CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Optimizing Log Data Storage and Query Performance: A Study on Industrial Applications

Master's Thesis in Computer science and engineering

JIALONG HE & EDIZ GENC

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2025

Master's Thesis 2025

Optimizing Log Data Storage and Query Performance: A Study on Industrial Applications

JIALONG HE & EDIZ GENC

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2025

Optimizing Log Data Storage and Query Performance: A Study on Industrial Applications

Supervisor: Daniel Strüber
Examiner: Francisco Gomes de Oliveira Neto

Master's Thesis 2025
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX, Gothenburg, Sweden 2025

Optimizing Log Data Storage and Query Performance: A Study on Industrial Applications

He, Jialong & Genc, Ediz

Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

This thesis covers an in-depth analysis of storage efficiency optimization and query performance improvement in highly scalable log management systems. As log volumes are experiencing exponential growth, optimization of storage as well as retrieval mechanisms has become more and more important to ensure scalability of the system and responsiveness of operations.

To that end, we systematically applied and assessed a variety of optimization strategies. In particular, the work investigated state-of-the-art compression schemes, used selective partitioning schemes taking into account inherent data distributions, and intertwined outer-layer encodings such as delta encoding, dictionary encoding, and run-length encoding. We also considered using Bloom filters on chosen attributes to facilitate more effective pruning of queries, striking a delicate balance between performance improvements versus metadata costs. All our testing was performed using a Spark-based experimental framework, utilizing industry-grade log data in Parquet columnar form as input to mimic real-world operating scenarios.

Experimental findings affirm the efficiency of the introduced methods. Among compression techniques, Zstandard (Zstd) performed best consistently, achieving high compression ratios coupled with fast decompression speed. Schema flattening along with specialized encoding patterns, further optimized compressibility by taking advantage of structural redundancies in log data. Partitioning significantly cuts down query response time by reducing the extent of data scans. Additionally, Bloom filters offered significant query speedup, especially for selective ones, having almost no negative effect on overall storage overhead.

Briefly, this work illustrates that a strategically combined set of compression, encoding, partitioning, and indexing methods can make substantial progress in both storage space efficiency, as well as query performance, in log-centric data stores. The results provide real-world advice on how to close the gap between academic optimization methods and their utilization in contemporary big data environments in industry settings.

Keywords: Log Analysis, Compression, Partitioning, Encoding, Storage, Bloom Filter, Query Performance

Acknowledgements

We would like to express our deepest gratitude to our academic supervisor, Daniel Strüber, whose continuous guidance, support, and expertise were instrumental throughout the course of this project. We also extend our sincere thanks to our examiner, Francisco Gomes de Oliveira Neto, for his thoughtful feedback and constructive evaluation. Furthermore, we are grateful to our industry supervisors, Carl A. Svensson and Alexander Andersson, for their practical insights and direction during our work at Ericsson. Finally, we thank our colleagues at Ericsson for their valuable input and collaboration, which enriched our research experience.

Jialong He & Ediz Genc, Gothenburg, June 2025

Contents

List of Figures

List of Tables

1

Introduction

The explosive growth of data created on cloud platforms, distributed services, and mobile networks has fundamentally changed how contemporary systems operate and how observability is implemented. Logs, traces, and metrics are the basic building blocks of debugging, analytics, security, and planning for the infrastructure. Among these, logs produced by infrastructure, services, and applications take a central position in offering detailed insights into the behavior of the system over time. However, the raw volume of the logs and duplicative data present in the logs lead to humongous storage overhead and analytical query performance bottlenecks.

For large deployments in the likes of Uber, Alibaba, and Ericsson, log data can readily scale up to the order of petabytes. Wang et al. [1] mention that Uber has over 10 petabytes of semi-structured logs per day with more than 60 PB of ingestion per week. These logs are mostly verbose, redundant, and schema-variant in nature.

Despite advances in scalable data infrastructure, traditional storage systems and file systems lack the cost and performance capability to support such workloads. The motivation for this work comes from the increasingly exponential discrepancy between log generation and efficiency of storage and query systems. There is a significant need for cost-effective, scalable, and robust methods of processing logs by removing redundancy and enabling fast queries.

1.1 Problem Statement

Storing and querying large quantities of log data present a number of related challenges. Firstly, log data tends to be very redundant. Redundancy occurs on a number of levels: the same log entries on multiple nodes, duplicated key-value pairs, and repeated timestamp sequences. Without deduplication or compression, storage becomes prohibitively costly. Secondly, the semi-structured form of the logs, wherein records are represented as key-value maps, causes schema diversity, making efficient querying and indexing more difficult.

Paulo and Pereira [2] discussed the complexity of building deduplication systems that are scalable and accurate. Their characterization indicates that deduplication efficiency is influenced by chunking strategy, indexing model, and processing locality.

Wang et al. [1] demonstrate that traditional search engines like Elasticsearch do not perform well with highly compressed semi-structured logs and thus motivate systems such as µSlope. In contrast, systems like Delta Lake [3] and FlatStore [4] support transactional safe storage layers and do not typically integrate compression with schema management and query optimization tightly enough.

This observation motivates the central problem of this thesis: how to design and evaluate a log storage and query architecture that systematically integrates deduplication, compression, schema optimization, and encoding strategies, providing robust performance on real-world, diverse log formats and scaling to enterprise-scale datasets.

1.2 Research Objectives

The overall objective of this thesis is to investigate and examine integrated methods for optimizing distributed environments' log data storage as well as query performance. This work uses enterprise-scale datasets provided by Ericsson, specializing in telecommunications and network infrastructure. The data sets include operational logs from large-scale distributed systems with billions of semi-structured records that have been collected of continuous use.

The choice to use data from Ericsson is strategic and critical. Synthetic datasets or academic benchmarks typically do not capture the characteristics of real production systems. While Ericsson's operational logs provide an unprecedented chance to analyze log data in its most realistic context. Such logs feature enormous volume, high redundancy, diverse formats, and rich semantics. They model the issues of log data storage management and querying at industrial scale, such as storage overhead, schema evolution, performance bottlenecks, and noise filtering.

This actual environment is a rich context in which to conduct research. It presents the redundancy, scale, diversity, and complexity characteristic of industrial log systems in today's era—features that may be difficult to model in synthetic baselines or in research prototypes. Thus, what is learned here is both of academic value and of practical use, and it offers solutions of ready applicability to industrial deployments.

Based on this background, the particular research aims of this thesis are:

1. To evaluate the structure, redundancy, and access patterns of real-world log data.

2. To develop a schema-aware pipeline to perform deduplication, encoding, and compression while ensuring queryability.

3. To compare different encoding methods (e.g., Delta, Dictionary, RLE) and determine the impact of each on compression ratio and on reading performance.

4. Examining the impact of metadata-aware optimizations like partitioning, bloom

filters, and column statistics on query performance.

1.3 Research Questions

Two overall research questions frame this work for direction. Each research question is clearly tied to research objectives and targets, identified gaps in existing solutions, adding new understanding about scalable log management using advanced methods.

RQ1: How does the integration of schema flattening, targeted encoding, and compression techniques affect storage efficiency in industrial-scale log data systems?

This research question addresses Objectives 1–3: It focuses on creating a unified optimization pipeline that combines multiple reduction techniques in a cohesive manner. While individual techniques like compression or deduplication have been separately studied, their coordinated integration with schema transformations in real-world log systems remains underexplored. By systematically analyzing and integrating these methods, this work aims to fill a gap between isolated optimizations and holistic, reproducible frameworks suitable for semi-structured, redundant datasets at industrial scale.

RQ2: How do metadata-aware optimizations, including partitioning and Bloom filter indexing, when aligned with workload characteristics, impact query performance in log analytics?

This investigation targets Objective 4. It is concerned with the query execution efficiency of metadata organizations at the storage level in distributed log processing systems. Though methods such as partitioning and filtering are widely used, their joint implications, especially in semi-structured enterprise log contexts, are not well understood. In this research, it is empirically assessed how metadata-driven methods can minimize scan overhead, enhance predicate pushdown, as well as query selectivity.

In both research questions, the focus is not merely on applying known methods but on developing integrated, scalable, and practically validated frameworks that bridge academic innovations with real-world industrial needs. The research questions are explored in this paper using Ericsson's industrial logs, which — as we discuss later in the thesis — are representative of large-scale, semi-structured log systems in enterprise environments.

1.4 Scope and Limitations

This thesis focuses on offline batch processing of log data in enterprise-scale distributed environments. The system under investigation assumes a Hadoop-Spark ecosystem using Parquet as the base storage format. Real-time stream processing is not covered, nor are NoSQL stores like Cassandra or MongoDB. Furthermore, while deduplication is studied in detail, this work is limited to lossless, block- or field-level

techniques and does not address approximate or semantic deduplication.

Also, while the quality of the query is assessed, the level of query engine optimization (such as Catalyst within Spark or Calcite within Flink) is beyond the scope of this research. Lastly, because of the sensitivity of the data, the evaluations are performed within a controlled environment on pre-approved datasets from our industrial partner, Ericsson.

1.5 Contributions

This thesis contributes the following to the study of log data storage and querying:

- A reproducible experimental framework for benchmarking storage efficiency and query performance using deduplication, encoding, and compression techniques.
- A detailed analysis of how encoding sequences and schema flattening affect columnar formats such as Parquet, using real-world datasets.
- A comprehensive evaluation of bloom filters, partitioning, and metadata indexing as methods of optimizing query pruning and decreasing latency.
- Guidelines and insights for implementing storage optimizations and deduplication within industrial settings.

1.6 Thesis Organization

The remainder of this thesis is organized as follows:

Chapter 2 – Background: Presents related background, terminologies, and necessary mechanism knowledge for this thesis.

Chapter 3 – Related Word: Presents an in-depth review of the literature on deduplication, compression, schema-aware storage systems, and real-time log querying. It also provides theoretical underpinnings, taxonomies, and prior work analysis of both academic and industrial systems.

Chapter 4 – System Design and Optimization Framework: Introduces the developed system architecture and layered optimization framework. It outlines the design goals, component responsibilities, encoding and layout strategies, and the workflow used to transform raw log data into a query-efficient structure.

Chapter 5 – Methods: Explains the experimental configuration, system architecture, parameter settings, and benchmarking procedure. It details the encoding methods, pipelines for deduplication, and storage configurations adopted throughout the evaluation.

Chapter 6 – Results: Quantitative storage efficiency results, compression ratio results, query latency results, and the effects of schema transformations are presented. Plots and statistical analysis breakdowns are included to support the conclusions.

Chapter 7 – Discussion: Offers explanation and interpretation of the results, consideration of the implications and the limitations of those results, and directions for follow-up studies.

Chapter 8 – Conclusion: Summary of main results and directions for future work.

2

Background

Efficient storage and fast querying are critical for modern big-data log management systems. Storage footprints need to be minimized and query latencies optimized as datasets expand. The underlying concepts, storage methods, and related work upon which our solution is based are presented within this chapter.

2.1 Overview of Storage and Query Optimization

Columnar structures and compression are increasingly the basis of large-scale storage systems optimized for analytics [5]. Columnar storage enables reads on the columns of interest only, while compression minimizes the cost of disk I/O and network transfers. Query optimization also uses metadata (e.g., partitioning, indexing) to keep scanned data minimal and accelerate results retrieval [6].

2.2 Key Terminology and Mechanisms

To provide clarity and consistency, this section establishes fundamental definitions and mechanisms for concepts relevant to data storage and processing that underpin the methods utilized throughout this research.

Partitioning divides a dataset into parts based on the values of selected columns, allowing queries to skip irrelevant partitions and thus improving query efficiency [7].

Hash bucketing is for partitioning the data into a number of segments, based on the output of a hash function which is applied to specific column values to enhances the efficiency of data retrieval [8].

Parquet is a column-oriented data storage format designed to store each column's data separately rather than grouping values by rows to allow for more effective compression. Parquet makes it possible to read only the specific columns needed for that specific task provided by authors, thus improving performance and reducing resource usage [3].

Columnar storage organizes data so that values from the same column are stored on disk to contrast with traditional row-based storage and also offers notable advan-

tages for analytical workloads for the reason that data from the same column tends to have similar characteristics to be able to ease compression algorithms to perform more efficiently. In addition, columnar storage allows queries to skip irrelevant columns entirely, reducing disk I/O and accelerating execution time. The columnar storage approach is beneficial in frequent read operations, such as log analysis [5].

Cardinality is the number of unique values within a column [6]. Low-cardinality columns generally have a limited number of unique values compared to the total number of records. For instance, a "status" field that contains values like "success," "failure," or "pending." On the other hand, a high-cardinality column contains a great many unique values, like a "user_id" or "transaction_id" column where each record entry tends to be unique.

Encoding techniques are integrated to reduce the storage footprint of columnar data by converting values into more space-efficient representations. Several encoding strategies we employed to achieve are:
- **Run-Length Encoding (RLE)** compresses sequences of repeated values by storing the value once along with a count of its occurrences, making it especially effective for columns with low cardinality [6].
- **Dictionary Encoding** replaces frequently occurring values with shorter integer codes, significantly reducing storage space and improving processing speed, particularly in categorical data columns [6].
- **Delta Encoding** captures the difference between consecutive values rather than storing each value outright for the reason that this is especially efficient for numerical sequences with predictable patterns, such as timestamps.

Bloom filters are probabilistic data structures that help quickly determine whether a value may exist in a dataset, with a small chance of false positives and no false negatives. By enabling bloom filters on key columns, row groups that definitely do not contain the target entity can be skipped entirely during query execution. [9].

Hadoop Distributed File System (HDFS) is a scalable, fault-tolerant storage system designed for distributed environments [10]. It supports most big data platforms by offering dependable storage of information within clusters of commodity hardware.

Hive is a data warehouse system built on top of Hadoop, offering a SQL-like interface (HiveQL) for querying and managing large datasets stored in HDFS [3]. It abstracts the work of programming with MapReduce and allows for complex querying using familiar relational constructs. It serves as the hub of most big data landscapes, offering the metadata and planning features required for interacting with storage layouts such as Parquet.

2.2.1 Encoding Fundamentals

Columnar storage in the modern era generally adopts a two-stage procedure [6]. Lightweight encoding methods like bit-packed encoding are first used to eliminate

redundancy and decrease entropy within each column. It prepares the information for the second stage, where heavyweight compression methods like Snappy further compress the encoded information, decreasing the storage footprint and enhancing the effectiveness of the questions asked.

The effectiveness of this two-stage compression relies on the regularity and organization of the input data. In implementations such as Apache Parquet, the internal encoding strategies are optimally decided on the basis of column statistics and value distributions while the file gets written. Users have no significant direct control over these internal strategies.

But by using **external encoding strategies** like Delta Encoding prior to storage writeout, we get indirect control over the redundancy and structure of the data. Well-organized external layouts enable the internal encoding of Parquet to work more optimally, leading to more compact compression and improved query execution.

Therefore, external encoding is a key step of preparation that facilitates the quality of the internal encoding process.

2.2.2 Metadata Structures

Metadata structures like min/max statistics and Bloom filters at the row group level allow the skipping of unrelated blocks by the query engines [9]. Predicate pushdown also minimizes the amount of data scanned by applying the query filters earlier while reading the file [3].

2.3 Introduction to Log Data Storage Challenges

The data deluge in today's distributed systems has made log data a key resource for operational visibility, debugging, security auditing [11], and system optimization. Logs record fine-grained events from applications, network equipment, cloud environments, and software services, creating a priceless record of activities amenable to retrospective analysis. As companies scale, though, the volume, variety, and rate of log data have become staggering, creating significant storage management and real-time querying challenges.

One large-scale business can produce tens to hundreds of terabytes of logs each day, from structured telemetry events to semi-structured map records and unstructured text blobs. Handling this flood of data poses several technical hurdles. Storage costs increase proportionally to data volume, while ensuring low-latency query performance across enormous repositories of logs grows increasingly challenging. Traditional relational databases, designed mostly for structured and transactional workloads, tend to fall behind under data ingestion and query patterns common in log datasets [1]. Additionally, modern logs' semi-structured, heterogeneous nature makes indexing and query designs more challenging, further deepening performance

bottlenecks.

Solutions such as "μSlope" [1] and Delta Lake [3] have risen to meet some of these issues by incorporating schema evolution, compression, and indexing. They require a reevaluation of storage architectures, encodings, and retrieval methods by adapting to the distinct nature of log data. The following sections present the fundamental concepts required to truly understand modern methods of log storage optimization.

2.4 Foundations of Data Redundancy and Deduplication

Data redundancy is a fundamental property of the log system. Heartbeat signals, status messages, and error messages frequently repeat with minimal or without any variation over time [2]. The redundancy results in inefficient storage and impedes query performance as redundant information needs to be scanned unnecessarily.

Deduplication is a method created to remove redundant data by recognizing identical or very alike pieces of data and substituting these with pointers towards a stored unique copy. Deduplication may take place on any of these levels: file level, block level, and byte level. File-level deduplication stores the repeated files a single time, while block-level deduplication breaks up the files into small pieces and removes the redundant blocks [12].

The level of granularity of deduplication greatly influences system trade-offs between metadata cost and storage gain. While fine-grained deduplication results in higher storage gain at the cost of higher indexing costs and lookups, distributed environments also have the problem of dealing with factors such as locality of the data, scalability, and consistency across multiple nodes.

A key idea in deduplication systems is chunking. Static chunking partitions the data into fixed-size blocks, while dynamic chunking, like content-defined chunking, varies block borders according to the content of the data and thus results in increased rates of redundancy detection. Chunking algorithms need to achieve a balance between computational overhead and effectiveness of redundancy capture.

2.5 Compression Techniques for Large-Scale Log Data

Compression minimizes the physical storage footprint by representing the data more densely. Compression uses statistical patterns within the data to provide space savings, while unlike deduplication, it eradicates entire duplications.

Lossless compression methods guarantee that the original data will be reconstructible from the compressed data and thus are suitable for storing logs where there is a need for fidelity of the original data. Such compression methods include:

Gzip, using sliding-window compression of LZ77 together with Huffman coding, achieves good compression rates at very high computational expenses on decompression. It is good for storage for archiving purposes but not ideal for low-latency querying.

Snappy, developed by Google, prioritizes speed over compression ratio. It is particularly effective in interactive query workloads where decompression latency can become a bottleneck.

Zstandard (Zstd) finds a balance between compression ratio and decompression speed and offers multiple compression levels and dictionary-based compression. Its characteristics favor both hot and cold data tiers.

Besides general-purpose compression methods, specialized encoding methods optimized for the data type can yield additional gains. For frequent strings, dictionary encoding reduces the codes to smaller lengths. For repeated values that appear serially as a group, Run-Length Encoding (RLE) encodes the runs of values and is useful for status columns or category logs. For temporal locality where the values do not differ drastically between consecutive ones (as in the case of a set of timestamps), delta encoding encodes the difference between consecutive values.

Storage formats like Parquet use a mix of these methods internally and choose encoding strategies on a column-by-column basis selectively, thus accommodating the statistical characteristics of each attribute.

2.6 Partitioning, Indexing, and Access Optimization

Partitioning is the process of subdividing datasets into discrete units based on partitioning keys, allowing query engines to prune irrelevant partitions during query execution [7]. Partitioning minimizes the volume of the scanned data and results in significant improvement in the performance of the query as well as resource utilization.

In log storage systems, partitioning is commonly performed on fields with strong temporal or categorical correlation, such as event timestamps, server identifiers, or application types [13]. Partitioning by month or by application region, for instance, can confine query scans to the respective subsets.

Unless partitioning decisions are made carefully, the skews between the partitions cause unevenly sized partitions and resource imbalances as well as query hotspots. Dynamic partitioning, bucketing, and composite partitioning approaches have been suggested as methods of addressing the skews.

Bucketing, or hashing, further subdivides partitions by hashing a field into a fixed number of buckets, enabling more efficient joins and predicate push-downs[14].

While bucketing involves maintenance overhead and careful management, it can have a substantial impact on large join-heavy workloads.

Indexing augments partitioning by establishing auxiliary structures that speed up data retrieval. Standard indexes like the B-tree or bitmap index are not always viable at large log scales because they have maintenance overhead. Light structures like Bloom filters and min/max statistics are common. Bloom filters provide the ability to have fast probable membership tests and thus eliminate non-matching partitions. Min/max statistics summarize the value range within a data block, enabling data skipping during range queries [9].

2.7 Schema Optimization and Encoding Strategies

The raw logs usually have semi-structured features drawn from varied and changing sources. Flexible serialization types such as JSON, Avro, and Protocol Buffers are widely used in order to handle variability of the schema, although this comes at the cost of significant storage overhead and query inefficiency.

Schema optimization includes reconstructing data representation to maximize space and access efficiency. Flattening nested structures into flat relational schemas simplifies data access and allows better exploitation of columnar storage benefits. By expanding maps, arrays, and structs into individual columns, query engines can selectively scan only the required fields, significantly improving IO efficiency.

Normalization of the schema, when repeating or redundant columns are broken into referential tables, also minimizes the duplication of data and enhances compression efficiency. Normalization should be balanced against the necessity for denormalized representations that are conducive to analytical workloads.

Encoding methods like dictionary encoding, delta encoding, RLE, and bit-packing can be selectively used on columns depending on the statistical characteristics of the columns [6]. For example, columns that have a small number of unique values are most suited for dictionary encoding, while sequential date-time columns are perfect for delta encoding.

An important consideration is encoding ordering. Applying dictionary encoding before RLE versus applying RLE before dictionary encoding can lead to different compression outcomes depending on the data characteristics of the data. Optimal encoding pipelines should be determined experimentally for each dataset in order to provide maximum space gains and read efficiency.

Schema evolution, the support for seamless additions, removals, or changes of fields over the passage of time, is another challenge. Backwards compatibility and adaptive schema management platforms are required for the long-term viability of the data within evolving application landscapes.

2.8 Columnar Storage Formats and Their Role

Traditional row-oriented storage structures hold complete records sequentially, while they are not ideal for analytics workloads where typically a fraction of the fields are accessed. Columnar storage structures keep each column separate, support individual access of columns, have improved compression rates, and support efficient vector-driven computation.

Apache Parquet and **Apache ORC** are the two leading open-source columnar formats designed for big data analytics. Both provide support for rich types of data, nested data structures, predicate pushdowns, and complex compression schemes.

Parquet arranges data into row groups containing a column chunk for each column. Inside each column chunk, the data gets arranged as pages. Each page can be individually compressed and may carry column statistics, enabling efficient filtering and pruning during query execution [5].

Parquet's modularity enables varying compression and encoding strategies on a column and a page basis to take advantage of each attribute's unique characteristics. For example, a column may employ Zstd compression and RLE encoding, while another may employ Snappy compression and dictionary encoding.

Columnar storage, paired with effective schema optimization, significantly enhances read throughput for analytic queries by keeping IO at a minimum, lowering memory pressure, and allowing for CPU cache-friendly access patterns. Write workloads are prone to being slowed down by the cost of the management of pages and metadata.

In log storage cases where appends and bulk loads are the norm, thoughtful batching and page-sizing strategies have to be adopted to reconcile write efficiency and later query effectiveness.

2.9 Query Efficiency in Distributed Log Systems

Querying big-data logs involves addressing the issues presented by the scale of data, semi-structured data types, and heterogeneity of access patterns. Platforms like Apache Spark [3].

Spark SQL optimizes the execution of the query using its Catalyst optimizer by producing efficient execution plans that take advantage of predicate pushdowns, partition pruning, vector reads, and broadcast joins where applicable. Query performance depends a great deal on physical data layout, availability of metadata, and tuning of the system.

Partition pruning enables query engines to skip entire partitions that do not meet the conditions of a query. Column pruning permits the reading of columns required for a query. Predicate pushdown facilitates the execution of filters as close as possible to the storage level to lessen the movement of data and computation.

Statistical optimization, where query planners make use of histograms, min/max values, and cardinality estimates, also improves the quality of the query plan. Statistics gathering and maintenance do take extra overhead and need to be weighed against the operational restrictions.

3

Related Work

This chapter surveys past systems and work focusing on effective storage and querying of large-scale, semi-structured log, then synthesizes these to determine missing gaps and design our work around these gaps.

3.1 Introduction to Technological Landscape

This section discusses past approaches and methods that have tackled similar problems to our work, specifically around storage and log optimization. It provides an overview of key technologies and methodologies, such as schema-aware columnar storage, log-oriented storage engines, real-time analytics, and lifecycle management, that have shaped the field. The following sections will explain that work, setting the foundation for identifying the gaps our research aims to fill.

3.1.1 Schema-Aware Columnar Storage and Encoding

Abadi et al.'s C-Store unifies RLE and dictionary coding into a column store, reducing I/O by up to 6×, along with significant scan speed improvements, by directly accessing compressed data [6]. Armbrust et al.'s Delta Lake incorporates ACID transactions and partition pruning driven by metadata into Parquet, allowing subsecond planning on petabyte-scale tables through file-level statistics [3]

3.1.2 Log-Oriented Storage Engines

Chen et al.'s FlatStore flattens semi-structured records into a columnar layout on persistent memory and maintains a global in-memory index, delivering low write amplification and high read throughput on NVDIMMs [4]. Wang et al.'s μSlope builds a merged parse tree and schema map over JSON logs, grouping records by shared structure for up to 186:1 compression and direct search on compressed data [1].

3.1.3 Real-Time Analytics Databases for Logs

Apache Druid separates deep storage from compute, using columnar segments and bitmap indexes to support real-time ingestion and sub-second OLAP queries over event streams [5]. Elasticsearch employs inverted indexes, doc-values, and tiered

hot-warm-cold architectures; recent 8.x releases add 8-bit quantized vectors for approximate k-NN, cutting vector storage by 4× with negligible accuracy loss. Grafana Loki indexes only metadata "labels" and stores entries as compressed chunks in object storage, enabling scalable retention with LogQL push-down filtering at query time.

3.1.4 Columnar TTL and Lifecycle Optimization

ClickHouse provides per-column TTL rules, where data are deleted or rolled up based on a specified interval, allowing hot-warm-cold storage management without performing complete compactions. GreptimeDB supports SQL-style TTL policies by table or column, making it easier to manage lifecycles of time-series and log tables.

3.1.5 Bit-Vector and Bitmap Indexing

O'Neil and Quass' variant indexes show how bit-sliced methods can enable faster group- and top-k-queries by directly querying compressed bit-vectors [15]. Golynski et al. present asymptotically optimal sparse bit vector rank/select indexes, enabling compact storage of binary vectors in contemporary search engines [16].

3.2 Prior Work Summary

Research in storage optimization can be categorized into several domains.

Deduplication systems evolved from simple backup systems to complex, real-time systems. Earlier research by Min et al. [17] presents context-aware chunking algorithms that enhance the uniqueness of fingerprints and minimize false positives. Collaborative Partial Indexing (CPI) is proposed by Wei et al. [18], which improves the throughput of lookups by dividing fingerprint indexes among multiple data structures. Decentralized deduplication is used by the DEDIS system [19] for primary storage in cloud setups to allow global deduplication among virtual machine snapshots without depending upon copy-on-write techniques. At the edge, Cheng et al. [20] propose LOFS, which integrates Bloom filters, Locality-Sensitive Hashing (LSH), and bucketization to enable lightweight, online deduplication on resource-constrained devices.

Compression-oriented systems have demonstrated substantial gains in both storage and query performance. µSlope achieves compression ratios exceeding 186:1 by using schema-based grouping and entropy-aware encodings [1]. Fei et al. [21] demonstrate that delta encoding is particularly effective for timestamp-dense enterprise logs.

Partitioning and schema optimization are also addressed in systems such as AutoPart [13], which applies vertical partitioning of scientific databases driven by the workload. Schism [14] and Horticulture [7] use graph-based partitioning and cost-based modeling to reduce distributed transactions in OLTP settings. Aulbach et al.

[22] propose schema-mapping techniques for multi-tenant SaaS systems, balancing flexibility and performance via chunk-based storage mappings.

Modern storage engines like FlatStore [4] decouple indexing from data storage to make the most of persistent memory characteristics. Parquet [3] is improved upon with the additions of ACID compliance, time travel, and schema evolution to form a foundation for lakehouse architectures. Both systems integrate encoding, compression, and metadata indexing into first-class components.

Zhang et al. [11] offer a wide survey of log parsing and consider 17 open-source parsers' design and performance. They call for reliable template-based solutions that can adapt to the diversity of current log sources, with particular importance for anomaly detection and root cause investigation applications.

3.3 Gap Identification

Most current systems have addressed the log data storage and query performance problems from isolated perspectives, either compressing, being schema-aware, or indexing, but seldom providing an integrated and configurable system for covering the entire stack of optimization. Systems like FlatStore, µSlope, and Delta Lake each have valuable mechanisms but are prone to specialize in narrow areas of the optimization pipeline. Our solution seeks to fill this gap and provide an integrated and reproducible setup that combines schema flattening, control over external encoding, dynamic choice of compression, metadata-aware indexing (like Bloom filters), and partition-based querying pruning. Table 3.1 shows our system along with major existing methods, comparing the relative effectiveness of optimization strategies.

Table 3.1: Comparison of Optimization Features Across Systems

System	Schema Transformation	Compression	Indexing	Encoding Control	Query Optimization
FlatStore	Yes	Yes	No	No	Partial
µSlope	Yes	Yes	No	No	No
Delta Lake	No	Yes	Yes	No	Yes
Parquet Native	No	Yes	Optional	No	Yes
Elasticsearch	No	No	Yes	No	Yes
This Work	Yes	Yes	Yes	Yes	Yes

Existing research falls behind on some of the most significant gap areas. Perhaps most notably missing is internal encoding order and its impact on compression and search performance. For example, ordering dictionary encoding prior to delta encoding will result in different outcomes than ordering them in reverse, yet few systems offer tunability or visibility into such situations.

A second key challenge is the lack of utilization of metadata. While formats such as Parquet allow for the tracking of min/max statistics and predicate pushdown, there is limited empirical study of the performance impact of omitting metadata in dynamic log settings. In addition, most benchmark studies use synthetic datasets such as TPC-H that do not necessarily reflect the irregularities in structure or the

schema volatility of real-world log data.

Edge environments present unique challenges, such as constrained compute and memory resources. While some systems like LOFS [20] address these limitations, edge-specific deduplication and compression strategies remain underexplored in comparison to their cloud counterparts.

Finally, the lack of end-to-end systems that combine deduplication, compression, encoding, schema optimization, and query execution in a unified framework is a structural gap in the current study area.

3.4 Justification for This Research

This thesis fills the above gaps by creating a single storage optimization pipeline that is purpose-built for enterprise-scale logs. Differing from existing works on compression and deduplication in isolation, this work combines schema reorganization, encoding order analysis, and metadata improvement in order to enhance storage efficiency and analytical performance.

The research is based on realistic datasets from our industrial partner Ericsson and thus provides practical applicability and avoids the limitations of simulated benchmarks. In addition to that, all experiments include careful documentation of environment parameters and configurations and datasets to allow for reproducibility.

The research also explores internal encoding order and metadata use in detail and provides insight on empirical evidence effects for querying performance. By applying optimizing strategies to edge environments and focusing on schema-aware approaches, this paper addresses pressing requirements in both research and industry areas.

4

System Design and Optimization Framework

4.1 Overview of the Developed Framework

The five-layered framework of the proposed system consists of ingestion and preprocessing, data reduction, storage organization, indexing and metadata management, and optimizing the execution of queries.

4.2 Workflow of the Optimization Framework

As shown in Figure 4.1, the workflow of the optimization framework follows a layered approach. Raw log ingestion constitutes the entry point, where unstructured or semi-structured data is parsed and standardized. Preprocessing ensures consistency in data formats and prepares the records for reduction. In the data reduction layer, deduplication eliminates redundant records based on content-defined chunking, while columnar encoding techniques such as delta encoding, dictionary encoding, and run-length encoding are selectively applied based on attribute characteristics.

After reduction, the storage organization layer applies dynamic partitioning techniques. Partitioning is guided by cardinality analysis of fields like date and status and is applied to high-cardinality fields like user IDs to balance data distribution.

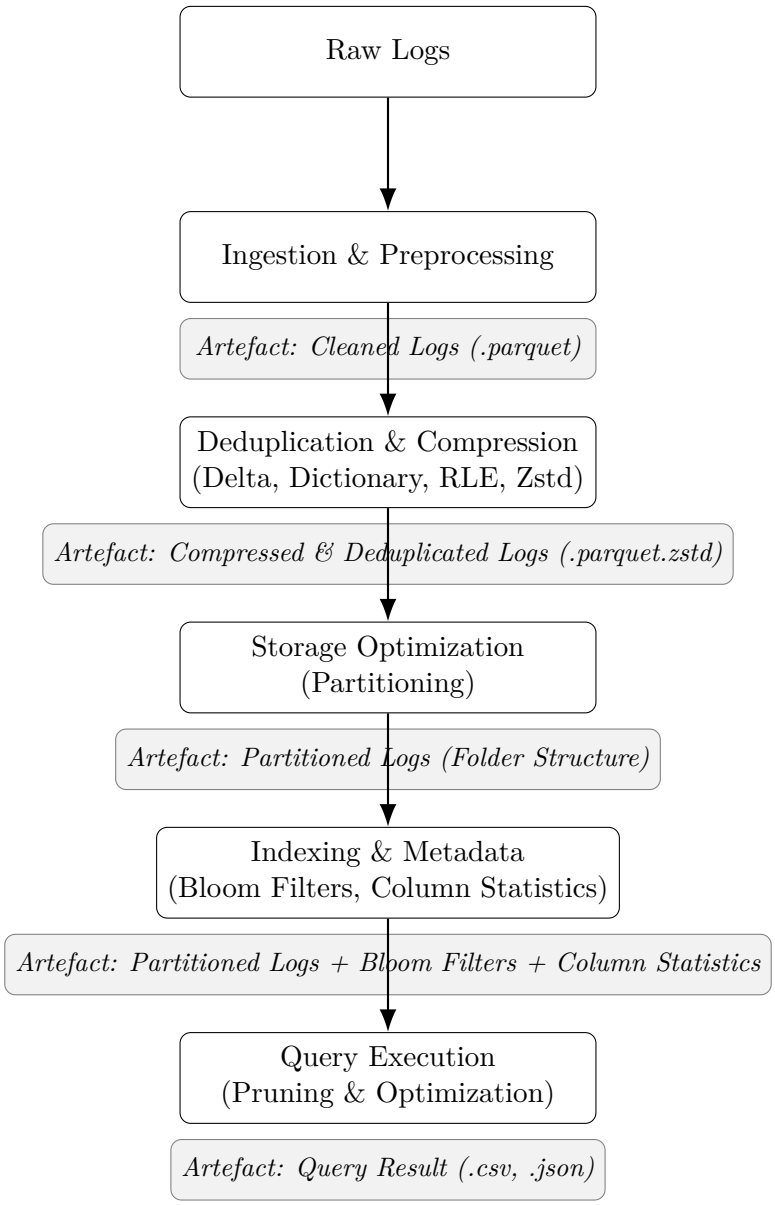

Figure 4.1: Enhanced Workflow of the Log Storage and Query Optimization Framework, including artefacts produced at each step.

Query performance is also improved through indexing and metadata management layers. Frequently accessed fields are represented by bloom filters so that unwanted parts of the data can be easily dropped during scan operations. Column statistics with minimum and maximum values are also gathered to assist in predicate pushdown and data skipping during the execution of queries.

Lastly, the execution layer of the query takes advantage of these optimizations to reduce input/output operations, the cost of shuffling, and computation overhead. The framework exploits Apache Spark's Catalyst optimizer and vectorized Parquet readers to fully leverage the structured benefits of the organized data.

4.3 Component Descriptions

The ingest and preprocess component parses diverse input formats into a common schema. Structurally diverse logs are unfolded to tabular formats so that the embedded attributes are unraveled into separate columns. Missing fields, type mismatches, and corrupted data are addressed in this step to ensure high data quality.

De-duplication is conducted in the data reduction layer with the help of algorithms that eliminate duplicate records upon detection. Encoding is subsequently done on a column-by-column basis.

The storage organization layer focuses on reorganizing the optimized data into partitioned Parquet files. During the partitioning of data, records are grouped based on columns that tend to have only a few distinct values, such as dates or status labels. Partitioning helps speed up queries by letting the system skip over chunks of data that aren't relevant, as if you are only looking at one day's logs, it will not need to scan the rest.

The indexing and metadata management layer is for data structures that help the query engines prune out irrelevant data in advance. Bloom filters are created on certain columns to allow rapid existence checks with little storage overhead. Column statistics such as minimum and maximum values are stored in the Parquet metadata as well. These statistics support predicate pushdown so that the query engines can avoid entire row groups that do not meet the conditions of a query.

When a query runs, the SQL engine — for example, Spark — tries to follow the most efficient path it can. It uses partition pruning, column pruning, and sometimes bloom filters to avoid scanning too much data. These optimizations are applied on the fly, depending on what the query needs. Also, with Parquet files being column-based, the system can read chunks of values all at once using something called vectorized reading. This makes things faster since the CPU does not have to handle each value one by one.

4.4 Workflow of the Framework

The process starts with parsing, cleaning, and flattening raw log data to a uniform schema. This is followed by de-duplication algorithms that remove duplicate records to eliminate redundancy. Next, selective column encoding and compression minimize the size of data further. Transformed records are then stored in partitioned Parquet files with metadata and indexing structures in place to enable efficient querying. Subsequently, analytical queries are run against the optimized data with reduced I/O operations and fast access paths.

A number of key design decisions are done according to the framework's workflow. Schema flattening is put up front to get the most out of columnar storage and compression. Partitioning is done on low-cardinality attributes to limit the number of

partitions to be read in querying. Bloom filter is used in the case of frequently searched columns, where it is acceptable to have some false positive matches and where eliminating unwanted file scanning will be more valued for performance. Column statistics stored in metadata help to achieve efficient predicate push-down.

5

Methods

5.1 Research Methodology and Question Alignment

The methodology aims to combine aspects of **design science research** (DSR) and **empirical evaluation** so that both assessment of practical, effective solutions and design take place.

- **RQ1** focuses on identifying and designing storage and query optimization methods for log data. This forms the **design science** part of our effort, where existing methods in both academia and industry are combined, transformed, and integrated into a coherent framework best suited to industrial-scale systems.

- **RQ2** assesses how effective these improvements are in actual scenarios. It is addressed by an **empirical evaluation** approach in terms of benchmarking and quantitative analysis to ensure the improvements in performance.

To address **RQ1**, which focuses on designing and integrating storage optimization techniques, we followed a structured design science approach. This involved identifying key pain points in log storage through analysis of real-world datasets, surveying relevant academic and industrial methods, and engineering a unified optimization framework. Techniques such as schema flattening deduplication, dictionary and delta encoding were selected and adapted based on the specific characteristics of the dataset. Design iterations were guided by feasibility constraints, theoretical insights, and expected performance trade-offs.

To answer **RQ2**, we perform a controlled, empirical analysis. We analyze how various storage and query optimization techniques drive log data performance. We implement our experimental setup by integrating:

- Real-world log data supplied by our industry partner.

- A Spark-based benchmarking framework for repeatable testing.

- Comparative analysis of encoding, compression, partitioning, and schema op-

timization techniques.

- Quantitative measurements of storage, query latency.

These empirical methods are used in a realistic but controlled environment, allowing generalizable knowledge for industrial-scale log management systems.

5.2 Environment Setup and Configuration

Our experiments were conducted in a high-capacity computing environment, consisting of fundamental tools like Python, PySpark, Spark, and Hadoop. Since our focus is on the impact of storage formats and query methods rather than hardware limitations, we ensured that resource constraints did not interfere with results.

We fine-tuned the Spark session configuration to make the most of available memory and computing resources while maintaining adaptability across varying workloads. The following table summarizes the Spark parameters used:

Parameter	Value
spark.driver.memory	50g
spark.executor.cores	7
spark.executor.memory	55g
spark.dynamicAllocation.enabled	true
spark.dynamicAllocation.minExecutors	1
spark.dynamicAllocation.maxExecutors	4
spark.shuffle.memoryFraction	0.5
spark.shuffle.service.enabled	true
spark.driver.maxResultSize	12gb
spark.unsafe.sorter.spill.read.ahead.enabled	false

Table 5.1: Spark Session Configuration

These settings were chosen to maximize memory efficiency and let Spark dynamically adjust resource use depending on the workload. This helps us avoid running out of memory even during heavy operations.

With enough resources at our disposal, these configurations enable us to experiment with how storage forms and query methods fare in optimal circumstances. We configure things that way to separate the influence of operations from being affected by hardware constraints. It focuses our studies but remains realistic and close to real-world applications.

5.3 Data Feature Analysis

5.3.1 Data Feature Modeling

In industrial environments, log data typically exhibits strong semi-structured characteristics, often appearing in nested key–value formats, which leads to highly diverse data schemas. For instance, logs produced by different devices or services may contain different combinations of fields, resulting in irregular and evolving structures. At the same time, log data is highly redundant: not only are there many identical log entries, but there is also strong temporal correlation, such as sequential timestamps or repeated status values. Besides, lookups on high-cardinality fields (such as device IDs) are also tricky. These characteristics impose strong requirements on both storage and query performance.

5.3.2 Data Source and Scale

The datasets were collected from the log systems of a base station cluster over the past 18 months from our industrial partner Ericsson. The original data volume exceeded 10TB. After data desensitization and filtering, two experimental datasets of sizes 11GB and 4.6GB were retained.

It is important to note that these two datasets serve as representative, desensitized samples for illustration and evaluation purposes in this thesis. Although the main experiments are based on the 11GB and 4.6GB datasets, we also conduct smaller-scale sample testing on subsets. The following examples, Log1 and Log2, are such sample datasets.

Both datasets present distinct characteristics. They exhibit redundancy and varying degrees of relevance, posing challenges for efficient storage and retrieval.

5.3.3 Constraints Definition

This work focuses exclusively on offline batch-processing scenarios and does not address real-time streaming. All datasets are stored in Parquet columnar format to fully exploit its superior compression and query-pruning capabilities.

To comply with data confidentiality requirements, all column names used in this study have been anonymized. The renaming is consistent and semantically neutral, ensuring that the structural and statistical properties of the data remain intact. This allows us to analyze representative datasets without violating privacy constraints.

5.3.4 Dataset Characteristics

The default compressed size throughout the tables in this paper refers to the native data format using `.snappy` compression.

5.3.4.1 Log1

Column Name	Uncompressed Size (bytes)	Compressed Size (bytes)	Compression (%)	Size % (Total)
name	7048	2774	60.64	3.54
source	21190	4876	76.99	10.66
event_time	81758	56396	31.02	41.11
key	37	39	-5.41	0.02
value	37	39	-5.41	0.02
process_time	84674	43511	48.61	42.58
id	4081	1971	51.70	2.05
flag	30	32	-6.67	0.02
Total	198855	109638	44.87	100

Table 5.2: Summary of Log1 Dataset

Table 5.2 introduces the basic storage information of Log1. There is also a mix of low-cardinality and high-cardinality columns with skewed-value distribution in this dataset.

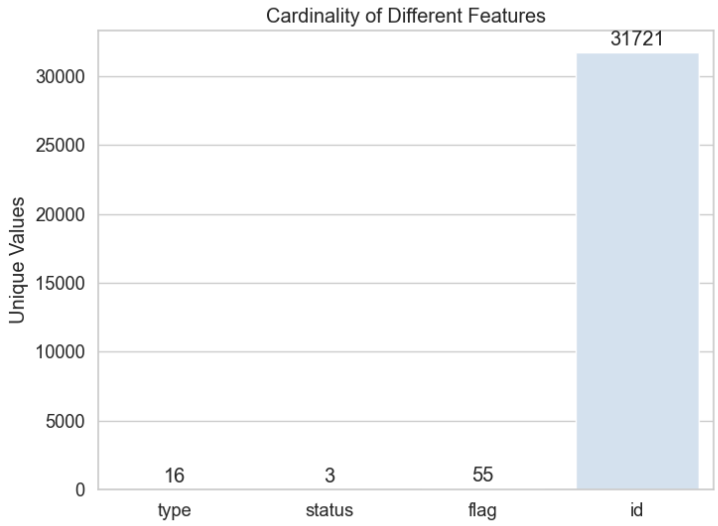

Figure 5.1: Example Column Cardinalities in Log_1 Dataset

The dataset has various levels of cardinality, as shown in Figure 5.1. The `type` has **16 unique values**, whereas the `status` has only **3** unique values. In contrast, the `id` has as many as **31,721** unique values, which implies the existence of a large number of log sources. It is important to point out that at present, the `type` and `status` are partitioned indices and are stored as superdirectory names. Consequently, they are not presented in the preceding table. This chart merely includes these 4 columns with the purpose of illustrating the primary features of this dataset.

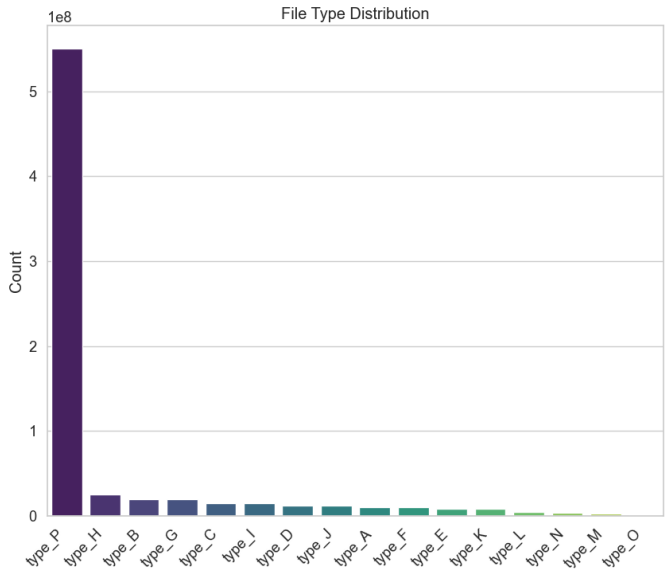

Figure 5.2: Example Value Distribution of Type

As shown in Figure 5.2, the distribution of values in the `type` field of the dataset is as follows:

The value `type_P` is overwhelmingly dominant in the distribution, with a frequency of occurrence far higher than any other file type. In contrast, the distribution of the remaining values is much sparser, with each value occurring only a small proportion of the time. Overall, the distribution is highly unbalanced, with a clear concentration on a single dominant category.

5.3.4.2 Log2

This dataset mainly consists of low-cardinality indexed columns and an attribute field with arbitrary key-value pairs.

Column Name	Uncompressed Size (bytes)	Compressed Size (bytes)	Compression (%)	Size % (Total)
service_name	2848	2756	3.23	0.01
timestamp	93	92	1.08	0.00
context	735	735	0.00	0.00
category	27346	11320	58.60	0.11
dependency	10933187	949056	91.32	45.32
key	1583984	369744	76.66	6.57
value	11575413	2212194	80.89	47.98
Total	24123606	3545897	85.30	100

Table 5.3: Summary of Log2 Dataset

As shown in Table 5.3, currently, the `dependency`, `key` and `value` columns, which are key-value pair (KV) structures, dominate the storage size in Log2. Still, these

KV formats do not utilize Parquet's built-in compression and encoding capabilities to the fullest extent.

Parquet is extremely efficient at compressing data with lots of repeating or null values and reading selectively from a column to maximize query efficiency. Hence, reorganizing the data from KV into a flat structure can indeed improve storage and access speed significantly.

ID	Attributes
1	{k1: v1, k2: v2, k3: v3}
2	{k2: v2, k4: v4}
3	{k4: v4}

Table 5.4: Example of Original Key-Value Format

Table 5.4 shows the typical and current structure of the key-value pair Log2 dataset.

ID	k1	k2	k3	k4
1	v1	v2	v3	null
2	null	v2	null	v4
3	null	null	null	v4

Table 5.5: Flattened Key-Value Data

Ideally, the KV pairs should be flattened into a structured columnar format as shown in Table 5.5.

5.4 Experimental Evaluation Framework

This section provides a thorough description of the experimental evaluation methodology employed in this work to provide a standardized and reproducible evaluation approach. The experimental methodology begins by first creating a baseline (default Parquet configuration) using a controlled environment and then incrementally adding optimization methods, including deduplication, encoding, partitioning, flattening, and bloom filters, to enable comparative evaluation.

5.4.1 Datasets

- **Data Source**: Industrial partner-provided base station cluster desensitized logs, featuring fields such as timestamps, types, IDs, and key-value pairs.
- **Dataset Size**: Main experimental data are approximately 11GB and 4.6GB, referred to as Log1 and Log2, respectively. Preliminary validations are performed using small test sets.

5.4.2 Comparative Experiment Design

The comparative experiments fall into two broad categories:

- **Baseline Scheme**: Employ Spark default settings and Parquet default configuration (Snappy compression, non-optimized partitioning, no outer encoding).
- **Optimized Schemes**: Apply the following techniques separately or in combination:
    1. Key-Value Flattening: Transform nested key-value fields into individual columns;
    2. Deduplication Strategy: Remove duplicate records to reduce data redundancy;
    3. Custom Encoding: Use encoding methods such as dictionary encoding, run-length encoding (RLE), and delta encoding to improve compressibility based on field characteristics;
    4. Compression Algorithm Comparison: Evaluate Zstandard, Snappy, and Gzip over different data layouts;
    5. Partitioning: Divide data by timestamp or business dimensions to improve query pruning;
    6. Bloom Filters: Use Bloom filters to accelerate filtering queries and reduce unnecessary scanning.

Each optimization strategy has its effectiveness measured by comparing its metrics in identical test scenarios.

5.4.3 Evaluation Metrics

The primary metrics used in this study include:

- **Storage Size (Compression Ratio)**: Ratio of original data size to compressed data size;
- **Average Query Latency**: Average response time over multiple filtering queries;

Analysis was complemented using a Spark-based benchmarking framework along with PySpark for data transformation and query execution. Metrics were gathered using our own scripts as well as Spark-provided instrumentation tools. Execution was tracked through Spark UI, offering detailed information on job execution time, shuffle statistics at a stage level, and RDD/DataFrame memory usage, enabling performance bottleneck areas like skewed tasks or shuffling to be identified.

5.4.4 Experimental Evaluation Procedure

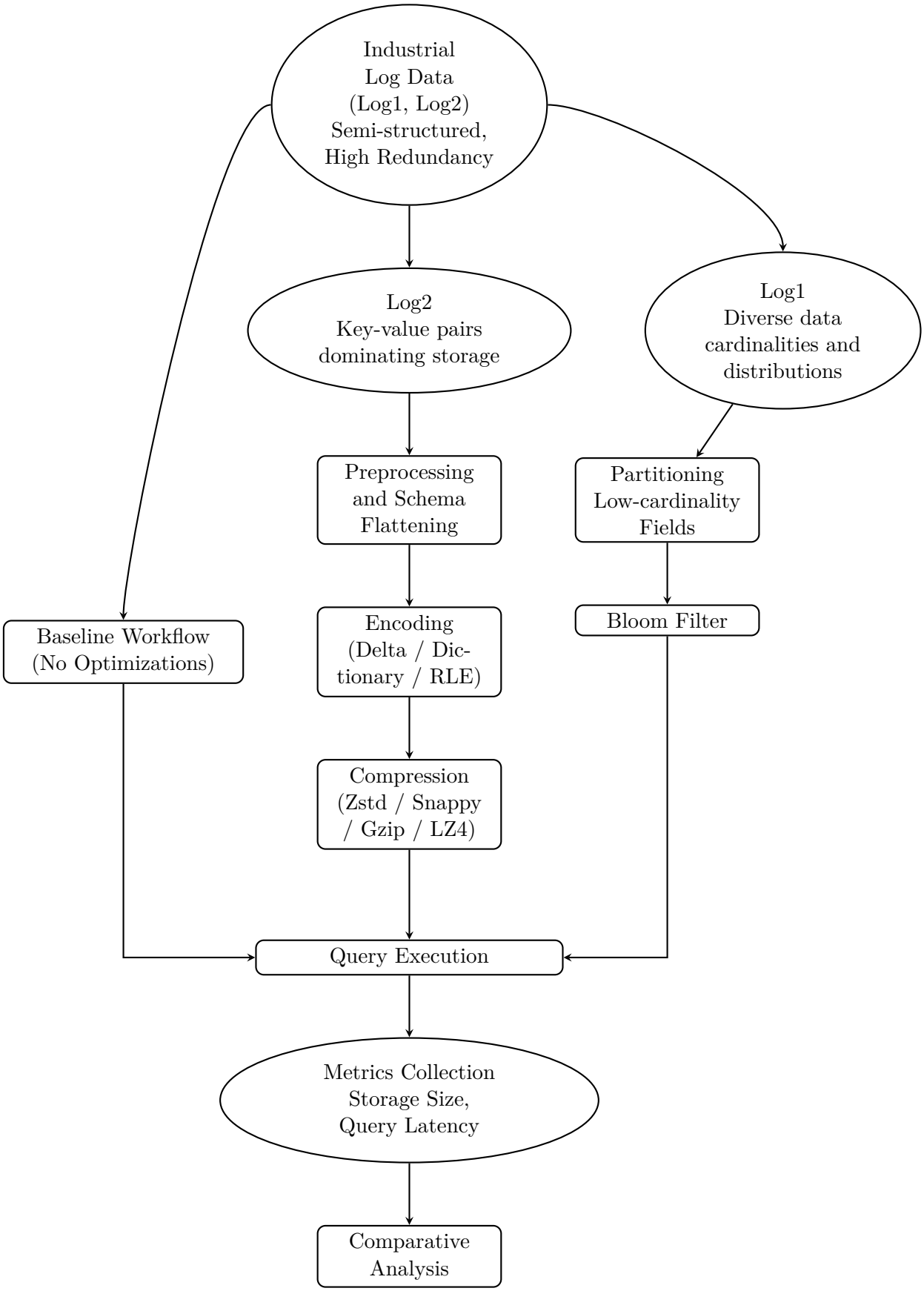

Figure 5.3: Experimental workflow of the optimization framework

Figure 5.3 shows the general workflow of the log data optimization framework. Beginning from industrial logs in their raw form, the data goes through a few processing steps to yield diverse schemes (both baseline and optimized ones). These are then subjected to queries with performance readings taken for comparison purposes.

5.4.5 Data Analysis

The analysis of experimental results in this study is based on the empirical comparison of engineering metrics across optimization configurations. For each experiment, the primary metrics of interest—**compression ratio** and **average query latency**—were collected over multiple runs to ensure consistent and robust measurements.

To facilitate interpretation, results are visualized using **plots** such as bar charts, as shown in t.ex. Figure 6.1 and line charts as in t.ex. Figure 6.3 & Figure 6.2, illustrating the comparative effects of schema flattening, encoding strategies, compression algorithms, partitioning, and Bloom filter usage on the target metrics. The resulting plots and comparative charts can be seen in **??**, where detailed results for storage efficiency and query performance are presented and explained.

No inferential statistical tests (such as t-tests or ANOVA) were conducted, as the primary objective of this thesis is to evaluate the **practical impact of system-level optimizations** rather than to test statistical hypotheses. Furthermore, the nature of the workload (industrial log data with engineering-driven performance goals) prioritizes **practical significance and reproducibility** of system behavior over formal statistical significance.

This approach was selected to ensure that findings remain actionable for practitioners seeking to improve large-scale log data storage and query performance, aligning the analysis with the engineering objectives of the research.

5.5 Design of the Optimization Stack

5.5.1 Schema Flattening (RQ1)

Nested KV pairs are expanded into independent columns, converting semi-structured data such as maps into a wide-table form. In this way, null values can be processed using Parquet's efficient null-value storage mechanism, reducing the space waste caused by null-value storage. Meanwhile, the data structure becomes more regular, allowing easier follow-up queries and analysis.

5.5.2 Compression and Encoding (RQ1)

Encoding and compression are performed at the column level using various strategies. Delta encoding is applied to timestamp fields by storing a difference between consecutive values to take advantage of temporal continuity. Dictionary encoding is

applied to low-cardinality string fields by assigning frequent strings to short numerical indices. Run-Length Encoding (RLE) is applied to fields having high repetition in values by storing sequences as pairs of counts and values.

Besides encoding, we also compare the performance of several different compression schemes, such as `LZ4`, `Zstd`, `Snappy`, and `gzip`. We test each of these compression methods against both compression ratio and query performance, taking into account how compression effectiveness can be impacted by a variety of data characteristics as well as data layouts.

5.5.3 Partitioning (RQ2)

Low-cardinality fields like `status` and `type` are used as partitioning keys, enabling query pruning and efficient scan size reduction. High-cardinality fields like `ID` can also be used to bucket using hash distribution, improving query throughput when performing equality queries. Deduplication advantages are also achieved implicitly here, as partitioned layouts reduce redundancy.

5.5.4 Blooming Filter (RQ2)

For high-cardinality query fields like `id`, there are applied Bloom Filter indices to improve filtering operations, minimize unnecessary I/O, and optimize query latency.

5.6 Reproducibility of Experiments

5.6.1 Datasets

Due to non-disclosure agreements with Ericsson, the original dataset and code used in this thesis cannot be open-sourced. However, the methodology in this study can be reproduced using publicly available datasets with similar structural characteristics, including timestamped records, nested key-value fields, and mixed cardinalities.

We suggest the following publicly available datasets for replication purposes:

- **TelemetryTracesLab01**[1]: An OpenTelemetry dataset with nested JSON logs. There are rich key-value nesting and time-based events present in this dataset, making it suitable for evaluating flattening schemas, encoding schemes, and compression techniques. Tools like OpenTelemetry sources can be used to create bigger or personalized datasets as well.

- **Loghub**[2]: A group of semi-structured log data sets such as HDFS, BGL and Thunderbird logs. Such logs have timestamped events with different cardinalities appropriate for query performance benchmarking.

[1] `https://huggingface.co/datasets/SzymonSt2808/TelemetryTracesLab01`
[2] `https://github.com/logpai/loghub`

5.6.2 System

The system configuration used in all experiments is described in Section 5.2. All experiments were conducted using Apache Spark 3.5.3.

5.6.3 Implementation

The main techniques used in this thesis include schema flattening, encoding, compression, partitioning and Bloom filtering. Although the original implementation is not open-sourced, each technique is reproducible using public tools or code generators.

Schema Flattening is performed by exploding nested arrays (e.g., key-value pairs), projecting them into individual column, grouping and aggregating them by identifiers. This produces a flat tabular structure more suitable for encoding and storage optimization.

Encoding includes both external and internal techniques. External encoding (e.g., Delta Encoding) can be scripted in PySpark. Internal encoding is automatically applied by Parquet. These are documented in the Parquet specification[3] and can be inspected using tools such as *PyArrow*.

Compression and Partitioning are applied at write time using PySpark, with Parquet supporting algorithms like Snappy and Zstandard. Partitioning by low-cardinality fields allows for predicate pushdown and I/O pruning. A comprehensive tutorial is available online[4].

Bloom Filters are used to pre-filter high-cardinality keys, reducing row group scans. Spark provides native support via `org.apache.spark.util.sketch.BloomFilter`[5].

[3] `https://parquet.apache.org/docs/file-format/data-pages/encodings/`

[4] `https://medium.com/@reetesh043/understanding-apache-parquet-a-detailed-guide-5cd7e1b30e2e`

[5] `https://spark.apache.org/docs/3.5.3/api/java/org/apache/spark/util/sketch/BloomFilter.html`

6

Results

In this section, we present the results of our experiments to address the research questions: RQ1 (storage optimization via integrated data reduction techniques) and RQ2 (query performance optimization through data layout strategies). Our integrated method includes schema flattening, targeted encoding, compression, partitioning, and Bloom filter indexing.

6.1 Storage Efficiency

6.1.1 Schema Flattening (RQ1)

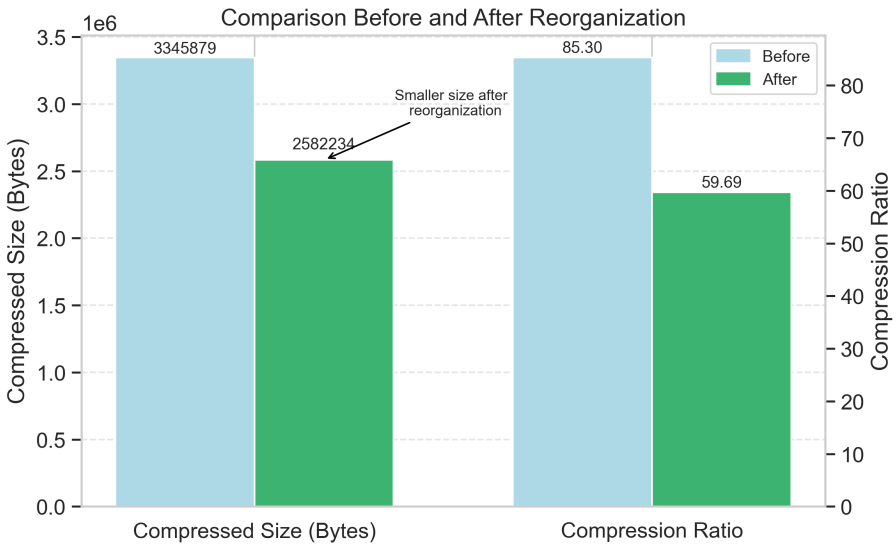

Figure 6.1: Comparison of Key Metrics Before and After Flattening

To evaluate how schema flattening and deduplication impact storage, we measure compression ratios and metadata overhead by comparing Parquet file sizes before/after restructuring nested data. Figure 6.1 shows that, by converting nested key–value pairs into independent columns, a ∼22% reduction is observed in raw Parquet file size with .snappy compression.

This flattening enables Parquet's null-value encoding to compress sparse columns more effectively. The resulting regular column layout then makes it easier to apply subsequent encoding techniques such as dictionary encoding, further decreasing data entropy. By reorganizing the structure overall, schema flattening enhances the efficiency of the compression and is well suited for deeply nested semi-structured logs with high sparsity.

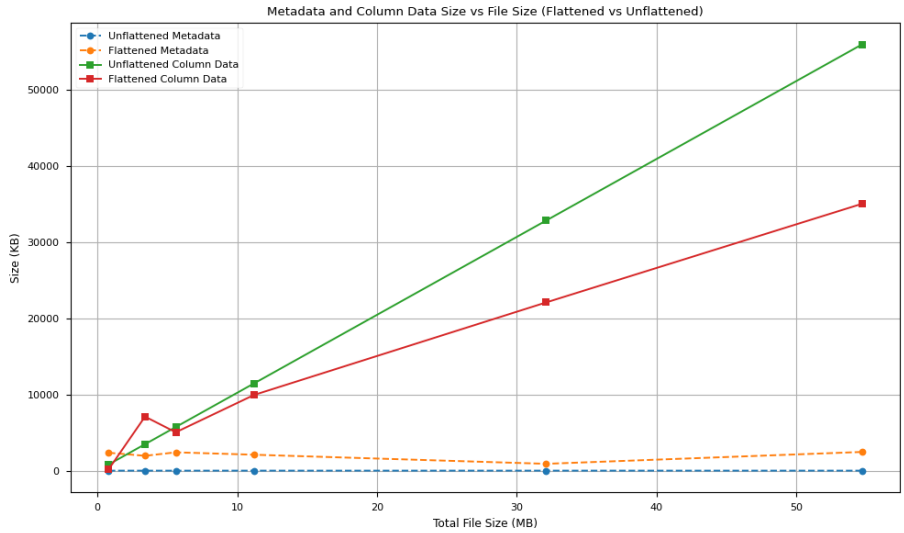

Figure 6.2: Comparison of Metadata & Column data size with File size

To further analyze the trade-off introduced by schema flattening, we compared metadata and column data sizes across increasing file sizes using both flattened and unflattened schema formats.

As shown in Figure 6.2, the unflattened format holds a low and consistent metadata footprint (around 2KB in size), with almost the entire storage being used up by column data. Meanwhile, the data indicates that the flattened format incurs a relatively consistent metadata overhead that is approximately 2MB across file sizes, more than two orders of magnitude larger than the 2KB observed in the unflattened format. Against small files (e.g., 804KB), the metadata occupies more than 90% of the file size in the case of flattening and far exceeds the column data content. While increasing the file size (e.g., 54.7MB) decreases the relative cost imposed by the metadata portion, its absolute size does not decrease. This pattern identifies a key scalability problem: flattening introduces an upfront extra cost in terms of metadata that reduces only when the dataset size is larger. Although flattening improves encoding (Section 6.1.2), its cost in terms of metadata might prove to be a bottleneck in usage cases with many small files.

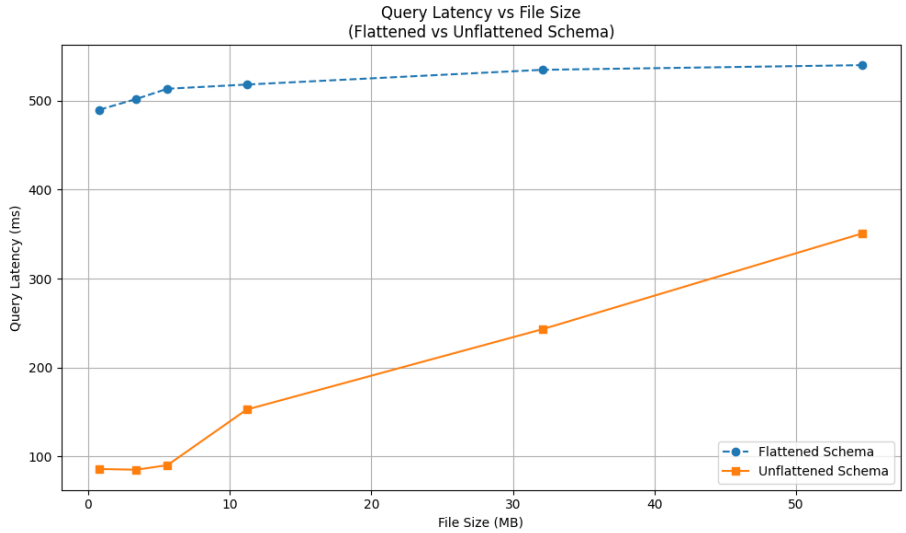

Figure 6.3: Comparison of Query Latency with File size

The data shows that while schema flattening increases encoding efficiency, it introduces an additional metadata overhead of up to 2 MB per file, which can dominate total file size for smaller datasets, as shown in earlier results. To compare scalability trade-offs, we simulated query latency over increasing file size for both flattened and unflattened datasets.

As shown in Figure 6.3, the flattened schema incurs a higher initial latency, largely due to increased metadata parsing and decoding overhead. However, its latency grows very slowly with file size, benefiting from more efficient columnar encoding and reduced structural complexity. This makes the flattened design more stable and scalable for large-scale data.

The flattened schema has an increased initial latency, as seen in Figure 6.3, owing mainly to higher overheads of metadata parsing and decoding and heavy column table scans. However, its latency increases at a slower rate as the file becomes larger, with better columnar encoding and lower structural complexity. This makes the flattened design even more stable and scalable at large scales.

In contrast, the flattened schema has lower latency initially, with the advantage of low overhead metadata and easier structural access at a small scale. However, as the file gets larger, latency increases more steeply, likely because of the increasing cost of navigating deep nested key-value fields and unoptimized encoding schemes.

Overall, an important trade-off is shown here: flattened schema prioritizes scalability with consistent performance at large scale, and unflattened schema can be better for small, straightforward files but suffers with volume.

6.1.2 Encoding Strategies (RQ1)

To compare the encoding effectiveness, we measure compression ratios for Delta, Dictionary, and RLE on timestamp and low-cardinality fields, considering both Parquet's built-in and external encoding layers. Table 6.1 shows that, Run-Length Encoding (RLE) and Dictionary Encoding resulted in nearly 0 compression gain on the tested column, resulting in what we refer to as "encoding saturation" (Discussion 7.2).

In contrast, Delta Encoding achieved a compression ratio of 0.66 by only storing differences between adjacent values instead of the actual values. This approach takes advantage of the temporal correlation commonly found in time-series data, making delta encoding particularly effective for compressing such data [23]. Therefore, external encoding methods need to be chosen according to the nature of the data.

Encoding Method	Compression Ratio	Observation
Run-Length Encoding (RLE)	≈ 1.00	No compression gain
Dictionary Encoding	≈ 1.00	No compression gain
Delta Encoding	0.66	Effective for time-series or numeric fields

Table 6.1: Encoding Method Compression Comparison

6.2 Compression Algorithms (RQ1)

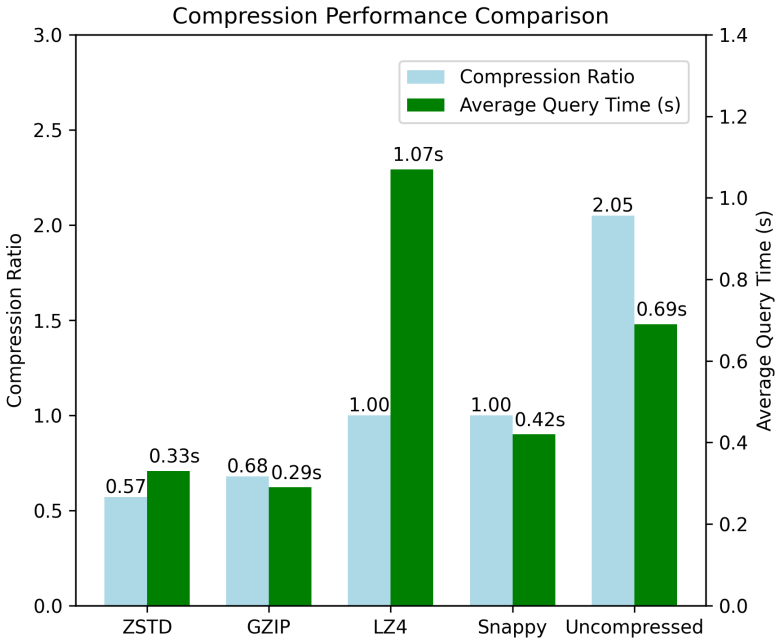

Figure 6.4: Compression Performance Comparison

To compare compression trade-offs, we measure compression ratios and query latency for Zstd, Gzip, and Snappy, analyzing I/O and CPU costs in Spark. Figure Figure 6.4 provides a comparison of compression schemes on the basis of compression ratio and query time. GZIP has a compression ratio of 0.68 and 0.29 seconds of query time on average. Zstd has a ratio of compression of 0.57 and 0.33 of query time on average, showing a compromise between query responsiveness and compaction efficacy based on our sample dataset.

Snappy and LZ4 both possess the same 1.00 compression ratio, with LZ4 having the most query time (1.07 sec). In comparison with Zstd and Gzip, Snappy creates a query time of 0.42 sec but results in zero file size savings (compression ratio of 1.00), with evidence of preferring query speed over space efficiency. Uncompressed data is the largest in size (2.05) with a query time in the middle (0.69 sec).

The choice of compression algorithm should meet workload priorities: Gzip saves storage cost at the cost of CPU usage, Snappy prioritizes low-latency access while using more I/O, and Zstd does both in the middle. In practical deployment, it is advised to use the combination of those strategies. For cold batch data, using Gzip, and for active logs, using Zstd or Snappy, can optimize resource efficiency both in storage and in compute.

6.3 Bloom Filter **(RQ2)**

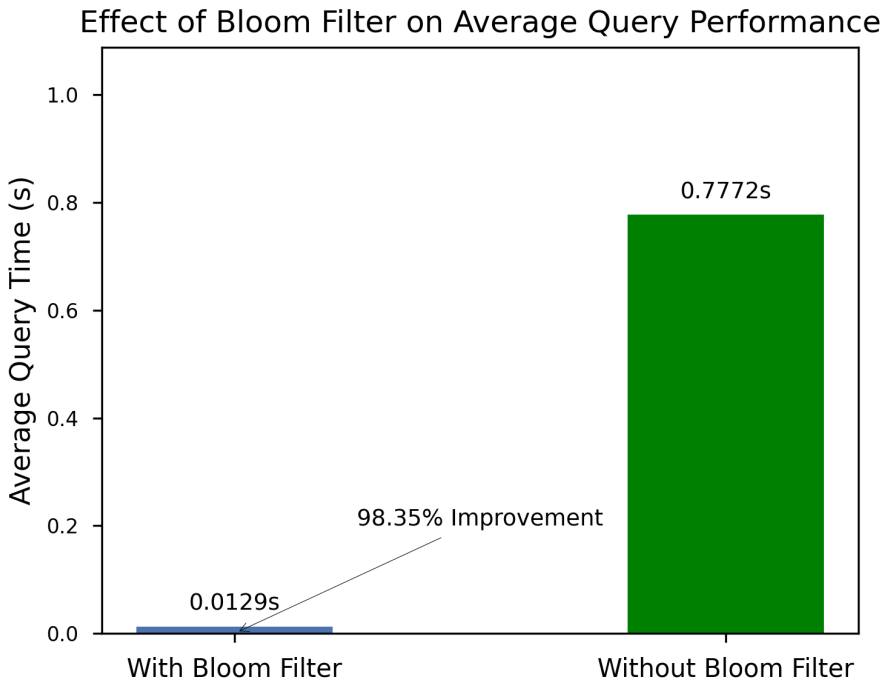

Figure 6.5: Query Time Comparison With and Without Bloom Filters

To test Bloom filters' efficiency in pruning high-cardinality fields and compare query times for existing/missing entities and storage overhead, we measure latency for non-existent entity queries and track storage overhead, using Spark to simulate high-cardinality filtering. Figure 6.5 compares query times with and without bloom filters. For this test, we issued queries on entities that were known to be absent from the dataset. Under this condition, bloom filters delivered a 98.35% improvement in query time (from 0.7772 seconds down to 0.0129 seconds), with only 947 bytes of additional metadata overhead, as shown in Table 6.2. This represents less than 0.01% of the total Parquet file size.

Metric	**Value**	**Unit**
Bloom Filter Size	947	Bytes
Original File Size	3,547,766	Bytes
Additional Storage Ratio	< 0.01	%

Table 6.2: Bloom Filter Storage Overhead

Through probabilistic hashing, Bloom filters are able to reject irrelevant blocks effectively and thus prevent full-table scans. Bloom filters showed high effectiveness for high-cardinality columns like IDs, particularly by avoiding full-table scans in our benchmark.

It is important to note that the measured improvement here is specific to queries that target non-existent entities. The actual system-wide benefit depends on the proportion of such queries in the total workload. We define the expected overall speedup as:

$$\text{Overall Bloom Filter Speedup} = p \times r \tag{6.1}$$

where p is the fraction of this query type, and r is the speedup factor (here 98.35%). For example, suppose that 30% of all queries in an actual logging system are of the form of an entity-based lookup in which the entity may or maynot be present in the dataset (i.e., $p = 0.30$). Based on our measurements, the speedup factor r provided by by the Bloom filter for such queries is around 98.35%, as shown in Figure 6.5. Through the formula, the system-wide speedup as a result would then be $0.30 \times 0.9835 = 0.295$, or approximately a 29.5% improvement in average query responsiveness. This describes how a customized optimization such as Bloom filters, when applied on certain query patterns, can provide notable system-level improvement.

6.4 Partitioning **(RQ2)**

To evaluate partitioning's impact, we measure query latency and data scan reduction for low-cardinality fields, using Spark to evaluate pruning efficiency in Parquet. Figure 6.6 illustrates storage size as well as query time before and after partitioning. Storage size was reduced a small amount from 10,762,926 KB to 10,622,978 KB with

minimal space savings, while there was a 95.4% reduction in mean query time when partitioning was applied, decreasing from 4.09 seconds to 0.19 seconds.

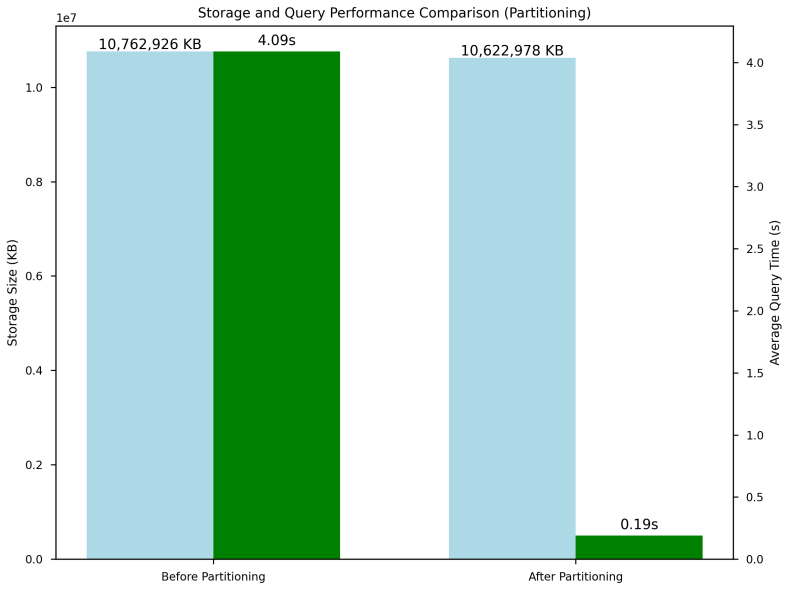

Figure 6.6: Storage and Query Performance Comparison (Partitioning)

The key advantage of partitioning is its ability to make pruning feasible. By partitioning and querying based on partitioning columns (for example, *WHERE status = 'success'* or *WHERE type = 'log_P'*), the system can skip irrelevant partitions completely and thereby narrow the scope of the scan. In this case, the status column contained merely three unique values, and consequently, two-thirds of the data could be bypassed during such queries.

The performance improvement relies upon partition key usage frequency. If more than half the requests filter based upon partition columns such as time or category, the performance would reflect nicely. However, if the majority are based upon high-cardinality fields such as ID, partitioning will have fewer performance benefits. This is consistent with typical industrial log systems where requests tend to filter upon dimensions such as time range or service type. Therefore, partitioning improves query performance on low-cardinality fields when those fields are used in WHERE clauses, as observed in our pruning tests during query execution. If those fields happen to serve as filters in the query more frequently. Practically, it is recommended to choose the most frequently queried dimensions, like timestamps or service IDs, as partition keys to optimize pruning effectiveness and system responsiveness overall.

Similar to our analysis of Bloom filters, we can also estimate the system-wide query performance gain of partitioning by the formula for expected speedup.

$$\text{Overall Partition Speedup} = p \times r \tag{6.2}$$

Here $r = 0.954$, the 95.4% measured improvement in query time with partition pruning (to 0.19 from 4.09 seconds). Should partition-based queries account for, say, 50% of the system workload ($p = 0.50$), the partitioning impact on system-wide performance would be estimated as $0.50 \times 0.954 = 0.477$, or 47.7% improvement. This puts a measure on how much partitioning can help in a situation where query pattern and partition keys correspond.

The system-wide expected query performance gain by combining the contribution of Bloom filter and partition separately can be estimated by summarizing their respective contributions:

$$\text{Overall System Query Speedup} = p_1 \times r_1 + p_2 \times r_2 \tag{6.3}$$

From our examples, if $p_1 = 0.30$ and $r_1 = 0.9835$ for Bloom filters, and $p_2 = 0.50$ and $r_2 = 0.954$ for partitioning, the integrated speedup would be $t = 0.30 \times 0.9835 + 0.50 \times 0.954 = 0.295 + 0.477 = 0.772$, or an estimated 77.2% improvement in average query speed system-wide. This simple model shows how optimizations for specific query types contribute to system-wide performance additively.

The storage savings are limited since partitioning reorders the data instead of compressing it. Nevertheless, partitioning improves the performance of Parquet's internal encodings (e.g., Run-Length Encoding), providing secondary compression benefits by colocating similar content.

6.5 Answers to Research Questions

RQ1: How does the integration of schema flattening, targeted encoding, and compression techniques affect storage efficiency in industrial-scale log data systems?

The results shows that the use of schema flattening, column-wise encoding strategies, and high-performance compression algorithms can improve storage efficiency, and when integrated they yield cumulative benefits. Schema flattening yielded a 22% file size decrease by flattening nested entities into individual sparse columns and enhancing compressibility. Delta encoding of timestamp and numeric columns resulted in a 34% size decrease. Zstd outperformed Snappy in generating 43% smaller files on average with comparable decompression performance. Combined, these methods achieved a cumulative storage saving of about 70% across our evaluation data sets, compared to the baseline with Snappy compression. This serves to answer RQ1 where these components contribute separately to total space savings and how combining them achieves compounding advantages.

RQ2: How do partitioning and Bloom filter indexing, when aligned with workload characteristics, impact query performance in large-scale log analytics?

Our analysis indicates that metadata-aware optimizations like partitioning and Bloom filters can enhance query performance if used on dominating query patterns. Specifically, Bloom filters minimized query latency by 98.35% for queries targeting non-existent entities in the dataset (e.g., querying non-existing IDs). The improvement brought down average query time with an additional storage overhead of less than 0.01% of the original data size.

Partitioning resulted in a 95.4% decrease in query latency for low-cardinality columns. Though it provided little storage improvement (only 1.3% reduction) but helped with aggressive pruning in queries on partition columns.

To quantify the system-wide impact of these optimizations, we used a weighted speedup model based on estimated proportions of queries. With 30% of queries benefiting from Bloom filters ($p_1 = 0.30$) and 50% from partitioning ($p_2 = 0.50$), the total expected query speed improvement is obtained as $t = p_1 \times r_1 + p_2 \times r_2 = 0.30 \times 0.9835 + 0.50 \times 0.954 = 0.772$, or 77.2%. This answers RQ2 by showing the isolated advantage of each technique as well as the impact in combination in real workloads where query types are mixed.

7

Discussion

The experimental results answer the two questions of this thesis. For RQ1, we illustrated that advanced data reduction methods like schema flattening, delta encoding, and compression methods like ZSTD have a significant impact on storage efficiency for big-log datasets. For RQ2, we demonstrated that layout optimizations such as partitioning, bucketing, and Bloom filter indexing improve the query performance by eliminating redundant scans of the data and optimizing access overhead. Collectively these results confirm the efficacy of our integrated optimization system and outline a practical guideline for big-log data management in industry.

Based on the results, this discusses the implications of these results and suggests possible ways of future work towards improving storage and query effectiveness in large-scale logging systems.

7.1 Effect and Potential Mechanism of Schema Flattening

In the experiments, flattening nested key-value structures into individual columns reduced the overall Parquet file size by approximately 22% under default Snappy compression. This outcome could be due to the following:

- Sparse columns (null values) take advantage of the Parquet null-value optimisation, enhancing compression performance;
- Columnar distributions are made more regular by flattening, improving the efficacy of dictionary or run-length encoding (RLE), with additional compression benefits coming downstream.

It can be inferred that a similar flattening strategy applied to other high-dimensional or deeply nested log fields may yield comparable benefits. However, it is also important to note that an excessive number of columns may increase metadata overhead and prolong query initialization.

7.2 Applicability and Order of Encoding Strategies

Parquet files have inherent automatic encodings at the column level, such as Run-Length Encoding (RLE), bit-packing, and simple dictionary encoding. These encodings are implemented internally based on the native features of the data. In our experiments, we evaluated the additional impact of applying external encodings on top of these built-in mechanisms.

To better understand the effectiveness of external encodings, we introduce the concept of a "trophy." This refers to the scenario where internal encoding has already optimized the data to such an extent that little to no further entropy can be removed. Once the "trophy" is taken as the internal encoding performs the initial encoding of the data, there will not be much additional external encoding that will provide much gain. Our experiments confirm this, where external encodings such as RLE and dictionary encoding provided minimal gains in terms of storage efficiency after the initial encoding by the internal encoding.

To evaluate if external transforms might be of continuing value, we tried varying the types of the data. In particular, we cast timestamp columns into integer types that contained seconds, and then applied Delta encoding. The two-phase transformation had incremental gains: first, the conversion of the data type precompressed the data, and then the application of the Delta encoding compressed this precompressed data again.

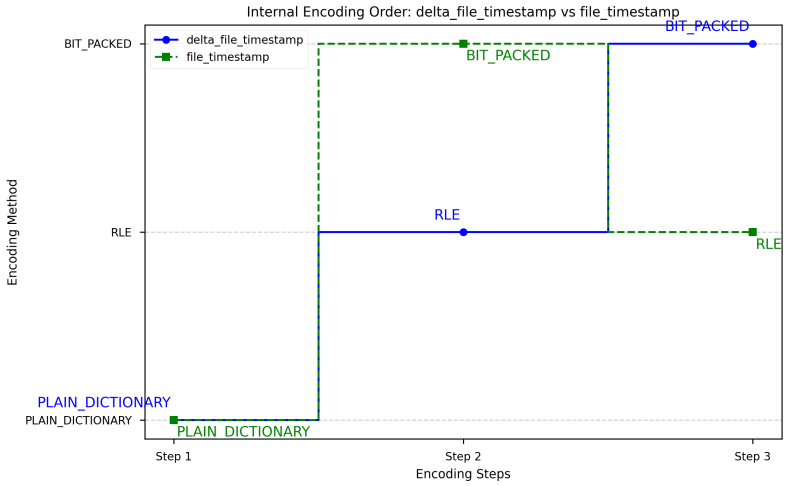

Figure 7.1: Comparison of Encoding Orders and Their Impact on Storage Efficiency

As seen in figure 7.1. Our findings suggest that modifying data types and selectively applying external encodings can complement Parquet's internal encoding strategies, achieving significant storage gains. However, when the internal encoding is already highly effective, as in the case of well-encoded data, further external encoding often

results in minimal or no improvement, likely due to the trophy situation.

This means that Parquet's inherent encoding strategies respond dynamically to the data as they go through pre-processing, again reflecting the role of optimizing the usage of internal encodings by means of strategic data manipulation.

7.3 Trade-offs Among Compression Algorithms

In our tests, Gzip and Zstandard (ZSTD) gave a satisfactory trade-off between compression ratio and query latency under our tests. Snappy and LZ4 displayed very small file size reduction, and uncompressed files, being the biggest, had a moderate level of latency.

Hardware configuration, such as 7 cores/executor and 55GB of executor memory, plays a key role in these results. When there are plenty of resources available, the overhead of decompression by such methods as Gzip becomes negligible. The fact that the compressed file size is smaller means less data have to be brought into memory, speeding I/O operations considerably. In such a resource-rich scenario, the decompression cost "gets lost in the noise," and the advantages of quicker I/O and smaller data size prevail over the decompression cost. That accounts for the surprisingly good Gzip's performance despite its generally CPU-bound nature, making Gzip seem more preferable than one might expect.

The ZSTD algorithm, boasting a good compression ratio as well as a decent decompression speed, was chosen as the most suitable outer-layer compression algorithm. It strikes a good balance between storage-size minimization and keeping the query times efficient and thus suitable for offline batch-processing applications.

Overall, the ideal compression algorithm relies very much on the hardware setup.

7.4 Enhancements from Partitioning

Partitioning on low-cardinality fields such as `status` and `type` resulted in only minor storage size reductions, yet it dramatically improved query performance. It is possible that, though well-organized partitions enhance the effectiveness of techniques like dictionary encoding and run-length encoding inside Parquet files, those low-cardinality fields already benefit a lot from built-in encoding, so the trophy is already taken.

It is also important to note that partition effectiveness is bounded by the "slot" principle: only a limited number of columns can be practically optimized via partitioning, bucketing, or clustering. Each slot consumed introduces overhead—additional metadata, management complexity, and potential skew if the key distribution is uneven. Another significant factor is partition size. According to best practices delineated by recent research, partitions should preferably be between 128MB and 512MB when flushed to disk. Small partitions are inefficient due to overhead such

as metadata scanning and file I/O amplification, while very large partitions decrease pruning effectiveness. Thus, careful slot allocation is necessary: we prioritize the most common filter keys to maximize the benefits without introducing the "small file problem." In more delicate cases, bucketing (hash-based organization) could also be layered on top to handle high-cardinality columns, further balancing partition size and query performance.

7.5 Bloom Filter

Applying Bloom filters significantly improved query performance in cases targeting non-existent entities with negligible metadata overhead. Bloom filters in this case serve as a fast, space-efficient mechanism for approximate set membership testing. They can quickly identify when a queried element is definitely not in a set, avoiding expensive lookups in a database with very little overhead. In other words, if a query matches an existing item, the extra cost of checking the filter is nearly imperceptible.

Bloom filters, however, are probabilistic structures. They never produce false negatives (they will not miss an item that is actually in the set), but they can yield false positives (a query can falsely appear to match an item that isn't really present). Due to this probabilistic character, the net advantage of a Bloom filter varies depending on workload. If most of the queries are for keys that do not happen to be present in the data, the filter typically pays for itself by eliminating such cases in a short time. If nearly all the queries hit a present record, the cost of updating and querying the filter could offset its usefulness.

Though false positives are the main risk when using Bloom filters, they cause an unnecessary check of the underlying data. There are tuning strategies to mitigate this issue, which we will not cover here.

In summary, profiling Bloom filter behavior on historical query logs can indicate whether deploying a filter will significantly improve performance for the expected workload.

7.6 Threats to Validity

To add more credibility and generalizability to our findings, we discuss the validity of our research by looking at threats to external, internal, and construct validity, with related mitigations.

7.6.1 External Validity

The main external validity threat here is the study's reliance on industrial datasets sourced from Ericsson. This creates valid concerns about generalizability to other industries or systems of logs. However, our datasets provide solid reasons for generalizability on closer inspection.

Ericsson's logs show a few common characteristics of large-scale industrial systems. They are semi-structured with nested key-value pairs affecting schema variability. They are also highly redundant with repetitive values and have scalability issues with billions of records and multi-terabyte sizes. These problems are not limited to telecommunications; they are common to diverse industries like finance, cloud computing, and manufacturing [24][25][26].

Our suggested optimizations also support our findings to be more generalizable. Techniques like schema flattening, encodings, and Bloom filters apply to general data properties, including redundancy and access patterns, instead of domain-specific logic.

However, it is worth noting the limitations. Environments with completely unstructured text logs or with minimal redundancy may get lower efficiency in our optimizations. However, such cases are quite uncommon in business environments [11].

7.6.2 Internal Validity

Our experimental design can be challenged by potential confounding variables. External hardware and software setup biases can impact our results. In addition, Parquet defaults such as Snappy compression used in base case comparisons may not reflect real deployments. Data processing operations in data can also unintentionally change data properties.

To mitigate these threats, we implemented a controlled experimental setup. Our experiments applied a fixed Spark setup, described in Table 5.1, with dynamic resource allocation. This isolated our optimizations from hardware variability. We applied incremental testing on each optimization. This isolated the performance gains to each optimization. We also prioritized data integrity while processing data. Deduplication was done while preserving unique records, and schema flattening kept all fields without losing data.

7.6.3 Construct Validity

The misalignment between our research questions and the measurement tools we used mainly threatens the construct validity. The metrics, compression ratio and query latency, might not fully reflect our research objectives of "storage efficiency" and "query performance." In addition, our experiment's benchmark of synthetic queries may not fully represent real-world workloads.

To mitigate these threats, we took a more holistic metric way. For storage efficiency, we took into account both raw size savings and metadata overhead, such as metadata from schema flattening. For query performance, we evaluated end-to-end latency, including Spark's Catalyst optimization overhead. For benchmark relevance, our queries were designed to carefully mimic real-world log analysis tasks, such as filtering by timestamps and high-cardinality IDs. These queries were also validated by

getting inputs from Ericsson domain experts to ensure industrial relevance.

7.6.4 Generative AI Usage

As required by the Chalmers Generative AI Policy, we disclose that generative AI, specifically ChatGPT, was used during this thesis. These tools were used only for the purpose of grammar correction and clarifying technical terms, ideas, and relevant scholarly citations. The AI tools were neither used for writing any portion of the thesis nor for generating experiment results, analysis, or diagrams.

The application of generative AI in this context could introduce construct validity risks, especially if it is over-depended upon or taken for granted without critique. To prevent this, all suggestions generated through AI were edited and reviewed for consistency with our desired meaning and scholarly standards. We believe that such a limited and transparent application is in alignment with the guidelines and does not diminish the scientific or academic credibility of the research.

7.7 Industrial Feedback

During the project, Ericsson provided feedback on the applicability and utility of the suggested log data optimization techniques. The company reported general satisfaction with the results and verified the applicability of the methods, especially with respect to AI-related workflows.

One of the key highlights was the role of delta encoding and schema flattening in preprocessing numeric-intensive logs. Such optimizations were recognized as helpful for compressing the size of the data and speeding up the AI model's training process, where large amounts of telemetry and metric-related logs are being consumed.

Ericsson also focused on the possibility of compressing and structurally restructuring large amounts of semi-structured log data, directly benefiting downstream machine learning pipelines. The coupling of flattening and encoding with partitioning was also thought of as being well-suited as the basis of effective storage frameworks of both traditional analytics and systems based on AI.

The company considered the thesis as an important input that fits well into the new needs of the industry and suggested further research could be done into real-time or near-real-time processing of logs for streaming applications.

8

Conclusion

The motivation for this thesis arises from the increasing challenge of processing and extracting value from enormous amounts of log data in industrial systems. As distributed systems scale, the expense of storage and query latency grows exponentially. Current methods tend to focus on specific optimizations individually and lack focus on a comprehensive, reproducible, and scalable solution specific to semi-structured, redundant logs. This thesis aimed to address those challenges by proposing and evaluating one integrated framework that combines schema transformation, encoding, compression, partitioning, and indexing strategies.

This research was carried out in close cooperation with Ericsson, based on actual production logs from deployed base station clusters in operation. Industrial cooperation based the research on practical constraints and ensured that the approaches proposed solved actual needs. The datasets employed were structurally and cardinally varied, representative of complexity in large network operation.

By systematic testing, this study demonstrated the quantifiable effect of various approaches. Schema flattening exhibited compressibility and query homogenization improvements, especially for deeply nested key-value columns. Delta encoding reduced storage for time-series columns, and Zstandard always generated a desirable balance of decompression time and ratio of compressed size over uncompressed size. Using bloom filters on high-cardinality columns resulted in query pruning with negligible overhead on space. Partitioning further improved query performance. Collectively, all of these strategies resulted in query and storage efficiency improvements.

The implications of these results are two-fold. They yield concrete evidence that diligent encoding and restructuring of log data can reduce the resource overhead of storage systems significantly. They imply that enterprise log analytics pipelines, especially those deployed in AI model training, can be optimized with preprocessing tools optimized for columnar formats such as Parquet. Ericsson's feedback corroborated that applications such as numeric-heavy datasets employed in the training workflow, for instance, can reap benefits directly from effective encoding since less computing cost is incurred.

Future research could look into extensions of this framework for real-time or edge computing scenarios where resource predictability is lower and latency budgets are

tighter. One way is gathering runtime performance information like CPU, memory, and I/O statistics within the Spark framework, which could give us a better understanding of performance bottlenecks during query execution and data transformation. Another is the design of adaptive partitioning and compression strategies that adjust to the workload conditions. Additional support for query pruning could be provided through more granular row-group histograms of metadata. Advanced clustering methods incorporating Z-order curves could enable more efficient multi-dimensional filtering and sorting. Last, the modeling of dataset characteristics, query latency, and resource usage through statistics could help capacity planning and maximize system scalability for production deployments.

Bibliography

[1] Rui Wang, Devin Gibson, Kirk Rodrigues, Yu Luo, Yun Zhang, Kaibo Wang, Yupeng Fu, Ting Chen, and Ding Yuan. {μSlope}: High compression and fast search on {Semi-Structured} logs. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 529–544, 2024.

[2] João Paulo and João Pereira. A survey and classification of storage deduplication systems. *ACM Computing Surveys (CSUR)*, 47(1):1–30, 2014.

[3] Michael Armbrust et al. Delta lake: High-performance acid table storage over cloud object stores. *Proceedings of the VLDB Endowment*, 13(12):3411–3424, 2020.

[4] Youmin Chen et al. Flatstore: An efficient log-structured key-value storage engine for persistent memory. *ACM SIGOPS Operating Systems Review*, 54(2):1–14, 2020.

[5] Avrilia Floratou, Jason Teletia, David J. DeWitt, Jignesh M. Patel, and Donghui Zhang. Column-oriented storage techniques for mapreduce. In *Proceedings of the VLDB Endowment*, volume 4, pages 419–429, 2011.

[6] Daniel Abadi et al. Integrating compression and execution in column-oriented database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 671–682, 2006.

[7] Andrew Pavlo et al. Skew-aware automatic database partitioning in shared-nothing, parallel oltp systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 61–72, 2012.

[8] D Viji and S Revathy. Hash-indexing block-based deduplication algorithm for reducing storage in the cloud. In *2023 International Conference on Computing*, pages 1–6, 2023.

[9] Kai Ren et al. Indexfs: Scaling file system metadata performance with stateless caching and bulk insertion. In *SC '14: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 237–248, 2014.

[10] Muralikrishnan Ramane, Sharmila Krishnamoorthy, and Sasikala Gowtham.

An experimental evaluation of performance of a hadoop cluster on replica management. *arXiv preprint arXiv:1411.1931*, 2014.

[11] Tianzhu Zhang, Han Qiu, Gabriele Castellano, Myriana Rifai, Chung Shue Chen, and Fabio Pianese. System log parsing: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 2022.

[12] Niteesha Sharma et al. Data deduplication techniques for big data storage systems. *International Journal of Recent Technology and Engineering*, 8(2):3062–3065, 2019.

[13] Stratos Papadomanolakis and Anastassia Ailamaki. Autopart: automating schema design for large scientific databases using data partitioning. In *Proceedings of the International Conference on Scientific and Statistical Database Management*, pages 383–392. IEEE, 2004.

[14] Carlo Curino et al. Schism: a workload-driven approach to database replication and partitioning. In *Proceedings of the VLDB Endowment*, volume 3, pages 48–57, 2010.

[15] Patrick O'Neil and Dallan Quass. Improved query performance with variant indexes. In *SIGMOD Record*, volume 26, pages 38–49. ACM, 1997.

[16] Alexander Golynski, Alessio Orlandi, Rajeev Raman, and Srinivasa S. Rao. Optimal indexes for sparse bit vectors. *Algorithmica*, 69(4):906–924, 2014.

[17] Jaehong Min, Daeyoung Yoon, and Youjip Won. Efficient deduplication techniques for modern backup operation. *IEEE Transactions on Computers*, 60(6):824–840, 2011.

[18] Yixun Wei, Zhichao Cao, and David H.C. Du. Cpi: A collaborative partial indexing design for large-scale deduplication systems. *IEEE Transactions on Parallel and Distributed Systems*, 2025.

[19] Joao Paulo and Joao Pereira. Efficient deduplication in a distributed primary storage infrastructure. *Cluster Computing*, 19:1–15, 2016.

[20] Geyao Cheng et al. Lofs: A lightweight online file storage strategy for effective data deduplication at network edge. *IEEE Transactions on Parallel and Distributed Systems*, 33(5):1201–1215, 2022.

[21] Peng Fei, Zhou Li, Zhiying Wang, Xiao Yu, Ding Li, and Kangkook Jee. SEAL: Storage-efficient causality analysis on enterprise logs with query-friendly compression. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2987–3004. USENIX Association, 2021.

[22] Stefan Aulbach et al. Multi-tenant databases for software as a service: schema-mapping techniques. In *Proceedings of the ACM SIGMOD International Con-*

ference on Management of Data, pages 1195–1206, 2008.

[23] Jinyang Liu, Jieming Zhu, Shilin He, Pinjia He, Zibin Zheng, and Michael Lyu. Logzip: Extracting hidden structures via iterative clustering for log compression, 09 2019.

[24] Xue Zhou. The application and challenges of cloud computing in financial services. *Financial Engineering and Risk Management*, 7:117–123, 2024.

[25] Theofanis P. Raptis, Andrea Passarella, and Marco Conti. Data management in industry 4.0: State of the art and open challenges. *IEEE Access*, 7:97052–97093, 2019.

[26] Ziyan Yao. Application of cloud computing platform in industrial big data processing. *arXiv preprint arXiv:2407.09491*, 2024.