–value stores, despite their scalability and simplicity, still lack this capability. The challenge is to design lightweight secondary indexing for key–value stores that enables low-latency metadata queries, minimizes memory overhead, and preserves scalability.

3 Overview

At a high level, this thesis proposes a metadata-aware indexing layer that complements a key-value store. While the underlying storage system (e.g., Redis, RocksDB) stores and loads records through primary keys, the indexing layer complements it by introducing secondary structures to store GDPR-related metadata, such as subjects, purposes, and retention periods. The design eliminates the need for full scans to resolve compliance queries. Operations such as fetching all records for a single user, retrieving data for a specific purpose, or identifying records beyond their retention period can be answered directly by these indices.

The high–level design is motivated by GDPR compliance requirements. For example, the right of access requires retrieval of all data linked to a subject or processing purpose, while the right to erasure requires complete and timely deletion of a subject's records. Supporting these obligations at scale requires indexing mechanisms that can efficiently answer metadata–driven queries while preserving the scalability and simplicity of the underlying key–value store. Key design goals include:

Low-latency metadata queries: Support queries such as "find all records for User X" or "list all records collected for Purpose Y" to return results in sub-second time, even over millions of records. To achieve these tasks, the design avoids full dataset scans by maintaining direct pointers to relevant records.

Throughput at scale: Ensuring the system can sustain high operation rates as the number of client requests and dataset size grow. The indexing structures are designed for parallel access, using techniques such as lock-free synchronization or fine-grained locking to minimize contention. This enables inserts, updates, and deletions from multiple clients to proceed simultaneously while maintaining stable performance under heavy loads.

Compliance support: Provide mechanisms that directly satisfy GDPR obligations. Insert operations must register new records across all relevant indices, subject access requests must return complete and current results, and deletion operations (e.g., the right to be forgotten) must remove all traces of a user's records efficiently and consistently.

3.1 Architecture and Components

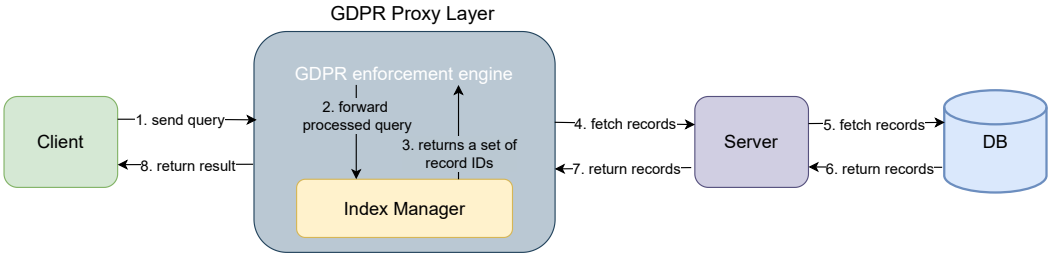


Figure 3.1: System overview highlighting the GDPR Proxy Layer, which coordinates client queries by invoking the Index Manager to resolve metadata lookups and retrieve the corresponding records from the database.

The system is organized around a dedicated *Indexing Layer* that augments an underlying key–value store. The store itself (e.g., Redis, RocksDB) is responsible for persisting the actual records and supporting primary-key access. At the same time, the indexing layer maintains secondary data structures over GDPR-related metadata, including user identifiers, declared purposes, and expiration times. The separation enables the system to retain the scalability of a key-value store while allowing for efficient metadata queries required for compliance.

Figure 3.1 shows the high-level architecture. Clients interact with the system through a GDPR proxy layer that processes queries and routes them through an *Index Manager*. The Index Manager transforms metadata-driven queries into index lookups, returning a set of record identifiers that match the request. These identifiers are then used to fetch the actual records from the underlying database through the server component. By handling client queries in this way, the proxy ensures that GDPR-specific operations are executed efficiently, without requiring clients to interact directly with the indices.

At the core of the proxy layer lies the *Index Manager*, illustrated in Figure 3.2. The Index Manager consists of a *Query Coordinator*, which receives client requests and determines which index or combination of indices to access, and an *Index Interface*, which defines a standard API implemented by the three specialized structures (B^+ tree, inverted index, and skip list). The modular design ensures that the complexity of maintaining multiple specialized indices remains hidden from higher layers, while enabling efficient query resolution. It also keeps the system extensible, since new index types can be added behind the same interface without changing how higher layers issue queries.

The system currently supports three specialized index structures, each tailored to a

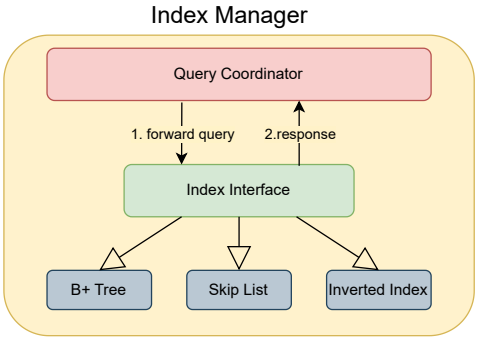


Figure 3.2: Index Manager core components showing the Query Coordinator, Index Interface, and specialized indices (B+ Tree, Skip List, Inverted Index).

distinct GDPR workload:

- **Retention Index.** Tracks expiration times using a *skip list* sorted by timestamp. The skip list supports concurrent inserts and ordered traversal, allowing the system to quickly identify records due for removal and perform bulk deletions when many records expire simultaneously.
- **Subject Index.** Maps subject identifiers (e.g., user IDs) to their associated records. Implemented as a *B+ tree*, it supports efficient point queries and range traversals, making it possible to retrieve all records for a given user with logarithmic lookup time.
- **Purpose Index.** Organizes records by their declared processing purpose. Implemented as an *inverted index*, it maps each purpose value to a list of record identifiers, enabling efficient grouping and retrieval of all records linked to a given purpose.

Together, these indices allow inserts, deletes, and lookups by user, purpose, or expiration time to be executed without scanning the entire key-value store. The Index Manager ensures that all operations consistently update the relevant indices, providing accurate query results even under high concurrency.

3.2 System Workflow

The workflow of the system is illustrated in Figure 3.1. The numbered arrows indicate the main steps executed during an operation:

1. **Client request.** A client submits a query or update request to the GDPR proxy layer.
2. **Index lookup.** The GDPR enforcement engine forwards the processed query to the Index Manager.
3. **Index resolution.** The Index Manager consults the corresponding index structure and returns a set of record identifiers matching the metadata condition (e.g., all records for a subject, all records under a purpose, or records past expiration.
4. **Record fetch request.** The GDPR enforcement engine passes the record identifiers to the server component.
5. **Database access.** The server queries the underlying database.
6. **Database response.** The database returns the requested records to the server.
7. **Server response.** The server returns the retrieved records to the GDPR proxy layer.
8. **Response to client.** The GDPR proxy layer returns the final results to the client.

The workflow applies consistently across inserts, queries, and deletions. For inserts, the Index Manager updates all relevant indices before acknowledging completion. For queries, it resolves metadata lookups through the indices, avoiding full scans of the base store. For deletions, it ensures that all related entries are consistently removed across indices, ensuring no stale references remain.

3.3 System Goals

The system separates the concerns of durable storage and metadata indexing. The underlying KV store remains responsible for persistence and primary-key access, while the indexing layer maintains secondary structures to serve GDPR queries.

Its main goals include:

- **Scalability:** Each index structure is designed for concurrent access, enabling the system to maintain high throughput as the client load increases. This ensures responsiveness even when multiple inserts, lookups, and deletions co-occur.
- **Consistency:** Metadata updates are propagated to all relevant indices before an operation completes. As a result, queries always reflect a consistent and up-to-date view of the data, which is crucial for correctness under GDPR workloads.

- **Extensibility:** The Index Manager hides the complexity of managing multiple specialized indices behind a standard interface. The modularity enables the integration of new index types without requiring changes to higher layers or client applications.

4 Design

4.1 Index Structures and Rationale

This section introduces the three index structures used in the system: a B+ Tree for user identifiers (Subject Index), an inverted index for purposes (Purpose Index), and a skip list for expiration times (Retention Index). Each index is chosen to support a specific type of GDPR query efficiently.

4.1.1 Retention Index

The Retention Index is implemented as a *skip list*, a probabilistic ordered data structure that achieves logarithmic-time search, insert, and delete operations by maintaining multiple levels of forward pointers. In this system, the skip list is keyed by expiration time (Time-To-Live (TTL)) and functions as a sorted timeline of records. Each node corresponds to a record or group of records that share the same expiration timestamp. This design allows the earliest record due to expire to be accessed in constant time by reading the head of the list. Range queries, such as retrieving all records expiring between T_1 and T_2, are supported by traversing the list from the first node $\geq T_1$ until T_2 is reached.

Design rationale. The skip list was chosen for the retention index because of its simplicity and strong concurrency properties. Unlike balanced trees, skip lists do not require global rebalancing operations such as rotations or splits, which makes them more suitable for workloads with frequent insertions and deletions. In in-memory settings, skip lists often scale better under concurrency due to their lighter structural constraints (See Chapter 5). A lock-free skip list further enhances this property by allowing multiple threads to insert new expiration entries or remove expired records without serializing on a global lock. This ensures that the time-sensitive task of expiring records does not block other operations on the index.

As shown in Figure 4.1, the skip list design enables efficient expiry management: the earliest record due to expire is constantly accessible at the head of the list. At the same time, higher levels allow searches across larger ranges to complete in logarithmic time.

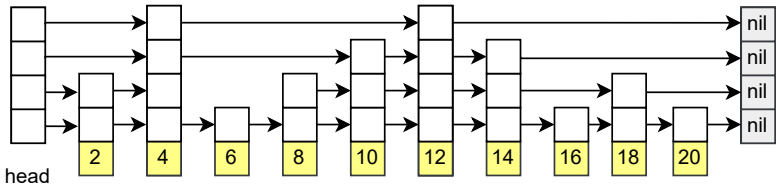


Figure 4.1: Skip list structure used for the Retention Index. The bottom row contains all records in order of expiration time, while higher levels act as express lanes that accelerate traversal.

4.1.2 Subject Index

The Subject Index is implemented as a B+ Tree keyed by user identifier (or more generally, the data subject ID). Each leaf entry stores an array of record references, which clusters all of a user's records together for efficient scans. The B+ Tree is a balanced search structure in which data entries are stored exclusively in the leaf nodes, and these leaves are linked in sorted order to support sequential traversal. This organization enables both efficient point lookups of a single user's records and ordered iteration over consecutive users. The Subject Index, therefore, realizes the mapping *UserID* $\rightarrow$ *[RecordIDs]* in a form that scales well with large datasets.

The B+ Tree provides logarithmic complexity for search, insert, and delete operations, along with strong support for range queries. Although range scans on user identifiers are less common in GDPR workloads, point queries are essential, and ordered traversal of records remains a valuable capability. The linked leaf nodes make it straightforward to traverse from one user's data to the next.

Design rationale. The B+ Tree is a well-established index structure in databases and scales effectively when the number of users grows large. Its high fan-out minimizes I/O when stored on disk, while still being efficient in in-memory contexts. We selected a B+ Tree over alternatives such as hash indexes because it combines efficient point lookups with the ability to perform range queries and ordered traversals. This balance is valuable not only for current workloads but also for potential extensions, such as prefix-based queries, if user identifiers have an internal structure. Furthermore, the B+ Tree integrates well with concurrency control mechanisms, such as latch coupling or optimistic locking, which are discussed in Chapter 5.

4.1.3 Purpose Index

The Purpose Index is implemented as an inverted index. It maps each purpose value (e.g., *Marketing*, *Research*, *Fraud Detection*) to the set of records associated with that value.

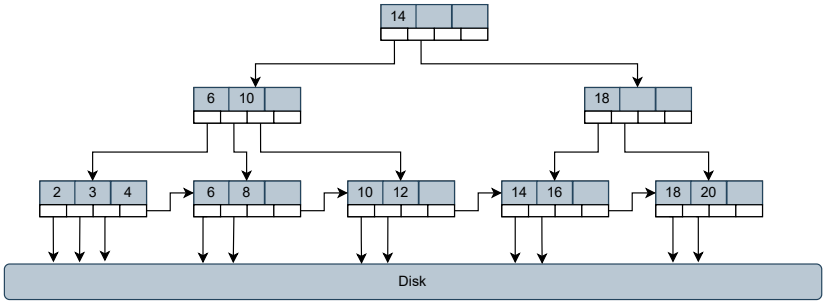


Figure 4.2: B+ Tree structure used for the Subject Index. The leaf nodes store user identifiers in sorted order, and internal nodes guide efficient lookups.

The structure resembles a search engine index, where the "term" is the purpose and the "documents" are the records. Implementation-wise, the index maintains a dictionary of purposes, with each entry pointing to an list of record identifiers. Identifiers lists are kept in sorted order to support efficient merging and deletions.

Inverted indexes are well-suited for exact-match queries, such as retrieving all records with *purpose = Marketing*. They can also be extended to support boolean combinations of attributes, although this prototype focuses on single-purpose lookups. Because purposes are categorical values, range queries are not applicable. Prefix queries over purpose names are possible by scanning the dictionary. Still, such operations are inefficient with a hash-based dictionary and are not a primary target of this design.

Design rationale. The inverted index is the most direct way to represent the many-to-many relationship between purposes and records. A single purpose may correspond to thousands of records, and storing these as duplicate keys in a tree would waste space and complicate deletions. By maintaining one identifiers list per purpose, lookups are compact, and deletions can be performed at the list level. Although updates to identifiers lists may be costly in an array-based implementations, this can be mitigated using sharding techniques (see Section 4.3 and Chapter 5).

4.2 Trade-offs and Comparisons

Each index structure offers distinct strengths and trade-offs, so selecting the right one for each metadata attribute was essential.

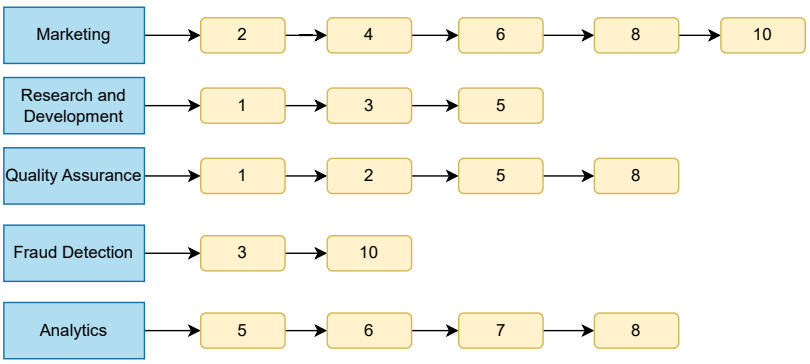


Figure 4.3: Inverted index for the Purpose Index, mapping each purpose to the set of associated record identifiers.

4.2.1 Skip List vs. B+ Tree

Both skip lists and B+ trees maintain sorted order and achieve logarithmic search complexity. Skip lists are lightweight, in-memory structures with probabilistic balancing. They are simple to implement, avoid costly node splits, and can be made highly concurrent using lock-free Compare-and-Swap (CAS) operations. The trade-off is higher memory overhead from the multi-level forward links maintained by each node.

In contrast, B+ trees store many keys within each node, providing better cache locality than skip lists. Their block structure also maps naturally onto disk pages, which makes them efficient when the index must be stored persistently. They are, however, more complex to implement in a lock-free manner and typically rely on locking protocols. While single-threaded performance is comparable, skip lists often scale better under high concurrency because they avoid contention on structural locks.

In our design, this motivated the use of a skip list for the *TTL expiration index*, which must support highly parallel operations such as continuous traversal to detect expired records and bulk deletions when many records expire simultaneously. In contrast, the *user index* was implemented as a B+ tree, as point lookups dominate its workload. Both trade-offs were considered acceptable, since the absolute memory footprint of pointers in the skip list and unused slots in B+ tree nodes is small compared to the size of the indexed records themselves.

4.2.2 Skip List vs. Inverted Index

Skip lists and inverted indexes are suited to distinct access patterns; skip lists support ordered range scans, while inverted indexes excel at bulk retrieval on categorical values.

While it is possible to organize identifiers lists within an inverted index using a skip list (or any ordered structure) to accelerate deletions and point removals, our workloads rarely require such fine-grained operations. Instead, queries typically consume entire lists in a single step. For this access pattern, a simple dynamic array representation is both sufficient and efficient. To minimize overhead, identifiers lists are maintained as arrays combined with shard-level locking, thereby avoiding the additional complexity of per-term skip lists, which would provide little benefit under our access patterns.

4.2.3 B+ Tree vs. Inverted Index

B+ trees and inverted indexes serve different purposes. B+ trees are well suited for ordered attributes such as user identifiers or timestamps, where both point lookups and range queries are common. Inverted indexes, by contrast, are optimized for categorical attributes, such as processing purposes or objection flags, where the workload requires grouping and bulk retrieval rather than ordered scans.

Using a B+ tree for categorical values would waste space and introduce unnecessary balancing overhead, while using an inverted index for ordered attributes would make range queries inefficient. Accordingly, our design uses B+ trees for attributes that benefit from order-preserving indexes and inverted indexes for categorical metadata.

Summary. The design leverages skip lists where concurrency and sorted queues are needed, B+ trees where balanced indexing and range capability matter, and inverted indexes where grouping by a value is essential. This combination efficiently covers our targeted GDPR use cases.

4.3 Sharding

In this thesis, sharding is applied to the problem of scaling inverted indexes for handling high volumes of concurrent requests. Unlike many distributed databases that dynamically adjust the number of shards, the architecture considered here employs a fixed number of shards, established at system design time. This choice simplifies shard management and avoids the overhead of repartitioning data, but also places greater importance on selecting an effective sharding strategy that ensures balanced load distribution across the available shards. Within this constraint, the objective is to exploit parallelism across shards to reduce query latency and improve throughput, while addressing challenges related to cross-shard query execution and consistency of index updates.

5 Implementation

5.1 Retention Index

The *retention index* is implemented as a *lock-free skip list* to support highly concurrent access and to enforce data retention policies based on TTL values. This structure maintains all records in a sorted order by their expiration timestamps, allowing for efficient retrieval and timely removal of the oldest entries. In contrast to alternatives such as B+ trees, skip lists provide simpler concurrency control and do not require structural rebalancing, making them well-suited for highly dynamic workloads.

Each skip list node represents a single expiration timestamp and contains an array of *forward pointers* (`next[]`) that form its tower across the levels. Alongside the timestamp key, the Node maintains a collection of record identifiers that are all scheduled to expire at this time. This collection is stored as a `std::unordered_set<V>` and is protected by a `std::shared_mutex`, which allows multiple threads to read the values concurrently while still permitting safe exclusive updates. By isolating value updates within the Node and maintaining lock-free structural pointers, the design enables concurrent operations on the skip list without introducing global locks.

We implemented a lock-free skip list by applying the Harris algorithm for non-blocking linked lists [Har01] to each level of the structure. All structural modifications use atomic CAS operations, allowing threads to traverse and update the list without mutexes. Nodes are first marked as logically deleted by tagging their pointers, and are later physically unlinked by other threads during traversal. The current implementation does not reclaim memory for removed nodes. Once a node has been marked and unlinked, it remains allocated. This approach avoids the complexity of integrating a safe memory reclamation scheme, such as hazard pointers [Mic02], and eliminates the risk of use-after-free while focusing on correctness.

Node Structure. Each `Node <K, V>` is allocated as a single contiguous memory block and contains the following fields:

- `K key`: the expiration timestamp used as the sort key.
- `std::unordered_set<V> values`: the set of record identifiers that expire at this timestamp.

- `std::shared_mutex valueMutex`: a fine-grained lock used only for safe concurrent modifications to the `values` set.
- `int level`: the number of levels this node spans.
- `std::atomic<Node*> next[level]`: an array of atomic forward pointers, one for each level from 0 to `level-1`.

Pointer Marking Utilities. Algorithm 1 shows the low-level Pointer tagging utilities used in the skip list. The Least Significant Bit (LSB) of each `next` Pointer is reserved as a *mark bit*. Setting this bit with `get_marked_ref` tags the Pointer to indicate that the referenced Node has been logically deleted. `get_unmarked_ref` clears the bit to recover the original pointer value. During traversal, threads use these helpers to detect and skip marked nodes.

Algorithm 1: Pointer marking operations used for logical deletion

Function `IsMarked` (ptr)
  return $(ptr \ \& \ 1) \neq 0$;

Function `GetMarkedRef` (ptr)
  return $ptr \mid 1$;

Function `GetUnmarkedRef` (ptr)
  return $ptr \ \& \ \sim 1$;

Atomic Updates. Algorithm 2 illustrates the CAS operation used throughout the skip list to modify shared pointers atomically. Each insertion or deletion step first reads the expected current pointer value and then attempts to replace it with a new value using CAS. If another thread has modified the Pointer in the meantime, the operation fails and is retried. Because CAS is an atomic hardware instruction, it guarantees that only one thread can succeed, which is essential for maintaining consistency without locks. In deletion, the new value passed to CAS is often a pointer tagged with the mark bit (see Algorithm 1), which signals that the Node has been logically deleted.

Algorithm 2: Atomic CAS operation used for structural updates

Function `CAS` $(addr, expected, new)$
  if $*addr = expected$ **then**
    $*addr \leftarrow new$;
    return `true`;
  return `false`;

Insertion Algorithm. To insert a new record with expiration time T_{new}, the thread first selects a random height for the latest Node using a geometric distribution with success probability $p = 0.5$. This is implemented by repeatedly sampling a thread-local Bernoulli generator until it returns zero, incrementing the level each time, up to a fixed `MAX_LEVEL`. Using a thread-local generator avoids contention on shared state and preserves the expected logarithmic height distribution of the skip list. After determining the node height, the thread traverses the list from the top level down to level 0, collecting the immediate predecessor nodes for each level in an array $\texttt{pred}[i]$. Once the insertion position is identified, the new Node's forward pointers are initialized to point to the corresponding successors, and atomic CAS operations are used to link the Node into the list. The insertion starts at level 0, which is the linearization point, and then links the higher levels.

Deletion Algorithm. Removing a node requires special care to maintain lock-free correctness. We use a lazy deletion strategy in which a node with key T_x is first logically deleted by atomically marking its lowest-level `next[0]` pointer with a mark bit. Once marked, any concurrent traversal that encounters the Node treats it as removed and may facilitate its physical unlinking by updating predecessor pointers via atomic CAS so that they bypass the Node. This two-phase process, which first marks a node and then unlinks it, prevents threads from accessing freed memory while preserving linearizability. We do not reclaim memory during execution. Unlinked nodes remain allocated until the process is torn down, which eliminates use-after-free risks in this prototype but increases memory usage as deletions accumulate.

Search Algorithm. Searching for a key T_x begins at the top level of the skip list and proceeds level by level down to level 0. At each level, the algorithm follows forward pointers until it reaches a node whose key is greater than or equal to T_x. Then, it descends one level and continues. During traversal, any nodes encountered with marked pointers are treated as logically deleted and are skipped. When possible, the search also helps by unlinking marked nodes using CAS before continuing. By cleaning up marked nodes during normal traversal, the structure remains well-formed without relying on separate maintenance passes. If a node with key T_x is found and is unmarked, its associated value set is returned. If no such node exists, the search terminates with a miss. Because search operations only read pointers and do not modify shared state, they are wait-free and complete without blocking other threads.

The skip list is designed to support high levels of parallelism without global locks. Search operations are *wait-free*, meaning they always complete in a finite number of steps

and never block other threads. Insertions and deletions are *lock-free*, ensuring overall progress even if individual operations must retry due to contention. These properties allow multiple threads to insert new expiration times or remove expired nodes in parallel. Because operations only modify local pointers and never acquire coarse-grained locks, contention remains low even at high thread counts. This concurrency model enables background expiration tasks and client-driven operations to execute in parallel without interfering with each other.

5.2 Subject Index

The *subject index* is implemented as a *concurrent B+-tree* to support efficient lookups and range scans of records grouped by user identifiers. Unlike the retention index, which must handle frequent deletions, the subject index serves mostly read-heavy workloads over a relatively stable set of keys. A B+-tree fits this pattern well because of its wide branching factor, logarithmic search depth, and contiguous node layout, which improves cache locality and reduces traversal cost. Leaf nodes are connected via right-sibling pointers, which support efficient range scans by enabling sequential access across adjacent user records.

Node Structure. Each `Node <K, V>` is allocated as a single contiguous memory block and contains the following fields:

- `std::vector<K> keys`: the sorted separator keys stored in the node.
- `std::vector<std::unordered_set<V>> values`: the record identifier sets (only in leaf nodes).
- `std::vector<std::shared_ptr<Node>> children`: the child pointers (only in internal nodes).
- `std::shared_ptr<Node> next`: a right-sibling pointer linking leaf nodes for range scans.
- `std::shared_mutex node_mutex`: a fine-grained lock used for safe concurrent modifications.

Internal nodes contain k keys and $k+1$ child pointers, while leaf nodes hold between $\lceil m/2 \rceil$ and m keys, where m is the node order. Keys are kept sorted within each Node to support binary search during lookups.

Lock Coupling. Lock coupling is used to maintain the tree structure's stability during concurrent traversals. A thread descends from the root to the target leaf, acquiring a lock on a child before releasing its parent to maintain a stable path. Readers use shared locks and writers use exclusive locks, which allows many readers to move through the tree in parallel while ensuring that writers update a consistent path. Because each thread holds at most two locks at a time (parent and child), lock hold times are short and contention stays low. Algorithm 3 shows this pattern for search traversal.

Algorithm 3: Lock-coupled Traversal

```
Function GetLeaf (node, key)
    lock_shared(node);
    while node is not a leaf do
        child ← getChild(node, key);
        lock_shared(child);
        unlock(node);
        node ← child;
    return node ;                    // leaf now locked
```

Lock Coupling Invariants. Lock coupling relies on a few key invariants to remain correct in the presence of concurrency. Locks are always acquired from the root downward, and no thread ever attempts to lock an ancestor after locking a descendant. A child must be locked before being released to their parent, ensuring a safe handoff during descent. Readers use only shared locks, and writers use only exclusive locks; no thread upgrades a lock in place. If a validation check fails (for example, if the root changes during traversal), the operation releases all held locks and restarts from the root. These rules prevent deadlocks, maintain the tree structure consistency for concurrent readers, and ensure that writers constantly update a stable path.

Insertion Algorithm. Insertions use a two-phase strategy that optimizes for concurrency in the typical case while ensuring safety during structural modifications. Threads' first attempt at an *optimistic insert* by descending the tree with minimal locking and acquiring an exclusive lock only on the target leaf. If the leaf has space, the key is inserted directly and the lock is released. If the leaf is full, the attempt is aborted and retried using a *pessimistic insert*. In the fallback path, the thread follows a lock-coupling approach, taking exclusive locks as it traverses the tree and releasing each parent only after its child is locked, thereby maintaining the structure's stability during modifications. When a node overflows, it is split by creating a new sibling, redistributing half of the keys, and inserting the promoted separator key into the parent. Splits may

propagate upward if parents also become full. This approach maintains the typical case's speed while ensuring correctness during structural changes.

Search Algorithm. Lookups follow a top-down traversal that utilizes shared locks to enable concurrent reads while ensuring the tree's structure remains unchanged during traversal. The search begins at the root, where a shared lock is acquired on the current Node and the root pointer is validated to detect concurrent structural changes. If the root has changed, the search restarts from the beginning. Traversal continues using lock coupling. At each internal Node, the thread identifies the appropriate child with `binary search`, acquires a shared lock on the child, and then releases the lock on the parent before descending further. Once a leaf is reached, its shared lock is held while locating the target key, and a copy of the associated value set is returned. This approach ensures that readers hold at most two shared locks at any time, allowing many lookups to run in parallel while writers acquire exclusive locks only on the nodes they modify.

Range Search. A range query `[start_key, end_key)` begins by descending to the leaf that may contain `start_key`, using the same lock-coupled traversal as in the search algorithm with shared locks for safe concurrent access. Once at the starting leaf, the thread locates the first key $\geq$ `start_key` using `lower_bound` and scans forward, collecting all key–value pairs until reaching `end_key`. If the end of the leaf is reached before `end_key`, the scan follows the `next` pointer to the right sibling, acquiring its lock before releasing the current one to maintain consistency. This approach keeps at most one leaf locked at a time during the scan, allowing range queries to run in parallel with other readers without blocking the rest of the tree.

The B+-tree is designed to support concurrent lookups and updates while preserving structural consistency. Readers traverse the tree using lock coupling with shared locks, so multiple threads can search in parallel without blocking each other. Writers use the same pattern with exclusive locks, holding at most two locks at a time as they descend and releasing each parent as soon as its child is locked. Because locks are fine-grained and short-lived, contention remains low even at high thread counts. This allows the B+-tree to handle frequent lookups and moderate updates in parallel, making it well-suited for the subject index, where predictable read performance under concurrency is essential.

5.3 Purpose Index

The *purpose index* is implemented as a *sharded inverted index* to support efficient lookups from purpose identifiers to the set of associated record IDs. Unlike the retention and subject indexes, which each map records to a single key attribute (expiration time or user ID) in a strictly ordered structure, the purpose index supports a many-to-many mapping where each record can appear under multiple purpose keys. This design reflects typical GDPR workloads, where a single record can be tagged with multiple purposes and queries often retrieve all records associated with a given purpose.

To achieve scalability under concurrent access, the inverted index is divided into a fixed number of *shards*. Each shard contains a disjoint subset of keys and is protected by its own reader–writer lock. This sharding strategy spreads operations across independent locking scopes, reducing contention when many threads perform lookups or updates in parallel. Within a shard, each key is mapped to a *bucket* containing the set of associated record IDs. Each bucket has its own fine-grained reader–writer lock, allowing concurrent reads even when updates occur on other keys in the same shard.

Bucket Structure. Each key maps to a `Bucket` object that contains:

- `std::unordered_set<V> values`: the set of record identifiers associated with the key.
- `std::shared_mutex mutex`: a fine-grained lock used to protect concurrent reads and updates to the `values` set.

Shard Structure. To distribute load and minimize lock contention, the index is partitioned into a fixed number of `NUM_SHARDS` shards, stored in `std::array<Shard, NUM_SHARDS>`. Keys are assigned to shards using a hash-based modulo operation (Algorithm 4), which distributes operations across independent lock scopes. Each shard contains:

- `std::unordered_map<K, std::shared_ptr<Bucket» index`: the mapping from keys to their corresponding buckets within this shard.
- `std::shared_mutex mutex`: a shard-level lock that synchronizes access to the `index`.

Algorithm 4: Shard selection by key hash

```
Function getShard (key)
    h ← hash(key);
    i ← h mod NUM_SHARDS;
    return shards[i];
```

Insertion Algorithm. To insert a `(key, value)` pair, the thread first acquires an exclusive lock on the shard and locates the corresponding bucket. If the bucket is missing, the thread upgrades to an exclusive shard lock to create it safely. Once the bucket is obtained, an exclusive lock is taken on the bucket, and the value is inserted into its set. This approach limits contention to the specific shard and bucket associated with the key, allowing other shards and buckets to be accessed concurrently.

Deletion Algorithm. Deletions can remove either a single value or an entire key. For a value-level deletion, the thread locks the shard exclusively, finds the bucket, and then locks the bucket to erase the value from its set. If this leaves the bucket empty, it is removed from the shard's map. A full-key deletion skips the bucket step and instead takes an exclusive lock on the shard before removing the entire bucket entry from the shard's map in one operation. This process reclaims unused buckets and helps maintain memory usage within control.

Search Algorithm. A search operation first acquires a shared lock on the shard that contains the key and locates the corresponding bucket. If found, it takes a shared lock on the bucket and returns a copy of its value set. Because both locks are shared, many readers can run in parallel—even on the same shard or the same bucket. All operations acquire locks in a fixed order: shard, followed by bucket, which avoids deadlocks.

The inverted index is designed to support high read concurrency. Lookups use only shared locks on the target shard and bucket, so many readers can run in parallel without blocking each other. Writes use short exclusive locks on just the affected bucket, while sharding spreads writes across many shards to prevent contention from building up in one place. This keeps lock conflicts low, even under heavy load, and makes the structure well-suited as a purpose index, where most queries are lookups over a large set of keys whose associated record sets are frequently updated.

6 Evaluation

This chapter evaluates the three indexing structures designed to support efficient GDPR-related metadata queries in our system: (i) a lock-free Skip List used as the Retention index, (ii) a B+ Tree used as the Subject index, and (iii) a sharded Inverted Index used as the Purpose index. These indices are designed to reduce query latency and increase throughput for GDPR workloads, which often involve high concurrency, large metadata volumes, and diverse query patterns such as lookups by user, by purpose, or by expiration time.

6.1 Methodology

6.1.1 Objectives

The objective of this evaluation is to assess how effectively each index structure supports GDPR-related metadata workloads at scale. Specifically, we focus on the following aspects:

- **Performance:** Measure the throughput of each structure as workload intensity and the number of concurrent worker threads increase.
- **Scalability:** Assess how effectively each structure maintains performance when scaling from 1 to 16 worker threads under varying workloads.
- **Memory usage:** Measure how the Resident Set Size (RSS) of each data structure changes as more records are inserted during workload execution.

6.1.2 Workload Model

We evaluate index structures using the standard YCSB workloads. Each benchmark has two phases. The *load* phase initializes the index by inserting $1M$ key-value pairs. The *run* phase executes a YCSB-generated trace that mixes reads, writes, and scans.

Trace entries are of the form `PUT("k","v")`, `GET("k")`, or `SCAN("start","end")`. In our implementation, `PUT` maps to `insert`, `GET` to `search`, and `SCAN` to a range query, executed only on indices that support it.

We use the standard YCSB core workloads:

- **Workload A (Update heavy)**: 50% reads and 50% updates.
- **Workload B (Read mostly)**: 95% reads and 5% updates.
- **Workload C (Read only)**: 100% reads.
- **Workload D (Read latest)**: 95% reads and 5% inserts, focusing on recently added records.
- **Workload E (Short ranges)**: 95% short range scans and 5% inserts.
- **Workload F (Read–modify–write)**: reads a record, modifies it, and writes it back.

To evaluate how system performance scales under different conditions, we varied the number of worker threads (1 to 16) to assess scalability under increasing concurrency, and the value size (64 B to 4 KB) while keeping the key size fixed at 64 B to examine the impact of payload size on throughput.

6.1.3 Evaluation Setup

All experiments were run on a dedicated server with the following configuration:

- **Operating System:** NixOS (Linux kernel 6.15.11, x86_64)
- **CPU:** 2× Intel Xeon Gold 6326 @ 2.90 GHz
- **Cores and Threads:** 32 physical cores (64 hardware threads with 2-way hyper-threading)
- **NUMA:** 2 NUMA nodes with 32 threads each
- **Cache:** 1.5 MiB L1d, 1 MiB L1i, 40 MiB L2 (aggregate), and 48 MiB shared L3

This setup offers a high level of parallelism and large cache capacity, making it well-suited for evaluating the scalability and concurrency characteristics of the index structures.

6.1.4 Metrics

We collect two primary metrics:

- **Throughput (ops/sec):** The number of successful operations per second, used to assess performance and scalability.
- **RSS:** Memory consumption over time during inserts, used to track growth as the dataset expands.

6.2 Retention Index

The lock-free skip list is used as the Retention index to manage record expiration times. Its primary function is to support frequent insertions of new records with associated expiration timestamps while concurrently removing expired ones in the background. This workload pattern is inherently write-intensive, yet must remain highly concurrent to avoid interfering with client operations.

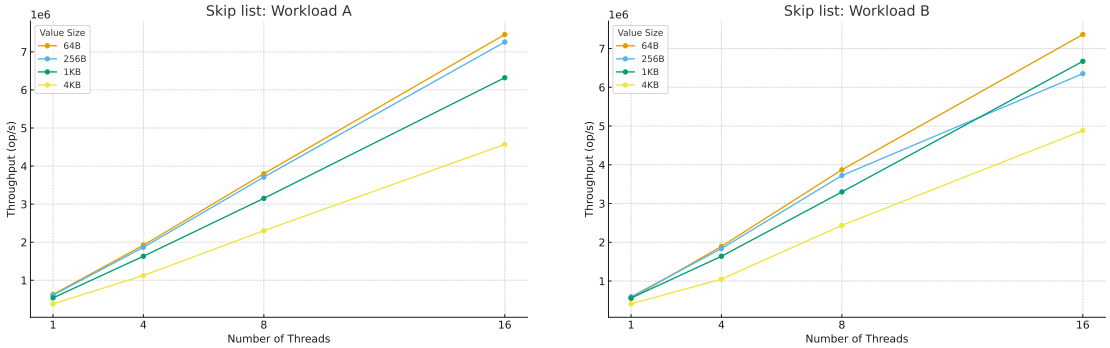
6.2.1 Throughput Results

Figure 6.1 shows the throughput of the skip list across the five YCSB workloads (A, B, C, D, and F) with varying thread counts (1, 4, 8, 16). Each workload stresses different mixes of read, update, and insert operations, allowing us to observe how the structure behaves under increasing contention.

Analysis. Across all workloads, the skip list scales nearly linearly with thread count, showing that both read-heavy and update-heavy operations benefit from its lock-free design.

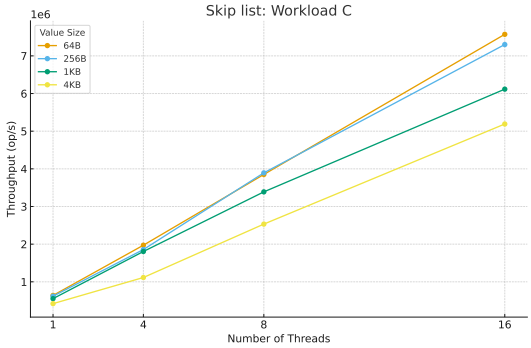
Under Workload A, which mixes reads and updates evenly, throughput increases from about 6.25×10^5 op/s with a single thread to around 7.46×10^6 op/s at 16 threads for 64 B values, giving a speedup of about $12\times$. Even with larger 4 KB values, throughput rises from roughly 3.8×10^5 op/s at 1 thread to about 4.57×10^6 op/s at 16 threads, a speedup of about $11.76\times$, showing the added cost of handling larger records while still maintaining good scalability.

At 256 B values, throughput in Workload B grows from about 5.9×10^5 op/s at 1 thread to 6.53×10^6 op/s at 16 threads, a speedup of about $11\times$, while Workload C increases from 6.17×10^5 op/s to 7.3×10^6 op/s, a speedup of about $11.8\times$. These results demonstrate that read-heavy access patterns scale effectively with the addition of extra threads.

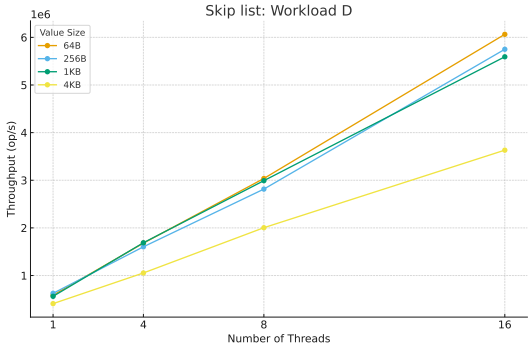


(a) Workload A

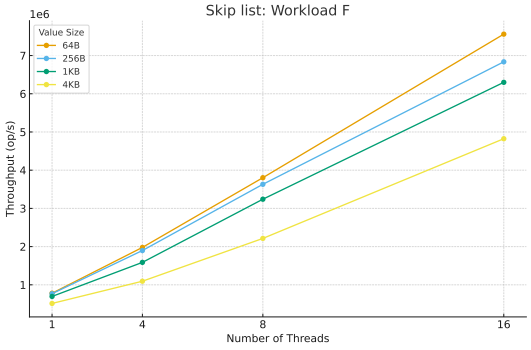
(b) Workload B



(c) Workload C



(d) Workload D



(e) Workload F

Figure 6.1: Throughput of the skip list under YCSB workloads as the number of concurrent threads increases.

Workloads D and F include write operations. Workload D combines 95% reads with 5% inserts, while Workload F mixes 50% reads with 50% read-modify-write updates, where each update reads a record and then writes back a modified version. In Workload D with 256 B values, throughput increases from about 6.2×10^5 op/s at 1 thread to 5.75×10^6 op/s at 16 threads, a speedup of about 9.3×. In Workload F with 256 B values, throughput rises from about 7.67×10^5 op/s at 1 thread to 6.83×10^6 op/s at 16 threads, a speedup of about 8.9×. The similar slopes of these curves show that contention on CAS-based pointer updates is minimal, allowing concurrent updates and lookups to proceed in parallel without interfering with each other.

6.2.2 Memory Usage

We also measured the memory footprint of the skip list during the execution of Workload A to understand how memory usage evolves across the load and run phases. Figure 6.2 shows the RSS over time.

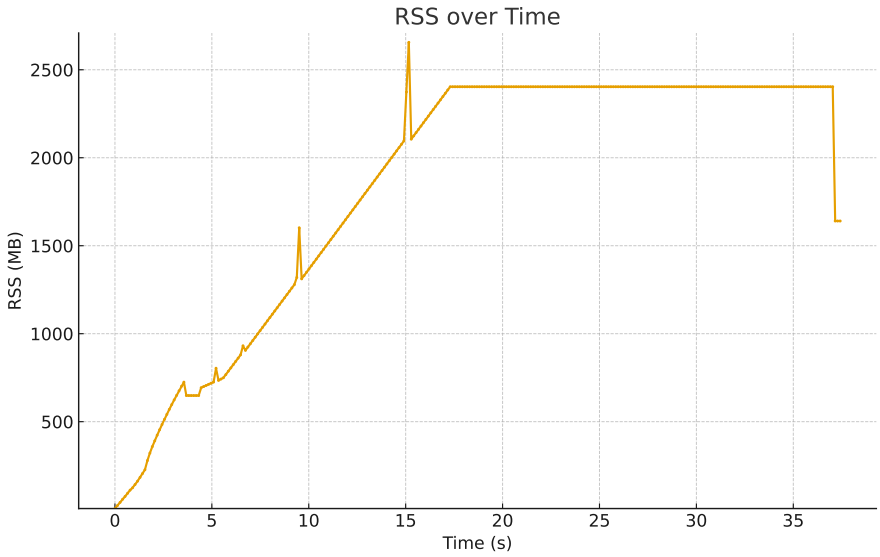


Figure 6.2: Resident set size (RSS) of the skip list under Workload A. Memory rises linearly during the load phase as records are inserted, then flattens in the run phase, where updates overwrite existing entries.

Analysis. Memory use grows steadily during the load phase, which inserts new records and builds the skip list structure. This linear growth reflects the combined cost of storing each key, its value, and the multiple forward pointers associated with each

node. Given 1 million keys of 64 B each and values of 64 B, the total payload amounts to approximately 128 MB. The observed peak RSS during the load phase reaches around 2460 MB. This means that only around 5% of the memory footprint comes from the actual key–value data, while the rest is overhead. The additional memory use is partly due to the skip list structure internals, and partly due to the experimental setup, which maintains auxiliary arrays, structures needed to run the workload. Once the dataset is loaded, the run phase of Workload A mainly performs updates and reads. When the workload shifts to the run phase, which primarily performs updates and reads, memory consumption stabilizes because updates overwrite existing values rather than creating new nodes. Only minor fluctuations are observed near the transition point, but the overall trend remains consistent.

Conclusion. The skip list exhibits predictable memory usage, with linear growth during record insertion and stability under update-intensive workloads. This enables the direct estimation of memory needs from the number of inserted records and the configured key and value sizes.

6.2.3 Summary

These results demonstrate that the skip list effectively handles high concurrency while maintaining stable memory overhead. Its lock-free design allows background expiration to proceed without blocking foreground operations, which is essential for maintaining system responsiveness as the data volume and query rate increase. This makes the skip list a practical choice for implementing the Retention index.

6.3 Subject Index

The Subject index is implemented as a B+ tree, where user identifiers serve as the keys, and each leaf node stores references to all records belonging to a user. Its primary role is to support efficient point lookups and range scans over user IDs, grouping all of a user's records while preserving sorted order. This structure is expected to see mostly lookup-heavy workloads, but must also handle frequent inserts as new records are added.

6.3.1 Throughput Results

Figure 6.3 shows the throughput of the B+-tree across the six YCSB workloads (A–F) under varying thread counts (1, 4, 8, 16). These workloads include both read-heavy and

update-heavy mixes, allowing us to observe how the structure scales under increasing concurrency.

Analysis. The B+-tree shows moderate scaling across the workloads. In Workloads A, B, C, and F, throughput increases from 1 to 4 threads, drops noticeably at 8 threads, and then stays roughly flat at 16 threads instead of recovering. This behavior reflects the overhead of lock coupling, as operations must acquire locks on shared upper-level nodes while traversing the tree, and these nodes become bottlenecks as more threads compete for them.

In Workload A (50% read and 50% update), throughput rises from about 8×10^5 op/s at 1 thread to 9.02×10^5 op/s at 4 threads, then drops sharply to around 2.76×10^5 op/s at 8 threads, and remains low at 2.69×10^5 op/s at 16 threads for 64 B values. This drop at 8 and 16 threads shows that once many threads operate concurrently, contention on the upper nodes dominates overall performance.

Workloads B (95% reads and 5% updates) and C (100% reads) show a similar pattern despite being read-heavy. Workload B rises from about 8.47×10^5 op/s at 1 thread to 8.85×10^5 op/s at 4 threads, drops to 3.17×10^5 op/s at 8 threads, and further to 2.54×10^5 op/s at 16 threads for 64 B values. Workload C increases from 8.43×10^5 op/s to 9.32×10^5 op/s at 4 threads, falls to 3.24×10^5 op/s at 8, and 2.84×10^5 op/s at 16 for 64 B values. Even though these workloads only require shared locks, many threads still attempt to lock the same upper-level nodes simultaneously. This forces them to wait on each other, creating delays that slow down overall throughput. Since the extra threads do not spread the work across more parts of the tree, contention stays high, and throughput continues to drop instead of improving.

Workload D (95% reads, 5% inserts) shows a different pattern. throughput increases from about 8.77×10^5 op/s at 1 thread to roughly 1.09×10^6 op/s at 4 threads, then declines almost linearly, reaching around 3.14×10^5 op/s at 16 threads for 64 B values. This steady decline suggests that inserts cause structural changes that create contention on upper-level nodes, where lock coupling causes threads to wait on each other as concurrency increases, thereby limiting scalability.

Workload E (short range scans) scales more steadily, rising almost linearly from about 5.07×10^5 op/s at 1 thread to around 6.17×10^5 op/s at 16 threads for 64 B values. This indicates that range scans can benefit from parallelism because different threads often scan different leaves simultaneously. However, their overall throughput stays relatively low because every scan still passes through the same upper-level nodes, which creates lock contention as more threads run. Each scan touches multiple leaves, which means each operation does more work than a simple point lookup.

Workload F (50% reads, 50% read-modify-write) follows a pattern similar to Work-

loads A, B, and C. throughput rises from about 1.07×10^5 op/s at 1 thread to around 1.14×10^6 op/s at 8 threads, but then stays close to 1.07×10^6 op/s at 16 threads, instead of continuing to scale. The write operations trigger node splits and increase lock contention on upper nodes, which keeps the throughput from improving further.

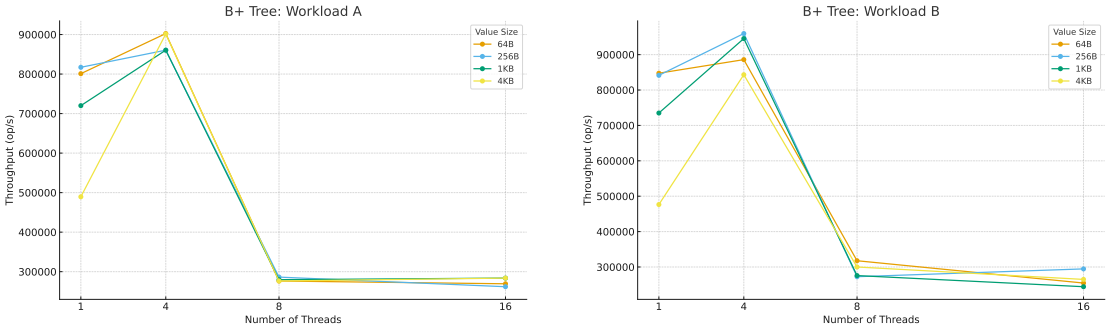
Workload F (50% reads and 50% read–modify–write) follows the same shape as Workloads A–C. throughput increases from 1 to 4 threads, drops at 8 threads, and then stays roughly the same at 16 threads. Because the modify step requires exclusive locks, threads spend more time waiting on the upper-level nodes. Once more than 8 threads are active, this waiting becomes the main bottleneck, so adding further threads does not improve throughput.

6.3.2 Memory Usage

We also tracked the memory footprint of the B+ tree under Workload A. The results are presented in Figure 6.4.

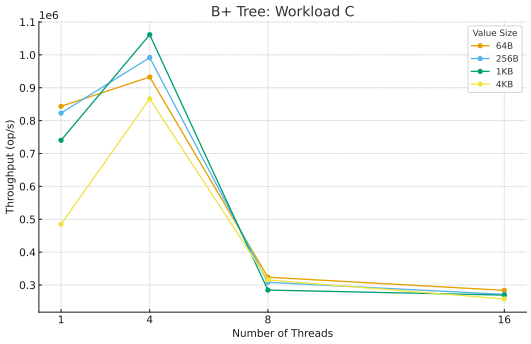
Analysis. During the load phase, memory usage grows in a nearly linear manner as records are inserted and new nodes are allocated. This continues until the dataset is fully loaded, after which the run phase begins. In this phase, most operations are updates that overwrite existing entries rather than creating additional nodes, so the overall size of the structure remains stable and memory usage levels off. With 1 million keys, each measuring 64 bytes, and associated values of the same size, the approximate total data size is around 128 MB. The highest recorded RSS during the loading process is about 2417 MB. This suggests that roughly 5% of the memory consumption relates to the actual key-value pairs, while the remaining portion is linked to overhead. The elevated memory usage can be partially attributed to the characteristics of the B+ tree architecture, as well as to the experimental setup, which entails additional arrays and structures needed to carry out the workload. After the dataset has been fully loaded, the execution phase of Workload A mainly consists of update and read operations. The run phase also completes in less time compared to skip list, reflecting the efficiency of the B+-tree in handling update workloads once the structure has been built.

Conclusion. The B+-tree shows a clear trend, with memory use rising steadily during the load phase and remaining stable once updates take over. This predictable footprint and efficient handling of the run phase make it a good fit for workloads that involve frequent lookups alongside moderate updates.

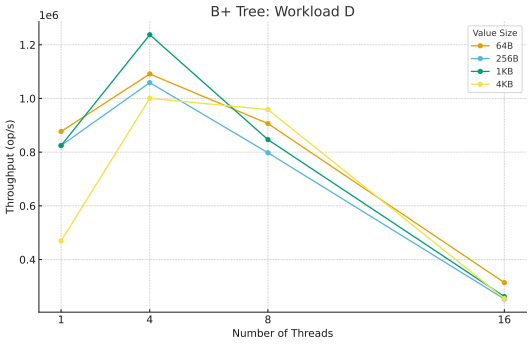


(a) Workload A

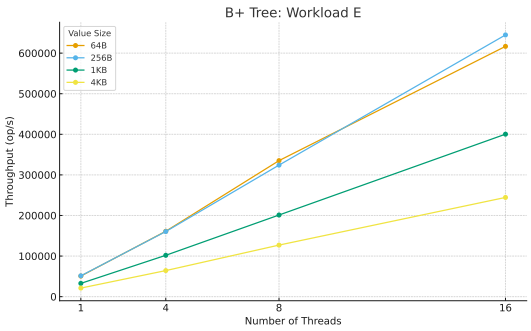
(b) Workload B



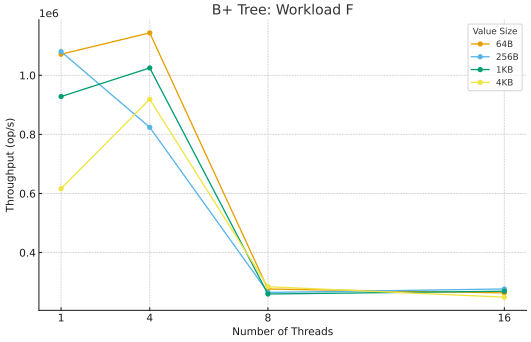
(c) Workload C



(d) Workload D



(e) Workload E



(f) Workload F

Figure 6.3: Throughput of the B+ Tree under YCSB workloads as the number of concurrent threads increases.

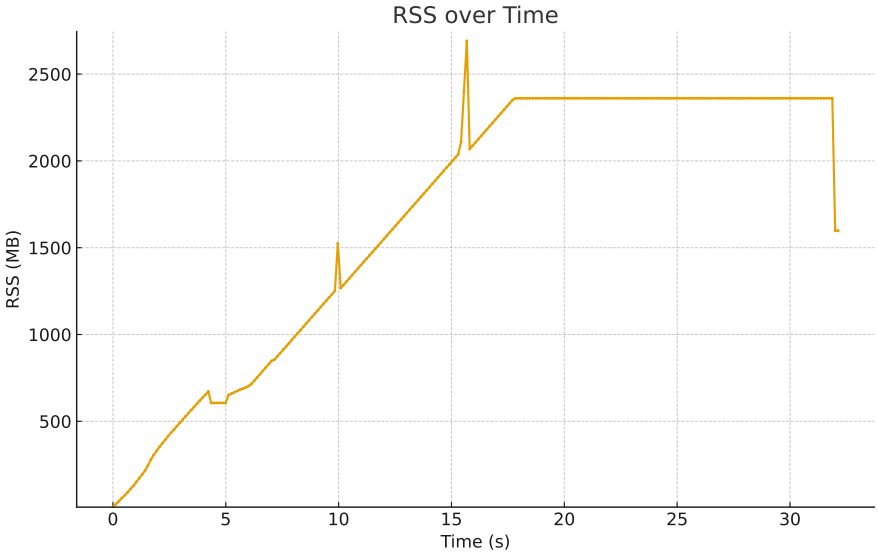


Figure 6.4: Resident set size (RSS) of the B+ tree under Workload A. Memory grows linearly during the load phase, but with a steeper slope than the skip list, as insertions are completed more quickly. Once loading finishes, memory usage levels off during the update phase.

6.3.3 Summary

The results show that the B+-tree can handle concurrent workloads with mixed reads and writes, but its scalability is limited. While it performs well at low thread counts, contention on upper-level nodes restricts throughput as concurrency increases, and performance does not improve beyond four threads. This makes the B+-tree most suitable as the Subject Index, where efficient grouping by user and fast lookups are required, but workloads involve only moderate levels of concurrency.

6.4 Purpose Index

The Purpose index uses a sharded inverted index that maps each purpose to its set of record identifiers. To support concurrent access, it combines fine-grained bucket-level locks with shard-level partitioning to limit contention. This design is well-suited to lookup-heavy workloads, as records can be tagged with one or more purposes, and the system must support frequent updates when records are inserted or removed.

6.4.1 Throughput Results

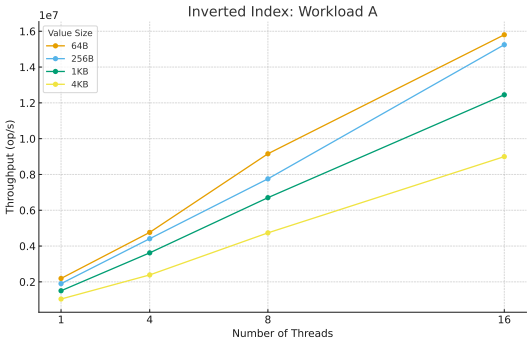
Figure 6.5 shows the throughput of the inverted index across the five YCSB workloads (A, B, C, D, and F) with varying thread counts (1, 4, 8, 16). Each workload stresses different mixes of read, update, and insert operations, allowing us to observe how the structure behaves under increasing contention.

Analysis. The Purpose index demonstrates strong scalability across all workloads, with throughput generally rising in proportion to the thread count as it leverages available parallelism.

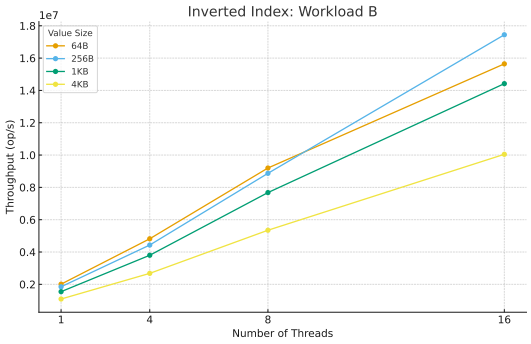
In Workload A, which has a balanced mix of reads and updates, throughput grows from about 2.19×10^6 op/s with a single thread to over 1.58×10^7 op/s at 16 threads for 64 B values, achieving a 7.2× speedup. When the record size increases to 4 KB, throughput at 16 threads drops to around 9×10^6 op/s, reflecting the extra cost of handling larger records while still maintaining good scaling.

With 256 B values, read-heavy workloads also scale well. Workload B improves from roughly 1.85×10^6 op/s at 1 thread to 1.74×10^7 op/s at 16 threads ($\approx 9.4\times$ speedup), and Workload C rises from 1.55×10^6 op/s to 1.71×10^7 op/s ($\approx 11\times$ speedup). These results show that additional threads continue to improve throughput when contention is low.

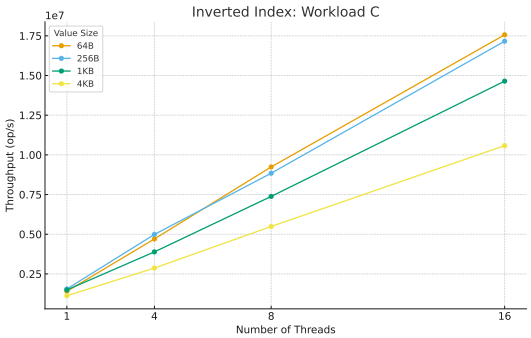
Workloads D and F introduce write operations. Workload D includes 5% inserts alongside 95% reads, while Workload F performs 50% reads and 50% read-modify-



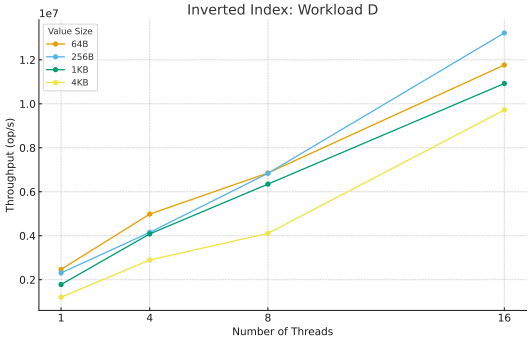
(a) Workload A



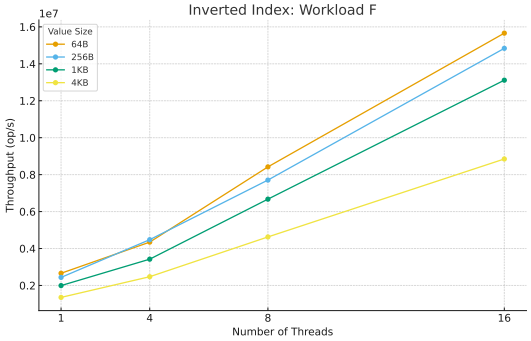
(b) Workload B



(c) Workload C



(d) Workload D



(e) Workload F

Figure 6.5: Throughput of the inverted index under YCSB workloads as the number of concurrent threads increases.

write updates. In Workload D with 256 B values, throughput increases from about 2.3×10^6 op/s at 1 thread to around 1.32×10^7 op/s at 16 threads ($\approx 5.7\times$ speedup). In Workload F, throughput goes from about 2.43×10^6 op/s at 1 thread to around 1.48×10^7 op/s at 16 threads ($\approx 6.1\times$ speedup). The similar growth pattern suggests that write operations do not introduce major bottlenecks. Inserts and updates on a single-purpose key proceed independently of lookups on other keys, allowing operations to run in parallel without interference.

6.4.2 Memory Usage

Finally, we measured the memory footprint of the inverted index under Workload A. The RSS trend is shown in Figure 6.6.

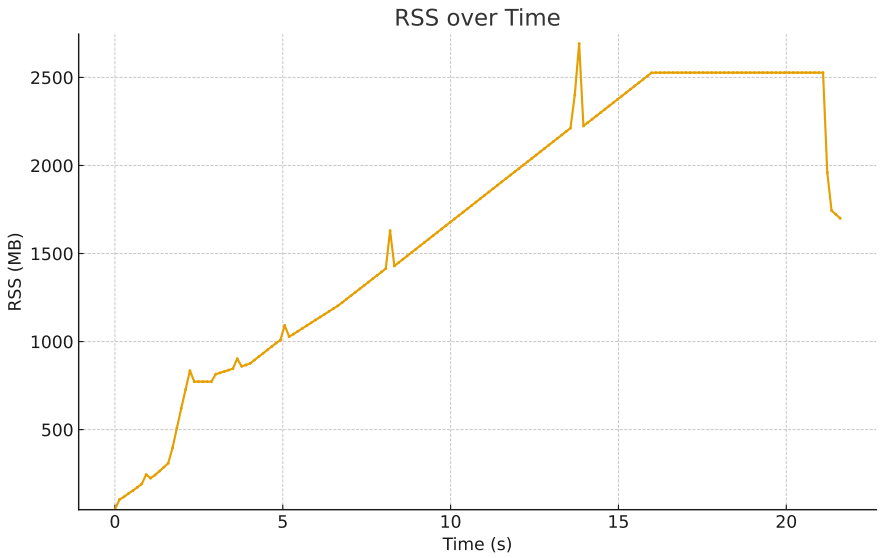


Figure 6.6: Resident set size (RSS) of the inverted index under Workload A. Memory increases sharply in the load phase due to growing identifiers lists, then stabilizes in the run phase, where most operations update existing entries.

Analysis. In the load phase, memory usage rises quickly because each new record adds entries to identifiers lists associated with its purpose. This expansion is more memory-intensive than the skip list or B+-tree because identifiers lists can grow long and are often distributed across buckets, each with locking structures. Once the load phase is complete, the run phase primarily performs updates, which replace values without significantly altering the size of identifiers lists. As a result, memory consumption

stabilizes after the initial sharp growth. With 1 million keys, each 64 bytes in size, and corresponding values of 64 bytes, the overall data size totals roughly 128 MB. The maximum RSS observed during the loading phase is approximately 2588 MB. This indicates that only about 4.9% of the memory usage is attributed to the actual key-value pairs, while the remainder consists of overhead. The extra memory consumption results partly from the internal structure of the inverted index and partly due to the experimental setup, which includes maintaining additional arrays and structures necessary for executing the workload. After loading the dataset, the execution phase of Workload A primarily focuses on performing updates and reads.

Conclusion. The inverted index has the steepest memory growth of the three structures, reflecting the overhead of maintaining identifiers lists for purpose-based queries. However, once the data is loaded, memory usage remains stable during update workloads, making it suitable for workloads where grouping by purpose is essential despite higher memory cost.

Analysis. During the load phase, memory usage increases steadily as records are inserted, since each record contributes to the lists of identifiers maintained for its purpose. Compared to the skip list and B+-tree, this structure requires more memory overall, because the purpose-based lists grow large and are spread across multiple buckets with their own synchronization overhead. Once the load phase is complete, the run phase mainly performs updates, which overwrite values without significantly changing the size of these lists. As a result, memory usage stabilizes after the dataset has been loaded.

Conclusion. The inverted index shows the highest memory consumption of the three structures, reflecting the extra cost of maintaining purpose-based record lists. However, its memory footprint stabilizes once the data is loaded, making it suitable for workloads that rely on grouping records by purpose, even if it requires more memory than the other indices.

6.4.3 Shard Scaling

Figure 6.7 shows how sharding affects the throughput of the inverted index under Workload A (50% reads and 50% updates). throughput improves sharply as the number of shards increases from 1 to 64, rising from about 3.8×10^6 op/s to nearly 5.9×10^6 op/s. With only a few shards, many keys are mapped to the same shard, so multiple threads frequently attempt to update or read from the same bucket. This

contention causes threads to wait on each other, resulting in low throughput. Increasing the number of shards spreads keys more evenly across buckets, reducing conflicts and making it less likely that threads block one another. As a result, throughput improves significantly with higher shard counts..

Beyond 64 shards, throughput doesn't improve and remains close to 5.9×10^6 op/s even as the shard count increases further. At this point, contention within shards is minimal, and other costs such as coordination across shards and memory overhead dominate. This plateau indicates that the index saturates once the workload is sufficiently distributed, and adding more shards does not yield further gains. In practice, configuring around 64 shards provides a good balance between higher throughput and system overhead.

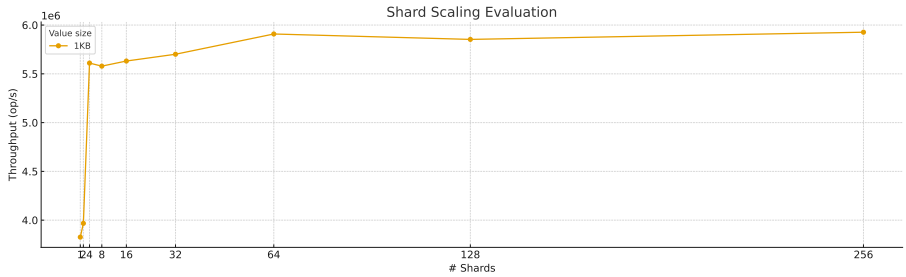


Figure 6.7: throughput of the inverted index under Workload A with different shard counts. Performance improves as contention decreases, but adding shards beyond 64 does not increase throughput.

6.4.4 Summary

The results show that the inverted index supports efficient purpose-based queries but incurs the highest memory cost among the three structures. Its design allows records to be grouped and retrieved by processing purpose with low latency, which is essential for GDPR workloads that involve filtering or aggregating by declared purpose. However, the additional overhead of maintaining large purpose lists and synchronization across buckets makes it more memory-intensive. This trade-off makes the inverted index well-suited as the Purpose Index, where query efficiency outweighs memory considerations and grouping by purpose is a central requirement.

7 Related Work

The emergence of the GDPR has significantly impacted not only regulatory requirements but also the design of database systems and storage architectures. Essentially, the regulation sets out rights for individuals, including rights of access, rectification, portability, and erasure, as well as obligations for organizations, such as timely breach notification, accountability, and evidence of compliance [Reg16]. Turning these legal articles into practical system operations has exposed significant conflicts between regulatory demands and database design [Sha+19a; Sha+19b; Aga+21]. For example, the right to erasure requires that all data about a subject be removed promptly and thoroughly, which is a non-trivial operation in distributed systems that replicate and index data across multiple nodes.

Meanwhile, decades of research in indexing and concurrency control provide a rich toolbox for efficient search and update operations. B-trees and their variants dominate classical database systems [BM70; LY81], while skiplists and log-structured merge trees power modern key–value stores [Pug90; ONe+96; Don+21]. Inverted indexes form the foundation of large-scale search engines [ZM06], and more recently, learned indexes have been proposed as a way to achieve improved trade-offs between memory efficiency and query performance [Kra+18; Din+20]. Concurrency techniques, such as lock coupling, optimistic lock coupling, and latch-free designs, allow these indexes to scale across multiple cores and threads [LY81; Lei+16]. However, most existing index structures have not been designed with GDPR-specific requirements in mind, such as the need for timely erasure, efficient subject-level queries, or maintaining auditable access trails.

This section reviews related work in three areas: system studies addressing GDPR compliance, indexing structures relevant to key-value data stores, and concurrency control mechanisms for high-performance indexing. The review concludes with an identification of research gaps that motivate the design of a GDPR-aware indexing layer for key-value stores.

7.1 GDPR and Storage Systems

The GDPR is the most comprehensive data protection law that has been issued to date [Reg16]. It grants individuals rights, such as the right to access, rectification, portability,

and erasure of their personal data. It imposes obligations on organizations, including obtaining explicit consent, notifying them of breaches, and demonstrating provable accountability. Non-compliance can lead to severe financial penalties, up to €20 million or 4% of the company's global annual revenue, whichever is higher.

While expressed as legal rights and obligations, the GDPR's articles translate into concrete system-level requirements for data storage and management. For example, article 17 (Right to Erasure) mandates that all personal data be deleted across all storage locations, including backups and replicas, in a timely and verifiable manner. Articles 15 and 20 (Right to Access and Right to Data Portability) require efficient mechanisms for retrieving all data related to a subject, often spanning multiple tables or storage engines. Furthermore, Articles 5, 24, and 30 establish accountability requirements, obligating systems to maintain verifiable records of data access and demonstrate compliance with the regulation. Collectively, these requirements pose significant technical challenges for storage engines and database systems, which are optimized for performance and scalability but not necessarily for regulatory compliance. Several studies have investigated system-level adaptations to address these challenges.

Shah et al. [Sha+19a] investigate the implications of GDPR on storage systems by retrofitting Redis with compliance features. Their study shows that enforcing strict real-time compliance through synchronous per-request logging can reduce throughput by as much as 20 times. They highlight that more than 30% of GDPR articles relate directly to storage and argue that compliance should be viewed as a spectrum rather than a binary property. Importantly, their work outlines six key features that storage engines should provide in order to achieve GDPR compliance: (i) *timely deletion*, through mechanisms such as time-to-live (TTL) counters that enforce automatic removal of expired data; (ii) *monitoring and logging*, requiring audit trails of both data-path operations (reads and writes) and control-path operations (metadata changes, access control); (iii) *indexing via metadata*, to enable efficient retrieval and deletion of subject-level data or data grouped by purpose; (iv) *access control*, providing fine-grained, dynamic policies that define which entities may access data, for what purposes, and for how long; (v) *encryption*, ensuring that personal data remains protected at rest and in transit, thereby safeguarding against exposure even if storage media or communication channels are compromised; and (vi) *managing data location*, providing mechanisms to enforce GDPR's geographical restrictions on storage and movement of personal data. These findings underscore the extent of architectural changes necessary for storage systems to comply with regulatory requirements.

Shastri et al. [Sha+19b] extend the analysis of GDPR by translating its articles into database-level capabilities and workloads. They identify metadata explosion as a central challenge, noting that compliance requires storing and managing substantial volumes of auxiliary metadata in addition to personal data. To systematically evaluate

system readiness, they introduce GDPRbench, a benchmark comprising GDPR-specific workloads and metrics. Using this framework, they modify Redis, PostgreSQL, and a commercial database system, implementing GDPR features according to established best practices. Their evaluation shows that these systems perform poorly in compliance workloads and that scalability deteriorates as data volume increases. These results indicate that current database systems do not adequately meet GDPR requirements, highlighting the need for new architectures that integrate compliance as a primary design goal.

Agarwal et al. [Aga+21] address the challenge of GDPR compliance in legacy relational databases, where identifying and extracting all data associated with an individual is often a manual and error-prone task. They present GDPRizer, a semi-automated tool that uses foreign keys, query logs, data-driven techniques, and administrator-provided annotations to retrieve all data belonging to an individual. In case studies on three widely used web applications, GDPRizer achieved 100% precision and 96-100% recall, significantly reducing the effort required compared to hand-written queries. While manual verification of results remains necessary, the study demonstrates that automated extraction tools can substantially ease compliance in practice. However, the reliance on schema annotations and application-specific heuristics illustrates the limitations of retrofitting approaches and signals the need for more integrated compliance solutions in future system designs.

Integrating these capabilities into high-performance key-value stores remains an open challenge. In summary, GDPR requires features such as timely deletion, subject-level data retrieval, and auditable access that existing systems cannot easily provide. Prior work on GDPR compliance in storage and database systems shows that compliance cannot simply be retrofitted. Instead, it requires a fundamental redesign of core storage and indexing mechanisms.

7.2 Indexing Structures for Data Systems

Efficient indexing is fundamental to the performance of database systems, as it enables fast retrieval, insertion, and deletion of records without scanning entire datasets. Over the decades, a wide variety of indexing structures have been developed, each optimized for different workloads and storage environments. These include balanced search trees such as B-trees and their variants [BM70; LY81], probabilistic structures such as skip lists [Pug90], log-structured merge trees for write-intensive workloads [ONe+96; LC19], inverted indexes that support large-scale text search [ZM06], and, more recently, learned indexes that leverage machine learning models for prediction-based lookup [Kra+18; Din+20]. While these structures have been highly successful in optimizing

query latency and throughput across relational, key-value, and search systems, they were not designed with regulatory compliance in mind. As a result, supporting GDPR requirements such as subject-level data retrieval, provable erasure, or auditable access trails requires revisiting the assumptions underlying traditional index design.

7.2.1 B-trees and Variants

The B-tree, introduced by Bayer et al. [BM70], is the foundational indexing structure in relational database systems. It organizes keys in a balanced search tree, ensuring logarithmic time complexity for search, insertion, and deletion. The B-tree quickly became the dominant index in relational database systems due to its ability to manage large datasets on disk while minimizing I/O operations efficiently. Over the years, numerous variants have extended its capabilities. The Blink-tree, introduced by Lehman and Yao [LY81], extends the B-tree with an additional link pointer between sibling nodes, enabling traversals to proceed safely even in the presence of concurrent modifications. By reducing the locking scheme to a small, constant number of nodes and eliminating the need for read locks, it became a practical solution for scaling B-tree performance in concurrent, multi-user environments. While these studies have firmly established the B-tree and its variants as cornerstones of database indexing, they were developed with performance and concurrency in mind rather than compliance. GDPR-specific requirements such as subject-level data retrieval, verifiable erasure, and accountable access logging remain outside the scope of these designs. B-trees still form the backbone of transactional databases, but adapting them for compliance-aware indexing requires a fundamental redesign.

7.2.2 Skip Lists and LSM-Trees

Skip lists, introduced by Pugh et al. [Pug90], offer a probabilistic alternative to balanced trees, enabling efficient search and insertion through a layered linked list structure. Their structural simplicity makes them especially appealing for concurrent and distributed settings [Fra04], and they are frequently employed as in-memory components within log-structured merge trees (LSM-trees).

The LSM-tree, proposed by O'Neil et al. [ONe+96], improves write performance by buffering updates in memory and flushing them to disk sequentially. This design forms the basis of widely used key-value stores, such as LevelDB [GD11] and RocksDB [Don+21], and has become the standard indexing technique for large-scale cloud systems. Over time, LSM-trees have been refined with optimizations, including tiering merge policies, Bloom filters, and hybrid designs [LC19]. Despite their efficiency, LSM-trees handle deletion through tombstones, which mark records as invalid but only

remove them during background compaction. This deferred process is fundamentally misaligned with GDPR's requirement for timely and verifiable erasure.

To address this limitation, Sarkar et al. [Sar+20] proposed Lethe, an LSM variant that treats deletions as first-class operations. Lethe introduces lightweight metadata, delete-aware compaction policies, and a new physical layout that weaves the sort and the delete key orders. This design enables configurable guarantees on deletion latency, efficient range deletes on secondary keys, and improved read throughput and space amplification, at the cost of moderately higher write amplification. By providing predictable and timely erasure, Lethe demonstrates the trade-offs required to move LSM-based stores closer to GDPR compliance.

7.2.3 Learned Indexes

Learned indexes represent a recent shift in indexing design, treating indexes as predictive models rather than purely algorithmic structures. Kraska et al. [Kra+18] demonstrated that machine learning models can outperform traditional B-trees for static workloads by predicting key positions more compactly, while Ding et al. [Din+20] extended this approach with ALEX, an adaptive structure supporting dynamic inserts, updates, and deletes. These designs achieve impressive space-time trade-offs, highlighting the efficiency potential of model-based indexing. However, they remain focused on performance optimization rather than regulatory guarantees. GDPR-related requirements, such as subject-level data extraction, verifiable erasure, and accountable access, have not been addressed in this line of work, leaving open questions about how compliance features can be integrated into predictive index models.

7.3 Concurrency Control

Concurrency control is fundamental to the design of indexing structures, as it guarantees correctness, isolation, and consistency in the presence of concurrent operations. The widespread adoption of multi-core architectures and the growth of highly concurrent workloads have driven the evolution of indexing techniques toward supporting parallel access while reducing contention and synchronization costs.

Early approaches relied on classical lock-based protocols, which provide strong correctness guarantees but often suffer from blocking and contention under heavy concurrency. To address these limitations, refinements such as lock coupling and its extension, optimistic lock coupling, were introduced. These techniques achieve a trade-off between safety and performance by reducing unnecessary blocking while still coordinating concurrent traversals and updates. More recently, researchers have

explored lock-free designs, which eliminate locks in favor of non-blocking synchronization. By relying on atomic hardware primitives, lock-free indexes can guarantee system-wide progress and avoid many pitfalls of lock-based methods, though often at the cost of more complex design and implementation.

This chapter provides an overview of concurrency control in indexing structures, tracing its development from early lock-based protocols to advanced non-blocking methods designed for high scalability.

7.3.1 Classical Locking and Lock Coupling

The earliest attempts to provide concurrency in indexing structures relied on coarse-grained locks, such as tree-level or page-level locking. These methods were simple and guaranteed correctness, but they did so at the cost of scalability. With a coarse lock in place, only one operation could proceed in the protected region of the tree at a time, which quickly became a bottleneck as the number of concurrent users increased.

Lehman et al.'s Blink-tree [LY81] represented a significant advance in the design of concurrent B-trees. Their design introduced sibling pointers, allowing traversals to continue safely even when another process splits a node. This mechanism was combined with what is now called *lock coupling*, where a thread acquires the lock on a child before releasing the lock on its parent. In this way, a traversal holds only a constant number of locks, while still ensuring that concurrent operations see a consistent view of the structure. The algorithm required only minor changes compared to a single-threaded B-tree, but it opened the door to practical fine-grained concurrency.

Lock coupling proved effective, but it also exposed the weaknesses of classical locking. Contention remained high near the root of the tree, and the blocking nature of locks meant that slow or blocked threads could delay others. These limitations became more severe on modern multi-core processors, motivating the development of new approaches, such as Optimistic Lock Coupling (OLC) and non-blocking designs, that reduce contention and avoid blocking altogether.

7.3.2 Optimistic Lock Coupling

While classical lock coupling provided a practical approach to concurrency in B-trees, its reliance on holding locks during traversals limited scalability on modern multi-core processors. To address these limitations, Leis et al. [Lei+16] introduced *OLC* as part of their work on synchronizing the Adaptive Radix Tree (ART). OLC represents a middle ground between traditional fine-grained locking and complex lock-free data structures. The central idea is that traversals proceed without holding locks, and locks are only acquired when a thread is about to apply a modification. At that point, the node's

version number is validated to ensure that no conflicting updates occurred since the traversal began.

This optimistic validation minimizes the time locks are held, thereby reducing contention and enabling better utilization of modern hardware parallelism. Compared to lock-free methods, OLC is simple to implement, requiring only minor changes to existing data structures such as ART or B-trees. Leis et al. demonstrated that OLC scales efficiently on multi-core systems and performs particularly well under read-heavy workloads, where the probability of conflicts is low.

However, OLC is not free from limitations. In write-intensive or contentious workloads, failed validations may force repeated retries, which reduces their effectiveness. Furthermore, OLC still relies on locks during updates, and thus does not provide the non-blocking progress guarantees of fully lock-free data structures. Nevertheless, its simplicity, portability, and robust performance in practice make it a significant step in the progression from classical lock-based protocols to modern non-blocking synchronization techniques.

7.3.3 Latch-Free and Lock-Free Designs

While optimistic lock coupling reduced contention by shortening the duration of critical sections, it still relied on locks during updates. To further push scalability, researchers have explored latch-free and lock-free designs that eliminate blocking synchronization altogether. In these approaches, threads rely on atomic hardware primitives (such as compare-and-swap) to guarantee correctness without acquiring latches or locks. The goal is to ensure that at least one thread always makes progress, thereby avoiding problems such as deadlock and priority inversion.

Levandoski et al. introduced the *Bw-tree* [LLS13], a latch-free variant of the B-tree designed to scale on modern multi-core processors and to work efficiently with main memory and flash storage. The Bw-tree replaces in-place updates with delta records linked to existing nodes, combined with an indirection mapping table that enables threads to traverse and update the structure without latches concurrently. This design reduces contention, avoids the coherence overhead of locks, and takes advantage of processor caches in parallel environments. In addition, the authors integrated a log-structured storage manager optimized for flash memory, blurring the distinction between page and record storage. Their evaluation showed that the Bw-tree achieves high performance under concurrent workloads, demonstrating the benefits of rethinking traditional indexing techniques for new hardware.

Fraser et al.'s work [Fra04] is widely regarded as a milestone in the development of non-blocking data structures. He showed that although mutual-exclusion locks are widely used for synchronization, they scale poorly as the number of threads increases.

Even with fine-grained strategies, lock-based approaches are challenging to design correctly and remain prone to problems such as deadlock, convoying, and priority inversion. They can also degrade performance on multiprocessor systems by generating excessive cache-coherence traffic, even when protecting read-only data. To address these challenges, Fraser proposed a set of lock-free abstractions that make it easier to construct concurrent data structures. He then applied these ideas to implement and evaluate several lock-free search structures, most notably a lock-free skip list. His results showed that lock-free designs could achieve performance comparable to, and in some cases better than, advanced lock-based methods, while avoiding their inherent drawbacks. This work demonstrated that lock-free algorithms were not only feasible but also practical on modern multiprocessor hardware, helping establish them as a serious alternative to classical lock-based protocols in highly concurrent settings.

Latch-free and lock-free designs eliminate blocking and scale well under parallel workloads, but they also come with trade-offs. Implementations are often complex, rely heavily on hardware support for atomic operations, and may incur overhead from indirection or retries.

8 Conclusion

The work presented in this thesis aims to bridge the gap between GDPR requirements and the capabilities of modern key-value stores. While key-value stores are widely used for their simplicity and scalability, they only support primary key access. Therefore, they can not efficiently handle the GDPR workloads that heavily depend on secondary metadata, such as retrieving all records linked to a user, grouping data by purpose, or removing expired records.

To address this issue, we introduced an *Index Manager* within the proxy layer. It coordinates queries across a set of specialized indexes, allowing metadata-driven queries to be executed efficiently while keeping the system independent of any specific key-value store. Three index structures were implemented to handle the most common GDPR operations: a *Retention Index* using a skip list for managing expiration, a *Subject Index* with a B+ tree for user access, and a *Purpose Index* built as a sharded inverted index for grouping by purpose. Collectively, these indexes eliminate the necessity for expensive full scans and guarantee that both retrievals and modifications remain efficient under concurrent workloads.

Through experiments based on YCSB workloads, we evaluated the index structures and demonstrated that each structure is optimized for a distinct class of GDPR queries. The skip list proves effective for deletions based on retention, the B+ tree ensures rapid subject lookups, and the inverted index scales effectively for queries based on purpose. These results highlight that a single data structure can not efficiently address all GDPR requirements, but that the combination of specialized indexes balances between throughput, scalability, and memory usage.

The primary contributions of this thesis are the design and implementation of an extensible indexing layer for key-value stores, along with an experimental assessment that illustrates which types of indexes are most effective for various GDPR tasks. The complete source code, implemented in C++, has been made available to support further research and experimentation. This work focused on three index structures; however, the *Index Manager* can be extended to support additional compliance features with other structures. We can also add new types of indices to cover queries that go beyond single attributes. For example, multi-attribute indices could support queries that combine subject and purpose in a single lookup, while temporal or interval indexes could help manage records that are valid only within certain time ranges. Bloom filters, a

lightweight probabilistic structure, might also be used to speed up existence checks at the cost of a small false positive rate. Finally, evaluating the system on workloads having GDPR-specific metadata fields such as `TTL`, `USR`, `PUR`, `SHR`, and `OBJ` to each key–value record, would help validate its performance in practice.

Abbreviations

GDPR General Data Protection Regulation

KV Key–value

LSM log-structured merge

CAS Compare-and-Swap

OLC Optimistic Lock Coupling

TTL Time-To-Live

LSB Least Significant Bit

YCSB Yahoo! Cloud Serving Benchmark

RSS Resident Set Size

ART Adaptive Radix Tree

EU European Union

List of Figures

Bibliography

[Aga+21] A. Agarwal, M. George, A. Jeyaraj, and M. Schwarzkopf. "Retrofitting GDPR compliance onto legacy databases." In: *Proceedings of the VLDB Endowment* 15.4 (2021).

[BM70] R. Bayer and E. McCreight. "Organization and maintenance of large ordered indices." In: *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*. 1970, pp. 107–141.

[Din+20] J. Ding, U. F. Minhas, J. Yu, C. Wang, J. Do, Y. Li, H. Zhang, B. Chandramouli, J. Gehrke, D. Kossmann, et al. "ALEX: an updatable adaptive learned index." In: *Proceedings of the 2020 ACM SIGMOD international conference on management of data*. 2020, pp. 969–984.

[Don+21] S. Dong, A. Kryczka, Y. Jin, and M. Stumm. "Rocksdb: Evolution of development priorities in a key-value store serving large-scale applications." In: *ACM Transactions on Storage (TOS)* 17.4 (2021), pp. 1–32.

[Dsi+17] J. V. D'silva, R. Ruiz-Carrillo, C. Yu, M. Y. Ahmad, B. Kemme, Y. Ioannidis, J. Stoyanovich, and G. Orsi. "Secondary indexing techniques for key-value stores: Two rings to rule them all." In: *International Workshop On Design, Optimization, Languages and Analytical Processing of Big Data (DOLAP)*. 2017.

[Fra04] K. Fraser. *Practical lock-freedom*. Tech. rep. University of Cambridge, Computer Laboratory, 2004.

[GD11] S. Ghemawat and J. Dean. *LevelDB*. `https://github.com/google/leveldb`. Google. Accessed: 2025-09-03. 2011.

[Har01] T. L. Harris. "A pragmatic implementation of non-blocking linked-lists." In: *International Symposium on Distributed Computing*. Springer. 2001, pp. 300–314.

[Kra+18] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. "The case for learned index structures." In: *Proceedings of the 2018 international conference on management of data*. 2018, pp. 489–504.

[LC19] C. Luo and M. J. Carey. "On performance stability in LSM-based storage systems (extended version)." In: *arXiv preprint arXiv:1906.09667* (2019).

[Lei+16] V. Leis, F. Scheibner, A. Kemper, and T. Neumann. "The ART of practical synchronization." In: *Proceedings of the 12th International Workshop on Data Management on New Hardware*. 2016, pp. 1–8.

[LLS13] J. J. Levandoski, D. B. Lomet, and S. Sengupta. "The Bw-Tree: A B-tree for new hardware platforms." In: *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE. 2013, pp. 302–313.

[Lun19] I. Lunden. *UK's ICO fines British Airways a record £183M over GDPR breach that leaked data from 500,000 users*. July 2019. URL: https://techcrunch.com/2019/07/08/uks-ico-fines-british-airways-a-record-183m-over-gdpr-breach-that-leaked-data-from-500000-users/.

[LY81] P. L. Lehman and S. B. Yao. "Efficient locking for concurrent operations on B-trees." In: *ACM Transactions on Database Systems (TODS)* 6.4 (1981), pp. 650–670.

[Mic] Microsoft Docs. *Non-relational data and NoSQL*. Accessed: 2025-09-25. URL: https://learn.microsoft.com/enus/azure/architecture/data-guide/big-data/non-relational-data.

[Mic02] M. M. Michael. "High performance dynamic lock-free hash tables and list-based sets." In: *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*. SPAA '02. Winnipeg, Manitoba, Canada: Association for Computing Machinery, 2002, pp. 73–82. ISBN: 1581135297. DOI: 10.1145/564870.564881.

[ONe+96] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. "The log-structured merge-tree (LSM-tree)." In: *Acta informatica* 33.4 (1996), pp. 351–385.

[Pug90] W. Pugh. "Skip lists: a probabilistic alternative to balanced trees." In: *Communications of the ACM* 33.6 (1990), pp. 668–676.

[Reg16] P. Regulation. "Regulation (EU) 2016/679 of the European Parliament and of the Council." In: *Regulation (eu)* 679.2016 (2016), pp. 10–13.

[Sar+20] S. Sarkar, T. I. Papon, D. Staratzis, and M. Athanassoulis. "Lethe: A Tunable Delete-Aware LSM Engine." In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. SIGMOD '20. Portland, OR, USA: Association for Computing Machinery, 2020, pp. 893–908. ISBN: 9781450367356. DOI: 10.1145/3318464.3389757.

[Sat19] A. Satariano. *Google is fined $57 Million Under Europe's Data Privacy Law*. Jan. 2019. URL: https://www.nytimes.com/2019/01/21/technology/google-europe-gdpr-fine.html.

[Sha+19a] A. Shah, V. Banakar, S. Shastri, M. Wasserman, and V. Chidambaram. "Analyzing the impact of {GDPR} on storage systems." In: *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*. 2019.

[Sha+19b] S. Shastri, V. Banakar, M. Wasserman, A. Kumar, and V. Chidambaram. "Understanding and benchmarking the impact of GDPR on database systems." In: *arXiv preprint arXiv:1910.00728* (2019).

[ZM06] J. Zobel and A. Moffat. "Inverted files for text search engines." In: *ACM computing surveys (CSUR)* 38.2 (2006), 6–es.