Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

Improving Parquet Compression Using Global Dictionaries in Delta Lake

Author: Eames Trinh (2782581 - etr142)

1st supervisor:	Peter Boncz
daily supervisor:	Lars Kroll (Databricks)
2nd reader:	Daniele Bonetta

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

July 31, 2025

Abstract

As data lakes grow in scale, optimizing storage efficiency is essential for maintaining performance and controlling costs. Apache Parquet, a widely used columnar storage format, offers dictionary encoding to reduce data size, but this encoding is limited to individual row groups and cannot take advantage of redundancy across files. This thesis introduces global dictionary compression for Parquet files in Delta Lake, enabling shared dictionaries that span multiple files and ensure consistent encoding of repeated values. We present a system design that integrates global dictionaries with Delta Lake's metadata and transaction mechanisms, supports hybrid encoding with local fallbacks, and maintains compatibility with existing Parquet infrastructure. Extensive benchmarks on both synthetic and real-world datasets demonstrate storage savings of well over 10 percent in favorable scenarios, with minimal overhead during reads and writes. Additionally, we show how global dictionary generation can be effectively packaged into a background job, contributing to long-term storage optimization with minimal disruption in the write path. These findings establish global dictionary encoding as a practical and effective enhancement to Parquet-based data lake systems.

Contents

1

Introduction

In today's data-driven world, organizations increasingly rely on large-scale, semi-structured datasets to support analytical workloads, machine learning, and real-time decision-making. As data volumes continue to grow, so does the demand for storage formats that are both space-efficient and performant. Apache Parquet has emerged as a widely adopted solution in this space due to its columnar layout, which supports efficient I/O, and its built-in compression techniques, particularly dictionary encoding. By replacing repeated values in a column with compact integer representations, dictionary encoding helps reduce file sizes and speeds up query execution.

However, Parquet's dictionary encoding is inherently limited: it operates only within the boundaries of a single chunk of a single column in a single file. This local scope means it cannot exploit redundancies that appear across multiple parts of a file, much less across different files. In many real-world scenarios, such as logs, user activity records, or e-commerce transactions, datasets contain numerous repeated values (e.g. user IDs, country codes, or categories) that recur across files. Parquet's lack of cross-file awareness results in missed opportunities for improved compression and efficient data processing.

Delta Lake, a storage layer built on top of Parquet, introduces a metadata-driven architecture that offers a potential solution to this limitation. By maintaining a centralized transaction log and schema information, Delta Lake enables coordination and optimization across many Parquet files in a dataset. This centralized control opens the door to implementing global dictionaries: shared dictionaries that span multiple files and ensure consistent encoding of values throughout the dataset. Such a mechanism could lead to significant storage savings and potentially faster query execution (due to integer-based filtering across files).

1.1 Research Objective

The primary objective of this thesis is to explore the feasibility, design, and storage/performance implications of implementing global dictionaries in Delta Lake. While dictionary encoding already provides substantial compression benefits within individual Parquet row groups,

this research aims to extend those benefits across files by enabling shared, "global" dictionaries. The work is motivated by the need to improve storage efficiency and query performance in large-scale data lake environments, particularly in scenarios where datasets contain significant redundancy across files.

To guide this investigation, the following research questions are defined:

- **RQ1:** How can a global dictionary be designed and managed across distributed systems, considering file immutability and evolving datasets? Can global dictionaries coexist with local dictionaries in a complementary manner?
- **RQ2:** What are the trade-offs in write, read and storage performance when using global dictionaries compared to local dictionaries?
- **RQ3:** How does the system handle newly arriving data with previously unseen values, and how does this affect dictionary maintenance?
- **RQ4:** How can global dictionaries be integrated with Delta Lake's metadata mechanisms, such as metadata and add actions?
- **RQ5:** Can global dictionaries be efficiently utilized during compaction or checkpointing processes?

1.2 Methodology Overview

The research approach consists of multiple phases, beginning with a literature review and feasibility study. This is followed by the design and implementation of several prototypes of global dictionary encoding within the Delta Lake framework. Each prototype is evaluated under different workloads to understand its impact on compression efficiency, query latency, and ingestion throughput.

This work is conducted at Databricks, the original creator of Delta Lake. The implementation and evaluation take place within the Databricks environment, using the Databricks Runtime (DBR), an extended and optimized version of Apache Spark. This environment provides practical insights into real-world system behavior and production constraints. Throughout the project, care is taken to ensure that the core implementation remains as compatible as possible with and contributes to the open-source Parquet and Delta Lake projects, so as to make it such that the findings and solutions developed here can be of value to the broader data engineering and research community.

The methodology involves several key phases. It begins with a review of existing compression and dictionary encoding techniques in Parquet and related systems. Based on this foundation, a global dictionary architecture is designed and implemented to be compatible with Delta Lake. The prototype is then iteratively refined through performance testing and analysis of integration challenges. Finally, the prototypes are evaluated using both real-world and synthetic datasets to measure their effects on write and read performance, storage savings, and overall system overhead.

1.3 Contribution

This thesis contributes a practical investigation into the design and implementation of global dictionaries for Delta Lake. Specifically, it offers:

- A novel architecture for supporting global dictionary encoding across Parquet files using Delta Lake's metadata coordination layer.
- Multiple working prototypes demonstrating different trade-offs in terms of update strategies, hybrid dictionary usage, and fallback mechanisms.
- A comprehensive performance evaluation across various datasets.
- Design insights and implementation guidelines for integrating global dictionaries into real-world data lake systems, including discussion of production feasibility and future improvements.

Perhaps equally important to the contributions in storage savings, the omission of compute savings must be addressed. Due to the nature of having a single global dictionary that provides consistent encodings across an entire dataset, potentially massive read-path optimizations becomes available. This can take the form of efficient cross-file comparisons on encoded data and aggregation query speedups. Although placed beyond the scope of this research, we discuss briefly what its impacts may be in Section 6.

2

Background

2.1 Database Fundamentals

In this section, we provide a comprehensive overview of fundamental database concepts necessary to understand the context and motivation for this thesis. The discussion spans the relational model, query execution, storage models, and characteristics of analytical workloads. These topics set the foundation for understanding the role of columnar storage and compression techniques, particularly dictionary compression in Apache Parquet, and how global dictionaries can offer improvements in performance and efficiency.

2.1.1 The Relational Model and SQL

Relational databases represent data in terms of relations, commonly referred to as tables. Each table consists of rows (tuples) and columns (attributes), where each column has a specific data type and semantic meaning. The relational model, proposed by E. F. Codd in 1970 (1), forms the basis of most modern database systems and defines a formal framework for data storage and manipulation using mathematical set theory and predicate logic.

Structured Query Language (SQL) is the standard language used to query and manipulate relational databases. SQL supports a wide range of operations including data definition (DDL), data manipulation (DML), and data querying. A typical SQL query retrieves data from one or more tables using a combination of operations such as selection (filtering rows), projection (selecting specific columns), joins (combining rows from different tables), aggregation (summarizing data), and ordering.

SQL queries are declarative, meaning that the user specifies what data is desired without detailing how to obtain it. The database query engine is responsible for parsing the SQL statement, optimizing the query plan, and executing it efficiently.

2.1.2 OLAP Workloads and Analytical Queries

Online Analytical Processing (OLAP) workloads are characterized by complex, read-intensive queries over large volumes of data (2). Unlike Online Transaction Processing (OLTP),

which emphasizes rapid transactional updates and inserts, OLAP focuses on gaining insights from historical and current data through aggregations, filters, and multi-dimensional analysis.

A common design pattern for OLAP databases is the star schema, where a central fact table stores quantitative data (e.g., sales, transactions) and is connected to multiple dimension tables that provide descriptive attributes (e.g., time, customer, product). Analytical queries typically access a subset of rows and columns, performing operations such as grouping, summation, and joining with dimensions to provide context.

Given the scale of data in OLAP systems, efficient query execution becomes critical. Optimization techniques such as columnar storage, data compression, predicate pushdown, and vectorized execution are employed to reduce I/O, memory usage, and computation time (3, 4).

Recent system-level studies reinforce the importance of such optimizations. Snowflake's disaggregated cloud-native architecture highlights the need for elasticity, ephemeral intermediate storage, and caching to handle OLAP workloads with diverse and unpredictable characteristics (5). Meanwhile, Redshift shows how cloud DBMS systems can improve by, for example, decoupling storage and compute or sharing precompiled queries anonymously across users (6), a model also employed by Snowflake and Databricks.

2.1.3 Query Execution

Query execution involves translating a high-level SQL query into a series of low-level operations that can be performed on data. The typical stages include:

- **Parsing and Validation**: The SQL query is parsed into an abstract syntax tree (AST), which is validated against the schema and syntax rules.
- **Logical Plan Generation**: The query is converted into a logical plan representing relational algebra operations such as scans, filters, projections, and joins.
- **Optimization**: The logical plan is transformed through rule-based and cost-based optimizations, such as join reordering, predicate pushdown, and subquery flattening.
- **Physical Plan Generation**: The optimized logical plan is converted into a physical plan with concrete operators that define how each operation will be executed.
- **Execution**: The physical plan is executed by the query engine, often using pipelined and vectorized execution strategies to improve performance (7).

Execution engines in modern distributed systems, such as Apache Spark, compile query plans into low-level bytecode or native code to eliminate interpretation overhead. Whole-stage code generation and just-in-time (JIT) compilation are commonly used techniques (8).

2.1.4 Storage: Row-Oriented vs Column-Oriented

The way data is stored on disk and in memory significantly impacts query performance. Two primary storage models exist:

Firstly, in row-oriented storage, data for each row is stored contiguously. This model is efficient for OLTP workloads where entire rows are frequently accessed or modified. However, it can be suboptimal for OLAP queries that only access a subset of columns but many rows, as unnecessary data must be read and processed.

Column-oriented storage, on the other hand, stores data column by column. All values of a single column are stored contiguously, allowing efficient access to relevant data for OLAP queries. This layout enables several optimizations:

- **I/O Efficiency**: Only the required columns are read from disk, reducing I/O.
- **Better Compression**: Columns often have homogeneous data types and repeated values, enabling high compression ratios using techniques such as dictionary encoding, run-length encoding (RLE), and bit-packing (9).
- **Vectorized Execution**: Processing columnar data in concurrent vectors of (columnar) data yields several benefits, such as reduced query expression interpretation cost (in the absence of code generation optimizations) as well as compiler based loop optimization techniques like SIMD, loop hoisting, and strength reduction (10).

2.2 Apache Parquet

Apache Parquet is an open-source, columnar storage format designed for efficient data analytics on large-scale datasets (9). It is widely used in modern big data ecosystems due to its support for flexible schemas, efficient compression, and performance-optimized read/write operations. Parquet is particularly suited for OLAP workloads, where queries typically involve reading a small subset of columns from large datasets.

It is worth noting that Parquet is simply one of many data storage formats, and often competes with contemporaries such as ORC (9), which contains many similar features. Other nascent formats exist only as research proposals, like Fastlanes (11) or BtrBlocks (12), which attempts to rely cascading and lightweight encodings like dictionary, run-length, or FSST encoding (13) in favor general purpose compression techniques like Snappy, Zstd, or LZ4.

Nevertheless, this section details Parquet's data organization, focusing on its logical structure, physical layout, and encoding/compression schemes. Particular attention is given to dictionary encoding, which plays a central role in this thesis.

2.2.1 Logical Data Organization

Parquet stores data in a hierarchical manner, enabling efficient access and scalability. The core components of the logical structure are shown in Figure 2.1 (9):

- **File**: The top-level container that includes a magic number at the beginning, the metadata in the footer, and one or more row groups in between.
- **Row Group**: A logical horizontal partitioning of data, containing data for a subset of rows (as they would correspond to in a traditional DBMS) across all columns.

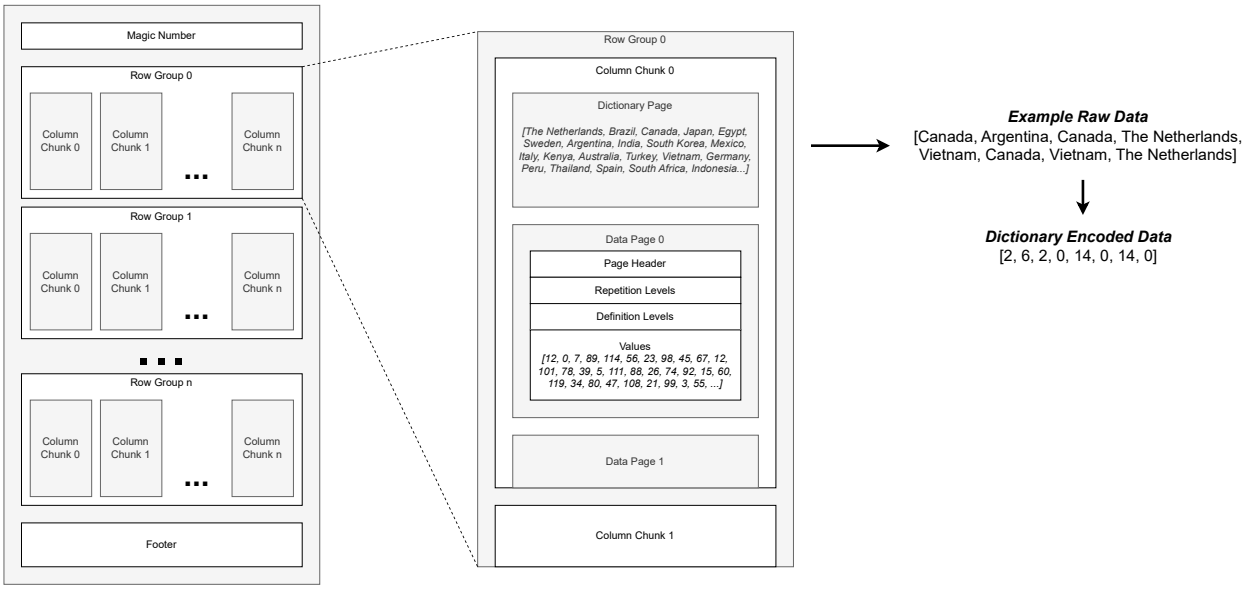

Figure 2.1: Parquet file format structure, emphasizing the page structure within column chunks, and how dictionary encoding is applied within it.

Row groups allow for optimizations such parallel read/write operations and range filtering.

- **Column Chunk**: A contiguous block of data for a single column within a row group. That is, the row group is made up of n column chunks, where n is the number of columns in the Parquet schema.
- **Page**: The smallest unit of data. Each column chunk consists of one or more pages (of 1MB size by default). Pages can be of different types (i.e., data pages and dictionary pages).

The file's footer stores important metadata, including the schema, row group statistics, and offsets for efficient navigation during reads.

2.2.2 Physical Data Layout and Compression

Parquet's physical storage is optimized for efficient I/O and space utilization. Data is stored in a columnar fashion, meaning that values of the same column are stored together (within a row group). This design facilitates the application of compression schemes tailored to the specific data type and value distribution of each column.

There are two general page types in Parquet:

- **Dictionary Page**: An optional page containing a dictionary of unique values for a column chunk when dictionary encoding is used.
- **Data Page**: Contains the actual encoded and compressed data. In the case of dictionary encoded data pages, the values are array offset referencing dictionary

indices.

Parquet supports multiple encoding schemes[1]. However, the de facto encodings are those included in Parquet V1, the most (if not only) used Parquet version in practice. It includes two encoding types, plain (raw; without compression) and dictionary encoding (mapping *unique values*, stored in a dictionary page, to *integer indices*, stored in the data page, as surrogate keys).

Beyond encoding, Parquet also uses general-purpose compression (e.g., Snappy, GZIP, LZO, Zstd) to further compress pages. The compression is applied after encoding, and is written immediately after the page header. By default, Snappy compression is used.

2.2.3 Dictionary Encoding in Detail

Dictionary encoding is a two-level compression technique that is particularly effective for string or categorical data with a limited number of unique values (3). The steps involved are as follows:

1. As each new value is written to the Parquet, it's existence is checked in the current dictionary. If absent, add it to the dictionary array before finally encoding its offset in the dictionary.
2. If the dictionary size passes a certain size threshold (typically 1MB), rewrite all values using their plain representations. All subsequent pages will be plain encoded, as well.
3. The dictionary is written to a dictionary page; the indices are written using RLE and bit-packing hybrid encoding.

The dictionary page enables significant space savings and efficient query execution since comparisons can be done at the index level rather than on full values.

However, Parquet's dictionary encoding operates at the *column chunk level*, meaning each row group's column chunk has its own independent dictionary (see Figure 2.1). This design facilitates parallelism and localized compression but introduces several limitations:

- **Redundancy**: Repeated values across column chunks lead to duplicated dictionaries. This effect is exacerbated when the column already has a limited domain across the entire dataset.
- **Limited Scope**: Dictionary size is constrained by row group size and memory limits, affecting high-cardinality columns.
- **Reader Overhead**: Each reader must load and manage multiple dictionaries for a column across different files and row groups.
- **Inconsistent Encoding**: The same value may have different dictionary indices in different files, complicating optimizations and indexing as their encoded formats cannot be directly compared.

To address these challenges, this thesis proposes externalizing dictionaries from the Parquet file structure and managing them globally in Delta Lake. Global dictionaries can

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

improve compression efficiency, reduce redundancy, and enable advanced query optimizations by leveraging shared value mappings across datasets.

This architectural change requires modifications to both Parquet writers and readers to support external dictionary lookups during encoding and decoding, as well as integration with Delta Lake's metadata and transaction management systems.

2.2.4 Mechanics of RLE_DICTIONARY Encoding

While dictionary encoding in Parquet provides significant space and performance improvements, the internal mechanics of the encoding process reveal further details critical for understanding its efficiency and its limitations, particularly in the context of this thesis.

Parquet's `RLE_DICTIONARY` encoding, the primary dictionary-based compression method in the original (V1) Parquet specification, combines multiple techniques:

- **Dictionary Compression**: A unique dictionary is built for each column chunk, assigning an integer code to each distinct value.
- **Run-Length Encoding (RLE)**: Sequences of identical dictionary codes are stored compactly by representing the repeated value and its count, rather than listing each occurrence individually.
- **Bit-Packing**: To minimize space further, the dictionary codes are stored using only as many bits as necessary to represent the maximum code value (e.g., 2 bits if there are 4 unique values).

When writing a column chunk with `RLE_DICTIONARY` encoding, Parquet first emits a *dictionary page* containing all distinct values, followed by one or more *data pages* that store sequences of dictionary indices compressed using a mixture of RLE and bit-packing runs. This layout allows efficient lookups during query execution, but has critical design trade-offs:

- If a value is highly repeated (e.g., a country column dominated by "United States"), RLE enables highly compact representation.
- If values are more uniformly distributed, bit-packing ensures that even without repetition, the space usage remains minimal.

However, dictionary encoding in Parquet V1 is restricted to operate *per column chunk*, and dictionary page sizes are capped, leading to situations where high-cardinality columns may exhaust the dictionary, forcing fallback to plain encoding within the same column chunk.

This detail is particularly relevant for this thesis, as the local, per-chunk nature of dictionary compression in Parquet is a motivating factor for introducing **global dictionaries** managed externally through Delta Lake. A global dictionary can overcome the fragmentation and duplication across chunks, offering improved compression efficiency and query optimization opportunities.

2.2.5 Parquet Specification Versions: V1 vs. V2

Most existing Parquet implementations, including those used in Spark and Delta Lake, remain based on the original V1 specification.

The V2 specification introduces more advanced encoding techniques, such as delta encoding variants for integers and byte arrays, which in theory offer better compression for specific patterns like incremental or prefix-sharing data. However, V2 has not been widely adopted in practice. This is largely due to the lack of support across major processing engines, concerns about interoperability with existing tools, and the additional complexity it introduces without sufficiently clear benefits to justify replacing the stable and well-understood V1 formats.

Naturally, this raises concerns about why extending the V1 specification with global dictionaries would not run into the exact difficulties V2 did. Conducting this research within Databricks makes this investigation particularly worthwhile, as the managed table infrastructure provides a controlled environment where introducing enhancements to storage formats is more realistic than in open, heterogeneous systems. In such a context, the possibility of coordinating global dictionaries transparently for users becomes feasible and valuable, justifying the focus of this work.

2.3 Delta Lake and Metadata Management

Delta Lake is an open-source storage layer that brings ACID (Atomicity, Consistency, Isolation, Durability) transactions, scalable metadata handling, and unified batch and streaming data processing to existing data lakes. Built atop Apache Spark and cloud storage systems (e.g., Amazon S3, Azure Data Lake), Delta Lake introduces a transaction log and schema enforcement mechanisms that ensure reliability and consistency of data (14).

Traditional data lakes rely on append-only file systems, often leading to issues such as inconsistent reads, schema conflicts, and complex job orchestration. Delta Lake addresses these limitations by layering transactional guarantees and scalable metadata management over existing storage formats, particularly Apache Parquet. It ensures that read and write operations are isolated and consistent even in highly concurrent environments. In addition, it supports strong data typing with schema enforcement and evolution, enabling structured growth of datasets. Delta Lake also offers time travel functionality, allowing users to query previous versions of data, and it includes performance optimizations such as file compaction and data skipping, which leverage collected statistics like min/max values and null counts.

In this section, we present an overview of Delta Lake's architecture, focusing on its metadata management capabilities, and discuss how its design can support external or global dictionary management for Parquet data.

2.3.1 Architecture Overview

Delta Lake augments Parquet-based storage with a transaction log and metadata that reside alongside the data files.

Firstly, the core of Delta Lake is its *transaction log* (stored in the `_delta_log` directory), which tracks every change to a Delta table. The log consists of a sequence of JSON or Parquet files (log entries), each recording data operations (e.g., inserts, deletes, updates), schema changes, and metadata updates.

The log follows a Write-Ahead Log (WAL) pattern and allows for:

- **Atomic commits**: Multiple files can be added or removed atomically.
- **Versioning**: Each transaction corresponds to a new version of the table.
- **Concurrency control**: Enables multi-writer scenarios with optimistic concurrency.

Delta Lake stores metadata at multiple levels:

- **Table Metadata**: Table schema, partitioning, and configuration.
- **File Metadata**: For each file, Delta Lake tracks statistics such as minimum and maximum values per column, null counts, and record count.
- **Custom Metadata**: Delta Lake supports storing arbitrary key-value pairs, enabling extensibility.

2.3.2 Global Dictionary Storage in Delta Lake

Delta Lake's robust metadata and transaction mechanisms provide a natural foundation for externalizing dictionary compression by managing global dictionaries outside of individual Parquet files. The transaction log ensures consistency and versioning, making it feasible to manage dictionaries in a reliable and scalable manner.

Some of these features are of particular use in storing global Parquet data. Most important is the centralization delta lake provides. Any Parquet reader or writer can find a global dictionary file by querying the transaction log to find its file path or version. In addition, its version control allows safe updates to dictionaries with rollback support, meaning we can still read old data using old dictionary versions.

2.3.3 Challenges and Considerations

Despite its benefits, several challenges must be addressed for effective integration. First, performance overhead is a concern, as dictionary lookups during reads and writes must be optimized, techniques such as caching and batch retrieval may be necessary. Second, schema evolution introduces complexity: dictionary updates need to remain compatible with changes in schema while preserving backward compatibility. Finally, tool compatibility presents a hurdle, since many existing tools expect dictionaries to reside within Parquet files, whereas external dictionaries require adapted readers and infrastructure.

In the subsequent chapters, we propose and evaluate a system for managing external dictionaries using Delta Lake and explore its performance and storage benefits in realistic big data scenarios.

2.4 Databricks Tooling

Parquet files are typically read and written using the open-source Apache Parquet library. The main Java implementation of this library is known as **parquet-mr** (also called **parquet-java**)[1]. Although Parquet has implementations in multiple languages, the Java version is most relevant here because Apache Spark and much of the Databricks ecosystem are written in Scala and Java, which runs on the JVM and interfaces easily with Java libraries.

The reader within parquet-mr supports two primary modes: a basic sequential row-based reader (sometimes called the "naive" reader) and a more optimized vectorized reader. The vectorized reader processes data in columnar batches, which allows for both better memory locality and more efficient CPU usage. It also enables parallelism by allowing multiple row groups or column chunks to be read concurrently, which is essential for distributed data processing at scale.

Databricks, however, goes beyond the open-source tooling. Internally, it has developed multiple proprietary Parquet readers and writers, with various iterations aimed at improving performance and reducing overhead. One of the more recent developments is the *Native Frame Reader*, a high-performance reader written in C++. This reader is designed for greater speed and lower memory usage compared to the Java-based implementation. Despite these advantages, this thesis intentionally avoids relying on Databricks-specific optimizations. Our goal is to ensure the results are generalizable and reproducible in environments that use the open-source stack rather than Databricks-managed enhancements.

Finally, an important feature in the Databricks platform that ties together its I/O infrastructure is the **DBIO cache**. This cache is a block-level caching layer that prefetches entire blocks from disk into memory—even when only a small portion of the block is required for a query. This design choice minimizes the overhead of frequent disk I/O operations and can significantly improve read performance for workloads that repeatedly access similar data regions.

[1] https://github.com/apache/parquet-java

3

Related Work

The concept of global dictionaries sits at the intersection of several active areas of research in data storage and query optimization. While dictionary encoding has long been used as a standard compression technique in columnar formats like Parquet, global dictionaries remain rare and difficult to implement successfully. They have not seen widespread adoption, partly because new versions of Parquet are hard to deploy broadly in the database community. Nevertheless, we pursue this idea because, beyond the obvious storage benefits, global dictionaries have the potential to optimize compute by, for example, making dictionary-encoded values comparable across row groups or files and avoiding expensive string operations (see Chapter 6). By managing them within managed tables, we can sidestep the need for wider ecosystem support, which makes this a promising direction to be explored.

Existing literature related to this topic generally falls into three broad categories. The first includes work focused on improving dictionary compression itself, either through better encoding schemes or space-time tradeoffs, with some of these works touching on, or implicitly benefiting from, global dictionary structures (13, 15, 16, 17). The second category centers on optimizing access and lookup speed within dictionary-encoded systems, often introducing techniques such as ordered or mostly ordered dictionaries (18, 19, 20, 21). Finally, a smaller but growing set of research explicitly investigates mechanisms for transitioning from local to global dictionary management, aiming to reduce redundancy, improve consistency, and enable richer query optimizations across distributed datasets (18, 22, 23). In the following, we discuss representative work from each of these areas and situate our approach within this broader landscape.

3.1 Current Body of Literature

3.1.1 Metadata Management in Data Lake Table Formats

Modern data lakehouse table formats like **Delta Lake**, **Apache Hudi**, and **Apache Iceberg** introduce an explicit metadata layer on top of flat files to support ACID transactions and efficient queries. **Delta Lake** maintains a transaction log in cloud storage

(with periodic Parquet "checkpoint" files) that tracks all files and table versions (14). This design avoids expensive object store directory listings by allowing query engines to quickly read prepared metadata even for billions of partitions instead of enumerating files on each read (14, 24). **Hudi** similarly writes timeline delta-logs and also builds a specialized metadata table that indexes file names and statistics (24, 25). This Hudi metadata table acts as a cached lookup for file listings, dramatically speeding up query planning on large tables by bypassing repeated full directory scans (25). In contrast, **Iceberg** employs a hierarchical metadata tree: a top-level JSON table definition points to manifest lists and manifest files that enumerate the data files and their statistics (24, 26). Each commit in Iceberg creates a new snapshot and atomically updates the top-level pointer. This hierarchy reduces write contention and allows readers to load just a handful of small metadata files (the latest table metadata and a few manifests) to discover relevant data files, rather than scanning a long log (24). The trade-off is that Delta Lake and Hudi can leverage distributed processing (e.g., Spark jobs) to scan or compact their metadata for extremely large tables, whereas Iceberg's single-file manifests are simple and efficient for moderate table sizes but may become a planning bottleneck at extreme scale (24).

Recently, the **DuckDB** project proposed **DuckLake**, an alternative take on the metadata layer. DuckLake forgoes the file-based log or manifest approach and instead stores table metadata (schema, file index, statistics) inside a single SQLite database accessed via DuckDB (27). In essence, DuckLake replaces the myriad of JSON/Parquet metadata files with a traditional SQL catalog stored in one file (27). This design can greatly reduce round-trip latency to cloud storage and leverage indexing for faster lookups. For instance, a global dictionary in DuckLake could simply be another table indexed by values or codes, making dictionary lookups or updates transactional and fast. The trade-off, however, is compatibility and openness: whereas formats like Delta and Iceberg use human-readable, file-based metadata (JSON, Avro) for engine interoperability, DuckLake's approach ties metadata to a specific database engine format (27). Nonetheless, it demonstrates an extreme point in the design space: pushing all metadata (and potentially dictionaries) into a single ACID database to minimize overhead. This underscores how crucial the metadata layer is for any cross-file structure like a global dictionary, whether via log files, manifest indexes, or an embedded database, the architecture must efficiently broadcast and version these dictionaries across the data lake.

3.1.2 Emerging Columnar Formats and Compression Techniques

Beyond table formats, a parallel thread of research is innovating on the columnar file formats themselves to improve compression and performance. Several new formats aim to outperform standard Parquet and ORC in both size and speed. **FastLanes** (11) redesigns columnar compression layouts to exploit modern CPUs. It generalizes bit-packing with a 1024-bit virtual SIMD register model and a unified tuple layout, enabling decoding of dictionary, delta, frame-of-reference and RLE encodings at over 100 billion integers per second in purely scalar code (11). The key insight is that Parquet's adaptive bit-packing (with

variable run lengths) hinders vectorization, whereas FastLanes uses fixed-size interleaved blocks that compilers can auto-vectorize (11). This yields extremely fast decompression, with reported speedups of 40× over Parquet decoding while still reducing storage size by approximately 40% (11). Such advances illustrate that one can have high compression and high throughput by carefully co-designing the format with hardware capabilities.

Another example is **BtrBlocks** (28), a columnar format purpose-built for cloud data lakes. BtrBlocks analyzes columns in small blocks (e.g., 1MB chunks) and chooses the best lightweight encoding for each chunk (such as bit-packing, dictionary, delta encoding). Importantly, all encodings in its arsenal are designed for fast decompression rather than maximum compression (28). The authors argue that in disaggregated storage (e.g., S3 and 100 Gbit networks), Parquet often becomes CPU-bound due to decoding costs, something that global dictionaries may likely solve through making decoding unnecessary in many cases (see Chapter 6). By using simpler schemes and compressing each block independently, BtrBlocks can significantly speed up scan throughput while still achieving compression on par with or better than Parquet with ZSTD (28). On TPC-H data, BtrBlocks was shown to compress approximately 1.5× smaller than Parquet+Zstd and scan faster, by eliminating the heavy per-tuple decoding logic of Parquet in favor of streamlined operations (28).

Other new formats prioritize not just speed but also flexibility and advanced capabilities. **Vortex** (29) is an extensible columnar format built on Apache Arrow's in-memory structures, enhanced with advanced compression and embedded metadata. A Vortex file is essentially a collection of Arrow arrays with additional metadata such as per-column min/max statistics and optional custom compute annotations (29). In its canonical uncompressed form, a Vortex file resembles an Arrow IPC file, enabling seamless zero-copy conversion to Arrow in-memory structures (29). Vortex adds column-level compression and stores summary statistics to enable efficient predicate pushdown (29). For example, min/max values in the footer allow query engines to skip segments that do not match a filter (29). The format is explicitly designed to be extensible, so developers can introduce new compression methods or specialized column types without defining a new file format (29). Overall, Vortex combines Arrow's interoperability with columnar database performance while enabling rich metadata such as internal indexes or custom operations.

Lance (30) is a newer format from the machine learning community aimed at balancing fast sequential scans with fast random access. It was designed for workloads like vector search, where systems often need to fetch small subsets of data rather than scan entire columns (30). Parquet struggles in these scenarios because even small reads require scanning footers and row groups. Lance instead organizes data as a single logical row group with internal indexing, enabling direct access to any row or batch (30). Adaptive structural encodings, such as row ordering and column clustering, further optimize access patterns (30). While it still applies compression (such as LZ4 or dictionary encoding), Lance's strength lies in its structural layout (30). By avoiding the overhead of multiple row groups and embedding auxiliary indexes, Lance significantly improves point lookups and small-range

scans, particularly in cloud storage environments (30). Early results show scan performance similar to Parquet on full reads, with much lower latency for selective queries (30), making it well suited for machine learning workloads with mixed access patterns.

3.1.3 Dictionary Compression in Columnar Formats

A substantial body of research focuses on improving how dictionaries themselves compress data. Müller et al. (15) propose adaptive dictionary compression for in-memory databases, selecting the optimal compression technique dynamically based on observed access patterns. Their system chooses between different compression formats (e.g., bit-packing vs. Huffman coding) depending on data properties, showcasing how flexible dictionary management can achieve strong space and speed tradeoffs.

Doblander (16) explores dictionary compression in publish/subscribe systems, where minimizing the cost of transmission is paramount. One notable insight from this work is the emphasis on online adaptation: dictionaries must evolve as new data arrives, a principle directly relevant to our goal of supporting incremental updates.

Boncz et al. (13) introduce FSST, a very lightweight string compression scheme based on static symbol tables. Although FSST is not explicitly global, its focus on extremely fast decompression and high random-access efficiency align with the performance goals our system must meet when externalizing dictionaries.

Finally, Tong et al. (17) study principled dictionary pruning for corpus compression. Using relative Lempel-Ziv (RLZ) coding, they show that aggressive pruning of rarely used substrings can dramatically shrink dictionary size with little loss of compression quality. This result highlights a tradeoff we must consider: global dictionaries must be compact but also broadly representative across datasets.

3.1.4 Optimizing Dictionary Access and Lookup

A second line of research focuses on dictionary layout and lookup optimizations. Antoshenkov (20) proposes order-preserving dictionary compression, enabling compressed data to be compared directly without decompression. This work is crucial because global dictionaries must often support efficient range queries; however, strict order preservation may require large dictionaries or complicated encoding, limiting practicality.

Binnig et al. (19) extend this idea by designing dictionaries that are both compressed and order-preserving, allowing faster equality and range scans in main-memory column stores. They demonstrate that minor sacrifices in compression ratios can yield large performance benefits when query speed is the primary concern.

Martínez-Prieto et al. (21) also study RDF dictionaries, emphasizing the need for both fast ID-to-string lookups and efficient string-to-ID encodings. Their findings underscore that even slight inefficiencies at lookup time can dominate query costs at scale.

Liu et al. (18) propose "Mostly Order Preserving Dictionaries" (MOP), which strike a compromise between fully ordered dictionaries and purely random ones. Instead of guaranteeing strict ordering, they maintain order for a large fraction of the values, improving

the efficiency of range queries while tolerating some disorder. This model is especially appealing for dynamic or incrementally updated datasets and hints at hybrid strategies like those we consider in this thesis.

3.1.5 From Local to Global Dictionaries

The third group of works moves beyond local dictionaries and explores how dictionaries can be shared, reused, or coordinated across datasets.

In a complementary line of work, Gubner et al. (31) propose the Unique Strings Self-aligned Region (USSR), a lightweight, dynamic query-time dictionary that opportunistically compresses and caches frequent strings to reduce memory usage and improve cache locality. Unlike persistent global dictionaries as explored in this thesis, the USSR is constructed on-the-fly during query execution and resides entirely in memory, avoiding the overhead of maintaining consistent dictionaries across files or transactions. Nevertheless, it highlights the performance potential of dictionary-based approaches beyond static storage optimization, supporting the broader view that dictionary management can benefit both storage and compute layers.

Foufoulas et al. (22) introduce an adaptive compression scheme for string columns that uses "differential dictionaries" across file blocks. Inspired by I-frames and P-frames in video compression, their method selectively uses local or differential dictionaries to balance compression and lookup speed. They show that by reducing redundancy between blocks, scan speed can improve dramatically, up to an order of magnitude in some cases. While their system does not fully externalize dictionaries, it demonstrates the feasibility of cross-file dictionary sharing, an idea this thesis extends into a more explicit, transactional system with Delta Lake.

Lemke et al. (23) describe techniques for operating directly on compressed data in SAP's TREX engine. Although their work primarily focuses on compression for memory savings, they stress that direct query processing over compressed representations (e.g., dictionary-encoded integers) is critical for modern OLAP systems. They note that maintaining a balance between compression efficiency and query speed is essential.

Apache CarbonData provides perhaps the most complete implementation of a global dictionary (32). It includes support for generating global dictionaries by reading/preprocessing the entire dataset before the table's first write, as we do in Chapter 4. It even supports incremental updates, through an append-only dictionary. However, it has no support for any hybrid dictionary solution to prevent premature fallback (see Section 4.5) and thus achieves worse compression ratios on TPC-DS. It also has no support for global dictionary versioning workflows (see Figure 5.1).

Liu et al. (18) (MOP again) illustrate that partially global dictionaries can still yield most of the benefits of full globality while remaining flexible for evolving datasets. Their notion of reserving space for "unknown" values and handling ordering conflicts is an interesting solution in our use case, where we might want to segment the global dictionary between a global and local section.

Lastly, while much of the literature treats Parquet as a fixed standard, recent work by Saeedan and Eldawy demonstrates that it can, in fact, be extended to support new data types and compression methods (33). Their design of Spatial Parquet integrates custom geospatial encodings and lightweight indexing into the standard Parquet layout, while remaining compatible with existing readers and preserving columnar benefits. This reinforces the view that Parquet is not a static format but an evolving platform capable of accommodating novel features. Their work supports the feasibility of this thesis's aim: to fundamentally alter and augment a key facet of Parquet's structure.

3.1.6 Summary and Gaps in Existing Work

While extensive research addresses dictionary compression, lookup optimization, and even preliminary forms of cross-file dictionaries, several gaps remain:

- **Transactional Consistency**: Existing global or semi-global dictionary proposals do not integrate with transactional systems like Delta Lake, where concurrent updates and consistent snapshots are necessary.
- **Incremental Global Updates**: Few solutions support incremental extension or replacement of global dictionaries without rewriting large portions of the dataset.
- **Hybrid Global-Local Strategies**: There is little systematic exploration of hybrid models that combine global dictionaries with smaller local augmentations for flexibility and compatibility.
- **Compatibility with Standard Formats**: Most prior work assumes bespoke storage formats; adapting global dictionaries cleanly into the Parquet + Delta Lake ecosystem is still an open challenge.

This thesis advances the field by proposing a practical architecture for global dictionaries that satisfies ACID properties through Delta Lake's transaction log, supports efficient hybrid dictionary designs, and can be extended to incremental update scenarios without major disruption to existing Parquet infrastructure.

4

Global Dictionary Compression for Static Datasets

The goal of this work is to demonstrate that global dictionaries can reduce the storage footprint of Parquet files by eliminating redundant local dictionary pages across row groups and files. In this section, we lay out the motivation, assumptions, and expected benefits of the approach, particularly in the context of static datasets with stable schemas.

Parquet encodes each row group's column chunk independently, including its own dictionary page if dictionary encoding is applied. This design favors local parallelism but leads to significant redundancy when a column has a (large) finite set of recurring values. Global dictionaries address this by externalizing a single dictionary per column and reusing it across all chunks and files in the dataset. This decoupling enables compression benefits and consistency in encoding.

We measure compression effectiveness using the following ratio (34):

$$\text{Compression Ratio} = \frac{\text{Size}_{\text{Baseline Parquet}}}{\text{Size}_{\text{Parquet + Global Dict}}} \tag{4.1}$$

This compares the total on-disk footprint of the dataset when using global dictionaries (including both Parquet and dictionary files) against a baseline of conventional Parquet files using local dictionaries. When useful, we will also refer to *space saving* (expressed in %) to emphasize proportional improvement, defined as:

$$\text{Space Saving} = 1 - \frac{\text{Size}_{\text{Parquet + Global Dict}}}{\text{Size}_{\text{Baseline Parquet}}} \tag{4.2}$$

For example, a 10% space saving is equivalent to a compression ratio of 1.11

While the idea of reusing a global dictionary might seem universally beneficial, its actual impact depends heavily on the characteristics of the column being encoded. Our hypothesis is that the approach is *most effective* for columns with **medium-sized domains**, typically with cardinalities in the hundreds or low thousands. The reasoning is as follows:

- **Low-cardinality columns** already compress well using local dictionaries because their dictionary pages are small. Moving to a global dictionary introduces metadata overhead and may offer negligible improvement.
- **High-cardinality columns** often exceed the capacity of dictionary encoding altogether, causing a fallback to plain encoding. These are generally unsuitable for dictionary-based compression, whether local or global.
- **Medium-cardinality columns** strike a balance: their domains are large enough that local dictionaries are expensive to replicate across row groups, but still small enough to fit comfortably into a single global dictionary. Reusing this dictionary across chunks removes redundancy while maintaining encoding benefits.

This sweet spot is common in analytical workloads, where such columns include categorical identifiers like product categories, country codes, device types, or user segments.

4.0.1 Scope and Exclusions

In this static compression setting, we assume the entire dataset is available in advance and the global dictionary is constructed in a preprocessing step before any files are written. This is realistic for batch ETL pipelines or data warehousing scenarios where table content is rewritten in bulk.

While this chapter introduces a structural change to Parquet file layout by replacing per-chunk dictionaries with shared, global ones, we limit our scope strictly to evaluating the effects on **storage compression**. Specifically, we measure the total on-disk size of Delta tables and global dictionary files combined, without attempting to optimize write speed, read latency, or memory usage.

Several adjacent concerns are explicitly excluded from our implementation and subsequent evaluation of this design:

- **Query execution performance.** Although global dictionaries may improve query performance via direct encoded value comparison or faster filtering (see Chapter 6), we do not investigate such effects here.
- **Write-time performance.** We do not evaluate the overhead introduced during dictionary generation or encoding. In some scenarios, these operations may slow down data ingestion pipelines.
- **Reader optimizations.** We make minimal changes to the read path to support external dictionaries, but do not explore cache-aware lookup schemes, in-memory compaction, or lazy decoding.
- **Advanced metadata-driven strategies.** While external dictionaries enable possibilities like adaptive skipping, index building, or predicate pushdown based on frequency histograms, these are considered out of scope for this work.

We intentionally adopt this narrow focus in order to isolate the question: *Can global dictionaries, in their simplest form, improve storage compression in analytical data lakes?*

Other potential benefits or costs while promising for future work are not addressed in this evaluation.

4.1 Constructing Global Dictionaries

Global dictionaries must be built before any files are written, making their construction a critical first step in applying the static compression strategy described earlier. This section outlines the process of generating a single, reusable dictionary per column, including value selection, normalization, encoding, and storage.

We assume that the entire static dataset is available and represented as a logical Delta table prior to write time. This includes the full contents of each column and its associated metadata. We do not require that the data is materialized in memory at once, but we assume access to column-level scans or aggregations to extract frequent values efficiently.

4.1.1 Value selection

The core idea behind a static global dictionary is to precompute a list of the most common values in a column and assign each a fixed index. In this chapter, we use a simple frequency-based approach: the top-k most frequent values are inserted into the dictionary, where k is constrained by either a target memory budget or a fixed cardinality cap.

The frequency distribution is computed over the full input dataset. The goal is to balance two conflicting objectives:

- **Coverage:** The dictionary should contain as many of the actually-used values as possible to reduce fallback.
- **Compactness:** The dictionary should attempt to remain small and avoid introducing overhead. Moreover, allowing the dictionary to become too large may indicate the dictionary is working on a large-domain column (e.g. timestamps), and likely has limited use.

Several heuristics can be used to select which values to include in a global dictionary, each aiming to balance dictionary compactness and overall compression benefit:

- **Frequency-based selection:** Include the most frequent values across the column, under the assumption that coverage is maximized.
- **Length-frequency product:** Score values by multiplying their string length by their frequency, prioritizing values that consume the most space overall.
- **Entropy reduction:** Choose values that, if dictionary-encoded, most reduce the overall entropy of the column, measured via estimated information gain.

In this chapter, we adopt frequency-based selection as a practical and effective default. It is simple to compute using standard aggregation tools, produces stable results across a variety of distributions, and typically ensures that the most common values, those most likely to repeat across row groups, are encoded. While other strategies may offer additional gains in specific scenarios (e.g., where long low-frequency values dominate), they require

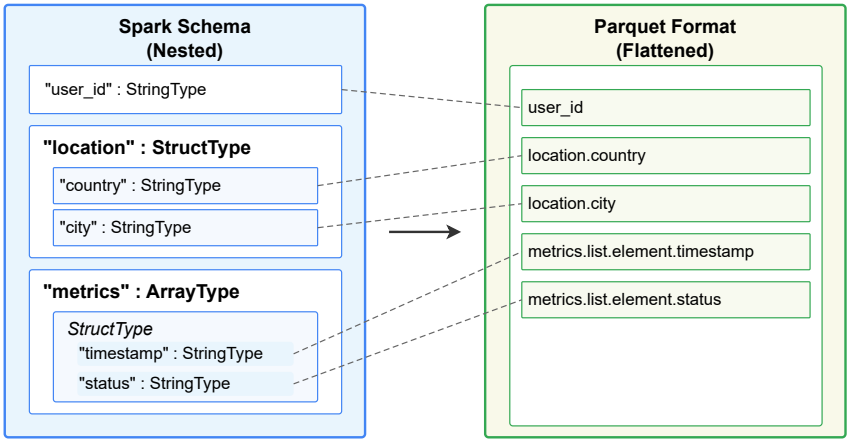

Figure 4.1: Example of a Spark DataFrame schema with nested fields and mixed types. Global dictionaries must normalize fields like `location.city` and `metrics[].status` into flat, byte-encoded values for compatibility with Parquet encoders.

more metadata, more passes over the data, or additional tuning. Our focus here is to establish a robust baseline for static dictionary compression without introducing unnecessary complexity.

4.1.2 Value normalization

Before values are inserted into the dictionary, they must be normalized into the form expected by the Parquet writer. This ensures consistency between the top-k keys as returned by the Spark job, and the byte representation that *parquet-mr* will eventually read back. This includes:

- Serializing nested or complex types (e.g., `struct`, `array`, or `map` types) by flattening their schemas and converting their leaf nodes into a deterministic byte format.
- Canonicalizing nulls and corner cases (e.g., treating `null` and `None` consistently).
- Enforcing ordering rules (e.g., UTF-8 lexicographic order for strings) to guarantee repeatable index assignments.

Figure 4.1 shows an example Spark schema that includes both primitive and nested types, and their corresponding Parquet schema. A column like `location.city` is easily extracted and normalized, but arrays of structs (e.g., `metrics.status`) require consistent flattening and serialization to avoid mismatches between dictionary encoding and the actual written output. It is critical that we rename the flattened columns to what Parquet expects them to be called. For example, `metrics.status` is a single column called `metrics.list.element.status` in the Parquet format. Any inconsistency may result in invalid lookups or decoding errors at read time. We ensure that normalized dictionary entries match the exact byte form emitted by the Parquet writer.

Once the normalized set of dictionary values is finalized. Their order/offsets in the global dictionary will become the values that will be written to data pages via RLE/bit-packing

compression.

The choice of index ordering can influence performance if the dictionary is ever used for range queries or binary search, but in this chapter, we do not exploit this ordering beyond compact representation. A simple frequency-descending or lexicographic order is sufficient.

4.1.3 Storage format

Global dictionaries are written to standalone files, separate from any Parquet data. These files must be:

- **Efficient to load:** Readers must be able to map encoded indices back to values without scanning the full file.
- **Self-contained:** Each file is structured *exactly* as a normal dictionary page, with a header containing metadata about the encoding, number of values, size, etc. and the data portion.
- **Compatible:** The format must match the reader and writer implementation in byte layout and encoding (e.g., prefix length for variable-length strings).

In our implementation, each dictionary is stored as a single file in a dedicated dictionary directory. For reference, the file layout mimics that of the parquet-mr dictionary page (see Figure 2.1).

4.1.4 Resource considerations

Building global dictionaries over large datasets raises practical questions around memory usage, compute cost, and parallelism. While our implementation supports single-pass aggregation and approximate sketches (see next chapter), the approach in this chapter assumes that sufficient resources are available to compute exact frequency counts per column.

These potential optimizations are orthogonal to the main results and not necessary for evaluating the feasibility of global dictionary compression in static datasets.

4.2 Modifying the Write Path

To apply global dictionary compression effectively, the Parquet writer must be modified to support encoding values using an external dictionary. This requires changes to the encoding logic, fallback handling, and metadata emission for each column chunk. In this section, we describe the integration strategy, focusing on minimal disruption to the standard write pipeline.

4.2.1 Dictionary Creation

Before each write operation, a dedicated dictionary generation step is triggered. This step is implemented as a Spark job that performs a `groupBy` followed by a `count` on each target column to compute the global frequency of every distinct value. Once the counts are

available, a filter is applied based on a minimum count threshold to control dictionary size and reduce noise from rare or unhelpful values. The resulting top values are then serialized into global dictionary files, which are reused across all row groups during encoding.

This preprocessing step is mandatory for enabling global dictionary compression, as the encoding logic depends on the availability of a complete and filtered dictionary in advance. However, this also introduces a major limitation: the dictionary generation is relatively expensive and cannot be skipped or delayed. As such, it makes the approach unsuitable for incremental updates or streaming ingestion, where data is appended in small batches over time. In these scenarios, re-running the full dictionary generation job before every write would introduce prohibitive overhead and negate the performance benefits of dictionary reuse. Consequently, this design is best suited to static datasets that are written in batch mode, where the cost of a one-time dictionary generation is amortized over the entire write workload.

4.2.2 Encoding Control Flow

The key decision during writing is whether the values in a given column chunk can be entirely encoded using the global dictionary. If so, we emit encoded indices into data pages and omit the dictionary page entirely. If not, we fall back to an alternative encoding strategy, such as plain encoding or a hybrid local+global approach. The control flow is shown in Algorithm 1.

This flow preserves compatibility with the Parquet specification: values are still written as encoded pages, but dictionary pages are omitted when the dictionary is externalized. The global dictionary is assumed to be discoverable in a static location with a predictable file naming schema.

4.2.3 Fallback Strategy

Fallback is triggered if even a single value in a chunk is missing from the global dictionary. In our static implementation, we treat this as an all-or-nothing decision to preserve simplicity and avoid mixed modes within the same chunk. Possible fallback strategies include:

- **Plain encoding:** Store raw, uncompressed values as-is.
- **Local dictionary:** Build a local dictionary for this chunk and proceed with standard RLE_DICTIONARY encoding.
- **Hybrid mode:** Use global dictionary for encodable values and add local indices for unknowns (discussed in Section 4.5).

In this section, we prefer plain or local fallback depending on dataset characteristics and Parquet writer capabilities.

Algorithm 1 Writing Column Data with Global Dictionary in Parquet

```
1: function WriteColumnWithGlobalDictionary(columnData, globalDictionary-
   Path)
2:     globalDict ← LoadGlobalDictFromFile(globalDictPath)
3:     lookupMap ← BuildDictionaryLookup(globalDict)
4:     encodedValues ← []
5:     maxDictID ← 0
6:     needsFallback ← false
7:     for each value in columnData do
8:         dictID ← lookupMap.get(value)
9:         if dictID exists then
10:            append dictID to encodedValues
11:            maxDictID ← max(maxDictID, dictID)
12:        else
13:            needsFallback ← true
14:            break
15:        end if
16:    end for
17:    if not needsFallback then
18:        bitWidth ← CalculateMinBitWidth(maxDictID)
19:        header ← [bitWidth]                    ▷ 1-byte header with bit width
20:        encoder ← RLEBitPackingEncoder(bitWidth)
21:        for each id in encodedValues do
22:            encoder.writeInt(id)
23:        end for
24:        compressedData ← header+ encoder.getBytes()
25:        return compressedData
26:    else
27:        return WritePlainEncoding(columnData)      ▷ Fall back to plain encoding
28:    end if
29: end function
```

4.2.4 Static Dictionary Lookups

Unlike local dictionary encoding which commits the dictionary page after writing a chunk, no dictionary page is written to the Parquet file when using a global dictionary. Instead, we rely on external lookup at read time, and encode this dependency via metadata.

To ensure that readers can resolve the correct global dictionary during decoding, each dictionary file must be named predictably according to the column:

```
"user_id" ->
        "path/to/delta/dictionaries/user_id.dict"
"location.city" ->
        "path/to/delta/dictionaries/location.city.dict"
```

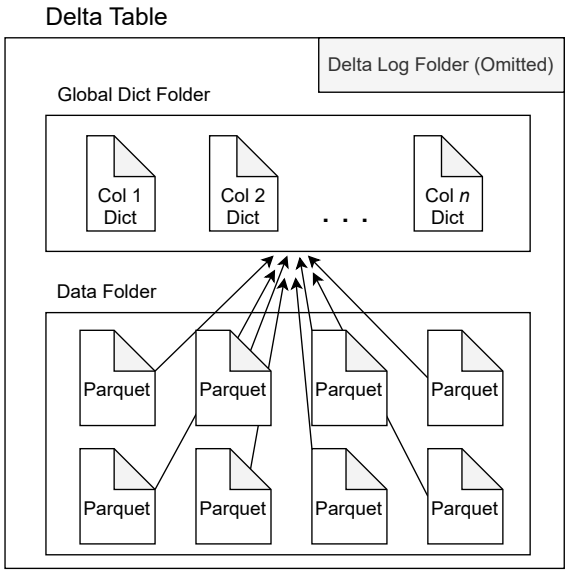

Figure 4.2: Illustration of a global dictionary file structure, including metadata, dictionary entries, and value offsets for fast access.

All dictionary pages are stored in the delta table's folders under a separate *dictionaries* directory and named as *column_ name.dict* (see Figure 4.2 for a high level design of the Delta table). The path to the delta table's *dictionaries* directory is passed in via a key:value pair in the config:

```
parquet.global.dictionary.directory.path ->
path/to/delta/dictionaries
```

This highlights the need for the original delta global dictionary page writer to infer the column name *as Parquet would see it.* For example, from the schema in Figure 4.1, the column, `metrics.timestamp` (as Spark sees it), cannot be aggregated and written to the file *metrics.timestamp.dict* as Parquet operates using the column name, `metrics.list.element.timestamp` and will look for *metrics.list.element.timestamp.dict.*

4.2.5 Encoding Type Identifier

To ensure that readers can distinguish data pages encoded with global dictionaries from those using standard Parquet encodings, we introduce a new encoding identifier embedded in the data page header. This is necessary because global dictionary pages omit the dictionary payload and rely on an external lookup, breaking the default assumptions of the Parquet reader.

We define a new custom encoding constant:

```
RLE_GLOBAL_DICTIONARY
```

This encoding is used in the Parquet data page header's `encoding` field in place of `RLE_DICTIONARY`.

It is worth pointing out that introducing a new encoding is perhaps one of the more "invasive" changes, as it requires the Parquet Thrift format to be updated. Specifically, the `Encoding` enum in the Parquet format specification must be extended to include `RLE_GLOBAL_DICTIONARY`, which affects both the writer and all downstream readers. While Parquet is designed to tolerate unknown encodings gracefully (typically by failing to decode pages), full support requires recompiling the Thrift definitions and updating dependent tooling. This limits out-of-the-box compatibility but enables unambiguous decoding and better long-term maintainability when the extension is properly documented and adopted.

4.3 Reading with Global Dictionaries

Once a column chunk has been encoded using a global dictionary, the reader must be modified to decode the values using that external dictionary instead of expecting an in-file dictionary page. In this section, we describe the extensions necessary to support global dictionary decoding with minimal disruption to the existing Parquet read path.

The first step in decoding is to determine whether a given data page was written using a global dictionary. This is achieved by checking the `encoding` field in the data page header for the custom identifier `RLE_GLOBAL_DICTIONARY` (see Section 4.2.5).

Once detected, the reader must resolve the correct dictionary file to load. In our implementation, this is based on a convention-driven filename scheme (see Section 4.2.4). The reader reconstructs the dictionary path using:

- The fully-qualified column name, as seen by Parquet (e.g., `metrics.list.element.status`).
- A configuration or table-level pointer to the dictionary directory.

If the dictionary file cannot be located or parsed, the reader will fail fast with an informative error, as decoding is impossible without access to the dictionary mapping.

Once the external dictionary is loaded into memory, decoding proceeds as follows:

1. Read the one-byte header from the data page to extract the bit width used for the encoded indices.
2. Decode the remaining bytes using standard RLE/BitPacking logic to recover the integer indices.
3. Use the indices to look up the actual values in the loaded global dictionary.

This mirrors the decoding logic used for local dictionaries, with the primary difference being that the dictionary is not embedded in the file and must be loaded independently. The decoding function is shown in Algorithm 2.

Caching is **disabled** for both local and global writers and readers. This is due to difficulty of reaching the DBIO cache from within parquet-mr. The DBIO cache is a block level cache that can cache entire blocks containing one or multiple dictionary pages, depending on their

Algorithm 2 Decoding a Column Chunk with Global Dictionary

```
1: function ReadColumnWithGlobalDictionary(dataPage, globalDictPath)
2:     globalDict ← LoadGlobalDictFromFile(globalDictPath)
3:     bitWidth ← dataPage.readHeaderByte()
4:     decoder ← RLEBitPackingDecoder(bitWidth, dataPage)
5:     decodedValues ← [ ]
6:     while decoder.hasNext() do
7:         dictID ← decoder.readInt()
8:         value ← globalDict.getValue(dictID)
9:         append value to decodedValues
10:    end while
11:    return decodedValues
12: end function
```

sizes (see Section 2.4). However, while possible to implement, bringing a core Delta/Spark dependency into parquet-mr is too cumbersome to bring into scope in this chapter.

Global dictionary decoding introduces a new point of failure, external dictionary resolution. Still, the system remains robust under reasonable assumptions:

- If a data page uses `RLE_GLOBAL_DICTIONARY` but no dictionary is found, the reader fails explicitly, avoiding silent corruption.
- If an unknown encoding type is encountered, standard Parquet readers will fail gracefully, as per the format specification.
- Since the dictionary is referenced by file path and not by content hash, care must be taken to maintain dictionary immutability and reproducibility across versions (e.g., during Delta replay events).

4.4 Compression Evaluation

To evaluate the effectiveness of global dictionary compression in static datasets, we conducted extensive benchmarks across four diverse datasets. This section presents our findings in terms of overall compression ratios and space savings, along with a deeper per-column analysis to understand which structural features lead to the most (or least) benefit from global dictionaries.

4.4.1 Benchmark Setup and Datasets

Many works in the field of DBMS's tend to rely on common synthetic datasets like TPC-DS. While this is not inherently bad, it opens experiments to the dataset and benchmark query bias of just using one benchmark (35). Our evaluation is based on four datasets (three of them being real-world) that vary significantly in schema complexity, data size, and distribution characteristics:

Table 4.1: Table-level compression results across datasets. Global Dict Total Size is broken down into Parquet and Dictionary components. Higher ratios indicate better compression from global dictionaries.

Dataset	Baseline Size	Parquet Size (Global Dict)	Dict Size	Total Size	Ratio	Space Saving
TPC-DS 10GB	4.50 GB	3.74 GB	0.25 GB	3.99 GB	1.125	0.1111
Common Government	7.67 GB	7.85 GB	0.28 GB	8.13 GB	0.944	-0.059
NYC	1.04 GB	1.09 GB	0.05 GB	1.14 GB	0.907	-0.102
Library Inventory	5.68 GB	5.65 GB	0.12 GB	5.77 GB	0.985	-0.015

- **TPC-DS 10GB:** A well-known analytical benchmark with synthetic data; includes a mixture of categorical, numeric, and timestamp fields pertaining to a company's operations. The synthetic nature of the dataset causes its distributions to be rather uniform in value and frequency.
- **CommonGovernment[1] (PublicBI):** Contains detailed records of U.S. federal government procurement contracts, including agency information, contract values, vendor data, and classification codes.
- **NYC[2] (PublicBI):** Captures 311 service request data for New York City, documenting complaints by location, agency, type, and resolution status.
- **Library Collection Inventory[3] (NextiaJD):** Lists inventory details of a library's collection, including bibliographic data, item type, and location within the system.

All datasets were processed using the same pipeline. Each dataset was written twice, once using conventional local dictionaries and once using global dictionaries, using a frequency threshold of $n = 1$ (see Section 4.4.2) and a maximum global dictionary page size of 2 MB. Table 4.1 summarizes the total size and compression ratios for each dataset. It highlights that while global dictionaries can significantly reduce file size in synthetic benchmarks, they may regress performance on real-world data with sparse or inconsistent domain reuse.

4.4.2 Frequency Threshold Sensitivity (TPC-DS)

A key design parameter in constructing global dictionaries is the minimum number of times a value must occur in a column to be included. This frequency threshold determines how aggressive the dictionary is in covering rare values, which in turn affects both the size of the dictionary and the likelihood of fallback to plain encoding. In this section, we analyze the impact of varying this threshold in the TPC-DS 10GB dataset.

[1] https://event.cwi.nl/da/PublicBIbenchmark/CommonGovernment
[2] https://event.cwi.nl/da/PublicBIbenchmark/NYC
[3] https://event.cwi.nl/da/NextiaJD

Table 4.2: Compression outcomes on TPC-DS (10GB) with varying minimum key occurrence thresholds, with sizes in gigabytes (GB). Higher ratios indicate better overall compression.

Min Count	Baseline Size (GB)	Global Parquet Size (GB)	Global Dict Size (GB)	Total Global Size (GB)	Ratio
≥ 5	4.188	3.899	0.086	3.985	1.050
≥ 4	4.188	3.889	0.093	3.982	1.052
≥ 3	4.188	3.865	0.103	3.969	1.054
≥ 2	4.188	3.841	0.118	3.959	1.058
≥ 1	4.188	3.569	0.152	3.721	1.125

Experimental setup. We evaluated five minimum thresholds for a value's number of appearances in the dataset to be included in the global dictionary. The results are summarized in Table 4.2.

The results show a clear trend: **lower thresholds consistently yield better compression**, with the best result achieved when **all values are included** (i.e., threshold ≥ 1). This is somewhat counter to the intuition that larger dictionaries might incur substantial storage or indexing overhead. In practice, the size of the dictionary file grows modestly (from 86MB to 152MB across thresholds) while the benefit from being able to encode more chunks is substantial.

Allowing more values into the global dictionary increases the probability that an entire column chunk is encodable using dictionary indices. If even a single value is missing from the dictionary, the entire chunk falls back to plain encoding, which is significantly less efficient. As such, the primary limiting factor is not dictionary size, but the ability to prevent fallback.

Even in the most permissive configuration, the global dictionary represents only ∼4% of the dataset's total size. This negligible figure is reflected in the other datasets as well, as shown in Table 4.1. This confirms that, in practice, **dictionary size is not a bottleneck as far as compression is concerned**. As long as the dictionary remains small enough to be loaded into memory and indexed efficiently, a more inclusive approach yields better outcomes.

In the case of TPC-DS, the best compression is achieved when no threshold is applied at all. While this may not generalize to all datasets, it highlights that overly aggressive pruning of infrequent values can backfire by forcing chunks to fall back to less efficient encodings. A permissive threshold, even down to 1, is justified when dictionary overhead remains small.

4.4.3 Column-Level Compression Analysis

While global dictionaries may yield promising table-level compression on some datasets, their effectiveness ultimately hinges on individual column characteristics. This is because,

in a production-level application of this technology, it would be wasteful to generate global dictionaries for columns (or datasets) that have no use for them. In this section, we investigate which column-level features most strongly correlate with compression gains so that it can be predicted which columns might benefit from global dictionary encoding before encoding anything. We analyze both successful and failed compression cases across TPC-DS and other benchmark datasets.

We collected a range of metrics per column to explain compression performance, including:

- **Data type**: (e.g., string, decimal)
- **Row count**, **cardinality**, **uniqueness proportion** ($\frac{\text{cardinality}}{\text{row count}}$)
- **Null percentage**
- **Coefficient of variation**, **average value length**, **entropy**
- **Min**, **max**, **percentiles** (25/50/75)

Among the collected metrics, we further explain the rationale behind the **coefficient of variation (CoV)** and **Shannon entropy** (36). These provide complementary perspectives on value distribution.

The Coefficient of Variation measures the relative dispersion of a dataset, defined as:

$$\text{CoV} = \frac{\sigma}{\mu}$$

where σ is the standard deviation and μ is the mean. CoV is unitless and useful for comparing variability across columns with different scales. In the context of dictionary compression, a high CoV may indicate noisy numeric data or long-tailed distributions, which reduce the likelihood of repeated values and increase fallback risk.

Shannon entropy, on the other hand, quantifies the unpredictability or uniformity of a value distribution. For a discrete set of values with empirical probabilities p_i, entropy is defined as:

$$H(X) = -\sum_{i=1}^{n} p_i \log_2 p_i$$

Higher entropy implies a more uniform distribution (i.e., less repetition), which typically leads to worse dictionary compression. Conversely, columns with a few dominant values (low entropy) are highly compressible and benefit significantly from dictionary reuse.

We considered several other statistical measures, but omitted them for practical or interpretability reasons:

- **Kurtosis**: Measures the "tailedness" or extremity of values. While potentially useful for numeric columns, it is difficult to interpret in the context of compression and often correlates with CoV.
- **Skewness**: Captures asymmetry in the distribution. Although helpful for understanding certain long-tailed patterns, its additional insight over entropy or CoV was limited in our analysis.

- **Gini coefficient**: Often used for inequality, this metric can reflect concentration of values but is computationally intensive and non-intuitive for column-level heuristics.
- **Jaccard similarity across row groups**: Could capture domain reuse across chunks, but would require chunk-level scans and pairwise comparisons which are prohibitively expensive for large datasets.

Ultimately, we focused on metrics that are (a) computable from global statistics, (b) interpretable in the context of compression, and (c) diverse enough to capture value repetition, distribution shape, and dispersion. While more advanced metrics may help fine-tune predictive models in future work, the ones used here already provide strong explanatory power.

4.4.4 Gathering Column-Level Statistics

To perform compression analysis at the column level, we require detailed size information for each column's Parquet and dictionary encoding. Rather than parsing metadata directly from existing files, we adopt a simpler and more modular strategy: for each column in the dataset, we write it to a separate Parquet file using our standard encoding logic, then measure the resulting file and dictionary sizes individually.

This approach significantly reduces implementation complexity, as it leverages the same encoding pipeline used for full-table compression, avoiding the need for intricate metadata parsing or custom tooling. It also ensures consistency between the encoding method used for evaluation and that used in actual compression.

The downside of this method is that file-level overhead, such as headers, footers, and file metadata, is repeated for each individual column file. This can slightly inflate the measured file sizes, especially for small columns where overhead forms a proportionally larger share. However, we mitigate this in two ways: we exclude very small columns (file size below 2KB) from the column-level analysis, as they are both less impactful in total file size and more susceptible to metadata distortion. In addition, Parquet file metadata is relatively compact compared to the actual encoded data, so the distortion remains minor for medium to large columns.

A more precise but complex alternative is to use `parquet-cli`, an open-source command-line tool for inspecting Parquet file metadata. It exposes per-column statistics, including uncompressed and compressed byte sizes, by parsing the file footer directly without reading the full data. In principle, this would allow us to analyze column sizes without rewriting or re-encoding data.

However, integrating *parquet-cli* or replicating its functionality in custom code is non-trivial:

- Its output is not structured for programmatic consumption, as it pretty-prints its values to the terminal; parsing and interpreting its output requires additional tooling or wrappers.

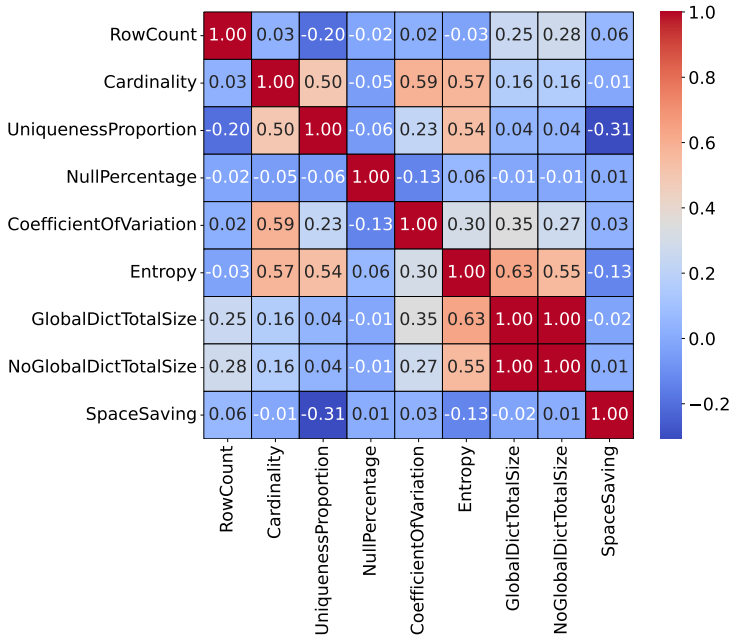

Figure 4.3: Correlation heatmap of column-level metrics across all four datasets. Space Saving is most negatively correlated with Uniqueness Proportion.

- It has no integration with global dictionaries, and does not know how to read a `RLE_GLOBAL_DICTIONARY` encoded data page if need be. However, this is something a full production-ready implementation should be able to handle, meaning we will place into the list of future works.
- The effort to reliably extract per-column metrics at scale, especially for dictionary pages, is high compared to our current pipeline.

For these reasons, we prioritize simplicity and robustness over exact byte precision, accepting minor noise in the data in exchange for much lower implementation overhead and better compatibility with our global dictionary evaluation setup.

4.4.5 Correlation Analysis of Column Features

To better understand the relationships between column-level statistics and compression effectiveness, we computed Pearson correlation coefficients between key features across all datasets. The results are shown in the heatmap in Figure 4.3. Note that, especially in TPC-DS, there are a large number of tables/columns with very few rows (e.g. categorical data such as department). We consider the space saving data on these columns too be too small to be relevant, and thus disregard anything below 2KB.

The most notable correlation is between **Uniqueness Proportion** and **Space Saving**, with a coefficient of -0.31. This suggests that columns with fewer unique values relative to their row count tend to benefit more from global dictionary encoding. This follows from

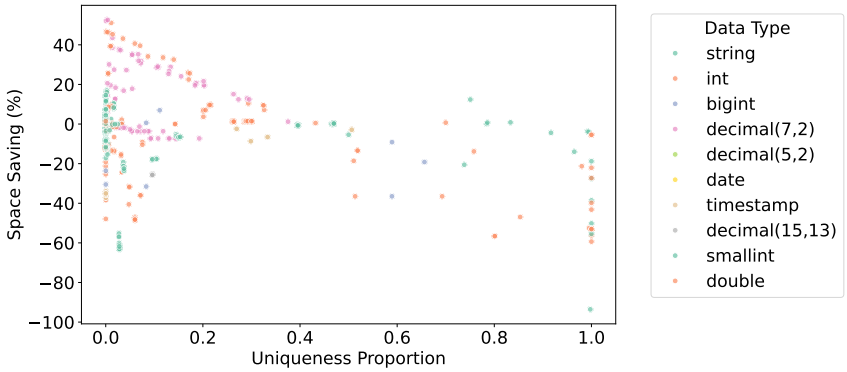

Figure 4.4: Space saving vs. uniqueness proportion across all four datasets, colored by data type. Lower uniqueness generally correlates with better compression.

the nature of dictionaries, which prefer encoding finite domain columns, such that their dictionary entries can be reused. Other moderate correlations include:

- **Entropy**: Correlated at -0.13 with Space Saving.
- **Cardinality**: Weak correlation, likely due to interaction with other variables.
- **Coefficient of Variation (CoV)**: Near-zero correlation, indicating that relative dispersion may not be a strong standalone predictor.

It is also worth noting that GlobalDictTotalSize and NoGlobalDictTotalSize are tightly correlated with entropy and CoV, which is expected. Larger, more variable columns tend to occupy more space regardless of encoding.

4.4.6 Uniqueness Proportion Across Datasets

While Uniqueness Proportion showed the strongest overall correlation with compression effectiveness, its predictive power varies significantly by dataset. In the TPC-DS 10GB dataset, for instance, the correlation was substantially stronger at -0.53.

Figure 4.4 shows how space saving (expressed as percent change) varies with uniqueness proportion, broken down by data type across all datasets.

From this scatter plot, several patterns emerge:

- Columns with uniqueness below 0.4 are the most likely to yield large space savings.
- Decimal types (especially low-precision decimals) often outperform strings even at comparable uniqueness levels.
- String columns show wide variability: some compress well while others regress, supporting the idea that domain size alone is not enough to predict success.

These findings support the idea that global dictionaries are particularly sensitive to columns with "localized" subdomains. For example, if even a single value in a column chunk is missing from the global dictionary, the entire data page will be reverted to plain encoding. Local dictionaries do not have this limitation as their dictionaries are built only from the values present in that particular column chunk, and therefore have the space to

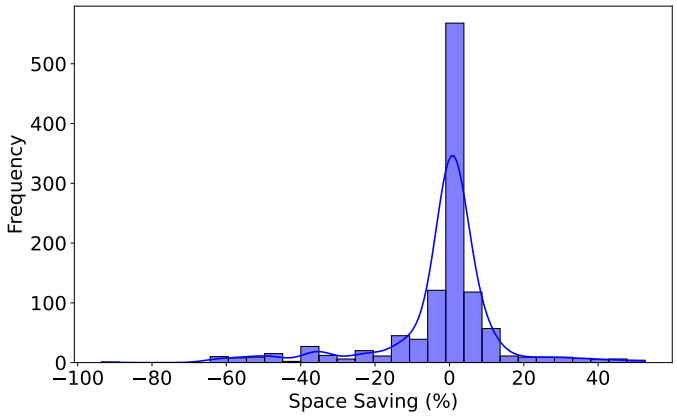

Figure 4.5: Distribution of space savings per column across all four datasets. Most columns see modest gains or losses, but a few yield extreme improvements or regressions.

include these localized values completely (and avoid reverting to plain encoding). This will be addressed in Section 4.5

Despite the clearer trends observed in TPC-DS, results on other datasets were more mixed. In CommonGovernment and Library Collection Inventory datasets, correlations between uniqueness and space saving were much weaker or non-existent. In these real-world datasets, the assumptions of uniform chunk reuse and global repetition often break down due to sparse, long-tailed, or dirty data distributions.

This raises a critical question: *Can we generalize compression heuristics across datasets?* Our findings suggest caution. Even when metrics like uniqueness proportion or entropy appear predictive in aggregate, they may not consistently translate to real-world benefit on arbitrary datasets.

Uniqueness proportion is a strong, though not universal, predictor of compression gain. While it can be used to guide global dictionary enablement, it should be considered alongside dataset-specific factors such as data cleanliness, chunk boundaries, and row group variability. In practice, adaptive and column-aware strategies are necessary for robust deployment (see Chapter 5).

4.4.7 Variance in Compression Outcomes

The impact of global dictionaries varies widely across columns, even within the same dataset or table. Figure 4.5 shows the overall distribution of per-column space savings across all datasets. While most columns cluster near zero, there is a long left tail of regressions and a shorter right tail of highly compressible fields.

Several columns achieved more than 50% space savings, typically due to highly repeated decimal values:

- `store_sales.ss_list_price` — `decimal(7,2)`
- `web_sales.ws_list_price` — `decimal(7,2)`
- `store_returns.sr_return_time_sk` — `int`

- `web_returns.wr_fee` — `decimal(7,2)`

These fields share a common structure: they use a constrained, low-precision domain (e.g., dollar or time fields) with frequent value repetition across files and row groups.

In contrast, other columns consistently performed poorly or regressed in size:

- `item.i_product_name` — `string`
- `commongovernment_5.vend_dunsnumber` — `string`
- `customer.c_customer_sk` — `int`
- `web_returns.wr_refunded_cdemo_sk` — `int`

Interestingly, the best- and worst-performing columns can coexist within the same table. In `web_returns`, for example, `wr_fee` saw over 50% compression gain, while `wr_refunded_cdemo_sk` increased in size due to fallback and inefficient encoding.

This wide variance underscores the necessity of column-aware strategies that, for example, understand that compression gain is highly feature-specific. This means not all columns in a well-compressing table are good candidates. Additionally, numeric columns with coarse domains and repeated values (like list prices or return fees) are ideal targets for global dictionaries. This is particularly true when the datatype has significant waste (e.g. decimal types for a limited amount of unique prices).

Even within a single dataset or table, global dictionaries must be applied selectively. The presence of both extreme gains and losses supports a hybrid approach or a predictive selection mechanism informed by simple metrics like uniqueness proportion, entropy, and domain size.

4.4.8 Data Type–Level Compression Patterns

Compression outcomes vary not just by column structure, but also by data type. Figure 4.6 summarizes the distribution of space savings across all tested data types.

The best-performing data type is `decimal(7,2)`, where global dictionaries consistently yield large space savings. These fields typically represent monetary values and exhibit limited precision and high repetition (e.g. 0.99, 1.50, 9.99, etc.), which are ideal conditions for dictionary compression. Even `decimal(5,2)` and `smallint` fields perform well, albeit with lower variance due to smaller domains.

Despite conventional wisdom suggesting that `string` and `binary` fields benefit most from dictionary encoding, we observe limited gains. Most values cluster near 0% compression improvement, with many even regressing. This could potentially be explained by Parquet's local dictionary encoding already handling repeated string values efficiently. Global dictionaries often fail to add further benefit unless there is substantial cross-file repetition. Moreover, `string` and `binary` types tend to be larger than any other datatype. Consequently, fewer values are able to be placed into a global dictionary of a constant, 2MB, size. We know that *the smaller the cardinality of the global dictionary the worse its ability is to be generalized across an entire column.*

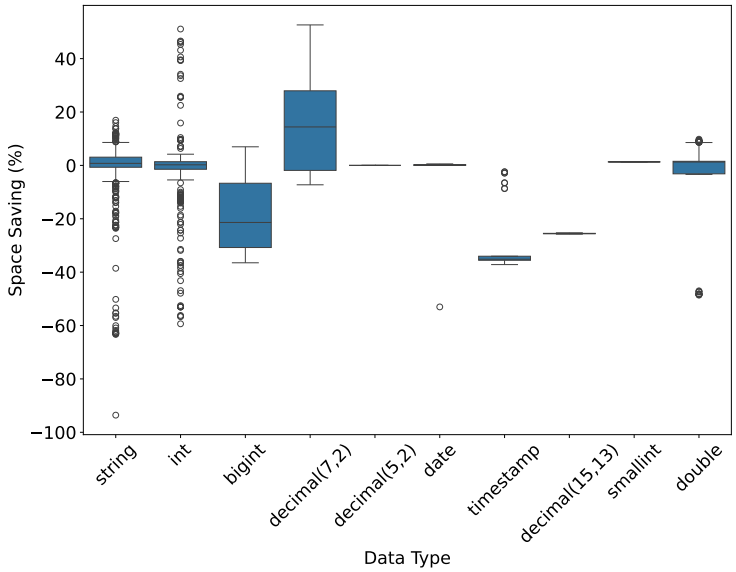

Figure 4.6: Space saving breakdown by data type across all four datasets. Decimal types tend to benefit the most from global dictionaries, while timestamp and string types often show limited gain.

This suggests that global dictionaries are not *necessarily* additive for strings as they may duplicate what local encoding already does well.

Timestamp fields show consistent compression regression. This is most likely caused by their larger encoded widths and has little to do with the data type itself. Due to the nature of global dictionaries, encoded values (in the data page) typically require greater bitwidths than their local dictionary encoded counterparts (e.g. 16-bit vs. 8-bit), particularly when the cardinality is just right (or just wrong). These timestamp columns tended to have a cardinality of ∼2000 which is in the "goldilocks zone" for such a phenomenon to occur. This is expanded upon further in our discussion of indirection dictionaries in Section 4.6.4.

To conclude, while decimal and low-cardinality numeric fields benefit most from global dictionaries, string and timestamp types often regress or stagnate. Local dictionary encoding, encoding optimizations, and fallback sensitivity all play a role. These insights draw us away from the conclusion that any particular data type carries greater compression potential over local dictionaries.

Our results suggest that compression effectiveness could, in principle, be predicted in advance by a small set of column-level statistics:

- Cardinality relative to dictionary size
- Uniqueness proportion
- Entropy (when cheap to compute)

However, computing these features at scale is non-trivial. For example, determining the uniqueness proportion of a column typically requires a full scan and a distributed

`distinct` operation in Spark. Internally, these operations translate to global aggregations, often involving:

- A full table scan to materialize the column
- Serialization and shuffling of keys across the cluster
- Hash-based aggregation and cardinality estimation (or full materialization, if exact)

This results in high memory pressure and network I/O, especially for wide or unbounded domains (e.g., strings, timestamps). In many real-world pipelines, such computations are prohibitively expensive for all columns in large datasets.

Entropy and coefficient of variation also require second-order statistics (value frequencies, variances), which are expensive to collect unless approximate sketches or histograms are precomputed.

Therefore, these statistics may not justify their computation cost in real-time. We propose approximate or sketch-based alternatives be left for future work (see Chapter 6).

These findings motivate hybrid encoding schemes, which we explore next.

4.5 Hybrid Dictionary Design

While global dictionaries offer strong compression potential for certain columns, their utility is limited by an all-or-nothing fallback strategy: if even one value in a row group is missing from the dictionary, the entire column chunk reverts to a less efficient encoding. Hybrid dictionaries aim to overcome this by combining the reuse benefits of global dictionaries with the adaptability of local ones.

4.5.1 Hybrid Mode Implementation

The hybrid dictionary encoding mode modifies the Parquet write path to support mixed dictionary references. Values are encoded using global dictionary indices where possible. For values not present in the global dictionary, a local dictionary is constructed dynamically and assigned a separate index range.

4.5.2 Index Range Partitioning

We divide the index space into two contiguous intervals:

- $[0, G)$: Reserved for global dictionary values, where G is the size of the global dictionary.
- $[G, G + L)$: Assigned to local dictionary entries constructed during encoding, where L is the number of locally seen values not found in the global dictionary.

This partitioning allows both dictionaries to coexist without ambiguity, as there is no overlapping index space between the two. A sample mapping is illustrated in Figure 4.7.

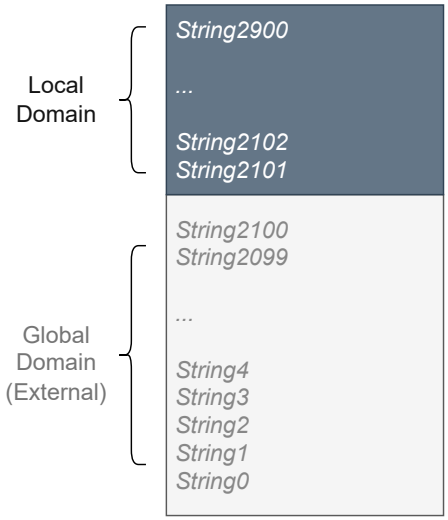

Figure 4.7: Illustration of hybrid dictionary encoding for a string column. Values are first mapped to the global dictionary (gray). Missing values are assigned to a local dictionary stored in the Parquet file (dark gray) with non-overlapping indices.

4.5.3 Control Flow and Layout

During writing, we maintain a lookup map for the global dictionary. For each value:

- If it exists in the global dictionary, we emit the global index directly.
- Otherwise, we insert it into a local dictionary and emit a new index offset by G.

Note that, if all values are present in the global dictionary, a local dictionary will never be written.

Both dictionaries are serialized into separate pages and wrapped into a unified hybrid dictionary page structure. This structure contains *a)* the global dictionary portion (preloaded from external file) and *b)* the (optional) local dictionary portion (generated on-the-fly).

This layout minimizes ambiguity at read time and supports efficient lookup with minimal branching.

We define a new encoding constant:

`RLE_HYBRID_DICTIONARY`

This value is written to the Parquet data page header and allows readers to invoke the appropriate decoding logic. In the event that no auxiliary local dictionary was used, the data page encoding remains `RLE_GLOBAL_DICTIONARY`. Hybrid decoding is discussed further in Section 4.5.4.

The algorithm for encoding values using a hybrid dictionary is summarized in Algorithm 3.

Algorithm 3 Write Value Using Hybrid Dictionary

Require: Value v, Global dictionary G, Local dictionary L, Bitwidth estimator $\mathcal{E}$
Ensure: Encoded index appended to $EncodedValues$

```
id ← lookup(v) in G
if id = −1 then
    if L = null then
        L ← new empty dictionary
        usingLocalDictionary ← true
        missingValuesInGlobalDictionary ← true
    end if
    id ← lookup(v) in L
    if id = −1 then
        id ← G.size() + L.size()
        insert v into L with index id
        dictionaryByteSize ← dictionaryByteSize + sizeof(v)
    end if
end if
maxUsedDictionaryEntry ← max(maxUsedDictionaryEntry, id)
append id to EncodedValues
update E with id
```

The combined dictionary is never explicitly materialized during encoding. Only the local dictionary is emitted in the Parquet file's dictionary page. The global dictionary is stored externally and reused across chunks.

4.5.4 Read Path

When reading a hybrid-encoded column chunk, the decoder must reconstruct the full dictionary by merging the global and local components in a consistent index space. The process is as follows:

1. Detect whether the column chunk uses hybrid dictionary encoding by inspecting the encoding type (`RLE_HYBRID_DICTIONARY`) in the page header.
2. Load the global dictionary from the configured external dictionary path. This is parsed into a list of values indexed from $[0, N)$.
3. Decode the column chunk's local dictionary page and append its entries to the global dictionary, assigning indices from $[N, N + M)$. Note, that if a hybrid encoded data page is present, a local dictionary page *must* be present.
4. Read the one-byte bitwidth header and decode the rest of the page using RLE/bit-packing.
5. Use the combined dictionary to resolve values by index.

This read strategy ensures that the same index-space is maintained as during encoding. Since all dictionary entries (global + local) are immutable and ordered, decoding remains

efficient and deterministic.

Suppose the global dictionary contains `["A", "B", "C", "D", "E"]`, and the local dictionary contains `["X", "Y"]`. During writing, values are encoded as:

$$\texttt{["A", "B", "X", "Y", "C"]} \rightarrow [0, 1, 5, 6, 2]$$

Note that $N = 5$, meaning that local dictionary indices must start at 5 even though "D" and "E" were never encoded.

At read time, the decoder rebuilds:

$$\texttt{[0:"A", 1:"B", 2:"C", 3:"D", 4:"E", 5:"X", 6:"Y"]}$$

ensuring correct decoding of hybrid-indexed values.

This mechanism allows hybrid chunks to remain compatible with readers aware of external dictionaries and simplifies the fallback story. If no local dictionary exists for a given chunk, the global dictionary alone suffices.

4.5.5 Compression Results

Experimental setup. We evaluated each dataset for the thresholds: 1, 2, 4, 8, and 16. Each row corresponds to a different minimum count required for a value to be included in the global dictionary.

Table 4.3 summarizes compression outcomes with hybrid dictionaries enabled, across multiple datasets and varying minimum thresholds (i.e. minimum number of key occurrences) for global dictionary inclusion. We observe that the best overall compression ratios are achieved at intermediate thresholds, specifically between ≥ 4 and ≥ 8, where both the Parquet file size and the final total size (Parquet + dictionary) are well-optimized.

As the threshold increases to ≥ 16, compression performance deteriorates. This is primarily because fewer values are included in the dictionary, resulting in more fallback to plain encoding, which inflates the Parquet file size. Conversely, at the lowest threshold (≥ 1), almost all values are included, minimizing fallback and achieving the smallest Parquet size. However, the accompanying dictionary becomes substantially larger, which increases the total storage footprint and slightly worsens the final compression ratio compared to the optimal range.

Table 4.4 zooms in on this phenomenon by showing the impact of varying frequency thresholds on compression performance for the CommonGovernment dataset.

This highlights a key tradeoff in dictionary design: lower thresholds reduce encoded data size but inflate metadata overhead, while higher thresholds reduce dictionary cost but trigger inefficient fallback. The optimal point lies in balancing the two.

An interesting phenomenon emerges at the lowest threshold (≥ 1): in some cases, the column's unique values happen to occupy a combined dictionary size between 1–3MB. This size range exceeds what a local dictionary (typically capped at ∼1MB) could hold but remains within the bounds of the hybrid dictionary capacity (up to ∼3MB). When

Table 4.3: Compression Ratios Across Thresholds for All Datasets. Threshold is the minimum frequency for a value to be included in the global dictionary. Higher ratios indicate better compression. The highest ratio per dataset is **bolded**.

Min Count	NYC	Library Collection	TPC-DS 10GB	Common Gov.
≥ 16	1.012	1.085	1.211	1.034
≥ 8	**1.014**	**1.086**	1.245	1.037
≥ 4	1.011	1.086	**1.260**	1.044
≥ 2	1.008	1.084	1.257	**1.045**
≥ 1	1.005	1.084	1.250	1.044

Table 4.4: CommonGovernment: Compression Metrics at Different Frequency Thresholds (GB)

Min Count	Baseline Size (GB)	Parquet Size (GB)	Dict Size (GB)	Total Size (GB)	Ratio
≥ 16	7.670	7.355	0.059	7.414	1.034
≥ 8	7.670	7.299	0.096	7.395	1.037
≥ 4	7.670	7.186	0.160	7.347	1.044
≥ 2	7.670	7.148	0.198	7.346	1.045
≥ 1	7.670	7.122	0.225	7.347	1.044

this occurs, the column can be entirely dictionary-encoded without any fallback, leading to substantial compression gains. By looking at column-level statistics, this sometimes reduces space usage by 30–70%. This suggests that dynamically adjusting the global dictionary limit based on the observed size of unique values may be a worthwhile strategy. While such a heuristic might initially appear ad hoc, it is arguably justified in the context of preprocessed datasets, where we already invest effort in extracting global statistics to inform encoding strategies.

4.5.6 Byte Width Rounding Effects in Dictionary Encoding

Although hybrid and global dictionaries aim to reduce encoded data size by reusing shared entries, an unexpected source of inefficiency arises from how Parquet encodes dictionary indices. Internally, indices are stored using variable-width integer encodings. However, these widths are not byte-exact but bucketed into fixed *tiers*:

- **Tier 1:** ≤ 8 bits $\rightarrow$ 1 byte (int8)
- **Tier 2:** $\leq$ 9–16 bits $\rightarrow$ 2 bytes (int16)
- **Tier 3:** $\leq$ 17–24 bits $\rightarrow$ 3 bytes (int24)
- **Tier 4:** $\leq$ 25–32 bits $\rightarrow$ 4 bytes (int32)

The assigned tier depends on the highest index written to the data page. For example, a dictionary of size 600 requires $\lceil \log_2(600) \rceil = 10$ bits. Since 10 bits exceeds 8, the index

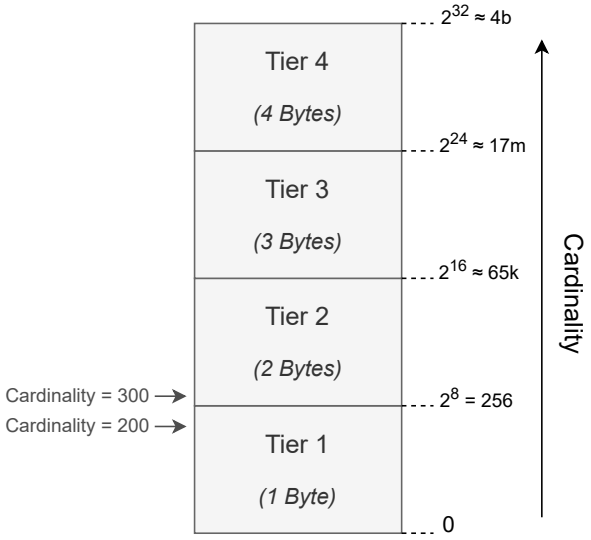

Figure 4.8: Bitwidth tier assignment based on dictionary size. Values that fall within Tier 1 (≤ 256 entries) benefit from int8 encoding. Global or hybrid dictionaries with slightly higher cardinality (e.g., 300) get bumped to int16, incurring overhead.

tier is bumped to Tier 2 (16 bits), wasting 6 bits per value.

Local dictionaries often operate over subdomains of the dataset, especially in sparse or skewed distributions, leading to smaller effective cardinalities in specific row groups. Suppose the full dataset has 600 distinct values (Tier 2), but a row group only observes 200 (Tier 1). In this case, the local dictionary index requires just 8 bits, saving 1 byte per value compared to a global/hybrid encoding.

This can result in paradoxical outcomes: even though global or hybrid dictionaries provide broader reuse, the local dictionary encodes values more compactly due to lower tier assignment. Figure 4.8 shows this rounding effect.

This effect is not due to suboptimal compression behavior per se, but a side-effect of internal encoding tiering. Ideally, hybrid and global dictionaries would only store the highest index *used*, allowing the encoder to assign a lower tier if usage permits. While we implemented such an optimization to track the maximum used index, it rarely had measurable impact, as the highest index tends to appear frequently in large datasets.

This issue points to an inherent tension between reuse (via large dictionaries) and local efficiency (via smaller index tiers). In practice, this tradeoff must be accounted for when tuning global dictionary thresholds.

4.5.7 Column-Level Analysis

To further investigate the effect of byte-width rounding described in Section 4.5.6, we analyzed column-level compression statistics for several datasets. For each column, we recorded:

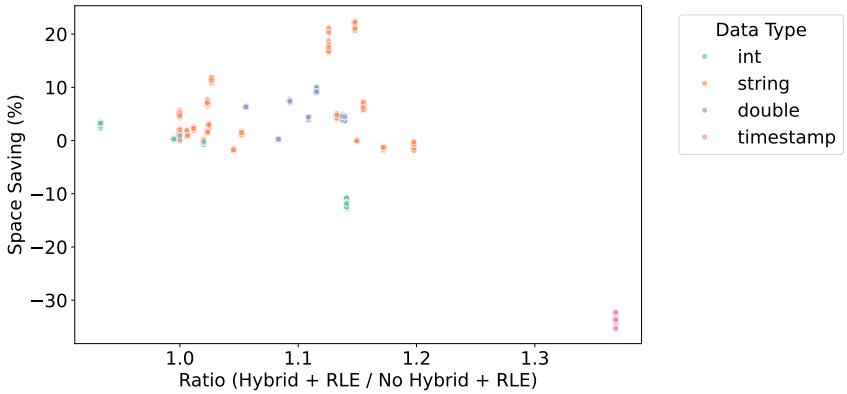

Figure 4.9: Ratio of average RLE encoded value size (Hybrid vs. No Hybrid) against the total compression improvement using the CommonGovernment dataset. Columns are grouped by data type. Values above 1.0 on the x-axis suggest that hybrid encoding produced larger value representations, even after RLE and bit-packing.

- Total number of unique values (i.e., dictionary size).
- Maximum index used in encoded data pages.
- Assigned byte-width (1–4).
- Total size of encoded data pages with hybrid/global dictionaries.
- Total size of same data pages under local dictionary encoding.

We found multiple cases where the global or hybrid dictionary exceeded the Tier 1 or 2 threshold, but local dictionaries for the same column remained within it. The consequence was consistent: local dictionary pages used fewer bytes per value, as the global/hybrid encoding habitually pushes the index space over key thresholds such as $2^8 = 256$ or $2^{16} = 65{,}536$. Even a single value crossing such a boundary can inflate every encoded index in the page. In the case of the boundary being crossed being 256 entries, the byte width will double (thus, the page size could ostensibly double). In the case of 65,536, it would increase by a factor of 1.5.

4.5.8 Byte-Width Analysis

To validate that this effect is not being mitigated by downstream encoding optimizations (i.e., RLE/bit-packing), we computed the **average encoded value size** after both compression steps for each column, once with hybrid dictionaries enabled, and once without. As statistics of such specific nature are not naturally exposed by the Parquet metadata, it was necessary to rig Parquet-mr to provide us these values. We then computed the ratio:

```
Avg Encoded Size (Hybrid) / Avg Encoded Size (No Hybrid)
```

The resulting values reflect the final size of each value after RLE and bit-packing, and are shown in Figure 4.9. This allows us to isolate whether the inefficiencies in dictionary byte-width tiers persist through encoding.

We make several observations from the graph:

- Many columns cluster near $x = 1.0$, indicating equivalent per-value sizes under both modes.
- A significant number of columns exceed $x = 1.1$, implying that hybrid dictionaries inflated the encoded value size by 10% or more.
- All columns appearing at the far right (≥ 1.3) are timestamp columns (manually verified), showing inflated RLE sizes with hybrid encoding and correlating with their previously observed poor compression.

It is worth noting that there do not appear to be ratios greater than 2.0, as would be expected with the byte width doubling from 1 to 2 in the 2^8 boundary case. This is caused by fact that *a)* these are average values, meaning that local dictionaries do not always have optimally sized dictionaries either and *b)* using local dictionaries, within a column chunk, the data pages that benefit from smaller byte widths are often only the first ones, as the dictionary is not yet fully constructed and has a lower cardinality. This last point is less a facet of local dictionaries benefiting from only working with lower cardinalities within subdomains of the dataset, and more a facet of the dictionary building algorithm (which global/hybrid dictionaries cannot take advantage of either). Although, local dictionaries deal with the final cardinality of the column chunk it is encoding, but the algorithm can encode based on the *currently observed cardinality* which is slowly incremented during the writing of the column chunk (until it reaches the final cardinality at some point).

Therefore, we confirm that the tier-based byte width expansion of dictionary indices is not being entirely optimized away by RLE or bit-packing. Instead, the larger dictionary domain introduced by global or hybrid dictionaries can lead to measurable post-encoding bloat. This problem may be subtle but has meaningful impact, particularly when a column's global cardinality is just above a width threshold like 256 or 65,536, while its local subsets could have stayed below. Even though this does not affect the semantic correctness of encoding, it suggests that better compression could be achieved by byte-width-tier-aware dictionary allocation or smarter bit-width tracking during encoding.

4.6 Indirection Dictionaries Design

4.6.1 Usage Efficiency of Global Dictionaries

In the previous section, we examined how large dictionary domains can inflate the bitwidth of encoded indices, leading to avoidable storage overhead. The problem is exacerbated when only a fraction of the dictionary is actually used in a given column chunk, yet the full domain size determines the index width.

To better understand this, we computed the *average global dictionary usage proportion* per column. That is, for each column chunk, we measured what fraction of the global dictionary entries were actually referenced, and then averaged this across all chunks.

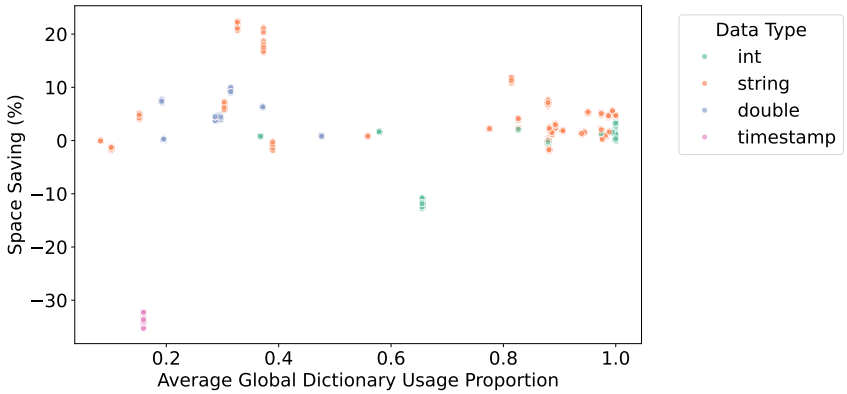

Figure 4.10: Average global dictionary usage proportion vs compression ratio using the CommonGovernment dataset. Usage proportion is averaged across all column chunks. Higher values suggest a more efficient utilization of the dictionary space.

Figure 4.10 plots this proportion against the overall compression ratio. The aim is to determine whether global dictionary size is being fully utilized, and whether high dictionary occupancy correlates with better compression outcomes.

Several key observations emerge:

- A large cluster of columns achieves near 100% usage, often correlating with modest but consistent compression benefits. This suggests that for these columns, the dictionary is well-aligned with actual values and the cost of a larger index width is amortized by high reuse.
- A sizable portion of columns has usage proportions between 0.3 and 0.6. These exhibit a wide spread in compression results, including negative outcomes. This reflects the overhead of maintaining a large dictionary index width (e.g., 2 bytes) even when much of the dictionary is unused.
- Timestamp types once again underperform, clustering at low usage ratios with consistently poor compression. This reinforces their previously noted issues with wide value domains and limited redundancy.

4.6.2 Impact of Inflated Byte-Widths

While Figure 4.9 shows a generally weak correlation between the bytewidth ratio and the overall compression ratio, it nonetheless reveals that a non-negligible number of columns suffer meaningful regressions. Most columns cluster around neutral impact, but outliers, particularly those with large index inflation, demonstrate substantial losses in compression efficiency. Even if this phenomenon does not dominate overall compression outcomes, addressing it remains worthwhile.

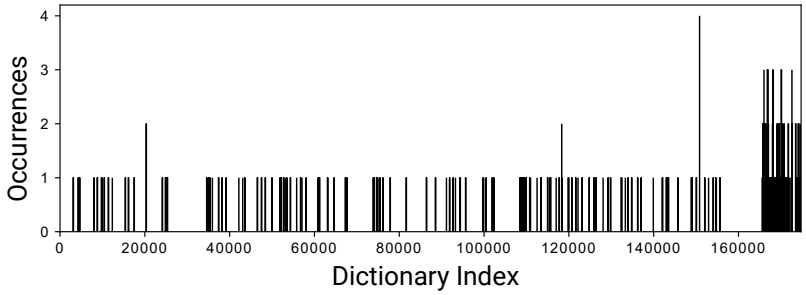

Figure 4.11: Example of a patternless or fragmented global dictionary index distribution from a particular row group of the Closed_Date column in the NYC dataset. Usage appears irregular and not strongly correlated with global frequency.

4.6.3 Visual Analysis of Global Dictionary Usage

To better understand how global dictionaries are utilized across column chunks, we performed a visual inspection of the distributions of global dictionary indices used per chunk. These plots aimed to expose patterns in index usage that could help explain compression success or failure. We analyzed these distributions for columns that compressed well and those that did not.

Three distinct shapes consistently emerged across column chunks:

1. **Heavily right-skewed distributions:** These are the most common and expected pattern. A small set of low-index dictionary entries are reused heavily, while higher indices are rarely referenced. These align with columns where a few values dominate.
2. **Patternless or fragmented usage:** These graphs appear scattered and inconsistent. While certain indices are reused, there is no clear correlation to overall frequency in the dataset. These cases reflect local subdomain variation or data drift, which global dictionaries fail to capture. An example of such a distribution is shown in Figure 4.11.
3. **Right-skewed with gaps:** Similar to (1), but with noticeable holes in the index space. That is, some dictionary indices within the expected active range are never used. This suggests that global dictionaries overfit to a superset of values not uniformly distributed across all row groups. An illustration of this type is shown in Figure 4.12.

Critically, we found no visible correlation between the shape of these index usage distributions and whether a column compressed well. Columns with the same distribution shape (especially right-skewed) could yield both high compression and compression regressions, depending on other factors such as fallback behavior, byte-width inflation, or value sparsity.

This observation raises the important point: **index distribution shape alone is not a predictor of compression effectiveness**. Even well-shaped usage profiles may lead to poor results if fallback is triggered or if dictionary indices cross byte-width thresholds unnecessarily.

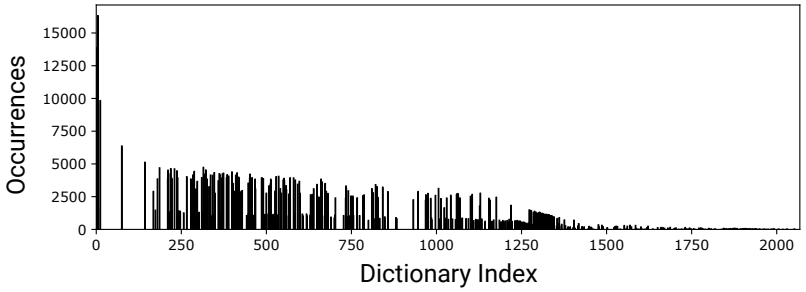

Figure 4.12: Example of a right-skewed global index usage distribution with visible gaps from a particular row group of the contract_signeddate column in the CommonGovernment dataset. Several midrange indices are entirely unused despite proximity to frequently referenced entries.

These findings support several conclusions:

- Global dictionary efficiency is not just about which values are included, but how often those values are actually used across chunks. A well-populated dictionary is wasteful if large portions of it are never referenced.
- Compression regressions can result from minor differences in index usage. For example, two chunks may differ only in using index 256 instead of staying under 256, thus incurring a jump from 1-byte to 2-byte indices across the entire chunk.
- The need for smarter indexing mechanisms becomes apparent. The indirection dictionary design (Section 4.6.4) is a direct response to these observations. It allows selectively rewriting only high-index lookups while maintaining tight bit-width constraints for the rest, avoiding unnecessary bloat.

4.6.4 Indirection Dictionary Design

While global dictionaries improve compression by avoiding redundancy, they can suffer from inefficiencies due to sparsity in index usage. In particular, encountering even a single global dictionary ID with a high value (e.g., 1027) can inflate the required bit-width for all values in a page, even if most fall under a tight range like 0–127. The indirection dictionary is designed to address this by compressing and remapping global dictionary IDs into a more compact, chunk-specific space.

The core idea is to introduce a local remapping layer between the global dictionary and the encoded stream. This layer, the *indirection dictionary*, assigns new contiguous indices to global IDs in a page, optimizing bit-width usage. The design supports three operational modes:

1. **Global Mode**: As long as all global dictionary IDs used in a page fall below a chosen threshold (e.g., 256 or 65536), values are directly encoded using those IDs.
2. **Pointer Mode**: If a value above the threshold is encountered, it is mapped into a free slot below the threshold, and a mapping is recorded to its true global ID. This allows continuing to encode all values using low-width IDs.

3. **Fallback Mode**: If a value not found in the global dictionary is encountered, the page transitions into a hybrid mode where the lower part of the ID space is reserved for (remapped) global IDs and the upper part is used for locally encoded raw values.
4. **Plain Fallback Mode**: If a value not found in the global dictionary is encountered, or an out-of-order global dictionary value is encountered (and a pointer needs to be created for it), and there is no space left in the dictionary, fall back to plain encoding for the column chunk.

Finally, data pages encoded with indirection dictionaries receive the encoding constant:

`RLE_INDIRECTION_DICTIONARY`

This similar to the global and hybrid dictionary designs, except for the fact that, whereas hybrid and global could leave the dictionary page as `PLAIN` encoded, we now need to specify in the dictionary page header that it is `RLE_INDIRECTION_DICTIONARY` encoded. This is because the dictionary has a specialized structure that requires a separate decoding algorithm.

Take, as example, Figure 4.13 and consider a page that uses values with global IDs {12, 77, 210, 300}. Without indirection, the presence of 300—being greater than the encoding threshold of 256—forces all values to be encoded using 9 bits. With indirection, we remap 300 to an unused ID below the threshold (say, 0), allowing all values to be encoded using 8 bits. The indirection dictionary stores a pointer indicating that 0 maps to 300. Later, we encounter a new value, *"delta"*, which is not present in the global dictionary. This value cannot be encoded through indirection and instead triggers a fallback to plain encoding using a new ID, such as 211. From now on, all codes ≥ 211 are exclusively designated for plain encoding. This hybrid approach preserves encoding compactness for known values while still supporting dynamic additions.

The encoding logic chooses between thresholds (e.g., $2^8 = 256$, $2^{16} = 65536$) based on the maximum ID in the global dictionary. This defines the maximum range for compact encoding. Only values under this threshold can be directly represented; others must use pointer remapping or trigger fallback.

Each page with indirection encoding stores:

- An integer offset: the index where fallback to local dictionary begins, or -1 if it never happens.
- A bitmap of size T: indicates which remapped IDs are identity mappings (i.e., `localID == globalID`) and which are pointers.
- Pointer entries (i.e. the global ID they map to).
- Locally stored values (if fallback happened).

To efficiently encode the bitmap, we use a **Roaring Bitmap** format (37). A naive bitmap of size 2^{16} would consume 8 KB per page, even if very few bits are set. Roaring Bitmaps mitigate this by partitioning the bitmap into 16-bit chunks and applying a compact encoding per chunk based on its density. Sparse chunks are stored as sorted arrays

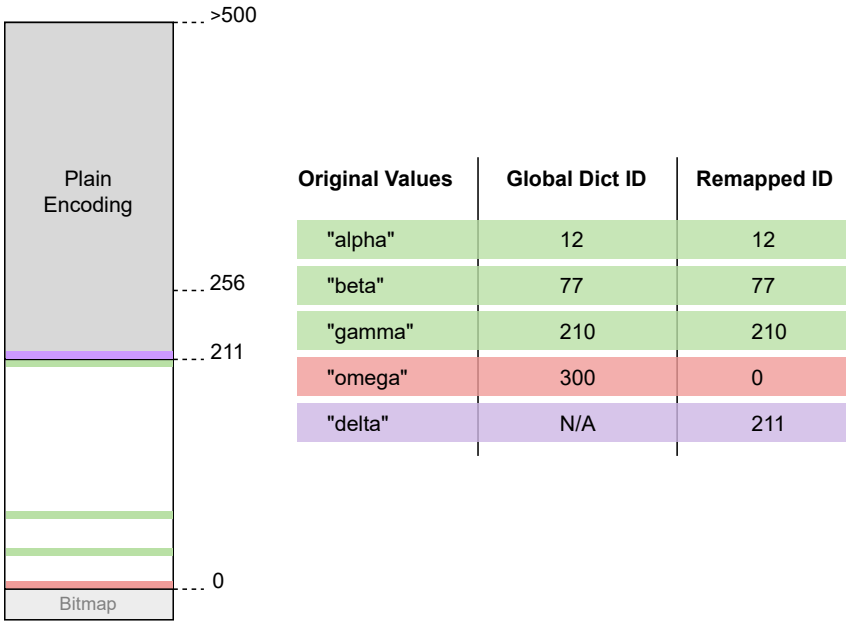

Figure 4.13: A three-stage indirection dictionary design. The indirection dictionary maps chunk-local indices to global IDs or local raw values, depending on fallback status. A bitmap indicates pointer vs identity mappings.

of set bit positions, while dense ones use run-length encoding or bitmaps. This offers both high compression and fast membership queries during decoding, making it well-suited for large-scale dictionary remapping.

To decode an entry:

1. Read the indirection ID from the encoded stream.
2. If $id <$ fallbackStart, check the bitmap:
   - If it's a pointer, resolve to global ID and decode.
   - Else, directly decode using global dictionary at that index.
3. If $id \geq$ fallbackStart, decode raw local value.

4.6.5 Decoder-Aware Dictionary Compaction Optimization

One subtle yet effective optimization targets dictionary space wastage introduced during pointer remapping. Consider a case where values with global dictionary IDs $\{0, 1, \ldots, 100\}$ are used directly, but then a value with ID 300 appears. Since it's above the threshold (e.g., 256), the system falls back to pointer mode and tries to map 300 to the next available unused index under the threshold, say, 101.

Internally, the dictionary is represented as an array. Placing a pointer at index 101 implies that the decoder must find its value at offset 101 in the serialized dictionary array. However, offsets 0–100 are now "garbage": although they were never written (as no local dictionary was yet needed), the decoder expects data there.

Algorithm 4 Indirection Dictionary Encoding with Fallback

```
1: T ← threshold based on global dict size
2: used[T] ← bitmap of used indices below threshold
3: mapping ← empty map from globalID to remappedID
4: fallbackStart ← −1
5: for each value in column chunk do
6:     globalID ← lookup(value, globalDict)
7:     if globalID == −1 then                          ▷ Global dictionary miss
8:         if fallbackStart == −1 then
9:             fallbackStart ← maxUsedIndirectionID + 1
10:        end if
11:        id ← allocate next local ID
12:        encode as raw local value at id
13:    else if globalID < T then
14:        mapping[globalID] ← globalID
15:        mark used[globalID]
16:    else                                            ▷ Value above threshold
17:        id ← first unused index < T
18:        mapping[globalID] ← id
19:        mark used[id]
20:    end if
21:    add id to encoded stream
22: end for
23: write fallbackStart, mapping bitmap, and remap table to page header
```

A naïve solution is to pad indices 0–100 with zeroes so that the offset aligns. This relies on compression algorithms like Snappy to compress away these redundant zero entries.

Instead of padding, we introduce a smarter decoder-side strategy that allows us to compact the dictionary payload and eliminate unused prefix slots:

- We write only the **compact set of pointer values**, sequentially from index 0 in the dictionary array.
- The decoder uses the **bitmap** to determine which indices are identity-mapped and which are pointers.
- As it iterates from index 0 to threshold:
  - If the bitmap entry is identity, it does not consume an entry from the dictionary array, it uses the global dictionary directly.
  - If the bitmap entry is a pointer, it consumes the next entry in the array.

This optimization: *a)* eliminates padding overhead in the dictionary array, *b)* reduces total page size (especially in pointer-heavy chunks), and *c)* leverages the decoder's sequential nature to reconstruct mappings with minimal state.

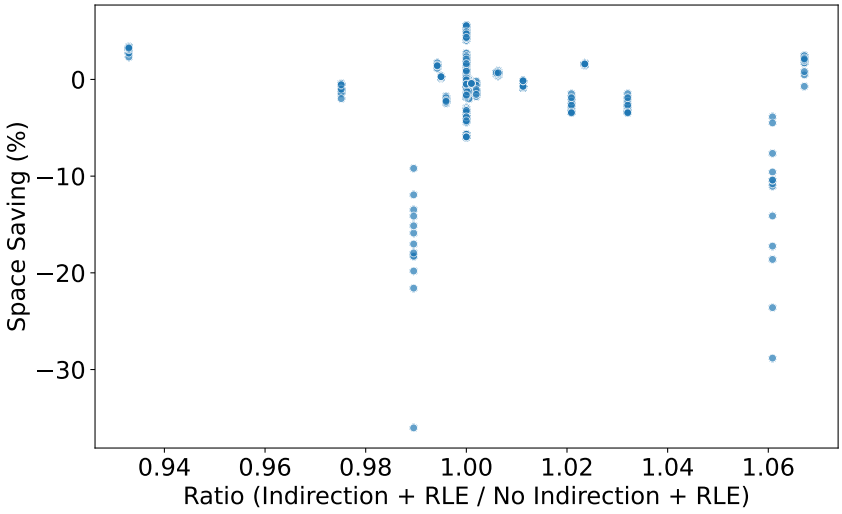

Figure 4.14: Ratio of average encoded value sizes (Indirection + RLE vs. No Indirection + RLE) versus compression ratio across data types using the CommonGovernment dataset.

4.6.6 Byte Width Optimization Verification

Figure 4.14 illustrates the comparative effectiveness of hybrid encoding versus non-hybrid RLE encoding by plotting the encoded value size ratio against the compression ratio, broken down by data type. Each point represents a distinct column chunk.

The key takeaway is that the ratios are tightly clustered around 1.0, indicating that **the bit-width padding inefficiency observed previously** (where values had to be stored using wider-than-necessary widths due to fragmentation or sparse distribution) **has been effectively mitigated**. In other words, the encoding is now operating at near-optimal bit widths, ensuring there is no unnecessary overhead in the value stream caused by sparsity in dictionary IDs.

This reinforces the success of the indirection encoding strategy, which remaps sparse or oversized dictionary IDs into a compact domain.

4.6.7 Indirection Compression Results

Experimental setup. We evaluated the impact of enabling indirect dictionary encoding on four datasets, comparing it against the baseline where global dictionary encoding was disabled. The comparison captures three key metrics: the size of the Parquet file with indirect dictionary enabled, the size contribution from dictionary pages alone, and the combined total. The ratio represents the total size with indirect dictionary enabled relative to the baseline size.

The results demonstrate that indirect dictionary encoding performs competitively, yet similarly to having the global dictionary disabled, yet in some cases, significantly better:

- **Library Collection Inventory** saw the greatest benefit, with a 3.6% size reduction. This aligns with our hypothesis that indirect dictionaries are highly effective on

Table 4.5: Final file sizes with indirect dictionary encoding enabled vs. global dictionary disabled. Global Dict Total Size is broken down into Parquet and Dictionary components. Higher ratios indicate better compression from indirect dictionaries.

Dataset	Baseline Size	Parquet Size	Dict Size (Global Dict)	Total Size	Ratio	Space Saving
Library Collection Inventory	5.68 GB	5.47 GB	0.009 GB	5.48 GB	1.036	0.035
NYC	1.04 GB	1.06 GB	0.019 GB	1.08 GB	0.967	-0.034
TPC-DS 10GB	4.50 GB	4.44 GB	0.086 GB	4.53 GB	1.001	0.001
Common Gov.	7.67 GB	7.69 GB	0.125 GB	7.82 GB	0.988	-0.012

datasets with fragmentation or sparse high-cardinality values, where fallback to local dictionaries can be minimized.

- **TPC-DS 10GB** showed almost no change, with a ratio of 1.001. This suggests that the dataset's structure may not benefit from indirection, perhaps because its values already lie well within the encoding threshold, or because fallback was frequent, negating the benefits.
- **NYC** and **Common Government** exhibited small regressions (+3.4% and +1.2%, respectively). In these datasets, the indirection mechanism likely caused overhead due to a larger number of values exceeding the limit (e.g., 256 or 65,536), triggering more fallback and expanding both dictionary and data sizes. Additionally, their cardinality distributions may not have allowed effective reuse of IDs via the pointer strategy.

4.6.8 Analysis of Indirection Dictionary Dynamics

The internal behavior of the indirection dictionary mechanism reveals several nuanced dynamics that influence its overall effectiveness, particularly compared to traditional dictionary or plain encoding strategies.

In principle, the indirection dictionary is designed to exploit situations where values that are not frequent enough to remain in the core domain of dictionary IDs (e.g., under 256 or 65,536) can still be encoded efficiently by referencing them via remapped pointers. This design intends to delay or avoid transitioning into local dictionary or plain encoding, particularly when the global dictionary contains heavy entries like long strings or binary blobs.

Under this assumption, indirection should outperform local dictionary encoding when:

- The raw values (e.g., strings or binary blobs) are significantly larger than the size of the pointers (4-byte integers).

- The remapped values are used across multiple column chunks, making the cost of repeated raw storage high.

However, our results do not show a standout performance improvement for string or binary types, which is counterintuitive given their size characteristics. This suggests two possible contributing factors:

1. The repeated values stored in the global dictionary may not occur frequently enough across chunks to amortize the pointer overhead.
2. The actual average size of strings and binary blobs in our dataset is small—sub-30 byte strings and 8–12 byte timestamps—so the savings from storing 4-byte pointers rather than full values is modest at best.

One structural limitation of the indirection dictionary is its fallback design. The pointer mapping section is terminated the moment an unknown (global-dictionary-absent) value is encountered. At this point, encoding switches to a raw-value local dictionary region. If such a value appears early in the column chunk, the indirection dictionary's effective domain is truncated before it can build up meaningful reuse, reducing its usefulness.

This creates a situation where the pointer section is sparsely utilized or sometimes barely initialized before being cut off. While this fallback logic is essential to preserve the compact index space, it reduces the potential benefits of the remapping mechanism. Some solutions from the literature have tried to avoid similar problems by reserving a larger than necessary domain (18). Unfortunately, it is still difficult to fully avoid this behavior without relaxing the tight index-space constraints (e.g., extending the pointer range beyond 256/65536), which reintroduces the same fragmentation problem the method aims to eliminate.

Empirically, the indirection dictionary behaves similarly to a local dictionary. When fallback occurs early or frequently, the entire structure collapses into local dictionary encoding, and the performance ends up nearly identical to that of the original fallback-based strategies. Thus, the expected hybrid efficiency does not always materialize, especially when input distributions are not favorable.

While theoretically promising, the indirection dictionary's real-world gains are marginal. The most meaningful space savings come not from the remapping layer itself but from its ability to *a)* extend the effective use of global dictionaries and delay plain encoding fallback and *b)* avoid dictionary duplication across column chunks.

However, in datasets with relatively small or infrequent dictionary values, these advantages are muted. Given its comparable performance to existing encodings and increased implementation complexity, the indirection dictionary approach may not warrant production deployment in its current form.

Future evaluations should consider datasets with:

- Longer and more repeated strings.
- Column chunks large enough to build substantial pointer sections before fallback.
- Deliberately fragmented or tiered value distributions.

Only under such conditions might the true promise of indirection-based remapping be fully realized.

4.7 Takeaways From Global Dictionaries on Static Datasets

This chapter has explored the design, implementation, and evaluation of global dictionary compression techniques for static datasets in Parquet-based Delta tables. Through extensive empirical analysis across multiple datasets, we have shown that global dictionaries offer strong compression benefits for columns with medium-cardinality domains and repeated categorical values, particularly when redundancy is high across row groups.

Among the strategies examined, the **hybrid dictionary approach emerges as the most effective**. It consistently delivers the best compression performance across datasets while mitigating the primary weakness of pure global dictionary encoding: the risk of all-or-nothing fallback when unseen values are encountered. Hybrid dictionaries preserve compression gains by combining reuse of global entries with the adaptability of local dictionaries. This avoids performance regressions even when value distributions vary across row groups. Importantly, hybrid encoding achieves these benefits without imposing significant metadata overhead or complicating the read path beyond practical limits. It also reduces the average encoded value size, thanks to more efficient byte-width usage.

However, this chapter has operated under the assumption that datasets are *static*, that is, fully available a priori and written in bulk. This simplifies dictionary construction but limits the applicability of the approach in many real-world scenarios where data is appended incrementally. The challenge of supporting **incremental updates**, where new values must be encoded without rewriting existing files or global dictionaries, is left unaddressed.

The next chapter tackles this challenge directly. We introduce mechanisms to extend or adapt dictionary encoding strategies to evolving datasets without sacrificing compression or correctness. In particular, we explore dictionary versioning, dynamic appends, and approximate sketches for dictionary maintenance under incremental ingestion workloads.

5

Supporting Incremental and Adaptive Use

The compression strategies discussed in the previous chapter assume a static dataset context, one in which the entire table is available in advance, and files are rewritten in bulk. While this is realistic for initial table construction or batch ETL pipelines, it does not reflect the operational model of modern data lakes built on Delta Lake, where data is appended continuously and snapshots evolve over time.

In such environments, dictionary-based compression must adapt to an evolving value domain without sacrificing correctness, compatibility, or performance. Unlike static writes, incremental appends introduce several challenges:

1. Evolving Value Domains New appends may introduce values not seen in the original dictionary. In a static dictionary setup, this leads to failure or fallback to plain encoding. For an incremental system, however, dictionary evolution must be managed carefully to avoid invalidating previously encoded data. A viable system must support either:

- Dynamic extension of dictionaries over time, or
- Versioning strategies to distinguish between multiple dictionary generations.

2. Snapshot Consistency Delta Lake guarantees snapshot isolation through its transaction log, meaning that every reader observes a consistent view of the table as of a specific commit (14). This has critical implications for dictionary usage: encoded values must be interpretable solely based on metadata available in that snapshot. This necessitates making dictionary metadata fully discoverable and snapshot-scoped, while ensuring that appended data encoded with new dictionary versions does not corrupt the semantic integrity of older reads.

3. Writer Scalability Incremental writes are often small and frequent. If dictionary construction requires a full table scan or large memory footprint, it becomes impractical in production environments. Therefore, dictionary construction must be computationally efficient, ideally bounded in space and parallelizable across partitions or streaming micro-batches.

4. Metadata Management This chapter addresses these challenges by extending the dictionary compression design to support **incremental and adaptive use**. Our goals are twofold:

1. Enable dictionary-based encoding to be applied to new data as it arrives, even when the full dataset is no longer available for global analysis.
2. Make the system robust to changing column characteristics, allowing selective and statistical decisions about when and where to apply dictionary compression.

We present a combination of metadata integration, lightweight sketching techniques, and runtime heuristics that enable scalable and flexible global dictionary usage in dynamic data lake scenarios.

The next sections explore the design implications of these requirements, beginning with how dictionary awareness is embedded in the Delta transaction log.

5.1 Delta Log Integration

Delta log extension is essential to support dictionary-aware incremental writes. To ensure consistent encoding and decoding across appends and table snapshots, writers must communicate which dictionary version was used in each data file. This requires tight integration with the Delta transaction log, a key mechanism for maintaining atomicity, isolation, and versioned metadata in Delta Lake.

5.1.1 Designing the Dictionary Metadata Layer

A central question in our design was: *at what level of granularity should the mapping between data files and dictionaries be maintained?*

In Parquet, dictionary encoding is applied at the level of individual column chunks within row groups, suggesting that it would be possible to record separate dictionary usage per column or even per row group. However, this granularity quickly becomes problematic in a production context, particularly in cloud-based data lakes with immutable object storage such as S3.

First, while the Parquet format does support per-column dictionaries internally, Delta Lake organizes its transactional state at the level of entire files. In practice, when appending or rewriting data, we operate on full Parquet files (not partial chunks) making finer-grained metadata difficult to enforce.

Second, tracking dictionary usage per column chunk or per row group would drastically increase metadata volume. For each file, we would need to store a mapping from every

column to a corresponding dictionary version, resulting in $O(N)$ metadata entries per file, where N is the number of columns. Moreover, Delta's current transaction log format does not provide native support for per-file, per-column metadata. Supporting this would require designing a new metadata structure and corresponding serialization logic, which would add nontrivial implementation and performance overhead.

5.1.2 Choosing AddFile-Level Metadata

To avoid this complexity, we chose to record dictionary usage at the file level. Specifically, we annotate each data file with a mapping from column names to the dictionary version IDs used for encoding. This mapping is embedded directly into the `AddFile` entries in the Delta transaction log.

This approach is grounded in the structure of Delta's transaction log, which consists of a sequence of versioned JSON or Parquet actions, including:

- `AddFile` – announces a newly written data file, including partitioning, stats, and optional metadata.
- `RemoveFile` – indicates that a file has been logically deleted in a later snapshot.
- `Metadata` – captures the schema and configuration of the table at a given version.
- `CommitInfo`, `Protocol`, etc. – additional metadata used for versioning and concurrency control.

Of these, only `AddFile` supports per-file custom metadata annotations, making it the natural place to attach dictionary version references. Our implementation introduces a new optional field:

```
"globalDictionaryPath":"dict_v1"
```

This dictionary informs the reader which global dictionary version was used to encode all columns in the file. Note, that we choose a "file-level" granularity because the append-only nature of the underlying object storage mandates that any edit to a Parquet must trigger an atomic rewrite of the entire file, anyway, meaning that they might as well all use the newest dictionary.

5.1.3 Multi-Column Dictionary Packaging

Another practical consideration was how to store the actual dictionary data. Initially, we stored each column's dictionary in a separate file. While simple, this leads to poor scalability in systems with large numbers of columns or frequent appends, as each dictionary necessarily required its own I/O transaction and metadata reference.

It is important to realize that a Parquet file now points to a single dictionary path, instead of pointing to a path per column per file (which would incur a significant storage overhead in the transaction log).

This is because we transitioned to bundling all dictionary pages for a given write operation into a *single* dictionary file. Each file contains a sequence of binary dictionary pages

Algorithm 5 Read Dictionary Page for a Column from Global Dictionary File

```
1: function ReadDictPageFromGlobalFile(columnPath, globalDictFilePath, conf)
2:     fs ← GetFileSystem(globalDictFilePath, conf)
3:     inputStream ← fs.open(globalDictFilePath)
4:     fileLength ← fs.getFileStatus().getLen()
5:     footerIndex ← ReadFooterIndex(inputStream, fileLength)
6:     offset ← footerIndex[columnPath]
7:     if offset = null then
8:         throw MissingColumnException(columnPath)
9:     end if
10:    inputStream.seek(offset)
11:    pageHeader ← ReadPageHeader(inputStream)
12:    return ReadCompressedDictionary(pageHeader, inputStream)
13: end function
14: function ReadFooterIndex(inputStream, fileLength)
15:    inputStream.seek(fileLength − 8)                    ▷ Read footer pointer
16:    footerStart ← inputStream.readLong()
17:    inputStream.seek(footerStart)
18:    index ← EmptyMap()
19:    while inputStream.getPos() < fileLength − 8 do
20:        nameLength ← readInt()
21:        nameBytes ← readBytes(nameLength)
22:        colName ← UTF8Decode(nameBytes)
23:        offset ← readLong()
24:        index[colName] ← offset
25:    end while
26:    return index
27: end function
```

corresponding to each encoded column. To enable a lookup during reads, we append a compact footer at the end of the file, mapping each column name to its byte offset in the file.

5.1.4 Reading a Column Dictionary from a Shared Dictionary File

Algorithm 5 outlines the complete procedure for locating and loading the dictionary page for a given column path/name.

Given Delta Lake's snapshot isolation model, readers must decode Parquet files using only metadata available in the active snapshot. This is crucial for correctness as different files in the same table version may have been written using different dictionary versions, and using the wrong dictionary would result in corrupted decoding.

During a read operation, the system traverses the list of active files defined by the current snapshot and groups them according to their declared dictionary metadata. If a

file includes a `globalDictionaryPath` field in its `AddFile` entry, the reader:

1. Loads the corresponding global dictionary file.
2. Uses the footer index (as described in Algorithm 5) to extract the dictionary page for the required column.
3. Applies that dictionary to decode values in the Parquet column chunk.

Files that do not include a dictionary reference simply fall back to the default Parquet decoding mechanism (typically plain or RLE encoding). This ensures backward compatibility and graceful degradation in the absence of dictionary metadata.

5.2 Decoupling Dictionary Creation from Writes

One of the key goals of supporting global dictionary compression in a scalable data lake environment is to ensure that dictionary construction does not become a bottleneck for data ingestion. In many real-world pipelines, write operations are latency-sensitive, small in size, and frequent, making it impractical to build dictionaries on-the-fly for every write.

To address this, we decouple dictionary construction from the write path. Instead of generating dictionaries as part of the write job itself, we enable dictionaries to be produced asynchronously by dedicated background processes. Writers then reference these precomputed dictionaries by version ID when encoding new data.

This section outlines how dictionary generation can be externalized, how initial dictionaries can be bootstrapped when none exist, and how this architecture improves both performance and usability.

5.2.1 Standalone Dictionary Jobs

We introduce a standalone job type, referred to as a *dictionary builder job*, that scans data already written to the table to produce one or more global dictionaries. These jobs can be scheduled periodically, or triggered manually after large ingests, and operate independently of the write path.

The dictionary builder performs the following steps:

1. **Frequency Estimation:** Use either full scans or approximate sketches (e.g., Misra-Gries) to determine the most frequent values in each column.
2. **Dictionary Encoding:** Construct dictionary pages from these top values and serialize them using the shared dictionary file format (as described in Section 5.1.3).
3. **Versioning and Output:** Write the resulting dictionary file to a designated storage path with a unique version identifier.
4. **Metadata Registration:** Update the Delta log's `DomainMetadata` field with the new dictionary path. This field acts as a canonical reference to the most recent shared dictionary for the table.

The use of `DomainMetadata` provides a lightweight and snapshot-consistent way to publish the latest dictionary path. When a writer begins a new append, it reads the current Delta table snapshot and extracts the latest dictionary path from the `DomainMetadata`. This path is then passed into the writer logic as part of its configuration.

5.2.2 Integration into the Write Path

A key advantage of our design is that it does not require any changes to the Parquet writer internals (`parquet-mr`). The dictionary path is passed via the standard Hadoop `Configuration` object, which is already used to propagate encoding options and table-specific metadata.

Initially, we were concerned about the possibility of race conditions such as a write job might load one dictionary path from the configuration, while a concurrent dictionary builder updates the `DomainMetadata` field and produces a new dictionary, resulting in an inconsistency between the file content and the associated `AddFile` entry.

However, experimentation and inspection revealed that each Hadoop configuration is treated as an immutable copy within the scope of a job. This means the value passed into the write job is fixed at the time of configuration construction, and is not affected by updates to the underlying table metadata that may occur during concurrent dictionary builds. Consequently, the same value is safely used both for encoding the file and for registering the correct dictionary version in the resulting `AddFile` entry.

This immutability property ensures consistency between the encoded data and its metadata declaration, without the need for additional locking or coordination logic. The result is a clean and modular integration that enables dictionary reuse without coupling the writer to dictionary lifecycle management.

5.2.3 Invoking Dictionary Generation Jobs

To make dictionary construction accessible and testable during development, we expose it through a user-facing command interface. Users can explicitly trigger dictionary construction using either SQL or the programmatic API:

- SQL: `GENERATE DICTIONARY my_delta_table`
- Scala/Java API: `DeltaTable.generateDictionary()`

Internally, this command instantiates the dictionary builder job described above, scans recent data in the specified Delta table, and produces an updated shared dictionary file. It then updates the table's `DomainMetadata` with the new dictionary path, making it available to future write jobs.

Although this command is currently user-visible for debugging and demonstration purposes, we expect that in most production scenarios, dictionary generation will be handled automatically in the background. In particular:

- The logic behind `GENERATE DICTIONARY` may be triggered periodically by the system based on ingestion volume or change detection.

- Alternatively, the dictionary builder could be invoked as part of existing maintenance operations such as `OPTIMIZE`, `ZORDER`, or data compaction commands.

This architecture preserves flexibility: users and systems can both initiate dictionary creation explicitly or implicitly, depending on workload requirements and operational policies. Over time, we envision dictionary generation becoming an invisible optimization step entirely abstracted away from end users.

5.2.4 System Integration Overview

To illustrate the complete flow from dictionary generation to data writing, we include a high-level system diagram in Figure 5.1.

This diagram shows how the dictionary generation job (which generates and "publishes" new dictionary versions to the delta table's `DomainMetadata`) and the delta table writes (which reads this metadata) are decoupled from each other. They can both occur asynchronously, dramatically reducing write times on the write path.

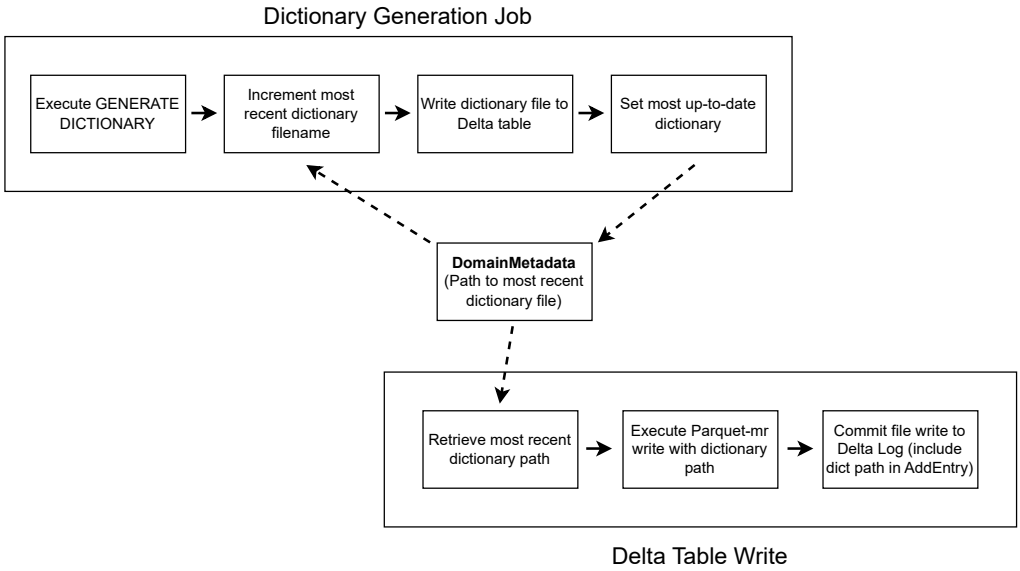

Figure 5.1: Overview of Dictionary Generation and Usage in Delta Write Pipeline

5.3 Experimental Evaluation: Incremental Write Performance

To understand the performance implications of using global dictionaries in an incremental ingestion setting, we designed a set of experiments that simulate how real-world data lakes ingest data in discrete batches over time. It is important to note that as we have seen in Chapter 4, hybrid dictionary encoding is the most effective in terms of storage compression. For the remainder of this thesis, we solely test using this encoding. Global and hybrid dictionary encoding are often used interchangeably.

Experimental setup: batch-based ingestion. In many production pipelines, large datasets arrive in recurring intervals or are split across multiple ingestion streams. To replicate this, we divide our datasets into evenly sized ingestion batches and evaluate the effect of global dictionary compression across these batch writes. For each dataset, we split the input data into $n \in \{3, 6, 9\}$ ingestion batches. Each batch may span multiple source files, but collectively the batches maintain the same overall row count. The batching logic ensures that:

- For non-synthetic datasets (e.g., NYC, CommonGov, Library Inventory), we select tables with identical schemas and evenly partition the union of all rows across all such tables into n batches.
- For the TPC-DS synthetic dataset, we focus only on the largest table with a stable schema (`store_sales`), as other tables vary significantly in row count and schema complexity. This allows us to avoid the added complexity of cross-schema joins or column mismatch handling.

Each batch is written independently to a Delta table, simulating incremental ingestion. Between batch writes, we optionally enable the dictionary builder to update the global dictionary used for encoding subsequent data.

In practice, it may also be the case that delta tables are updated constantly in smaller increments each time. In this case, we would not propose generating new dictionaries between each small write, as the overhead of generating dictionaries would become immense and yield no additional benefit. Our experiment set-up remains valid in this case, as the core utility of removing the dictionary generation out of the write path and into a separate job, is we can wait until a sufficient volume of data has been written, or until the dataset distribution has changed sufficiently to run the dictionary generation job.

We compare two configurations:

- **Baseline (no dictionary):** All batches are written using plain encoding or default Parquet dictionary encoding. No global dictionaries are used.
- **Hybrid Dictionary (enabled):** After each batch is written, the `GENERATE DICTIONARY` command is invoked to build a shared dictionary from the current table state. This dictionary is then referenced in the Hadoop configuration and used to encode the next batch.

Note that the first batch in the dictionary-enabled setting is always written without compression, since no dictionary exists at that point. From batch 2 onward, each write uses the most recently generated dictionary, ensuring a realistic approximation of evolving ingestion pipelines.

As with all other compression tests, for each experiment, we collect the total output size across all batches, including the Parquet and global dictionary size. *Hybrid* dictionary encoding is used with a minimum key count of 6.

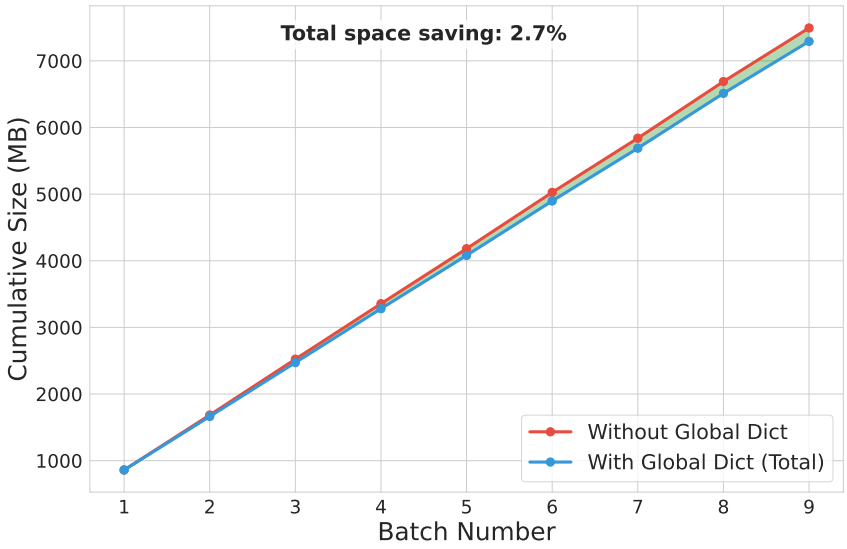

Figure 5.2: Cumulative storage footprint progression with and without hybrid dictionary encoding for the CommonGovernment dataset. Results are based on writing 9 equal-sized ingestion batches.

5.3.1 Storage Size Results

We begin our experimental evaluation by analyzing the storage behavior of the Common Government dataset, which serves as a representative real-world case with structured tabular data. This dataset contains a large number of rows distributed across multiple files with the same schema, allowing us to divide it evenly into ingestion batches.

Figure 5.2 presents the results of writing this dataset in 9 sequential batches. Each batch writes an approximately equal number of rows. We compare two configurations: one using hybrid dictionary encoding (enabled after the first batch), and one using baseline Parquet encoding without hybrid dictionaries.

As expected, both configurations produce identical storage sizes for the first batch. This is because the hybrid dictionary mechanism has not yet been initialized since no prior data exists from which to generate a dictionary. Consequently, the first batch is written using default encoding in both cases.

From the second batch onward, we observe consistent space savings in the configuration that employs hybrid dictionaries. Figure 5.2 shows per-batch percentage savings relative to the baseline. These savings begin at 1.3% in batch 2 and gradually increase to approximately 2.7% by batch 9. The bottom panel displays cumulative storage usage, highlighting the growing divergence between the two configurations over time. The total cumulative space reduction is 2.7%.

These results reveal several key observations that hold across most datasets and batching scenarios in our evaluation:

- **Linear Storage Growth.** The cumulative storage footprint grows linearly with the number of ingestion batches in both configurations. This is an expected property due

to uniform row counts across batches. It also indicates that the presence of global dictionaries does not introduce nonlinear behavior such as file compaction or data skew.

- **Compression Benefit/Loss Magnifies.** The space savings due to global dictionaries accumulate consistently across batches, without large fluctuations. This implies that each new dictionary version retains enough representativeness from prior data to capture recurring patterns effectively. However, while the space savings appear to only grow after we observe positive space savings from the first batch, the inverse is true for datasets that exhibit compression loss. For example, in Figure 5.3, we see that the loss begins at -1.4% (at batch 2) and magnifies until it reaches -5.2% at batch 9. This is observation is critical for production systems as it means that, in cases where we want to apply global dictionaries, *we can observe their performance over a single batch and compare the per-row compression statistics compared to a previous, non-globally encoded batch.* We can then reliably use these compression statistics to decide if we want to continue.
- **Predictable Behavior Across Batches.** The fact that per-batch savings increase smoothly without sharp drops or spikes indicates that the dataset does not exhibit abrupt shifts in cardinality or value domains between partitions. This stability improves the reliability of dictionary reuse across versions.

The consistent but moderate savings suggest that global dictionaries are beneficial even in scenarios where data distributions are uniform or slowly evolving. The additive nature of the savings further highlights the potential for long-running tables to gradually benefit from dictionary reuse without requiring dramatic structural changes.

Although the Common Government dataset illustrates favorable and consistent results, we note that not all datasets behave identically. In particular, the NYC dataset demonstrates a more curved growth pattern. However, even in that case, the trend appears to be driven more by inherent properties of the data than by the use of global dictionaries themselves.

5.3.2 Space Savings and Asymptotic Behavior in TPC-DS

To better understand the long-term effect of hybrid dictionaries on storage efficiency, we analyze the TPC-DS `inventory` table. The table is written in nine ingestion batches, and Figure 5.4 shows the cumulative space savings achieved after each batch when using hybrid dictionaries. It displays, by far, *the largest space savings of any dataset*, reaching up to 31% reduction in storage.

We define:

- $f(n)$: the total storage size (in bytes) of the first n batches written *without* global dictionaries,
- $g(n)$: the total storage size of the first n batches written *with* global dictionaries, and
- $S(n) = 1 - \frac{g(n)}{f(n)}$: the cumulative space savings function after batch n.

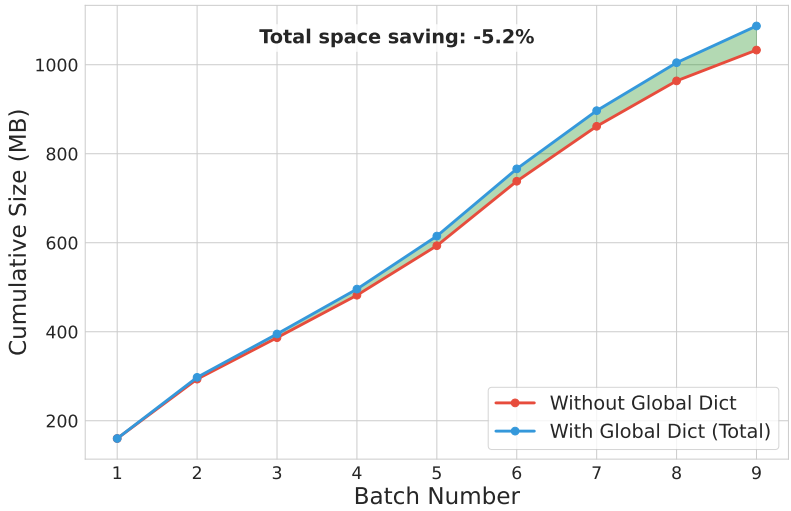

Figure 5.3: Cumulative storage footprint progression with and without hybrid dictionary encoding for the NYC dataset. Results are based on writing 9 equal-sized ingestion batches.

We observe a pattern seen in *all* trials across all datasets and batch counts. In the first few batches, savings increase sharply from 0% to nearly 18% after just two batches. However, this growth rate gradually slows down, with savings reaching approximately 31% by the ninth batch. This trajectory is characteristic of an *asymptotic convergence pattern*, where the difference in performance between two systems (in this case, hybrid dictionary vs. non-dictionary writes) increases rapidly at first but approaches a limit over time.

Mathematically, this suggests:

$$\lim_{n \to \infty} S(n) = S^* \approx 31\%$$

This implies that both $f(n)$ and $g(n)$ grow approximately linearly with batch count, but that $g(n)$ maintains a constant proportional advantage in terms of size. In asymptotic terms, this means:

$$g(n) \in \Theta(f(n)),$$

indicating that while global dictionaries improve compression, they do not change the fundamental scaling behavior of storage growth.

This pattern reflects a natural saturation point in dictionary learning. Early batches introduce the most common and compressible domain values, enabling significant gains. Subsequent batches add incrementally less value to the dictionary as they begin to resemble previously seen data. This leads to a flattening of the savings curve.

Interestingly, even though individual batches may differ in content, the relative effectiveness of global dictionaries converges. The per-batch bars in Figure 5.4 also begin to stabilize in height toward the right side of the plot, visually reinforcing this notion of convergence.

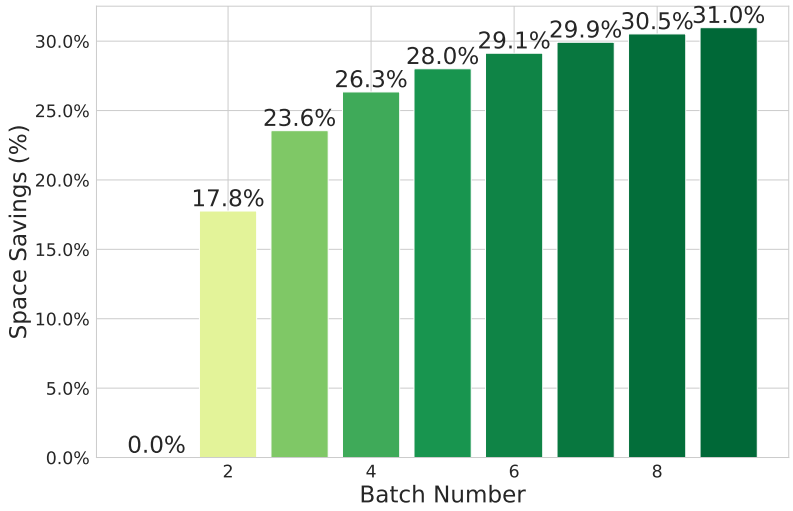

Figure 5.4: Per-batch space savings using hybrid dictionaries on the `inventory` table from TPC-DS.

From an engineering standpoint, this asymptotic behavior justifies frequent dictionary refreshes in early stages of ingestion but suggests diminishing returns beyond a certain point. For large-scale or continuous ingestion workflows, it may be optimal to update the dictionary aggressively during early ingestion phases and then reduce the frequency of regeneration as savings plateau.

5.3.3 Effect of Batch Count on Compression Effectiveness

To further understand the behavior of global dictionaries under incremental ingestion, we analyze how the number of ingestion batches impacts compression effectiveness. Figure 5.5 plots the compression ratio, defined as the total size of files written using hybrid dictionaries divided by the size of those written without, for 3, 6, and 9 batch configurations across four representative datasets.

A clear pattern emerges: the number of ingestion batches amplifies the effect, either positively or negatively, of global dictionaries. When dictionary encoding is beneficial, as in the case of TPC-DS, the advantage becomes more pronounced as batches increase. The compression ratio improves from approximately 1.32 at 3 batches to 1.45 at 9 batches. Conversely, when dictionaries are detrimental, as observed in the *nyc* dataset, this negative impact becomes more severe with more batches.

This trend is not coincidental. When a new dictionary is generated after every batch, fewer rows are written using a stale dictionary that may no longer match the evolving distribution of the data. If upcoming records include values not yet seen or sufficiently frequent in earlier batches, using the older dictionary risks encoding them inefficiently or not at all. By regenerating dictionaries more frequently, we reduce the number of rows subject to such mismatch, and effectively limit the duration over which a suboptimal dictionary is applied.

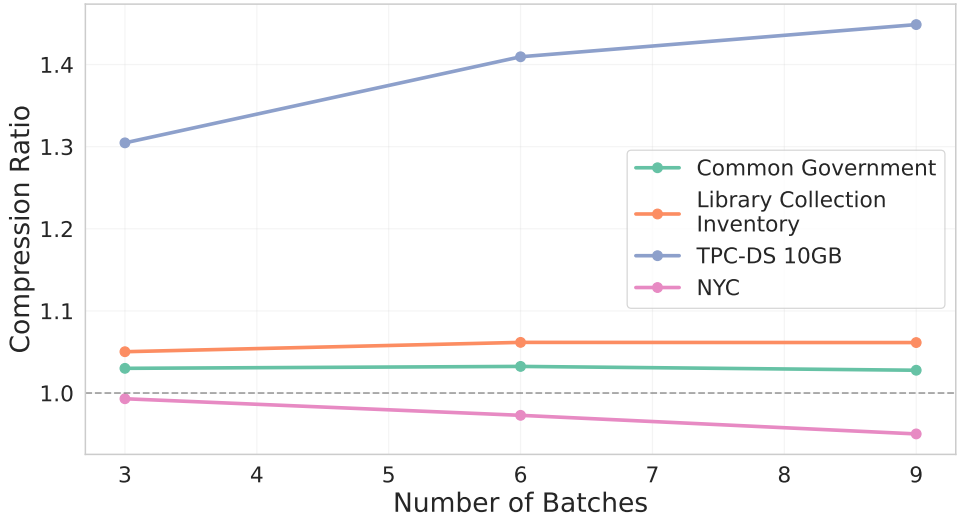

Figure 5.5: Compression ratio (Hybrid / Non-Hlobal) vs. Number of ingestion batches

In this light, the incremental update model acts as a distribution-aligned checkpointing mechanism. Rather than committing to a single global vocabulary from the start, the system adapts periodically, allowing each batch to calibrate its dictionary to the most recently observed value distributions. Datasets with stable or slowly evolving domains, like *CommonGovernment* and *LibraryCollectionInventory*, show modest but consistent gains under this model. Their compression ratios remain above 1 and trend upward with more batches, confirming that dictionaries increasingly capture repeated patterns as the dataset grows.

For *NYC*, on the other hand, the compression ratio rises with batch count, indicating that even frequent dictionary refreshes cannot keep up with shifting distributions. In such cases, stale dictionaries introduce encoding inefficiencies that accumulate with each batch. These results suggest that batch-based dictionary updates are highly effective when the data distribution remains relatively stable but sorely need strategies to adaptively shut global dictionary encoding off when inefficient.

5.4 Comparing Incremental vs. One-Shot Dictionary Generation

In earlier sections, we evaluated global dictionaries in a static setting, where dictionaries were constructed from a batch of data and used immediately for encoding. We believe that, while, unrealistic in practice, such an approach could provide the more optimal dictionaries as they take the entire dataset's value distribution into account beforehand. In this section, we investigate if this is indeed the case, or if incremental dictionary generation benefits from using localized value distributions. However, the static approach tightly couples dictionary

generation with the write path, meaning the data must be available in full before any write operation begins.

In realistic ingestion pipelines, especially those involving streaming or append-only batch ingestion, this assumption no longer holds. The dictionary must be constructed *incrementally*, based only on data that has already been ingested. Once new data arrives, a dictionary can be refreshed to incorporate the updated domain knowledge. This motivates a key shift in system design: global dictionaries are now decoupled from the data write path, generated as a separate asynchronous job. As a result, it becomes infeasible to precompute a dictionary over the entire dataset before ingestion begins.

5.4.1 Stealing a Dictionary from an Existing Table

To simulate a static, one-shot dictionary under this new asynchronous model, we introduce a mechanism for **dictionary stealing**. The idea is simple: if a full dataset has already been ingested and its global dictionary constructed incrementally, a second table can reuse that dictionary for evaluation purposes.

Specifically, the second table updates its `DomainMetadata` field in the Delta transaction log to point to the final dictionary file generated by the first table. As described in Section 5.1, this metadata is read at the beginning of each write operation and passed into the writer via Hadoop configuration. Since each write job receives a copy of the configuration object, the metadata remains stable and isolated across jobs, even in the presence of concurrent updates.

5.4.2 Implementation Considerations

There were multiple possible approaches to implementing this dictionary stealing protocol.

Symbolic Link Approach. The initial implementation used a lightweight symbolic link: the second table's `DomainMetadata` simply stored the path to the first table's dictionary file. This avoids any actual data copying and enables fast reuse. We implement and expose such behavior through the Scala/Java API via *DeltaTable.setSymbolicDictionary(sourceTablePath)*.

However, this approach suffers from long-term fragility. If the source table is vacuumed, a common operation in Delta Lake that removes unused files, or deleted entirely, the target table's dictionary reference would become invalid. Worse, such failures would likely manifest only during later read operations, resulting in brittle downstream behavior.

It is worth noting that Delta Lake's `SHALLOW CLONE` feature uses a similar symbolic referencing model, where a cloned table points to the original table's data files. However, such shallow clones are typically short-lived, used for testing or staging. In contrast, global dictionaries are intended to persist throughout the lifetime of a production table and must remain durable and consistent.

Physical Copy Approach. To avoid these issues, we implemented a more robust solution: copying the dictionary file directly into the target table's storage directory. This guarantees isolation from upstream lifecycle events, ensuring that the copied dictionary remains valid even if the source table is later vacuumed or deleted.

Upon copying, the receiving table updates its own `DomainMetadata` to reference the new local dictionary file. From this point forward, the table operates entirely independently, reading the dictionary as if it had generated it itself. We expose this functionality through the Scala/Java API via *DeltaTable.copyDictionaryFrom(sourceTablePath)*.

5.4.3 Experimental Design

To compare the effectiveness of incremental and one-shot global dictionary generation, we conduct the following experiment:

1. We first ingest each dataset incrementally in 3, 6, and 9 batches, updating the dictionary between each batch using the standard incremental workflow.
2. After all data has been ingested, we extract and save the final dictionary generated.
3. We then create a new table, copy the final dictionary into it using the physical copy method described above, and update its `DomainMetadata` accordingly.
4. Finally, we write the same dataset again into this new table, using the copied (one-shot) dictionary from the beginning of ingestion and *not* refreshing it across batches.

This setup allows for a controlled comparison between the two:

- **Incremental Dictionary Mode**, where the dictionary evolves and improves as ingestion progresses,
- **One-Shot Dictionary Mode**, where a fixed dictionary, generated after full ingestion, is reused from the outset.

Each setting uses identical batch counts and input data. By measuring cumulative and per-batch storage sizes, we can assess whether the one-shot dictionary approach yields better compression, and whether the incremental approximation converges toward the one-shot ideal as more data is ingested.

5.4.4 Evaluating Incremental vs. One-Shot Dictionary Approaches

In the final set of experiments, we compare the effectiveness of two dictionary generation strategies: the incremental approach described earlier, where a new dictionary version is generated after each batch, and a one-shot dictionary approach, in which a dictionary is created using the full dataset and reused across all future batches via dictionary stealing.

Figure 5.6 presents the results for the Common Government dataset. The plot shows per-batch cumulative file sizes as partitions are incrementally added to the dataset.

The overall trend is consistent and unsurprising: the one-shot dictionary consistently outperforms the incremental strategy in terms of compression. The cumulative savings reach approximately 2.3% in favor of the one-shot method. On a per-batch basis, we

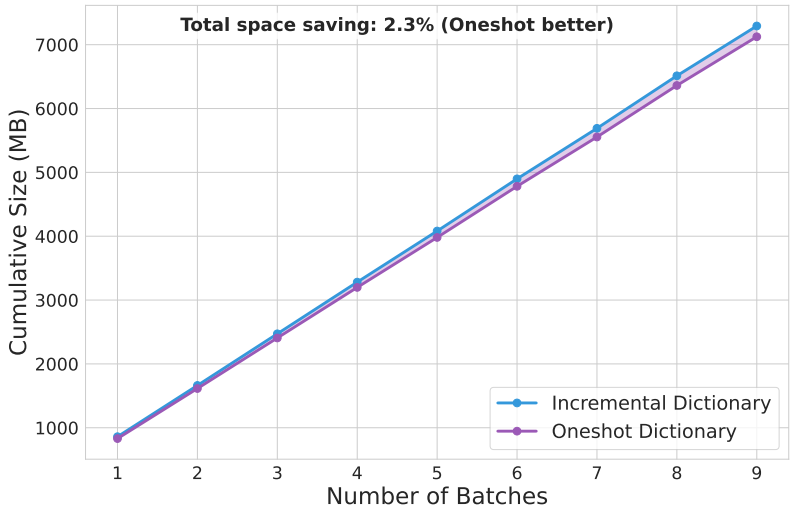

Figure 5.6: Comparison of Incremental vs. One-Shot Dictionary Cumulative Storage Sizes for Common Government Dataset

observe that the initial batches benefit significantly from having access to a comprehensive dictionary built on the full domain as batch 1 achieves a savings of 3.4%, compared to essentially zero in the incremental setup (which lacks any prior dictionary at that point).

However, the relative difference between the two approaches narrows as more batches are ingested. While the oneshot dictionary offers an initial "headstart", this has no impact on its future performance other than that the table's size remains smaller by a relatively constant (or slowly growing) difference. As both tables grow, this difference grows proportionally less significant.

This pattern of strong initial performance gap followed by convergence is consistently observed across the other datasets as well. It supports two key conclusions:

1. The one-shot strategy unsurprisingly has an advantage, since it encodes the entire dataset with the most complete dictionary possible.
2. Incremental dictionaries, although slightly suboptimal, still achieve comparable compression in the long run, with minimal overhead.

From a system design perspective, this is a favorable result: it indicates that even without access to the full data up front, an unrealistic assumption in streaming or transactional settings, Delta Lake can achieve nearly optimal compression using only incrementally generated dictionaries. The minor differences observed in early batches can often be tolerated or amortized across the lifetime of the table.

5.5 Efficient Dictionary Estimation with Sketches

Throughout this thesis, we have focused solely on compression. However, to make this solution viable in production systems, it becomes necessary to balance the compression gains against the inevitable performance hit caused by generating the dictionaries.

The goal of this section is to formalize the problem of dictionary candidate estimation under resource constraints and show how sketch-based methods can deliver scalable and accurate approximations that serve as a drop-in replacement for exact frequency histograms.

5.5.1 Why Exact Frequency Computation is Expensive

The existing implementation for generating global dictionary candidates relies on computing exact value frequencies using group-by aggregation (as explained in Section 4.2.1). For each column, we perform an aggregation that groups all non-null values, counts the occurrences of each unique value, filters those below a frequency threshold, and returns the most common values in descending order.

This approach is executed in Spark as a multi-stage transformation involving a full data shuffle. Each executor node first computes partial value counts, which are then shuffled across the cluster for merging into global aggregates. Because the grouping key determines the partitioning of data, this operation triggers a wide dependency, resulting in expensive disk and network I/O.

Moreover, these group-by aggregations must be executed independently for each column. Since each column's values must be partitioned and grouped based on their own unique values, there is no opportunity to share execution plans or reuse scans. This means that writing a dataset with N columns can require N separate aggregation jobs, each scanning and shuffling the data in full.

Figure 5.7 shows the physical execution plan Spark generates for a typical frequency estimation query on a single column. This diagram represents a sequential dataflow through Spark's physical operators:

The plan starts with a *Parquet scan*, followed by a filter to exclude nulls. The data is then grouped and counted using a *hash aggregation*, which builds an in-memory hash table to track the count of each unique value. Once local aggregation is complete, the data is shuffled via an *exchange*, meaning that partial results are redistributed across the cluster so that values with the same key end up on the same node for global aggregation. Spark then performs an *AQEShuffleRead*, a mechanism that dynamically optimizes shuffle partition sizes at runtime to improve load balancing. Finally, the results are sorted and projected to select the top values. While Spark's internal optimization mechanisms attempt to fuse stages where possible, this plan still incurs a large materialization and shuffle cost.

As datasets scale in size and column count, this cost becomes prohibitive. Recomputing exact value counts for every column at every write is not sustainable for large-scale ingestion pipelines. These limitations motivate the exploration of approximate algorithms that can deliver similar value distributions with far less computational overhead. We explore this direction in the following section.

5.5.2 Sketch Selection and Design Tradeoffs

To avoid the full materialization and shuffle costs associated with group-by aggregations, we adopt an approximate frequency estimation method based on the *Misra-Gries* algorithm

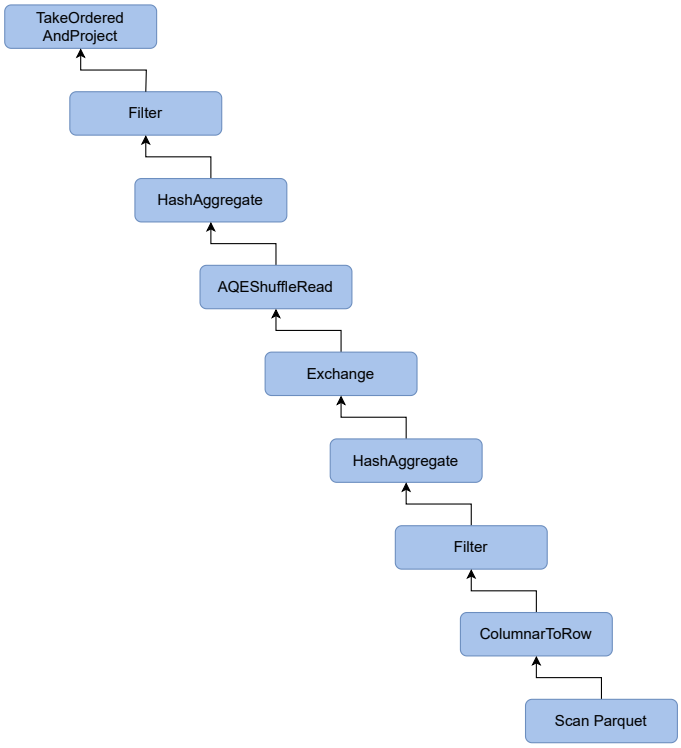

Figure 5.7: Spark physical plan for group-by and count aggregation on one column.

(38). This sketching algorithm offers a scalable and memory-efficient alternative to exact counting while still identifying the most common values in each column.

The core idea behind Misra-Gries is to maintain a bounded-size summary of the input stream by keeping track of at most k item-count pairs. The algorithm processes data in a single pass. For each incoming value, it checks if the value is already in the map:

- If it is, its count is incremented.
- If it is not and the sketch has fewer than k items, it is added with a count of one.
- If the sketch already tracks k items, the count of every item is decremented by one. Items whose count reaches zero are evicted.

This mechanism ensures that any element with a frequency greater than $\frac{1}{k+1}$ will be present in the sketch at the end, albeit with some overestimation. The final output consists of the top k items sorted by approximate count.

The actual implementation used in this thesis is the native *approx_top_k* function built into the Databricks platform. Internally, this function augments the basic Misra-Gries logic with an eviction policy that uses the median count of all tracked items to decide which entries to prune and by how much. This approach is adapted from Reduce-By-Median strategies that offer improved robustness for skewed data distributions. After every insertion or merge, the sketch may run a *compression step* to evict half of the least frequent items and decrement the others to stay within its target capacity.

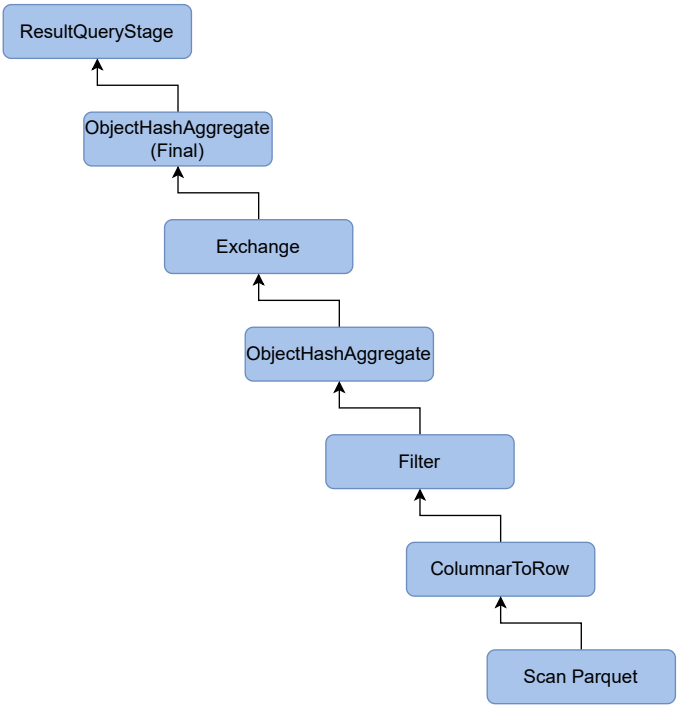

Figure 5.8: Spark physical plan for sketch-based estimation across multiple columns.

Beyond its theoretical advantages, one of the most compelling benefits of sketch-based estimation lies in its practical execution speed when deployed in distributed systems like Spark.

With exact dictionary generation, each column requires its own group-by aggregation. This means performing a full column scan and data shuffle N times for a table with N columns. Because the grouping logic is specific to each column, Spark cannot reuse the plan or combine aggregations across columns, resulting in duplicated compute and I/O.

In contrast, sketch-based estimation avoids this redundancy entirely. Because sketches can be updated incrementally and independently for each row, the same input scan can feed multiple sketches in parallel. This means that all columns can be processed in a single pass over the dataset. Only one shuffle and aggregation stage is required, regardless of the number of columns being processed. As the dataset will be too large for a single node, we horizontally partition the rows such that each node gets their own range(s) of rows containing all of the column data to process.

Figure 5.8 illustrates the physical execution plan used by Spark to apply `approx_top_k` aggregations over multiple columns in a single job.

This plan begins with a *Parquet scan*, followed by a filter to remove nulls. The data is then processed using an *ObjectHashAggregate*, which applies the sketch update logic for each row and each column in the same stage. The intermediate results are *exchanged* across the cluster for merging, followed by a final aggregation phase that produces the top-k values per column.

Unlike traditional hash aggregation, which must maintain and merge large, distinct hash tables per grouping key, the sketch aggregation logic is constant-bounded in memory and highly parallelizable. There is no need to materialize large intermediate state or coordinate between columns.

In essence, sketch-based estimation:

- Requires only a single scan over the data, independent of column count.
- Incurs just one shuffle and merge step for all columns collectively.
- Uses compact in-memory data structures with bounded space guarantees.
- Eliminates the overhead of materializing complete group-by keys or counts.

This enables it to scale gracefully even for less-wide tables or streaming data pipelines, offering a practical and performant solution to global dictionary candidate generation.

5.5.3 Experimental Evaluation: Efficiency Gains from Sketching

Experimental setup. To evaluate the performance benefits of sketch-based global dictionary estimation, we benchmarked the runtime of our approach against the traditional group-by method across multiple datasets. Each experiment was repeated 10 times on a fixed compute environment to ensure consistency and eliminate external noise.

Cluster Configuration (AWS):

- Platform: Amazon Web Services (AWS)
- Instance Type: EC2 m6id.2xlarge
- Driver: 1 instance (8 vCPUs, 32 GiB memory, 474 GB NVMe SSD)
- Workers: 2 instances (same as driver), each with up to 12.5 Gbps network and 10 Gbps EBS bandwidth

These general-purpose instances are run Intel Xeon Scalable processors (Ice Lake 8375C), with up to 3.5 GHz clock speed.

We ran the dictionary generation logic on five datasets of varying size and complexity. For each dataset, we compared:

- The baseline method, which performs an exact group-by count on every column individually.
- The sketch-based method using horizontal partitioning, which uses the Misra–Gries approximation to estimate top values in all columns in a single pass.

The benchmark was conducted using the same logic and sampling parameters, with k and $maxItemsTracked$ equal to the estimate maximum number of elements (that fit into a 2MB dictionary page). The key metric is total execution time in seconds for each estimation job.

Observations:

- The sketch-based implementation was consistently faster across all datasets.

Table 5.1: Average execution time and speedup for baseline, horizontally partitioned, and vertically partitioned sketches.

Dataset	Baseline Avg (s)	Horizontal Sketch Avg (s)	% Speedup (Hori.)	Vertical Sketch Avg (s)	% Speedup (Vert.)
TPC-DS 1TB	95.72	59.48	37.85%	80.86	15.52%
TPC-DS 10GB	14.41	11.84	17.88%	14.07	2.36%
Library Inventory	52.36	49.41	5.63%	39.76	24.02%
NYC	58.59	51.29	12.46%	32.53	44.49%
Common Government	398.48	311.91	21.78%	260.36	34.71%

- The largest gains were observed on the 1TB TPC-DS dataset (nearly 38% faster), where the baseline method took close to 100 seconds on average, versus under 60 seconds with sketches.
- Smaller datasets such as TPC-DS 10GB and Library Collection Inventory still benefited, with up to 18% and 6% reductions in runtime respectively.
- The Common Government dataset, which is extremely wide, saw over 20% speedup, highlighting the benefit of avoiding multiple passes over the data.

Larger datasets benefited more from sketching due to a combination of amortized overhead (single scan over multiple columns), reduced shuffling, and more efficient parallel execution. As the cost of exact group-by operations increases linearly with column count and data volume, the one-pass sketch-based method scales much more favorably.

5.5.4 Sketch Optimization: Vertical Partitioning

We notice from running the horizontally partitioned sketch, that the network overhead becomes large, especially with wider tables containing more columns. Therefore, in an attempt to tackle this, we minimize the network overhead by *vertically partitioning* the dataset.

This entails, instead of giving a particular process all the column data for a specific range of rows, we give that process all of the data for a set of columns. Since the process maintains complete ownership of these columns, they need exchange no intermediate data over the network pertaining to any sketch passes. The drawback to this is that the dataset now needs several passes over the dataset again.

Such a method can become useful in the event of selective column global dictionary generation, where we only build dictionaries for columns we predict will compress well. In such cases, we can eliminate a scan over that column, saving compute.

The results are also presented in Table 5.1. It can be seen that, in general, this approach works better for the heavier datasets like NYC and Common Government. This makes sense as they will likely have more data to send over the network and consider such latency a bottleneck. For smaller datasets like TPC-DS 10GB, the network overhead is minimal,

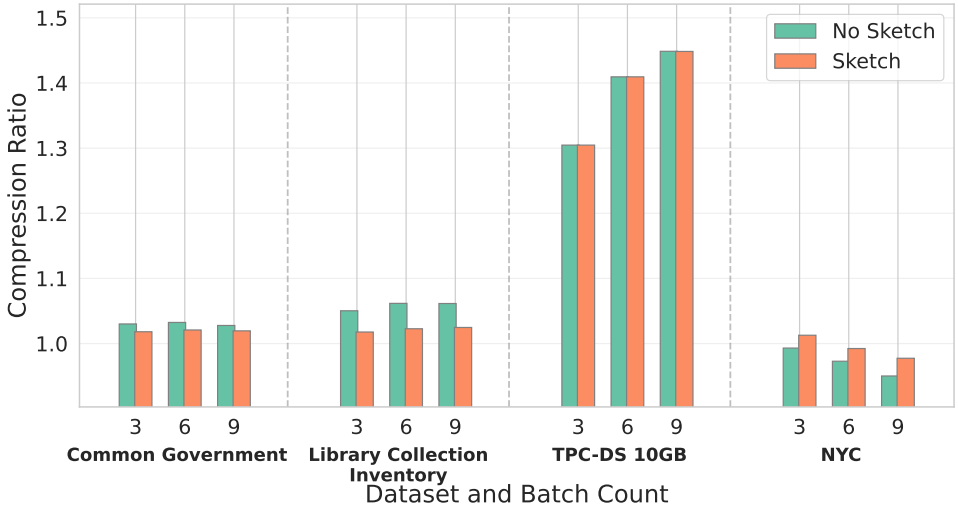

Figure 5.9: Final compression ratio with and without sketches (higher is better).

and superfluous scans over the dataset becomes the bottleneck. In realistic scenarios, we expect datasets to be far larger and heavier than what is presented here, making vertical partitioning a sensible choice in our sketch design.

5.5.5 Compression Accuracy of Sketch-Based Dictionaries

While sketches significantly reduce the computational overhead of dictionary generation, it is important to assess how this efficiency impacts final compression outcomes. In this section, we compare the compression ratio achieved using approximate dictionaries (generated with sketches) against exact dictionaries based on full-frequency histograms.

Figure 5.9 presents the final compression ratio for each dataset at each batch count.

Overall, we find that exact dictionaries consistently produce slightly smaller/better outputs. However, the degradation in space efficiency when using sketches is modest and typically remains within a 1–3% range. In many cases, such as the *Common Government* and *Library Collection Inventory* datasets, the differences are minimal suggesting that the Misra-Gries sketch is able to recover the most frequent values sufficiently well to preserve compression benefits.

Interestingly, in the *NYC* dataset, sketches even produce marginally better results than the baseline. This is likely due to the inherently weak compressibility of the dataset with either method, and small statistical fluctuations or selection artifacts may have led the sketch to slightly favor compressible subsets.

In the case of the highly compressible *TPC-DS 10GB* dataset, sketches show neither a decline nor improvement in performance relative to exact dictionaries. However, even in this scenario, the benefit of using sketches must be weighed against the significantly improved runtime and resource efficiency documented earlier. When deploying compression pipelines in production environments, this tradeoff is likely to be well justified. Exact

dictionaries remain the gold standard for achieving maximum compression efficiency, but sketches offer a practical and scalable approximation with only minor compression loss. Especially in contexts where ingestion latency or resource usage is a concern, sketch-based methods provide a compelling balance between performance and compression quality.

5.5.6 Further Optimizations

While the introduction of sketch-based frequency estimation significantly reduces the cost of dictionary generation, there remain additional opportunities for improving efficiency further, particularly when considering how and when data is sampled for dictionary creation.

First, as demonstrated in earlier sections, generating high-quality dictionaries does not necessarily require scanning the entire dataset. In many cases, recent ingestion batches alone offer a sufficient representation of the column value distribution. This observation opens the door to partial estimation strategies, where only a sliding window of recent data is analyzed to build or update dictionary candidates. Such a strategy would dramatically reduce I/O and computation, especially in environments where older data is less relevant to future distributions (e.g., in event logs or streaming append scenarios).

Second, the nature of the Misra-Gries algorithm itself presents a more powerful optimization opportunity. Since the algorithm is designed for single-pass streaming scenarios, we can model dictionary candidate estimation as a long-running aggregation over an infinite data source, namely, the incoming ingestion stream. The state of the algorithm (i.e., its internal count map) can be periodically suspended and persisted between batches. By storing this internal state structure, specifically, the key-count pairs that comprise the sketch alongside the global dictionaries themselves within the Delta table or auxiliary metadata files, we enable resumable estimation.

This means each new ingestion batch need only update the persisted sketch state with its own data, rather than recomputing statistics from scratch. Operationally, this transforms dictionary generation into an incremental, stateful process where each sketch update only sees the delta of new data. By persisting and loading this sketch state efficiently, the system can continue making frequency-based encoding decisions over long-running ingestion pipelines without repeated full-table scans or aggregations.

Taken together, these optimizations could allow global dictionary systems to become virtually invisible to the user in terms of runtime impact, offering near-constant-time updates and continuously adaptive compression behavior in production environments.

5.5.7 Recovering Cardinality Awareness with HyperLogLog

While sketch-based estimation using Misra–Gries successfully replaces full-frequency histograms for identifying high-frequency values, it comes with a notable limitation: it provides no information about column cardinality. This is a significant drawback, as potential future compression heuristics, particularly the hybrid dictionary throttling logic discussed in Section 4.5.5, depend heavily on cardinality signals to determine whether to throttle the size of the global dictionary. An increase in dictionary storage size can accommodate

an increase corresponding increase in cardinality in such a way that it matches/exceeds that of the column. We found that in such cases, the column compression ratio improves dramatically.

To address this, we propose augmenting the dictionary estimation pipeline with a second sketch: *HyperLogLog* (HLL), a probabilistic algorithm for estimating the number of distinct elements in a multiset (39). HLL works by hashing each element and recording the position of the leftmost 1-bit in the hash value. The distribution of these positions across the dataset forms the basis for an extremely compact estimate of the number of unique values. With fixed-size memory (typically 1–2 KB per column), HLL achieves relative error rates below 2% in practice.

By pairing each Misra–Gries sketch with an HLL sketch, the system can recover cardinality-aware behavior even under approximate estimation. The encoding planner can use the HLL output to either estimate column uniqueness and compute the uniqueness ratio (a useful heuristic as discussed in Section 4.4.6) or inform adaptive dictionary size tuning or fallback tolerance thresholds.

5.6 Read and Write Performance Analysis

While the primary goal of using global dictionaries is to reduce storage space, it is important to evaluate the trade-offs this introduces in terms of read and write performance. Specifically, since the dictionary is stored externally, both read and write operations require an additional access step to retrieve dictionary values. This introduces an additional I/O step absent in standard Parquet encoding.

In this section, we evaluate the extent (or existence) of performance degradation during typical data access scenarios. Dictionary generation is excluded from this analysis, as we focus solely on the performance of writing values to disk and reading them back when the external dictionary is already available.

Some level of degradation is expected due to the added I/O and lookup complexity. However, the goal of this evaluation is to determine whether the performance loss is minor and acceptable, or whether it poses a practical limitation to adoption.

To evaluate this impact, we benchmarked five datasets in two configurations: a baseline without global/hybrid dictionaries and an initial implementation that uses global/hybrid dictionaries (without any caching). Each test was run for 10 iterations, and the average execution time was recorded for both read and write operations. The results for the initial write and read performance are presented in Table 5.2 and Table 5.3 respectively.

The results show that performance degrades noticeably with hybrid dictionaries, even for relatively small datasets. While the TPC-DS 10GB dataset shows only a mild increase in execution time, larger datasets such as NYC and Common Gov exhibit slowdowns in excess of 60%, with read performance degrading by over 100%. These results suggest that the current implementation of global dictionaries introduces significant I/O overhead, particularly in cases where dictionary files are large or when many dictionary pages are accessed independently across columns.

Table 5.2: Adjusted write performance across baseline, global dictionary, and cached global dictionary configurations. Each result is the average of 10 trials.

Dataset	Baseline Time (s)	Global Time (s)	% Slower	Cached Time (s)	% Slower
TPC-DS 1TB	74.37	78.44	5.47%	77.03	4.93%
TPC-DS 10GB	15.86	17.07	7.61%	17.45	10.01%
Library Collection	34.01	39.99	17.57%	36.53	7.46%
NYC	43.55	70.81	62.59%	50.05	14.86%
Common Gov	352.64	574.96	63.05%	405.02	14.86%

Table 5.3: Adjusted read performance across baseline, global dictionary, and cached global dictionary configurations. Each result is the average of 10 trials.

Dataset	Baseline Time (s)	Global Time (s)	% Slower	Cached Time (s)	% Slower
TPC-DS 1TB	27.31	30.57	11.91%	26.04	-4.68%
TPC-DS 10GB	4.82	5.34	10.78%	5.03	4.25%
Library Collection	13.83	21.73	57.09%	15.06	8.96%
NYC	13.45	31.23	132.27%	15.00	11.57%
Common Gov	88.94	239.49	169.26%	100.46	12.91%

Initial attempts to diagnose the cause using Spark job statistics were inconclusive. Many metrics relevant to I/O behavior—such as cache hit rates or read amplification—were either unavailable or sparsely populated by parquet-mr. However, one consistent signal across all jobs was the elevated network traffic associated with global dictionary reads. This suggests that the performance bottleneck is likely due to the need to repeatedly fetch dictionary content from remote storage during read and write operations. Because the global dictionary is external, every lookup incurs additional latency compared to standard inline dictionary encoding in Parquet.

5.6.1 Caching Optimization

To mitigate the I/O bottlenecks introduced by global dictionaries, we implemented a basic caching strategy. We reason that the size of the global dictionary files is not the problem (see Table 4.3), but rather the I/O wait time. This is because for each column in each row group, we make a request to the dictionary file, incurring a roundtrip wait each time. To fix this, instead of this repeated fetching, we prefetch the entire dictionary file once (per node) and keep it in an in-memory Guava cache. This reduces the number of round trips to cloud storage and significantly improves performance. The results for the cached write and read performance are likewise presented in Table 5.2 and Table 5.3. It can be seen that, relatively speaking, the read and write times reach a near parity with their non-global

counterparts. In the case of reading TPC-DS 1TB, we even see a slight *improvement*, which is likely caused by the dramatically reduced storage size.

In the write path, prefetching the entire dictionary file is always a good idea, as we will always need the dictionaries for all columns. For the read path, this may be wasteful, as we may often need dictionary pages for only a few columns. This is naturally the whole point of columnar storage in OLAP systems.

Nevertheless, this, albeit rudimentary, caching solution serves as a proof of concept. A more integrated and performant alternative would be to leverage Databricks' *DBIO cache*, which is designed to fetch blocks at a time into memory for efficient reuse. This could mean that when we grab a column's dictionary page, we could also fortuitously grab a nearby column's dictionary. Unfortunately, this block-level caching system is not currently supported by the open-source `parquet-mr` library we rely on (see Section 2.4). As such, our solution avoids Databricks-specific enhancements to preserve compatibility and reproducibility in broader open-source environments.

These results demonstrate that external global dictionaries can introduce measurable overhead, particularly for datasets with large or complex dictionaries. However, this overhead is not inherent to the concept—rather, it stems from the additional I/O roundtrips. With even a minimal caching layer, much of the performance degradation can be alleviated. In production-grade systems that integrate block-level caching natively, the remaining overhead could likely be reduced further, making global dictionaries both a space- and performance-efficient option.

5.7 Global Dictionary Cost/Benefit Analysis

While earlier sections focused on storage efficiency and performance tradeoffs, this section evaluates the economic viability of global dictionaries by comparing their compute cost against the storage cost savings they offer. The goal is to estimate how long it takes for the savings from compression to recoup the cost of generating the dictionaries—i.e., the breakeven time.

5.7.1 Point Estimate: Best-Case Breakeven Time

We begin by computing a basic breakeven estimate using the static, one-shot dictionary approach. This assumes that the entire dataset is available up front and that a single dictionary is built before writing the data. While not always realistic, this gives an optimistic lower bound on breakeven time.

For each dataset, we calculate:

- The compute cost of generating the dictionary, based on measured job runtime, a compute price of \$0.40/hour, and a 3-node cluster.
- The monthly storage saving, by measuring the difference in output size between hybrid-dictionary-encoded and baseline (non-dictionary) tables, and applying a cloud storage price of \$0.023/GB/month.

Table 5.4: Static breakeven analysis using one-shot dictionary generation. Compute cost is a one-time cost, while savings are monthly.

Dataset	Compute Cost ($)	Storage Saved (GB)	Monthly Saving ($)	Breakeven (Months)
TPC-DS 10GB	0.0039	0.86	0.0199	0.20
Library Collection Inventory	0.0165	0.42	0.0097	1.70
NYC	0.0171	0.01	0.0003	53.57
Common Government	0.1040	0.30	0.0069	14.98

- The breakeven time in months, by dividing compute cost by monthly saving.

The compute time data is taken from the non-sketch dictionary generation job times in Table 5.1 and the storage savings data is taken from the compression statistics in Section 4.5.5.

These results show that for datasets like *TPC-DS 10GB*, hybrid dictionaries become cost-effective almost immediately. Others, such as *Common Government*, take longer, and in the case of *NYC*, the space savings are so minimal that breakeven may never be reached.

5.7.2 Limitations of Static Estimates

The calculation above is intentionally optimistic and includes several simplifications:

- It assumes the dictionary is generated once over the full dataset, which is rare outside of table initialization.
- In practice, dictionaries are built incrementally, meaning compute costs grow with each generation.
- The compression gain also depends on how frequently the dictionary is regenerated and how much the value distribution shifts between batches.
- Some tables might reuse dictionaries from others (e.g., via cloning or symbolic linking), or integrate dictionary generation into other commands like `OPTIMIZE`, which would amortize cost differently.

Given these variables, estimating real-world breakeven time requires a more dynamic analysis including features not yet implemented (see Chapter 6).

5.7.3 Incremental Breakeven Evaluation

To simulate more realistic usage, we evaluate breakeven time as data is ingested incrementally in multiple (9) batches. After each batch, a dictionary generation job is executed using only the data seen so far, simulating a periodic background job. The compute cost is accumulated across generations, and the cumulative storage saving is tracked. The compute time data is taken from the sketches in Table 5.1 and the storage savings data is taken from the compression statistics in Section 5.4.4.

Figure 5.10 shows the evolving breakeven time for each dataset as more batches are ingested and more dictionary generations are performed. The NYC dataset is omitted here, as it showed negative compression gains and therefore has infinite breakeven time.

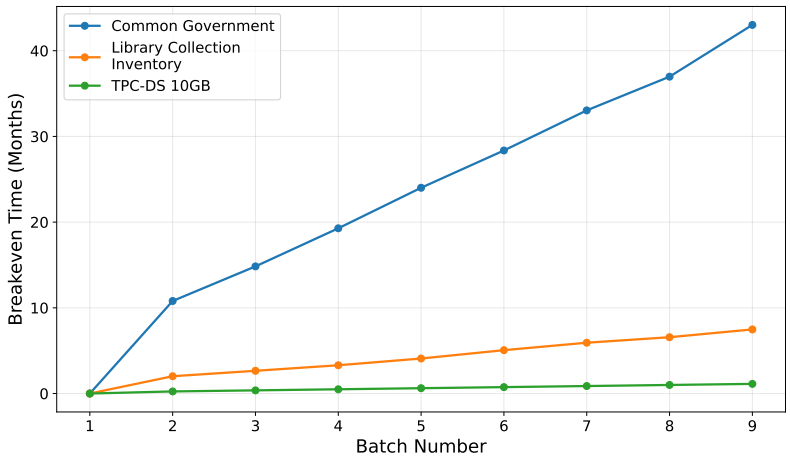

Figure 5.10: Breakeven time vs. batch number (cumulative compute cost model). NYC omitted due to negative savings.

As shown, breakeven time generally increases with more batches. This is because additional dictionary generation jobs add compute cost, while the storage savings do not increase proportionately. For example, in the *Common Government* dataset, breakeven time grows linearly with the number of batches, reaching over 40 months at 9 batches.

For highly compressible datasets like *TPC-DS 10GB*, hybrid dictionaries are cost-effective in the static and incremental models. For the others, cost-effectiveness depends heavily on how the dictionary job is run (e.g. full vs. incremental input, frequency of generation). In all cases, it's clear that recomputing dictionaries from scratch after every batch becomes inefficient quickly and should be avoided.

5.7.4 Improving Efficiency via Incremental Dictionary Updates

We also simulate a more efficient model where each dictionary generation job processes only the new data (rather than recomputing over the full dataset). This reflects a practical optimization where sketches or frequency estimations are stateful and updated incrementally as discussed in Section 5.5.6.

Figure 5.11 illustrates the breakeven times under this model. Unlike the cumulative compute cost model, the breakeven time remains much more stable, and even decreases slightly, across batches.

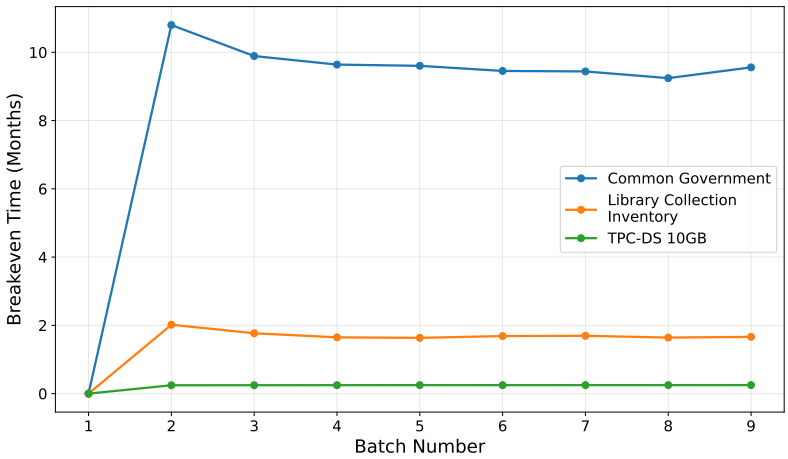

Figure 5.11: Breakeven time vs. batch number (individual compute cost per generation) NYC omitted due to negative savings.

This suggests that implementing stateful or per-batch sketching strategies may significantly reduce breakeven time and make global dictionaries more economically viable across a wider range of datasets.

6

Open Issues and Future Work

While this thesis presents a functional implementation and evaluation of global dictionary compression for static datasets in Delta Lake, several limitations remain. Furthermore, there are multiple opportunities to extend and enhance the system in ways that improve its applicability, performance, and integration within the broader data ecosystem.

This section outlines key open challenges and proposes directions for future work across four areas: support for streaming and incremental sketching, limitations related to nested data types, optimizations tied to the lifecycle of Delta tables, and potential query performance improvements through better use of dictionary information.

6.1 Delta Table Lifecycle and Cross-Table Optimizations

There remains a broader set of opportunities for improving the usability, maintainability, and performance of global dictionaries by considering them within the full lifecycle of a Delta table, or even across multiple related tables. This section outlines several directions where optimizations can occur either above the Delta layer or in coordination with it.

6.1.1 Dictionary Reuse via CREATE TABLE AS SELECT

A natural starting point is support for dictionary reuse in the context of table duplication or derivation. Many data pipelines involve creating derived tables using SQL commands like CREATE TABLE AS SELECT (CTAS), often replicating the schema and data of an existing Delta table. In such scenarios, the new table may benefit from reusing the global dictionary of its source table, especially when the data values are largely the same.

As discussed in Section 5.4, dictionaries can be directly copied or even referenced (via a symbolic link), meaning the core functionality of such an operation already exists. Future work therefore only needs to hook this functionality into existing pipelines. Such an optimization doesn't simply cut down on compute time (like the sketches from Section 5.5), but eliminates it entirely, potentially introducing storage savings with at no cost at all.

6.1.2 Integrating Dictionary Generation with Table Optimization

At present, dictionary generation is implemented as a standalone manual job. This makes it flexible but also burdensome to operate, especially in production pipelines. A promising direction is to integrate dictionary construction into existing Delta Lake commands that already involve rewriting Parquet files, most notably the OPTIMIZE command.

OPTIMIZE rewrites files to improve layout and compaction, and since this is a naturally expensive operation, it presents an ideal opportunity to share the cost of dictionary generation. The implementation could optionally scan the data for global dictionary candidates and write the encoded output in the same pass. This co-location would reduce the input and output overhead and ensure dictionary freshness in environments where OPTIMIZE is routinely used.

6.1.3 Tracking and Exposing Dictionary Effectiveness via Metadata

A major barrier to effective dictionary usage is the lack of visibility into its performance impact. However, it is abundantly clear that, although global dictionaries can be applied very successfully, it is difficult to predict on which datasets *before trying it out and collecting the statistics*. Currently, there is no persistent metadata stored that allows users or systems to quantify the value of a global dictionary on a given table or column.

To address this, Delta Lake could be extended to maintain lightweight statistics that capture:

- Compression size per row, and if this was with and without dictionary encoding
- The cardinality of dictionary-encoded columns. For example, we could apply a condition that the uniqueness percentage of all the values in a column must be below 0.4 as these were seen to generally be the only columns with positive space savings (see Section 4.4.6).
- The memory footprint of dictionary structures

Such metadata could be exposed via the Delta transaction log or through table-level statistics APIs. It would enable informed decision-making, such as:

- Determining whether a table benefits from global dictionaries
- Identifying columns where dictionary encoding is unnecessary or counterproductive
- Deciding whether to regenerate dictionaries after schema or data drift

6.1.4 Column-Level Control of Dictionary Encoding

Building on the availability of column statistics, another optimization is to enable fine-grained control over which columns are dictionary-encoded. Rather than applying a global dictionary to all eligible columns, the system could use heuristics or thresholds to determine suitability on a per-column basis.

For instance, a column with high cardinality but low repetition might offer little compression benefit from a dictionary and could instead fall back to alternative compression

methods such as bit-packing or run-length encoding. This approach allows the storage system to adaptively balance dictionary size, encoding complexity, and expected benefit.

6.1.5 Throttling Dictionary Sizes Based on Cardinality Profiles

Future implementations should also consider mechanisms for throttling or bounding global dictionary size, particularly for columns with cardinalities outside of beyond what a local dictionary could hold, but within what a hybrid dictionary could hold. This was already discussed in Sections 4.5.5 and 5.5.7.

6.1.6 Managing Dictionary File Retention

As global dictionaries are stored as external files, their lifecycle must be properly coordinated with the Delta table that references them. Currently, Delta's vacuum operation removes files that are no longer tracked by the transaction log. This introduces two related challenges.

First, Delta must be aware not to vacuum dictionary files that are still in use by one or more active Parquet files. Failure to do so could result in missing dictionary data during read operations.

Second, dictionary files that are no longer referenced by any table version or Parquet file should eventually be garbage collected to free storage space. Implementing safe and efficient cleanup requires tracking dictionary usage across snapshots and coordinating this with Delta's retention logic.

Both problems require dictionary files to be included in the Delta table's metadata model. This could involve extending the transaction log to track dictionary file versions, reference counts, or dependencies across data files. Doing so would allow Delta to distinguish between live and obsolete dictionaries and maintain them accordingly.

Overall, these enhancements point toward a more intelligent and integrated model for managing dictionary lifecycles. Global dictionaries should be treated not merely as compression artifacts but as first-class metadata objects that participate fully in the evolution and maintenance of Delta tables.

6.2 Query Optimization Opportunities

Beyond storage efficiency, global dictionaries also present compelling opportunities to optimize query execution, particularly in the domains of aggregation and caching. Since a global dictionary inherently unifies the codes data pages use across file and captures a column's unique values and their frequencies, this structured representation can be exploited by query engines for more efficient data processing. In fact, even without any significant storage improvement, global dictionaries have the potential to optimize compute immensely. This makes it **perhaps the most important future direction of investigation**.

6.2.1 Standardized Codes and Query Performance

Global dictionaries may benefit from the fact that they standardize the codes used to represent values across the entire dataset. This property enables encoded values to be directly comparable across column chunks and files, eliminating the need to decode and remap values during operations like joins, groupings, or distinct computations. Standardized codes also allow query engines to operate entirely on compact integers during execution, which can significantly reduce memory consumption and improve CPU efficiency compared to operating on longer, variable-length strings.

To illustrate the potential benefits of working with standardized codes in Spark and Delta Lake, we conducted a microbenchmark. In this benchmark, we created a large Delta table with 100 million rows containing two columns: a string column and a corresponding integer column representing the dictionary-encoded version of the same string values. The string values were generated with a fixed length of 1024 characters to simulate high-cost string comparisons. We varied the number of unique values in the column (10, 100, 1,000, and 10,000) to observe the effect of cardinality on performance.

We then executed a simple `SELECT COUNT(*) FROM table GROUP BY col` query on each column and measured the query runtimes across multiple iterations. Figure 6.1 presents the results for the string length of 1024. It can be observed that the `GROUP BY` on the integer column, which simulates a globally encoded dictionary representation, was able to outperform the string column by a substantial margin. Depending on the cardinality of the column, the speedup exceed 8x in some cases.

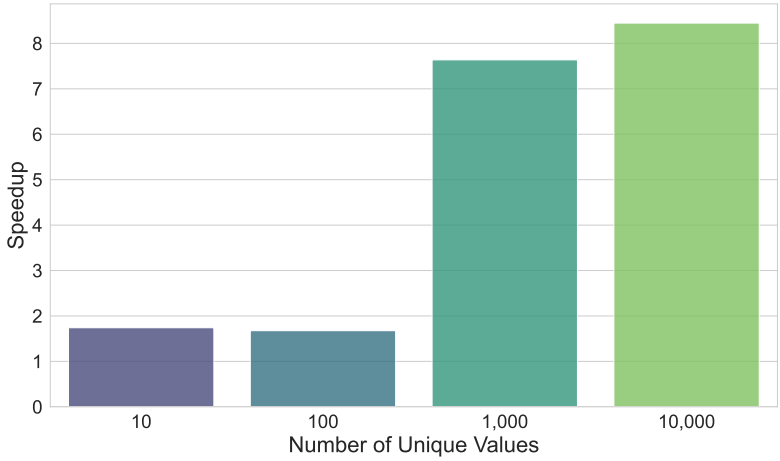

Figure 6.1: Query speedup for `GROUP BY` on integer columns against baseline string columns in a Delta table with 100M rows and string length 1024.

These results demonstrate that standardized codes from a global dictionary can materially improve query performance by replacing expensive string comparisons with more efficient integer operations. This effect becomes especially pronounced in high-cardinality

datasets or when working with very long strings, where the cost of string processing dominates execution time.

While global dictionaries enable standardized, comparable codes, this property is limited when using hybrid schemes and dictionary versioning. In hybrid approaches, values not present in the global dictionary are encoded with local, per-row-group dictionaries whose codes are not comparable beyond that row group. Similarly, with versioning, codes are only consistent within the set of files encoded with the same dictionary version. Comparing data across different versions or beyond the range of the global dictionary requires decoding and remapping, which reduces the efficiency gains. Developing mechanisms to extend comparability across versions and hybrid boundaries remains an open challenge, but have been partially addressed in other works (31).

6.2.2 Accelerating Aggregations with Precomputed Frequencies

Many analytical queries involve operations like `GROUP BY`, `COUNT`, `SUM`, and `AVG`, which require scanning column values and grouping by their identities. If the global dictionary stores not only the unique values but also frequency statistics (i.e., how often each value appears across the table), then certain aggregate queries can be partially or fully resolved using just the dictionary itself.

For example:

- A `COUNT` of each distinct category value (or similar `GROUP BY` operations) in a column can be derived directly from the dictionary's frequency map.
- A `DISTINCT` query on a column could leverage the set of unique values already maintained in the dictionary, avoiding the need to construct this set at query time.

Such dictionary-aware optimizations could dramatically reduce query latency, especially for large tables with low-cardinality dimensions.

However, these opportunities come with challenges. In evolving datasets where new values appear over time and old values become obsolete, the dictionary can become stale, bloated, or less representative of the current data distribution. This reduces its effectiveness for both compression and query acceleration. A global dictionary that does not adapt to the dataset's evolution may include many entries that no longer occur, wasting space and weakening query optimizations.

Addressing this issue will require additional mechanisms, such as dictionary maintenance policies, versioning strategies, or periodic rebuilding to ensure that the dictionary remains compact and relevant.

6.2.3 Enhancing Cache Efficiency with Block-Aware Dictionaries

Another direction for performance gains lies in improved caching. Modern storage engines, including Delta Lake's DBIO layer, employ block-level caching to minimize disk reads. If the presence of a global dictionary allows the engine to predict data access patterns or compress data into more cache-friendly blocks, this can yield substantial I/O savings.

Furthermore, by co-locating dictionary-encoded columns with their associated dictionary blocks, it becomes possible to cache only the minimal set of blocks needed to decode and process a given query. For repeated queries over the same dimension column (e.g., `region` or `status`), this can reduce memory pressure and boost responsiveness.

6.2.4 Toward Dictionary-Aware Query Planning

Realizing these benefits requires enhancements to the query planner, which must become aware of dictionary availability and contents. Planners should:

- Detect when a dictionary can answer a query partially or fully.
- Avoid scanning base data when dictionary-level aggregates suffice.
- Choose physical plans that prioritize dictionary-optimized paths when available.

This opens an exciting line of future work at the intersection of storage and execution layers, where dictionaries evolve from passive compression tools into active enablers of faster, more intelligent query processing.

By extending global dictionary functionality beyond compression into query optimization and caching, the system can achieve compound performance gains, shortening query times and reducing resource usage.

6.3 Sketch Optimizations

6.3.1 Streaming and Incremental Sketches

One current limitation of the global dictionary generation process is its batch-oriented nature. In its current form, the dictionary is created through a single-pass scan over the entire table, requiring already used parts of the table (for dictionary generation) to be read again. This becomes inefficient and impractical in scenarios where data arrives incrementally or continuously, such as in streaming pipelines or frequently updated Delta tables.

To address this, future work could explore the design and implementation of streaming-compatible sketching algorithms. In particular, the Misra-Gries sketch used in 5.5, was originally designed for streaming use cases. However, its current implementation in the Databricks environment does not allow for this, particularly due to its inability to suspend (and resume) the state of the algorithm. An exact description of how this might be tackled is described in Section 5.5.6

Additionally, incremental sketches should support thresholding logic to decide whether dictionary updates are beneficial based on observed value frequencies and cardinality drift. The system could also leverage Delta's change data feed (CDF) feature, if enabled, to more efficiently isolate new data for sketch updates.

6.3.2 Limitations with Nested Data and Schema Flattening

The current implementation of global dictionary compression is designed for flattened, tabular data structures. However, many real-world Parquet datasets contain nested types such as *struct*, *array*, and *map*, which present additional complexity.

To support global dictionaries in such cases, it is often necessary to first *flatten* the schema, transforming nested fields into separate top-level columns. While this enables compatibility with dictionary encoding mechanisms, it introduces several complications:

- **Schema explosion:** Flattening can significantly increase the number of columns, particularly in deeply nested schemas, making the data harder to manage and less readable.
- **Row misalignment:** Flattening can break the assumption of consistent row counts across columns. For instance, an array field may generate multiple entries per parent row, while other fields remain single-valued. This undermines algorithms that might depend on consistent row granularity, such as single-pass frequency sketches like Misra-Gries.
- **Complex dictionary mapping:** Associating flattened columns back to their original nested structure during decoding or analysis may require additional metadata or transformation logic.

An illustrative example of schema flattening and its effects can be found in Section 4.1. These challenges highlight the need for future work on native support for nested types within the global dictionary framework, potentially by extending sketching algorithms and dictionary encoding mechanisms to operate hierarchically or at the nested field level.

Until such support is developed, global dictionary compression remains most practical and effective for already-flattened datasets, limiting its applicability to a subset of use cases.

7

Conclusion

7.1 Conclusion

This thesis explored the feasibility and design of global dictionary compression techniques for Parquet-based Delta Lake storage, aiming to improve storage efficiency while maintaining compatibility with real-world data lake operations. The research was guided by five core questions, each of which is answered below.

RQ1: How can a global dictionary be designed and managed across distributed systems, considering file immutability and evolving datasets? Can global dictionaries coexist with local dictionaries in a complementary manner?

Global dictionaries can be effectively managed using Delta Lake's transaction log and metadata mechanisms, while the dictionaries themselves are stored in the regular data portion of Delta tables. This allows centralized storage and versioning of dictionary files, enabling consistent use across files. The hybrid design demonstrated in this thesis confirms that global and local dictionaries can coexist, with global dictionaries handling common values and local fallbacks covering rare or newly seen values.

RQ2: What are the trade-offs in write, read and storage performance when using global dictionaries compared to local dictionaries?

Global dictionaries improve storage efficiency for suitable columns, especially those with medium cardinality and frequent repetition. However, they introduce additional lookup overhead and metadata management complexity. Write and read performance may be impacted due to the need to preload dictionaries, the extent of which strongly depends on efficient caching and lookup strategies. Nevertheless, space savings of over 10 percent were occasionally observed in some datasets, validating the trade-off. However, careful monitoring of the trade-off at runtime is necessary, in case space savings are too low or overhead is too high.

RQ3: How does the system handle newly arriving data with previously unseen values, and how does this affect dictionary maintenance?

To handle newly arriving values, the system includes mechanisms for incremental dictionary updates and fallback encoding. Values not present in the current global dictionary can be encoded using a local dictionary or stored plainly, and the global dictionary can

be expanded periodically through background jobs while previously written files still use older dictionary versions. This ensures that compression benefits can be preserved even in evolving datasets without rewriting existing files.

RQ4: How can global dictionaries be integrated with Delta Lake's metadata mechanisms, such as metadata and add actions?

Integration is achieved by extending Delta Lake's metadata actions to reference external dictionary files. Each file includes a pointer to the dictionary version it uses, and the global dictionary directory is structured to allow consistent resolution by readers and writers. In addition, a global (per-table) metadata field is maintained, pointing to the most recent global dictionary version. This approach ensures that dictionary references are atomic, versioned, and resilient to concurrent updates.

RQ5: Can global dictionaries be efficiently utilized during compaction or checkpointing processes?

Yes, global dictionaries can be effectively leveraged during compaction. When small Parquet files are rewritten as part of Delta Lake's compaction process, the system can re-encode data using an up-to-date global dictionary. This allows previously scattered or inconsistently encoded values to be unified under a single dictionary version, improving both compression and query efficiency. This reduces redundancy introduced by unused older dictionary versions, local dictionaries and plain encodings in earlier files, leading to long-term space savings and more consistent encoding across the dataset. This process can be triggered periodically or adaptively based on storage metrics and column characteristics.

7.2 Final Remarks

This thesis has shown that global dictionary compression can serve as a powerful tool for reducing storage overhead in data lake systems built on Parquet and Delta Lake. By designing a practical and extensible implementation that supports both static and incremental workloads, this work bridges a gap between theoretical compression techniques and real-world system constraints.

Beyond the concrete storage savings achieved, this research contributes a reusable architecture and set of design principles that can inform future developments in data engineering infrastructure. While some challenges remain, including compatibility, query optimization, and adaptive dictionary management, the foundation laid here provides a solid basis for continued innovation.

Global dictionaries represent more than just a compression technique. They offer a pathway toward more intelligent, centralized data representation in distributed systems. As data volumes continue to grow, such approaches will be critical to sustaining the performance, scalability, and efficiency of modern data platforms.

References

[1] E. F. Codd. **A relational model of data for large shared data banks**. *Commun. ACM*, **13**(6):377–387, June 1970. 4

[2] Surajit Chaudhuri and Umeshwar Dayal. **An overview of data warehousing and OLAP technology**. *SIGMOD Rec.*, **26**(1):65–74, March 1997. 4

[3] Daniel Abadi, Samuel Madden, and Miguel Ferreira. **Integrating compression and execution in column-oriented database systems**. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, page 671–682, New York, NY, USA, 2006. Association for Computing Machinery. 5, 8

[4] Peter A Boncz, Marcin Zukowski, and Niels Nes. **MonetDB/X100: Hyper-Pipelining Query Execution.** In *Cidr*, **5**, pages 225–237, 2005. 5

[5] Midhul Vuppalapati, Dan Truong, Justin Miron, Ashish Motivala, Rachit Agarwal, and Thierry Cruanes. **Building An Elastic Query Engine on Disaggregated Storage**. 5

[6] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J. Green, Monish Gupta, Sebastian Hillig, Eric Hotinger, Yan Leshinksy, Jintian Liang, Michael McCreedy, Fabian Nagel, Ippokratis Pandis, Panos Parchas, Rahul Pathak, Orestis Polychroniou, Foyzur Rahman, Gaurav Saxena, Gokul Soundararajan, Sriram Subramanian, and Doug Terry. **Amazon Redshift Re-invented**. In *Proceedings of the 2022 International Conference on Management of Data*, pages 2205–2217, Philadelphia PA USA, June 2022. ACM. 5

[7] Thomas Neumann. **Efficiently compiling efficient query plans for modern hardware**. *Proc. VLDB Endow.*, **4**(9):539–550, June 2011. 5

[8] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. **Spark SQL: Relational Data Processing in Spark**. In *Proceedings of the 2015 ACM SIGMOD International Conference on*

Management of Data, SIGMOD '15, page 1383–1394, New York, NY, USA, 2015. Association for Computing Machinery. 5

[9] Chunwei Liu, Anna Pavlenko, Matteo Interlandi, and Brandon Haynes. **A Deep Dive into Common Open Formats for Analytical DBMSs**. *Proceedings of the VLDB Endowment*, **16**(11):3044–3056, July 2023. 6

[10] Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998. 6

[11] Azim Afroozeh and Peter A. Boncz. **The FastLanes Compression Layout: Decoding >100 Billion Integers per Second with Scalar Code**. *Proc. VLDB Endowment*, **16**(9):2132–2144, 2023. 6, 14, 15

[12] Maximilian Kuschewski, David Sauerwein, Adnan Alhomssi, and Viktor Leis. **BtrBlocks: Efficient Columnar Compression for Data Lakes**. *Proceedings of the ACM on Management of Data*, **1**(2):1–26, June 2023. 6

[13] Peter Boncz, Thomas Neumann, and Viktor Leis. **FSST: Fast Random Access String Compression**. *Proceedings of the VLDB Endowment*, **13**(12):2649–2661, August 2020. 6, 13, 16

[14] Michael Armbrust, Tathagata Das, Liwen Sun, Burak Yavuz, Shixiong Zhu, Mukul Murthy, Joseph Torres, Herman Van Hovell, Adrian Ionescu, Alicja Łuszczak, Michał Świtakowski, Michał Szafrański, Xiao Li, Takuya Ueshin, Mostafa Mokhtar, Peter Boncz, Ali Ghodsi, Sameer Paranjpye, Pieter Senster, Reynold Xin, and Matei Zaharia. **Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores**. *Proceedings of the VLDB Endowment*, **13**(12):3411–3424, August 2020. 10, 14, 56

[15] Ingo Müller, Cornelius Ratsch, and Franz Färber. **Adaptive String Dictionary Compression in In-Memory Column-Store Database Systems**, 2014. 13, 16

[16] Christoph Doblander. **Compression in Publish/Subscribe Systems**. 13, 16

[17] Jiancong Tong, Anthony Wirth, and Justin Zobel. **Principled Dictionary Pruning for Low-Memory Corpus Compression**. In *Proceedings of the 37th International ACM SIGIR Conference on Research & Development in Information Retrieval*, pages 283–292, Gold Coast Queensland Australia, July 2014. ACM. 13, 16

[18] Chunwei Liu, McKade Umbenhower, Hao Jiang, Pranav Subramaniam, Jihong Ma, and Aaron J. Elmore. **Mostly Order Preserving Dictionaries**. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1214–1225, Macao, Macao, April 2019. IEEE. 13, 16, 17, 54

[19] Carsten Binnig, Stefan Hildenbrand, and Franz Färber. **Dictionary-Based Order-Preserving String Compression for Main Memory Column Stores**. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, pages 283–296, Providence Rhode Island USA, June 2009. ACM. 13, 16

[20] Gennady Antoshenkov. **Dictionary-Based Order-Preserving String Compression**. *The VLDB Journal The International Journal on Very Large Data Bases*, **6**(1):26–39, February 1997. 13, 16

[21] Miguel A. Martínez-Prieto, Javier D. Fernández, and Rodrigo Cánovas. **Compression of RDF Dictionaries**. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pages 340–347, Trento Italy, March 2012. ACM. 13, 16

[22] Yannis Foufoulas, Lefteris Sidirourgos, Eleftherios Stamatogiannakis, and Yannis Ioannidis. **Adaptive Compression for Fast Scans on String Columns**. In *Proceedings of the 2021 International Conference on Management of Data*, pages 554–562, Virtual Event China, June 2021. ACM. 13, 17

[23] Christian Lemke, Kai-Uwe Sattler, Franz Faerber, and Alexander Zeier. **Speeding Up Queries in Column Stores**. In Torben Bach Pedersen, Mukesh K. Mohania, and A Min Tjoa, editors, *Data Warehousing and Knowledge Discovery*, **6263**, pages 117–129. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. 13, 17

[24] Paras Jain, Peter Kraft, Conor Power, Tathagata Das, Ion Stoica, and Matei Zaharia. **Analyzing and Comparing Lakehouse Storage Systems**. In *Proc. 13th Conf. on Innovative Data Systems Research (CIDR)*, 2023. 14

[25] Apache Hudi Project. **Apache Hudi Metadata Table Documentation**. `https://cwiki.apache.org/confluence/display/HUDI/Metadata+Table`, 2022. 14

[26] Ted Gooch. **Why and How Netflix Created and Migrated to a New Table Format: Iceberg**. `https://www.dremio.com/subsurface/why-and-how-netflix-created-and-migrated-to-a-new-table-format-iceberg/`, 2020. 14

[27] Lindsay Clark. **Industry reacts to DuckDB's radical rethink of Lakehouse architecture**. `https://www.theregister.com/2025/06/05/ducklake_db_industry_reacts/`, 2025. 14

[28] Maximilian Kuschewski, David Sauerwein, Adnan Alhomssi, and Viktor Leis. **BtrBlocks: Efficient Columnar Compression for Data Lakes**. *Proc. ACM Manage. Data*, **1**(2):118:1–118:26, 2023. 15

[29] Spiral Data. **Vortex: High-Performance Columnar Format (Specification v0.36)**. `https://docs.vortex.dev/specs/file-format`, 2023. 15

[30] Weston Pace, Chang She, Lei Xu, Will Jones, Albert Lockett, Jun Wang, and Raunak Shah. **Lance: Efficient Random Access in Columnar Storage through Adaptive Structural Encodings**. *CoRR*, **abs/2504.15247**, 2025. arXiv preprint. 15, 16

[31] Tim Gubner, Viktor Leis, and Peter Boncz. **Optimistically compressed Hash Tables Strings in the USSR**. *ACM SIGMOD Record*, **50**(1):60–67, May 2021. 17, 89

[32] The Apache Software Foundation. **Apache CarbonData**. `https://carbondata.apache.org/`, 2021. Accessed: 2025-07-06. 17

[33] Majid Saeedan and Ahmed Eldawy. **Spatial Parquet: A Column File Format for Geospatial Data Lakes**. In *Proceedings of the 30th International Conference on Advances in Geographic Information Systems*, pages 1–4, Seattle Washington, November 2022. ACM. 18

[34] Kil Joong Kim, Bohyoung Kim, Seung Wook Choi, Young Hoon Kim, Seokyung Hahn, Tae Jung Kim, Soon Joo Cha, Vasundhara Bajpai, and Kyoung Ho Lee. **Definition of Compression Ratio: Difference Between Two Commercial JPEG2000 Program Libraries**. *Telemedicine and e-Health*, **14**(4):350–354, 2008. PMID: 18570564. 19

[35] Alexander Van Renen, Dominik Horn, Pascal Pfeil, Kapil Vaidya, Wenjian Dong, Murali Narayanaswamy, Zhengchun Liu, Gaurav Saxena, Andreas Kipf, and Tim Kraska. **Why TPC Is Not Enough: An Analysis of the Amazon Redshift Fleet**. *Proceedings of the VLDB Endowment*, **17**(11):3694–3706, July 2024. 28

[36] E.A. Unger, L. Harn, and V. Kumar. **Entropy as a measure of database information**. In *[1990] Proceedings of the Sixth Annual Computer Security Applications Conference*, pages 80–87, 1990. 31

[37] Samy Chambi, Daniel Lemire, Owen Kaser, and Robert Godin. **Better bitmap performance with Roaring bitmaps**. *Software: Practice and Experience*, **46**(5):709–719, 2016. 49

[38] J. Misra and David Gries. **Finding Repeated Elements**. *Science of Computer Programming*, **2**(2):143–152, 1982. 73

[39] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. **HyperLogLog: The Analysis of a Near-Optimal Cardinality Estimation Algorithm**. In *Proceedings of the 2007 Conference on Analysis of Algorithms (AOFA)*, pages 137–156, 2007. 79